

SERIE DESARROLLO

## NOVEDADES DE VISUAL BASIC 9.0

**GUILLERMO "GUILLE" SOM**



**SolidQ**

## ADVERTENCIA LEGAL

Todos los derechos de esta obra están reservados a Guillermo Som y SolidQ™ 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

© SolidQ™ Press, 2008

## **Novedades de Visual Basic 9.0**

*Serie Desarrollo*

Autor: Guillermo “Guille” Som

Editado por SolidQ™ Press

Apartado de correos 202

03340 Albatera, Alicante, España

<http://www.solidq.com>

ISBN: **978-84-936417-0-2**



**Microsoft® Partner**

Gold Data Platform  
Gold Business Intelligence  
Silver Portals and Collaboration  
Silver Web Development  
Silver Communications  
Silver Search

Da un giro a tu carrera profesional.  
*Es tiempo de oportunidades.*

MASTERS CERTIFICADOS  
POR SolidQ

## Máster en BI



<http://www.solidq.com/es/MasterBI>

## Máster SQL Server DBA



<http://www.solidq.com/es/MasterSQLServerDBA>

## Máster en SharePoint



<http://www.solidq.com/es/MasterSharePoint>

**¡Infórmate Ya!**

Conviértete en un profesional altamente  
especializado en tecnologías Microsoft.

Bonificable a través de la Fundación Tripartita



Fundación Tripartita  
PARA LA FORMACIÓN EN EL EMPLEO

Para más información llama al 800.300.800 o +34 91 414 8950 o bien manda un e-mail a: [ventasib@solidq.com](mailto:ventasib@solidq.com)

# Prólogo

---

## “El Guille”: una leyenda viva entre nosotros

Mis comienzos en la informática nada tienen que ver con las herramientas que utilizamos hoy en día, ni con las potentes plataformas en las que nos movemos. Eran otros tiempos y nos enfrentábamos a otros retos. Es más, mis estudios nada presagiaban que terminaría dedicado a las tecnologías de la información.

Defraudado por el inminente fracaso de promesas que otras plataformas aún ofrecían, hace casi 20 años decidí pasarme a utilizar tecnologías de Microsoft. Para que el salto de tecnología no fuera un simple detalle en mi vida, lo acompañé de un salto más importante: acepté un puesto en un proyecto de micro-centrales hidroeléctricas en Indonesia (tranquilos, yo tampoco tenía muy claro dónde quedaba Indonesia cuando me hablaron por primera vez de este proyecto), financiado por la Comisión de las Comunidades Europeas (entonces se llamaba así).

En ese proyecto estaba previsto crear aplicaciones, que funcionaran en MS-DOS y Novell 3.1 y que se basaran en pantallas convencionales de 25 filas por 80 caracteres por fila. Sin embargo, algo dentro de mí rechazaba esa perspectiva con repugnancia. Algunos años antes había dirigido la producción de la versión en castellano de True Basic para Commodore Amiga y Atari ST, y no podía resignarme a relegar mis aplicaciones a unas sosas pantallas verdes carentes de vida.

Corría el año 1991, y decidí que el proyecto utilizara Windows 3.11 for Workgroups, que acababa de salir al mercado, con lo que el sistema operativo empezaba a parecerse más a mi querido GEM de Digital Research, que había sido mi vida durante los cuatro años anteriores.

Probé diferentes herramientas de programación en este entorno, todas tan lejos del magnífico True Basic, o del simple pero eficiente GfA Basic. Ninguna llegó a entusiasmarme por completo, hasta que un día leí un artículo en la revista oficial de MS Basic de Fawcette Publications (BasicPro Magazine, que luego se convertiría en el Visual Basic Programmers Journal) sobre un nuevo lenguaje de programación llamado Visual Basic, basado en programación orientada a eventos, y que tenía una pinta muy similar a los entornos que yo estaba acostumbrado a utilizar.

Me las ingeníé para comprarlo *online* en una tienda de Estados Unidos (Compuserve ya existía por aquellos tiempos), esperando que el paquete lograra llegar a Yakarta. Y llegó, ¡vaya que si llegó! Aquél proyecto se desarrolló en Visual Basic 1.0, y luego pasé por el Professional Toolkit, en cuanto volví a España tras un año en Indonesia. Y desde entonces trabajé con todas y cada una de las versiones de este lenguaje que han salido al mercado.

Durante los años siguientes, mi trabajo me llevó a vivir fuera de España, ajeno al mercado informático español, mientras que un programador de Visual Basic, afincado en Nerja (Málaga), se iba convirtiendo poco a poco en una leyenda.

Nuestros caminos se cruzaron por primera vez en el 2001, cuando Karen Watterson (editora de Apress por aquel entonces) necesitó traducir al inglés un artículo del Guille para el décimo aniversario de Visual Basic, y su traductor habitual se había rajado a última hora. Intenté traducir este artículo conservando lo máximo posible el estilo inconfundible del Guille, pero no llegué a conocerle en persona aún. Por aquél entonces, yo ya conocía de sobra el trabajo que el Guille estaba haciendo, y el prestigio que tenía en todo el mundo hispano parlante.

Un par de años más tarde, pude por fin conocer en persona al Guille, en una cena ofrecida por Microsoft a los MVP de España. A mí no me conocía nadie de esa cena (yo creo que ni de nombre). Claro que todos conocían al Guille de nombre, pero al parecer nunca le habían visto en persona (cosas de la comunicación electrónica), hasta que le saludé efusivamente y entonces se dieron cuenta todos de quién tenían allí. Recuerdo mi nerviosismo al poder hablar con El Guille por fin, y lo extraño que me pareció al principio ese gracioso acento andaluz pronunciado por una persona de apariencia tan seria y respetable.

Por aquél entonces, acababa de fundar Solid Quality Learning (a la que ahora llamamos **SolidQ**) y empezó la maravillosa marea que nos ha envuelto desde entonces.

Los años pasaron, nuestros caminos fueron evolucionando en trayectorias distintas, pero por suerte ambas ascendentes, y seguíamos viéndonos en conferencias, reuniones y fiestas.

Ineta nos invitó al Guille y a mí a impartir juntos una serie de charlas en México (cinco ciudades en una semana). Esto me dio la oportunidad de compartir bastante tiempo con él. Este viaje me mostró de un modo muy claro la importancia que El Guille tiene para cientos de miles (sino millones) de programadores en ambos lados del Atlántico.

Yo he sido testigo de una larga cola de programadores esperando a que El Guille les firmara un autógrafo en alguno de sus libros, o a que aceptara tomarse una foto con ellos. He estado en muchas conferencias en muchos países del mundo, y nunca he visto a nadie tener este nivel de adoración.

Aún recuerdo los ojos de algunas de esas personas, agradeciéndole que se hubiera tomado la molestia de viajar tan lejos de su casa para ir a compartir con ellos sus conocimientos.

Decidimos que era buena idea enlazar nuestros caminos, y El Guille pasó a ser un SolidQ.

Este libro que estás leyendo es el primero de una serie de libros electrónicos que estamos editando. Para el primer título, no podía pensar en mejor candidato que Guillermo Som, nuestro querido Guille. Este libro es una continuación natural del trabajo que lleva haciendo durante tantos años en su Web. Estamos seguros de que se convertirá en un clásico instantáneamente.

Espero que lo disfrutéis tanto como yo.

Fernando G. Guerrero  
Managing Partner, CEO  
SolidQ

*Marzo de 2008, desde el aeropuerto de Gatwick, esperando volar de vuelta a casa.*

# Novedades de Visual Basic 9.0

---

## Introducción

Este libro, dedicado exclusivamente a las nuevas características de Visual Basic 9.0 (también conocido comercialmente como Visual Basic 2008), surge a raíz de otro libro que estoy preparando desde mediados del año pasado: “Las recetas del Guille para Visual Basic”.

El libro de las recetas lo dejé en modo “pausa” a la espera de la salida de Visual Studio 2008 en español, prevista para el mes de marzo de 2008. Como en las recetas también iba a incluir cosas relacionadas con la nueva versión de Visual Basic, decidí crear unos apéndices explicando las novedades de Visual Basic 9.0 de forma independiente. Pero en una de las charlas virtuales con otros compañeros de **SolidQ**, finalmente decidimos extraer esas novedades, ampliar el contenido y publicarlas de forma independiente, pero en lugar de hacerlo como un libro “de papel” (que es el formato pensado para el libro de las recetas), decidimos hacerlo en formato electrónico. Por dos razones: la primera para facilitar la distribución, ya que al ser en formato electrónico, cualquiera con acceso a Internet lo podría conseguir con un par de clics de ratón; la segunda razón es el precio, ya que un libro electrónico es más económico que uno impreso.

## ¿A quién va dirigido este libro?

En este libro explico, con todo lujo de detalles, todas las nuevas características de Visual Basic 9.0 (ó 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 Visual Basic para .NET, concretamente Visual Basic 2005 (o VB 8.0). En particular el conocimiento de las peculiaridades de Visual Basic, 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.

Si el lector quiere profundizar sobre Visual Basic 2005, le recomiendo que “se regale” mi libro “Manual Imprescindible de Visual Basic 2005”, ya que en ese libro explico todas las novedades de Visual Basic y .NET Framework 2.0, como son los *generics*, la sobrecarga de operadores, y otros conceptos sobre la programación orientada a objetos, etc., pero explicados incluso para los no iniciados. En mi sitio Web puede solicitarlo para que se lo envíe (previo pago) a cualquier parte del mundo. Si ese es su interés, puede usar este enlace: <http://www.elguille.info/NET/MIVB2005/comprar.aspx>.

En cualquier caso, y como tampoco quiero que parezca que todo lo explicado en este libro está hecho de forma que obligue a comprar materiales extras, en el sitio Web con el material de apoyo (ver

más abajo) encontrará una dirección de correo a la que podrá escribir sugiriendo los temas que le gustaría que aclarara con más detalle, tanto para “no iniciados”, como para desarrolladores que no han tenido la oportunidad de utilizar Visual Basic con anterioridad, y por tanto, no tienen claros los conceptos que los que llevamos más tiempo utilizándolo sí tenemos. Para mí será un placer poder escribir artículos (o notas) que aclaren todos los conceptos que yo he obviado en este libro (por razones de espacio y por dar un sentido más avanzado al contenido, ya que casi siempre que se habla de Visual Basic parece que estamos escribiendo para “no iniciados”, y no siempre es ese el caso), puesto que mi intención, así como la de mis compañeros de SolidQ, siempre es que las cuestiones relacionadas con la programación queden lo más claras posible, y tal como dice nuestro amigo **Fernando Guerrero**: *Comparte lo que sabes, aprende lo que no sepas*.

## 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=1>. Desde ahí podrá bajar en formato ZIP los diferentes proyectos. La mayoría, al estar escritos solo en Visual Basic, los podrá usar tanto con la versión comercial de Visual Studio 2008 como con la versión gratuita (Visual Basic 2008 Express Edition); solo para las soluciones que utilizan proyectos en C# y Visual Basic, necesitará utilizar Visual Studio 2008.

Y a pesar de las muchas comprobaciones que hemos hecho al contenido de este libro, es posible que surja alguna errata, en esa misma dirección indicaremos las que encontremos e incluso el propio lector podrá indicarnos las que él encuentre. Porque por más cariño y cuidado que hemos puesto en la confección de este libro, somos humanos y ya se sabe que errar es de humanos. Confíemos que no haya ninguna.

## Agradecimientos

Quiero agradecer a todos los mentores de **SolidQ** 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 quiero agradecer a **Marino Posadas** y **José Quinto** por las revisiones que han hecho de los capítulos del libro, capítulos que finalmente Daniel ha 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 mí (y en Visual Basic), y por saber esperarme casi todos los meses para que le entregue casi a última hora los artículos, particularmente mientras escribía este libro, aunque dice que ya no trabajará más los fines de semana, je, je, ¡ya lo veremos!

Tampoco quiero olvidarme del “gefe”, **Fernando Guerrero**, que ha sabido motivarme y alentarme para que este libro sea una realidad y estuviera listo en un tiempo récord, además de haber podido sacar un poco de tiempo de sus muchas tareas para escribir el prólogo del libro.

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”. Lo mismo que mis dos “monstruitos” **David** y **Guille**, que tienen a bien bajar el volumen de la tele y de la música para que no me distraiga más de la cuenta; aunque al Guille algunas veces le cuesta aguantarse, sobre todo cuando juega el Barça, y marca un gol.

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 Visual Basic 9.0 trae de nuevo.

Nos vemos  
Guillermo  
Nerja, 25 de febrero de 2008



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

## Contenido

|   |    |
|---|----|
| Prólogo .....   | 3  |
| “El Guille”: una leyenda viva entre nosotros .....  | 3  |
| Novedades de Visual Basic 9.0.....  | 5  |
| Introducción .....  | 5  |
| ¿A quién va dirigido este libro? .....  | 5  |
| Material de apoyo .....   | 6  |
| Agradecimientos .....   | 6  |
| Parte 1 El entorno de desarrollo.....   | 15 |
| Capítulo 1 El entorno de Visual Basic 2008 .....  | 17 |
| Introducción .....  | 17 |
| Nombres y versiones .....   | 17 |
| ¿Visual Basic 2008 o Visual Basic 9.0?.....   | 17 |
| Las diferentes versiones de .NET.....   | 18 |
| ¿Cuáles son las diferencias entre estas versiones de .NET? .....  | 19 |
| Las diferentes versiones de Visual Basic .....  | 20 |
| El IDE de Visual Basic 2008 .....   | 22 |
| Mejoras en IntelliSense .....   | 22 |
| Visual Basic 2008 y las líneas continuadas .....  | 24 |
| Tipos de proyectos y versión de .NET Framework.....   | 26 |
| Cambiar la versión de .NET una vez creado el proyecto .....   | 27 |
| ¿Se pueden usar aplicaciones creadas con versiones de .NET superiores a la instalada en el equipo del usuario?..... | 28 |
| Opciones de compilación de Visual Basic .....   | 28 |
| Nuevo diseño del cuadro de diálogo de agregar elementos a un proyecto.....  | 29 |
| El IDE de Visual Studio 2008 actualizado para el UAC de Windows Vista .....   | 31 |
| Configuración del entorno para Visual Basic.....  | 31 |
| Parte 2 Características generales del lenguaje .....  | 35 |
| Capítulo 2 Características generales del lenguaje (I).....  | 37 |
| Introducción .....  | 37 |
| Inferencia de tipos en variables locales .....  | 37 |
| La inferencia de tipos y Option Strict .....  | 38 |
| La inferencia de tipos en proyectos convertidos desde versiones anteriores .....                                    | 39 |
| ¿Cuándo y dónde podemos usar la inferencia de tipos?.....   | 40 |
| Inicialización de objetos .....   | 42 |

|  |    |
|--|----|
| Inicialización de objetos con clases que definan solo constructores con parámetros .....   | 43 |
| Inicialización de objetos e inferencia de tipos.....                                       | 44 |
| Inicializaciones de arrays y colecciones.....  | 44 |
| Métodos parciales .....  | 46 |
| Condiciones para los métodos parciales.....  | 48 |
| Tipos anulables .....  | 50 |
| ¿Cómo declarar variables de tipos anulables?.....  | 50 |
| Recordando un poco a los tipos anulables.....  | 51 |
| Ensamblados amigos.....  | 53 |
| Requisitos para acceder a los ensamblados amigos .....                                     | 54 |
| Requisitos para ensamblados firmados con nombre seguro .....                               | 54 |
| Requisitos para los ensamblados que no están firmados con nombre seguro.....               | 56 |
| Ejemplo de uso de los ensamblados amigos .....   | 56 |
| ¿Cómo generar la clave pública usada en el atributo InternalsVisibleTo? .....              | 57 |
| Agilidad del runtime .....   | 59 |
| Compilar código de Visual Basic sin usar el runtime de Visual Basic .....                  | 60 |
| ¿Es posible crear una aplicación de Visual Basic sin usar el runtime de Visual Basic?..... | 60 |
| Crear nuestro propio runtime de Visual Basic .....   | 61 |
| Habilitar la comprobación de errores.....  | 64 |
| Capítulo 3 Características generales del lenguaje (II) .....                               | 65 |
| Introducción .....   | 65 |
| Relajación de delegados.....   | 65 |
| Contravarianza: usar parámetros diferentes a los definidos en los delegados .....          | 65 |
| ¿La relajación de delegados solo para los eventos?.....                                    | 67 |
| Varianza: usar valores devueltos diferentes a los definidos en los delegados .....         | 67 |
| Los delegados y las conversiones implícitas (con Option Strict On).....                    | 69 |
| Los delegados y las conversiones con Option Strict Off.....                                | 71 |
| Otros usos de la relajación de delegados con las expresiones lambda .....                  | 72 |
| Respuesta a la pregunta hecha después del listado 3.17 .....                               | 73 |
| Operador ternario .....  | 73 |
| El operador If evalúa solo la parte que debe evaluar .....                                 | 76 |
| Soluciones explicadas de los ejercicios .....  | 78 |
| ¿Cuándo y dónde podemos usar el operador If? .....   | 79 |
| Solución al ejercicio.....   | 80 |

|  |     |
|--|-----|
| El operador "binario" If .....   | 81  |
| Aclaraciones sobre el por qué de algunas de las "inferencias" del listado 3.28 ..... | 82  |
| Capítulo 4 Características generales del lenguaje (III) .....                        | 85  |
| Introducción .....   | 85  |
| Tipos anónimos .....   | 85  |
| Definir un tipo anónimo .....  | 86  |
| ¿Cómo de anónimos son los tipos anónimos? .....                                      | 87  |
| Definir un tipo anónimo con propiedades de solo lectura .....                        | 89  |
| Los tipos anónimos y la inferencia automática de tipos .....                         | 92  |
| Los tipos anónimos solo los podemos usar a nivel local .....                         | 93  |
| Tipos anónimos que contienen otros tipos anónimos .....                              | 94  |
| Recomendaciones .....  | 94  |
| Solución al ejercicio del listado 4.16 .....   | 95  |
| Expresiones lambda .....   | 95  |
| Entendiendo a los delegados .....  | 96  |
| Definir una expresión lambda .....   | 97  |
| Las expresiones lambda son funciones en línea .....                                  | 99  |
| Las expresiones lambda como parámetro de un método .....                             | 99  |
| Ámbito en las expresiones lambda .....   | 102 |
| Utilizar una expresión lambda como un método de evento .....                         | 104 |
| Ampliando la funcionalidad de las expresiones lambda .....                           | 104 |
| ¿Cómo clasificar una colección de tipos anónimos? .....                              | 105 |
| Consideraciones para las expresiones lambda .....                                    | 109 |
| Capítulo 5 Características generales del lenguaje (IV) .....                         | 111 |
| Introducción .....   | 111 |
| Métodos extensores .....   | 111 |
| Los métodos extensores e IntelliSense .....  | 114 |
| Desmitificando los métodos extensores .....  | 114 |
| El ámbito de los métodos extensores .....  | 117 |
| Precedencia en el ámbito de los métodos extensores .....                             | 118 |
| Definir los módulos con métodos extensores en su propio espacio de nombres .....     | 119 |
| Ejemplo de conflicto de nombres de métodos extensores .....                          | 120 |
| Conflictos con métodos existentes .....  | 123 |
| Sobrecargas en los métodos extensores .....  | 124 |

|  |     |
|--|-----|
| Sobrecargas y parámetros opcionales en los métodos extensores .....                | 124 |
| Métodos extensores que entran en conflicto con instrucciones de Visual Basic ..... | 126 |
| ¿Qué tipos de datos podemos extender?.....   | 127 |
| Reflexionando sobre los métodos extensores.....                                    | 130 |
| Parte 3 Visual Basic y LINQ .....  | 131 |
| Capítulo 6 Visual Basic y LINQ (I).....  | 133 |
| Visual Basic y LINQ.....   | 133 |
| Un ejemplo de LINQ para ir abriendo boca .....                                     | 133 |
| Los diferentes sabores de LINQ .....   | 137 |
| LINQ to Objects.....   | 137 |
| Elementos básicos de una consulta LINQ.....  | 138 |
| Ejecución aplazada y ejecución inmediata .....                                     | 139 |
| La cláusula Select .....   | 141 |
| Ordenar los elementos de la consulta.....  | 142 |
| No es oro todo lo que reluce .....   | 142 |
| Capítulo 7 Visual Basic y LINQ (II).....   | 145 |
| Instrucciones de Visual Basic para las consultas de LINQ.....                      | 145 |
| Select .....   | 145 |
| From y Join.....   | 146 |
| Aggregate .....  | 148 |
| Las funciones Average, Count, LongCount, Max, Min y Sum .....                      | 149 |
| Las funciones All y Any .....  | 151 |
| Filtrar las consultas con Where .....  | 153 |
| Distinct .....   | 155 |
| Indicar la ordenación de los elementos de una consulta .....                       | 157 |
| Agrupaciones y combinaciones: Group By y Group Join.....                           | 158 |
| Group By.....  | 159 |
| Group Join.....  | 161 |
| Divide y vencerás: Skip y Take.....  | 163 |
| Vuelve un clásico de los tiempos de BASIC: Let .....                               | 165 |
| Respuesta al ejercicio de la sección dedicada a Where.....                         | 167 |
| Funciones agregadas personalizadas .....   | 169 |
| Capítulo 8 Visual Basic y LINQ (III) .....   | 171 |
| Visual Basic y LINQ to XML.....  | 171 |

|  |     |
|--|-----|
| Integración total de XML en Visual Basic 9.0 .....                     | 171 |
| Continuador de líneas y nombres XML .....                              | 173 |
| Elementos XML .....  | 173 |
| Diferencia entre documentos, elementos y atributos XML.....            | 174 |
| Expresiones incrustadas en los literales XML .....                     | 176 |
| Acceder a los elementos y atributos de las variables de tipo XML ..... | 176 |
| Habilitar IntelliSense en el código XML integrado .....                | 177 |
| IntelliSense siempre nos facilita la escritura de código XML.....      | 177 |
| Consultas LINQ en elementos XML.....                                   | 178 |
| Consultas LINQ para generar elementos XML.....                         | 179 |
| Consultas LINQ para realizar transformaciones .....                    | 180 |
| Importar espacios de nombres XML .....                                 | 181 |
| Dónde podemos usar los literales XML .....                             | 183 |
| Otros literales XML de trato especial.....                             | 183 |
| Capítulo 9 Visual Basic y LINQ (IV) .....                              | 185 |
| Visual Basic y LINQ to ADO.NET .....                                   | 185 |
| LINQ to DataSet .....  | 185 |
| Añadir un DataSet al proyecto .....                                    | 187 |
| Crear un adaptador y llenar una tabla.....                             | 187 |
| Consultar los datos de una tabla.....                                  | 188 |
| Consultas LINQ en DataSet no tipado.....                               | 188 |
| Añadir más elementos al DataSet.....                                   | 191 |
| LINQ to SQL .....  | 193 |
| Modelo de objetos de LINQ to SQL .....                                 | 194 |
| Crear el código automáticamente .....                                  | 197 |
| Crear las clases usando el diseñador relacional.....                   | 198 |
| Índice alfabético .....  | 203 |

(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 la nueva versión de Visual Basic.



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

# Capítulo 1

## El entorno de Visual Basic 2008

---

### Introducción

En este primer capítulo del libro sobre las novedades de Visual Basic 9.0 (o Visual Basic 2008, que es como se conoce comercialmente) 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 Visual Basic, y acabaremos conociendo algunos de los cambios realizados en el entorno de desarrollo (IDE) recomendado para sacarle el máximo provecho a esta nueva versión de uno de los lenguajes con más “solera” entre los compiladores de Microsoft.

#### *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 Basic.*

### Nombres y versiones

#### ¿Visual Basic 2008 o Visual Basic 9.0?

Visual Basic 2008 es el nombre comercial de la última versión de Visual Basic 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 Basic) 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 Visual Basic, tendremos que elegir la versión que se conoce como **Visual Basic 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 comple-

to; más adelante, veremos algunas de las limitaciones de esta versión gratuita, pero antes sigamos hablando de los diferentes nombres de Visual Basic.

El nombre o la numeración 2008 es solo una forma comercial de llamar a este lenguaje, ya que internamente se mantiene la numeración iniciada con la primera versión del lenguaje cuando hizo su aparición en el año 1991. Por tanto, el nombre interno del lenguaje es Visual Basic 9.0 que es la versión del compilador que se distribuye con .NET Framework 3.5.

Como seguramente ya sabrá el lector, el compilador de Visual Basic 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 Visual Basic 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 Visual Basic 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 Visual Basic.*

## **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 Visual Basic 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 Basic 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*.

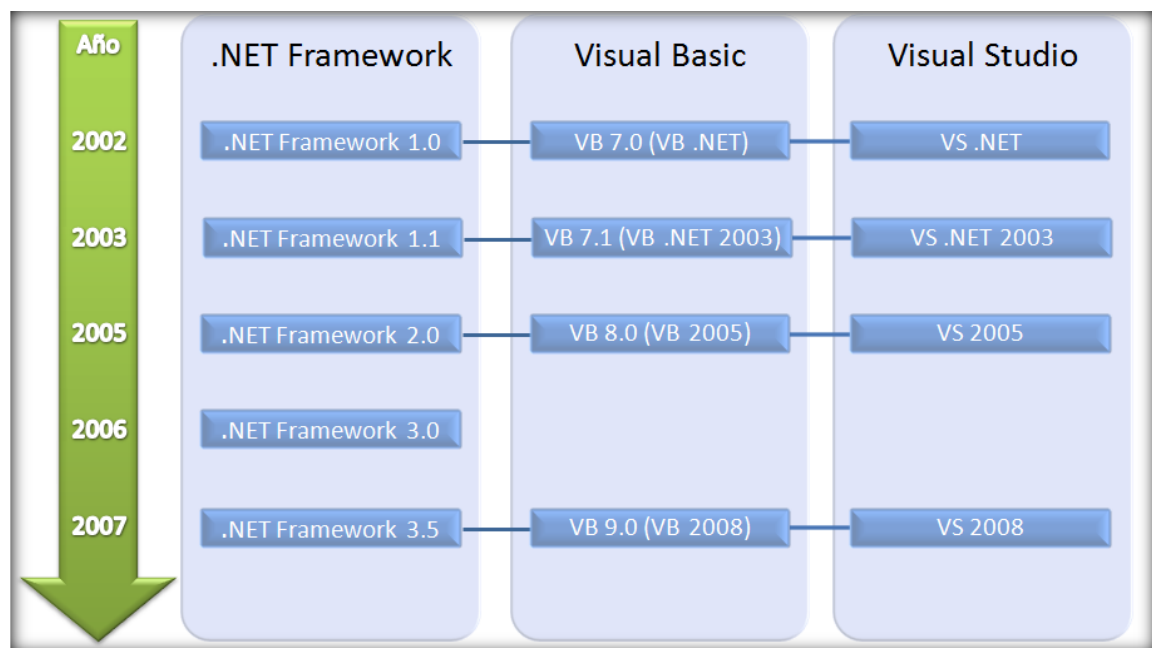


Figura 1.1. Las diferentes versiones de .NET Framework y su relación con Visual Studio

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 Basic 2008, lo que significa es que la funcionalidad que tenemos al crear las aplicaciones programadas en Visual Basic, 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 Visual Basic

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

La primera versión de Visual Basic que permitía crear aplicaciones para .NET Framework es la que apareció a principios del año 2002 junto con Visual Studio .NET. En esa primera versión de Visual Basic, el motor en tiempo de ejecución (CLR) es el conocido como .NET Framework 1.0. El nombre interno del compilador es Visual Basic 7.0 o también Visual Basic .NET 2002.

Un año después hizo su aparición Visual Studio .NET 2003 y con él llegó Visual Basic 7.1 o Visual Basic 2003, la versión de .NET que incluía esa versión es la 1.1.

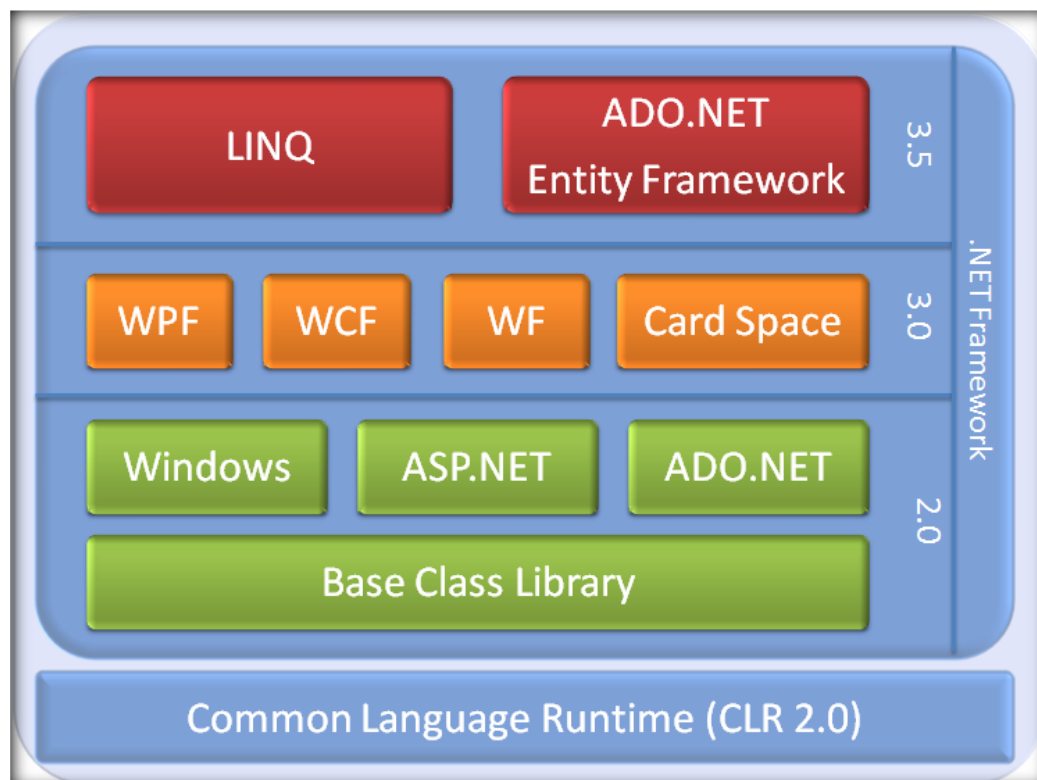


Figura 1.2. Las versiones de .NET Framework que están relacionadas con la versión 2.0 del motor en tiempo de ejecución (CLR)

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. Cuando se hace referencia al nombre de Visual Basic, siempre se usa ese mismo número, pero en realidad la numeración interna del compilador es la 8.0. La versión de .NET Framework también cambió de numeración, pasando a ser la versión 2.0.

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 Visual Basic 8.0 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 Visual Basic 9.0, que es el protagonista de este libro. 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 (Visual Basic 9.0) 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 Visual Basic, 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 Visual Basic 9.0 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 Visual Basic. 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 Basic 2008

Tal como he comentado, la creación de programas en Visual Basic 9.0 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 Visual Basic 9.0 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 Visual Basic, 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 Visual Basic. 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 Visual Basic, la versión gratuita es Visual Basic 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 Visual Basic.

### ***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 Basic 2008 Express Edition).*

## Mejoras en IntelliSense

Lo primero que notaremos al usar el IDE de Visual Basic 2008, y que desde mi punto de vista es una característica que facilita la escritura de código, es que nada más empezar a escribir ya vemos las instrucciones que podemos usar. Los que hayan usado el lenguaje C# para escribir sus aplicaciones ya sabrán que esta característica estaba presente en la edición anterior del entorno integrado,

pero en Visual Basic se ha mejorado, ya que solo nos muestra las instrucciones que podemos usar en cada contexto en el que empezamos a escribir el código.

En la figura 1.3 vemos cómo IntelliSense nos muestra las opciones que podemos usar a nivel de un archivo de clase (en esta captura, en una aplicación de tipo consola).

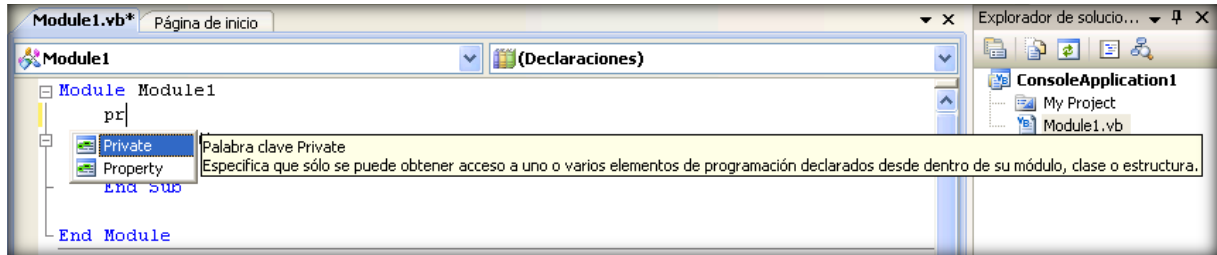


Figura 1.3. IntelliSense en Visual Basic 2008

Además de esta nueva funcionalidad, IntelliSense es siempre sensible al contexto -esto ya lo vimos en la versión anterior-, pero ahora solo muestra las opciones que tenemos disponibles acorde a la región de código donde nos encontramos, como podemos apreciar en la figura 1.3, que solo muestra las opciones que tenemos disponibles y que empiezan por las letras que hemos tecleado.

Como veremos a lo largo de este libro, se han agregado nuevas instrucciones al lenguaje. Algunas de esas nuevas palabras reservadas solo son aplicables en ciertos contextos, y en estos casos, IntelliSense también muestra solo aquellas instrucciones que podemos usar, tal como vemos en la figura 1.4 cuando escribimos un código que utiliza una de las nuevas instrucciones que dan soporte a LINQ.

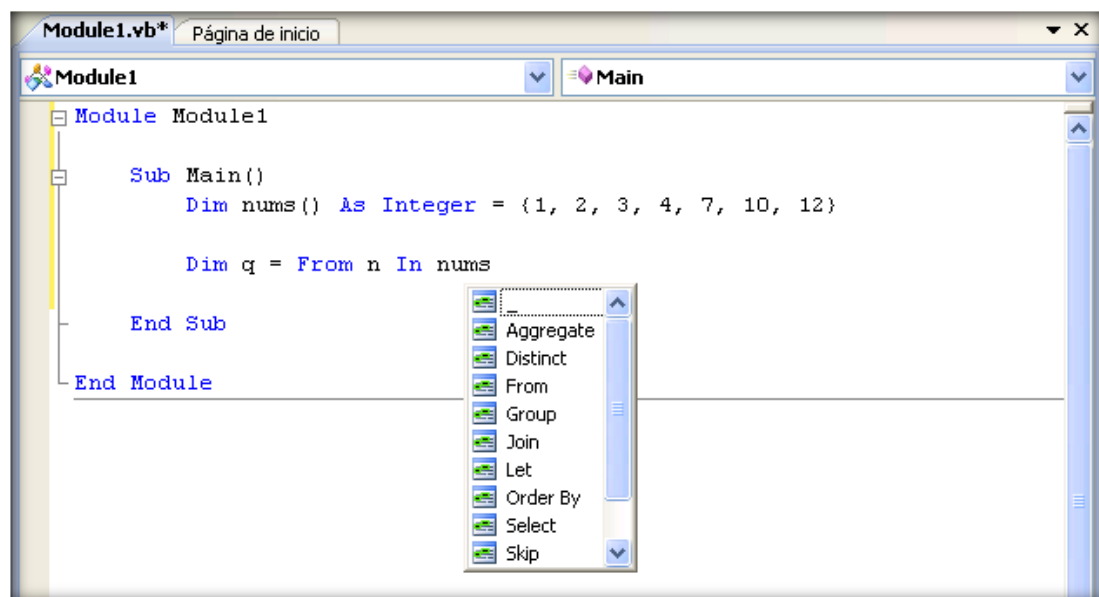


Figura 1.4. IntelliSense y las instrucciones de LINQ



Además de mostrar las instrucciones que podemos utilizar en cada momento, también mostrará las variables y otros elementos que hayamos definido en nuestro código, tal como podemos apreciar en la figura 1.5. Esto es bastante útil en situaciones como la mostrada en esa figura o cuando escribimos dentro de un procedimiento, ya que, además de las clases e instrucciones propias del lenguaje, nos mostrará las variables usadas como parámetros y aquellas que se encuentren definidas como accesibles por su alcance, como las declaradas a nivel de la clase o módulo, o las públicas y globales.

## Visual Basic 2008 y las líneas continuadas

Como sabemos, Visual Basic es un lenguaje en el que cada instrucción debe estar en una misma línea (además de aceptar varias instrucciones en una misma línea si esas instrucciones están separadas por dos puntos, pero esto último, afortunadamente, cada día se usa menos). Aunque desde hace unas cuantas versiones (si mi memoria -y la de **Daniel Seara**- no falla, fue en Visual Basic 4.0) se puede continuar una línea física si usamos el continuador de líneas (guión bajo).

El IDE de Visual Basic 2008 ha mejorado el uso de esa continuación de líneas, de forma que las instrucciones que queremos continuar se mantienen en la misma columna que la de la línea anterior (como podemos comprobar en la figura 1.5).

En otras ocasiones, esa sangría puede ser diferente (según el contexto), por ejemplo, al definir un método con parámetros y usar la continuación de líneas, los parámetros se alinean en la columna en la que empieza el primero de ellos, tal como vemos en el listado 1.1.

Esta forma de alinear las líneas es la predeterminada de Visual Basic y se usará cuando en las opciones de sangría esté la opción **Inteligente** (*Smart*). Si cambiamos el valor a cualquiera de los otros dos (ver la figura 1.6), la sangría se hará al principio de la línea siguiente (justo en la primera columna con código) si elegimos **Bloque** (*Block*) o simplemente no se realizará ninguna sangría si elegimos la opción **Ninguna** (*None*).

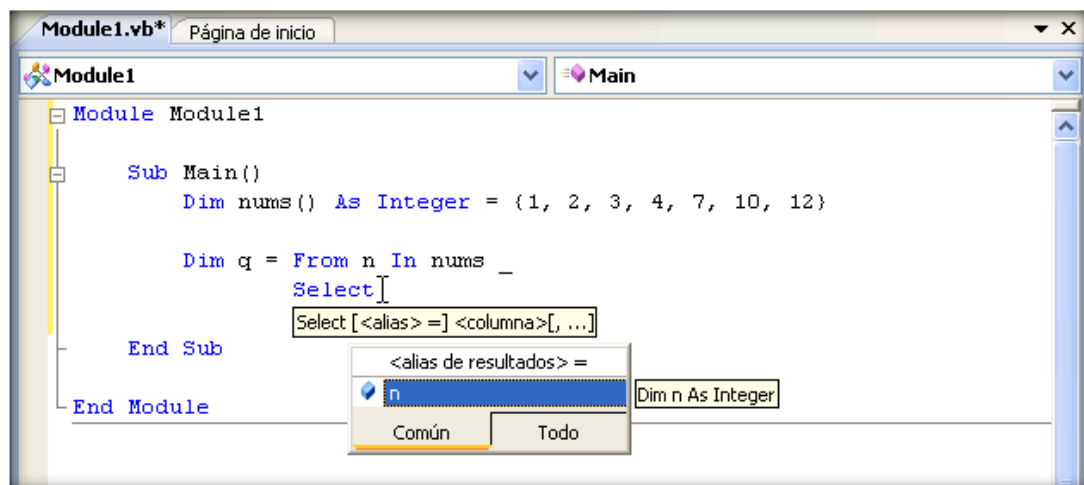


Figura 1.5. Las variables también se incluyen en la lista de IntelliSense

```
Private Sub Form1_Load( ByVal sender As Object, _  
                        ByVal e As EventArgs) _  
                        Handles MyBase.Load
```

Listado 1.1. Sangría de los parámetros de los métodos al usar la continuación de líneas

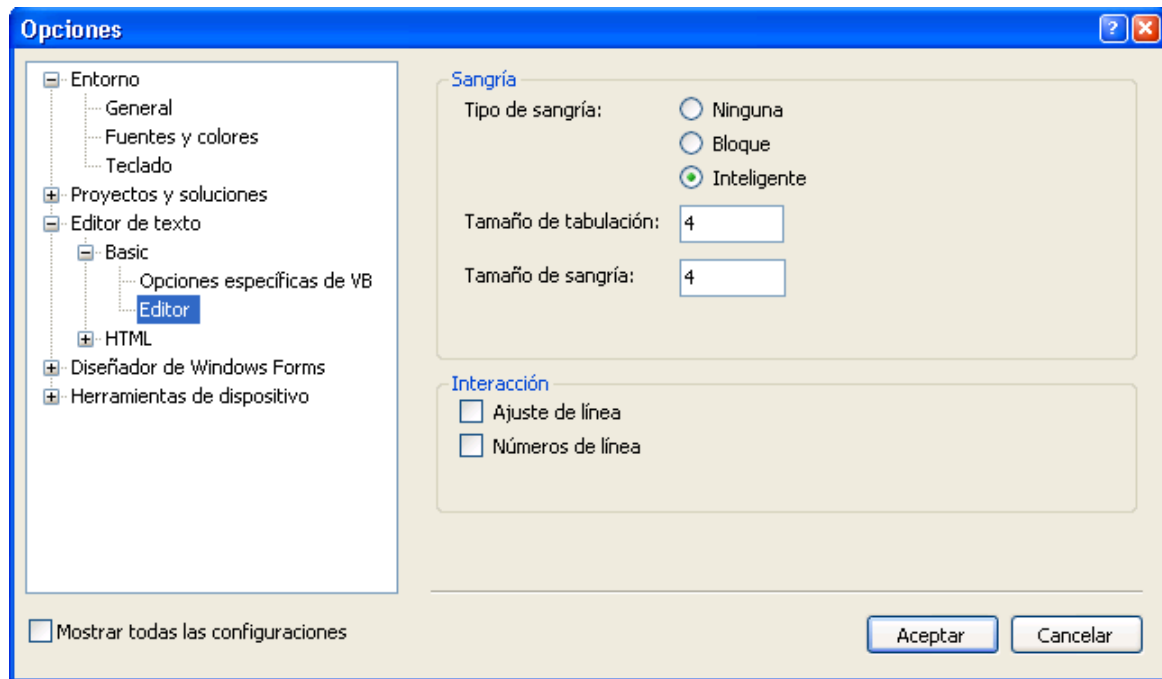


Figura 1.6. Opciones de sangría de Visual Basic

Para mostrar estas opciones de sangrado, lo haremos desde **Opciones** (*Options*) del menú **Herramientas** (*Tools*).

En cualquier caso, si usamos la opción **Inteligente** (*Smart*) y no existe un modelo de sangría que el propio Visual Basic controle, los parámetros y el resto de alineaciones se harán siempre en la misma columna en la que empieza la línea anterior al uso del continuador de líneas. Si el IDE de Visual Basic propone una columna y nosotros decidimos cambiarla (porque nos guste más o porque se adapte a nuestras preferencias), el resto de líneas se alinearán con la primera columna de la línea anterior, tal como vemos en el código del listado 1.2.

```
Private Sub Button1_Click(ByVal sender As Object, _  
                          ByVal e As EventArgs) _  
                          Handles Button1.Click
```

Listado 1.2. En cualquier momento podemos usar la alineación de nuestra preferencia

## Tipos de proyectos y versión de .NET Framework

Como ya he comentado, con Visual Basic 9.0 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 Visual Basic 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 Visual Basic.

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

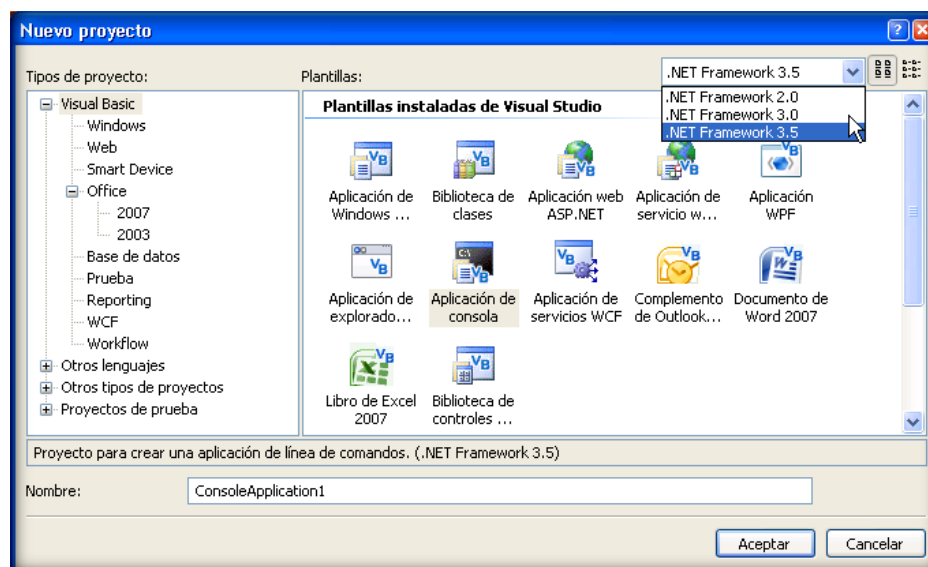


Figura 1.7. Los proyectos de Visual Basic para .NET Framework 3.5

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 Basic 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 **Compilar** (*Compile*), particularmente desde el botón **Opciones de compilación avanzadas** (*Advanced Compile Options*) en las propiedades del proyecto, y tal como vemos en la figura 1.8, existe una opción para 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.*

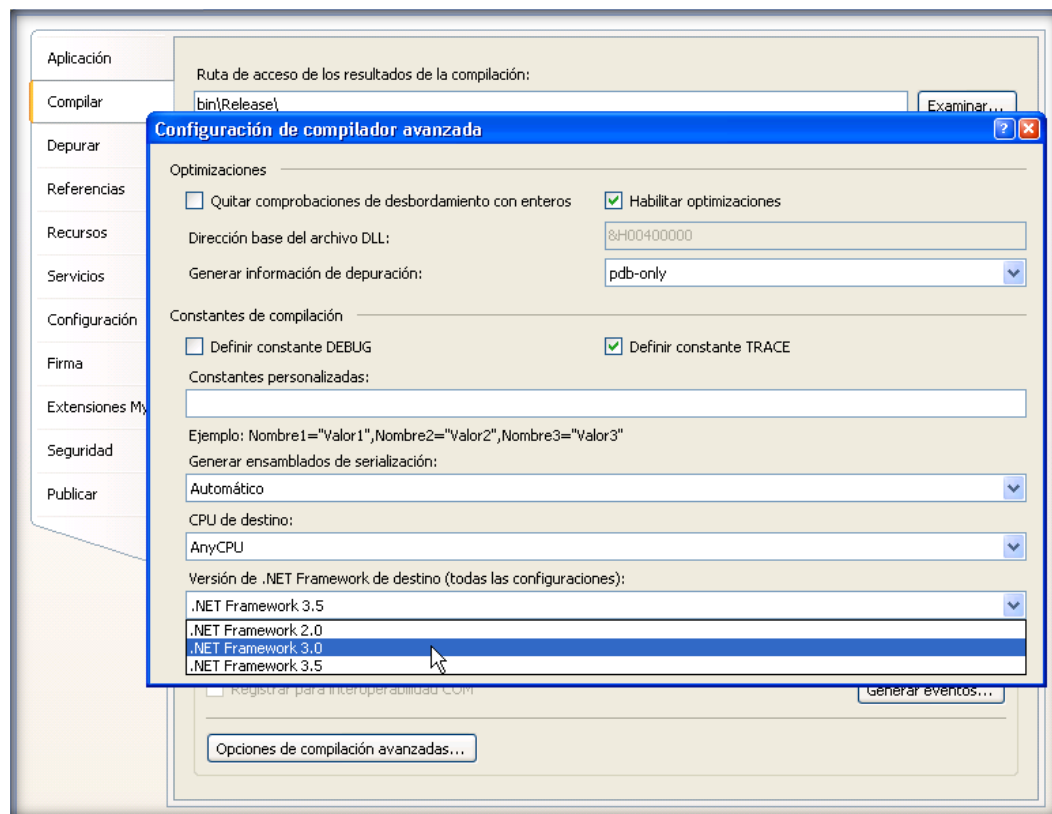


Figura 1.8. Cambiar la versión de .NET desde las propiedades del proyecto

## ¿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 Visual Basic 9.0 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.

## Opciones de compilación de Visual Basic

Desde la aparición de Visual Basic para .NET, siempre han existido tres opciones que afectan a la compilación del código, principalmente en la forma que el compilador tratará la declaración y uso de ciertas asignaciones o conversiones entre tipos diferentes.

En Visual Basic 9.0 se ha agregado una nueva, que como veremos más adelante, influye en la forma de declarar las variables. En la figura 1.9 vemos esas cuatro opciones que podemos configurar en el entorno de desarrollo de Visual Basic 2008.

La novedad es la última opción: **Option Infer**, de la que nos ocuparemos en el próximo capítulo. El valor predeterminado que tienen esas opciones al instalar Visual Basic/Studio 2008 son las mostradas en la figura 1.9. Como vemos, **Option Strict** está desactivado (tiene un valor **Off**), y **Option Infer** tiene un valor predeterminado de **On** (activado).

### ***Nota***

*Más adelante volveremos a hablar sobre las opciones de Visual Basic, principalmente de Option Strict y Option Infer y cómo pueden afectar a las declaraciones de las variables.*

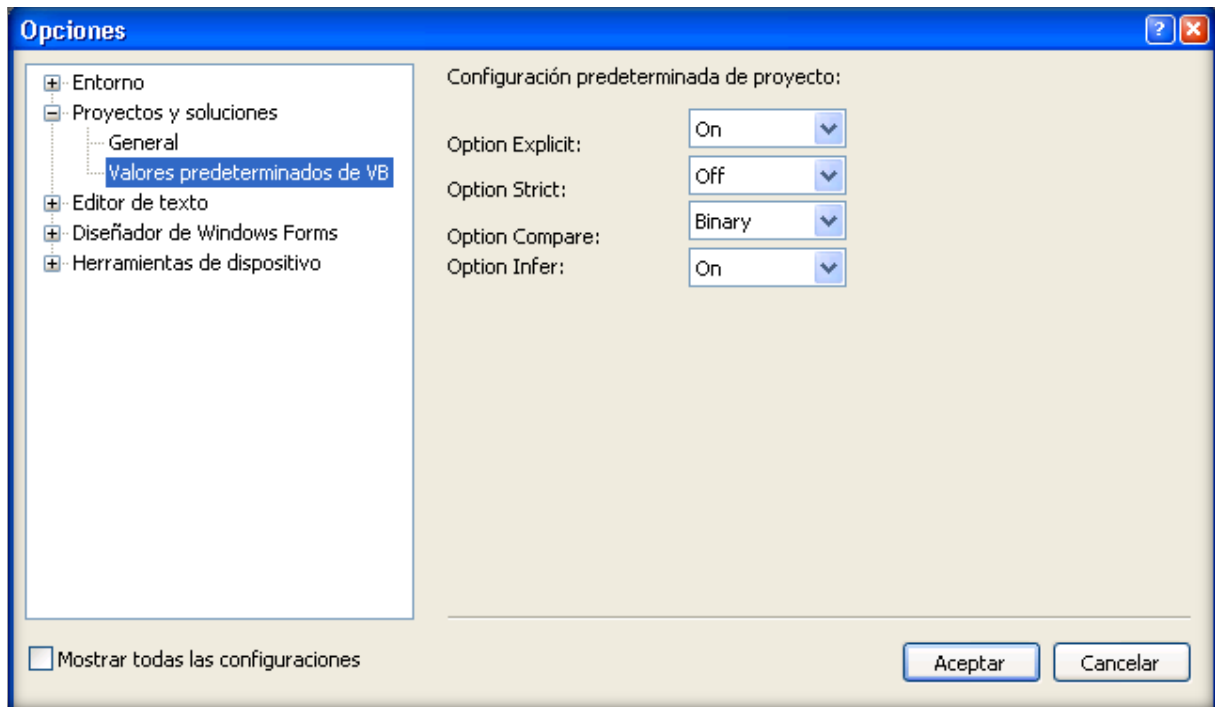


Figura 1.9. Opciones de compilación de Visual Basic

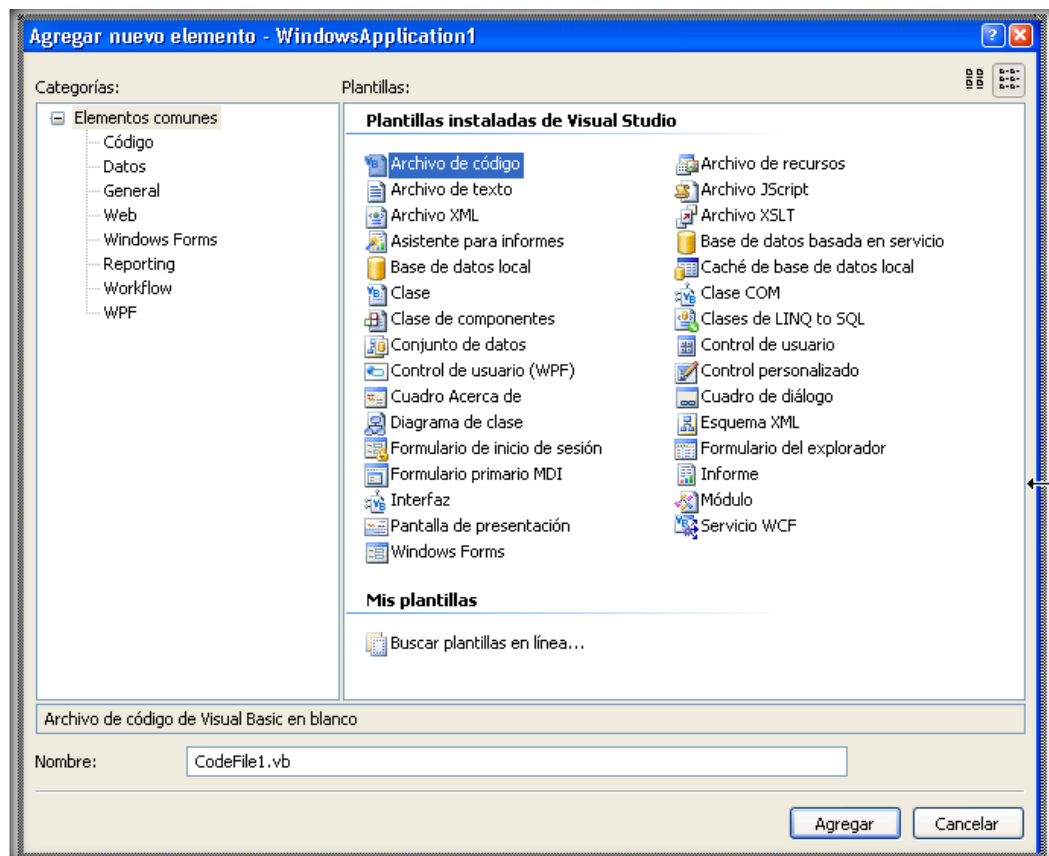
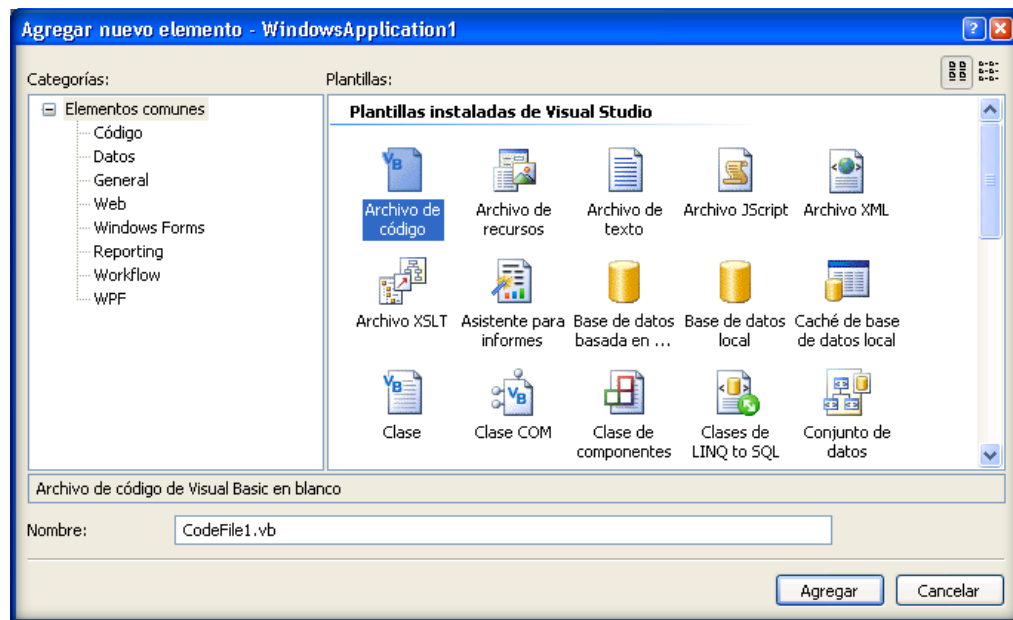
## Nuevo diseño del cuadro de diálogo de agregar elementos a un proyecto

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 1.10) 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 Basic/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 1.11.

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.



## 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 Basic

Antes de dar por finalizado este capítulo sobre el entorno de desarrollo de Visual Basic 2008, me gustaría comentar algo que, si bien no es nuevo, sí que considero importante que cualquier usuario de Visual Basic 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 C# 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 Basic), 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 1.12 podemos ver las opciones en una instalación de Visual Studio 2008 Professional.

Si Visual Basic es el lenguaje con el que habitualmente trabajaremos, seguramente elegiremos la opción del entorno de desarrollo para programadores de Visual Basic (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.



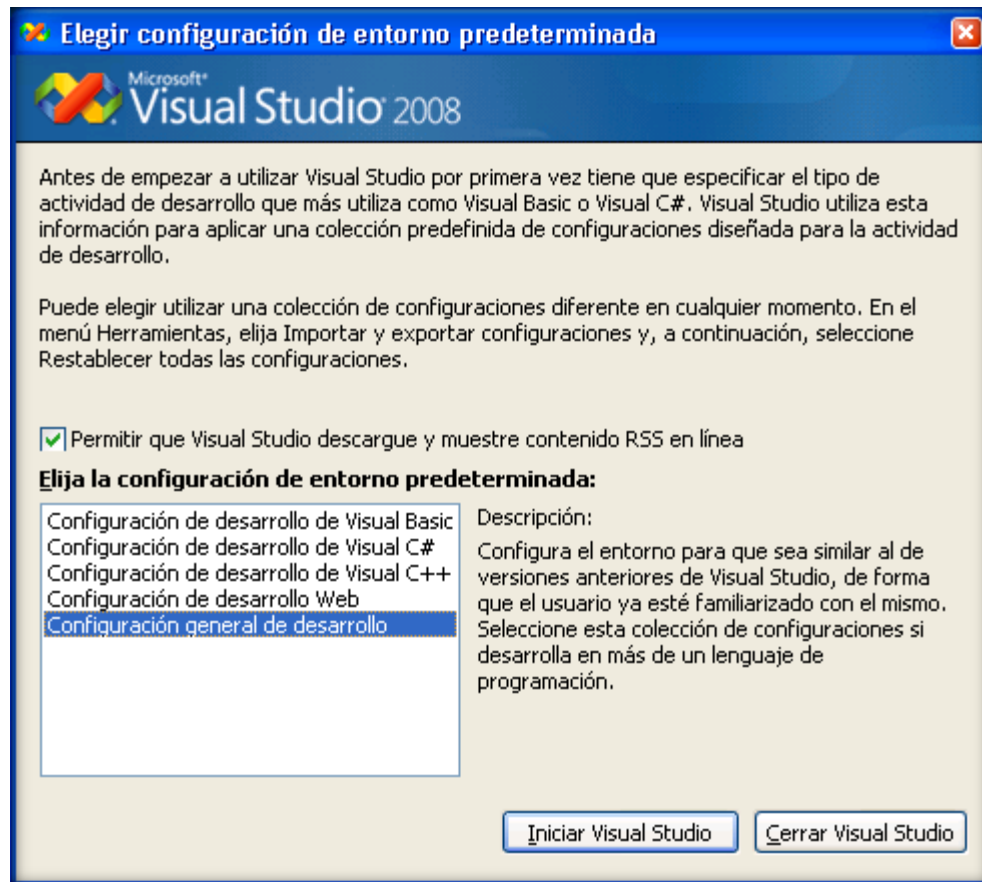


Figura 1.12. Al iniciar Visual Studio por primera vez podemos elegir la configuración del entorno

El que yo seleccione las opciones generales en lugar de Visual Basic es más bien por costumbre de usar las combinaciones propias del IDE con el que trabajo. 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.

Pero no son solo las combinaciones de teclas las que cambian de una configuración a otra. Por ejemplo, en 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 pensaran. Por ejemplo, si elegimos la configuración para los desarrolladores de Visual Basic, no tendremos a nuestra disposición las opciones del tipo de compilación que queremos crear. Esas opciones de compilación nos permiten elegir si el proyecto que estamos creando será una versión final (**Release**) o una versión de depuración (**Debug**).

Si queremos cambiar la forma de crear nuestros proyectos, podemos hacerlo fácilmente desde las opciones generales de configuración del entorno de desarrollo. En el menú **Herramientas** seleccionamos **Opciones** y en las opciones de **Proyectos y soluciones** seleccionamos la opción **Mostrar opciones avanzadas de compilación** (esto es aplicable tanto a Visual Studio como a la versión Express).

La configuración de Visual Basic también oculta algunas opciones de ese cuadro de diálogo, por tanto, si queremos tenerlas todas a la vista, o bien elegimos otra configuración del entorno de trabajo, o seleccionamos la opción mostrada en la parte inferior izquierda de ese cuadro de diálogo, y, tal como podemos ver en las figuras 1.13 y 1.14, la cantidad de opciones es significativa. Cuando seleccionamos la opción general de desarrollo, todas las opciones están siempre a nuestra disposición.

Independientemente del entorno que hayamos elegido al iniciar por primera vez Visual Studio (figura 1.12), 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 1.15 podemos ver que tenemos las mismas opciones que podemos elegir al iniciar Visual Studio.

### 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.*

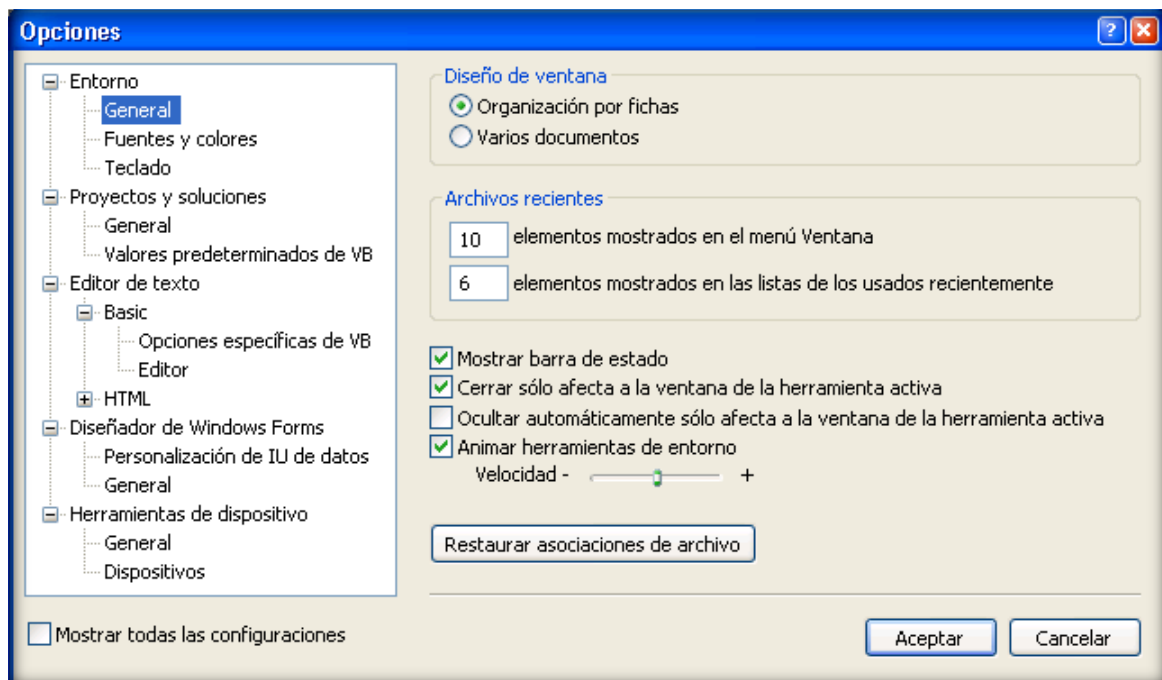


Figura 1.13. Opciones mínimas en la configuración de Visual Basic

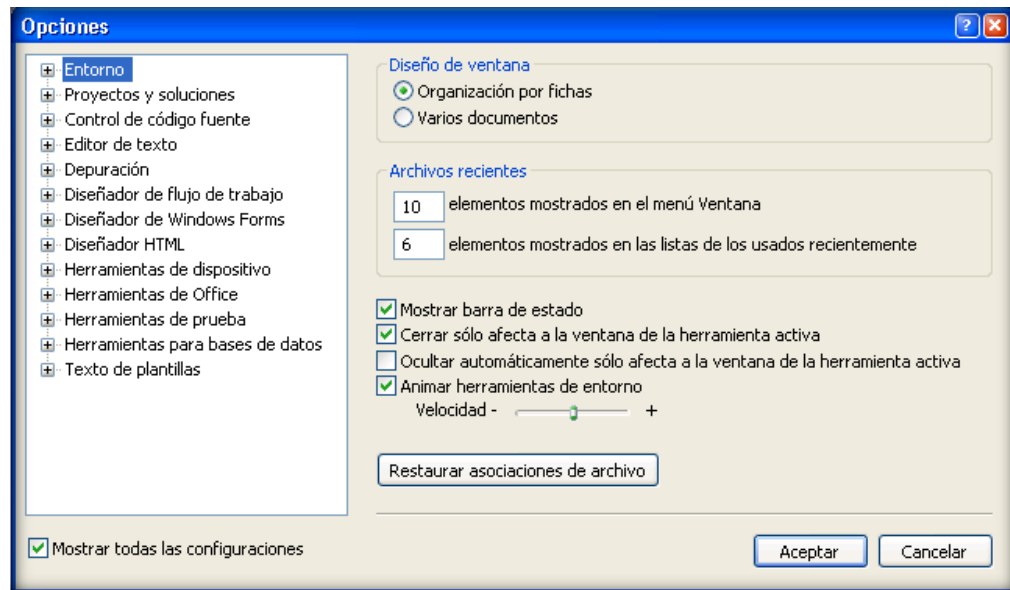


Figura 1.14. Podemos ver todas las opciones siempre que queramos

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 9.0 de Visual Basic.

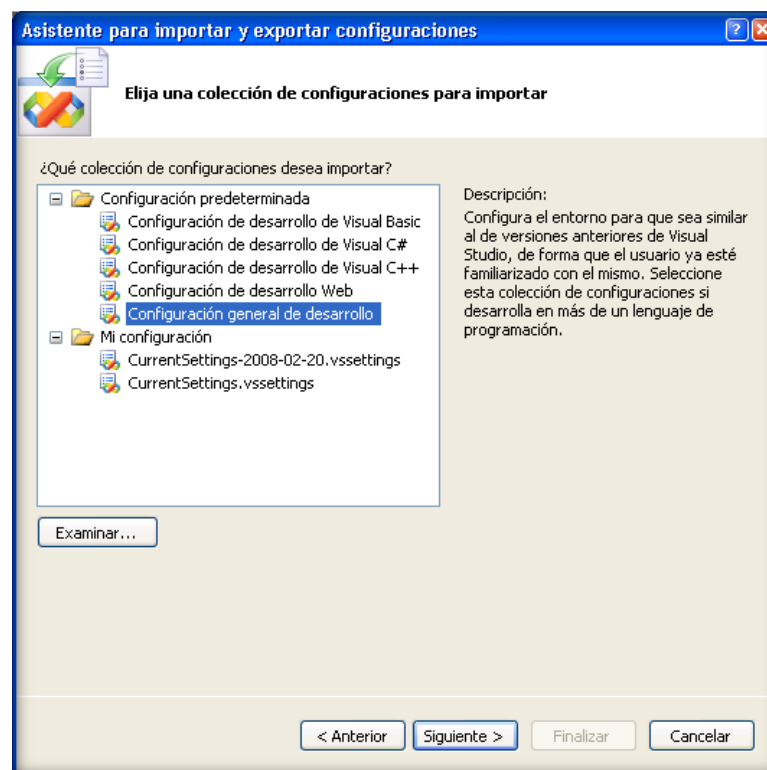


Figura 1.15. Las opciones del entorno en exportar e importar configuraciones

## Parte 2

# Características generales del lenguaje

---

En esta segunda parte del libro, empezaremos viendo las novedades de Visual Basic, pero centrándonos más en el compilador, es decir, las novedades, que de alguna forma están más relacionadas con el compilador de Visual Basic 9.0 que con el resto de ensamblados que acompañan a .NET Framework 3.5.

Estas características (al igual que casi todas las novedades de Visual Basic 9.0) 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 la siguiente parte del libro.

Algunas de ellas las volveremos a ver con más detalle o desde otros puntos de vista, pero unas pocas solo las trataremos en esta parte del libro.

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 Basic 2008 Express Edition).*

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

# Capítulo 2

## Características generales del lenguaje (I)

---

### Introducción

En esta primera aproximación a las novedades de Visual Basic 9.0 veremos los siguientes temas:

- Inferencia de tipos en variables locales.
- Inicialización de objetos.
- Métodos parciales.
- Tipos anulables.
- Ensamblados amigos.
- Agilidad del *runtime* de Visual Basic.

### Inferencia de tipos en variables locales

La inferencia automática de las variables locales está relacionada con la opción **Option Infer**, que como vimos en el capítulo anterior está activada de forma predeterminada en Visual Basic 2008.

La inferencia automática de tipos es una característica de Visual Basic por medio de la cual podemos dejar que sea el propio compilador el que se encargue de asignar el tipo que tendrá una variable a partir del valor que le estamos asignando.

Esto supone que si el compilador de Visual Basic es capaz de averiguar el tipo de una variable según el valor que estamos indicando en la parte derecha de la asignación, creará esa variable como si hubiésemos indicado el tipo de datos al declarar esa variable.

Por ejemplo, si tenemos esta asignación:

```
Dim nombre = "Guillermo"
```

El compilador (de forma interna) en realidad hará esta declaración:

```
Dim nombre As String = "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.

Para disponer de esta inferencia automática de tipos, debemos usar la instrucción **Option Infer On**, la que podemos activar tanto a nivel de entorno de desarrollo, como a nivel de proyecto o a nivel de archivo.

### ***Nota***

*Tal como podemos deducir por el título de esta sección, la inferencia automática de tipos solo la podemos usar a nivel de variables locales, es decir, de las variables que usemos dentro de un procedimiento, pero nunca a nivel de la clase o para el valor devuelto por una función o los parámetros de cualquier método o delegado.*

## **La inferencia de tipos y Option Strict**

La inferencia de tipos es independiente de cómo usemos **Option Strict**, es decir, podemos tener desactivada la comprobación estricta del código y activada la inferencia de tipos.

Esto último es importante tenerlo en cuenta, ya que puede influir (y mucho) en nuestro código.

En las versiones anteriores de Visual Basic, y debido a que el valor predeterminado de **Option Strict** está desactivado (**Off**), cuando declaramos una variable como acabamos de ver, esa variable será de tipo **Object**, ya que **Object** es el tipo usado por Visual Basic cuando no se indica el tipo de datos, por tanto, en Visual Basic 2005 (o cualquier versión anterior de .NET), la siguiente declaración crearía una variable de tipo **Object**:

```
Dim nombre = "Guillermo"
```

Y debido a que en este caso *nombre* es de tipo **Object**, nada nos impide asignarle cualquier otro valor, por ejemplo un número:

```
nombre = 125
```

Esto es totalmente correcto, bueno, más que correcto debería decir “válido”, ya que si no tenemos activada la comprobación estricta de tipos, nada nos impide asignar un valor cualquiera a esa variable, incluso si tuviésemos activada la opción **Option Strict**, también sería “legal” hacer esa asignación, por la sencilla razón de que el tipo **Object** acepta cualquier valor y de cualquier tipo de datos.

Ésta es una de las razones por la que insisto que siempre trabajemos con **Option Strict On**, ya que si tenemos activada la comprobación estricta, la declaración de esa variable la teníamos que haber

hecho indicando el tipo de datos que queremos usar, y si nuestra intención es que fuese **Object**, con idea de que acepte cualquier valor, tendremos que definirla de esta forma:

```
Dim nombre As Object = "Guillermo"
```

De esta forma, es evidente que el tipo de datos de la variable *nombre* es **Object**.

Pero si nuestra intención es que la variable *nombre* fuese de tipo cadena, la tendríamos que haber declarado usando el tipo **String**:

```
Dim nombre As String = "Guillermo"
```

El problema de tener **Option Strict** desactivado es que, a pesar de definir la variable como **String**, podemos asignarle cualquier valor, por ejemplo, un número. Pero si activamos **Option Strict**, al asignar un valor diferente al tipo de datos de la variable, el compilador nos avisará de que eso no es posible, e incluso nos ofrecerá una posible solución, que consiste en usar una función de conversión. De esta forma no solucionamos el problema, pero seremos conscientes de que es posible que esa conversión falle, al menos si el valor que indicamos a la derecha de la asignación no contiene un valor válido del tipo al que queremos convertir.

No quiero insistir en las bondades de tener activado **Option Strict**, entre otras cosas porque nos estaríamos desviando del tema que nos ocupa. Pero sí quiero que quede claro que nos evitaremos algún que otro quebradero de cabeza si escribimos todo nuestro código usando **Option Strict On**.

Volvamos a retomar el tema de la inferencia de tipos. Si **Option Infer** está activado (que es como lo tendremos en los nuevos proyectos que creemos), el compilador usará el tipo adecuado en las declaraciones en las que no indiquemos el tipo de datos, y como ya he comentado, esa “averiguación” del tipo de datos es independiente de cómo tengamos **Option Strict**. Por tanto, si declaramos la variable *nombre* sin indicar el tipo de datos, esa variable tendrá el mismo tipo que el dato que estamos asignando (en nuestro ejemplo, una cadena).

Si además de **Option Infer** también tenemos activado **Option Strict**, el compilador no nos dejará asignar un valor que no se corresponda con el tipo de datos que la variable ha “inferido”, ya que la variable tendrá definido un tipo concreto de datos y **Option Strict** no nos permite hacer asignaciones “a la ligera”.

## La inferencia de tipos en proyectos convertidos desde versiones anteriores

Con Visual Studio 2008 (y también con la versión Express) podemos abrir cualquier proyecto que hayamos creado con versiones anteriores de Visual Basic, incluso si la versión es anterior a Visual Basic para .NET (en este caso, solo se admiten proyectos de Visual Basic 6.0).



El asistente convertirá el proyecto para que lo podamos usar con Visual Basic 9.0, y debido a que **Option Infer** no existía en esas versiones anteriores, en esos proyectos se asignará un valor desactivado a la inferencia de tipos. Esto más que nada es para mantener la compatibilidad hacia atrás, de forma que si tenemos código en el que no se indica el tipo de datos de las variables, esas variables así declaradas serán de tipo **Object**.

Por supuesto, podemos activar la inferencia de tipos en proyectos que hemos convertido, pero debemos tener en cuenta que es posible que algunas líneas de código las tengamos que reformar, sobre todo si también activamos **Option Strict**.

Si el proyecto lo convertimos a partir de Visual Basic 6.0, el valor de **Option Strict** será desactivado, y si la conversión la hacemos a partir de proyectos de Visual Basic 7.0 o superior, se mantendrá el valor que tuviésemos en esos proyectos. Pero en ambos casos, **Option Infer** estará desactivado.

## ¿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. Si este automatismo lo aderezamos con la comprobación estricta de tipos y conversiones, seguro que no resultará ningún inconveniente, ya que no habrá posibilidad de que creemos por error (o dejadez) variables “atipadas”.

Por un lado, tenemos 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 IDE de Visual Basic, al pasar el puntero del ratón por una variable, nos indicará de qué tipo es (ver la figura 2.1).

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 Visual Basic nos ofrece.

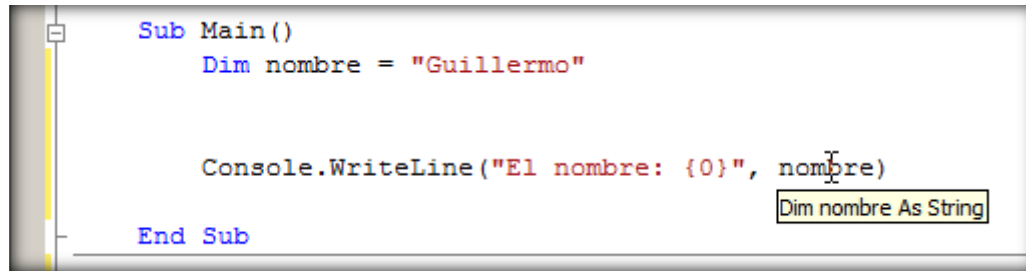


Figura 2.1. El IDE de Visual Basic nos indica siempre el tipo de datos de las variables

Un sitio propicio al uso de la inferencia de tipos es en los bucles, ya sean del tipo **For** o **For Each**, 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 i = 1 To 10
    Console.WriteLine(i.ToString)
Next

Dim nombres() As String = {"Pepe", "Luis", "Eva"}

For Each s In nombres
    Console.WriteLine(s)
Next
```

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 Visual Basic sea capaz de averiguarlo, y si no es capaz, simplemente nos avisará con un error, y debido a que el IDE “pre compila” el código mientras lo escribimos, ese error lo veremos antes de que entreguemos la aplicación al usuario final.

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.

```
Dim sKey = TextBox2.Text
Dim 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 Visual Basic sabe que ése debe ser el tipo de datos de la variable *tvn*.

Cuando veamos las novedades de Visual Basic 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**

La inicialización de objetos es otra 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?

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 Visual Basic, 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.

```
Dim cli As 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

Visual Basic también nos permite el uso de la instrucción **With**, que suele ser útil en situaciones como la mostrada en el listado 2.4, ya que nos permite alcanzar a los miembros *accesibles* del objeto indicado, con lo que evitamos tener que repetir el nombre de la variable que contiene, en este caso, las propiedades que queremos asignar.

```
Dim cli As New Cliente

With cli
    .Nombre = "Guillermo"
    .Apellidos = "Som Cerezo"
    .Correo = "guillermo@nombres.com"
End With
```

Listado 2.4. Creación y asignación de un objeto de la forma clásica, ayudados de With

En Visual Basic 9.0 esta inicialización y asignación la podemos simplificar uniendo en una sola instrucción el código mostrado en el listado 2.4. Esta nueva forma es la que podemos ver en el código del listado 2.5.

```
Dim cli As New Cliente With {.Nombre = "Guillermo", _
                             .Apellidos = "Som Cerezo", _
                             .Correo = "guillermo@nombres.com" _
                             }
```

Listado 2.5. Creación y asignación de un objeto usando la inicialización de objetos de VB9

Como vemos en el listado 2.5, lo que hacemos es una especie de mezcla del código mostrado en el listado 2.4, y unimos la declaración de la variable con la asignación de las propiedades.

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 2.5 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. Pero si el lector se ha quedado con la idea mostrada en la equivalencia del código del listado 2.4 y el del listado 2.5, le resultará fácil deducir lo que habría que hacer en este caso que comentamos. 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 2.6 sería totalmente correcto.

```
Dim cli2 As New Cliente2("1234567890") With {.Nombre = "Guillermo"}
```

Listado 2.6. 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 Visual Basic, 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 2.5 y 2.6) tal como vemos en el listado 2.7.

```
Dim cli = New Cliente With {.Nombre = "Guillermo", _  
                           .Apellidos = "Som Cerezo", _  
                           .Correo = "guillermo@nombres.com" _  
                           }  
  
Dim cli2 = New Cliente2("1234567890") With {.Nombre = "Guillermo"}
```

Listado 2.7. Inicialización de objetos e inferencia de tipos

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.

La verdad es que en el caso de Visual Basic y los objetos por referencia, la inferencia de tipos no nos ahorra mucho, al menos si el ahorro lo miramos por cuánto tenemos que teclear, ya que si nos fijamos en los cambios realizados en los tres últimos listados, solo tendríamos que cambiar el signo igual por la cláusula **As**.

## Inicializaciones de arrays y colecciones

Visual Basic 9.0 no soporta esta nueva característica que sí está presente en otro de los lenguajes de .NET Framework como es C#, pero la he querido resaltar como si se tratase de una novedad por si alguien ha visto ejemplos de las novedades de C# y quiere tener una equivalencia de esta novedad en Visual Basic.

En las charlas que doy (y en las que daré en el futuro, al menos hasta que Visual Basic disponga nativamente de esta característica), lo que suelo mostrar es un código que simula esta posibilidad de inicializar colecciones de objetos, es decir, definir una colección y asignarle objetos individuales que podemos crear al vuelo.

La forma de crear esas colecciones es como se muestra en el listado 2.8. Primero veremos cómo creamos las colecciones y después veremos el “truco”.

```
Dim clis = CrearLista(Of Cliente)(cli, cli2, _
    New Cliente With {.Nombre = "Juan"}, _
    New Cliente With {.Nombre = "Eva"} _
)
```

Listado 2.8. Un truco para inicializar colecciones en Visual Basic 9.0

En el código del listado 2.8 suponemos que las variables *cli* y *cli2* ya están instanciadas y son objetos del tipo *Cliente*. El truco está en la llamada al método *CrearLista*, el cual lo podemos definir tal como vemos en el listado 2.9.

```
Function CrearLista(Of T)(ParamArray datos() As T) As IEnumerable(Of T)
    Dim lista As New List(Of T)
    lista.AddRange(datos)
    Return lista
End Function
```

Listado 2.9. La función que permite simular la inicialización de colecciones

La función *CrearLista*, al usar un tipo *generic* y parámetros opcionales usando un *array*, hace que sea simple convertir ese *array* de parámetros opcionales en una colección. Incluso en el código de esa función podríamos haber usado un método extensor que en la versión 3.5 de .NET se añade a todas las clases que implementen *IEnumerable(Of T)*. El contenido de la función *CrearLista* lo podríamos dejar sólo con esta línea de código:

```
Return datos.AsEnumerable
```

En el caso de los *arrays*, podemos iniciarlos y asignar los valores, pero esto no ha cambiado con respecto a cómo se hacía en la versión anterior, la diferencia es que ahora podemos crear nuevos objetos individuales usando la inicialización de objetos, que vimos en el apartado anterior, tal como se muestra en el código del listado 2.10.

```
Dim clientes() As Cliente = {New Cliente With {.Nombre = "Pepe"}, _
    New Cliente With {.Nombre = "Luis"} _
}
```

Listado 2.10. La forma de inicializar los arrays no ha cambiado con respecto a versiones anteriores

También podemos usar el método *CrearLista* para asignar los valores a un *array*; en ese caso, volvemos a usar una de las nuevas características que ofrece .NET Framework 3.5 por medio de LINQ: los métodos extensores. En esta ocasión, usaremos el método *ToArray*, que es un método definido en el espacio de nombres *System.Linq*, que extiende la funcionalidad de la interfaz *IEnumerable(Of T)*. En el código del listado 2.11 podemos ver cómo usar ese método para crear un *array*.

```
Dim clientes() = CrearLista( _  
    New Cliente With {.Nombre = "Pepe"} _  
).ToArray
```

Listado 2.11. Creación de un array a partir de un método extensor

*Los métodos extensores los veremos con más detalle en otro capítulo y solo son utilizables con la versión 3.5 de .NET Framework.*

### 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 Basic anteriores a la versión 9.0.

En Visual Basic 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 Then  
    Private Sub soloEnDebug()  
        ' Lo que haga este método  
    End Sub  
#End If
```

Listado 2.12. Un método que solo se compilará si la constante DEBUG no es cero

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 tenga un valor falso (o cero) 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 el valor de la constante **DEBUG** sea distinto de cero (verdadero). El código del listado 2.13 muestra una forma de referirse a ese método.

```
#If DEBUG Then
    soloEnDebug()
#End If
```

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 ... #End If** 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 Private Sub unMetodoParcial()

End Sub
```

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 Then
    Private Sub unMetodoParcial()
        ' Lo que tenga que hacer este método
    End Sub
#End If
```

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 Visual Basic que no soportan esta nueva característica de creación de métodos parciales.

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 Visual Basic 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 **Sub** (no pueden devolver un valor).
- Hay que definirlos como privados (**Private**).
- Solo usaremos la instrucción **Partial** en el prototipo.
- Los nombres de los parámetros de la implementación del método deben coincidir exactamente con los nombres de los parámetros del prototipo.

Si el método parcial lo usamos como parámetro de **AddressOf**, la “magia” de los métodos parciales desaparece, por tanto, si esa es nuestra intención, dicha llamada o uso del método parcial con

**AddressOf** debemos realizarla solo 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 Visual Basic 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 después de **AddressOf** 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 indicado después de **Address-Of**, tal como se muestra en el listado 2.17.

```
Partial Private Sub paraUnDelegado()  
End Sub  
  
Private Delegate Sub unDelegado()  
  
#Const PROBANDO = False  
#If PROBANDO Then  
    Private Sub paraUnDelegado()  
        Console.WriteLine("Desde paraUnDelegado")  
    End Sub  
#End If  
  
Sub pruebaDelegado()  
    Dim t As New System.Threading.Thread(AddressOf paraUnDelegado)  
    t.Start()  
End Sub
```

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

```
#If PROBANDO Then  
  
    Sub pruebaDelegado()  
        Dim t As New System.Threading.Thread(AddressOf paraUnDelegado)  
        t.Start()  
    End Sub  
  
#End If
```

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

## Versiones

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

## Tipos anulables

Los tipos anulables ya estaban presentes en Visual Basic 8.0, aunque a diferencia de lo que ocurrió con C# 2.0, no se incluyeron como tipos propios de Visual Basic. Para usarlos teníamos que definirlos usando la clase **Nullable(Of T)**. Pero en Visual Basic 9.0 se incluyen dentro del propio lenguaje, es decir, existen instrucciones propias del lenguaje para definir variables de tipos anulables.

Las instrucciones (o palabras clave) para definir los tipos anulables son las mismas que para los tipos “normales”, solo que al final del nombre del tipo, debemos usar una interrogación. Por ejemplo, si queremos definir un tipo **Boolean** que también sea anulable (es decir, que pueda contener un valor nulo además de los valores propios del tipo), lo tendríamos que definir de esta forma:

```
Dim booleanAnulable As Boolean?
```

Los tipos anulables pueden contener un valor nulo además de los valores adecuados para el tipo y, al igual que ocurre con la clase **Nullable(Of T)**, esos tipos definen métodos y propiedades para averiguar si el contenido es un valor nulo, para recuperar el valor interno (si no es nulo) o para recuperar un valor predeterminado (en caso de que el valor sea nulo). Como vemos, todo lo que ya sepamos de los tipos anulables es aplicable a estos “nuevos” tipos de Visual Basic 9.0, la ventaja es que ahora forman parte del propio lenguaje, por lo que no es necesario que recurramos a declarar tipos usando la clase en la que se basan.

## ¿Cómo declarar variables de tipos anulables?

En Visual Basic podemos declarar variables de tipos anulables de dos formas diferentes: una es como la mostrada anteriormente, la otra es finalizando el nombre de la variable con una interrogación.

En el listado 2.18 tenemos dos formas de declarar un valor entero anulable.

```
Dim unEntero? As Integer
Dim otroEntero As Integer?
```

Listado 2.18. Dos formas de declarar tipos anulables en Visual Basic

El que definamos de una forma u otra los tipos anulables no tienen ninguna consecuencia, ya que en ambos casos obtenemos el mismo resultado, por tanto, la forma de usarlos es la misma, los declaremos como los declaremos, tal como vemos en el listado 2.19.

La inferencia de tipos también es aplicable a los tipos anulables, aunque en este caso no podremos inferir los valores a partir de constantes, aunque sí desde otra variable anulable, desde un procedimiento que devuelva un valor anulable o usando una construcción de un tipo que finalmente devuelva un valor anulable. En el listado 2.20 podemos ver algunos ejemplos de inferencia de valores anulables.

```
If unEntero.HasValue Then
    ' No contiene un valor nulo
End If

If otroEntero.HasValue Then
    ' No contiene un valor nulo
End If
```

Listado 2.19. Los declaremos de una forma u otra, siempre los usaremos de la misma forma

```
Dim uno = unEntero

Dim dos = New Integer?(22)

Dim tres = New Nullable(Of Integer)(17)

Dim cuatro = funcionAnulable()
```

Listado 2.20. Varias formas de inferir un valor de un tipo anulable

## Recordando un poco a los tipos anulables

Sin entrar en demasiados detalles, recordemos un poco qué se esconde detrás de los tipos anulables.

Tal como ya he comentado antes, los tipos anulables son tipos por valor, que pueden contener un valor en el rango del tipo base además de un valor nulo.

En los tipos de datos “normales”, cuando asignamos un valor nulo a una variable, en realidad lo que hacemos es asignar un valor “cero”. Por ejemplo, para un valor de tipo **Boolean** representaría un valor **False**, en los tipos numéricos la asignación de **Nothing** tendría como consecuencia de que el valor de esa variable se convierta en un “cero”, y en el caso de las fechas, el valor nulo representa el valor predeterminado del tipo **Date** (o **DateTime**), que en el caso de Visual Basic representa la fecha del 1 de enero del año 1 a las cero horas (01/01/0001 00:00:00).

Sin embargo, cuando usamos tipos anulables y asignamos un valor **Nothing** al declarar las variables (o simplemente no le asignamos nada), el contenido de esas variables es: “nada”, no el valor predeterminado de los tipos. Esto es así, porque los tipos anulables además del valor normal del tipo también pueden contener un valor nulo. Sí, ya sé que esto ya lo he repetido antes, pero no está de más dejar claro el nuevo comportamiento de este tipo de datos, ya que esa cualidad nos permitirá usar los tipos anulables en situaciones en las que antes era complicado de manejar, por ejemplo, al recuperar valores de una base de datos, que como sabemos, también pueden contener valores nulos, en ese caso para indicar que no se ha asignado ningún valor válido.

Si ejecutamos el código del listado 2.21 veremos que la salida es diferente según estemos trabajando con valores numéricos “normales” o estemos usando tipos anulables.

```
Dim b1 As Boolean = Nothing
Dim i1 As Integer = Nothing
Dim d1 As Double = Nothing
Dim f1 As Date = Nothing

Console.WriteLine("Boolean: {0}", b1)
Console.WriteLine("Integer: {0}", i1)
Console.WriteLine("Double : {0}", d1)
Console.WriteLine("Date   : {0}", f1)

Console.WriteLine()

Dim b2? As Boolean = Nothing
Dim i2? As Integer = Nothing
Dim d2? As Double = Nothing
Dim f2? As Date = Nothing

Console.WriteLine("Boolean: {0}", b2)
Console.WriteLine("Integer: {0}", i2)
Console.WriteLine("Double : {0}", d2)
Console.WriteLine("Date   : {0}", f2)
```

Listado 2.21. Diferencia en la asignación de un valor nulo a variables de tipos normales y anulables

Tal como vemos en la figura 2.2, en el primer caso, los valores que se mostrarán serán los “predeterminados” de esos tipos, mientras que en el segundo, simplemente no se mostrará nada, ya que no contienen nada (al usar los tipos anulables con el método **WriteLine** de la clase **Console**, se hace una llamada al método **ToString**; ese método devuelve una cadena vacía cuando la variable no contiene un valor.)

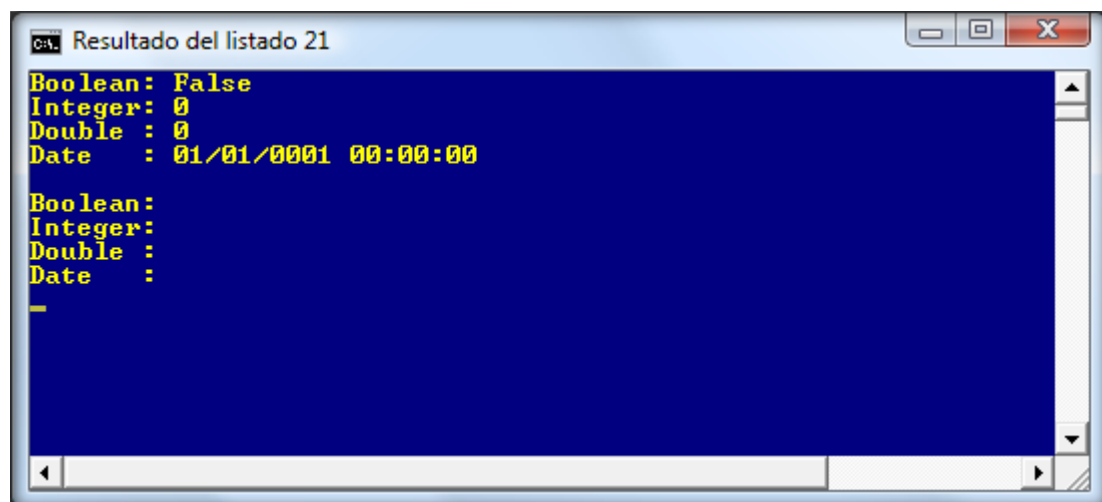


Figura 2.2. Resultado del listado 2.21

Por supuesto, los tipos anulables no solo los usaremos para trabajar con bases de datos, ya que hay otras circunstancias en las que nos pueden ser de gran utilidad, como es el caso indicado anterior-

mente en el que queremos tener la posibilidad de saber si a una variable le hemos asignado algún valor o no. Si inicialmente contienen un valor nulo, sabremos que aún no le hemos asignado un valor. En esos casos, podremos actuar de forma diferente de cuando esa variable tenga un valor “real” (o válido, para no confundir *real* con un valor numérico).

Los tipos anulables definen la propiedad **HasValue** que nos puede servir para saber si contiene un valor válido o un valor nulo. Esa propiedad simplemente devuelve un valor verdadero o falso, según tenga o no un valor válido.

En muchas circunstancias, seguramente optaremos por utilizar el método **GetValueOrDefault**. Este método devolverá el valor que tenga la variable (si dicho valor no es nulo), o bien el valor predeterminado. Este método tiene dos sobrecargas. Estas dos sobrecargas se usarán si la variable desde la que se llama no contiene un valor válido (ya que si tiene un valor que no es nulo, el valor devuelto por ambas sobrecargas será dicho valor, por tanto, es indiferente que usemos una u otra). La sobrecarga sin argumentos devuelve el valor predeterminado de ese tipo. Sin embargo, en la que recibe un argumento, lo que devuelve es el valor que indiquemos como tal. Esto nos permite acceder a los valores de los tipos anulables sin necesidad de hacer una comprobación previa para saber si tiene o no un valor nulo, y establecer valores predeterminados específicos en los casos particulares que necesitemos.

En la siguiente asignación, si la variable **i2** tiene un valor nulo, devolverá el valor indicado como argumento del método **GetValueOrDefault**; en caso de que dicha variable contenga un valor que no sea nulo, lo que devolverá será ese valor. Como vemos, el tipo de la variable que recibe el valor se infiere. En este caso, la variable **i** será de tipo **Integer** (porque el valor devuelto por ese método no es anulado, sino del tipo base, y en el ejemplo, la variable **i2** es de tipo **Integer?**).

```
Dim i = i2.GetValueOrDefault(125)
```

### Versiones

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

## Ensamblados amigos

Los ensamblados amigos son aquellos ensamblados (DLL o EXE) que permiten que podamos acceder a los elementos que definen, incluso aunque tengan ámbito **Friend**.

Cuando definimos una clase (o tipo) o los miembros de un tipo público usando el modificador de acceso **Friend**, esos elementos solo serán accesibles desde el propio ensamblado en el que se defi-

nen, por tanto, a esos elementos “amigos” no los podemos acceder desde otro ensamblado de forma externa.

La única forma de acceder a los elementos declarados con la instrucción **Friend** es si lo indicamos expresamente.

En Visual Basic 8.0/2005 podíamos indicar que otros ensamblados externos accedieran a los elementos **Friend** de nuestro ensamblado. Lo que no podíamos hacer es acceder a esos ensamblados amigos.

En Visual Basic 9.0/2008 esto ha cambiado, y ahora podemos crear ensamblados amigos y también acceder desde nuestro código a ensamblados amigos, incluso creados con otros lenguajes.

## Requisitos para acceder a los ensamblados amigos

Para que otros ensamblados puedan acceder a los elementos **Friend** de nuestro ensamblado deben cumplir algunos requisitos, que pueden variar dependiendo de que el ensamblado que define los elementos **Friend** esté firmado o no con nombre seguro (*strong name*).

Para que comprendamos mejor cómo utilizar los ensamblados amigos, empecemos viendo el código que usaremos en los dos tipos de ensamblados amigos que podemos definir. El código mostrado en el listado 2.22 lo vamos a usar en dos ensamblados, que serán del tipo biblioteca de clases. El primero de esos dos ensamblados no estará firmado con nombre seguro y tendrá el nombre **EnsambladoAmigo\_noStrongName**, el segundo estará firmado con nombre seguro y le daremos el nombre **EnsambladoAmigo** (los proyectos de ejemplo de estos ensamblados y la de los clientes que los utilicen, lo puede descargar desde el sitio de descargas del libro).

La primera clase está definida con un ámbito **Friend**, por tanto, solo la podremos acceder desde el mismo ensamblado en el que está definido o los que indiquemos como amigos.

La segunda clase es pública, por tanto, siempre está accesible; sin embargo, el método **SaludoAmigo2** de esa clase, al estar definido como **Friend**, tendrá las mismas restricciones que cualquier otro elemento que utilice ese modificador de ámbito.

## Requisitos para ensamblados firmados con nombre seguro

Si queremos acceder a un ensamblado, que define elementos **Friend** y que está firmado con nombre seguro, todas las aplicaciones que quieran usar dicho ensamblado también deben estar firmadas con nombre seguro, y además debemos conocer la clave pública de esa firma.

Esa información la tenemos que indicar en el ensamblado que expone los elementos **Friend**, porque si no indicamos los ensamblados a los que se autoriza el acceso, todos los elementos amigos no estarán visibles y por tanto no accesibles.

```

' Clase Friend solo accesible desde el mismo ensamblado
' o los ensamblados marcados como amigos
Friend Class ClaseAmiga
    ' Aunque los miembros sean públicos,
    ' el acceso está restringido por el ámbito del tipo que lo define
    Public Function SaludoAmigo() As String
        Return "Saludos desde la clase amiga"
    End Function

    Friend Function SaludoAmigo(ByVal nombre As String) As String
        Return "Saludos desde la clase amiga a " & nombre
    End Function

End Class

' Clase pública, siempre accesible
Public Class ClasePublica

    ' Los miembros públicos siempre están accesibles
    ' si la clase es pública
    Public Function SaludoPublico2() As String
        Return "Saludos desde un método público de una clase pública"
    End Function

    Friend Function SaludoAmigo2() As String
        Return "Saludos desde un método amigo en una clase pública"
    End Function

End Class

```

Listado 2.22. Las clases definidas en el ensamblado amigo

Para indicar el ensamblado o ensamblados a los que permitimos el acceso “amigo”, lo haremos por medio del atributo **InternalsVisibleTo**. En ese atributo (que podemos utilizarlo más de una vez) indicaremos el nombre del ensamblado y la clave pública. La forma de hacerlo será como vemos en el código del listado 2.23, que por brevedad, solo se muestran unos cuantos dígitos de la clave pública, ya que esta suele tener una longitud de 320 caracteres (160 *bytes* en formato hexadecimal de dos dígitos).

```
<Assembly: InternalsVisibleTo("ClienteEnsambladoAmigo, PublicKey=0024000...")>
```

Listado 2.23. En los ensamblados firmados con nombre seguro debemos indicar el nombre y la clave pública

Los dos ensamblados (o todos los que quieran ser amigos) deben estar firmados con nombre seguro (*strong name*), pero el nombre y la clave pública que indicamos en el atributo **InternalsVisibleTo** son del ensamblado que quiere usar el que define los elementos **Friend**.

Cada uno de los ensamblados, que quiera usar el nuestro, debe estar indicado en atributos como el mostrado en el listado 2.23, ya que solo los ensamblados que indiquemos ahí podrán acceder a los elementos definidos como **Friend**.



## Requisitos para los ensamblados que no están firmados con nombre seguro

Si queremos que otras aplicaciones utilicen en sus referencias, ensamblados que definan elementos **Friend**, que no están firmados con nombre seguro, en ese caso, esas aplicaciones no están obligadas a tener una firma con nombre seguro.

Pero también debemos indicarlas por medio del atributo **InternalsVisibleTo**, solo que en esta ocasión, tan solo debemos indicar el nombre del ensamblado, sin clave pública, entre otras cosas porque no existe esa clave pública. En el listado 2.24 vemos cómo indicar un ensamblado en el atributo **InternalsVisibleTo**.

```
' Solo podemos referenciar ensamblados sin firmar,  
' si este ensamblado no está firmado con nombre seguro.  
<Assembly: InternalsVisibleTo("ClienteEnsambladoAmigo_noStrongName")>
```

Listado 2.24. Si el ensamblado no está firmado con nombre seguro no es necesario indicar la clave pública

Para clarificar las cosas: el ensamblado que indicamos en el atributo **InternalsVisibleTo** es el que quiere acceder al ensamblado que define los elementos con el modificador **Friend**, y ese atributo tenemos que ponerlo en el ensamblado que autoriza que otros puedan ver los elementos declarados como amigos.

Si un ensamblado firmado con nombre seguro quiere acceder a esos elementos **Friend** del ensamblado que no está firmado, simplemente no podrá hacerlo. Aunque el compilador permitirá que lo incluyamos en el atributo **InternalsVisibleTo**, pero al compilar, nos dará un error indicándonos, que el ensamblado que tenemos referenciado, no está firmado con nombre seguro.

Por tanto, los requisitos para hacer amigables varios ensamblados son:

- Si el ensamblado con los miembros **Friend** no está firmado con nombre seguro, los ensamblados que quieran usarlo tampoco deben estar firmados.
- En ese ensamblado agregaremos tantos atributos **InternalsVisibleTo** como queramos, pero solo indicando el nombre del ensamblado al que autorizamos el acceso.
- Si uno está firmado con nombre seguro, todos deben estar firmados.
- Para permitir el acceso a los ensamblados firmados con nombre seguro debemos incluir el nombre del ensamblado y la clave pública en el atributo **InternalsVisibleTo**.

## Ejemplo de uso de los ensamblados amigos

Una vez que tenemos autorizados los ensamblados que pueden usar los elementos **Friend**, agregaremos una referencia a ese ensamblado y usaremos las clases como si éstas estuvieran definidas con el modificador **Public**, es decir, tendremos total acceso tanto a los tipos como a los métodos y otros elementos de los tipos que utilicen el modificador **Friend**. En el listado 2.25 vemos cómo usar las clases que vimos en el listado 2.22.

```
' Acceder al tipo Friend
' y a dos métodos Friend
Dim miAmigo As New ClaseAmiga
Console.WriteLine(miAmigo.SaludoAmigo)
Console.WriteLine(miAmigo.SaludoAmigo("Guille"))

' Acceso al tipo Public
Dim miAmigo2 As New ClasePublica
Console.WriteLine(miAmigo2.SaludoPublico2)

' Llamada a un método Friend
Console.WriteLine(miAmigo2.SaludoAmigo2)
```

Listado 2.25. Ejemplo de uso de los miembros Friend del ensamblado amigo

En el ZIP, con los ejemplos, tenemos varios proyectos según utilicen el ensamblado firmado con nombre seguro o el que no está firmado.

## ¿Cómo generar la clave pública usada en el atributo **InternalsVisibleTo**?

Como ya he comentado antes, si el ensamblado está firmado con nombre seguro, en el atributo **InternalsVisibleTo**, debemos indicar la clave pública del ensamblado al que autorizamos a usar nuestros elementos **Friend**. Esa clave pública la indicamos asignando una cadena a la propiedad **PublicKey** del mencionado atributo.

En un principio podríamos pensar que esa clave pública es la mostrada en el manifiesto del ensamblado o la que vemos al examinar el contenido de la carpeta **assembly** de Windows (la carpeta en la que se almacenan los ensamblados con nombre seguro o GAC –*Global Assembly Cache*–). En estos casos solo se muestra una clave pública conocida como *Public key token* cuya longitud es de 16 bytes.

En realidad, la clave pública sí que está en el manifiesto del ensamblado, concretamente en el atributo **.publickey** (generado por el lenguaje intermedio de Microsoft –MSIL–). Por tanto, podemos abrir el ensamblado con la utilidad **ildasm.exe** (*IL Disassembler*) y copiar ese valor.

En la figura 2.3 podemos ver el valor de la clave pública de un ensamblado. Esta ventana es la mostrada por la utilidad **ildasm.exe** al seleccionar el elemento **MANIFEST** de dicho ensamblado.

También podemos exportar la clave pública a un archivo de texto usando la utilidad **sn.exe** (*Strong Name Utility*). Esta exportación la haremos en dos pasos, primero creamos un archivo binario usando el parámetro **-p**. El archivo generado lo usaremos como argumento de la utilidad. En este caso indicaremos el parámetro **-tp** de la utilidad **sn.exe** para generar un archivo de texto en el que tendremos el contenido completo de la clave pública requerida por el atributo **InternalsVisibleTo**, además de la clave pública que podemos usar en otras utilidades del SDK de .NET Framework para hacer referencia a los ensamblados que estén firmados con esa clave.

Si el archivo de claves, que previamente hemos generado con el IDE de Visual Studio o con la utilidad **sn.exe**, se llama **misClaves.snk**, los pasos serán los mostrados en el listado 2.26 (en el primer paso también muestro cómo generar manualmente un archivo de claves). Este código lo podemos

ejecutar desde la línea de comandos usando el acceso directo a las herramientas de Visual Studio 2008 o bien escribiéndolo en un archivo del tipo **.bat** (o **.cmd**), pero siempre que tengamos acceso al *path* en el que se encuentra la utilidad **sn.exe**.

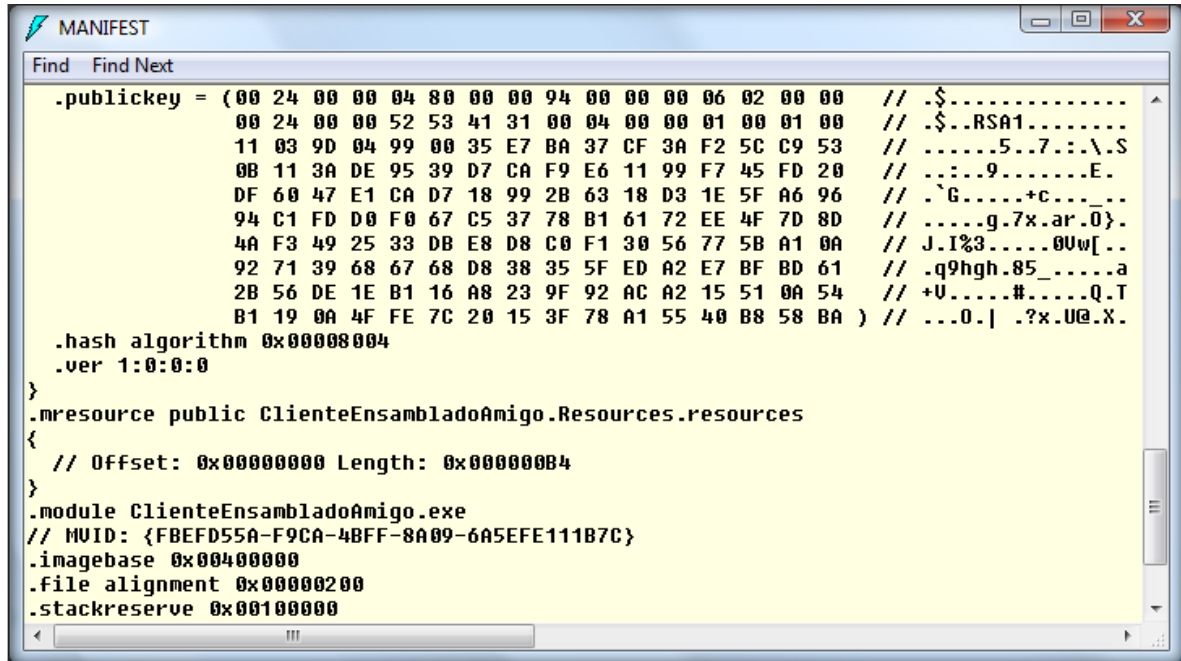


Figura 2.3. La clave pública de un ensamblado mostrada usando la utilidad ildasm.exe

```

Rem El archivo de claves lo generamos con:
sn -k misClaves.snk
Rem Creamos un archivo binario con la clave pública:
sn -p misClaves.snk misClaves.public
Rem Generamos un archivo de texto con la clave pública:
sn -tp misClaves.public > misClaves.txt

```

Listado 2.26. Pasos para generar un archivo de claves y la clave pública de ese archivo

Por último, aclarar que aunque los ensamblados estén firmados con nombres seguros y puedan estar alojados en el GAC, esto último no es un requisito para usarlos “amigablemente”, es decir, esos ensamblados pueden estar en el GAC, pero no es obligatorio; lo único importante es que todos ellos estén firmados con *strong name*.

### Nota

*Debido a que la clave pública se muestra dividida en varias líneas, tendremos que unir todas esas líneas (las de la clave pública) en una sola, antes de insertarla en el atributo `InternalsVisibleTo`.*

## Versiones

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

## Agilidad del runtime

Todos los programas que generemos con el compilador de Visual Basic siempre necesitan el *runtime* de Visual Basic para funcionar. Este motor en tiempo de ejecución es independiente del usado por el propio .NET y hasta esta versión de Visual Basic era necesario e imprescindible, es decir, quisiéramos o no, el *runtime* de Visual Basic siempre se ligaba con nuestros proyectos. En realidad, no se agregaba nada extra al código, salvo la dependencia con el ensamblado que contiene las definiciones de las funciones y otras características de Visual Basic, como puede ser toda la funcionalidad del objeto **My** o la posibilidad de usar módulos (tipo **Module**). Ese ensamblado (**Microsoft.VisualBasic.dll**) se instala siempre con todas las versiones de .NET, por tanto no nos obliga a distribuir ningún archivo extra (algo que sí ocurría en las versiones anteriores a .NET).

## Nota

*No confundir esta característica con la posibilidad de no agregar una importación al espacio de nombres `Microsoft.VisualBasic` para no usar las funciones del runtime, que es lo que podíamos hacer en las versiones anteriores para obligarnos a usar las clases de .NET en lugar de las funciones propias de Visual Basic, ya que aunque no usemos explícitamente esas funciones, el compilador las utiliza cuando necesita hacer alguna conversión de tipos u otras comprobaciones. Es decir, hasta la versión 9.0 siempre se añadía una referencia al ensamblado de Visual Basic aunque no quisiéramos usarlo.*

Pero en Visual Basic 9.0 ya no es una exigencia el uso de ese *runtime*. Por tanto, podemos prescindir del uso de ese ensamblado “especial” o usar uno diferente. Esto, en realidad, se ha hecho (al menos eso es lo que yo creo, ya que la documentación de Visual Studio es demasiado escueta en esta característica, y ni siquiera se muestra entre las novedades del lenguaje) para permitir que Visual Basic se pueda usar en la siguiente generación de compiladores (o intérpretes) conocidos como lenguajes dinámicos. De esta forma, en lugar de usar una DLL tan grande como la que actualmente existe, se podrá usar otra más pequeña y fácilmente distribuible. Pero ése seguramente será un tema en el que habrá que profundizar cuando aparezca la versión 10.0 de Visual Basic, mientras tanto, veamos cómo podemos usar esta funcionalidad, que solo está disponible desde la línea de comandos, es decir, no la podemos utilizar desde el IDE de Visual Studio 2008.

## Compilar código de Visual Basic sin usar el runtime de Visual Basic

Si queremos usar esta nueva posibilidad de compilar código de Visual Basic sin que utilice la biblioteca de tipos de Visual Basic, debemos hacerlo desde la línea de comandos y usar el nuevo modificador del compilador **/vbruntime**, el cual podemos usar de tres formas distintas:

- **/vbruntime+**. Es el valor predeterminado del compilador, es decir, el valor usado si no se indica esta opción de compilación. Esto incluirá el ensamblado predeterminado de Visual Basic (**Microsoft.VisualBasic.dll**).
- **/vbruntime-**. Esta opción le indicará al compilador que no utilice el ensamblado de Visual Basic. Debido a que hay ciertas características que se incluyen en esa biblioteca de tipos, no tendremos acceso a ninguna de ellas. Por ejemplo, no podremos usar las funciones propias de Visual Basic, definidas en el espacio de nombres **Microsoft.VisualBasic**.
- **/vbruntime:<nombre de la dll>**. Si tenemos otro ensamblado, que puede suplir algunas de las clases y características de Visual Basic, lo podemos indicar como argumento de este modificador del compilador.

## ¿Es posible crear una aplicación de Visual Basic sin usar el runtime de Visual Basic?

Después de todo lo comentado, evidentemente la respuesta es **sí**.

Pero recordemos que no podemos acceder a las funciones o constantes definidas en el espacio de nombres **Microsoft.VisualBasic** y tampoco podremos acceder al objeto **My** o usar el tipo **Module**, pero el resto de clases de .NET seguirán estando a nuestra disposición, así como las instrucciones y características del compilador que no dependan directa o indirectamente de todo lo que el *runtime* de Visual Basic ofrece.

Entre las funciones que no podremos usar están todas las de conversión entre tipos diferentes. Esto incluye incluso las concatenaciones a partir de una variable numérica, ya que el compilador de Visual Basic hace una llamada a las funciones necesarias para que ese código sea posible, particularmente porque, antes de realizar la concatenación, convierte el número en una cadena. En este caso, la solución es fácil, ya que podemos usar el método **ToString** para convertir el número en una cadena. Lo que no podremos hacer es usar la función **CStr** ni ninguna otra de las funciones de conversión. Si necesitamos convertir, por ejemplo una cadena en un número (u otro tipo intrínseco), tendremos que usar las funciones de conversión definidas en el tipo que queremos convertir o bien usar los métodos de la clase **Convert**. Incluso operaciones aparentemente simples, como es la comparación de dos cadenas, necesita del *runtime* de Visual Basic.

Independientemente de estas restricciones, la inferencia de tipos y otras características que no dependan directamente de las clases definidas en el espacio de nombres **Microsoft.VisualBasic** sí que las podremos seguir usando.

En el listado 2.27 vemos algunos ejemplos de qué podemos hacer si no tenemos a mano las funciones propias de Visual Basic, además de ver que la inferencia de tipos o los tipos anulables sí que podemos usarlos aunque usemos el modificador **/vbruntime-**.

```
Dim i1? As Integer = 22
Dim i2 = i1.GetValueOrDefault(999)

'Dim s = CStr(i2)
Dim s = i2.ToString
Dim n1 = "12"

'Dim v1 = CInt(n1)

Dim v1 = Integer.Parse(n1)
v1 = Convert.ToInt32(n1)
```

Listado 2.27. Si no tenemos acceso al runtime de Visual Basic algunas funciones no las podemos usar

Como vemos, en algunos casos será un poco complicado generar un ensamblado que no use la biblioteca de tipos de Visual Basic, particularmente si usamos el IDE de Visual Basic para escribir el código, ya que en el entorno de desarrollo no podemos avisar al compilador de que no queremos usar esa biblioteca, y hay muchas cosas que hace el compilador para facilitarnos la escritura del código, y algunas de esas cosas no las descubrimos hasta que dejamos de usarlo.

Afortunadamente (salvo que queramos hacer algo muy concreto), normalmente no nos estorbará esa biblioteca, por tanto, dudo que usemos este modificador del compilador como algo habitual. Pero como no está demás conocer su existencia, al menos, ya sabemos cómo usarlo.

### **Nota**

*Hay ciertas instrucciones que podemos usar, como es el caso de CType, etc., pero en caso concreto de CType, solo podremos usarlo para hacer conversiones de tipos definidos por el usuario o para clases que no exista una función equivalente en el lenguaje, por ejemplo, CInt, ya que si el compilador se encuentra con esta conversión: CType(n1, Integer) la sustituirá por CInt(n1) y como hemos visto en el listado 2.27 no es posible hacer esa conversión.*

## **Crear nuestro propio runtime de Visual Basic**

Como ya he comentado, la documentación de Visual Studio 2008 no ofrece mucha información sobre esta característica y aún menos de cómo podemos crear nuestra propia versión del *runtime* de Visual Basic, pero mirando los mensajes de error mostrados al intentar utilizar ciertas funciones, nos encontramos con pistas que nos pueden ayudar a hacer un ensamblado que sustituya el incluido con .NET.

Por ejemplo, si queremos convertir una cadena a un valor **Integer**, el mensaje de error mostrado es:

*Error BC35000: la operación solicitada no está disponible porque no está definida la función de biblioteca en tiempo de ejecución 'Microsoft.VisualBasic.CompilerServices.Conversions.ToInteger'.*

Es decir, para convertir un valor **String** a **Integer** el compilador necesita de la definición de una función llamada **ToInteger** que a su vez esté definida en la clase **Conversions** del espacio de nombres **Microsoft.VisualBasic.CompilerServices**.

De la misma forma, si queremos convertir de un tipo entero a una cadena, necesitamos una función **ToString** que reciba un valor **Integer** y devuelva **String**.

En el caso de las comparaciones de cadenas de caracteres, el compilador intenta buscar una función llamada **CompareString** que debe estar definida en la clase **Operators** del mismo espacio de nombres.

### **Nota**

*Las comparaciones con el resto de tipos implícitos se pueden realizar sin necesidad de definir una función específica; solo las comparaciones con tipos String son las que el compilador necesita que exista una función concreta que utilizar.*

Vamos a comprobar si todo esto es cierto. En el listado 2.28 vemos el código que podríamos escribir en Visual Basic para crear un ensamblado que sustituya (al menos en parte) al de Visual Basic.

Si el código del listado 2.28 lo tenemos en un archivo llamado **gsVBRuntime.vb**, lo podemos compilar desde la línea de comandos (sin que utilice el *runtime* de Visual Basic) usando el compilador de Visual Basic tal como vemos en el listado 2.29.

Para usar este nuevo *runtime* de Visual Basic podemos escribir en un archivo llamado **PruebasVBRuntime.vb**, un código como el del listado 2.30, el cual compilaremos usando la orden de la línea de comandos del listado 2.31.

Si queremos dar soporte al resto de tipos de datos integrados (los que se definen en el lenguaje a partir de los tipos de .NET Framework), tendremos que crear nuestras propias funciones de conversión, que tal como vemos en el listado 2.28, serán las que convierten esos tipos en **String** y viceversa, ya que, incluso sin una biblioteca en tiempo de ejecución (*runtime*), el compilador es capaz de hacer las conversiones implícitas (o de ampliación) entre los tipos numéricos, además de realizar las conversiones explícitas, las que necesitan de las funciones como **CInt**, **Cdbl**, etc.

```

Option Strict On
Imports System

Namespace Microsoft.VisualBasic.CompilerServices
    Public Class Conversions
        Public Shared
            Function ToInteger(ByVal value As String) As Integer
                Return Convert.ToInt32(value)
            End Function

            Public Overloads Shared
                Function ToString(ByVal value As Integer) As String
                    Return value.ToString
                End Function
            End Class
        End Class

        Public Class Operators
            Public Shared
                Function CompareString(ByVal str1 As String,
                                      ByVal str2 As String,
                                      ByVal textCompare As Boolean
                                      ) As Integer
                    Return String.Compare(str1, str2, textCompare)
                End Function
            End Class
        End Class
    End Namespace

```

Listado 2.28. Definición de funciones para un runtime de VB personalizado

```
vbc.exe /vbruntime- /out:gsVBRuntime.dll /t:library gsVBRuntime.vb
```

Listado 2.29. Compilar nuestra DLL para usar como runtime de Visual Basic

```

Option Strict On
Option Infer On
Imports System
Public Class PruebasVBRuntime
    Public Shared Sub Main()
        Dim i1 = 15
        Dim s = CStr(i1)
        Dim s1 = "12"
        i1 = CInt(s1)

        Console.WriteLine("i1= {0}, s = {1}", i1, s)
        If s = s1 Then
            Console.WriteLine("Son iguales")
        Else
            Console.WriteLine("Distintos '{0}' y '{1}'", s, s1)
        End If
        Console.ReadLine()
    End Sub
End Class

```

Listado 2.30. Código que utiliza nuestra versión del runtime

```
vbc.exe /vbruntime:gsVBRuntime.dll /t:exe PruebasVBRuntime.vb
```

Listado 2.31. Orden de la línea de comandos para compilar con nuestro runtime de VB



## Habilitar la comprobación de errores

Otra de las características mínimas que también debe tener nuestra versión personalizada del *runtime*, es la de poder detectar errores, ya que si no le damos soporte, no podremos usar las instrucciones **Try/Catch/Finally**. En realidad, sin hacer nada especial, compilará bien, pero al ejecutar el programa, recibiremos un error indicándonos que algo va mal, pero no por el propio error, si no porque se necesitan hacer ciertas comprobaciones. En esta ocasión el error mostrado no es demasiado explícito, al menos en lo que nosotros tenemos que hacer, ya que ese error será parecido a esto:

Excepción no controlada: System.BadImageFormatException: símbolo (token) de método incorrecto.

Para dar soporte al tratamiento de error, debemos definir los métodos ***ClearProjectError*** y ***SetProjectError*** en una clase llamada ***ProjectData*** (que también debe estar definida en el espacio de nombres ***Microsoft.VisualBasic.CompilerServices***). La definición más simple de esa clase es la mostrada en el listado 2.32.

```
Public Class ProjectData
    Public Shared Sub ClearProjectError()
        Try
        Finally
        End Try
    End Sub

    Public Shared Sub SetProjectError(ByVal ex As Exception)
        Try
        Finally
        End Try
    End Sub
End Class
```

Listado 2.32. Definición de los métodos para dar soporte al tratamiento de errores

En cualquier caso, para saber todo lo que realmente debemos implementar en nuestra versión personalizada del sustituto del *runtime* de Visual Basic, habrá que esperar a que Microsoft haga público algún documento en el que se den las instrucciones que debemos seguir para crearlo. Mientras eso ocurre, espero que todo lo comentado aquí (además del ejemplo más completo publicado en el sitio del libro) le sirva para crear su propia versión personalizada de la biblioteca de apoyo que necesita el compilador de Visual Basic para hacer su trabajo.

### Versiones

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

# Capítulo 3

## Características generales del lenguaje (II)

---

### Introducción

En este segundo capítulo sobre las novedades de Visual Basic 9.0 veremos los siguientes temas:

- Relajación de delegados.
- Operador ternario.

### Relajación de delegados

La relajación de delegados o las conversiones relajadas de delegados es una nueva característica de Visual Basic 9.0 que nos permite usar los delegados de una forma menos estricta a la que seguramente ya estaremos acostumbrados.

### Contravarianza: usar parámetros diferentes a los definidos en los delegados

Los delegados definen la firma que tienen las funciones (o procedimientos) a los que queremos acceder desde un objeto diferente al que implementa dicho procedimiento. Por ejemplo, cuando usamos un método para interceptar un evento, dicho método debe tener la misma firma que el delegado asociado a ese evento (todos los eventos siempre tienen asociado un delegado, incluso en Visual Basic, aunque el compilador nos lo oculte, esos delegados siempre están ahí).

Cuando queremos detectar las pulsaciones de las teclas que se han producido en un cuadro de texto (**TextBox**) usaremos el evento **KeyPress**. Este evento tiene asociado un delegado que es el que define cómo deben ser los métodos que quieran interceptarlo. El delegado en cuestión es **KeyPressEventHandler**. La definición de este delegado es como la mostrada en el listado 3.1. Y tal como podemos comprobar, define un método de tipo **Sub** (un procedimiento que no devuelve un valor) que recibe dos parámetros, el primero de tipo **Object** y el segundo de tipo **KeyPressEventArgs**. Si nos fijamos en la definición del método que intercepta el evento **KeyPress**, y que el propio diseñador de formularios añade (ver listado 3.2), los parámetros usados en ese método son exactamente los mismos que los definidos en el delegado.

En las versiones anteriores de Visual Basic la única forma de definir ese método es tal como lo vemos en el listado 3.2. Es decir, los dos parámetros deben ser exactamente del tipo que define el delegado, y lo mismo ocurre con el tipo de procedimiento, que, en este ejemplo, debe ser de tipo **Sub**.

```
Delegate Sub KeyPressEventHandler (ByVal sender As Object, _  
                                   ByVal e As KeyEventArgs)
```

Listado 3.1. Definición del delegado del evento KeyPress

```
Private Sub TextBox1_KeyPress (ByVal sender As Object, _  
                               ByVal e As KeyEventArgs) _  
    Handles TextBox1.KeyPress  
End Sub
```

Listado 3.2. Definición del método que intercepta el evento KeyPress

Sin embargo, en Visual Basic 9.0 podemos usar esa “relajación” de delegados, que también se conoce como *contravarianza*, en el sentido de que podemos usar en los parámetros cualquier tipo de datos que esté en la jerarquía de clases de los tipos definidos en el delegado. Es decir, podemos usar cualquier tipo de datos de los que se derive dicho tipo. Por ejemplo, la clase **KeyEventArgs** se deriva de **EventArgs** y ésta a su vez se deriva de **Object**, por tanto, en el segundo parámetro del método que intercepta ese evento podemos usar cualquiera de esas tres clases. Sabiendo esto, las definiciones de ese método que muestro en el listado 3.3 son totalmente válidas en Visual Basic 9.0 (en las versiones anteriores no se podrían usar).

```
Private Sub TextBox1_KeyPress (ByVal sender As Object, _  
                               ByVal e As EventArgs) _  
    Handles TextBox1.KeyPress  
End Sub  
  
Private Sub TextBox1_KeyPress (ByVal sender As Object, _  
                               ByVal e As Object) _  
    Handles TextBox1.KeyPress  
End Sub
```

Listado 3.3. Otras dos formas de definir el método que intercepta el evento Keypress

Si el lector ha usado C#, se habrá dado cuenta de que esto mismo ya era posible hacerlo en la versión anterior (la que se distribuyó con Visual Studio 2005).

Pero Visual Basic 9.0 va aún más lejos en la “relajación”, tanto que hasta permite que no indiquemos ningún parámetro. Por tanto, ese mismo método lo podemos definir tal como vemos en el código del listado 3.4.

```
Private Sub TextBox1_KeyPress () Handles TextBox1.KeyPress  
End Sub
```

Listado 3.4. La relajación de delegados nos permite incluso no usar parámetros

En todos estos ejemplos, particularmente en los dos últimos listados, solo tiene sentido cambiar los tipos de los parámetros si no queremos acceder a los miembros que definen los tipos “concretos”. Por ejemplo, si en ese evento no queremos tener en cuenta los datos pasados al método, podemos usar el código del listado 3.4. Si queremos saber qué tecla se ha pulsado o queremos indicar que esa pulsación ya la hemos “manejado” (asignando un valor verdadero a la propiedad **Handled** del segundo parámetro), lo indicado sería usar la clase concreta, la del código mostrado en el listado 3.2.

La verdad es que hay muchos métodos de evento en los que, en realidad, no usamos ninguno de los dos parámetros o en los que el segundo parámetro no nos sirve de mucho, como es el evento **Click** de cualquier control o todos aquellos eventos en los que el tipo del segundo parámetro es **EventArgs**, en todos esos casos podemos prescindir de esos parámetros y simplificar el código al máximo, tal como vimos en el listado 3.4.

### ¿La relajación de delegados solo para los eventos?

La primera impresión que nos puede dar es que ese será el uso de esta nueva característica de Visual Basic, pero esta relajación en los delegados va aún más allá. Y aunque ahora no veamos todas las posibilidades (cuando lleguemos al siguiente capítulo veremos otras formas de usar esta relajación de las conversiones en los delegados), sí que veremos otros detalles sobre este mismo tema aunque en contextos más clásicos, con idea de que sepamos mejor qué podemos hacer al trabajar con los delegados.

## Varianza: usar valores devueltos diferentes a los definidos en los delegados

Otra característica que podemos aprovechar de esta relajación en las conversiones de los delegados, es cambiar el tipo de procedimiento que en un principio debemos usar. Por ejemplo, si un delegado se define como una función que devuelve un valor, podemos usarlo sin tener en cuenta el valor devuelto, es decir, usarlo como si fuese un procedimiento de tipo **Sub**.

Además de esta posibilidad de ignorar totalmente el valor devuelto, también podemos usar lo que se conoce como *varianza* y que consiste en que si un delegado devuelve un valor de un tipo, ese valor lo podemos asignar en cualquier delegado que admita ese mismo tipo o cualquier otro que se derive de él.

Para ver esto último vamos a definir una clase llamada **Cliente** y otra llamada **Cliente2** que se deriva de **Cliente**. Además de estas dos clases, vamos a definir un delegado que devuelve un valor del tipo **Cliente** (la clase base). También definiremos un método que cumple con los requisitos de ese delegado, por tanto, podremos usar ese método para devolver un objeto del tipo **Cliente**. En los listados 3.5 y 3.6 vemos parte de ese código.

Las clases mostradas en el listado 3.5 no están completas solo por simplificar, ya que aquí lo que interesa es ver dónde podemos asignar el valor devuelto por el delegado (concretamente por el método **unCliente** definido en el código del listado 3.6).

```

Delegate Function ClienteCallback(ByVal nombre As String) As Cliente

Class Cliente
    ' Define las propiedades Nombre, Apellidos y Correo
End Class

Class Cliente2
    Inherits Cliente
    ' Define la propiedad NIF (Número de identificación)
End Class

```

Listado 3.5. Definición simplificada de las clases Cliente y Cliente2 y del delegado

```

Private Function unCliente(ByVal nombre As String) As Cliente
    Return New Cliente With {.Nombre = nombre}
End Function

```

Listado 3.6. Un método con la misma firma del delegado definido en el listado anterior

Ese método lo podemos usar a través de una variable del tipo del delegado a la que le asignamos la dirección del método que queremos usar (y que tiene la misma firma del delegado) por medio de **AddressOf** (que es la forma de acceder a los métodos en Visual Basic, vamos que no estamos descubriendo nada nuevo, al menos por ahora). En el código del listado 3.7 vemos la forma habitual de usar este delegado, es decir, creamos una variable del mismo tipo del delegado, le asignamos la dirección de memoria del método y usamos el valor devuelto para asignar una variable del tipo adecuado, en este caso del tipo *Cliente*.

```

Dim cliCallback1 As ClienteCallback = AddressOf unCliente
Dim cli1 As Cliente = cliCallback1("Pepe")

```

Listado 3.7. Uso de un método con la misma firma que el delegado

Hasta aquí todo como era antes de la llegada de Visual Basic 9.0. Salvo por la forma de crear el objeto del tipo *Cliente* en el método *unCliente* que aprovecha la nueva característica de inicialización de objetos.

Con Visual Basic 9.0, también podemos definir otro método que devuelva un objeto del tipo *Cliente2* (listado 3.8), en este caso, la firma de este método no cumple estrictamente lo impuesto por el delegado *ClienteCallback*, pero gracias a la relajación en las conversiones de delegados, podemos asignar ese método a una variable del tipo *ClienteCallback*, tal como vemos en el listado 3.9.

Como vemos en el listado 3.9, la variable *cliCallback2* es del tipo *ClienteCallback*, que en teoría solo debería aceptar direcciones de funciones con la misma firma que la de ese delegado, sin embargo, gracias a la *varianza*, también acepta un método que devuelve un objeto del tipo *Cliente2*, ya que ese tipo se deriva de la clase devuelta por el delegado.

```
Private Function unCliente2(ByVal nombre As String) As Cliente2
    Return New Cliente2 With {.Nombre = nombre}
End Function
```

Listado 3.8. Definición de un método que no cumple exactamente con la firma del delegado ClienteCallback

```
Dim cliCallback2 As ClienteCallback = AddressOf unCliente2
```

Listado 3.9. Asignamos a una variable de un delegado un método que no tiene la firma exacta de dicho delegado

En cualquier caso, aunque la “relajación de delegados” permita el uso de métodos con firmas distintas (siempre que el valor devuelto se derive del tipo devuelto por el delegado), esta relajación no llegará al extremo de permitir que la asignación se haga a un tipo inadecuado. Por ejemplo, el delegado de la variable *cliCallback2* (listado 3.9) no podríamos usarlo para asignar a una variable del tipo *Cliente2*, salvo que tengamos desactivado **Option Strict** o bien hagamos una conversión al tipo de datos de la variable. En el listado 3.10 vemos las dos posibilidades, pero en el caso de la primera, solo funcionará con **Option Strict Off**, la segunda siempre funcionará, independientemente del estado de la comprobación estricta del código.

```
' Esto dará error, se relajan los delegados, pero no tanto
' (salvo que tengamos Option Strict Off)
Dim cli2 As Cliente2 = cliCallback2("Luis")

' Habría que hacerlo con una conversión de tipos
Dim cli2 As Cliente2 = DirectCast(cliCallback2("Luis"), Cliente2)
```

Listado 3.10. Dos formas de usar un delegado, la primera solo funcionará con Option Strict Off

## Los delegados y las conversiones implícitas (con Option Strict On)

Además de la *varianza*, en Visual Basic también podemos usar los delegados con tipos diferentes a los indicados en la firma, siempre que se pueda hacer una conversión implícita, es decir, de ampliación. Por ejemplo, si un delegado devuelve un tipo **Integer**, lo podemos usar con otros tipos que se puedan “ampliar” a ese tipo, por ejemplo, un tipo **Short** o cualquier otro que el compilador de Visual Basic “sepa” convertir sin pérdida de información.

En el código del listado 3.11 tenemos la definición de un delegado que recibe un parámetro de tipo **Integer** y devuelve un valor también de ese tipo. En el listado 3.12 definimos un método con esa misma firma y otro que devuelve un valor de tipo **Short**. Finalmente en el listado 3.13 tenemos un ejemplo de cómo usar ese delegado para acceder a los dos métodos. Esto es parecido a lo que vimos en la sección anterior, solo que en este caso no existe ninguna relación de “herencia” entre los tipos **Short** e **Integer**, pero el compilador sabe que un valor **Short** se puede convertir a **Integer** sin pérdida de información, por tanto, nos permite usar el método *incrementaShort* aunque no tenga una firma igual a la definida en el delegado.

```
Delegate Function EnterosCallback(ByVal num As Integer) As Integer
```

Listado 3.11. Un delegado para tipos Integer

```
Private Function incrementaInteger(ByVal num As Integer) As Integer
    Return num + 1
End Function

Private Function incrementaShort(ByVal num As Integer) As Short
    Return CShort(num + 1)
End Function
```

Listado 3.12. Dos métodos válidos para el delegado del listado 3.11

```
Dim d1 As EnterosCallback = AddressOf incrementaInteger
Dim d2 As EnterosCallback = AddressOf incrementaShort
```

Listado 3.13. Uso del delegado con dos métodos de firmas diferentes

Las variables definidas en el listado 3.13 las podemos usar para uno u otro método, pero debemos darnos cuenta de que el delegado que se usa devuelve un valor de tipo **Integer** y el parámetro que espera también es de ese tipo, por tanto, debemos suponer que independientemente del método que usemos, los parámetros y el tipo de dato devuelto debe coincidir con el delegado. Veamos el listado 3.14 y comprobemos si esto debe ser así (en el código del listado 3.14 no he dejado que los tipos de las variables se infieran, ya que siempre lo harán a **Integer**, y he preferido dejar las declaraciones completas para que se vea claramente que Visual Basic permite “la relajación”, pero... sin pasarse).

```
' Varios tipos para los parámetros
Dim i1 As Integer = 10
Dim b1 As Byte = 126
Dim s1 As Short = 222

' Usamos el delegado con el método "adecuado"
Dim n1 As Integer = d1(i1)
n1 = d1(b1)
n1 = d1(s1)

' Usamos el delegado con el método de un tipo "convertible"
Dim n2 As Integer = d2(i1)
n2 = d2(b1)
n2 = d2(s1)
```

Listado 3.14. Varios ejemplos para usar el delegado y los métodos

Como vemos en el código del listado 3.14, todo lo comentado no solo es aplicable al tipo devuelto, sino también a los parámetros definidos en los delegados, ya que el compilador sabe cómo convertir de un tipo **Byte** o **Short** a **Integer**. El que los valores devueltos siempre sean de tipo **Integer** es porque el compilador no confía en una conversión que pueda suponer pérdida de información, por tanto, no dejará que asignemos esos valores a variables con menos capacidad que un **Integer**, pero si la variable que recibe el valor es de un tipo que no produzca pérdida de información, si que permitirá que se asignen a esa variable, como puede ser el caso de una variable de tipo **Long** (o **Double**), tal como vemos en el código del listado 3.15.

```
Dim n3 As Long = d1(i1)
n3 = d2(s1)

Dim n4 As Double = d1(i1)
n4 = d2(s1)
```

Listado 3.15. Visual Basic sabe cómo convertir de Integer a un tipo con más capacidad

## Los delegados y las conversiones con Option Strict Off

Todo lo que hemos visto en el apartado anterior lo podemos hacer sin recibir advertencias del compilador, es decir, que son conversiones aceptables y “controladas”. Pero ya sabemos que Visual Basic también permite la escritura de código sin tantas restricciones ni comprobaciones de que los tipos usados sean los adecuados.

Como ya he dejado patente en varias ocasiones, no soy partidario de usar **Option Strict Off**, pero vamos a ver cómo afecta la no comprobación estricta del código al usar la relajación de delegados, ya sea en el tipo devuelto como en los parámetros de los mismos.

Debido a que **Option Strict Off** deja al compilador libertad para hacer las conversiones en tiempo de ejecución, cuando usemos tipos de datos que no son los correctos, el compilador (o mejor dicho el *runtime*) de Visual Basic hará las conversiones que crea conveniente para que no se produzca una excepción por un uso inadecuado de los tipos de datos.

En el listado 3.16 vemos el código de dos métodos que podremos usar con el delegado del listado 3.11 aunque no tengan unas firmas “adecuadas”. Esos métodos los podemos usar tal como vemos en el código del listado 3.17, y aunque el parámetro esperado en ambos casos sea de tipo **Integer** (que es el tipo definido en el delegado), podemos usar un valor de tipo **Double**. Todo esto solo es posible si tenemos desactivado **Option Strict**.

```
Private Function incrementaLong(ByVal num As Long) As Long
    Return num + 1
End Function

Private Function incrementaDouble(ByVal num As Double) As Double
    Return num + 1
End Function
```

Listado 3.16. Dos métodos con firmas diferentes a la indicada por el delegado del listado 3.11

```
Dim d3 As EnterosCallback = AddressOf incrementaDouble
Dim n5 = d3(10.0)

Dim d4 As EnterosCallback = AddressOf incrementaLong
Dim n6 = d4(10.0)
```

Listado 3.17. Con Option Strict Off no es necesario que seamos estrictos con el uso de los delegados



Tal como vemos en el código del listado 3.17, las variables *n5* y *n6* no tienen definido un tipo, por tanto, el compilador de Visual Basic lo inferirá de la asignación usada (ya sabemos que la inferencia de tipos sigue funcionando aunque tengamos desactivada la comprobación estricta de tipos).

La pregunta es: ¿de qué tipo son esas variables?, ¿será la variable *n5* de tipo **Double**?, ¿será la variable *n6* de tipo **Long**?

La respuesta al final de esta sección. Pero inténtelo antes de ver la respuesta, seguramente parecerá obvio, pero es interesante saber qué cosas ocurren cuando **Option Strict** está desactivado.

Para usar un código similar al mostrado en los dos últimos listados, pero usando **Option Strict On**, tendríamos que hacer un par de cambios en las definiciones y en la forma de usarlos. En los listados 3.18 y 3.19 vemos esas modificaciones para que sean válidas con la comprobación estricta de tipos y conversiones.

```
Private Function incrementaLong(ByVal num As Long) As Integer
    Return CInt(num + 1)
End Function
Private Function incrementaDouble(ByVal num As Double) As Integer
    Return CInt(num + 1)
End Function
```

Listado 3.18. Definiciones de los métodos del listado 3.16 para usar con Option Strict On

```
Dim d3 As EnterosCallback = AddressOf incrementaDouble
Dim n5 = d3(CInt(10.0))
Dim d4 As EnterosCallback = AddressOf incrementaLong
Dim n6 = d4(CInt(10.0))
```

Listado 3.19. Con Option Strict On debemos adecuar los tipos a los que VB sabe convertir implícitamente

## Otros usos de la relajación de delegados con las expresiones lambda

Como ya comenté, hay otras nuevas características de Visual Basic 9.0 que se aprovecharán de la relajación de delegados, pero las dejaré para el próximo capítulo cuando veamos las expresiones *lambda*. Por ahora dejemos la intriga.

### **Versiones**

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

## Respuesta a la pregunta hecha después del listado 3.17

El comentario que puse sobre lo que **Option Strict Off** podía hacer, seguramente haría dudar a más de uno y decidir que **n5** era de tipo **Double** y **n6** de tipo **Long**, pero eso solo era para dar un poco de “morbo” al tema... el compilador, ni en modo **Option Strict Off**, hace las cosas tan mal.

Las variables **n5** y **n6** del listado 3.17 son de tipo **Integer**, ya que ese es el tipo de datos que devuelve el delegado usado para hacer la asignación, independientemente de que los métodos usados devuelvan otros tipos.

## Operador ternario

Debido a que en Visual Basic nunca hemos tenido un operador ternario, lo primero que habría que hacer es definir qué es un operador ternario.

Un operador ternario es un operador condicional, el cual, a partir de una expresión (que dará como resultado un valor verdadero o falso) usará una de las dos expresiones indicadas en ese operador, de ahí el nombre ternario.

Los lectores que hayan usado algún lenguaje que use la sintaxis de C (como puede ser C/C++, C#, Java o Javascript) seguramente habrán visto en más de una ocasión el operador **?:**, que es el operador ternario/condicional usado por esos lenguajes para los casos como el comentado en el párrafo anterior.

En Visual Basic lo más parecido al operador **?:** es la función **IIf**. Esta función está definida en el espacio de nombres **Microsoft.VisualBasic** y ya estaba presente en Visual Basic desde la versión 3.0. La funcionalidad de esta función es parecida a la del operador ternario de C++, el inconveniente que tiene (al menos para los que nos gusta usar **Option Strict On**) es que el valor devuelto por esa función siempre es de tipo **Object**. Da igual de qué tipos sean los valores a devolver cuando se cumpla o no la condición indicada, siempre se devuelve un valor de ese tipo. De todas formas, a mi particularmente nunca me ha gustado esa función, y ya desde las versiones anteriores a .NET procuraba no usarla en la medida de lo posible, en su lugar usaba una comparación, que es al fin y al cabo lo que se hace en esa función; ya sé que no es lo mismo trabajar con un **If/Then**, pero lo prefería por rendimiento y sobre todo porque desde antes de que llegara .NET, esa función me dio algún que otro quebradero de cabeza (errores). Pero dejemos “mis cosas” y volvamos a los problemas “innatos” de esa función de Visual Basic.

La función **IIf** devuelve siempre un tipo **Object**, por tanto, no es el valor que nos gustaría que devolviera si lo que queremos devolver es una cadena o un valor numérico. En estos casos concretos, siempre teníamos que hacer una conversión al tipo adecuado (o no activar **Option Strict**, pero esa tampoco es la solución). La solución idónea hubiera sido que esa función devolviera el tipo adecuado a los parámetros usados; una simple sobrecarga de la función lo hubiera solucionado, pero eso rompería la manida compatibilidad hacia atrás. Sin comentarios.

En Visual Basic 9.0 se ha resuelto ese problema, pero no creando las sobrecargas que muchos pedíamos para esa función, si no creando una nueva función que hace eso precisamente, es decir, devolver un tipo de datos que coincide con los argumentos indicados después de la expresión condicional. Esa función (u operador ternario de Visual Basic) es la función **If**.

Sí, **If** con solo una letra “i”, **If** como el **If** de **If/Then**, sí, el **If** de las comparaciones de Visual Basic.

Ahora, Visual Basic 9.0 (y las versiones que sigan) tiene un operador ternario camuflado en la instrucción **If**, pero usándola como función, ya que ese operador se usa tal como vemos en el código del listado 3.20, es decir, después de la instrucción **If** indicamos (dentro de paréntesis) la condición a evaluar y después los dos valores a devolver separados por comas. El primero (después de la condición) es el valor a devolver si la condición se cumple y el siguiente si no se cumple.

```
Dim s = If(True, "Verdadero", "Falso")
```

Listado 3.20. Uso simplificado del operador ternario de Visual Basic

Como vemos en el código del listado 3.20, también podemos usar la inferencia de tipos con el operador ternario, ya que el compilador sabe qué tipo de datos devolverá se cumpla o no la condición. En ese ejemplo, la variable que recibe el valor es de tipo **String**.

La condición que usaremos como primer argumento de esta función (no resisto la tentación de seguir llamándola función aunque la documentación de Visual Studio nos diga que es un operador) puede ser cualquier expresión que dé como resultado un valor de tipo **Boolean**. Sin embargo, los otros dos argumentos pueden ser expresiones de cualquier tipo, siempre que ambas sean del mismo tipo de datos o exista una conversión implícita entre ambos tipos de datos.

En el código del listado 3.21 vemos cómo usar dos valores de diferentes tipos de datos. En este ejemplo, el valor que tiene preferencia es **Long** ya que es de mayor capacidad, y debido a que **Integer** se puede “promocionar” a **Long** sin pérdida de datos, se puede usar sin problemas. También comprobamos que la expresión a evaluar puede ser de cualquier tipo, siempre que devuelva un valor de tipo **Boolean**.

```
Dim long1 As Long = 15
Dim nums() As Integer = {1, 2, 3, 4, 5, 6, 7}

' Si uno de los operandos es convertible al otro,
' se devuelve el tipo con mayor capacidad.
' En este ejemplo, el primero es Integer y el segundo es Long,
' por tanto se devuelve un Long
Dim n1 As Long = If(nums.Length > 4, nums.Length, long1)
```

Listado 3.21. Siempre se devuelve el tipo con mayor capacidad

### Nota

*Todo lo comentado aquí es partiendo de la base de que usamos **Option Strict On**, ya que si no tenemos activada la comprobación estricta, es posible usar este “operador” de forma inadecuada, aunque seguramente funcionará sin problemas, salvo en los casos que Visual Basic haga una conversión que después resulte en un error en tiempo de ejecución. Por tanto, insisto que **siempre** usemos **Option Strict On** para evitarnos sorpresas desagradables, ya que si no nos interesa hacer las cosas “bien”, podemos seguir usando la función **IIf** y no complicarnos con cosas nuevas.*

Afortunadamente el compilador de Visual Basic nos avisará cuando usemos de forma inadecuada el operador **If** (sí, al final me acostumbraré a decir “operador”), ya que si no es capaz de convertir de forma implícita los valores usados en los dos últimos argumentos (u operandos) para que los dos finalmente devuelvan un valor del mismo tipo, nos avisará con un error. Ese error lo veremos instantáneamente si usamos el IDE de Visual Basic 2008 gracias a la “pre compilación” que hace del código, tal como vemos en la figura 3.1, que a pesar de tener desactivado **Option Strict**, el compilador nos avisa de que los dos operandos deben ser del mismo tipo o al menos de tipos que se puedan convertir de forma implícita.

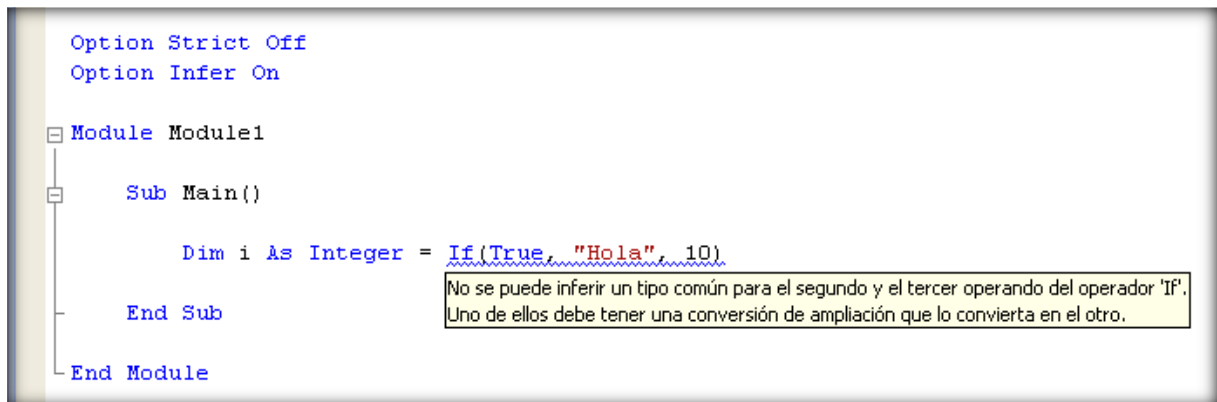


Figura 3.1. Los valores que devuelve el operador **If** deben ser de tipos convertibles

En el código del listado 3.22 usamos el operador **If** de forma inadecuada, ya que el valor devuelto debería ser de tipo **Double** (porque es el tipo con mayor capacidad de las dos expresiones a devolver), pero al estar desactivada la comprobación estricta, el compilador nos permite hacer esa asignación.

Lo correcto sería hacer la conversión del valor devuelto por el operador **If** a **Integer**. Tal como se muestra en el listado 3.23, esa conversión la tendremos que hacer si tenemos **Option Strict On**. Además, como sabemos, Visual Basic nos ayuda en estos casos en los que se necesita de una con-

versión y si lo usamos desde el IDE de Visual Studio 2008, nos ofrecerá la corrección que debemos hacer, tal como vemos en la figura 3.2.

```
Dim double1 As Double = 10.5
Dim nums() As Integer = {1, 2, 3, 4, 5, 6, 7}

' Esto funcionará solo con Option Strict Off,
' ya que el valor devuelto será de tipo Double
Dim n1 As Integer = If(nums.Length > 4, nums.Length, double1)
```

Listado 3.22. Option Strict Off no requiere que se hagan las cosas “adecuadamente”

```
' Si tenemos option Strict On habría que hacer una conversión
Dim n2 As Integer = CInt(If(nums.Length > 4, nums.Length, double1))
```

Listado 3.23. Option Strict On requiere que las cosas se “intenten” hacer bien

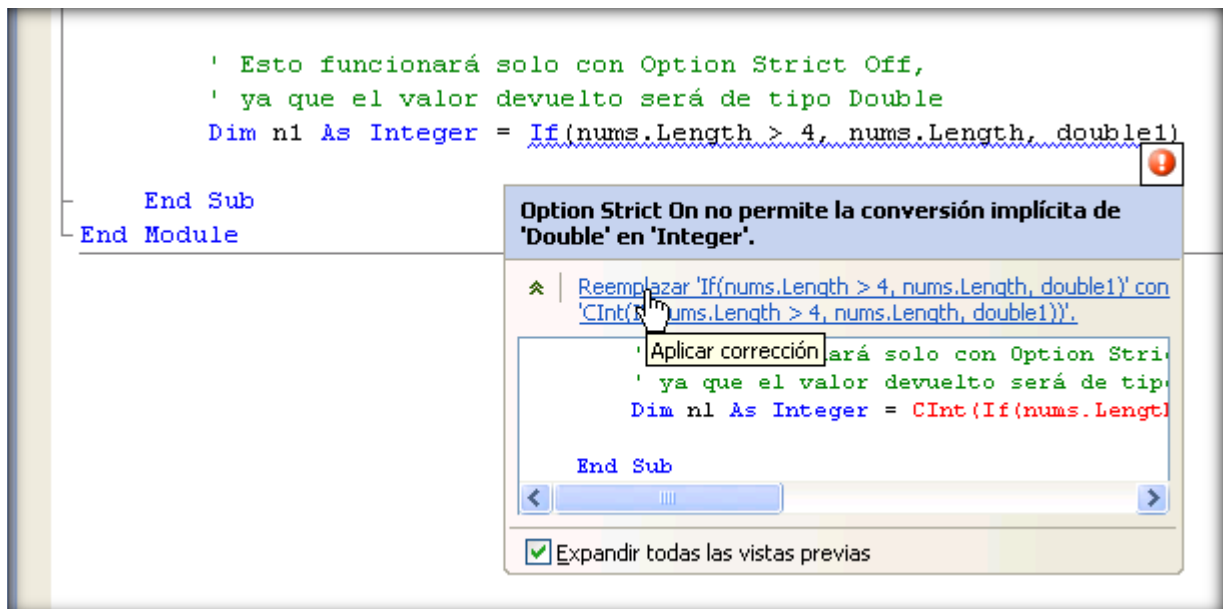


Figura 3.2. Corrección de VB en el tipo devuelto por el operador If

## El operador If evalúa solo la parte que debe evaluar

El operador **If** tiene otra ventaja sobre la función **IIf**, y es lo que se conoce como un operador *cortocircuitado*, es decir, cuando evalúa la condición y “sabe” qué valor debe devolver, solo evalúa ese argumento (u operando), dejando el otro de lado (la función **IIf** siempre evalúa los dos, independientemente del valor que devuelva y por tanto, utilice).

Esto es útil particularmente si los valores usados después de la condición son expresiones en lugar de valores fijos, ya que para devolver un valor que una expresión produce, hay que evaluarla. Esta última afirmación es obvia, pero la he querido dejar patente con idea de que sepamos que si no estuviera “cortocircuitado” el operador **If**, y se usaran dos expresiones, habría que evaluar las dos expresiones antes de devolver el valor que corresponda, por tanto, es evidente que al evaluar solo la que se va a devolver nos ahorramos algunos *ticks* de CPU, y con total seguridad evitamos efectos colaterales que no serían deseados.

Por ejemplo, en el listado 3.24 tenemos la definición de dos funciones que reciben un valor por referencia (por tanto, cualquier cambio que se haga en la función afectará a la variable usada como argumento al llamar a las funciones), una de las funciones incrementa el valor del parámetro y la otra le resta uno (también se define una enumeración que la usaremos para la expresión a evaluar, esto es solo para que veamos que la expresión a evaluar puede ser cualquier expresión que devuelva un valor verdadero o falso).

```
Enum Operaciones
    Inc
    Dec
End Enum

Function Inc(ByRef num As Integer) As Integer
    num += 1
    Return num
End Function

Function Dec(ByRef num As Integer) As Integer
    num -= 1
    Return num
End Function
```

**Listado 3.24.** Enumeración y funciones a usar en los siguientes listados

En el listado 3.25 usamos el operador **If** para devolver el valor de una u otra función, según se cumpla la condición indicada (en este ejemplo, se cumplirá, por tanto, se usará la llamada a la función **Inc**).

Como ejercicio, pensemos qué se mostrará al ejecutar ese código. La solución después de ver el siguiente ejemplo.

En el listado 3.26 en lugar de usar el operador **If**, usamos la función **IIf**; como vemos el código es igual al del listado 3.25, pero en este caso, además de usar una conversión a **Integer**, tendremos un efecto “colateral” debido a cómo se utiliza esta función, que como ya expliqué, siempre evalúa todas las expresiones antes de dar el resultado.

Ejercicio: ¿qué mostrará este código?, ¿de qué tipo es la variable *resultado*?

```
Dim operacion As Operaciones = Operaciones.Inc
Dim num As Integer = 22

Console.WriteLine("num antes= {0}", num)
Dim resultado = If(operacion = Operaciones.Inc, Inc(num), Dec(num))

Console.WriteLine("resultado= {0}", resultado)
Console.WriteLine("num después= {0}", num)
```

Listado 3.25. El operador If solo evalúa la expresión que debe usar para devolver el valor

```
Dim operacion As Operaciones = Operaciones.Inc
Dim num As Integer = 22

Console.WriteLine("num antes= {0}", num)
Dim resultado = IIf(operacion = Operaciones.Inc, Inc(num), Dec(num))

Console.WriteLine("resultado= {0}", resultado)
Console.WriteLine("num después= {0}", num)
```

Listado 3.26. La función IIf evalúa las dos expresiones antes de devolver el valor

## Soluciones explicadas de los ejercicios

Al ejecutar el código del listado 3.25, lo que nos mostrará es lo que vemos en la figura 3.3. Como vemos, solo se utiliza la función que corresponde según el valor de la expresión evaluada. En este ejemplo, se llama a la función *Inc*, por tanto, se incrementa también el valor de la variable usada como argumento, de ahí que tanto el valor de *resultado* como el de la variable *num* sean iguales.

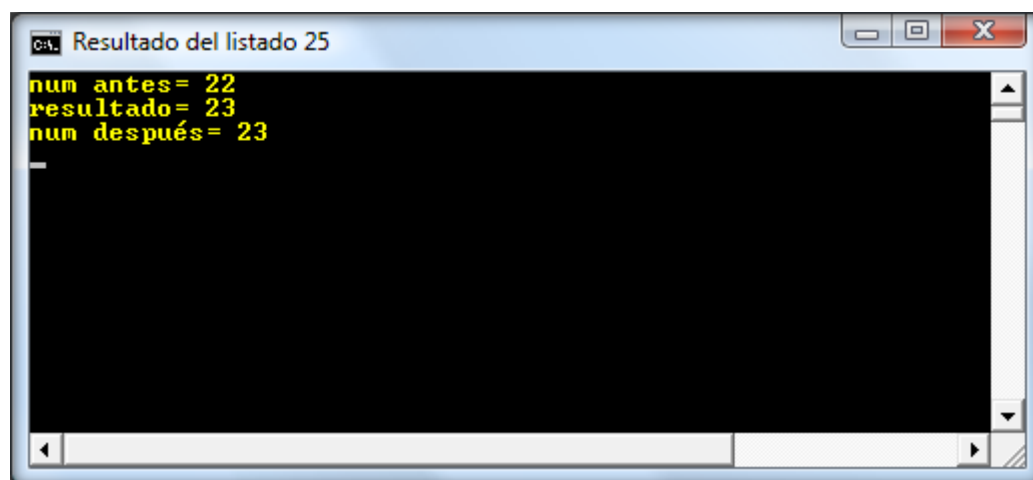


Figura 3.3. Resultado de ejecutar el código del listado 3.25

Sin embargo, al ejecutar el código del listado 3.26, (ver la figura 3.4), y debido a que la función **IIf** siempre evalúa todo el código indicado antes de devolver el valor, el contenido de la variable **num** no cambia, ya que primero se incrementa (al llamar a la función **Inc**) y después decrece (al llamar a la función **Dec**), sin embargo el valor de la variable **resultado** es el correcto (y el esperado).

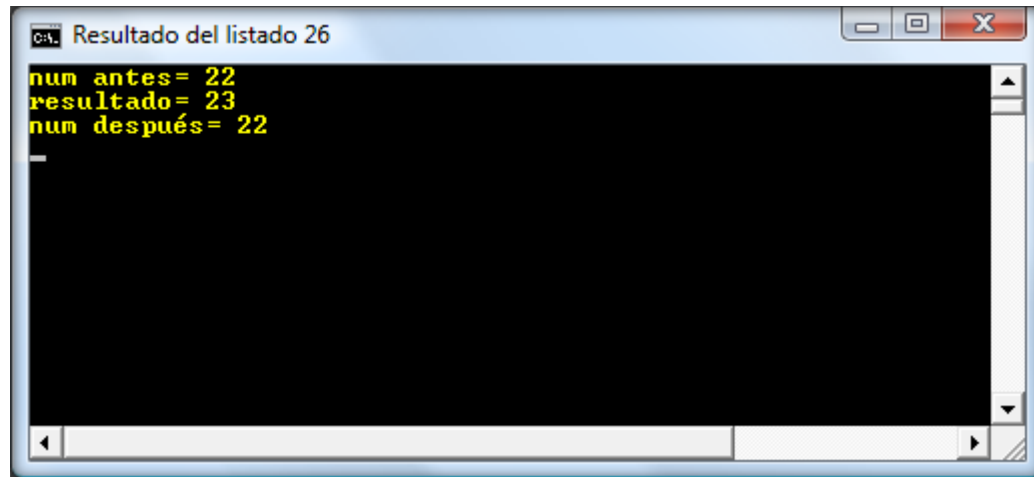


Figura 3.4. Resultado de ejecutar el código del listado 3.26

En cuanto a qué tipo de datos tendrá la variable **resultado**, el tipo inferido es **Object**, como podemos comprobar en la figura 3.5, pero en realidad, el tipo “interno” será **Integer** (o **Int32**).

```
Dim operacion As Operaciones = Operaciones.Inc
Dim num As Integer = 22

Console.WriteLine("num antes= {0}", num)
Dim resultado = IIf(operacion = Operaciones.Inc, Inc(num), Dec(num))
Dim resultado As Object
Console.WriteLine("resultado= {0}", resultado)
Console.WriteLine("num después= {0}", num)
```

Figura 3.5. La función IIf siempre devuelve Object

## ¿Cuándo y dónde podemos usar el operador If?

El operador **If** siempre lo usaremos para devolver un valor. Este valor lo podemos usar en una expresión en la que el tipo del valor sea adecuado, además de poder usarlo para asignar una variable, es decir, lo podremos usar siempre que creamos conveniente, pero teniendo en cuenta que el tipo que devuelve es el adecuado, y como hemos comprobado, el tipo de datos que devuelve este operador es el mismo que estamos usando en cualquiera de los dos últimos parámetros de esta “función”.



En el código del listado 3.27 usamos el operador **If** dos veces en la misma expresión, la segunda de esas dos veces es para formar una nueva expresión que se usará cuando el número de elementos del array *nums* no sea uno. En ese caso, se vuelve a evaluar la propiedad **Length** del array y se devolverá un texto dependiendo de que tenga cero o más elementos, después de evaluar cada una de las expresiones de los dos **If**, se usará una constante. Aunque un poco rebuscado, nos sirve para ver que este operador condicional lo podemos usar en cualquier situación en la que se requiera un valor del tipo que estamos usando, en todo ese ejemplo, ese valor es de tipo **String**, pero podría haber sido de cualquier otro.

```
Dim nums() As Integer = {1, 2, 3, 4, 5, 6, 7}
Dim res As String

res = "El array nums " & _
    If(nums.Length = 1, "tiene un elemento", _
        If(nums.Length = 0, _
            "no tiene", _
            "tiene " & nums.Length) & _
        " elementos") & _
    "."

Console.WriteLine(res)
```

Listado 3.27. El operador If lo podemos usar en cualquier expresión

Como ejercicio, añadir el código necesario (después de la definición del array *nums*) para que ese array solo tenga un elemento y para que no tenga ninguno, pero que no sea nulo (**Nothing**), ya que en ese caso, la expresión que asigna el valor a la variable *res* provocaría una excepción. Al final muestro algunas de las posibles soluciones.

### Solución al ejercicio

La eliminación de todos los elementos lo podemos hacer de tres formas, la primera es usando la instrucción **ReDim**, pero con el valor -1 como límite superior (recordemos que en Visual Basic la longitud de la dimensión es el límite superior más uno, por tanto, esta extraña forma de cambiar el tamaño del array es correcta, y como curiosidad, comentar que este “truco” está descrito en la sección de la ayuda para los cambios con respecto a cómo se hacía en Visual Basic 6.0), la segunda es asignando directamente un array vacío a la variable *nums*, la otra es usando el método **Resize** de la clase **Array** indicando que queremos que tenga cero elementos:

```
ReDim nums(0 To -1)

nums = New Integer() {}

Array.Resize(nums, 0)
```

Solución listado 3.27. Dejar cero elementos en el array

Para dejar solo un elemento, también podemos hacerlo de varias formas:

```
ReDim nums(0 To 0)
ReDim Preserve nums(0 To 0)

Array.Resize(nums, 1)

nums = New Integer() {22}
```

Solución listado 3.27. Dejar el array con un elemento

Si había pensado en usar el método **Clear** de la clase **Array**, comentar que ese método “limpia” el contenido de los elementos que haya, pero no los elimina, por tanto, si hacemos esto:

```
Array.Clear(nums, 0, nums.Length)
```

El número de elementos del *array* **nums** seguirá siendo el mismo, solo que todos ellos tendrán un valor cero.

Otras dos cosas que tampoco servirían, es asignar un valor nulo al *array* (o usar la instrucción **Erase**), ya que eso eliminará el *array* de la memoria, por tanto, si intentamos acceder a cualquier miembro del *array*, recibiremos una excepción indicándonos que la variable **nums** no está inicializada.

```
nums = Nothing
Erase nums
```

Esto elimina el objeto de la memoria, pero no lo deja a cero

## El operador "binario" If

La función **If** también la podemos usar con dos parámetros, en este caso la evaluación se hace sobre el primero de ellos, y si ese valor resulta que es un valor nulo, se devuelve el segundo valor. Si ese primer parámetro no es nulo, lo que devuelve este operador es precisamente ese primer parámetro. Debido a que se hace una comprobación de que el primer operando pueda ser un valor nulo, los tipos de datos que podemos usar en la evaluación deben ser tipos por referencia o anulables, pero en el segundo argumento podemos indicar cualquier otro tipo, ya sea por valor o por referencia, y al igual que ocurre con el operador ternario, ese segundo argumento debe ser de un tipo que se pueda convertir implícitamente al primero. Si esto último no lo tenemos en cuenta, recibiremos un error como el mostrado en la figura 3.6 (en el código de la figura 3.6, la variable **i2** es de tipo **Integer?** y la variable **b2** es de tipo **Boolean?**, esas declaraciones las podemos ver en el listado 3.28).

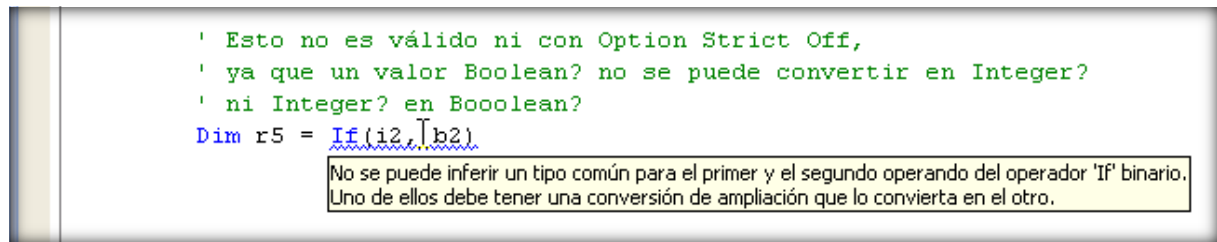


Figura 3.6. If necesita que los tipos evaluados sean de tipos “convertibles”

### Nota

*En algunos casos, el tipo de datos que devuelve el operador binario If no es el que tenga la primera expresión que se evalúa, ya que si la segunda es de un tipo al que la primera se puede “ampliar” (convertir implícitamente), se devolverá el tipo que tenga la segunda expresión.*

En el listado 3.28 vemos algunos ejemplos en los que podemos usar el operador **If** con solo dos argumentos. En ese listado vemos algunas de las cosas que podemos hacer y otras, que aunque sean “legales” en tiempo de compilación, pueden llegar a producir un error en tiempo de ejecución.

### Aclaraciones sobre el por qué de algunas de las “inferencias” del listado 3.28

En las asignaciones a las variables **r1** y **r2** en la que se utiliza la variable **ob**, se infiere **Object** porque al ser esa variable de tipo **Object** (aunque sea **Nothing**), no se puede inferir **String** (ni **Double**) porque un tipo base no se puede promocionar a uno más específico.

La conversión de **Integer?** a **Double** sí devuelve **Double**, porque Visual Basic puede inferir valores si hay una conversión implícita, es decir, un valor **Integer** se puede convertir a **Double** sin pérdida de información (esto no es herencia, es promoción de tipos).

Lo mismo ocurre al asignar la variable **rc1** en la que la variable **c3** contiene un valor nulo y en el segundo argumento usamos **c2** que es de tipo **Cliente2**, pero finalmente el tipo que se infiere es **Cliente**, no **Cliente2** (que es lo que seguramente pensaríamos al ver ese código); la explicación es porque un tipo **Cliente** no se puede convertir a **Cliente2** (ni de forma implícita –sin hacer una conversión–, ni explícita –con conversión–) ya que **Cliente2** se deriva de **Cliente**, y tal como comenté antes: *el segundo argumento debe ser de un tipo que se pueda convertir implícitamente al primero*. Sin embargo, siempre podremos asignar un valor de tipo **Cliente2** a una variable de tipo **Cliente**, ya que la variable de tipo **Cliente** obtendría la parte que *entiende* de **Cliente2**, es decir los métodos y propiedades que implementa la clase base **Cliente**.

Si el lector ha caído en estos “errores” no debe preocuparse, ya que muchas veces nos engañan las apariencias, por suerte, el editor de Visual Basic 2008 nos ayuda a comprobar qué tipos de datos son los que *realmente* se infieren en cada momento, ¡y sin necesidad de compilar!

```

Dim i1 As Integer = 15
Dim b1 As Boolean = True
Dim d1 As Double = 33.33
Dim ob As Object = Nothing

Dim i2? As Integer = Nothing
Dim b2? As Boolean = Nothing
Dim d2? As Double = 123.45

Dim r1 = If(ob, "Hola") ' Infiere Object
Dim r2 = If(ob, 125.33) ' Infiere Object
Dim r3 = If(i2, 125.33) ' Infiere Double

' Esto no se podrá usar ni con Option Strict Off,
' porque el tipo de i1 no es anulable o por referencia
'Dim r4 = If(i1, 22)

' Esto no es válido ni con Option Strict Off,
' ya que un valor Boolean? no se puede convertir en Integer?
' (ni Integer? en Boolean?)
'Dim r5 = If(i2, b2)

Dim r6 = If(i2, i1) ' Devuelve Integer

Dim r7 = If(d2, i1) ' Devuelve Double
Dim r8 = If(i2, d1) ' Devuelve Double

Dim c1 As New Cliente With {.Nombre = "Pepe"}
Dim c2 As New Cliente2 With {.Nombre = "Juan"}
Dim c3 As Cliente = Nothing
Dim c4 As Cliente2 = Nothing

Dim rc1 = If(c3, c2) ' Infiere Cliente
Dim rc2 = If(c4, c2) ' Infiere Cliente2
Dim rc3 = If(c1, c2) ' Infiere Cliente

' Esto no se puede hacer,
' ya que un tipo Cliente no se puede convertir en Cliente2
'Dim rc4 As Cliente2 = If(c1, c2)

' Y aunque hagamos la conversión,
' esto fallará si c1 no es nulo
Dim rc5 As Cliente2 = TryCast(If(c1, c2), Cliente2)

```

Listado 3.28. Varios ejemplos del operador If en forma binaria

## 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

## Características generales del lenguaje (III)

---

### Introducción

En este capítulo continuamos con las novedades de Visual Basic 9.0, 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 Visual Basic 9.0) 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 Visual Basic 9.0.

Las novedades que veremos en este cuarto 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 declaremos 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 Visual Basic 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, puesto que al ser anónimo, no podemos usar la cláusula **As** para definirlos. 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 anterior, no podemos indicar el tipo de datos, ya que es un tipo anónimo.

```
Dim tal = New With {.Nombre = "Guille", .Edad = 50}
```

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; esta inferencia de tipos siempre se hará de la forma adecuada, incluso si tenemos desactivado **Option Infer**.

La variable **tal** 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 **Integer**.

Cuando definimos un tipo anónimo de esta forma, las propiedades son de lectura/escritura, por tanto, podemos cambiar el contenido de las mismas, tal como vemos en el listado 4.2.

```
tal.Edad = 51
```

Listado 4.2. De forma predeterminada, las propiedades de los tipos anónimos son de lectura/escritura

Como ya comenté, Visual Basic 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. Después de la asignación del listado 4.2, 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 **VB\$AnonymousType\_0**, 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, recordemos que el signo \$ se utiliza en Visual Basic para indicar que una variable es de tipo **String**, por tanto, al encontrar ese signo acabaría la declaración del nombre del tipo).

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**.

```
Dim ta2 = New With {.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.

```
Dim ta3 = New With {.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 en Visual Basic.

```
class VB$AnonymousType_0`2<string, int32>  
class VB$AnonymousType_1`2<int32, string>
```

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

```
Class VB$AnonymousType_0(Of String, Integer)  
Class VB$AnonymousType_1(Of Integer, String)
```

Listado 4.6. Definición de los dos tipos anónimos al estilo de Visual Basic

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 Visual Basic 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 VBAnonymousType_0(Of T0, T1)  
    Public Nombre As T0  
    Public Edad As T1  
End Class  
  
Class VBAnonymousType_1(Of T0, T1)  
    Public Edad As T0  
    Public Nombre As T1  
End Class
```

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 (*VBAnonymousType\_0*) 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 no son de los tipos correctos.

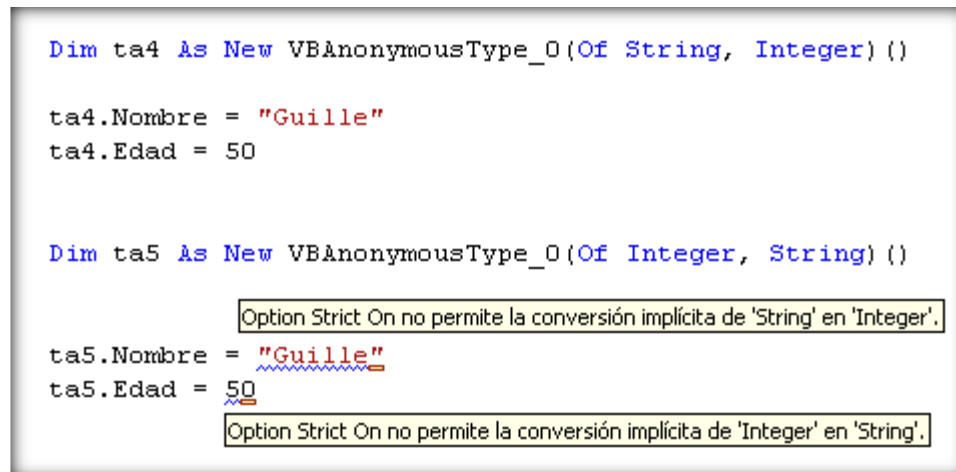


Figura 4.1. Error al asignar los valores a las propiedades del supuesto tipo anónimo

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).

## Definir un tipo anónimo con propiedades de solo lectura

De los métodos definidos en la clase **Object**, que son los únicos que tendrán un tipo anónimo, hay dos que se redefinen de una forma especial cuando el tipo anónimo define propiedades de solo lectura: **Equals** y **GetHashCode**.

El primero servirá para comparar dos instancias de tipos anónimos, ya que la comparación de esas instancias se realizará comprobando solo el contenido de las propiedades de solo lectura (el resto de propiedades se descartará en la comparación). Para realizar esa comparación se utiliza el valor generado por el método **GetHashCode** en cada una de las propiedades de solo lectura.

Para definir propiedades de solo lectura, tenemos que anteponer el modificador **Key** a la definición de la propiedad. Podemos definir tantas propiedades de solo lectura como queramos.

En el listado 4.8 tenemos la definición de un tipo anónimo con una propiedad de solo lectura.

```
Dim ta6 = New With {Key .ID = "009", .Nombre = "VB9"}
```

Listado 4.8. La propiedad ID es de solo lectura

Como es lógico, el contenido de las propiedades de solo lectura no lo podemos modificar. El valor se asigna en el constructor de la clase y permanece inalterado durante toda la vida de esa instancia. El constructor siempre lo define el compilador, y en todos los casos, este constructor recibe tantos parámetros como propiedades hayamos definido en el tipo anónimo. Pero el constructor es solo para uso interno del compilador, ya que nosotros no tenemos acceso a la creación de nuevas instancias de un tipo anónimo, (salvo cuando los definimos), esto no cambia ni aun cuando definamos propiedades de solo lectura.

Cuando definimos propiedades de solo lectura, esa característica también la tendrá en cuenta el compilador para decidir si el tipo se puede reutilizar o se debe crear uno nuevo. Por tanto, debemos añadir un nuevo condicionante a la lista que vimos en la sección anterior para saber cuándo se crea un nuevo tipo anónimo o se utiliza uno que hayamos definido anteriormente.

Si definimos un tipo anónimo como el mostrado en el listado 4.9, el compilador creará un nuevo tipo y no reutilizará el que definimos para asignar la variable *ta6*, ya que, si bien tiene las mismas propiedades y están en el mismo orden, la primera no es de solo lectura, por tanto, no coinciden las definiciones.

```
Dim ta7 = New With {.ID = "009", .Nombre = "VB9"}
```

Listado 4.9. Este tipo anónimo es diferente al definido en el listado 4.8

Como ya comenté, cuando comparamos dos instancias de tipos anónimos, dicha comparación se hará sobre las propiedades de solo lectura (las definidas con **Key**), por tanto, la comparación realizada en el código del listado 4.10 devolverá un valor falso, sin embargo, la comparación del listado 4.11 devolverá **True** (ejercicio: argumentar por qué esas dos comparaciones devuelven esos valores).

```
Dim sonIguales = ta6.Equals(ta7)
```

Listado 4.10. Esta comparación de igualdad fallará

```
Dim ta8 = New With {Key .ID = "009", .Nombre = "Visual Basic 9.0"}  
sonIguales = ta6.Equals(ta8)
```

Listado 4.11. Esta comparación devolverá un valor verdadero

### **Nota**

*Cuando queramos comparar dos instancias de tipos anónimos tendremos que usar el método Equals, ya que el operador de igualdad (=) no está sobrecargado para comparar tipos anónimos.*

La explicación de por qué las instancias de las variables *ta6* y *ta7* no son iguales es porque ambos tipos anónimos son diferentes, ya que en el primero definimos la propiedad *ID* de solo lectura, mientras que en el segundo es de lectura/escritura.

Sin embargo, las instancias asignadas a las variables *ta6* y *ta8* son del mismo tipo, y el valor de la propiedad de solo lectura es el mismo en ambas instancias, y como el contenido de las propiedades de lectura/escritura no se tiene en cuenta, al compilador le da igual lo que contengan esas propiedades, ya que no las evaluará.

En estos ejemplos solo hemos definido una propiedad de solo lectura, pero si los tipos definen más de una propiedad de solo lectura, la igualdad solo se cumplirá si todas y cada una de esas propiedades tienen el mismo valor en ambas instancias.

En el código del listado 4.12 tenemos varias instancias de un mismo tipo anónimo con dos propiedades de solo lectura; al comparar esas instancias se tendrá en cuenta el contenido de las dos para decidir si son iguales o no (recordemos que la propiedad de lectura/escritura no se tiene en cuenta en la comparación). El tipo anónimo de todas las variables es el mismo, sin embargo, la instancia de la variable *tac6* utiliza un tipo diferente para la propiedad *ID*, por tanto, fallará la comparación, ya que el valor *hash* creado para esa propiedad es distinto (el que devuelve el método *GetHashCode*).

```
Dim tac1 = New With {Key .ID = 10, Key .Codigo = "1234", .Precio = 125.5}
Dim tac2 = New With {Key .ID = 99, Key .Codigo = "1234", .Precio = 125.5}
Dim tac3 = New With {Key .ID = 10, Key .Codigo = "1234", .Precio = 888.3}
Dim tac4 = New With {Key .ID = 10, Key .Codigo = "ABCDE", .Precio = 125.5}
Dim tac5 = New With {Key .ID = 10, Key .Codigo = "ABCDE", .Precio = 777.3}
Dim tac6 = New With {Key .ID = "10", Key .Codigo = "1234", .Precio = 125.5}
' False, cambia el valor de ID
sonIguales = tac1.Equals(tac2)

' True, no se evalúan las propiedades de lectura/escritura
sonIguales = tac1.Equals(tac3)

' False, cambia el valor de Codigo
sonIguales = tac1.Equals(tac4)

' False, son del mismo tipo pero las propiedades son de tipo diferente
sonIguales = tac1.Equals(tac6)

' True, no se evalúan las propiedades de lectura/escritura
sonIguales = tac4.Equals(tac5)
```

Listado 4.12. En los tipos con varias propiedades de solo lectura, se tienen en cuenta todas ellas para comparar dos instancias

Si al lector le extraña que todas las variables sean del mismo tipo anónimo, le recuerdo que, en realidad, los tipos anónimos se definen con parámetros *generic* y al crear la instancia es cuando se indica el tipo de datos real que tendrá cada propiedad.

Por supuesto, este comportamiento solo ocurre si los nombres de las propiedades son iguales y están definidos en el mismo orden, ya que ese tipo anónimo, en realidad, se define como vemos en el seu-

docódigo del listado 4.13. La implementación de la clase es más compleja, pero de esta forma vemos los elementos básicos para tener una idea del comportamiento, en particular a la hora de crear la instancia, ya que en el constructor se utilizan los parámetros (propiedades) en el mismo orden en el que se han definido al crear el tipo anónimo. Si tuviésemos acceso a la definición de la clase anónima, las declaraciones del listado 4.12 serían como las del código del listado 4.14.

```
Class VBAnonymousType_4(Of T0, T1, T2)

    Public Sub New(ByVal ID As T0, _
                  ByVal Codigo As T1, _
                  ByVal Precio As T2)
        Me.ID = ID
        Me.Codigo = Codigo
        Me.Precio = Precio
    End Sub

    Public ReadOnly ID As T0
    Public ReadOnly Codigo As T1
    Public Precio As T2
End Class
```

Listado 4.13. Seudocódigo del tipo anónimo creado en el listado 4.12

```
Dim tac1 As New VBAnonymousType_4(Of Integer, String, Double) (10, "1234", 125.5)
Dim tac2 As New VBAnonymousType_4(Of Integer, String, Double) (99, "1234", 125.5)
Dim tac3 As New VBAnonymousType_4(Of Integer, String, Double) (10, "1234", 888.3)
Dim tac4 As New VBAnonymousType_4(Of Integer, String, Double) (10, "ABCDE", 125.5)
Dim tac5 As New VBAnonymousType_4(Of Integer, String, Double) (10, "ABCDE", 777.3)
Dim tac6 As New VBAnonymousType_4(Of String, String, Double) ("10", "1234", 125.5)
```

Listado 4.14. Así se definirían las variables usadas en el listado 4.12 si tuviésemos acceso al tipo anónimo

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 y la inferencia automática de tipos

Como ya comenté antes, cuando definimos un tipo anónimo, el compilador de Visual Basic utiliza la inferencia automática de tipos para decidir qué tipo de datos tendrán las propiedades e incluso para saber cuál será el tipo anónimo que debe utilizar, (uno existente, si usamos la misma firma o uno nuevo si no existe un tipo anónimo que coincida con el que estamos definiendo).

Pero debemos saber que esa inferencia de tipos la hará el compilador **incluso** si tenemos desactivado **Option Infer**, o casi. Veamos cuál es el “casi”.

Si tenemos desactivado **Option Infer**, también debemos desactivar **Option Strict**, ya que de no hacerlo, recibiremos el conocido error del compilador Visual Basic en el que nos avisa de que **Option Strict On** requiere que en todas las declaraciones de las variables usemos la cláusula **As**.

El problema de tener desactivadas estas dos opciones es que el tipo de la variable que recibe la instancia del tipo anónimo es **Object**. Por tanto, IntelliSense no nos mostrará las propiedades a las que podemos acceder, además de que cualquier acceso a esas propiedades se hará en tiempo de ejecución, utilizando lo que se conoce como enlace tardío (*late binding*). Así que, si no queremos padecer de los nervios y facilitar nuestro trabajo, lo mejor es que siempre tengamos activada **Option Strict**, además de **Option Infer**.

### **Nota**

*Independientemente de cómo tengamos Option Infer, los tipos de datos de las propiedades de los tipos anónimos siempre se inferirán correctamente, donde interviene el estado de Option Infer es en la inferencia de la variable que recibe el objeto del tipo anónimo.*

## **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 módulo (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 (en realidad Visual Basic no facilita este tipo de creaciones, salvo usando el truco que vimos en el capítulo dos), 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 4.15 vemos un ejemplo en el que la propiedad *Artículo* es a su vez un tipo anónimo.

```
Dim ta11 = New With {.ID = 1, _  
                    .Artículo = New With {.Cantidad = 12}, _  
                    .Descripción = "Prueba 11"}
```

Listado 4.15. Un tipo anónimo puede contener otros tipos anónimos

Y si necesitamos que una de esas propiedades sea un *array* o una colección, podemos usar el método *CrearLista* que vimos en el capítulo dos. En el listado 4.16 vemos un ejemplo.

```
Dim ta12 = New With {.ID = 2, _  
                    .Artículos = _  
                        CrearLista(New With {.Cantidad = 6}, _  
                                   New With {.Cantidad = 24}, _  
                                   New With {.Cantidad = 10}), _  
                    .Descripción = "Prueba 12"}
```

Listado 4.16. Incluso las propiedades pueden ser colecciones de tipos anónimos

Ejercicio: ¿de qué tipo será la propiedad *Artículos*? La respuesta, al final de la siguiente sección.

## 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 Visual Basic; 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.
- Siempre los crearemos usando la característica conocida como inicialización de objetos.
- Las propiedades pueden ser de solo lectura o de lectura/escritura.
- Las propiedades de solo lectura las definiremos anteponiendo la instrucción **Key**.

- 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, los tipos, los nombres (no se diferencia entre mayúsculas y minúsculas) y si son de solo lectura o de lectura/escritura.
- Solo podemos comparar la igualdad de dos instancias de tipos anónimos si son del mismo tipo y definen propiedades de solo lectura, en ese caso, la igualdad se comprobará teniendo en cuenta todas las propiedades de solo lectura.

## Solución al ejercicio del listado 4.16

Si usamos la definición del método *CrearLista* del listado 2.9 del segundo capítulo, la propiedad *Artículos* será del tipo *IEnumerable(Of el tipo\_anónimo)*, ya que el método *CrearLista* (con parámetros *generic*) infiere el tipo de datos que estamos usando como *array* de parámetros opcionales, es decir, el tipo anónimo que el compilador utilice para cada uno de los argumentos de esa función.

### Versiones

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

## Expresiones lambda

Según la documentación de Visual Studio 2008, una expresión *lambda* es una función sin nombre que podemos usar en cualquier sitio en el que un delegado sea válido.

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 Visual Basic 9.0 las expresiones *lambda* (o funciones anónimas o funciones en línea) son funciones que solo pueden tener una instrucción que será la encargada de devolver el valor. Esto significa que, en realidad, no es una función como las que estamos acostumbrados a definir en Visual Basic, entre otras cosas, por el hecho de que solo puede tener una instrucción, y cuando digo una instrucción me refiero a una sola instrucción, no a una sola línea de código (que también debe estar esa instrucción en una sola línea de código, aunque en esa línea de código podemos usar el guión bajo, pero solo para facilitar la lectura). Este comentario viene a cuento de que en Visual Basic podemos escribir en una misma línea varias instrucciones separadas por dos puntos.



En realidad, el contenido de las funciones anónimas se reduce a una sola expresión que devuelve un valor, de ahí que no podamos escribir más de una instrucción. Sabiendo esto, después veremos algunos trucos para ampliar el contenido de una función anónima.

### **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 Visual Basic 9.0. El primero es el término usado en la documentación de Visual Studio 2008, y que desde mi punto de vista es más apropiado para los que gustan de usar C# como lenguaje de programación, ya que el operador usado para las expresiones lambda de C# se llama “operador lambda” ( $\Rightarrow$ ); el segundo es porque en Visual Basic, en realidad, se definen como una función que no tiene nombre; y el tercero porque esa función la podremos usar directamente en el código, sin necesidad de crearla de forma independiente.*

## **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 Visual Basic, para indicar el método que queremos suscribir a un evento, utilizamos la instrucción **AddressOf** seguida del nombre del método. 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, y para asociar ese evento con ese método lo haremos tal como vemos en el listado 4.17.

```
AddHandler Button1.Click, AddressOf Button1_Click
```

Listado 4.17. AddHandler requiere un delegado en el segundo parámetro

El segundo parámetro de la instrucción **AddHandler** espera un delegado, éste indicará la dirección de memoria del método al que se debe llamar cuando se produzca el evento indicado en el primer parámetro.

En Visual Basic, para indicar la dirección de memoria de un método, tenemos que usar la instrucción **AddressOf** seguida del nombre del método. Y, como vemos en el listado 4.17, esa dirección de memoria la podemos indicar directamente.

Aunque, en realidad, el compilador lo que hace es crear una instancia del tipo del delegado y pasarle la dirección de memoria del método. Si quisiéramos hacer un poco más manual todo este proceso, el código del listado 4.17 lo podríamos sustituir por el del listado 4.18.

```
AddHandler Button1.Click, New EventHandler(AddressOf Button1_Click)
```

Listado 4.18. El equivalente del listado 4.17 usado por Visual Basic

La definición del listado 4.18 la podemos escribir también de la forma mostrada en el listado 4.19, en el que definimos una variable del mismo tipo del delegado que queremos usar con la instrucción **AddHandler**, ese delegado lo pasamos como segundo argumento de la instrucción.

```
Dim delegado As New EventHandler(AddressOf Button1_Click)  
AddHandler Button1.Click, delegado
```

Listado 4.19. Los delegados los podemos asignar a variables

En realidad, en Visual Basic todo esto es mucho más sencillo, pero en el fondo es lo que el compilador hace, y creo que nos servirá para comprender mejor la parte de la definición de las expresiones *lambda* en la que nos indica que las “podemos usar en cualquier sitio en el que un delegado sea válido”.

## Definir una expresión lambda

Las expresiones *lambda* las tenemos que definir usando la instrucción **Function** seguida de los parámetros que se usarán en la función y después de los parámetros usaremos una expresión; el tipo de datos que devuelva esa expresión será el tipo de datos de la función.

Al definir las expresiones *lambda* no indicaremos ni el tipo de datos que devuelve la función ni usaremos la instrucción **Return** para devolver ese valor.

Por ejemplo, si queremos sumar dos valores enteros y devolver el resultado de la suma, lo podemos hacer tal como vemos en el listado 4.20.

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

Listado 4.20. Definición de una expresión lambda

Para usar la función anónima del listado 4.20 lo haremos por medio de la variable *delSuma*, que para usarla tendremos que indicar los dos valores que queremos sumar y el resultado lo podemos guardar en una variable, usarlo en una expresión o lo que creamos conveniente. En el listado 4.21 tenemos un ejemplo de uso de ese delegado.

```
Dim i As Integer = delSuma(15, 22)
```

Listado 4.21. Uso del delegado que contiene una función anónima

El compilador en realidad lo que hace al encontrarse con la definición de una expresión *lambda*, es crear un delegado *generic* con esa misma firma, es decir, una función que recibe dos valores. El pseudocódigo generado por Visual Basic sería parecido al mostrado en el listado 4.22.

```
Delegate Function _
    VBAnonymousDelegate_0(Of T0, T1, T2)(ByVal n1 As T0, _
                                           ByVal n2 As T1 _
                                           ) As T2
```

Listado 4.22. Definición del delegado para la expresión lambda del listado 4.20

También crea una función que es la que se encarga de hacer la suma de los dos parámetros pasados a la expresión *lambda*; el pseudocódigo sería como el del listado 4.23.

```
Private Function _Lambda_1(ByVal n1 As Integer, _
                           ByVal n2 As Integer) As Integer
    Return n1 + n2
End Function
```

Listado 4.23. La función lambda creada por Visual Basic con la firma del delegado

La variable *delSuma* quedaría definida como vemos en el listado 4.24, y la forma de usar ese delegado será igual que el listado 4.21, eso no cambia.

```
Dim delSuma As New _
    VBAnonymousDelegate_0(Of Integer, Integer, Integer) _
        (AddressOf _Lambda_1)
```

Listado 4.24. Definición de la variable del tipo del delegado

Por supuesto, para utilizar las expresiones *lambda* no tenemos por qué saber todos estos pormenores, pero así tendremos una visión más clara de lo que tendríamos que hacer, si no existiera este tipo de funciones anónimas.

## Las expresiones lambda son funciones en línea

El código mostrado en los listados 4.20 y 4.21 lo podemos resumir en algo más conciso, de forma que no sea necesario crear un delegado intermedio; el código del listado 4.25 sería equivalente a esos dos.

```
Dim k = (Function(x As Integer, y As Integer) x + y) (15, 22)
```

Listado 4.25. Una expresión lambda usada directamente

En el ejemplo del listado 4.25, la variable *k* recibe el valor resultante de evaluar la expresión, es decir, es de tipo **Integer**, ya que en este caso no creamos ninguna variable intermedia para almacenar la función que realiza la operación. Por supuesto, el código que genera el compilador es parecido a todo lo que vimos anteriormente.

Como podemos comprobar, el código del listado 4.25 no tiene mucha utilidad ya que la variable *k* recibe el cálculo que hace la expresión *lambda* y para hacer esa suma no es necesario complicarse tanto, pero más que nada es para ver lo que podemos hacer con este tipo de funciones anónimas.

## 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 Visual Basic, 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.

```
Delegate Function SumaCallback(ByVal num1 As Integer, _  
                               ByVal num2 As Integer) As Integer
```

Listado 4.26. Definición del delegado

```
Function sumar(ByVal num1 As Integer, _
               ByVal num2 As Integer) As Integer
    Return num1 + num2
End Function
```

Listado 4.27. La función con la misma firma que el delegado

```
Sub conDelegado(ByVal x As Integer, _
               ByVal y As Integer, _
               ByVal d As SumaCallback)
    Console.WriteLine("x = {0}, y = {1}", x, y)
    Console.WriteLine("d(x,y) = {0}", d(x, y))
End Sub
```

Listado 4.28. Un método que recibe un delegado como parámetro

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

Listado 4.29. Una de las formas de usar el método del listado 4.28

Para usar el método del listado 4.28 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 4.29, 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 4.30.

```
Dim sumaDel As SumaCallback = AddressOf sumar
conDelegado(22, 33, sumaDel)
```

Listado 4.30. Otra forma de utilizar el método del listado 4.28

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 4.31 también sería válido.

```
conDelegado(30, 70, Function(x, y) x + y)
```

Listado 4.31. El método del listado 4.28 lo podemos usar con una expresión lambda

La ventaja de usar el código del listado 4.31 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 funciones anónimas sería una tarea fácil, ya que

solo tendríamos que cambiar la expresión que devuelve el valor, como en el listado 4.32 en el que efectuamos una resta.

```
conDelegado(30, 70, Function(x, y) x - y)
```

Listado 4.32. Al ser una función en línea, podemos cambiar el resultado a devolver

Si esto mismo quisiéramos hacerlo en Visual Basic 8.0/2005 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.

En .NET Framework 3.5 hay una serie de delegados *generic* que nos pueden servir para reducir aún más nuestro código. Como hemos visto en los listados 4.26 y 4.28, para crear ese método que recibe un delegado, antes hemos tenido que definir el delegado (también declaramos un método -listado 4.27- para usar al llamar al método del listado 4.26, pero con las expresiones *lambda*, esa definición nos la ahorramos), con los delegados que .NET 3.5 define, también nos podemos ahorrar el delegado. Por supuesto, si no definimos el delegado del listado 4.26 tendremos que cambiar la definición del método mostrado en el listado 4.28. La nueva definición será como la mostrada en el listado 4.33. **Func** es un delegado *generic* definido en .NET Framework 3.5, en este ejemplo, utiliza tres tipos de datos, los dos primeros representan los parámetros de la función y el tercero es el valor que devuelve esa función. Para usar ese método, lo haremos como vemos en el código del listado 4.34, la operación que realizamos en este caso es una multiplicación.

```
Sub conDelegado2(ByVal x As Integer, _
                ByVal y As Integer, _
                ByVal d As Func(Of Integer, Integer, Integer))
    Console.WriteLine("x = {0}, y = {1}", x, y)
    Console.WriteLine("d(x,y) = {0}", d(x, y))
End Sub
```

Listado 4.33. Nueva definición del método que recibe un delegado en el tercer parámetro

```
conDelegado2(50, 20, Function(x, y) x * y)
```

Listado 4.34. Llamada al método que define el delegado Func

.NET 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), todos ellos de tipo **Function**, ya que siempre devuelven algo; si necesitamos un delegado que no devuelva nada (equivalente a un procedimiento **Sub**), 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 4.33, los tres parámetros usados son de tipo **Integer**.

Volvamos a ver el código del listado 4.31 (o del listado 4.32 o del listado 4.34). ¿Qué nota en esa forma de usar la expresión *lambda* del tercer parámetro?

Si miramos la diferencia de los tres listados, puede que nos parezca que sea la expresión usada para devolver el valor, pero no es eso lo que debemos notar, ya que lo extraño en esas expresiones *lambda* es que no hemos indicado los tipos de datos de las dos variables usadas en la función anónima. En este caso, entra en funcionamiento la inferencia automática de tipos, y como el compilador sabe que el delegado que se utiliza para el tercer parámetro tiene dos parámetros de tipo **Integer**, asume que esas dos variables deben ser de ese tipo o de cualquier tipo que se pueda convertir de forma implícita al tipo **Integer**.

Pero debemos tener en cuenta que si no indicamos los tipos de datos, el compilador los infiere según la definición del delegado, pero si indicamos el tipo, ese tipo debe ser del “adecuado”, es decir, de tipo **Integer**. Además de que si indicamos el tipo de uno de los parámetros, debemos indicar el tipo de todos ellos; sabiendo esto, el código del listado 4.35 daría error, porque solo indicamos el tipo de datos del primer parámetro.

```
conDelegado2 (25, 71, Function(x As Integer, y) x + y)
```

Listado 4.35. Si indicamos el tipo de un parámetro, debemos indicarlos 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.

Lo que debemos tener en cuenta es que el “cuerpo” de una función anónima siempre será una expresión y debemos imaginarnos que delante de esa expresión hay una instrucción **Return**, de esta forma nos resultará más fácil saber qué es lo que puede contener la expresión.

En el listado 4.36 vemos un ejemplo en el usamos variables definidas fuera de la expresión *lambda*.

```

Dim i1 As Integer = 10
Dim d1 As Double = 22.5

Dim d = Function(n As Integer) (n * i1) + d1

Console.WriteLine(d(12))

```

Listado 4.36. 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 4.37 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.

```

Function conDelegado3(ByVal x As Integer, _
                    ByVal y As Integer _
                    ) As Func(Of Integer, Integer, Integer)
    Dim k = (x + y) \ 2
    Console.WriteLine("k = {0}", k)

    Return Function(n1 As Integer, n2 As Integer) n1 * k + n2 \ k
End Function

```

Listado 4.37. Un método que devuelve una expresión lambda

En el código del listado 4.38 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.

```

Dim d2 = conDelegado3(7, 4)
Dim s = d2(3, 3)
Console.WriteLine(s)

s = d2(5, 7)
Console.WriteLine(s)

```

Listado 4.38. 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.

## Utilizar una expresión lambda como un método de evento

Si retomamos el código que usamos al principio para asignar el método del evento **Click** de un botón, podríamos definirlo como una expresión *lambda* en lugar de indicar un método existente. En el listado 4.39 vemos cómo definir este tipo de función anónima, que la podemos hacer en el evento **Load** del formulario (o en cualquier otro sitio, pero antes de usar ese evento). El contenido de la expresión utiliza dos controles que tiene ese formulario, y como hemos comprobado, cuando llegue el momento (en este caso, cuando el usuario haga clic en ese botón) se evaluará esa expresión, por tanto, en cada una de esas invocaciones se añadirá al contenido de la colección **Items** del **ListBox** el texto que tenga el control *TextBox1*.

```
AddHandler btnAdd.Click, Function() ListBox1.Items.Add(TextBox1.Text)
```

Listado 4.39. Utilizar un método de evento por medio de una expresión lambda

En la definición de la expresión *lambda* del listado 4.39 nos aprovechamos de la relajación en las conversiones de los delegados para no tener en cuenta los dos parámetros que el delegado asociado al evento **Click** define.

## Ampliando la funcionalidad de las expresiones lambda

Sabiendo que en las expresiones de las funciones anónimas podemos usar cualquier elemento que esté definido en el código (siempre que esté en ámbito), caeremos en la cuenta de que si podemos usar variables u objetos de cualquier tipo, también podremos usar otras funciones, siempre y cuando tengamos en cuenta que esa expresión es solo una “simple” expresión, lo de *simple* es para enfatizar que es solamente una expresión, pero que podemos complicar todo lo que queramos, incluso utilizando otras funciones anónimas como parte de esa expresión.

En el listado 4.40 tenemos la definición de dos funciones anónimas donde la segunda utiliza la primera para realizar sus cálculos. En el listado 4.41 tenemos el equivalente en el que usamos dos expresiones *lambda*. En el listado 4.41 he tenido que cambiar el nombre de la primera variable usada como parámetro, ya que si dejara el mismo nombre que la que defino después en la segunda función anónima, el compilador daría un error de que ese nombre ya se está usando, error que es lógico, ya que ambas funciones anónimas están en una misma expresión.

```
Dim d1 = Function(x As Integer, y As Integer) x + y
Dim d2 = Function(x As Integer) x * d1(x, 2)

Console.WriteLine(d2(5))
```

Listado 4.40. Definición de dos funciones anónimas en la segunda se utiliza la primera

```
Dim d3 = Function(n As Integer) n *
    (Function(x As Integer, y As Integer) n + y)(n, 2)

Console.WriteLine(d3(5))
```

Listado 4.41. Código equivalente al código del listado 4.40 pero anidando las dos funciones anónimas

Por supuesto, además de usar funciones anónimas, también podemos usar funciones normales, es decir, cualquier cosa que podamos escribir en una expresión la podemos escribir en la expresión que devuelve el valor de una función anónima.

## ¿Cómo clasificar una colección de tipos anónimos?

Como vimos al hablar de los tipos anónimos, podemos usar la función *generic* **CrearLista** para asignar una colección de tipos anónimos a una propiedad de otro tipo anónimo, si modificamos esa función para que devuelva una colección de tipo **List(Of T)**, podríamos clasificar esa colección, ya que esa clase tiene un método **Sort**.

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(Of 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 relajación de delegados, 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 4.42 tenemos una adaptación del código mostrado en el listado 4.16, 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 4.43 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.

```
Dim ta13 = New With {
    .ID = 17, _
    .Artículos =
        CrearLista(New With {
            .ID = 95, .Cantidad = 6}, _
            New With {
                .ID = 22, .Cantidad = 24}, _
            New With {
                .ID = 95, .Cantidad = 1}, _
            New With {
                .ID = 39, .Cantidad = 10} _
        ), _
    .Descripción = "Prueba 13"}
```

Listado 4.42. Un tipo anónimo que tiene una colección de tipos anónimos

```
ta13.Artículos.Sort(Function(x, y) x.ID.CompareTo(y.ID))
```

Listado 4.43. Usamos una expresión lambda como función a usar con el método Sort

En el código del listado 4.43, la sobrecarga que usamos del método **Sort** es la que utiliza el delegado **Comparison(Of 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 4.42, 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 4.16, 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**.

Como comenté al principio, el método **CrearLista** tiene que devolver un objeto del tipo **List(Of T)**, por tanto, debemos cambiar la definición de ese método de forma que usemos la del listado 4.44, que como vemos es prácticamente la misma que la mostrada en el capítulo dos.

```
Function CrearLista(Of T) (ByVal ParamArray datos() As T) As List(Of T)
    Dim lista As New List(Of T)
    lista.AddRange(datos)
    Return lista
End Function
```

Listado 4.44. Definición de la función CrearLista para que devuelva un tipo que define un método Sort

En la comparación que hacemos en el cuerpo de la función anónima podemos comparar lo que queramos, no solo los valores solitarios de las propiedades de esa clase anónima. Por ejemplo, en el listado 4.45 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.Articulos.Sort(Function(x, y) _
    (x.ID.ToString("00") & x.Cantidad.ToString("000")) _
    .CompareTo _
    (y.ID.ToString("00") & y.Cantidad.ToString("000")) _
)
```

Listado 4.45. 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 (**Integer** 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 4.46 tenemos el equivalente al listado 4.45, pero usando una función externa, que es la definida en el listado 4.47.

```
ta13.Articulos.Sort(Function(x, y) _
    comparaEnteros( _
        x.ID, x.Cantidad, _
        y.ID, y.Cantidad) _
)
```

Listado 4.46. La expresión usada en la función anónima usa una función externa

```
Function comparaEnteros(ByVal x1 As Integer, ByVal x2 As Integer, _
    ByVal y1 As Integer, ByVal y2 As Integer _
) As Integer
    Dim x, y As String
    x = x1.ToString("00") & x2.ToString("000")
    y = y1.ToString("00") & y2.ToString("000")
    Return x.CompareTo(y)
End Function
```

Listado 4.47. La función usada por la expresión lambda del listado 4.46

En los listados 4.45 y 4.47 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 4.42, y mostrándolos como vemos en el listado 4.48, el resultado sería el mostrado en la figura 4.2.

```
Console.WriteLine("{0}, {1}", ta13.ID, ta13.Descripción)
For Each a In ta13.Artículos
    Console.WriteLine("  {0}", a.ToString)
Next
```

Listado 4.48. Mostrar el contenido del tipo anónimo y de la colección Artículos

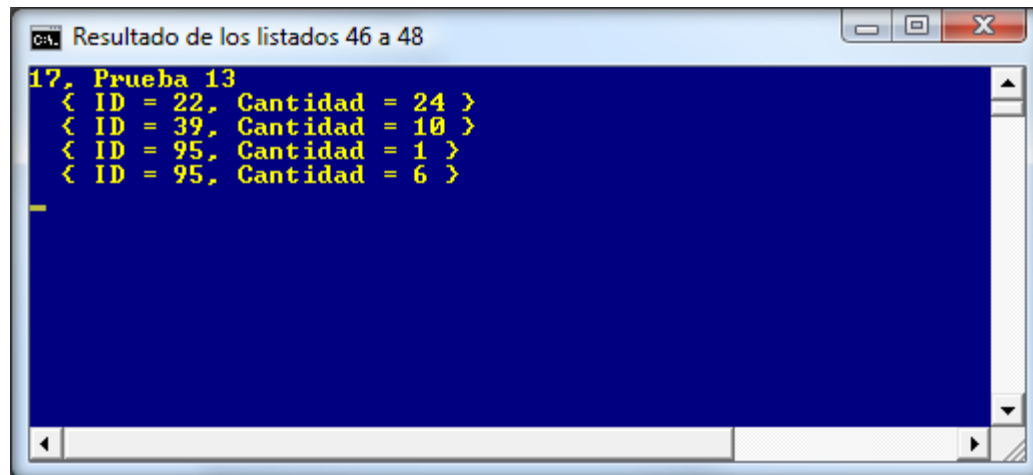


Figura 4.2. Resultado de mostrar los datos usando el listado 4.48

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.

*Por suerte*, Visual Basic nos permite hacer las cosas de forma “no estricta”, y en esta ocasión puede estar un poco justificado prescindir de **Option Strict On**. Para entender mejor este comentario, veamos el código del listado 4.49, en el que usamos una expresión *lambda* como argumento del método **Sort**. En esta ocasión usamos un método externo al que le pasamos los dos objetos (del tipo anónimo almacenado en la colección **Artículos**). Como sabemos, esto no se puede hacer si tenemos activada la comprobación estricta del código, ya que esa función debe definir los parámetros del tipo

adecuado, en este caso del tipo anónimo usado en la colección. No quisiera parecer repetitivo, pero a estas alturas ya sabemos que no podemos definir parámetros de un tipo anónimo.

```
ta13.Articulos.Sort(Function(x, y) Module3.comparaArticulos(x, y))
```

Listado 4.49. Clasificamos los elementos de la colección Artículos pasándole dos objetos del tipo anónimo a una función definida en otro módulo

Pero si desactivamos **Option Strict** podemos engañar al compilador y definir ese método para que reciba dos valores de tipo **Object**, y después usar la compilación tardía (*late binding*) para acceder a las dos propiedades que nosotros sabemos que tiene ese tipo anónimo, y eso es lo que hace el código del listado 4.50.

```
Option Strict Off
Option Infer On

Imports System

Module Module3
    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 Module
```

Listado 4.50. Una función que utiliza late binding para acceder a las propiedades de dos objetos

Ni qué decir tiene 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.

**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 la instrucción **Function**.
- A diferencia de los procedimientos de tipo **Function**, no se indica un nombre después de esa instrucción.
- No se indica tampoco el tipo de datos que devuelve, ese tipo siempre se infiere dependiendo del valor devuelto en la expresión.
- Para devolver el valor no se utiliza la instrucción **Return**. El valor devuelto es la expresión que indicamos en el cuerpo de la función.
- 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.
- Debido a que solo se usa una expresión como cuerpo de la función, no podemos indicar el final de la función (**End Function**).
- Las expresiones *lambda* se pueden usar con parámetros, si no se pueden inferir los tipos de esos parámetros, tenemos que indicarlos expresamente.
- 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 **Optional** ni **ParamArray**.
- 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.
- 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.*

# Capítulo 5

## Características generales del lenguaje (IV)

---

### Introducción

En este capítulo continuamos con las novedades de Visual Basic 9.0, 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 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 (Sub o Function) 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 sen-



tido 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 existente necesitamos dos cosas: la primera es usar las librerías de .NET Framework 3.5, en particular el ensamblado **System.Core.dll**; la segunda es utilizar el atributo **Extension** que está definido en el espacio de nombres **System.Runtime.CompilerServices**. Ese atributo lo aplicaremos al método que queremos usar para extender la funcionalidad de una clase.

Además de los dos mencionados, 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 un módulo (**Module**) y que en el primer parámetro de ese método indiquemos el tipo de datos que vamos a extender.

Juntando esos requisitos tendremos las siguientes condiciones que siempre debemos tener en cuenta a la hora de crear un método extensor:

1. Crear el proyecto para .NET Framework 3.5 (automáticamente tendremos una referencia al ensamblado **System.Core.dll**).
2. Añadir un módulo (**Module**) al proyecto para definir en ese módulo el método extensor.
3. Definir el método extensor cuyo primer parámetro debe ser del tipo que queremos extender.
4. Aplicar el atributo **Extension** a ese método.
5. Para facilitar el uso de ese atributo deberíamos agregar una importación del espacio de nombres **System.Runtime.CompilerServices**.

Después veremos que hay más requisitos o condiciones que debemos tener en cuenta, pero con estas cinco tenemos suficiente para empezar.

De los cinco puntos indicados anteriormente, el primero y el quinto están relacionados, y los puntos cuatro y cinco también están estrechamente relacionados. Toda esta interrelación es porque el atributo **Extension** está definido en el espacio de nombres **System.Runtime.CompilerServices**, y ese espacio de nombres está incluido en el ensamblado **System.Core.dll**.

El segundo requisito 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 un módulo (**Module**) siempre estará accesible, al menos en el mismo ensamblado en el que definamos ese módulo.

Independientemente de todos esos requisitos, el más importante es el tercero. Tan importante es la forma de definir el método extensor, que si no lo hacemos bien, el resto de requisitos da igual que los tengamos en cuenta. Pero si ese método no lo definimos en un módulo, tampoco servirá de nada, y si no le aplicamos el atributo **Extension**, el compilador no se enterará de que queremos “extender” la funcionalidad de un tipo, y si no usamos la versión 3.5 de .NET tampoco podremos usar ese atributo... Resumiendo: debemos tener en cuenta como mínimo esos cinco requisitos (como acabamos de comprobar, todos son necesarios, si no, no serían requisitos).

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.

```
Imports System.Runtime.CompilerServices

Module Extensiones
    <Extension()> _
    Public Function Len(ByVal str As String) As Integer
        If String.IsNullOrEmpty(str) Then
            Return 0
        End If

        Return str.Length
    End Function
End Module
```

Listado 5.1. Un método extensor para la clase String

En el código del listado 5.1 tenemos todos los requisitos comentados (la referencia al *core* de .NET 3.5 no la vemos, pero si no estuviera, no podríamos importar el espacio de nombres, etc., etc., etc.).

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.

```
Dim s1 = "Hola, extensiones"
Dim i1 = s1.Len
```

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

Cuando el compilador de Visual Basic 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 (**Nothing**). 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.

```
Dim s2 As String = Nothing
Dim 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.

En realidad, no era necesario asignar el valor **Nothing** a la variable **s2**, ya que si no asignamos nada a una cadena, ésta contiene un valor nulo, pero así nos evitamos la advertencia del compilador de que vamos a usar una variable antes de que se le haya asignado *algo*... aunque lo que le asignemos sea *nada*.

## 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 Visual Basic como a los definidos en otros lenguajes o en el propio .NET).

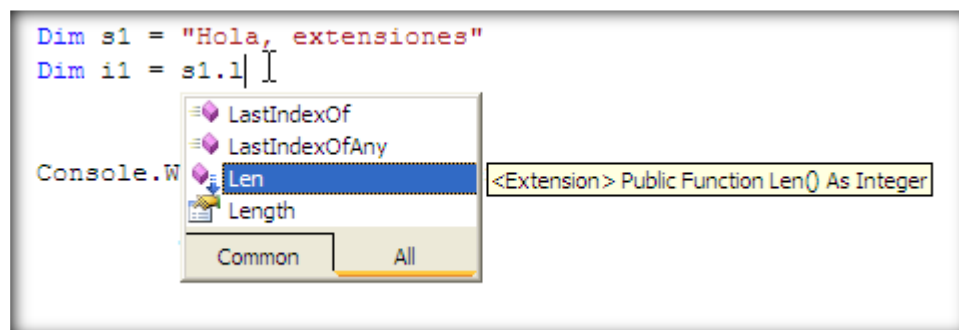


Figura 5.1. IntelliSense y los métodos extensores

## 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 métodos 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.

Pero en el fondo, los métodos extensores solo son eso: métodos. Un poco especiales, sí, pero métodos normales, al menos si usamos Visual Basic para definirlos, ya que excepto por la salvedad de que deben tener el atributo **Extension**, son métodos como cualquier otro que podamos escribir (en C# los métodos extensores no utilizan el atributo **Extension**, pero sí se definen de una forma inusual, ya que deben indicar la palabra clave **this** antes del primer parámetro, es decir, antes del tipo de datos que extienden).

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 módulo 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 módulo con el método **Main**). Sí, resulta que es idéntico al método extensor del listado 5.1, pero para usarlo lo tendremos que usar como 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.

```
Function Len(ByVal str As String) As Integer
    If String.IsNullOrEmpty(str) Then
        Return 0
    End If

    Return str.Length
End Function
```

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

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, aunque en este caso, al existir otro método con el mismo nombre, tendremos que usar el nombre completo, es decir, usando el nombre del módulo en el que está definido, tal como vemos en el listado 5.6.

```
i1 = Len(s1)
```

Listado 5.5. Uso habitual de un método

```
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 atributo **Extension**, que cuando los usamos desde el entorno integrado resulta más evidente esa especialización (porque IntelliSense lo re-

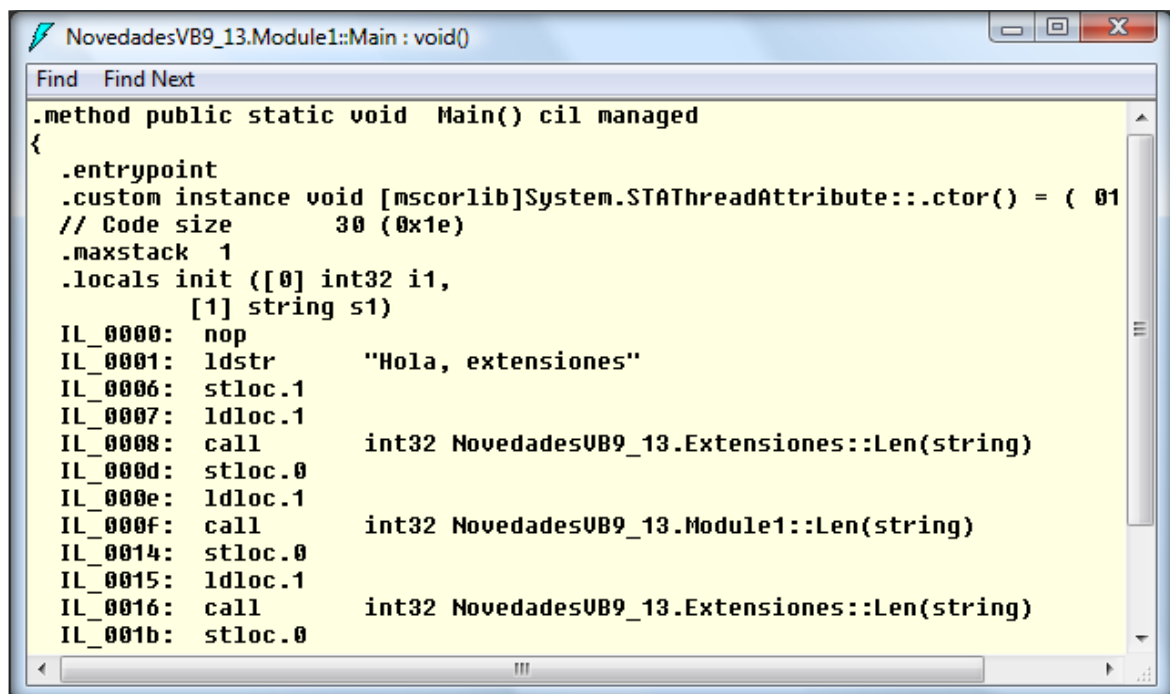
conoce, etc.). Por supuesto, también podemos usar los métodos extensores en cualquier archivo de código, incluso aunque no los escribamos con el IDE de Visual Basic, ya que es el compilador de Visual Basic 9.0 el que reconoce que es un método extensor y sabe qué tiene que hacer con él.

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 Visual Basic 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 Basic 9.0, 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. En el caso del método definido en el mismo módulo en el que está todo el código, la única diferencia es la clase en la que está definido el método, que en este ejemplo se llama *Module1*, mientras que el método extensor está definido en un módulo llamado *Extensiones*.



```
.method public static void Main() cil managed
{
    .entrypoint
    .custom instance void [mscorlib]System.STAThreadAttribute::.ctor() = ( 01
    // Code size      30 (0x1e)
    .maxstack 1
    .locals init ([0] int32 i1,
                  [1] string s1)
    IL_0000: nop
    IL_0001: ldstr      "Hola, extensiones"
    IL_0006: stloc.1
    IL_0007: ldloc.1
    IL_0008: call       int32 NovedadesVB9_13.Extensiones::Len(string)
    IL_000d: stloc.0
    IL_000e: ldloc.1
    IL_000f: call       int32 NovedadesVB9_13.Module1::Len(string)
    IL_0014: stloc.0
    IL_0015: ldloc.1
    IL_0016: call       int32 NovedadesVB9_13.Extensiones::Len(string)
    IL_001b: stloc.0
}
```

Figura 5.2. Los métodos extensores son métodos normales

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\_000f** es el listado 5.5 y la línea **IL\_0016** corresponde al listado 5.6. En el código intermedio (IL) siempre se usan las clases con el nombre completo, por eso, en las tres invocaciones al método **Len** se usa primero el espacio de nombres del proyecto, que en este ejemplo es **NovedadesVB9\_13**, seguido del nombre de la clase en la que se define el método y finalmente el método que se invoca.

## El ámbito de los métodos extensores

En Visual Basic los métodos extensores se definen en un módulo (**Module**). La peculiaridad de los módulos es que, por un lado, son clases en las que todos los miembros que definimos están compartidos, por tanto, siempre accesibles (no necesitamos crear una instancia de la clase para acceder a ellos), por otro lado, el compilador agrega una importación al nombre del módulo, de forma que tampoco sea necesario usar el nombre del módulo para acceder a los miembros que define. Debido a estas peculiaridades, podemos usar los métodos definidos en un módulo sin necesidad de indicar el nombre del mismo. Si usáramos el equivalente a los módulos (que es definiendo una clase con métodos compartidos), si no hacemos nada especial, siempre tendríamos que indicar el nombre de la clase para acceder a esos métodos.

Sí, ya sé que en este libro no tengo que explicar estas cosas que el lector ya conoce, pero he querido aclararlas por si alguien tenía dudas en el comportamiento de los módulos de Visual Basic.

Aclarado cómo se definen los módulos, y cómo facilita Visual Basic el acceso a los métodos que define, veamos cómo puede afectar todo esto a los métodos extensores.

Como vimos en el listado 5.6, para acceder de forma directa al método extensor definido en el módulo **Extensiones**, tuvimos que usar el nombre de dicho módulo, ya que si no lo hubiésemos indicado, el método **Len** que utilizaría esa llamada sería el definido en ese mismo módulo.

En este caso concreto, la definición de un método **Len** “normal” no entra en conflicto con el método extensor, pero ¿qué ocurriría si ese método “normal” lo convertimos en un método extensor?

Si Visual Basic se encuentra con dos métodos extensores que se llaman igual (y también se aplican al mismo tipo de datos), y éstos están definidos en el mismo espacio de nombres, lo que hace el compilador es usar el que esté en un ámbito más cercano. Esto no cambia con respecto al resto de métodos, es decir, siempre usará el que tenga en la misma clase (o tipo) antes que otro definido fuera de esa clase.

Pero si los dos métodos están definidos en módulos diferentes, y los dos módulos tienen el mismo ámbito, el compilador no sabrá cuál de ellos debe usar como extensión, por tanto, nos mostrará un error de compilación indicando que existe un conflicto de resolución de sobrecarga, tal como podemos ver en la figura 5.3.

Aclarar que tanto el módulo **Extensiones** como **Extensiones2** están definidos directamente en el mismo proyecto, por tanto, el espacio de nombres de los dos módulos es el mismo, en este caso, **NovedadesVB9\_14**.

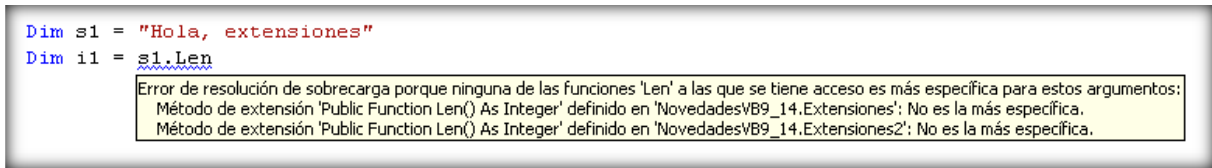


Figura 5.3. Error de conflictos de nombres en los métodos extensores

Este conflicto lo podemos solucionar añadiendo una importación a uno de los dos módulos. De esta forma, Visual Basic usará el indicado en esa importación, en este caso concreto, el código del listado 5.7 solucionará el conflicto, pero como veremos más adelante, hay otras formas de controlar mejor los métodos conflictivos que queremos usar como métodos extensores.

```
Imports NovedadesVB9_14.Extensiones
```

Listado 5.7. Una importación de un tipo con métodos extensores soluciona el error mostrado en la figura 5.3

Como vemos en el listado 5.7, la importación la hacemos al nombre del módulo que define los métodos extensores que queremos usar. Esto hace que Visual Basic encuentre una definición del método **Len** que no entra en conflicto con otras definiciones.

La pregunta que seguramente el lector se hará es: ¿por qué no hay errores al usar esa importación?

Y para el lector que no se haya dado cuenta de que, en realidad, ese truco no debería solucionar el conflicto, le pediría que se volviera a leer lo que comenté sobre las “peculiaridades de los módulos” al principio de esta sección.

Para entenderlo mejor, ampliaré la pregunta para que sea más evidente la duda: ¿por qué no hay errores al usar esa importación si Visual Basic agrega una importación a los módulos?

Es decir, en teoría Visual Basic agrega siempre una importación a los módulos que definimos, y esa importación es parecida a la mostrada en el listado 5.7.

Pero en este caso, el compilador se encuentra con una importación explícita, por tanto, le da preferencia a esa importación antes de la agregada “tras las bambalinas” (vamos, la que el propio compilador agrega para que siempre podamos acceder a los miembros de los módulos).

## Precedencia en el ámbito de los métodos extensores

Según la documentación de Visual Basic, para evitar conflictos en los métodos extensores, se utilizan las siguientes comprobaciones para decidir qué definición del método extensor conflictivo se debe usar:



1. Los definidos en el módulo actual.
2. Los definidos en tipos de datos del espacio de nombres actual o en cualquiera de sus padres, teniendo mayor preferencia los espacios de nombres hijos.
3. Los definidos en cualquier tipo importado en el archivo actual.
4. Los definidos en cualquier espacio de nombres importado en el archivo actual.
5. Los definidos en cualquier tipo importado a nivel de proyecto.
6. Los definidos en cualquier espacio de nombres importado a nivel de proyecto.

Finalmente nos dice que si no se puede evitar el conflicto, que no lo usemos como método extensor y lo utilicemos como un método normal (aunque seguramente tendremos que usar el nombre del tipo en el que está definido).

### Definir los módulos con métodos extensores en su propio espacio de nombres

Para evitar los conflictos con métodos extensores que tienen el mismo nombre, se recomienda que los módulos 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 (de esta forma, las importaciones automáticas de Visual Basic serán más fáciles de detectar, ya que los módulos definidos en un espacio de nombres no se importan automáticamente o al menos no al nivel del espacio de nombres usado en el proyecto).

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 (módulo) en el que definimos los métodos extensores. De esta forma, tendríamos “aislados” (o casi) los métodos extensores del resto del código de nuestro proyecto.

Además, sabiendo cómo resuelve Visual Basic los nombres de los métodos extensores (ver la lista del apartado anterior), nos resultará fácil agregar importaciones más adecuadas a nuestro proyecto.

Pero debemos tener en cuenta que Visual Basic es muy protector, por tanto, hay ocasiones en las que hace parte del trabajo que deberíamos hacer nosotros. Y en esto de los espacios de nombres no es una excepción. Cada vez que creamos un nuevo proyecto, Visual Basic crea un espacio de nombres para todo el proyecto, por tanto, todos los tipos (y espacios de nombres) que definamos en ese proyecto estarán “dentro” de ese espacio de nombres global que el compilador crea por nosotros.

Para clarificar todo lo comentado, veamos un ejemplo. En el listado 5.8 vemos cómo definir esos métodos extensores dentro de su propio espacio de nombres. En este ejemplo, si el proyecto se llama



**NovedadesVB9\_15**, la importación que tendríamos que hacer para acceder a esos métodos sería como vemos en el listado 5.9, ya que el espacio de nombres que envuelve al módulo **Extensiones** está en el espacio de nombres definido expresamente, además del espacio de nombres global que el propio Visual Basic crea por nosotros.

```
Namespace elGuille.Grup01
    Module Extensiones
        <Extension()> _
        Public Function Len(ByVal str As String) As Integer
            If String.IsNullOrEmpty(str) Then
                Return 0
            End If

            Return str.Length
        End Function
    End Module
End Namespace
```

Listado 5.8. Definición del módulo con los métodos extensores en su propio espacio de nombres

```
Imports NovedadesVB9_15.elGuille.Grup01
```

Listado 5.9. Al importar espacios de nombres definidos en el mismo proyecto, debemos anteponer el espacio de nombres del proyecto actual

Esta automatización en la creación de un espacio de nombres global debemos tenerla en cuenta, ya que si los métodos extensores los definimos en una biblioteca de clases, no será necesario crear un espacio de nombres adicional, ya que todos los tipos de datos (incluidos los módulos) estarán definidos dentro de ese espacio de nombres global (o espacio de nombres raíz que llama la documentación en castellano). Este automatismo lo veremos en el siguiente apartado.

## Ejemplo de conflicto de nombres de métodos extensores

Supongamos que tenemos dos ensamblados en las referencias de nuestro proyecto y que tenemos importaciones a los espacios de nombres principales de esos dos ensamblados. Si en esos ensamblados se definen varios métodos extensores, pero hay conflicto en el nombre de algunos de ellos (están definidos en los dos ensamblados), lo podemos solucionar cambiando una de las importaciones para que se importe el tipo en el que se definen esos métodos extensores. De esta forma, podremos seguir usando todos los métodos extensores (de los dos ensamblados), pero ya no habrá conflictos, ya que Visual Basic encontrará una definición con más “ámbito” en el ensamblado del que se ha importado el nombre del tipo, por tanto, los métodos que estén en el tipo importado tendrán preferencia sobre los métodos que están definidos en la otra clase de la que solo hemos importado el espacio de nombres que contiene la definición de la clase.

Para clarificar este maremágnum de importaciones, veamos cómo están definidos los métodos extensores de esos dos ensamblados que tenemos en las referencias y cómo usar las importaciones en el proyecto que usará esos métodos de los dos ensamblados.

En el listado 5.10 tenemos el código de un ensamblado escrito en C# que define dos métodos extensores. En este código, el espacio de nombres raíz se llama *Extensiones\_CS*. De los dos métodos extensores que define, uno entrará en conflicto con los definidos en el código del listado 5.11, que al ser un proyecto de Visual Basic, no es necesario indicar expresamente el espacio de nombres que contiene las declaraciones, ya que, como comenté antes, Visual Basic utiliza de forma predeterminada el espacio de nombres indicado en las propiedades del proyecto, que en el código del listado 5.11 es *Extensiones\_VB*. Destacar del código del listado 5.11, que el módulo está definido como **Public**, ya que si no lo hiciéramos así, el ámbito predeterminado que tendría sería **Friend**, por tanto, no lo podríamos usar externamente (salvo que ese ensamblado fuese un ensamblado amigo).

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

            return str.Length;
        }

        public static int Max(this string str, string str2)
        {
            int l1 = str.Len();
            int l2 = str2.Len();

            return Math.Max(l1, l2);
        }
    }
}
```

Listado 5.10. Métodos extensores definidos en C# para usar desde otro proyecto

Los métodos extensores mostrados en los listados 5.10 y 5.11 los importamos en otro proyecto de Visual Basic, y para acceder a esos métodos tendremos que añadir una importación a los espacios de nombres de cada uno de los ensamblados.

En el código del listado 5.12 importamos los dos espacios de nombres. Estas importaciones nos permiten acceder a todos los métodos extensores definidos en ambos ensamblados. En ese mismo código vemos cómo usar todos los métodos extensores definidos en los dos ensamblados, pero al usar “la extensión” *Len*, tendremos un conflicto de nombres, tal como vemos en la figura 5.4.

```

Public Module Extensiones
    <Extension()>
    Public Function Len(ByVal str As String) As Integer
        If String.IsNullOrEmpty(str) Then
            Return 0
        End If

        Return str.Length
    End Function

    <Extension()>
    Public Function MTrim(ByVal str As String) As String
        Dim sb As New System.Text.StringBuilder
        For Each c In str
            If Char.IsWhiteSpace(c) = False Then
                sb.Append(c)
            End If
        Next

        Return sb.ToString
    End Function
End Module

```

Listado 5.11. Métodos extensores definidos en un proyecto independiente

```

Option Strict On
Option Infer On

Imports System
Imports Extensiones CS
Imports Extensiones VB

Module Module1
    Sub Main()
        Dim s1 = "Hola, extensiones"
        Dim i1 = s1.Len
        Dim s2 = s1.MTrim
        Dim m1 = s1.Max(s2)

        Console.WriteLine("s1 = '{0}', Len = {1}", s1, i1)
        Console.WriteLine("s2 = '{0}', Len = {1}", s2, s2.Len)
        Console.WriteLine("Max = {0}", m1)

    End Sub
End Module

```

Listado 5.12. Uso de los métodos extensores de los dos ensamblados

La forma de solucionar este conflicto es cambiando una de las importaciones, de forma que importemos el nombre de la clase en la que se definen los métodos a los que queremos darle prioridad. En el ejemplo del código del listado 5.13, le damos preferencia a los métodos extensores definidos en el módulo *Extensiones* del proyecto de Visual Basic, pero sería igualmente válido haber importado el tipo del proyecto de C#, ya que lo importante es que una de las clases tenga mayor prioridad para poder resolver adecuadamente el conflicto de nombres.

```
Imports Extensiones_CS
Imports Extensiones_VB.Extensiones
```

Listado 5.13. Si importamos el tipo, le damos preferencia a los métodos extensores definidos en ese tipo

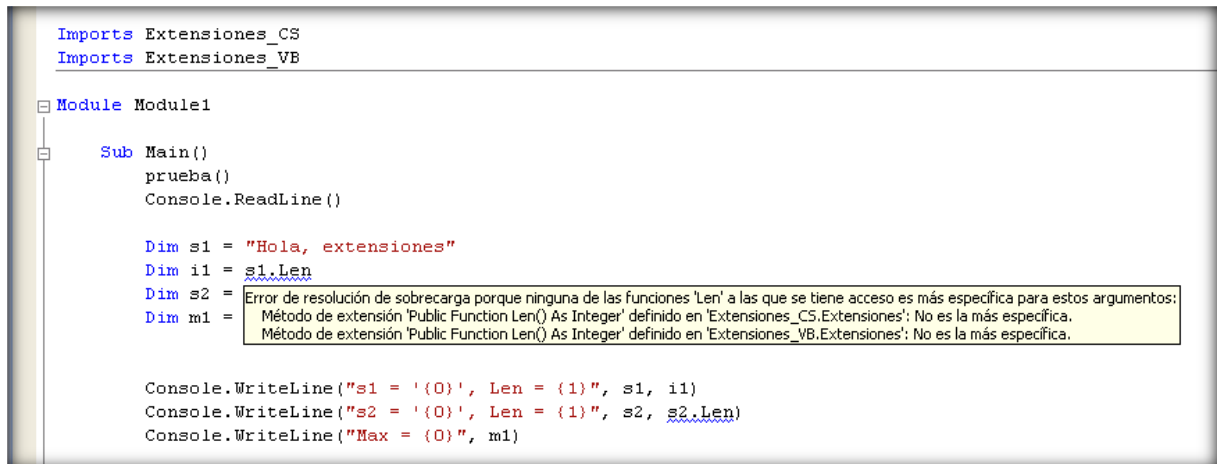


Figura 5.4. Conflicto de nombres en un método extensor definido en los dos espacios de nombres importados

## 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 estén compartidos, ya que los métodos compartidos siempre se deben usar indicando el tipo de datos que los define.*

*Aunque Visual Basic solamente nos muestre una advertencia cuando usamos los métodos compartidos desde una variable de instancia, no accederá a los métodos extensores que sobrecarguen a uno compartido ni aunque los usemos desde una instancia.*

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 de la clase (o 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 parámetros opcionales en los métodos extensores

Como sabemos, en Visual Basic podemos definir parámetros opcionales en los métodos. En los métodos extensores también podemos usar parámetros opcionales, 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.

Pero debemos tener en cuenta que el uso de parámetros opcionales, en realidad, es como si creáramos sobrecargas, por tanto, pueden entrar en conflictos 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 5.14 vemos una sobrecarga del método **Trim** que usa parámetros opcionales. Al usar este método extensor, IntelliSense nos mostrará ese método de la forma habitual de los parámetros opcionales (ver la figura 5.5), 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 para el segundo parámetro. 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.

```
<Extension()>
Public Function Trim(ByVal str As String,
                    Optional ByVal mtrim As Boolean = False) As String
    If mtrim Then
        Return str.MTrim
    Else
        Return str.Trim
    End If
End Function
```

Listado 5.14. Método extensor con parámetros opcionales

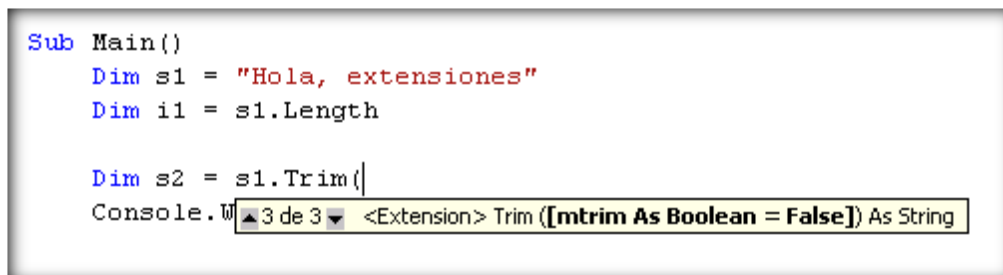


Figura 5.5. IntelliSense muestra los parámetros opcionales entre corchetes

### Nota

*Sin en la definición del método extensor mostrado en el listado 5.14, cambiamos el valor predeterminado a True, para que podamos usar ese método extensor, habrá que indicar expresamente el valor True (o el valor False), ya que si no indicamos nada, se usará el método Trim sin parámetros que define la clase String.*

## Métodos extensores que entran en conflicto con instrucciones de Visual Basic

En Visual Basic, los constructores se definen como procedimientos de tipo **Sub**, pero esto no significa que sean métodos, por tanto, no podemos crear métodos extensores que sobrecarguen a un constructor.

En este caso, si intentamos crear un método extensor que se llame **New**, el compilador no nos dejará hacerlo. Entre otras cosas, porque al estar definido en un módulo, ese constructor está compartido, y los constructores compartidos (**Shared**) no pueden recibir parámetros, por tanto, sería imposible extender la funcionalidad de una clase (o tipo) si no lo indicamos como primer parámetro del método extensor.

En cualquier caso, si nuestra intención es crear un método extensor que se llame **New** (y teniendo en cuenta que en realidad **New** es una palabra clave de Visual Basic), tendremos que encerrar ese nombre entre corchetes. De esta forma, el compilador sabrá que no estamos usando **New** como una instrucción, sino como un nombre.

Si algún programador de C# crea un método extensor con el nombre **New**, no recibirá error al definirlo, (porque **New** para C# no es una instrucción), pero al usarlo desde Visual Basic, tendremos que hacerlo usando corchetes (en realidad el propio IDE de Visual Basic le añadirá esos corchetes al nombre, para que no haya conflictos).

En el listado 5.15 vemos cómo definir un método extensor llamado **New** y en el código del listado 5.16 tenemos un ejemplo de cómo usarlo. Resulta evidente que esto solo es para mostrar cómo crear este tipo de métodos extensores conflictivos, ya que no tiene mucha utilidad práctica, al menos en este caso en concreto.

```
<Extension()> _  
Public Sub [New] (ByVal unColega As Colega, ByVal nombre As String)  
    unColega.Nombre = nombre  
End Sub
```

Listado 5.15. Método extensor con el nombre de una palabra reservada de Visual Basic

```
Dim c2 As New Colega With {.Correo = "guille@nombres.com"}  
c2.[New] ("Guillermo")  
Console.WriteLine(c2.Nombre)
```

Listado 5.16. Los nombres conflictivos debemos usarlos entre corchetes

En este caso en particular, he utilizado **New** por dos razones: una porque se define como un método **Sub** y la otra porque es una palabra reservada de Visual Basic. Por supuesto, si el nombre del método extensor que queremos definir entra en conflicto con cualquier otro nombre reservado por el compilador, tendremos que indicarlo entre corchetes.

## ¿Qué tipos de datos podemos extender?

Podemos crear métodos extensores para cualquier tipo de datos: clases, estructuras, interfaces e incluso delegados. Lo que no podremos extender son los módulos (**Module**), ya que, en realidad, no tiene mucho sentido, particularmente porque al ser un tipo del que no se pueden crear instancias, tampoco podemos usar ese tipo como primer parámetro del método extensor.

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.

En el código del listado 5.17 vemos la definición de una interfaz, una clase que la implementa directamente y otra clase que se deriva de esa clase. En ésta (que también hereda la implementación de la interfaz) definimos un método que coincide con el método extensor definido en el listado 5.18. La forma de usar estos métodos extensores la vemos en el listado 5.19, 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
    Function Mostrar() As String
End Interface

Public Class Prueba
    Implements IPrueba

    Public Contenido As String

    Public Function Mostrar() As String Implements IPrueba.Mostrar
        Return Contenido
    End Function
End Class

Public Class OtraPrueba
    Inherits Prueba

    Public Function Imprimir() As String
        Return Contenido
    End Function
End Class
```

Listado 5.17. Interfaz y clases que usan la interfaz



```
Public Module Extensiones
    <Extension()> _
    Public Function Imprimir(ByVal ipr As IPrueba) As String
        Return ipr.Mostrar
    End Function
End Module
```

Listado 5.18. Un método extensor para la interfaz

```
Dim pr1 As New Prueba With {.Contenido = "Prueba 1"}
Dim pr2 As New OtraPrueba With {.Contenido = "Prueba 2"}

' Se usa la extensión
Dim s1 = pr1.Imprimir
' se usa el método de instancia
s1 = pr2.Imprimir

Dim ipr As IPrueba = pr1
' Se usa la extensión
s1 = ipr.Imprimir

Dim ipr2 As IPrueba = pr2
' Se usa la extensión
' ya que aunque el objeto pr2 define ese método,
' la interfaz no lo define
s1 = ipr2.Imprimir
```

Listado 5.19. 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 (el que indicamos con **AddressOf** al crear la instancia del delegado) pueden ser métodos anónimos (casi como en C#, pero que en Visual Basic son más simples, tal como vimos en el capítulo anterior), 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 usamos 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 sea de tipo **Function** y 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 5.20 tenemos la definición de un delegado que recibe un parámetro de tipo **String** y devuelve un valor de ese mismo tipo. En el listado 5.21 tenemos la definición de un método extensor para ese delegado, el cual usaremos con dos argumentos de tipo cadena. En el código del listado 5.22 definimos un método que tiene la misma firma que el delegado, y finalmente, en el listado 5.23 vemos cómo podemos usar ese delegado y el método extensor.

```
Public Delegate Function PruebaDelegado (ByVal str As String) As String
```

Listado 5.20. Definición de un delegado para el que crearemos un método extensor

```
<Extension()> _
Public Function Saludo (ByVal deleg As PruebaDelegado, _
                        ByVal str As String, _
                        ByVal msg As String) As String
    Return deleg (msg & " " & str)
End Function
```

Listado 5.21. Un método extensor para el delegado del listado 5.20

```
Function pruebaDelegado1 (ByVal str As String) As String
    Return str
End Function
```

Listado 5.22. Un método con la misma firma del delegado del listado 5.20

```
Dim pd1 As PruebaDelegado = AddressOf pruebaDelegado1

Dim s5 = pd1 ("Mundo")

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

Listado 5.23. Ejemplo de uso del delegado y el método extensor

En el listado 5.24 vemos cómo usar el método extensor definido en el listado 5.21, pero en lugar de usar el delegado con un método existente, lo definimos de forma que utilice un método anónimo (o expresión *lambda*).

```
Dim pd3 As PruebaDelegado = Function (str As String) "Hola, " & str & ""
Console.WriteLine (pd3 ("Guille"))

s5 = pd3.Saludo ("Guille", "Hola")
```

Listado 5.24. Asignación de una función anónima 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, **NotInheritable** en Visual Basic o **sealed** en C#).

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 un módulo público. Esto último es fácil de conseguir, ya que si no cambiamos el ámbito, los módulos siempre serán para usar en el propio ensamblado (por defecto son **Friend**).

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, cualquiera puede crear un método extensor con el mismo nombre, darle mayor prioridad y el compilador usará ese último método antes que el que nosotros hemos definido.

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

# Visual Basic y LINQ

---

Esta es la parte final del libro en la que a lo largo de cuatro capítulos veremos todas las novedades de Visual Basic 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**.

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

# Capítulo 6

## Visual Basic y LINQ (I)

---

### Visual Basic y LINQ

El tema de este capítulo del libro de novedades de Visual Basic 9.0, 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 Visual Basic) 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) a nuestro código de Visual Basic, 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 Visual Basic 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 Basic 9.0.

### Un ejemplo de LINQ para ir abriendo boca

A lo largo de este capítulo, iremos viendo las nuevas instrucciones que se han añadido a Visual Basic 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 versiones de la función *CrearLista*, que hemos visto en los capítulos anteriores; si queremos tener la posibilidad de clasificar los elementos de la lista generada, usaremos la versión que devuelve un objeto del tipo *List(Of 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 utilizar la versión que devuelve el valor *List(Of T)* 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 Visual Basic nos permite crear ese tipo de colecciones de una forma muy simple y en realidad sin necesidad de usar una función de apoyo. En el código de los proyectos de ejemplo, puede ver otras formas de crear la “colección” de artículos sin usar la función *CrearLista*.

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.

```
Dim artículos = CrearLista( _
    New Artículo("01") _
        With {.Descripción = "Refresco lima", _
            .PrecioVenta = 0.92D}, _
    New Artículo("02") _
        With {.Descripción = "Refresco naranja", _
            .PrecioVenta = 0.9D}, _
    New Artículo("00") _
        With {.Descripción = "Refresco cola", _
            .PrecioVenta = 0.95D}, _
    New Artículo("11") _
        With {.Descripción = "Vino tinto brick", _
            .PrecioVenta = 0.55D}, _
    New Artículo("21") _
        With {.Descripción = "Rioja 0.75 ml", _
            .PrecioVenta = 1.2D} _
)
```

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 como campos públicos, pero solo por simplificar el código.

```
Class Artículo

    Public Código As String
    Public Descripción As String
    Public PrecioVenta As Decimal
    Public IVA As Decimal = 16D

    Public Sub New()
    End Sub

    Public Sub New(ByVal código As String)
        Me.Código = código
    End Sub

    Public Overrides Function ToString() As String
        Return String.Format("{0}, {1}, {2}", _
            Código, Descripción, PrecioVenta)
    End Function

End Class
```

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 Visual Basic. En la comparación de la cláusula **Where** he preferido usar la función **Left** en lugar de hacer la comparación con el método **StartsWith** de la clase **String** para “igualar” el código de T-SQL con el de Visual Basic.

```
Dim refrescos = From d In artículos _
                Where Left(d.Código, 1) = "0" _
                Order By d.PrecioVenta Ascending _
                Select 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, 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**) aparentemente tiene dos propiedades: **Descripción** y **PrecioVenta**, en realidad cada elemento de la colección es un tipo anónimo con esas dos propiedades (en un momento veremos por qué y cómo se ha generado ese tipo anónimo); 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 indiquemos, 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 mezclamos temas y sigamos con todas las novedades de Visual Basic relacionadas con LINQ.

```
For Each r In refrescos
    Console.WriteLine("{0} {1:0.00}", _
        r.Descripción.PadFillLeft(15), _
        r.PrecioVenta)
Next
```

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

```
<Extension()> _
Public Function PadFillLeft(ByVal str As String, _
    ByVal total As Integer) As String
    Return (str & New String(" ", total)).Substring(0, total)
End Function
```

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

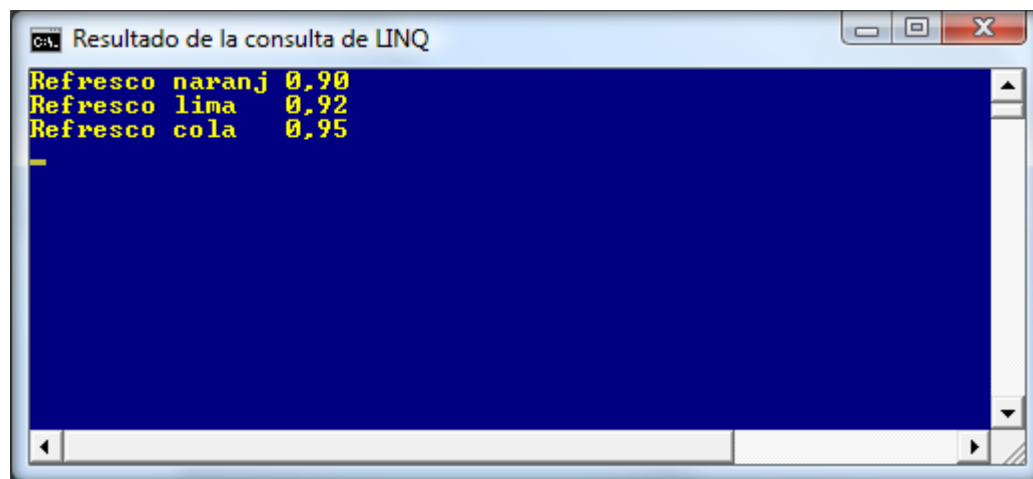


Figura 6.1. Resultado de la consulta LINQ

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 Visual Basic 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 Visual Basic del lenguaje de consultas, veamos qué variantes (o proveedores) de LINQ nos ofrece .NET Framework 3.5 y podemos usar desde Visual Basic.

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, que como veremos en Visual Basic se le ha dado un tratamiento especial, de forma que podamos utilizar XML directamente en nuestro código.
- **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 **SolidQ** (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 Visual Basic, 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 Basic 9.0.

## 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(Of 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 Visual Basic 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 Visual Basic.

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.

```
Dim nums() As Integer = {1, 9, 8, 2, 5, 7, 4, 3, 6}

Dim res = From n In nums Where n > 4

For Each v In res
    Console.WriteLine(v)
Next
```

Listado 6.7. Una consulta LINQ con un array numérico

Cuando usamos esa variable en el bucle **For Each** es cuando se pone en marcha toda la maquinaria de LINQ para hacer efectivo ese código, dando como resultado una colección del tipo **IEnumerable(Of Integer)** 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 **In**.

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

**Select n**, ¡Ups! ¡No hay instrucción **Select** en nuestra consulta! En realidad en Visual Basic es opcional finalizar la expresión de consulta con **Select**. Si se omite, como en este caso, el valor que se toma es el indicado después de **From**, en este ejemplo, el valor de *n*, es decir el valor de cada elemento de la colección que cumpla la condición de la cláusula **Where**.

La variable que recibe esa expresión de consulta es del tipo **IEnumerable(Of T)**, siendo *T* el tipo devuelto por **Select**. En este ejemplo un tipo **Integer**. 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, simplemente prepara la variable *res* para contener esa expresión, esto es lo que se conoce como consulta de ejecución aplazada, es decir, cada vez que utilizemos 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 **For Each**, 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:")
For Each v In res
    Console.WriteLine(v)
Next
```

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

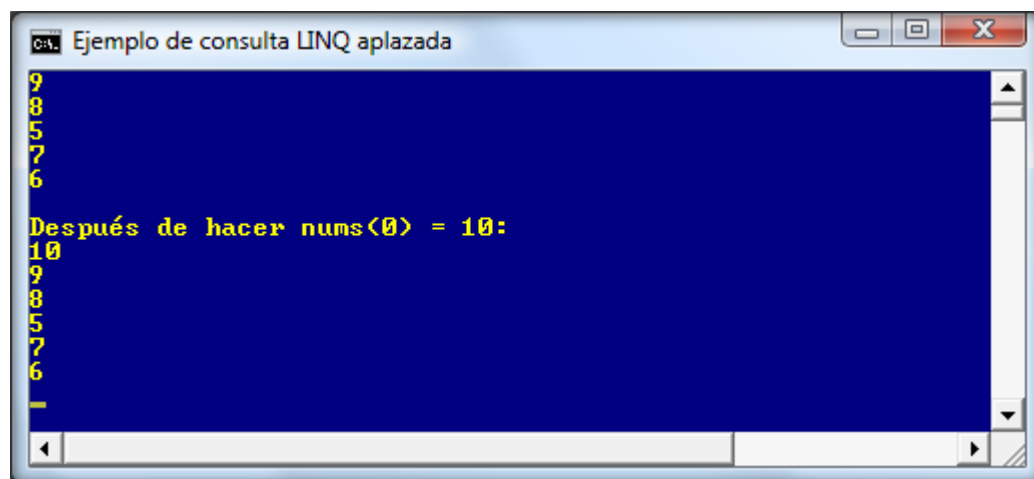


Figura 6.2. Resultado de ejecutar el código de los listados 6.7 y 6.8

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
ReDim Preserve nums (nums.Length - 1)
' Asignamos al primer elemento el valor 1
nums(0) = 1

' Se mostrará lo que había antes del cambio
Console.WriteLine()
For Each v In res
    Console.WriteLine(v)
Next
```

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 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. El resultado que obtenemos en la variable **res2** también lo podríamos obtener utilizando la cláusula **Aggregate** en lugar de **From**, ya que Visual Basic nos permite esas dos formas de iniciar una expresión de consulta LINQ. En el listado 6.11 vemos cómo obtener la suma de todos los elementos de la colección generada usando la cláusula **Aggregate**; en este caso, la ejecución también es inmediata y se devuelve el valor del tipo usado con la función **Sum**.

```
Dim res2 = (From n In nums Where n > 4).Sum
Console.WriteLine(res2)

Dim res3 = (From n In nums Where n > 4).ToList

For Each v In res3
    Console.WriteLine(v)
Next

nums(0) = 5
For Each v In res3
    Console.WriteLine(v)
Next
```

Listado 6.10. Consultas de ejecución inmediata

```
Dim res4 = Aggregate n In nums _  
    Where n > 4 _  
    Into Sum(n)
```

Listado 6.11. Podemos usar Aggregate si nuestra intención es usar una función agregada

## 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(Of 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. Si no utilizamos **Select**, el compilador utilizará como valor la misma variable indicada 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 6.12, 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 6.13.

```
Dim res = From n In nums _  
    Where n > 4 _  
    Select "Número " & n
```

Listado 6.12. Esta consulta devuelve los elementos como una cadena compuesta por el texto y el valor

```
Dim res2 = From n In nums _  
    Where n > 4 _  
    Select New With {.Valor = "Número " & n, _  
                    .EsPar = (n Mod 2 = 0)} _  
  
For Each v In res2  
    Console.WriteLine("{0}, es par = {1}", _  
        v.Valor, v.EsPar)  
Next
```

Listado 6.13. 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 **Boolean** que nos indica con un valor **True** si el número en cuestión es par. En el bucle **For Each** 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.

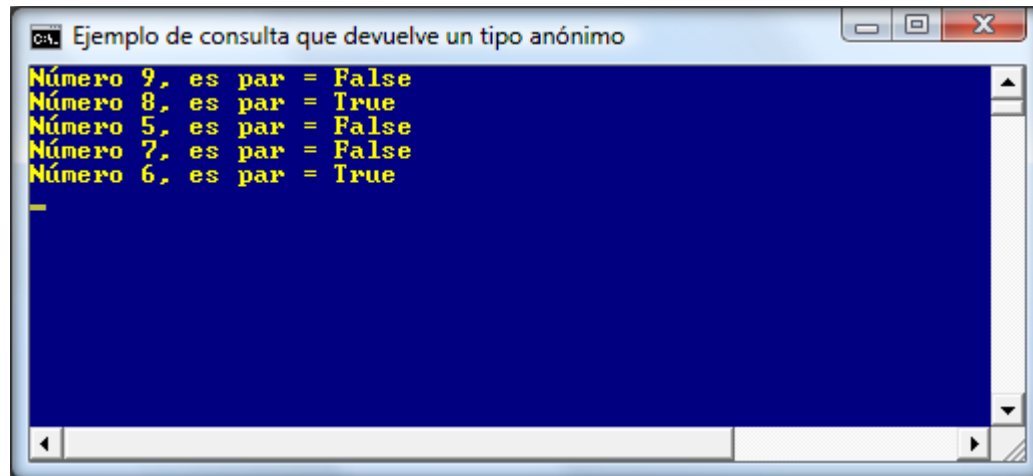


Figura 6.3. Resultado de ejecutar el código del listado 6.13

## 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: **Order By**. 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 6.14 vemos un ejemplo de uso de estas instrucciones de ordenación.

```
Dim res3 = From n In nums _  
           Where n > 4 _  
           Order By n Descending _  
           Select New With {.Valor = "Número " & n, _  
                           .EsPar = (n Mod 2 = 0) }
```

Listado 6.14. Consulta con ordenación de los elementos

## No es oro todo lo que reluce

Las nuevas instrucciones de consulta de Visual Basic 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 6.15 también la podríamos escribir como la mostrada en el listado 6.16.

```
Dim res4 = From n In nums _  
           Where n > 4 _  
           Order By n Descending _  
           Select n
```

Listado 6.15. Una consulta LINQ con las instrucciones de Visual Basic

```
Dim res5 = nums.Where(Function(n) n > 4). _  
                OrderByDescending(Function(n) n). _  
                Select(Function(n) n)
```

Listado 6.16. La consulta del listado 6.15 utilizando métodos extensores y expresiones lambda

Como podemos comprobar, es mucho más intuitivo el código del listado 6.15 que el correspondiente al listado 6.16, 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 Visual Basic 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 Basic 2008 incluyen esa referencia).*



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

# Capítulo 7

## Visual Basic y LINQ (II)

---

### Instrucciones de Visual Basic para las consultas de LINQ

Como ya he comentado en varias ocasiones, Visual Basic 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 **DataSet** (*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.

```
Dim noms () As String = {"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. Como ya vimos, el uso de esta cláusula es opcional, si la omitimos, el valor que se devuelve es el mismo usado después de **From**, y como veremos, otros valores que estén en rango.

Pero si la usamos, podemos personalizar el contenido de cada elemento de la colección devuelta por la consulta, tal como vimos en el capítulo anterior.

Además de crear tipos anónimos en la cláusula **Select** (ver listado 6.13 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 **For Each**, la colección resultante de esa expresión tendrá un tipo anónimo con las propiedades que hemos indicado en la cláusula **Select**.

```
Dim res = From a In artículos _
           Where a.PrecioVenta < 1.0 _
           Select Cod = a.Código, PVP = a.PrecioVenta

For Each a In res
    Console.WriteLine("{0} {1}", a.Cod, a.PVP)
Next
```

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 deben empezar con la cláusula **From** o con **Aggregate**, como ya vimos; la primera produce una consulta de ejecución aplazada y la segunda nos sirve para crear consultas de ejecución inmediata.

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 **For Each** usamos esos nombres de propiedades para mostrar los datos). En la figura 7.1 vemos el resultado de ejecutar ese código.

```
Dim res1 = From n In nums _  
           From s In noms _  
           Where s.Length = n _  
           Select s, n  
  
For Each v In res1  
    Console.WriteLine("{0}, {1}", v.s, v.n)  
Next
```

Listado 7.3. Dos sentencias From concatenadas

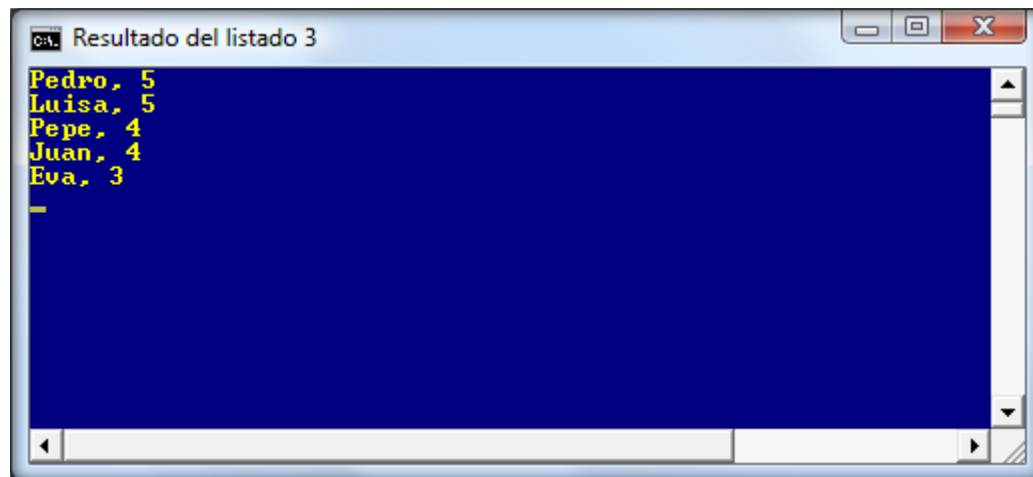


Figura 7.1. Resultado del listado 7.3

Cuando utilizamos varias cláusulas **From**, podemos separarlas con comas y solo indicar esa instrucción una vez; en el listado 7.4 vemos un ejemplo similar al anterior, en el que además utilizamos alias para nombrar las propiedades del tipo de datos devuelto como elemento de la colección. El resultado producido es el mismo que el código anterior.

Estos dos códigos 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 a los dos anteriores, en este caso, el resultado mostrado sería el mismo que el de la figura 7.1.

```
Dim res2 = From n In nums, _
           s In noms _
           Where s.Length = n _
           Select Nombre = s, Len = n

For Each v In res2
    Console.WriteLine("{0}, {1}", v.Nombre, v.Len)
Next
```

Listado 7.4. Si concatenamos varias sentencias From, podemos separarlas con comas

```
Dim res3 = From n In nums _
           Join s In noms _
           On s.Length Equals n _
           Select Nombre = s, Len = n

For Each v In res3
    Console.WriteLine("{0}, {1}", v.Nombre, v.Len)
Next
```

Listado 7.5. Ejemplo usando Join en lugar de dos From concatenadas

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. Después de **On** podemos indicar más condiciones, que también deben cumplirse (**And**), pero siempre debemos usar **Equals** para evaluar esas condiciones y debemos tener en cuenta lo comentado sobre la obligatoriedad de usar las dos variables que están en rango.

## Aggregate

Esta cláusula la podemos utilizar en dos contextos distintos: uno, como ya vimos antes, sirve para realizar una consulta y devolver un valor único, ese valor lo obtendremos por medio de algunas de las funciones de agregado definidas en Visual Basic o bien por las que nosotros creemos; también podemos utilizar esta instrucción como parte de la evaluación de otra consulta de LINQ, por ejemplo, como valor a utilizar con la cláusula **Where**, de forma que ese valor lo comparemos con cualquier otra cosa (**Where** siempre espera una expresión que devuelva un valor **Boolean**).

Las funciones de agregado definidas en Visual Basic son: **All**, **Any**, **Average**, **Count**, **LongCount**, **Max**, **Min** y **Sum**.

**Nota**

*Al final de este capítulo veremos cómo definir una función de agregado para que nos devuelva la suma de todos los números pares 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 **Clientes** cuyo campo **ID** sea mayor de 3.

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

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

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

Cuando usamos estas funciones con la cláusula **Aggregate** (o con **From**), los cálculos se hacen con los elementos que tenga la colección usada con esas cláusulas. 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 **Integer (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 devuelta por la expresión de consulta, si lo indicamos, ese 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 7.7 tenemos cuatro formas de usar la función **Sum**: las dos primeras son equivalentes, es decir, si no se indica un argumento, se usa la variable usada tras **Aggregate**; en el tercer ejemplo lo que se suma es el valor indicado como argumento, pero la variable no se utiliza, ese valor se sumará tantas veces como elementos tenga la colección, en este ejemplo, el *array nums* tiene 9 elementos; en el cuarto ejemplo, se suma el valor de cada elemento, pero sumándole 2. El resultado de ejecutar este código es 45 para los dos primeros, 27 para el tercero (3 multiplicado tantas veces como elementos tiene el *array*) y en el último, devolverá 63 ( $45 + 9 * 2$ ).

Como ya vimos antes, estas funciones de agregado las podemos usar con una expresión **From**, por ejemplo, usando el código del listado 7.8 obtenemos el mismo valor que en las dos primeras líneas del listado anterior.

```
' Las dos primeras son equivalentes
Dim res1 = Aggregate n In nums Into Sum()
Dim res2 = Aggregate n In nums Into Sum(n)
' Se sumará 3 tantas veces como elementos haya en la colección
Dim res3 = Aggregate n In nums Into Sum(3)
' Se le suma 2 al valor analizado y se devuelve el total
Dim res4 = Aggregate n In nums Into Sum(n + 2)
```

Listado 7.7. El parámetro indicado en la función es el que se sumará para cada uno de los elementos que tenga la colección

```
Dim res5 = (From n In nums).Sum()
```

Listado 7.8. Podemos usar una función de agregado a una expresión From

Aunque en estos casos no podemos usar argumentos en las funciones; al menos con valores “simples” como en el listado 7.7, ya que el valor que tenemos que indicar es la expresión *lambda* que se usará para realizar el cálculo, por ejemplo, si queremos obtener el mismo resultado que la última línea del listado 7.7, tendríamos que escribir algo como el código mostrado en el listado 7.9.

```
Dim res6 = (From n In nums).Sum(Function(n1) n1 + 2)
```

Listado 7.9. Podemos usar argumentos al aplicar una función de agregado a una expresión de consulta, pero en ese caso debe ser la función lambda a usar para realizar la operación

Al ejecutarse la consulta LINQ (**From n In nums**) se genera una colección con todos los números del *array* (al no haber una cláusula **Where** que haga algún tipo de filtro, se incluyen todos los valores), a ese resultado se le aplica la función **Sum**, que en realidad lo que hace es usar la expresión *lambda* que recibe como parámetro para operar con cada uno de los elementos efectuando la operación indicada en la expresión de la función anónima. Está claro ¿verdad? En serio, este tipo de operaciones es mejor no analizarlas, que después acabamos con la cabeza como un bombo, pero es bueno saber, aunque sea un poco, qué es lo que se cuece tras sentencias tan sencillas como la del listado 7.8. En realidad, lo importante es que sepamos que al usar esa función de agregado se suman todos los valores devueltos por la consulta realizada, de los pormenores ya se encarga el compilador, así que, dejemos que sea el compilador el que se siga encargando y sigamos viendo de forma “amigable” cómo utilizar el resto de funciones.

El resto de las funciones de agregado son parecidas a ésta que acabamos de ver, solo que realizando operaciones diferentes. En el listado 7.10 vemos cómo usar esas funciones; en los comentarios indico 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 7.11 (recordemos que la diferencia entre esas dos funciones es el tipo de datos que devuelven).

```
' Calculamos la media, el resultado es 5 (45 / 9)
Dim res1 = Aggregate n In nums Into Average()

' El total de elementos: 9
Dim res2 = Aggregate n In nums Into Count()

' El valor mayor (9)
Dim res3 = Aggregate n In nums Into Max()

' El valor menor (1)
Dim res4 = Aggregate n In nums Into Min()
```

Listado 7.10. Ejemplo de las funciones de agregado

```
Dim res5 = Aggregate n In nums Into Count(n > 3)
```

Listado 7.11. El total de elementos que sean mayores de 3

Tal como vimos en el listado 7.7, 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 indicamos después de **Aggregate** (o después de **From**, si optamos por hacer una consulta a la que aplicar la función de agregado) 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 7.12 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 el código del listado 7.12 se evalúan todos los artículos cuyo precio sea menor de 0.99.

```
Dim res1 = Aggregate a In artículos Into Average(a.PrecioVenta)
Dim res2 = Aggregate a In artículos Into Count(a.PrecioVenta < 0.99D)
Dim res3 = Aggregate a In artículos Into Max(a.PrecioVenta)
Dim res4 = Aggregate a In artículos Into Min(a.PrecioVenta)
Dim res5 = Aggregate a In artículos Into Sum(a.PrecioVenta)
```

Listado 7.12. Funciones de agregado que utilizan una propiedad de una clase

## Las funciones All y Any

Estas dos funciones devuelven un valor **Boolean** 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 cumple la condición o la colección está vacía.

En el listado 7.13 vemos un ejemplo de estas dos funciones.

```
' True si todos son mayor de 3
Dim res1 = Aggregate n In nums Into All(n > 3)
' True si alguno es mayor de 3
Dim res2 = Aggregate n In nums Into Any(n > 3)
```

Listado 7.13. Las funciones All y Any siempre devuelven un valor Boolean

Por supuesto, con estas funciones también podemos usar funciones *lambda* como argumento, particularmente cuando las queremos utilizar con expresiones de consulta. El código del listado 7.14 muestra el equivalente al del listado 7.13.

```
Dim res3 = (From n In nums).All(Function(n) n > 3)
Dim res4 = (From n In nums).Any(Function(n) n > 3)
```

Listado 7.14. También podemos usar funciones lambda con estas funciones de agregado

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 7.15 y 7.16 vemos cómo evaluar el precio de venta y hacer más de una comprobación. En los dos últimos casos, se deben cumplir las dos condiciones, ya que usamos el operador **AndAlso** (también podríamos usar **And**, pero **AndAlso** tiene mejor rendimiento, al igual que ocurre con **OrElse** frente a **Or**).

```
' Devuelve False, ya que no todos los precios son mayores de 0.75
Dim res5 = Aggregate a In artículos Into All(a.PrecioVenta > 0.75D)

' Devuelve True, porque algunos precios son menores de 0.95
Dim res6 = Aggregate a In artículos Into Any(a.PrecioVenta < 0.95D)
```

Listado 7.15. 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
Dim res7 = Aggregate a In artículos
    Into All(a.PrecioVenta > 0.15D AndAlso a.IVA = 16D)

' True si alguno tiene un precio menor de 0.95 y el IVA es 16
Dim res8 = Aggregate a In artículos
    Into Any(a.PrecioVenta < 0.95D AndAlso a.IVA = 16D)
```

Listado 7.16. 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 podemos usar tanto con **From** como con **Aggregate**, y para hacer el filtro tendremos que indicar una condición, que puede ser simple o múltiple (varias condiciones usando los operadores condicionales: **And**, **AndAlso**, **Or**, **OrElse**, **Not**, etc.)

En el listado 7.17 vemos un ejemplo de cómo filtrar una consulta realizada con **Aggregate**. En los ejemplos mostrados en ese listado, ponemos como condición que solo se tengan en cuenta los artículos que tengan un valor 16 en la propiedad **IVA**. En el ejemplo de la función **Any**, el resultado que obtenemos es el mismo que haciendo la comprobación en los argumentos de la función, sin embargo, con la función **All**, al usar **Where**, primero filtramos los elementos que se van a tener en cuenta (todos los que la propiedad **IVA** tenga un valor 16) y, por tanto, la comprobación de si todos cumplen la condición indicada en el argumento de la función no se hará sobre todos los datos de la colección, sino solo en los que hayan pasado por el filtro indicado tras **Where**. Lo mismo ocurre con la función **Sum**, ya que solo se suman los precios de venta de los artículos que cumplan esa condición.

```
' Para que no todos tengan 16 en el IVA
articulos(0).IVA = 7

' Si todos los artículos con el IVA 16
' tienen el precio de venta superior a 0.15
Dim res9 = Aggregate a In articulos
    Where a.IVA = 16D
    Into All(a.PrecioVenta > 0.15D)

' Si hay artículos cuyo IVA sea 16
' y el precio de venta sea menor de 0.95
Dim res10 = Aggregate a In articulos
    Where a.IVA = 16D
    Into Any(a.PrecioVenta < 0.95D)

' La suma total del precio venta de los artículos
' cuyo IVA sea 16
Dim res11 = Aggregate a In articulos
    Where a.IVA = 16D
    Into Sum(a.PrecioVenta)
```

Listado 7.17. Podemos filtrar los datos a utilizar con Aggregate

La cláusula **Where** también la podemos usar con **From**, tal como hemos visto en los primeros ejemplos de este capítulo y veremos más ejemplos en otras ocasiones. Pero para refrescar nuestra memoria, veamos cómo hacer un filtro por más de un valor que esté relacionado con los elementos analizados (condiciones múltiples).

En el listado 7.18 añadimos un nuevo artículo a la colección *articulos*, 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 **And** y **Or**. 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 esas 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.

```
articulos.Add(
    New Artículo("30")
    With {.Descripción = "Leche entera brick",
        .IVA = 7, .PrecioVenta = 0.1D})
Dim res1 = From a In artículos
    Where a.PrecioVenta > 0.45D
    AndAlso (a.Código.StartsWith("0")
    OrElse a.Código.StartsWith("3"))
    Select ID = a.Descripción,
        IVA = a.IVA,
        PVP = a.PrecioVenta
For Each a In res1
    Console.WriteLine("{0}, {1}, {2}",
        a.ID, a.IVA, a.PVP)
Next
```

Listado 7.18. Una consulta con varias condiciones

Con esas condiciones, no se debe mostrar el artículo que hemos añadido al principio del listado 7.18, 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 de **AndAlso** (ver el listado 7.19), 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.

```
Dim res2 = From a In artículos
    Where a.PrecioVenta > 0.45D
    AndAlso a.Código.StartsWith("0")
    OrElse a.Código.StartsWith("3")
    Select ID = a.Descripción,
        IVA = a.IVA,
        PVP = a.PrecioVenta
```

Listado 7.19. La misma consulta que en el listado 7.18, pero sin los paréntesis

Así es **AndAlso** (incluso **And**) 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 7.20, 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")  
    With {.Descripción = "Leche semi desnatada brick", _  
        .IVA = 7, .PrecioVenta = 0.85D})
```

Listado 7.20. 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 7.18 y 7.20 estarán en la colección devuelta por la consulta (ver la figura 7.2 con los artículos que se mostrarán).

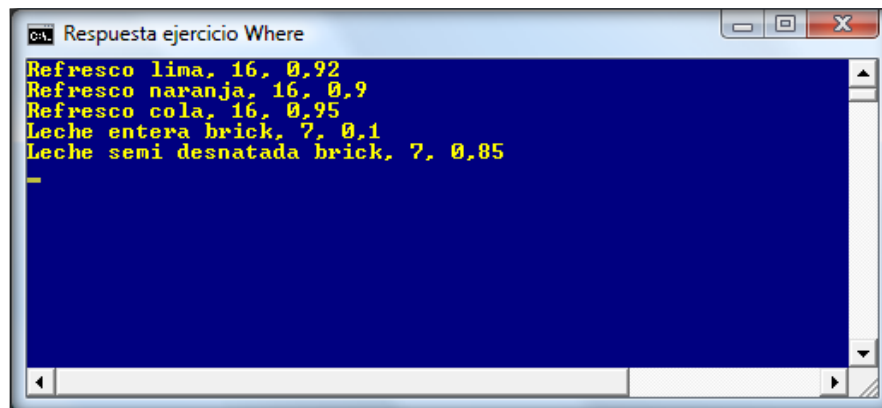


Figura 7.2. Los valores que se mostrarán con el código del ejercicio

## Distinct

Esta cláusula la usaremos cuando queramos eliminar duplicidades en el resultado de la consulta y siempre la usaremos después de **Select**, y en caso de que nuestra consulta no incluya **Select**, la pondremos al final de la misma.

Como sabemos, los campos usados en los valores devueltos por la consulta serán los que indiquemos después de la instrucción **Select**, o la variable usada después de **From** (si no indicamos los campos a seleccionar). Cuando aplicamos la cláusula **Distinct** se tendrán en cuenta esos campos, y las duplicidades que se evaluarán serán sobre esos valores, es decir, si existen duplicidades en campos que no se incluyen como parte del tipo de datos devuelto por la consulta, éstos no se tendrán en cuenta.

Por ejemplo, si queremos saber cada uno de los clientes que han realizado compras de nuestros productos, podemos utilizar **Distinct** para que solo nos devuelva en la consulta los códigos de esos clientes. El código del listado 7.21 nos muestra cómo hacerlo.

```
Dim res1 = From f In facturas Select f.Cliente Distinct
```

Listado 7.21. Los diferentes clientes incluidos en la colección facturas

En la consulta realizada en el listado 7.21, solo devolvemos el código del cliente, por medio de la propiedad **Cliente** del tipo **Factura**. Si en lugar de devolver ese valor hubiésemos indicado todo el dato de la factura, la cláusula **Distinct** no hubiera hecho nada, ya que supuestamente todas las facturas deben tener valores diferentes, al menos así es en este ejemplo concreto. Y es que, **Distinct**, solo filtra la consulta cuando en realidad debe hacerlo, es decir, si lo que indicamos después de **Select** puede contener valores repetidos. Por ejemplo, podríamos querer tener una lista de todos los clientes diferentes, pero solo si las fechas de las facturas también son diferentes, en ese caso, tendríamos que hacer algo como lo mostrado en el listado 7.22.

```
Dim res2 = From f In facturas
            Select Cliente = f.Cliente, _
               Fecha = f.Fecha.ToString("dd/MM/yyyy") _
            Distinct
```

Listado 7.22. Solo las facturas de clientes y fechas distintas

Como vemos en el listado 7.22, he tenido que utilizar el método **ToString** para dar formato a las fechas, bueno, en realidad más que para dar formato, es para filtrar las fechas, ya que si solo hubiese usado **f.Fecha**, se hubiera tenido en cuenta toda la fecha, incluida la “hora” de la factura, que casi con toda seguridad serían todas diferentes, salvo que todas se hubiesen realizado en la misma hora, minuto e incluso segundo. Además, al utilizar un método como parte del resultado de los campos indicados en **Select**, debemos indicar expresamente los nombres de los campos a devolver, ya que el compilador nos avisa de que solo se inferirán los nombres de los campos que no contengan un valor devuelto por un método, tal como vemos en la figura 7.3.

Al igual que el resto de cláusulas de las consultas, la equivalencia con las instrucciones de SQL es más que evidente, aunque se utilicen de forma diferente; esto ya lo hemos comprobado con las instrucciones que llevamos vistas y seguiremos “notando” ese parecido en las instrucciones que siguen a continuación.

```
Dim res2 = From f In facturas _
            Select f.Cliente, _
               f.Fecha.ToString("dd/MM/yyyy") _
            Distinct
```

El nombre de una variable de rango sólo se puede inferir a partir de un nombre simple o completo sin argumentos.

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

## 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 **Order By**, 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 7.23.

```
Dim res1 = From a In artículos
           Where a.PrecioVenta < 0.95 _
           Order By a.Descripción
```

Listado 7.23. Consulta LINQ que devuelve los elementos ordenados por la propiedad Código

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 7.24.

```
Dim res2 = From a In artículos
           Order By a.Descripción _
           Descending
```

Listado 7.24. Esta consulta devuelve los elementos ordenados de mayor a menor

Si queremos que la clasificación sea por varios campos, debemos indicarlos después de **Order By**, 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 7.25 se clasifican los clientes de forma ascendente por el país y de forma descendente por la empresa.

```
Dim res3 = From c In clientes
           Order By c.País Ascending, _
                  c.Empresa Descending
```

Listado 7.25. Podemos ordenar por varios campos

Cuando indicamos varios campos después de **Order By**, el orden de clasificación siempre será de izquierda a derecha. En el código del listado 7.25, primero se clasifican por **País** y después por **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 7.26 tenemos un ejemplo con varios campos y varios criterios.

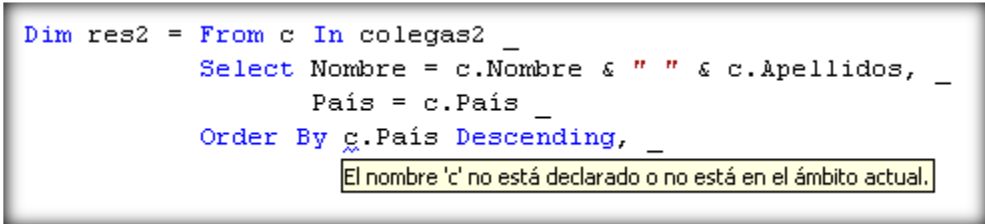
```
Dim res1 = From c In colegas2 _
           Order By c.País Descending, _
                  c.Nombre, _
                  c.Apellidos Descending
```

Listado 7.26. Varios criterios de ordenación

La cláusula **Order By** se puede usar antes o después de **Select**. Si la indicamos antes, solo podremos utilizar los campos de las variables de rango (las indicadas después de **From**), pero si lo utilizamos después de **Select**, tendremos que utilizar las variables que indiquemos en la selección. Por ejemplo, en el listado 7.27 indicamos nombres de nuestra elección para las propiedades del tipo anónimo que se creará como elemento individual del resultado de esa consulta, por tanto, podemos usar esos nombres en la cláusula **Order By**. De hecho, si usamos **Order By** después de **Select** no tendremos acceso a las variables usadas después de **From** (variables de rango), solo las que definamos en **Select**. En cualquier caso, la "pre compilación" de Visual Basic nos ayuda a detectar este tipo de usos inadecuados de las variables de rango, tal como vemos en la figura 7.4.

```
Dim res1 = From c In colegas2 _
           Select Nombre = c.Nombre & " " & c.Apellidos, _
                  País = c.País _
           Order By País Descending, _
                  Nombre
```

Listado 7.27. Si usamos Order By después de Select solo podemos usar las columnas creadas en Select



```
Dim res2 = From c In colegas2 _
           Select Nombre = c.Nombre & " " & c.Apellidos, _
                  País = c.País _
           Order By c.País Descending, _
```

El nombre 'c' no está declarado o no está en el ámbito actual.

Figura 7.4. Las variables de rango no las podemos usar después de Select

## Agrupaciones y combinaciones: Group By y Group Join

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 **Group Join** nos permitirá combinar dos colecciones basadas en las claves que coincidan de éstas.

Para ser precisos, **Group Join** no agrupa datos, sino que los combina, al igual que hace **Join**. De hecho, si buscamos equivalencias entre estas cláusulas de Visual Basic y las instrucciones de una consulta de T-SQL, la cláusula **Join** equivaldría a **INNER JOIN**, mientras que **Group Join** equivale a **LEFT JOIN**.

## Group By

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 7.28.

```
Dim res1 = From f In facturas _
           Group By f.Cliente _
           Into grupoFact = Group
```

Listado 7.28. Las facturas agrupadas por cada uno de los clientes

La variable *grupoFact* contiene cada una de las facturas del cliente examinado. Por tanto, si queremos mostrar cada cliente y a continuación todas las facturas de ese cliente, lo haremos tal como vemos en el listado 7.29.

```
For Each s In res1
    Console.WriteLine("{0} {1}", _
                      s.Cliente, Cliente.Items(s.Cliente).Empresa)
    For Each f In s.grupoFact
        f.CalcularTotal()
        Console.WriteLine(" {0}, {1}, {2} {3}", _
                          f.Numero, f.Fecha.ToString("dd/MM/yyyy"), _
                          f.Items.Count, f.Total)
    Next
Next
```

Listado 7.29. Mostrar los datos de la consulta del listado 7.28

Como podemos suponer por el código usado en el listado 7.29, el tipo de datos de cada elemento de la colección resultante, solo tiene dos propiedades: *Cliente* que es el campo indicado después de **Group By**, y *grupoFact* que es la colección de facturas de ese cliente.

El que el campo (o propiedad) *Cliente* se llame así, es porque el compilador utiliza el nombre de la propiedad usada para agrupar los datos, si en vez de ese nombre quisiéramos que el código del cliente (que es a lo que hace referencia *f.Cliente*) fuese *CodCli*, tendríamos que indicarlo expresamente:

```
Group By CodCli = f.Cliente
```



**Nota**

*En el código del listado 7.29 utilizo la propiedad compartida **Ítems** de un tipo llamado **Cliente** (definido en el proyecto de prueba) para acceder a los datos del cliente indicado como argumento, lo aclaro para que no pensemos que Visual Basic 9.0 es capaz de crear un tipo a partir de un nombre inferido en una consulta. En los proyectos de ejemplo que acompañan al libro se puede ver la definición de esa clase, así como otras clases usadas en los listados de este y el resto de capítulos.*

Si quisiéramos otros datos, tendríamos que haberlos indicado expresamente con una cláusula **Select**, pero siempre con los datos que intervengan en la consulta, que en este caso serían los campos indicados después de **Group By** y lo que indiquemos después de **Into**.

Entre **Group** y **By** podemos indicar los campos que queremos incluir en el grupo generado, si no indicamos nada, se incluirá la variable usada después de **From**, es decir, todas las propiedades del tipo que contiene la colección consultada.

Debido a que el grupo generado en realidad es otra colección, podríamos filtrar los datos de esa colección, aunque en realidad (salvo que indiquemos lo contrario), obtendríamos una colección con todos los datos indicados tanto después de **Group By** como el grupo generado después de **Into** y el nuevo “filtro” que usemos. Esto lo veremos mejor con el código del listado 7.30.

En el listado 7.30, estamos creando una propiedad (la variable **fv**) que contiene todo lo que se incluye en la consulta indicada en el segundo **From**, por supuesto esos datos se basarán en la colección generada por **Group By**.

```
Dim res1 = From f In facturas _
            Group By f.Cliente _
            Into grupoFact = Group _
            From fv In grupoFact _
            Where fv.Items.Count > 5 _
            Select fv, Cliente, grupoFact
```

Listado 7.30. Podemos complicar las cosas todo lo que queramos

Pero eso mismo, además, agrupado de verdad por los clientes (como en el listado 7.28), lo podemos hacer simplificando las cosas de forma que primero obtengamos todas las facturas con más de 5 artículos y agrupados por clientes. Ese código sería el del listado 7.31.

En la figura 7.5 vemos uno de los posibles resultados de ejecutar las consultas de los listados 7.30 y 7.31, el código usado para mostrar esos datos sería el del listado 7.32 (en el que he omitido la asignación a las dos variables que contienen el resultado de cada una de las dos consultas realizadas en los listados 7.30 y 7.31). Como podemos ver en el listado 7.32, para mostrar los datos de la consulta asignada a la variable **res1**, solo tenemos que hacer un bucle, ya que cada uno de los diferentes

clientes que se hayan almacenado en la colección resultante, estarán “desagrupados”. Esto es así, porque el agrupamiento se hizo antes de la segunda consulta. Sin embargo, en la variable *res2* utilizamos un segundo bucle para poder mostrar cada una de las facturas de cada uno de los clientes, que en esta ocasión sí que están agrupados.

```
Dim res2 = From f In facturas
           Where f.Items.Count > 5
           Group By f.Cliente
           Into grupoFact = Group
           Order By Cliente
```

Listado 7.31. También podemos agrupar los resultados de la consulta

```
Console.WriteLine("Resultado de la consulta del listado 30")
For Each s In res1
    Console.WriteLine("{0} {1}",
                      s.Cliente, Cliente.Items(s.Cliente).Empresa)
    s.fv.CalcularTotal()
    Console.WriteLine(" {0}, {1}, {2} {3}",
                      s.fv.Numero, s.fv.Fecha.ToString("dd/MM/yyyy"),
                      s.fv.Items.Count, s.fv.Total)
Next
Console.WriteLine()
Console.WriteLine("Resultado de la consulta del listado 31")
For Each s In res2
    Console.WriteLine("{0} {1}",
                      s.Cliente, Cliente.Items(s.Cliente).Empresa)
    For Each f In s.grupoFact
        f.CalcularTotal()
        Console.WriteLine(" {0}, {1}, {2} {3}",
                          f.Numero, f.Fecha.ToString("dd/MM/yyyy"),
                          f.Items.Count, f.Total)
    Next
Next
```

Listado 7.32. Mostrar los datos de las dos consultas anteriores

En el primer resultado también podemos usar la variable *grupoFact* para recorrer las facturas agrupadas en la primera parte del código del listado 7.30, pero debido a que hemos creado una nueva consulta dentro de la consulta original, volveremos a tener los datos duplicados. En los proyectos de ejemplo que acompañan al libro puede ver ese código, aunque si lo quiere probar, sería similar a la segunda parte del listado 7.32, pero usando la variable *res1* en lugar de *res2*.

## Group Join

La cláusula **Group Join** 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 **Group Join** podríamos hacerlo tal como vemos en el listado 7.33.

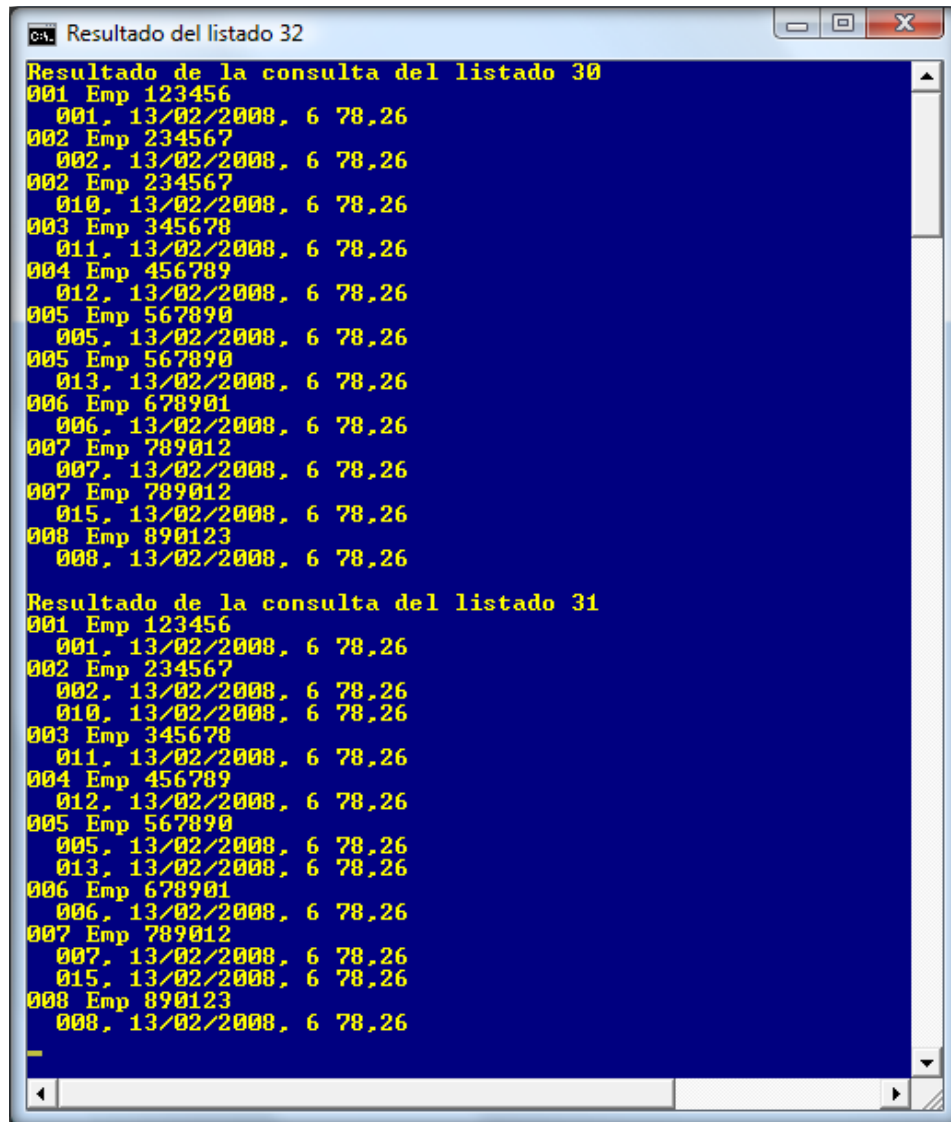


Figura 7.5. Resultado del listado 7.32

```
Dim res1 = From c In clientes _
            Group Join f In facturas _
            On c.Código Equals f.Cliente _
            Into lasFacturas = Group, TotalArt = Sum(f.Items.Count) _
            Select c.Empresa, lasFacturas, TotalArt _
            Order By Empresa, TotalArt
```

Listado 7.33. Una consulta usando Group Join

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 la función de agregado **Sum** nos dará

la suma total de todos los artículos de todas las facturas que ese cliente tenga. Para mostrar los datos obtenidos en esta consulta, podemos usar el código del listado 7.34.

```

For Each s In res1
    Console.WriteLine("{0} factura con {1} artículos en total", _
        s.lasFacturas.Count, s.TotalArt)
    For Each f In s.lasFacturas
        Console.WriteLine("Cliente N°: {0}, {1}", f.Cliente, s.Empresa)
        Console.WriteLine(", Factura: {0} {1:dd/MM/yyyy}", _
            f.Numero, f.Fecha)
        For Each v In f.Items
            Console.WriteLine("  {0}, {1}, {2} {3}", _
                v.Código, v.Descripción, _
                v.PrecioVenta, v.Total)
        Next
    Next
    Console.WriteLine()
Next

```

Listado 7.34. Mostrar los datos de la consulta obtenida en el listado 7.33

## Divide y vencerás: Skip y Take

Si de una consulta de LINQ queremos tomar solo una parte de la misma, utilizaremos las cláusulas **Skip** y **Take**. Después de esas instrucciones 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 7.35 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.

```
Dim res1 = From a In artículos Skip 3
```

Listado 7.35. Skip se saltará los elementos que le indiquemos

En el listado 7.36 vemos cómo usar **Take** para tomar solo los tres primeros elementos de la colección.

```
Dim res2 = From a In artículos Take 3
```

Listado 7.36. Take tomará el número de elementos que indiquemos a continuación

También podemos utilizar estas dos cláusulas para que se salte un número de elementos y a partir de esa posición tome los que indiquemos. Por ejemplo, en el listado 7.37 tomamos 2 elementos a partir del tercero (saltamos dos y tomamos dos).

```
Dim res3 = From a In artículos Skip 2 Take 2
```

Listado 7.37. Podemos usar Skip y Take en la misma consulta

Estas dos cláusulas las podemos usar para realizar paginaciones de datos, por ejemplo, en el listado 7.38 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**.

```
Dim actual = 0
Dim total = 5
Dim pg = 1

Do
    Dim res1 = From f In facturas Skip actual Take total
    If res1.Count = 0 Then
        Exit Do
    End If

    Console.WriteLine("Página {0}", pg)
    pg += 1
    For Each f In res1
        Console.WriteLine("{0}, {1} {2:dd/MM/yyyy}", _
            f.Numero, f.Cliente, f.Fecha)
    Next
    Console.WriteLine()
    actual += total
Loop
```

Listado 7.38. Podemos usar Skip y Take para realizar paginaciones

Con estas cláusulas podemos usar también la instrucción **While**, de forma que se salte o tome (según usemos **Skip** o **Take** respectivamente) los elementos mientras se cumpla la condición indicada después de **While**. En cuanto esa condición deje de cumplirse, se utilizarán los elementos que queden. Por ejemplo, si utilizamos el código del listado 7.39, 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).

```
Dim res1 = From a In artículos Skip While a.PrecioVenta > 0.9
```

Listado 7.39. Skip While 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 7.40.

```
Dim res2 = From a In artículos Where a.PrecioVenta <= 0.9
```

Listado 7.40. Si realmente queremos todos los elementos que cumplan una condición, es preferible usar Where

**Take While** 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 **Skip While**, 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. Por ejemplo, con el código del listado 7.41, 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.

```
Dim nums() As Integer = {1, 9, 8, 2, 5, 7, 4, 3, 6}
Dim res3 = From n In nums Take While n > 3
```

Listado 7.41. Take While 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 7.42, 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.

```
Dim res4 = From n In nums Skip 4 Take While n > 3
```

Listado 7.42. 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 Visual Basic 9.0 la quitará solo si esa instrucción no la usamos dentro de una consulta de LINQ, ya que ha “renacido” esta instrucción para permitirnos crear alias de variables dentro de una consulta. Ese alias lo 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 7.43 tenemos un ejemplo algo rebuscado de cómo usar la cláusula **Let** para hacer asignaciones intermedias en una misma consulta.

```
Dim res1 = From f In facturas _
           Let arts = f.Items _
           Let res = From a In arts _
                   Where a.Código.StartsWith("0")
```

Listado 7.43. Let permite crear variables intermedias en una consulta

En el código del listado 7.43, 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 ítems 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 7.44 vemos cómo mostrar algunos de los datos contenidos en la colección resultante de la consulta.

```
For Each r In res1
    Console.WriteLine(r.f.Numero)
    For Each a In r.res
        Console.WriteLine(" {0}", a)
    Next
Next
```

Listado 7.44. Mostrando los datos de la consulta del listado 7.43

También podemos usar **Let** para crear un valor intermedio que después podemos usar en la cláusula **Where**, como en el código del listado 7.45 en el que se asigna una variable llamada **par** que contendrá un valor verdadero si el número examinado es par.

```
Dim res1 = From n In nums _
           Let par = n Mod 2 = 0 _
           Where par = True _
           Select "El número " & n & " es par"
```

Listado 7.45. Podemos usar Let en construcciones simples

En el ejemplo del listado 7.45 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 7.46.

```
Dim res2 = From n In nums _
           Let par = n Mod 2 = 0 _
           Let parText = If(par, "par", "impar")
           Select "El número " & n & " es " & parText
```

Listado 7.46. Let lo podemos usar para crear variables a partir de otras ya creadas anteriormente

Y para terminar este repaso a las instrucciones de consultas que incluye Visual Basic 9.0, veamos en el listado 7.47 cómo usar la cláusula **Let** para asignar el valor de una función de agregado.

```
Dim res1 = From f In facturas _  
           Let totArt = Aggregate a In f.Items _  
           Into Sum(a.Cantidad * a.PrecioVenta)
```

Listado 7.47. 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.

### 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 Basic 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 **AndAlso** que es más eficiente) y la comparación será: precio  $\geq 0.75$  y precio  $\leq 0.95$ . A esta doble comparación le seguirá una instrucción **Or** (u **OrElse**) comprobando el valor de **IVA**. El listado 7.48 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 **OrElse** 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** (o **AndAlso**), 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 7.49.



```
Dim res3 = From a In artículos _
           Where a.PrecioVenta >= 0.75
           AndAlso a.PrecioVenta <= 0.95 _
           OrElse a.IVA = 7 _
           Select ID = a.Descripción,
                  IVA = a.IVA, _
                  PVP = a.PrecioVenta
```

Listado 7.48. Respuesta al ejercicio propuesto en Where

```
Dim res4 = From a In artículos _
           Where a.IVA = 7 _
           OrElse a.PrecioVenta >= 0.75 _
           AndAlso a.PrecioVenta <= 0.95
           Select ID = a.Descripción, _
                  IVA = a.IVA, _
                  PVP = a.PrecioVenta
```

Listado 7.49. 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 **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 7.50.

```
Where (a.PrecioVenta >= 0.75 _
AndAlso a.PrecioVenta <= 0.95)
OrElse a.IVA = 7

Where a.IVA = 7 _
OrElse (a.PrecioVenta >= 0.75
AndAlso a.PrecioVenta <= 0.95) _
```

Listado 7.50. Cuando usemos comparaciones con AND, mejor ponerlas entre paréntesis

## Funciones agregadas personalizadas

Las funciones agregadas (o métodos de agregación) las crearemos como métodos extensores de la interfaz *generic* **IEnumerable(Of 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 7.51 vemos la definición de esa función, que como podemos comprobar no es más que un método extensor de la interfaz **IEnumerable(Of Integer)**, 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 7.51, 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 7.52 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 cláusula **Aggregate** con la función **Sum**, y 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 7.51 (o el del listado 7.52), lo haremos como con cualquier otra función de agregado. En el listado 7.53 vemos varias formas de utilizar el método **SumaPares** (por supuesto, esta función solo la podremos usar con colecciones de tipo entero).

```
Option Strict On
Option Infer On

Imports System
Imports System.Runtime.CompilerServices
Imports System.Linq
Imports System.Collections.Generic
Public Module ExtensionesAgregados
    <Extension()> _
    Public Function SumaPares(ByVal source As IEnumerable(Of Integer) _
        ) As Integer
        Dim t As Integer = 0
        For Each n In source
            If n Mod 2 = 0 Then
                t += n
            End If
        Next

        Return t
    End Function
End Module
```

Listado 7.51. Método extensor para usar como función de agregado

```
<Extension()> _  
Public Function SumaImpares(ByVal source As IEnumerable(Of Integer) _  
                             ) As Integer  
    Dim res2 = Aggregate n In source _  
                      Where n Mod 2 <> 0 _  
                      Into Sum()  
  
    Return res2  
End Function
```

Listado 7.52. Función de agregado que internamente utiliza la cláusula Aggregate

```
Dim nums() As Integer = {1, 2, 3, 4, 5, 6, 7, 8, 9}  
  
Dim resPar = Aggregate n In nums _  
                  Into SumaPares()  
  
Dim resPar1 = Aggregate n In nums _  
                Where n > 3 _  
                Into SumaPares()  
  
Dim resPar2 = (From n In nums _  
               Where n > 3).SumaPares  
Dim resPar3 = nums.SumPares
```

Listado 7.53. Varios ejemplos que utilizan la función de agregado del listado 7.51

# Capítulo 8

## Visual Basic y LINQ (III)

---

### Visual Basic y LINQ to XML

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 Visual Basic, ya que este lenguaje se compenetra totalmente con todo lo relacionado con esta tecnología, de forma que incluso permite el uso de literales XML en el código.

### Integración total de XML en Visual Basic 9.0

Lo interesante de *LINQ to XML* desde el punto de vista del programador de Visual Basic es que esa tecnología se ha integrado totalmente en el lenguaje, tanto, que podemos usar literales XML en el código, desde crear elementos XML individuales hasta crear documentos completos de XML. Y para que comprendamos qué significa esta integración, un ejemplo vale más que mil palabras.

En el listado 8.1 vemos cómo podemos utilizar un documento XML directamente en el código de Visual Basic.

```
Dim docXML1 =  
<?xml version="1.0"?>  
<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>  
</colegas>
```

Listado 8.1. Un documento XML usado directamente en el editor de Visual Basic

El código del listado 8.1 creará un objeto del tipo **XDocument**. Este documento XML lo podremos usar como cualquier otro documento de ese tipo, es decir, el que se haya creado directamente no influye a la hora de acceder a los elementos que define. Si ese documento lo hubiésemos creado a partir de un archivo XML existente, no habría diferencia a la hora de recorrer los elementos. Por ejemplo, en el listado 8.2 vemos la forma de cargar el documento a partir de un archivo **.xml** existente usando el método **Load** de la clase **XDocument**, que está definida en el espacio de nombres **System.Xml.Linq** (ese archivo contiene exactamente lo mismo que el código XML mostrado en el listado 8.1).

```
Dim docXML1 = XDocument.Load("colegas.xml")
```

Listado 8.2. También podemos cargar un documento existente

En cualquiera de estos dos casos, podemos recorrer los elementos de ese documento tal como vemos en el listado 8.3.

```
For Each c In docXML1...<colega>  
    Console.WriteLine("{0} <{1}>", c.<nombre>.Value, c.<correo>.Value)  
Next
```

Listado 8.3. La forma de recorrer los elementos es independiente de cómo hayamos creado el documento XML

En este caso en particular, en el que el documento XML empieza con el literal **<?xml version="1.0"?>**, el bucle para recorrer los elementos lo podemos hacer tal como vimos en el listado 8.3 o de la forma mostrada en el código del listado 8.4.

```
For Each c In docXML1.<colegas>.<colega>  
    Console.WriteLine("{0} <{1}>", c.<nombre>.Value, c.<correo>.Value)  
Next
```

Listado 8.4. Otra forma de recorrer los elementos del documento mostrado en el listado 8.1

Si ese documento (o los literales usados directamente en el código) no incluyen el elemento **<?xml>**, el código del listado 8.4 no nos mostraría nada. Sin embargo, usando el código del listado 8.3 siempre funcionará. Para comprenderlo mejor, en el listado 8.5 tenemos otra forma de crear el documento XML (que en realidad no es un documento, sino un elemento) y cómo recorrer cada uno de los elementos individuales que contiene.

La variable **docXML2**, en realidad, es del tipo **XElement**, ya que un documento XML debe incluir el elemento **<?xml>** que lo define como documento (tal como vimos en el listado 8.1). Al menos así es como el compilador de Visual Basic infiere estos literales XML. Por supuesto, este “pequeño detalle” solo debemos tenerlo en cuenta si definimos expresamente el tipo de datos que contendrá el XML que asignemos a la variable, por ejemplo, si queremos que una variable esté definida como documento o elemento XML a nivel de tipo, ya que así, el compilador no infiere automáticamente el tipo de datos de las variables.

```

Dim docXML2 = _
    <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>
    </colegas>

For Each c In docXML2...<colega>
    Console.WriteLine("{0} <{1}>", c.<nombre>.Value, c.<correo>.Value)
Next

```

Listado 8.5. Otra forma de crear un documento y mostrar los elementos que contiene

## Continuador de líneas y nombres XML

Antes de entrar en más detalles, resaltar un par de cosas:

La primera es que no tenemos que usar el continuador de líneas para utilizar los literales de XML directamente en el código, es decir, podemos abrir un documento XML, copiar el contenido y pegarlo directamente en el editor de Visual Basic 2008. En realidad, el continuador de líneas solo lo tendremos que usar después de la definición de la variable, y siempre y cuando el código XML aparezca en la siguiente línea después de la definición de la variable.

La segunda es que estamos trabajando con XML y ese lenguaje, a diferencia de Visual Basic, distingue entre mayúsculas y minúsculas, por tanto, cuando usemos literales XML siempre debemos tener en cuenta que no es lo mismo <Nombre> que <nombre>.

## Elementos XML

Como sabemos, el compilador de Visual Basic 9.0 es capaz de inferir el tipo de datos que asignamos a una variable, y con los literales de XML no es una excepción. Tal como vimos en el listado 8.1, el

código XML que asignamos a la variable de ese listado es del tipo documento XML, particularmente del tipo **XDocument**. Por supuesto, esa inferencia de tipos debe ser a nivel local, es decir, si ese documento XML lo quisiéramos asignar a una variable a nivel de tipo, tendríamos que indicar el tipo de datos de forma explícita, porque la inferencia de tipos se hace solo en modo local, es decir, dentro de un procedimiento.

El otro tipo de datos relacionado con los literales XML que se puede inferir es el tipo **XElement** (también definido en el espacio de nombres **System.Xml.Linq**), y en esta ocasión, en lugar de hacer referencia a un documento completo, solo contiene un elemento individual (o a una serie de elementos individuales).

En el listado 8.6 tenemos un ejemplo de la asignación de un elemento XML.

```
Dim elemento = _
    <colega>
        <nombre>Pepe</nombre>
        <correo>pepe@nombres.com</correo>
        <telefono tipo="celular">666777888</telefono>
        <telefono tipo="casa">952520011</telefono>
    </colega>
```

Listado 8.6. Además de documentos XML también podemos asignar elementos individuales

## Diferencia entre documentos, elementos y atributos XML

No voy a entrar en detalles “profundos” del formato usado por XML, pero solo para aclarar conceptos diré que un documento XML normalmente está formado por un elemento principal (elemento raíz), digamos que sería el equivalente a una colección, que en nuestro código de ejemplo es el elemento raíz **<colegas>**. Ese elemento raíz, a su vez, tiene uno o más elementos (los elementos contenidos en esa colección), en nuestro ejemplo sería cada uno de los indicados con **<colega>**. Cada elemento, a su vez, tiene ciertas propiedades o subelementos; siguiendo con nuestro ejemplo, las “propiedades” de cada elemento **<colega>** en realidad son también elementos, como el nombre (**<nombre>**), el correo (**<correo>**), etc. Si uno de esos elementos tiene, a su vez, propiedades, éstas se indican con atributos XML, por ejemplo, el elemento **<telefono>** tiene un atributo que indica el tipo de teléfono que representa; en nuestro código de ejemplo sería el atributo “**tipo**”.

En este “mundillo” de XML en realidad todos son elementos que pueden estar contenidos en otros elementos, además de poder tener atributos. Por ejemplo, el elemento raíz (que debe ser único en un documento XML) tiene varios elementos, y cada uno de ellos, a su vez, tiene otros elementos. La pertenencia a un elemento o a otro se indica por el lugar que ocupa. Es decir, un elemento está contenido dentro de otro si lo indicamos antes del cierre del elemento, por ejemplo, el elemento raíz se cierra con **</colegas>**, por tanto, todo lo que haya entre **<colegas>** y **</colegas>** formará parte del elemento raíz del documento XML. Lo mismo ocurre con cada subelemento **<colega>**, todo lo que haya hasta encontrar el cierre del elemento (**</colega>**) pertenecerá a ese elemento individual.

Si necesitamos indicar atributos, éstos los pondremos dentro del elemento de apertura, como es el caso del atributo **tipo** en el elemento **telefono**. Y normalmente lo indicaremos asignándole un valor que debe estar entrecomillado. Esos atributos pueden aparecer en cualquier elemento y puede haber más de uno.

Todo esto viene a cuento de que si no formamos correctamente los elementos y no indicamos de forma adecuada los atributos, el compilador de Visual Basic se quejará, indicándonos que no lo estamos haciendo bien. En realidad, el compilador no nos indicará dónde está el error, solo que si el código XML que hemos usado no está bien cerrado, el IDE de Visual Basic entenderá que ese código continúa después del código real y... bueno... nos avisará de que falta algo, como es el caso de la figura 8.1 en la que hemos pegado un código XML sin el cierre del elemento raíz, y el compilador nos avisa de que algo hay mal en nuestro código.

Aquí, la única solución es cerrar adecuadamente el elemento raíz, pero en otras ocasiones, el editor de Visual Basic 2008 arreglará automáticamente los elementos que no cerremos adecuadamente, intentando emparejar los elementos de apertura y cierre. En cualquier caso, siempre nos mostrará un error cuando esos elementos no estén correctamente emparejados.

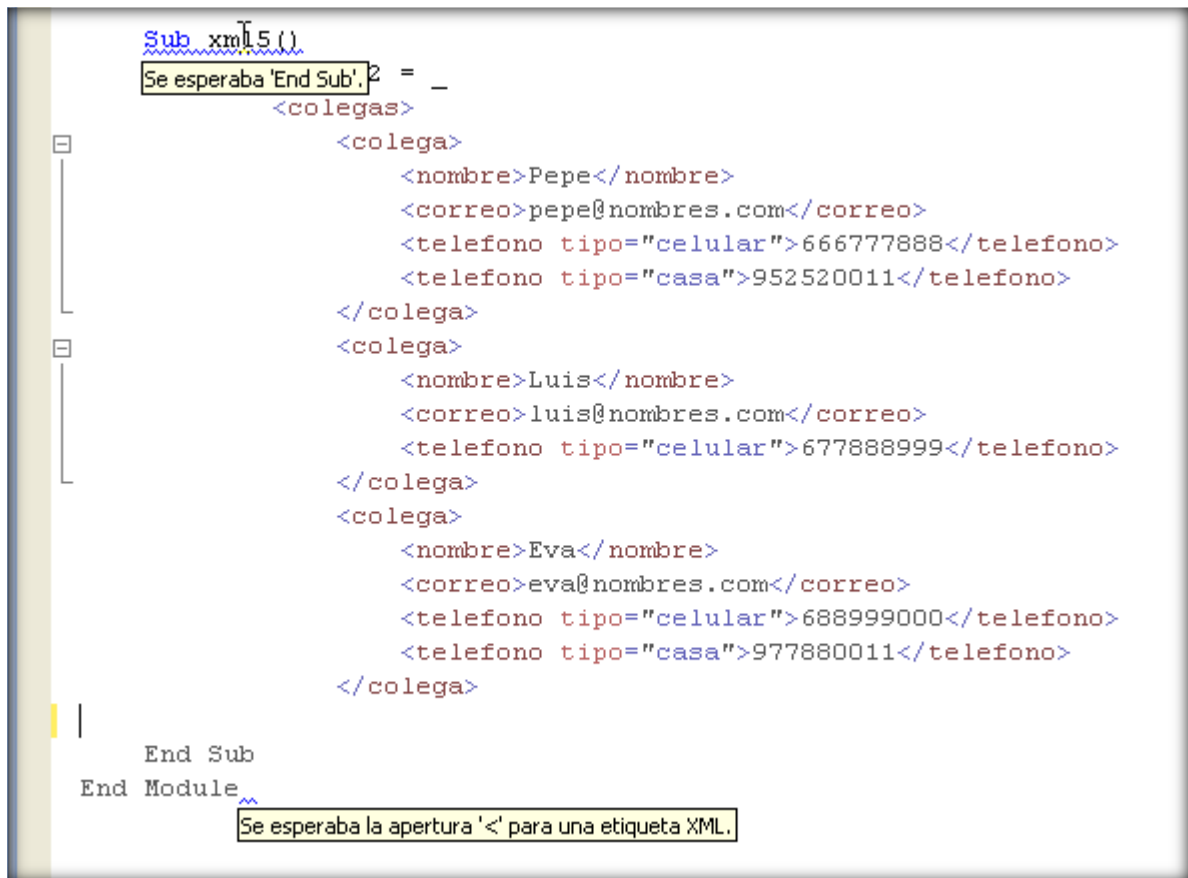


Figura 8.1. Errores en el IDE al no cerrar correctamente los elementos XML



## Expresiones incrustadas en los literales XML

Otra de las características que ofrece el uso de los literales XML desde Visual Basic es que podemos indicar valores al estilo de como se hace en ASP o ASP.NET, es decir, usando expresiones que empiezan con `<%=` seguidas del valor a asignar y que terminan con `%>`.

El uso de estas expresiones incrustadas (que es como las denomina la documentación en castellano de Visual Studio 2008), nos permite asignar valores a los elementos XML que no tienen porqué ser constantes, sino que podemos crear elementos usando los valores de variables. Por ejemplo, en el listado 8.7 asignamos los datos de un elemento a partir de los valores de unas variables a las que en otra parte de ese procedimiento le hemos dado los valores que queremos que tengan esos elementos XML.

Como vemos en el listado 8.7, los atributos no tenemos que indicarlos entrecomillados, si lo hiciéramos, recibiríamos un mensaje de error indicando que no debemos usar las comillas, ya que el compilador convertirá ese valor en una cadena cuando lo asigne al código XML que generará.

```
Dim nombre = "Guille"
Dim correo = "guille@nombres.com"
Dim telf1 = "678999888"
Dim telf1_tipo = "celular"

Dim elemento = _
    <colega>
        <nombre><%= nombre %></nombre>
        <correo><%= correo %></correo>
        <telefono tipo=<%= telf1_tipo %><%= telf1 %></telefono>
    </colega>
```

Listado 8.7. Podemos usar expresiones al estilo de ASP.NET para asignar valores en tiempo de ejecución

## Acceder a los elementos y atributos de las variables de tipo XML

Algo que seguramente haremos cuando tengamos los datos XML en variables, ya sean documentos o elementos individuales, es acceder a los distintos valores que contienen. Como ya vimos en los listados 8.3 a 8.5, para acceder a un elemento, simplemente indicaremos el nombre del elemento de la misma forma que está definido en el código XML, es decir, si queremos mostrar el nombre del elemento creado en el código del listado 8.7, lo haremos de esta forma: **elemento.<nombre>.Value**, es decir, después del nombre de la variable, indicamos el nombre del elemento a mostrar, y para que realmente se muestre el valor que contiene ese elemento, debemos usar la propiedad **Value**, ya que si no usáramos expresamente esa propiedad, lo que nos mostraría sería el nombre “interno” de esa clase.

Si lo que queremos mostrar es el contenido del atributo **tipo**, lo haremos de esta forma: **elemento.<telefono>.@tipo**. En este caso, no tenemos que usar **Value**, ya que el atributo siempre es de tipo **String**.

Así es como normalmente mostraremos los valores de los elementos XML que tenemos creados en el código, aunque para mostrar esos valores tendremos que “recordar” cuáles son los elementos y los atributos que tenemos en el código XML... o no... Veamos cómo podemos hacer que IntelliSense nos ayude con los nombres de los elementos y atributos que definen el documento (o elemento) XML que estamos usando.

## Habilitar IntelliSense en el código XML integrado

Si queremos que el entorno integrado de Visual Basic 2008 nos muestre por medio de IntelliSense los elementos a los que podemos acceder, tendremos que agregar un archivo de esquema (.xsd) al proyecto que tenemos abierto en el IDE de Visual Basic.

Ese archivo de esquema debe tener las definiciones de los elementos que estamos usando en nuestro código de XML. La forma más fácil de crear el esquema es agregando un documento XML con los datos que vamos a manipular (por ejemplo, creando directamente un archivo XML en el IDE de Visual Basic). Una vez abierto el documento, en el menú **XML** seleccionamos **Crear esquema**, lo guardamos y lo agregamos al proyecto actual. A partir de ese momento, cada vez que accedamos a las variables que tengan elementos XML, IntelliSense nos mostrará qué elementos o atributos podemos utilizar, tal como vemos en la figura 8.2.

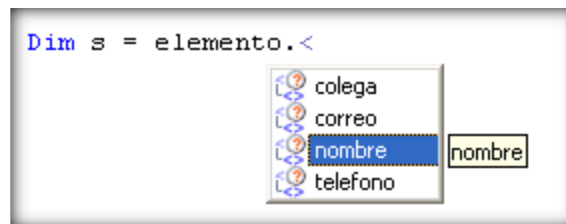


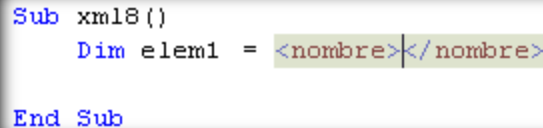
Figura 8.2. Podemos hacer que IntelliSense muestre los elementos de nuestro código XML

## IntelliSense siempre nos facilita la escritura de código XML

Independientemente de que agreguemos un archivo con la definición del esquema XML de los elementos que estamos usando, cuando escribimos código XML en el entorno de desarrollo de Visual Basic 2008, éste siempre nos ayudará a completar el código de XML que escribamos.

Por ejemplo, si estamos creando un elemento XML, cada vez que terminemos de escribir el inicio de un elemento (por ejemplo `<nombre>`), el editor de Visual Basic añadirá el elemento de cierre, en este ejemplo añadirá automáticamente `</nombre>` y dejará el cursor en el sitio adecuado para que empecemos a escribir.

Aunque en la figura 8.3 no se vea expresamente lo comentado, podemos imaginar que el IDE ha generado automáticamente la segunda parte del código (el cierre del elemento) y que ha dejado el cursor preparado para que empecemos a escribir en el contenido de ese elemento.



```
Sub xml18()  
    Dim elem1 = <nombre></nombre>  
End Sub
```

Figura 8.3. IntelliSense nos ayuda en la escritura de código XML

En cualquier caso, esto no es una nueva característica de Visual Basic 2008, ya que en la versión anterior del entorno de desarrollo también estaba este automatismo, aunque solo para los archivos de tipo XML, ya que en las versiones anteriores no podíamos usar literales XML directamente en un archivo de código de Visual Basic.

## Consultas LINQ en elementos XML

Como era de esperar, también podemos realizar consultas LINQ en variables que contienen valores XML, y tal como vimos en los listados anteriores, la forma de acceder a los elementos que contienen la variable es usando tres puntos seguidos y a continuación el elemento al que queremos acceder.

En el listado 8.8 tenemos un ejemplo en el que recorremos todos los valores de los elementos **<colega>**, de esa forma, podemos seleccionar cualquiera de los elementos o atributos que contiene cada uno de los nodos del código XML (en este ejemplo, la variable *docXML* contiene el mismo código que vimos en el listado 8.1).

```
Dim res1 = From c In docXML...<colega> _  
           Select c.<nombre>.Value, _  
                 c.<telefono>.@tipo, _  
                 c.<telefono>.Value
```

Listado 8.8. Una consulta LINQ sobre el contenido XML de una variable definida como en el listado 8.1

Cuando utilizamos los tres puntos seguidos, en realidad podemos indicar cualquiera de los elementos descendientes del documento XML, por ejemplo, si queremos acceder solo a los teléfonos cuyo tipo sea celular, podemos hacerlo como en el listado 8.9.

```
Dim res2 = From c In docXML...<telefono> _  
           Where c.@tipo = "celular"
```

Listado 8.9. En la consulta LINQ podemos acceder a cualquier elemento o sub-elemento del código XML

El problema es que, evidentemente, solo podemos acceder al valor del número de teléfono, ya que la variable usada para la consulta solo contendrá la información del elemento que hemos indicado des-

pués de la instrucción **In**. Si quisiéramos acceder al resto de elementos del nodo en el que está el teléfono analizado, podríamos usar la propiedad **Parent** y así poder incluir en la selección los valores que creamos conveniente. En el listado 8.10 vemos un ejemplo en el que podemos incluir el nombre al que corresponde ese teléfono.

```
Dim res3 = From c In docXML...<telefono> _
           Where c.@tipo = "celular" _
           Select c.Parent.<nombre>.Value, c.Value
```

Listado 8.10. Cuando descendemos demasiado, podemos ascender por medio de Parent

Sabiendo que podemos usar la propiedad **Parent** para acceder al elemento que contiene al que estamos examinando, y teniendo solo el código del listado 8.9, al recorrer cada uno de los elementos de la colección generada por la consulta, podríamos usar el código del listado 8.11 para mostrar el nombre del “colega” al que pertenece cada uno de los teléfonos incluidos en la consulta.

```
For Each r In res2
    Console.WriteLine("{0} {1}", r.Parent.<nombre>.Value, r.Value)
Next
```

Listado 8.11. La propiedad Parent de los elementos XML nos permite acceder al elemento que contiene al elemento actual

## Consultas LINQ para generar elementos XML

Otra forma de utilizar las consultas de LINQ es para generar elementos XML. Por ejemplo, si tenemos una colección de “colegas” y queremos crear un elemento XML con todos los datos de esa colección, podemos hacer algo como el código del listado 8.12. En el que podemos apreciar que en la cláusula **Select** podemos indicar literales XML y, por medio de las expresiones incrustadas, asignamos los valores que cada elemento o atributo deba tener.

```
Dim elem1 = From c In colegas _
            Select <colega>
                <nombre><%= c.Nombre %></nombre>
                <telefono tipo=<%= c.Telefono.Tipo %>>
                    <%= c.Telefono.Numero %></telefono>
                <correo><%= c.Correo %></correo>
            </colega>
```

Listado 8.12. En la cláusula Select también podemos usar literales XML

Y si en lugar de crear elementos XML quisiéramos crear un documento XML en toda regla, es decir, con el atributo **<?xml>** y el nodo raíz **<colegas>**, tendríamos que escribir un código como el del listado 8.13.

```

Dim elem2 = _
  <?xml version="1.0"?>
  <colegas><%=
    From c In colegas _
    Select <colega>
      <nombre><%= c.Nombre %></nombre>
      <telefono tipo=<%= c.Telefono.Tipo %>
        <%= c.Telefono.Numero %></telefono>
      <correo><%= c.Correo %></correo>
    </colega> %>
  </colegas>

```

Listado 8.13. La expresión a incrustar en el código XML puede ser cualquier expresión válida

Debemos tener en cuenta que la variable *elem1* es del tipo **IEnumerable(Of XElement)**, mientras que la variable *elem2* es del tipo **XDocument**. Es decir, en el primer caso simplemente se añaden elementos individuales a una colección, pero en el segundo, lo que se hace es crear un documento XML parecido al del listado 8.1.

## Consultas LINQ para realizar transformaciones

Algo que también es habitual realizar con el contenido de un archivo XML es hacer lo que se conoce como transformación XML, lo habitual es crear un archivo con la extensión **.xsl** en el que se codifica cómo queremos mostrar el contenido del archivo XML. Realizar esa misma tarea por medio de una consulta LINQ es muy sencillo usando los literales XML, por ejemplo, si queremos mostrar los datos de los colegas que tenemos en el código del listado 8.1 de una forma “amigable”, lo podemos hacer tal como vemos en el listado 8.14.

```

Dim html = _
  <html>
  <body>
    <%= From c In docXML...<colega> _
    Select <div>
      <h2>Nombre: <%= c.<nombre>.Value %></h2>
      <h3>Correo: <%= c.<correo>.Value %></h3>
      <h3>Teléfono: <%= c.<telefono>.Value %></h3>
      <hr/>
    </div> %>
  </body>
</html>

```

Listado 8.14. Usando una consulta LINQ y los literales XML es fácil generar un listado amigable

El contenido de la variable *html* (que es de tipo **XElement**) la podemos guardar como un archivo con la extensión **.htm** y tendríamos un documento HTML con los datos de todos los colegas de la variable *docXML*. Para guardar ese contenido solo tendríamos que llamar al método **Save** tal como vemos en el listado 8.15.

```
html.Save("Los_colegas.htm")
```

Listado 8.15. Guardamos el contenido del objeto html

## Importar espacios de nombres XML

Cuando definimos elementos XML, éstos pueden pertenecer a un espacio de nombres concreto (si no indicamos un espacio de nombres específico, siempre se utiliza uno genérico a nivel global de la aplicación, lo mismo que ocurre con todas las aplicaciones de Visual Basic), por ejemplo, cuando creamos aplicaciones WPF (*Windows Presentation Foundation*), siempre se definen dos espacios de nombres, uno genérico y otro que suele llevar el prefijo **x** (ver el listado 8.16). Si en esa aplicación queremos agregar otros espacios de nombres, los tendremos que indicar en el nodo raíz; al agregar esos espacios de nombres, podremos acceder a los elementos definidos en él simplemente anteponiendo el alias usado.

```
<Window x:Class="Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Window1" Height="300" Width="300">
```

Listado 8.16. Las aplicaciones WPF siempre definen dos espacios de nombres para usar en el código XAML

Por ejemplo, si quisiéramos usar en esa aplicación tipos de datos que hemos definido en otro ensamblado, podríamos agregar una importación al espacio de nombres en el que están definidos esos tipos, de esa forma podríamos hacer referencia a éstos directamente en el código XAML. Si ese ensamblado que queremos importar se llama *NovedadesVB9\_LINQ\_03*, lo podríamos hacer tal como vemos en el listado 8.17. Una vez creado ese alias (*linq3* en nuestro ejemplo), podremos acceder a los elementos que define anteponiendo ese alias, igual que vemos en el contenido del objeto **Label** del listado 8.17.

```
<Window x:Class="Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:linq3="clr-namespace:NovedadesVB9 LINQ 03; assembly=NovedadesVB9 LINQ 03"
  Title="Window1" Height="300" Width="300">
  <Grid>
    <Label>
      <linq3:Colega Nombre="Guille"></linq3:Colega>
    </Label>
  </Grid>
</Window>
```

Listado 8.17. En el código XAML de una aplicación WPF podemos definir espacios de nombres para acceder a ensamblados externos

En realidad, no es necesaria tanta complicación, ya que, aunque siempre podemos indicar los espacios de nombres como parte del elemento raíz del código XML, en Visual Basic podemos usar la instrucción **Imports** para importar espacios de nombres XML. La forma de hacerlo es como siem-

pre que usemos esa instrucción para importar espacios de nombres, pero usando la sintaxis propia de XML, tal como vemos en el listado 8.18.

```
Imports <xmlns:ns="http://MiEspacio">
```

Listado 8.18. Importación de un espacio de nombres XML

Una vez que tenemos definido ese espacio de nombres, podemos usar el prefijo para definir elementos XML que pertenezcan a ese espacio de nombres, por ejemplo, el código del listado 8.19 define un documento XML en el que los elementos usados son los definidos en el espacio de nombres que hemos importado en el código del listado 8.18.

```
Private docXML_ns As XDocument = _
    <?xml version="1.0"?>
    <ns:colegas>
        <ns:colega>
            <ns:nombre>Pepe</ns:nombre>
            <ns:correo>pepe@nombres.com</ns:correo>
            <ns:telefono tipo="celular">666777888</ns:telefono>
        </ns:colega>
        <ns:colega>
            <ns:nombre>Luis</ns:nombre>
            <ns:correo>luis@nombres.com</ns:correo>
            <ns:telefono tipo="celular">677888999</ns:telefono>
        </ns:colega>
    </ns:colegas>
```

Listado 8.19. Código XML que utiliza el espacio de nombres importado en el listado 8.18

Si mostramos el contenido de la variable *docXML\_ns*, obtendremos el resultado mostrado en la figura 8.4, ya que el compilador siempre genera el código XML con los espacios de nombres que correspondan a ese documento indicándolos dentro del propio código. Esto es así porque en realidad XML no sabe nada de importaciones externas de espacios de nombres, por tanto, la información debe estar “incrustada” en el propio código XML.

Por supuesto, si queremos acceder a los elementos de la variable *docXML\_ns*, tendremos que usar el alias del espacio de nombres que contiene los elementos, tal como vemos en el listado 8.20.

Y si queremos que IntelliSense sepa qué elementos define ese documento XML, tendremos que crear el archivo de esquema y agregarlo al proyecto, tal como ya expliqué anteriormente.

Sólo comentar que las importaciones de espacios de nombres XML se rigen por las mismas reglas que para las importaciones de espacios de nombres de .NET, es decir, la forma de declararlos, el ámbito que tienen, etc., no se diferencia de si es una importación de un espacio de nombres definido en un ensamblado de .NET o es un espacio de nombres XML. Incluso los podemos definir a nivel de archivo o a nivel de proyecto (agregándolo en las importaciones de las propiedades del proyecto).

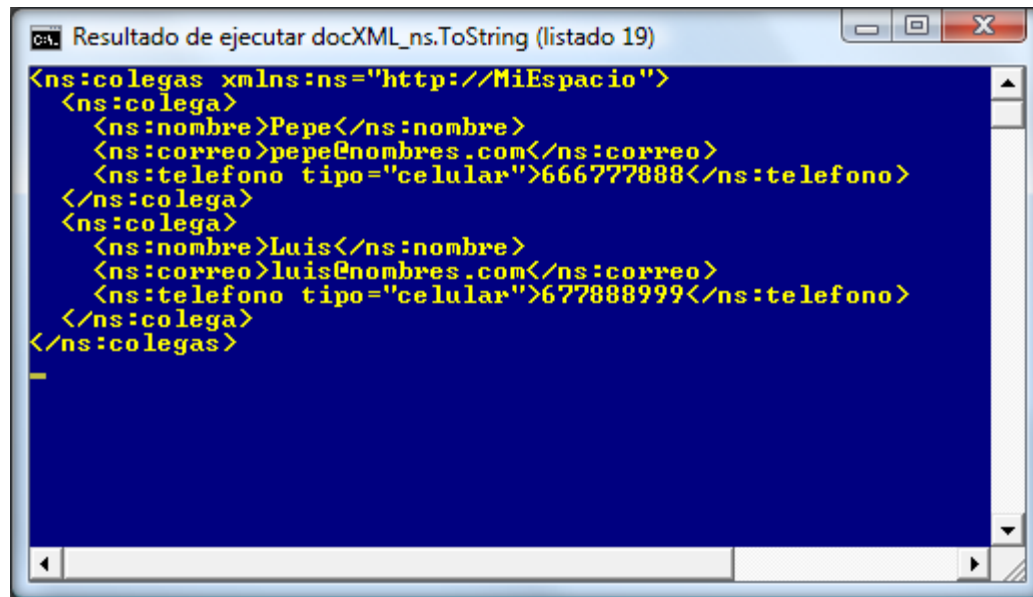


Figura 8.4. Resultado de mostrar el contenido de la variable definida en el listado 8.19

```
Dim res1 = From c In docXML_ns...<ns:colega> _
            Where c.<ns:telefono>.@tipo = "celular"

For Each r In res1
    Console.WriteLine("{0} {1}", _
                      r.<ns:nombre>.Value, _
                      r.<ns:telefono>.Value)
Next
```

Listado 8.20. Si los elementos están definidos en un espacio de nombres, tenemos que indicar el prefijo del alias usado

## Dónde podemos usar los literales XML

Para finalizar este capítulo, indicar que este tipo de literales lo podemos usar con cualquier tipo de aplicación de Visual Basic, pero si esa aplicación es de tipo ASP.NET, los literales solo los podremos usar en los archivos de código (con extensión **.vb**), no directamente en la sección **<script>** de la página ASPX. Esto mismo es aplicable a los archivos **.xaml** de las aplicaciones WPF.

## Otros literales XML de trato especial

Aunque no lo he usado en los ejemplos de este capítulo, entre los literales XML que podemos usar se incluyen **todos** los literales XML, incluyendo los **CDATA** y los comentarios, y la forma de usarlos sería como es habitual para esos elementos XML, y para muestra, en el listado 8.21 vemos un par de ejemplos de estos literales XML especiales.



```
Dim elem1 =  
<pruebas>  
  <!-- Esto es un comentario XML -->  
  <![CDATA[Lo que está dentro de CDATA no se evalúa]]>  
</pruebas>
```

Listado 8.21. También podemos usar comentarios y construcciones CDATA

Recordemos que todo lo que esté dentro de un elemento **CDATA** no se evalúa, por tanto, en ese elemento podemos incluir cualquier cosa que creamos conveniente, como código u otros elementos, que no serán analizados por el compilador ni por el lenguaje XML.

Lo que no podremos hacer ni con los elementos **CDATA** ni con los comentarios XML es anidar unos elementos dentro de otros, pero esto también es así en los documentos XML normales, por tanto, no es una limitación de Visual Basic.

Confío que todo lo tratado en este capítulo le sirva al lector para tener una idea clara de qué podemos hacer con los literales XML en nuestros proyectos de Visual Basic, y le sirva de base para seguir investigando y profundizando en todo este nuevo mundo que nos abre Visual Basic y la utilización de XML directamente en nuestros archivos de código.

### **Versiones**

*Esta característica solo la podemos usar con .NET Framework 3.5, para las consultas LINQ debemos tener una referencia a System.Linq.dll y para usar los literales XML debemos tener una referencia a System.Xml.Linq.dll (todas las plantillas de proyectos para .NET 3.5 de Visual Basic 2008 incluyen esas referencias).*

# Capítulo 9

## Visual Basic y LINQ (IV)

---

### Visual Basic y LINQ to ADO.NET

En este último capítulo del libro sobre las novedades de Visual Basic 9.0 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 sección de Visual Basic 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*.

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 Visual Basic 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, o en el caso que nos atañe, en tiempo de escritura, ya que Visual Basic 2008 detecta los errores conforme vamos escribiendo nuestro código.

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. Actualmente 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 **Modulo1** 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 9.1 vemos los pasos preliminares para dejar listos los objetos en memoria para acceder a los diferentes datos que tenemos en el **DataSet**.

```
Dim ta As New NorthwindDataSetTableAdapters.EmployeesTableAdapter
Dim datos As New NorthwindDataSet
ta.Fill(datos.Employees)
```

Listado 9.1. 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 9.2, en el que indicamos que solo queremos los empleados que en el país (propiedad **Country**) contengan el valor **USA**.

```
Dim res1 = From e In datos.Employees _
           Where e.Country = "USA" _
           Select e.FirstName, e.LastName, e.BirthDate, e.Country
```

Listado 9.2. Una consulta LINQ para acceder a los datos de la tabla Employees

El código del listado 9.2 sería el equivalente del listado 9.3, escrito con *Transact-SQL*.

```
SELECT FirstName, LastName, BirthDate, Country
FROM Employees
WHERE Country = 'USA'
```

Listado 9.3. 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 el uso de esa cláusula también es opcional, pero en este ejemplo en concreto, no la hemos omitido, ya que nuestra intención es obtener ciertos campos (o columnas) de esa tabla, si quisiéramos obtener todas las columnas podríamos haberlo hecho de dos formas: indicando después de **Select** la variable usada para obtener los datos, o simplemente no indicando la cláusula **Select**. ¡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** o no indicar la cláusula **Select**, 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 9.1. 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 desentender-nos de los asistentes de acceso a datos.

En el listado 9.4 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**.

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 9.5 para una posible solución.

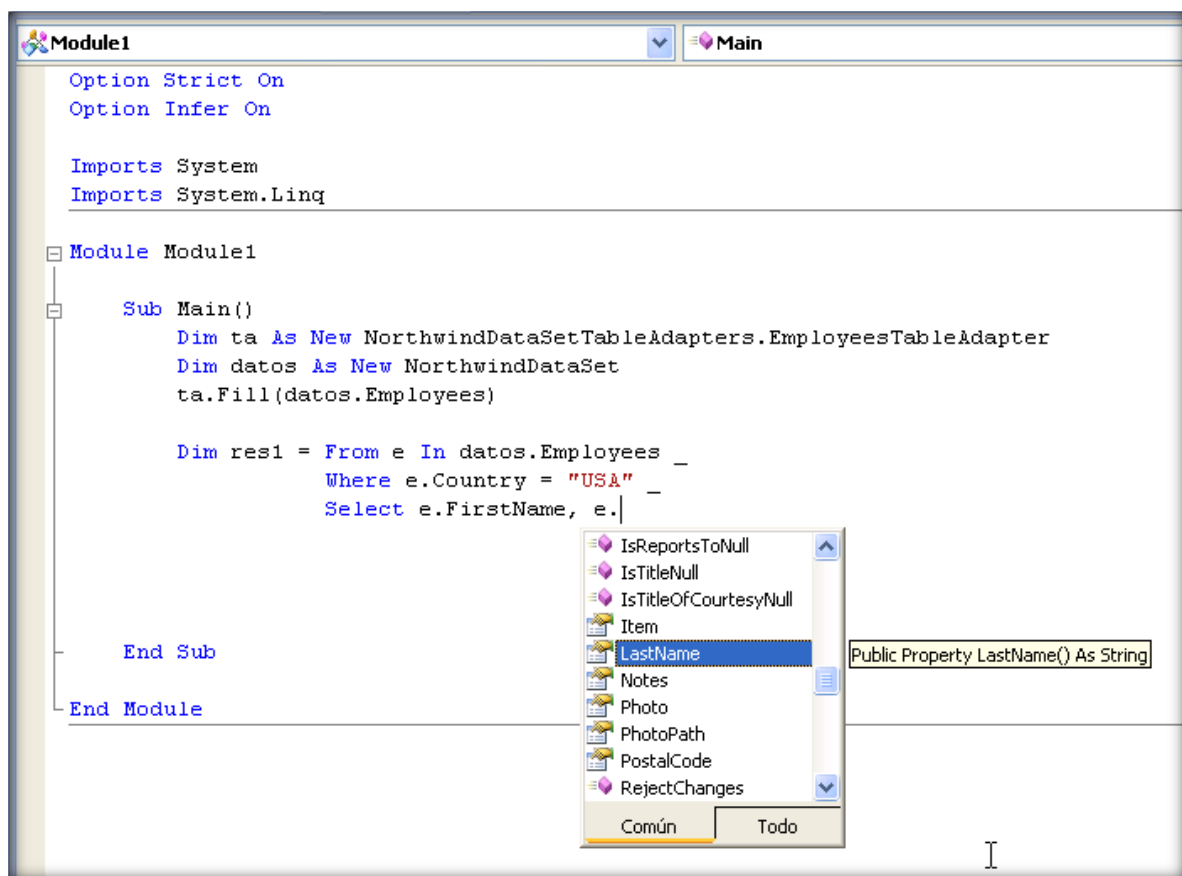


Figura 9.1. IntelliSense es de gran ayuda con los *datasets* tipados

```
Dim cs As New SqlConnectionStringBuilder
cs.DataSource = "(local)\SQLEXPRESS"
cs.InitialCatalog = "Northwind"
cs.IntegratedSecurity = True
Dim da As New SqlDataAdapter("SELECT * FROM Employees", cs.ConnectionString)
Dim dt As New DataTable
da.Fill(dt)
```

Listado 9.4. Creación de objetos sin necesidad de usar asistentes de acceso a datos

```
Dim res1 = From e In dt.AsEnumerable _
    Where e.Field(Of String)("Country") = "USA" _
    Select FirstName = e.Field(Of String)("FirstName"), _
    LastName = e.Field(Of String)("LastName"), _
    BirthDate = e.Field(Of DateTime)("BirthDate"), _
    Country = e.Field(Of String)("Country")
```

Listado 9.5. Consulta de LINQ para acceder a un objeto de tipo DataTable

Complicado el código del listado 9.5, ¿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 9.5, 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, como vimos en capítulos anteriores, 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 9.6.

```
Dim res2 = From e In dt.AsEnumerable _
    Where e("Country").ToString = "USA" _
    Select FirstName = e("FirstName"), _
    LastName = e("LastName"), _
    BirthDate = e("BirthDate"), _
    Country = e("Country")
```

Listado 9.6. Si no nos importa utilizar Object, siempre podemos optar por la vía fácil



Y si además de no importarle usar los tipos inadecuados, es de los que prefiere usar **Option Strict Off** (aunque seguramente dejará de ser mi *amiguito*), podría escribir todo ese código sin necesidad de usar ningún método de extensión y acceder directamente a la colección **Rows** del objeto **DataTable**, tal como vemos en el listado 9.7.

```
' Esto funcionará solo con Option Strict Off
Dim res3 = From e In dt.Rows _
    Where e("Country") = "USA" _
    Select FirstName = e("FirstName"), _
    LastName = e("LastName"), _
    BirthDate = e("BirthDate"), _
    Country = e("Country")
```

Listado 9.7. Para los no estrictos, Visual Basic aún puede hacer las cosas medianamente bien

En cualquier caso, por favor, no dejemos que el motor en tiempo de ejecución de Visual Basic haga las cosas que nosotros debemos hacer, 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 9.2.

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 9.1, 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 9.8.



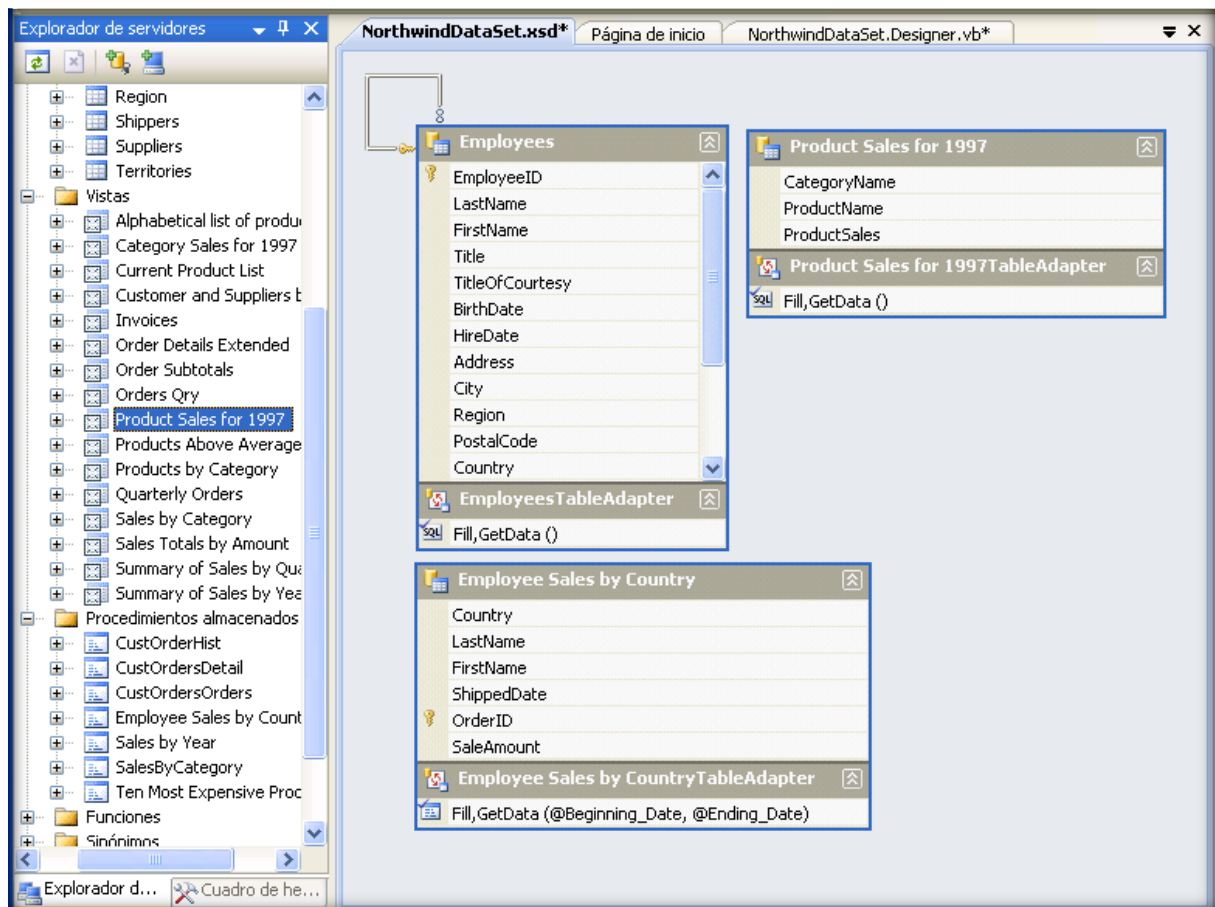


Figura 9.2. El DataSet con nuevos elementos de la base de datos Northwind

```
' Necesitamos una importación a NorthwindDataSetTableAdapters
Dim ta As New Product_Sales_for_1997TableAdapter
Dim datos As New NorthwindDataSet
ta.Fill(datos.Product_Sales_for_1997)
```

Listado 9.8. 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 9.9 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 9.9).

```
Dim res1 = From e In datos.Product_Sales_for_1997 _
           Select e.CategoryName, e.ProductName, e.ProductSales
```

Listado 9.9. 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 9.10 tenemos el código con la definición de los objetos adecuados para acceder a ese procedimiento almacenado.

```
Dim fecha1 = #1/15/1997#
Dim fecha2 = CType("31/01/1997", Date?)

' Necesitamos una importación a NorthwindDataSetTableAdapters
Dim ta As New Employee_Sales_by_CountryTableAdapter
Dim datos As New NorthwindDataSet
ta.Fill(datos.Employee_Sales_by_Country, fecha1, fecha2)
```

Listado 9.10. El método Fill está definido para aceptar los argumentos que necesite el objeto

En el código del listado 9.10, 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 nosotros podamos hacer un nuevo filtro al escribir la consulta de LINQ, tal como vemos en el código del listado 9.11, en el que solo queremos recuperar los datos que sean del día 16.

```
Dim res1 = From e In datos.Employee_Sales_by_Country _
           Where e.ShippedDate.Day = 16 _
           Select e.FirstName, e.LastName, e.ShippedDate, e.Country
```

Listado 9.11. 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 noveno 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 Visual Basic 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 9.3 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.

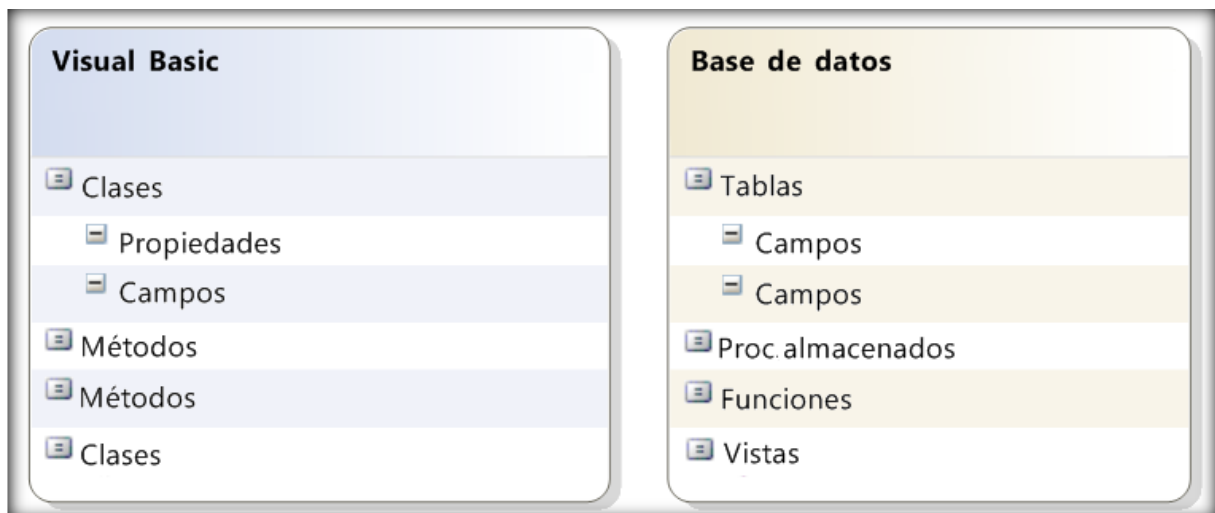


Figura 9.3. Equivalencias entre el código de LINQ to SQL y los objetos de la base de datos

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 9.12.

```
Imports System.Data.Linq
Imports System.Data.Linq.Mapping
```

Listado 9.12. 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 9.13 tenemos la definición de esa clase que ligaremos con la tabla de la base de datos.

Como vemos en el código del listado 9.13, 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 Nombre As String

    <Column (Name:="LastName")>
    Public Apellidos As String

    <Column (Name:="BirthDate")>
    Public FechaNacimiento As Date?

    <Column (Name:="Country")>
    Public País As String

End Class
```

Listado 9.13. 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, no es necesario indicar expresamente a qué elementos de la base de datos se refieren, esto lo podemos ver en el código del listado 9.14.

```
<Table()> _  
Class Employees  
    <Column()> _  
    Public FirstName As String  
  
    <Column()> _  
    Public LastName As String  
  
    <Column()> _  
    Public BirthDate As Date?  
  
    <Column()> _  
    Public Country As String  
End Class
```

Listado 9.14. 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 9.15 vemos el código.

```
Dim sCnn = "Data Source = (local)\SQLEXPRESS; " & _  
           "Initial Catalog = Northwind; " & _  
           "Integrated Security = True"  
  
Dim dc As New DataContext(sCnn)
```

Listado 9.15. 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 9.16, 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 9.13.

```
Dim losEmpleados = dc.GetTable(Of Empleados)()
```

Listado 9.16. 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 9.17 vemos la consulta de LINQ para obtener todos los empleados que el país empiece por la letra “U”.

```
Dim res = From emp In losEmpleados
           Where emp.País.StartsWith("U") _
           Order By emp.Apellidos
```

Listado 9.17. Una consulta LINQ utilizando la clase definida en el listado 9.13

Como vemos en el código del listado 9.17, 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 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 Visual Basic 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 Visual Basic con todos los elementos de la base de datos **Northwind**, podemos escribir el código del listado 9.18 para crear un archivo de código llamado **Northwind.vb** 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.vb
```

Listado 9.18. 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 9.15 a 9.17, 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 9.19, 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.

```

Dim sCnn = "Data Source = (local)\SQLEXPRESS; " &
           "Initial Catalog = Northwind; " &
           "Integrated Security = True"
Dim dc As New DataContext(sCnn)
Dim losEmpleados = dc.GetTable(Of Employees)()
Dim res = From emp In losEmpleados
           Where emp.Country.StartsWith("U")
           Order By emp.LastName

For Each r In res
    Console.WriteLine("{0} {1}, {2:dd/MM/yyyy} ({3})",
                      r.FirstName, r.LastName, r.BirthDate, r.Country)
Next

```

Listado 9.19. 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 9.4. 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.

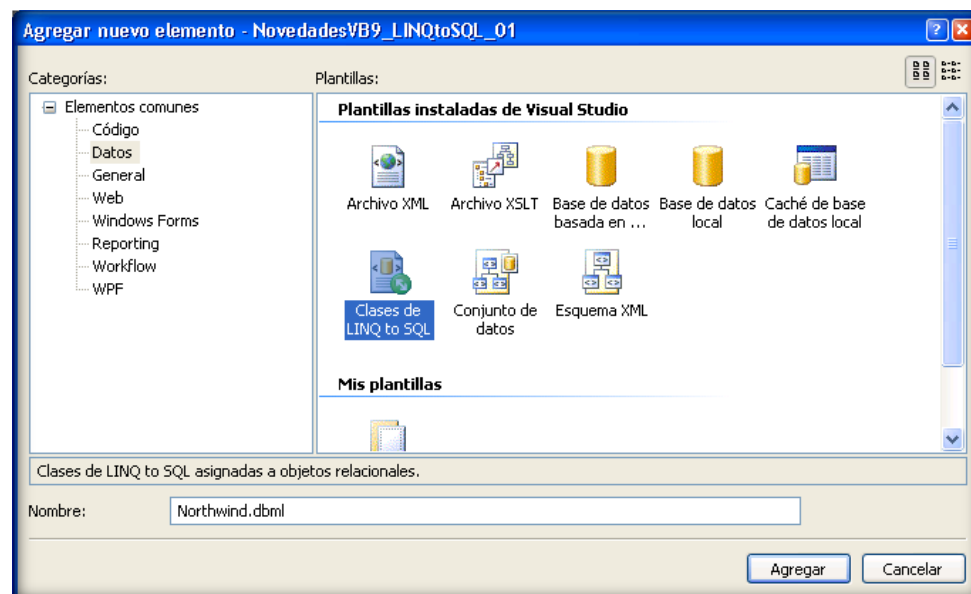


Figura 9.4. Agregar un nuevo elemento del tipo LINQ to SQL

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 creamos 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 9.5 podemos ver una captura en la que tenemos los mismos elementos que ya vimos en la figura 9.2.

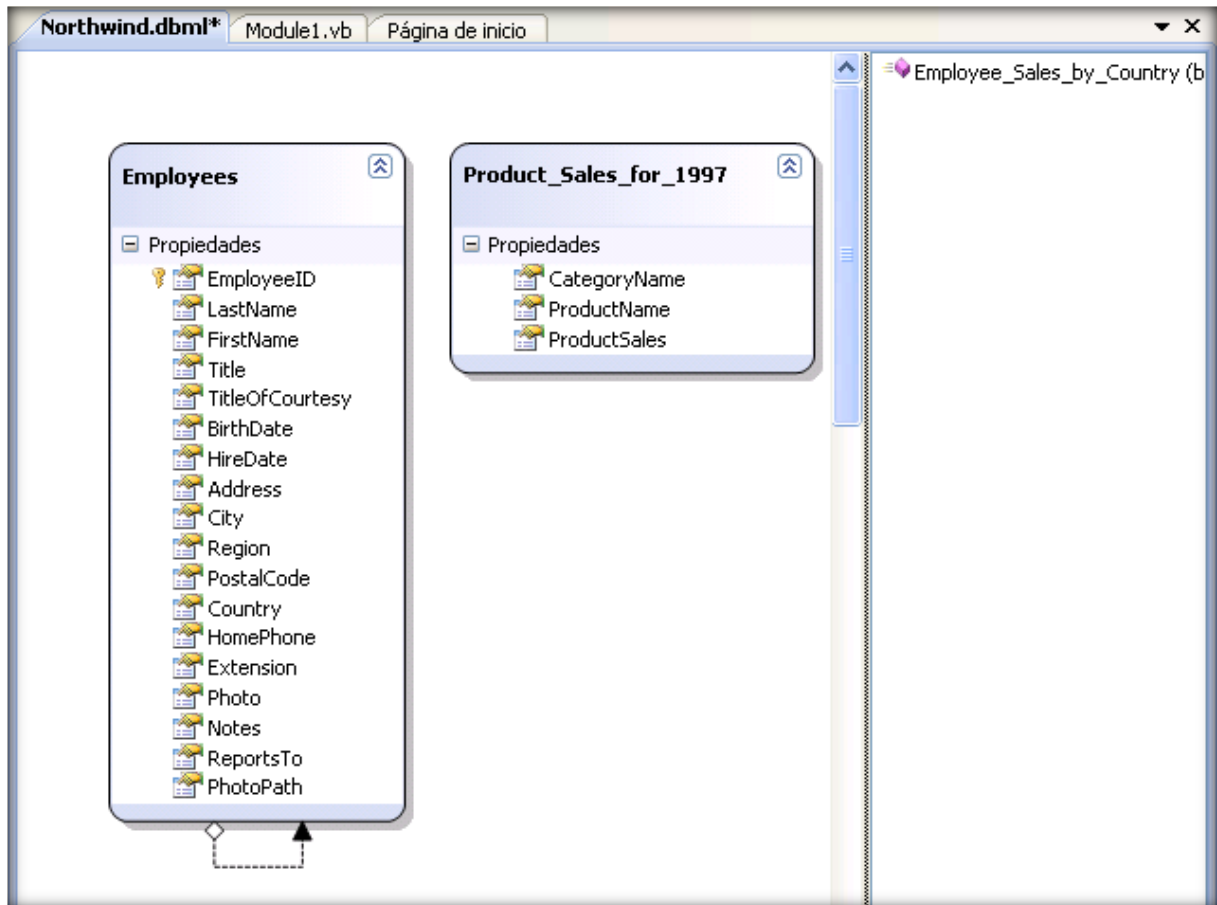


Figura 9.5. Varios elementos de la base de datos en el diseñador de objetos relacionales

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 9.19, 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 9.20, 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.



```

Dim db = New NorthwindDataContext
Dim losEmpleados = db.GetTable(Of Employees)()
Dim res = From emp In losEmpleados
           Where emp.Country.StartsWith("U")
           Order By emp.LastName
For Each r In res
    Console.WriteLine("{0} {1}, {2:dd/MM/yyyy} ({3})", _
                      r.FirstName, r.LastName, r.BirthDate, r.Country)
Next

```

Listado 9.20. 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 9.21, 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 **Date?**).

```

Dim db = New NorthwindDataContext
Dim fecha1 = New Date(1997, 1, 1)
Dim fecha2 = New Date(1997, 1, 31)
Dim lasVentas = db.Employee_Sales_by_Country(fecha1, fecha2)
Dim res1 = From v In lasVentas Where v.ShippedDate.Value.Day = 16
For Each v In res1
    Console.WriteLine("{0}, {1} {2:dd/MM/yyyy} {3:#,##0.00}", _
                      v.LastName, v.FirstName, _
                      v.ShippedDate, v.SaleAmount)
Next

```

Listado 9.21. 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 Visual Basic 9.0.

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 **SolidQ 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.*

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=1>.

Espero que haya disfrutado de la lectura tanto o más como yo en escribirlo.

Nos vemos.

Guillermo

P.S.

Si alguien se pregunta si es casualidad que sean 9 los capítulos de este libro sobre las novedades de la versión 9.0 de Visual Basic, decirle que puede ser causalidad más que casualidad.

Es que siempre me ha hecho gracia cuando en algunas tertulias, los tertulianos empiezan a buscar las razones de que el autor hiciera esto o aquello por esta o aquella razón. Este libro seguramente no será el tema de ninguna tertulia, pero si alguna vez lo es o alguien quiere saber si es casualidad o no que el libro tenga el mismo número de capítulos que la versión sobre la que trata, pues... habrá que esperar a ver si el libro que escriba sobre la siguiente versión de Visual Basic tiene 10 capítulos.

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

# Índice alfabético

---

## Símbolos

%>, véase LINQ to XML  
<%=, véase LINQ to XML  
?, véase Tipo anulable  
?:, véase Operador ternario

## A, B

Action, 101  
AddHandler, 96  
AddressOf, 48, 68, 96, 128  
Aggregate, 140, 146, 148  
All, 148, 151, véase Aggregate  
Any, 148, 151, véase Aggregate  
Ascending, 135, 142, 157  
AsEnumerable, 45, 190  
Atributos XML, véase LINQ to XML  
Average, 148, 149, véase Aggregate  
Binario, véase Operador binario  
Byte, 70  
Boolean, 50, 74, 141, 151  
Boolean?, 50, 81

## C

CDATA, véase LINQ to XML  
Cdbl, 62  
CInt, 61, 72  
Clases de LINQ to SQL,  
ClearProjectError, 64  
CLR, 19, 88, 116  
Column, 195  
Common Language Runtime, véase CLR  
CompareString, 62  
CompareTo, 105  
Comparison, 106  
CompilerServices, 62, 112  
Conjunto de datos, véase DataSet  
Consultas, véase LINQ  
Contravarianza, 65  
Conversions, 62  
Count, 148, 149, véase Aggregate  
CrearLista, 45, 94, 105, 133  
CStr, 60

CType, 61, 193

## D

DataContext, 196  
DataSet, 137, 185, 191  
DataTable, 189  
Date, 51, 200  
Date?, 193, 200  
DateTime, 51, 190  
Delegado, 65, 95, 96, 128  
Delegate, 49, 66, 98, 129  
Descending, 142, 157  
DirectCast, 69  
Diseñador relacional de objetos, 137, 193, 198  
Distinct, 155  
Documentos XML, véase LINQ to XML

## E

Elementos XML, véase LINQ to XML  
Employees, véase Northwind  
Ensamblados amigos, 53, 56  
Equals, 89, 148  
Erase, 81  
Explorador de servidores, 191, 198  
Extension, 112, 120, 169

## F

Field, 190  
Fill, 187, 192  
Friend, 53, 121, 130, véase Ensamblados amigos  
From, 135, 138, 141, 146, 178  
Func, 101, véase Lambda  
Funciones anónimas, véase Lambda  
Funciones en línea, véase Lambda  
Function, véase Lambda

## G

GetHashCode, 89, 91  
GetTable, 196  
GetValueOrDefault, 53

Group By, 158, 159  
Group Join, 158, 161

## H

Handled, 67  
HasValue, 51, 53  
Herramientas, menú, 25, 33

## I

IComparable, 105  
IComparer, 105  
IDE, 17, 22, 31, 57, 75, 114, 175  
IEnumerable, 45, 95, 137, 169, 180, 185  
If, véase Operador ternario  
IIf, véase Operador ternario  
Ildasm.exe, 57  
Importar y exportar configuraciones, 33  
Imports, 112, 118, 169, 182, 195  
    XML, véase LINQ to XML  
In, 41, 138, 146, 150, 179  
Inferencia de tipos, véase Option Infer  
Inicialización de objetos, 37, 42, 68, 86, 94, 134  
INNER JOIN, véase Group Join  
Int32, 78, 149  
Int64, 149  
Integer, 50, 62, 78, 149  
Integer?, 50, 81  
Integración de XML, véase LINQ to XML  
Integrated Development Environment, véase IDE  
InternalsVisibleTo, 557, 57, véase Ensamblados amigos  
Items, 104, 160

## J, K

Join, 146, 159, 162  
Key, 89, 94  
KeyPress, 65  
KeyPressEventArgs, 66  
KeyPressEventHandler, 65

## L

Lambda, 78, 85, 95, 102, 109, 150  
Left, 135  
LEFT JOIN, véase Group Join

Len, 113  
Let, 165  
LINQ, 19, 23, 35, 45, 85, 110, 131  
LINQ to ADO.NET, 137, 185  
LINQ to DataSet, 137, 145, 185  
LINQ to Objects, 137, 145  
LINQ to SQL, 131, 137, 185, 193, 200  
LINQ to XML, 137, 145, 171  
List, 45, 105, 133  
ListBox, 104  
Literales XML, véase LINQ to XML  
Long, 70, 149  
LongCount, 148, 149, véase Aggregate

## M

Max, véase Aggregate  
Métodos de extensión, véase Métodos extensores  
Métodos extensores, 45, 111-130  
Métodos parciales, 37, 46, 48  
Microsoft.VisualBasic, 59, 73  
Microsoft.VisualBasic.CompilerServices, 62  
Min, 148, 149, véase Aggregate  
Module, 59, 112, 127  
MTrim, 122

## N

New With, 86  
Nombre fuerte, véase Nombre seguro  
Nombre seguro, 54  
Northwind, 186, 187, 193, 197  
NotInheritable, 130  
Nullable, 50

## O

O/R, véase Diseñador relacional de objetos  
Object, 38, 66, 78, 82, 109, 187  
Opciones, menú, 25, 33  
Opciones de compilación, 27, 32  
Operador binario, 81  
Operador ternario, 73  
Operators, 62  
Option Infer, 28, 37, 86, 93  
Option Strict, 28, 38, 69, 71, 75, 108, 191  
Optional, 110, 125  
Order By, 142, 157

## P

PadFillLeft, 135  
ParamArray, 45, 110  
Parent, 179  
Partial, (tipos), 47  
Preserve, 81  
Proyectos y soluciones, 33

## R

ReDim, 80, 140  
Resize, 80  
Rows, 191  
Runtime, véase CLR y VBRuntime

## S

Sangría, 24  
Sealed, 130  
Select, 135, 141, 145, 155, 179, 188  
SetProjectError, 64  
Shared, 63, 126  
Short, 69  
Skip, 163  
Sn.exe, véase Nombre seguro  
Sort, 105  
SQL, véase T-SQL  
SqlDataAdapter, 187, 191  
SqlMetal.exe, 197  
StartsWith, 135, 157, 197  
String, 37, 62, 82, 123, 135  
Strong name, véase Nombre seguro  
Sum, 140, 148, 149, 169  
System.Core, 112  
System.Data, 189  
System.Data.Linq, 131, 195, 200  
System.Data.SqlClient, 189  
System.Linq, 45, 145, 184  
System.Xml.Linq, 172, 184

## T

Table, 195  
Take, 163  
Ternario, véase Operador ternario  
Tipo anónimo, 85-95, 105, 135, 141, 158, 190  
Tipo anulable, 51, 200  
ToArray, 45, 139  
ToInteger, 62  
ToList, 139  
ToString, 52, 62, 86  
Transformaciones XML, véase LINQ to XML  
Trim, 123  
TryCast, 83  
T-SQL, 133, 135, 146, 159, 188, 197

## U, V

UAC, 31  
User Account Control, 31  
VBRuntime, 60

## W

Where, 135, 138, 148, 153  
While, véase Skip y Take  
Windows Presentation Foundation, 19, 181  
Windows Vista, 18, 31  
WPF, 19, 181  
WriteLine, 41, 136

## X

XDocument, 172, 180  
XElement, 172, 180  
XML, véase LINQ to XML  
Xsd, véase LINQ to XML  
Xsl, véase LINQ to XML

# Novedades de Visual Basic 9.0

---



## 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 SolidQ, 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 .NET y Visual Basic 2005.

---

**Novedades de Visual Basic 9.0** contiene toda la información sobre la nueva versión del compilador de Visual Basic 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 Basic. En la segunda se explican con todo lujo de detalles, todas las novedades del lenguaje, desde las nuevas instrucciones y opciones del compilador, 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 y LINQ to SQL.

---

"SolidQ 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.

SolidQ 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.

