



Curso de Introducción a .NET con Visual Basic 2010

Autores

Guillermo Som, Unai Zorrilla, Jorge Serrano

Microsoft®

Bienvenido al curso de Introducción a .NET con Visual Basic 2010

En este curso podrás aprender a desarrollar aplicaciones Windows con **Visual Studio 2010**, y terminaremos desarrollando una aplicación real con los conceptos aprendidos. Al final de cada lección tendrás disponible un video en el que podrás ver los conceptos explicados de forma práctica sobre el entorno de Visual Studio 2010.

Este curso le enseñará entre otras cosas:

- Las características fundamentales del lenguaje Visual Basic
- Cómo crear aplicaciones Windows con Visual Studio 2010
- Utilizar controles en sus aplicaciones
- Trabajar con imágenes y gráficos en 2D y 3D
- Desplegar las aplicaciones que desarrolle
- Utilizar las clases que se incluyen en la librería de .NET
- Acceder a datos provenientes de una base de datos
- Conocer en qué consiste LINQ y como utilizarlo en Visual Basic
- Cómo exponer funcionalidad a otras aplicaciones mediante servicios Web.

Te recomendamos también que veas la aplicación [MSDN Video](#), que desarrollaremos al finalizar el curso y de la que podrás consultar su código fuente y videos explicativos.

Recuerda que si quieres poner en práctica este curso tienes disponible una versión sin limitaciones de Visual Basic 2010 Express, que incluye la base de datos SQL Server 2005 Express. Puedes descargarla en el área de [versiones Express](#).

¡Disfruta del curso!

Acerca de los autores

Guillermo Som



Guillermo Som, más conocido como "el Guille", es Microsoft MVP (Most Valuable Professional) de Visual Basic desde 1997. Mantiene el sitio web www.elguille.info dedicado a la programación con Visual Basic y todo lo relacionado con punto NET. También es autor de varios libros sobre Visual Basic .NET y C#, y escribe para revistas y otros medios especializados en programación. También es miembro de Ineta Speakers Bureau Latin América y Mentor de Solid Quality Mentors.

Unai Zorrilla



Lleva 10 años desarrollando software como consultor independiente y en diversas empresas privadas, realizando las tareas de arquitectura de software bajo la plataforma .NET, abarcando desde el desarrollo con dispositivos móviles y entornos web, hasta el desarrollo con Windows Forms. Es Microsoft MVP de Compact Framework, colabora activamente con Microsoft en eventos de formación y es autor de numerosos artículos técnicos en revistas especializadas como MTJ.NET, MSDN Online y DotNetManía. Actualmente trabaja como consultor de gran empresa en [Plain Concepts](#) empresa de la cuál es socio fundador.

**Jorge Serrano**

Jorge Serrano es MVP de Visual Basic y trabaja activamente con la comunidad de desarrollo en España y Andorra. Mantiene el sitio web www.portalvb.com y es escritor de varios libros sobre Internet y tecnologías de desarrollo Microsoft además de colaborador asiduo de las revistas especializadas del sector.

Módulo 1: Introducción a la plataforma .NET

- El entorno de ejecución CLR
- Lenguajes, CLS y tipos comunes
- La biblioteca de clases de .NET
- Acceso a datos con ADO.NET
- LINQ
- Aplicaciones Windows Forms
- Aplicaciones Web Forms

Contenido

Este módulo presenta con carácter general la plataforma .NET y cómo ésta se diferencia de otros sistemas de desarrollo tradicionales, como ASP.

- [**Lección 1: Introducción a la plataforma .NET**](#)
 - ¿Qué es la plataforma .NET?
 - El entorno de ejecución CLR
- [**Lección 2: El lenguaje intermedio y el CLS**](#)
 - El lenguaje intermedio
 - La especificación común de los lenguajes .NET
 - El sistema de tipos comunes
- [**Lección 3: La biblioteca de clases de .NET**](#)
 - La BCL
 - Los espacios de nombres
- [**Lección 4: Acceso a datos con ADO.NET**](#)
 - ADO.NET
 - La arquitectura de ADO.NET
 - Capa conectada de datos
 - Capa desconectada
- [**Lección 5: LINQ**](#)
 - Introducción a LINQ
- [**Lección 6: Aplicaciones Windows Forms**](#)
 - Introducción
 - WPF
- [**Lección 7: Aplicaciones Web Forms**](#)
 - Introducción
 - Silverlight

Módulo 1: Introducción a la plataforma .NET

El entorno de ejecución CLR

- Lenguajes, CLS y tipos comunes
- La biblioteca de clases de .NET
- Acceso a datos con ADO.NET
- LINQ
- Aplicaciones Windows Forms
- Aplicaciones Web Forms

Introducción a la plataforma .NET

Simplificando mucho las cosas para poder dar una definición corta y comprensible, se podría decir que la **plataforma .NET** es un *amplio conjunto de bibliotecas de desarrollo que pueden ser utilizadas por otras aplicaciones para acelerar enormemente el desarrollo y obtener de manera automática características avanzadas de seguridad, rendimiento, etc...*

En realidad .NET es mucho más que eso ya que ofrece un entorno gestionado de ejecución de aplicaciones, nuevos lenguajes de programación y compiladores, y permite el desarrollo de todo tipo de funcionalidades: desde programas de consola o servicios Windows hasta aplicaciones para dispositivos móviles, pasando por desarrollos de escritorio o para Internet. Son estos últimos de los que nos ocuparemos en este curso. Pero antes conviene conocer los fundamentos en los que se basa cualquier aplicación creada con .NET, incluyendo las que nos interesan.

El entorno de ejecución CLR

.NET ofrece un entorno de ejecución para sus aplicaciones conocido como *Common Language Runtime* o CLR. El CLR es la implementación de Microsoft de un estándar llamado *Common Language Infrastructure* o CLI. Éste fue creado y promovido por la propia Microsoft pero desde hace años es un estándar reconocido mundialmente por el ECMA.

El CLR/CLI esencialmente, define un entorno de ejecución virtual independiente en el que trabajan las aplicaciones escritas con cualquier lenguaje .NET. Este entorno virtual se ocupa de multitud de cosas importantes para una aplicación: desde la gestión de la memoria y la vida de los objetos, hasta la seguridad y la gestión de subprocesos.

Todos estos servicios unidos a su independencia respecto a arquitecturas computacionales, convierten al CLR en una herramienta extraordinariamente útil puesto que, en teoría, cualquier aplicación escrita para funcionar según la CLI puede ejecutarse en cualquier tipo de arquitectura de hardware. Por ejemplo Microsoft dispone de implementaciones de .NET para Windows de 32 bits, Windows de 64 bits e incluso para Windows Mobile, cuyo hardware no tiene nada que ver con la arquitectura de un ordenador común.

Módulo 1: Introducción a la plataforma .NET

- El entorno de ejecución CLR
- Lenguajes, CLR y tipos comunes
- La biblioteca de clases de .NET
- Acceso a datos con ADO.NET
- LINQ
- Aplicaciones Windows Forms
- Aplicaciones Web Forms

El Lenguaje Intermedio y el CLS

Al contrario que otros entornos, la plataforma .NET no está atada a un determinado lenguaje de programación ni favorece a uno determinado frente a otros. En la actualidad existen implementaciones para varias decenas de lenguajes que permiten escribir aplicaciones para la plataforma .NET. Los más conocidos son **Visual Basic .NET** o **C#**, pero existen implementaciones de todo tipo, como F#, Python, Fortran, e incluso COBOL.

Lo mejor de todo es que cualquier componente creado con uno de estos lenguajes puede ser utilizado de forma transparente desde cualquier otro lenguaje .NET. Además, como ya se ha comentado, es posible ejecutar el código .NET en diferentes plataformas y sistemas operativos.

¿Cómo se consigue esta potente capacidad?

Dentro del CLI, existe un lenguaje llamado **IL** (*Intermediate Language* o *Lenguaje Intermedio*) que está pensado de forma independiente al procesador en el que se vaya a ejecutar. Es algo parecido al código ensamblador pero de más alto nivel y creado para un hipotético procesador virtual que no está atado a una arquitectura determinada.

Cuando se compila una aplicación escrita en un lenguaje .NET cualquiera (da igual que sea VB, C# u otro de los soportados), el compilador lo que genera en realidad es un nuevo código escrito en este lenguaje intermedio. Así, todos los lenguajes .NET se usan como capa de más alto nivel para producir código IL.

Un elemento fundamental del CLR es el compilador JIT (*just-in-time*). Su cometido es el de compilar bajo demanda y de manera transparente el código escrito en lenguaje intermedio a lenguaje nativo del procesador físico que va a ejecutar el código.

Al final, lo que se ejecuta es código nativo que ofrece un elevado rendimiento. Esto es cierto también para las aplicaciones Web escritas con ASP.NET y contrasta con las aplicaciones basadas en ASP clásico que eran interpretadas, no compiladas, y que jamás podrían llegar al nivel de desempeño que ofrece ASP.NET.

La siguiente figura muestra el aspecto que tiene el código intermedio de una aplicación sencilla y se puede obtener usando el desensamblador que viene con la plataforma .NET.

The screenshot shows the Microsoft Intermediate Language (MSIL) disassembler (ILDASM.exe) window. The title bar reads "DataProtector::Decrypt : unsigned int8[](unsigned int8[],unsigned in...". The main area displays MSIL code:

```
IL_0128: stloc.0
IL_0129: ldloc.0
IL_012a: brtrue.s    IL_0142
IL_012c: call       class [mscorlib]System.Text.Encoding [ms
IL_0131: ldstr      ...
IL_0136: callvirt   instance unsigned int8[] [mscorlib]Syste
IL_013b: stloc.s    V_12
IL_013d: leave      IL_01e4
IL_0142: ldsfld    native int [mscorlib]System.IntPtr::Zero
IL_0147: ldloca.s   V_2
IL_0149: ldfld     native int Krasis.Cryptography.DataProte
IL_014e: call       bool [mscorlib]System.IntPtr::op_Inequal

IL_0153: brfalse.s  IL_0161
IL_0155: ldloca.s   V_2
IL_0157: ldfld     native int Krasis.Cryptography.DataProte
IL_015c: call       void [mscorlib]System.Runtime.InteropServices
IL_0161: ldsfld    native int [mscorlib]System.IntPtr::Zero
IL_0166: ldloca.s   V_6
IL_0168: ldfld     native int Krasis.Cryptography.DataProte
IL_016d: call       bool [mscorlib]System.IntPtr::op_Inequal

IL_0172: brfalse.s  IL_0180
IL_0174: ldloca.s   V_6
IL_0176: ldfld     native int Krasis.Cryptography.DataProte
IL_017b: call       void [mscorlib]System.Runtime.InteropServices
IL_0180: leave.s    IL_019b
} // end .try
catch [mscorlib]System.Exception
{
```

Figura 1.1. Código en lenguaje intermedio obtenido con ILDASM.exe

La especificación común de los lenguajes y el sistema de tipos comunes

Para conseguir la interoperabilidad entre lenguajes, no sólo se llega con el lenguaje intermedio, sino que es necesario disponer de unas "reglas del juego" que *definan* un conjunto de características que todos los lenguajes deben incorporar y cumplir. A este conjunto regulador se le denomina *Common Language Specification (CLS)* o, en castellano, especificación común de los lenguajes.

Entre las cuestiones que regula el CLS se encuentran la nomenclatura, la forma de definir los miembros de los objetos, los metadatos de las aplicaciones, etc... Una de las partes más importantes del CLS es la que se refiere a los tipos de datos.

Si alguna vez ha programado la API de Windows o ha tratado de llamar a una DLL escrita en C++ desde Visual Basic 6, habrá comprobado lo diferentes que son los tipos de datos de VB6 y de C++. Para evitar este tipo de problemas y poder gestionar de forma eficiente y segura el acceso a la memoria, el CLS define un conjunto de tipos de datos comunes (*Common Type System* o **CTS**) que indica qué tipos de datos se pueden manejar, cómo se declaran y se utilizan éstos, y de qué manera se deben gestionar durante la ejecución.

Si nuestras bibliotecas de código utilizan en sus interfaces hacia el exterior datos definidos dentro de la CTS no existirán problemas a la hora de utilizarlos desde cualquier otro código escrito en la plataforma .NET.

Cada lenguaje .NET utiliza una sintaxis diferente para cada tipo de datos. Así, por ejemplo, el tipo común correspondiente a un número entero de 32 bits (**System.Int32**) se denomina **Integer** en Visual Basic .NET, pero se llama **int** en C#. En ambos casos representan el mismo tipo de datos que es lo que realmente cuenta (**System.Int32**).

Nota:

En ASP 3.0 se suele usar VBScript como lenguaje de programación. En este lenguaje interpretado, al igual que en VB6, un *Integer* representaba un entero de 16 bits. Los enteros de 32 bits eran de tipo *Long*. Es un fallo muy común usar desde Visual Basic .NET el tipo *Integer* pensando que es de 16 bits cuando en realidad es capaz de albergar números mucho mayores. Téngalo en cuenta si usted está familiarizado con Visual Basic 6 o anteriores cuando empiece a programar.

Existen tipos por valor (como los enteros que hemos mencionado o las enumeraciones) y tipos por referencia (como las clases). En el siguiente módulo se profundiza en todas estas cuestiones.

Módulo 1: Introducción a la plataforma .NET

- El entorno de ejecución CLR
 - Lenguajes, CLR y tipos comunes
- La biblioteca de clases de .NET**
- Acceso a datos con ADO.NET
 - LINQ
 - Aplicaciones Windows Forms
 - Aplicaciones Web Forms

La biblioteca de clases de .NET

Todo lo que se ha estado comentando hasta ahora en el curso constituye la base de la plataforma .NET. Si bien es muy interesante y fundamental, por sí mismo no nos serviría de mucho para crear programas si debiésemos crear toda la funcionalidad desde cero.

Obviamente esto no es así, y la plataforma .NET nos ofrece infinidad de funcionalidades "de fábrica" que se utilizan como punto de partida para crear las aplicaciones. Existen funcionalidades básicas (por ejemplo todo lo relacionado con la E/S de datos o la seguridad) y funcionalidades avanzadas en las que se fundamentan categorías enteras de aplicaciones (acceso a datos, creación de aplicaciones Web...).

Toda esta funcionalidad está implementada en forma de bibliotecas de funciones que físicamente se encuentran en diversas DLL (bibliotecas de enlazado dinámico). A su conjunto se le denomina *Base Classes Library* (*Biblioteca de clases base* o **BCL**), y forman parte integral de la plataforma .NET, es decir, no se trata de añadidos que se deban obtener o adquirir aparte.

La siguiente figura ilustra a vista de pájaro la arquitectura conceptual general de la plataforma .NET. En ella se pueden observar los elementos que se han mencionado en apartados anteriores (lenguajes, CLR, CLS...) y en qué lugar se ubican las bibliotecas de clases base:

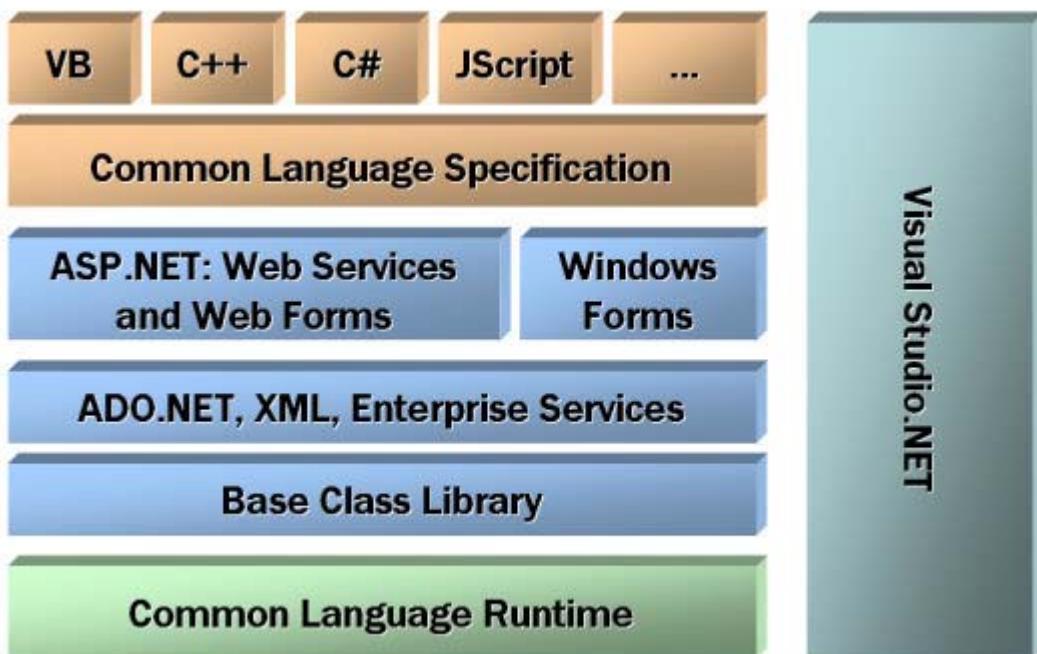


Figura 1.2. Distintos elementos de la plataforma .NET y cómo se relacionan entre sí.

Resulta muy útil para comprender lo explicado hasta ahora. No se preocupe si hay elementos que no conoce u otros elementos que echa en falta en este gráfico, más adelante estudiaremos todos los elementos y algunos que no aparecen aquí como por ejemplo LINQ, y el cual veremos más adelante.

Todo lo que se encuentra en la BCL forma parte de la plataforma .NET. De hecho existe tal cantidad de funcionalidad integrada dentro de estas bibliotecas (hay más de 4000 clases) que el mayor esfuerzo que todo programador que se inicia en .NET debe hacer es el aprendizaje de las más importantes, aumentando el conocimiento del resto a base de práctica. De todos modos Visual Studio ofrece mucha ayuda contextual (documentación, *Intellisense*...) y una vez que se aprenden los rudimentos resulta fácil ir avanzando en el conocimiento de la BCL a medida que lo vamos necesitando.

Los espacios de nombres

Dada la ingente cantidad de clases con la que podemos trabajar, debe existir alguna forma de organizarlas de un modo coherente. Además hay que tener en cuenta que podemos adquirir más funcionalidades (que se traducen en clases) de otros fabricantes, por no mencionar que crearemos continuamente nuevas clases propias.

Para solucionar este problema existen en todos los lenguajes .NET los **espacios de nombres** o **namespaces**.

Un espacio de nombres no es más que un identificador que permite organizar de modo estanco las clases que estén contenidas en él así como otros espacios de nombres.

Así por ejemplo, todo lo que tiene que ver con el manejo de estructuras de datos XML en la plataforma .NET se encuentra bajo el espacio de nombres **System.Xml**. La funcionalidad fundamental para crear aplicaciones Web está en el espacio de nombres **System.Web**. Éste a su vez contiene otros espacios de nombres más especializados como **System.Web.Caching** para la persistencia temporal de datos, **System.Web.UI.WebControls**, que contiene toda la funcionalidad de controles Web para interfaz de usuario, etc...

No obstante, en nuestros desarrollos internos crearemos nuestras propias clases dentro de nombres de espacio concretos. Esto nos ayudará a localizar e interpretar rápidamente la división y subdivisión de nuestros objetos. Un aspecto a tener en cuenta, es tratar de evitar crear nuestros nombres de espacio de forma idéntica a la que Microsoft ha utilizado la los nombres de espacio de .NET Framework, pero esto es algo que iremos solventando con práctica y atención.

Módulo 1: Introducción a la plataforma .NET

- El entorno de ejecución CLR
- Lenguajes, CLR y tipos comunes
- La biblioteca de clases de .NET

Acceso a datos con ADO.NET

- LINQ
- Aplicaciones Windows Forms
- Aplicaciones Web Forms

Acceso a datos con ADO.NET

El acceso a fuentes de datos es algo indispensable en cualquier lenguaje o plataforma de desarrollo. La parte de la BCL que se especializa en el acceso a datos se denomina de forma genérica como **ADO.NET**.

Si usted ha programado con Visual Basic 6.0 ó con ASP, ha empleado en su código con total seguridad la interfaz de acceso a datos conocida como ADO (*ActiveX Data Objects*), y posiblemente lo ha combinado además con ODBC (*Open Database Connectivity*). Si además es usted de los programadores con solera y lleva unos cuantos años en esto, es probable que haya usado RDO o incluso DAO, todos ellos métodos mucho más antiguos.

ADO.NET ofrece una funcionalidad completamente nueva, que tiene poco que ver con lo existente hasta la fecha en el mercado. Sin embargo, con el ánimo de retirar barreras a su aprendizaje, Microsoft denominó a su nuevo modelo de acceso a datos con un nombre similar y algunas de sus clases recuerdan a objetos de propósito análogo en el vetusto ADO.

ADO.NET es un modelo de acceso mucho más orientado al trabajo desconectado de las fuentes de datos de lo que nunca fue ADO. Si bien este último ofrecía la posibilidad de desconectar los *Recordsets* y ofrecía una forma de serialización de estos a través de las diferentes capas de una aplicación, el mecanismo no es ni de lejos tan potente como el que nos ofrece ADO.NET.

El objeto más importante a la hora de trabajar con el nuevo modelo de acceso a datos es el **DataSet**. Sin exagerar demasiado podríamos calificarlo casi como un motor de datos relacionales en memoria. Aunque hay quien lo asimila a los clásicos *Recordsets* su funcionalidad va mucho más allá como se verá en el correspondiente módulo.

Arquitectura de ADO.NET

El concepto más importante que hay que tener claro sobre ADO.NET es su modo de funcionar, que se revela claramente al analizar su arquitectura:

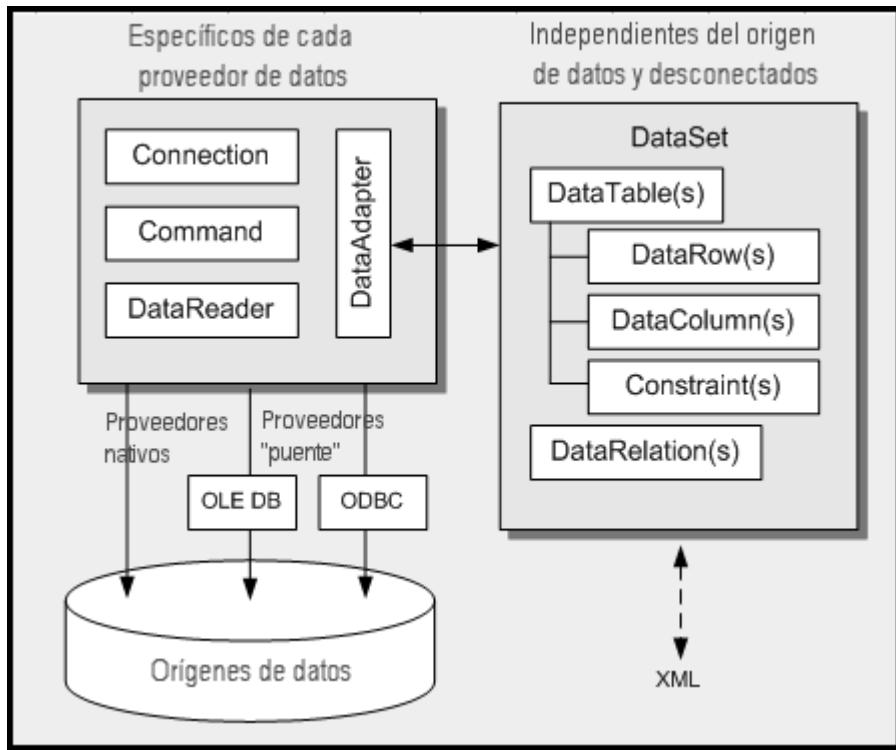


Figura 1.3.- Arquitectura de ADO.NET

Existen dos capas fundamentales dentro de su arquitectura: la **capa conectada** y la **desconectada**.

Capa conectada

La primera de ellas contiene objetos especializados en la conexión con los orígenes de datos. Así, la clase genérica **Connection** se utiliza para establecer conexiones a los orígenes de datos. La clase **Command** se encarga de enviar comandos de toda índole al origen de datos. Por fin la clase **DataReader** está especializada en leer los resultados de los comandos mientras se permanece conectado al origen de datos.

La clase **DataAdapter** hace uso de las tres anteriores para actuar de puente entre la capa conectada y la desconectada.

Estas clases son abstractas, es decir, no tienen una implementación real de la que se pueda hacer uso directamente. Es en este punto en donde entran en juego los **proveedores de datos**. Cada origen de datos tiene un modo especial de comunicarse con los programas que los utilizan, además de otras particularidades que se deben contemplar. Un proveedor de datos de ADO.NET es una implementación concreta de las clases conectadas abstractas que hemos visto, que hereda de éstas y que tiene en cuenta ya todas las particularidades del origen de datos en cuestión.

Así, por ejemplo, las clases específicas para acceder a SQL Server se llaman **SqlConnection**, **SqlCommand**, **SqlDataReader** y **SqlDataAdapter** y se encuentran bajo el espacio de nombres **System.Data.SqlClient**. Es decir, al contrario que en ADO clásico no hay una única clase **Connection** o **Command** que se use en cada caso, si no que existen clases especializadas para conectarse y recuperar información de cada tipo de origen de datos.

Nota:

El hecho de utilizar clases concretas para acceso a las fuentes de datos no significa que no sea posible escribir código independiente del origen de datos. Todo lo contrario. La plataforma .NET ofrece grandes facilidades de

escritura de código genérico basadas en el uso de herencia e implementación de interfaces. De hecho la versión 2.0 de .NET ofrece grandes novedades específicamente en este ámbito.

Existen **proveedores nativos**, que son los que se comunican directamente con el origen de datos (por ejemplo el de SQL Server o el de Oracle), y **proveedores "puente"**, que se utilizan para acceder a través de ODBC u OLEDB cuando no existe un proveedor nativo para un determinado origen de datos.

Nota:

Estos proveedores puente, si bien muy útiles en determinadas circunstancias, ofrecen un rendimiento menor debido a la capa intermedia que están utilizando (ODBC u OLEDB). Un programador novato puede sentir la tentación de utilizar siempre el proveedor puente para OLEDB y así escribir código compatible con diversos gestores de datos de forma muy sencilla. Se trata de un error y siempre que sea posible es mejor utilizar un proveedor nativo.

Capa desconectada

Una vez que ya se han recuperado los datos desde cualquier origen de datos que requiera una conexión ésta ya no es necesaria. Sin embargo sigue siendo necesario trabajar con los datos obtenidos de una manera flexible. Es aquí cuando la capa de datos desconectada entra en juego. Además, en muchas ocasiones es necesario tratar con datos que no han sido obtenidos desde un origen de datos relacional con el que se requiera una conexión. A veces únicamente necesitamos un almacén de datos temporal pero que ofrezca características avanzadas de gestión y acceso a la información.

Por otra parte las conexiones con las bases de datos son uno de los recursos más escasos con los que contamos al desarrollar. Su mala utilización es la causa más frecuente de cuellos de botella en las aplicaciones y de que éstas no escalen como es debido. Esta afirmación es especialmente importante en las aplicaciones Web en las que se pueden recibir muchas solicitudes simultáneas de cualquier parte del mundo.

Finalmente otro motivo por el que es importante el uso de los datos desconectados de su origen es la transferencia de información entre capas de una aplicación. Éstas pueden encontrarse distribuidas por diferentes equipos, e incluso en diferentes lugares del mundo gracias a Internet. Por ello es necesario disponer de algún modo genérico y eficiente de poder transportar los datos entre diferentes lugares, utilizarlos en cualquiera de ellos y posteriormente tener la capacidad de conciliar los cambios realizados sobre ellos con el origen de datos del que proceden.

Todo esto y mucho más es lo que nos otorga el uso de los objetos **DataSet**. Es obvio que no se trata de tareas triviales, pero los objetos *DataSet* están pensados y diseñados con estos objetivos en mente. Como podremos comprobar más adelante en este curso es bastante sencillo conseguir estas funcionalidades tan avanzadas y algunas otras simplemente usando de manera adecuada este tipo de objetos.

Nota:

Otra interesante característica de los *DataSet* es que permiten gestionar simultáneamente diversas tablas (relaciones) de datos, cada una de un origen diferente si es necesario, teniendo en cuenta las restricciones y las relaciones existentes entre ellas.

Los *DataSet*, como cualquier otra clase no sellada de .NET, se pueden extender mediante herencia. Ello facilita una técnica avanzada que consiste en crear tipos nuevos de *DataSet* especializados en la gestión de una información concreta (por ejemplo un conjunto de tablas

relacionadas). Estas nuevas tipos clases se denominan genéricamente ***DataSet Tipados***, y permiten el acceso mucho más cómodo a los datos que representan, verificando reglas de negocio, y validaciones de tipos de datos más estrictas.

Módulo 1: Introducción a la plataforma .NET

- El entorno de ejecución CLR
 - Lenguajes, CLR y tipos comunes
 - La biblioteca de clases de .NET
 - Acceso a datos con ADO.NET
- LINQ**
- Aplicaciones Windows Forms
 - Aplicaciones Web Forms

LINQ

LINQ es quizás una de las características más novedosas introducidas en Microsoft .NET Framework en los últimos años.

No cejaré en animarle a usar LINQ siempre que pueda, no porque sea mejor que otras formas de trabajar que encontrará en Microsoft .NET Framework, sino porque simplifica el código, acorta los tiempos de desarrollo, y bajo mi particular punto de vista, permite que el desarrollador desarrolle aplicaciones más productivas.

LINQ significa **Language INtegrated Query**, o lo que es lo mismo, lenguaje integrado de consultas.

Se trata de una de las novedades incluidas en Microsoft .NET Framework 3.5 y continuada en Microsoft .NET Framework 4.0, y representa una forma nueva de desarrollo tal y como la conocíamos hasta ahora.

LINQ está pensado desde la orientación a objetos, pero su objetivo fundamental es la manipulación y trabajo con datos.

Cuando hablamos de datos siempre pensamos en bases de datos, pero LINQ tiene una implicación mucho más generalista, de hecho, podemos trabajar con colecciones, matrices, etc.

Para que comprenda más aún la importancia de LINQ, diremos que la inmensa mayoría de las novedades introducidas en Microsoft .NET Framework 3.5 tienen su razón de ser gracias a LINQ, y que LINQ juega un importante papel en las novedades de Microsoft .NET Framework 4.0 y en las próximas novedades de Microsoft .NET Framework.

A la hora de trabajar con datos, con LINQ tenemos la posibilidad de seleccionar, manipular y filtrar datos como si estuviéramos trabajando con una base de datos directamente.

Evidentemente, podemos usar LINQ con bases de datos, pero también con matrices, colecciones de datos, etc como decíamos antes.

En todo esto, *Intellisense* juega un papel muy importante evitando que cometamos errores a la hora de escribir nuestras aplicaciones, errores como por ejemplo seleccionar el dato erróneo o el tipo de dato inadecuado, ya que el compilador nos avisaría de estos y otros errores.

Por lo tanto, el desarrollo de aplicaciones con LINQ, permite diseñar aplicaciones más seguras a la hora de trabajar con datos y sus tipos.

Orígenes de datos

Dentro de LINQ encontramos diferentes orígenes de datos dentro de Microsoft .NET Framework 3.5 ó posterior.

Estos orígenes de datos son los siguientes:

- LINQ to Objects: representa el uso de LINQ sobre objetos.
- LINQ to XML: representa el uso de LINQ sobre documentos XML.
- ADO.NET y LINQ: dentro de este grupo encontramos los diferentes orígenes de datos de LINQ que tienen relación directa con datos relacionales. Los orígenes de datos de LINQ y ADO.NET son:
 - LINQ to DataSet: representa el uso de LINQ sobre DataSet.
 - LINQ to SQL: representa el uso de LINQ sobre orígenes de datos de SQL Server.
 - LINQ to Entities: representa el uso de LINQ sobre cualquier origen de datos, SQL Server, Oracle, MySql, DB2, etc.

Como vemos, LINQ abarca más que el propio trabajo con datos de bases u orígenes de datos.

Así, con LINQ podemos manipular y trabajar con datos de objetos o de documentos XML. Este último representa una importante ventaja para los desarrolladores de VB respecto a los desarrolladores de C#, ya que el equipo de trabajo de VB ha hecho especial énfasis al trabajo de LINQ con documentos XML motivado por el *feedback* recibido por la Comunidad de desarrolladores de todo el mundo.

De todas las maneras, cabe recordar que no existen grandes diferencias entre los equipos de Microsoft encargados de dotar de novedades o mejoras a VB y a C#, ya que Microsoft ha empeñado sus esfuerzos más recientes en que los avances de ambos lenguajes vayan de la mano.

Más adelante en este curso, veremos como utilizar LINQ y como aprovechar todas las ventajas que nos ofrece.

Módulo 1: Introducción a la plataforma .NET

- El entorno de ejecución CLR
 - Lenguajes, CLR y tipos comunes
 - La biblioteca de clases de .NET
 - Acceso a datos con ADO.NET
 - LINQ
- Aplicaciones Windows Forms**
- Aplicaciones Web Forms

Aplicaciones Windows Forms

Las aplicaciones de escritorio son aquellas basadas en ventanas y controles comunes de Windows que se ejecutan en el sistema local. Son el mismo tipo de aplicaciones que antes construiríamos con Visual Basic 6 u otros entornos similares.

En la plataforma .NET, el espacio de nombres que ofrece las clases necesarias para construir aplicaciones de escritorio bajo Windows se denomina **Windows Forms**. Este es también el nombre genérico que se le otorga ahora a este tipo de programas basados en ventanas.

Windows Forms está constituido por multitud de clases especializadas que ofrecen funcionalidades para el trabajo con ventanas, botones, rejillas, campos de texto y todo este tipo de controles habituales en las aplicaciones de escritorio.

Visual Studio ofrece todo lo necesario para crear visualmente este tipo de programas, de un modo similar aunque más rico al que ofrecía el entorno de desarrollo integrado de Visual Basic.

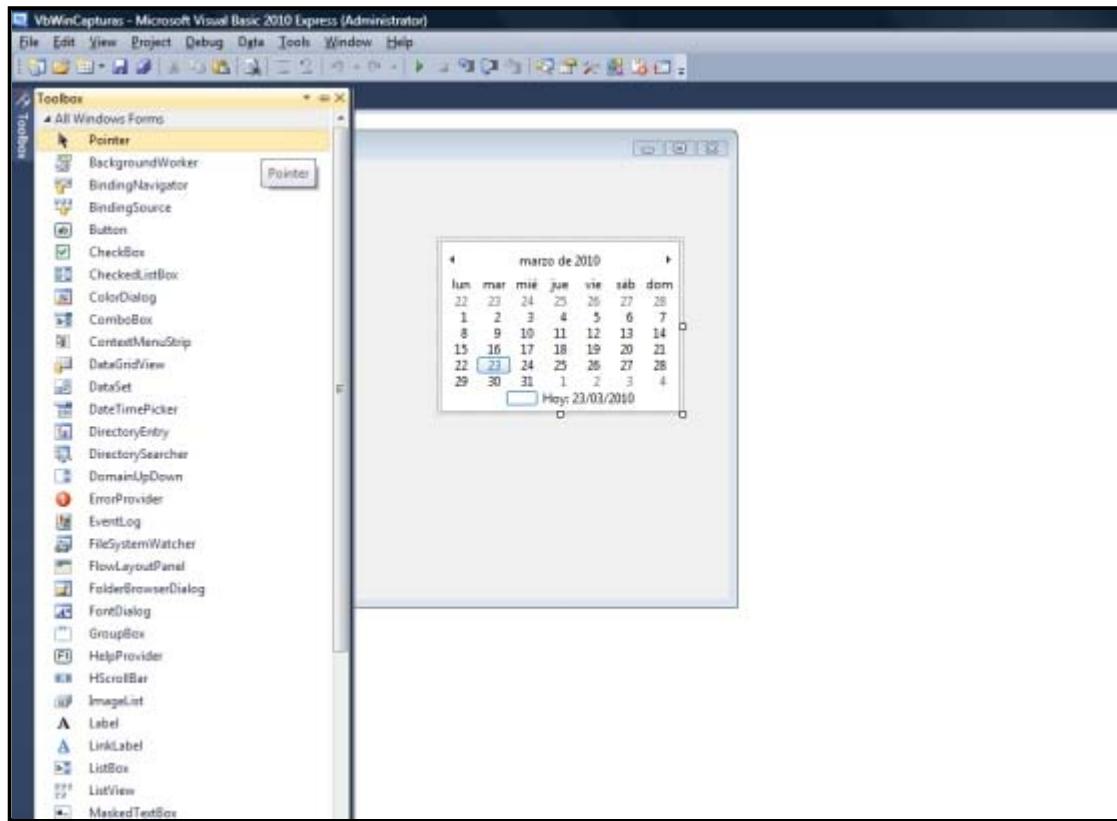


Figura 1.4.- Diseñador de interfaces de aplicaciones de escritorio con Windows Forms en Visual Studio.

Al contrario que en VB6, .NET proporciona control sobre todos los aspectos de las ventanas y controles, no dejando nada fuera del alcance del programador y otorgando por lo tanto la máxima flexibilidad. Los formularios (ventanas) son clases que heredan de la clase base **Form**, y cuyos controles son miembros de ésta. De hecho se trata únicamente de código y no es necesario (aunque sí muy recomendable) emplear el diseñador gráfico de Visual Studio para crearlas.

Este es el aspecto que presenta parte del código que genera la interfaz mostrada en la anterior figura:

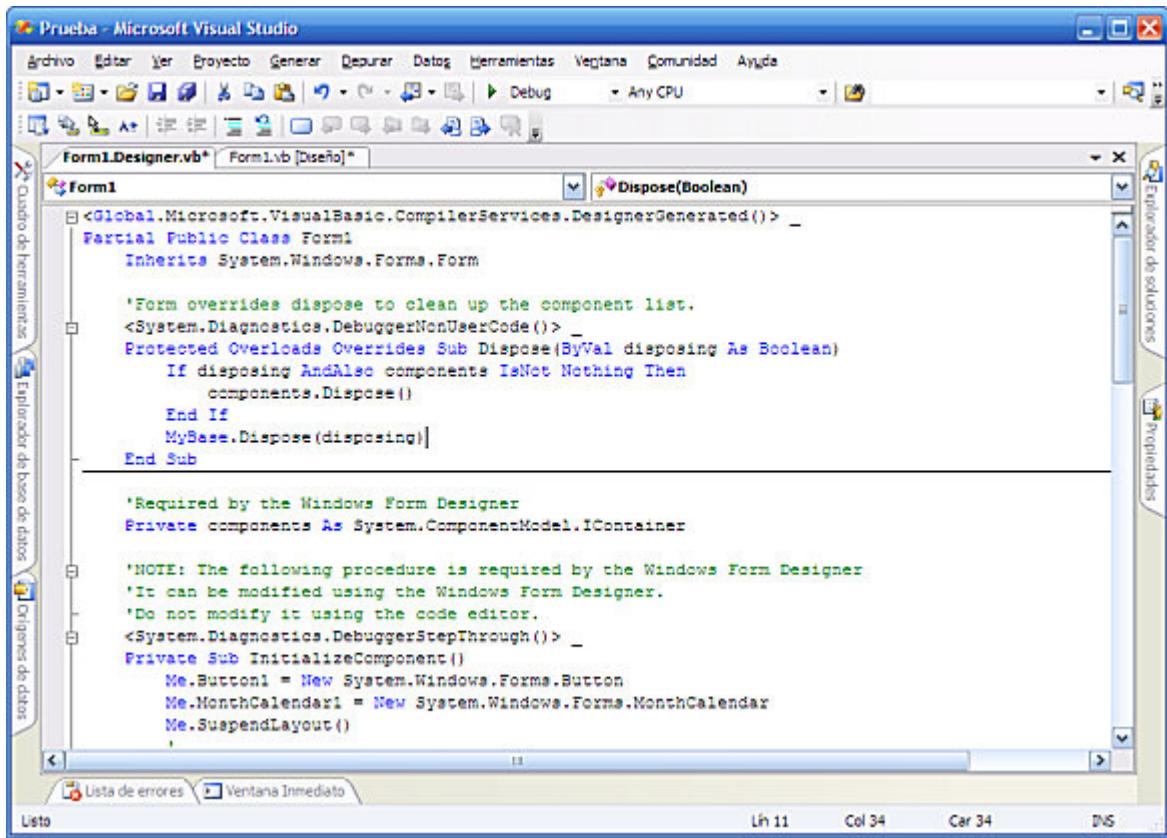


Figura 1.5.- Código autogenerado por Visual Studio para crear la interfaz de la figura anterior.

Al contrario que en Visual Basic tradicional, en donde siempre existían instancias por defecto de los formularios que podíamos usar directamente, en .NET es necesario crear un objeto antes de poder hacer uso de los formularios:

```
Dim frm As New MiFormulario
frm.Show()
```

Todos los controles heredan de una clase **Control** por lo que conservan una serie de funcionalidades comunes muy interesantes, como la capacidad de gestionarlos en el diseñador (moviéndolos, alineándolos...), de definir márgenes entre ellos o hacer que se adapten al tamaño de su contenedor.

WPF

Sin embargo, en los últimos años Microsoft ha estado trabajando intensamente en lo que se ha denominado como WPF, o lo que es lo mismo, Windows Presentation Foundation.

WPF es "*otra forma*" de crear aplicaciones de escritorio con un enriquecimiento en la experiencia del usuario.

Con la aparición de Windows Vista, se hizo mucho más presente que antes la presentación tridimensional de gráficos, imágenes, así como efectos multimedia muy variados.

Windows 7 por su parte, continúa la ruta marcada por Windows Vista para la creación de aplicaciones Windows ricas.

WPF facilita al desarrollador la creación de aplicaciones de escritorio más ricas, aportando al usuario una experiencia nunca vista en las aplicaciones de este tipo.

El origen de WPF no obstante, hay que buscarlo en Microsoft .NET Framework 3.0, y es a

partir de Microsoft .NET Framework 3.5 sobre todo, cuando esta tecnología ha empezando a crearse un hueco de referencia dentro del mercado de desarrollo de aplicaciones Software.

¿Quiere aprender WPF?

Recuerde que tiene la posibilidad de acceder gratuitamente a un curso dedicado en exclusiva a WPF.

Módulo 1: Introducción a la plataforma .NET

- El entorno de ejecución CLR
- Lenguajes, CLR y tipos comunes
- La biblioteca de clases de .NET
- Acceso a datos con ADO.NET
- LINQ
- Aplicaciones Windows Forms
- Aplicaciones Web Forms**

Aplicaciones Web Forms

Tradicionalmente las aplicaciones Web se han desarrollado siguiendo un modelo mixto que intercalaba código HTML y JavaScript propio de páginas Web (parte cliente), junto con código que se ejecutaría en el servidor (parte servidora). Este modelo contrastaba por completo con el modelo orientado a eventos seguido por las principales herramientas de desarrollo de aplicaciones de escritorio.

En el modelo orientado a eventos se define la interfaz de usuario colocando controles en un contenedor y se escribe el código que actuará como respuesta a las interacciones de los usuarios sobre estos controles. Si conoce el diseñador de VB6 o de Windows Forms mencionado en el apartado anterior sabe exactamente a qué nos referimos.

Hacer esto en una aplicación de escritorio no tiene mayor dificultad ya que todo el código se ejecuta en el mismo lugar. La principal característica de las aplicaciones Web sin embargo es que se la interfaz de usuario (lo que los usuarios de la aplicación ven) se ejecuta en un lugar diferente al código de la aplicación que reside en un servidor. Para mayor desgracia estas aplicaciones se basan en el uso del protocolo HTTP que es un protocolo sin estado y que no conserva la conexión entre dos llamadas consecutivas.

Por ejemplo, el siguiente código ilustra el código que es necesario escribir en ASP para disponer de una página que rellena una lista de selección con unos cuantos nombres (podrían salir de una base de datos y aún sería más complicado), y que dispone de un botón que escribe un saludo para el nombre que se haya elegido de la lista.

The screenshot shows a code editor window with an ASP script. The code includes HTML tags like <html>, <body>, and <form>. It also contains VBScript code within <% %> blocks, including an array declaration, a for loop to generate a dropdown menu, and an if statement to check if a selected name matches one in the array. The code ends with a submit button and a final if statement.

```
<html>

<%@ Language=VBScript %>
<%
nombres = array("Antonio",
    "Jose", "Alberto",
    "Luis", "Benito")
%>

<body>
<p>seleccione su nombre:</p>
<form method="POST" action="HolaMundo.asp">
    <p><select name="nombre" size="5">
<% for i = 0 to UBound(nombres) %>
        <option
            <% if Request.Form("nombre") = nombres(i) then %>
                selected <% end if %>
            <%=nombres(i)%></option>
<% next %>
    </select><br><br>

    <input type="submit" value="Di hola"></p>
</form>
<% if Request.Form("nombre") <> "" then %>
<p>Hola, <%=Request.Form("nombre")%></p>
<% end if %>

</body>
</html>
```

Figura 1.6.- Código ASP sencillo que genera una lista de selección y saluda al presionar un botón.

Obviamente se podría haber simplificado sin enviar el formulario al servidor usando JavaScript en el cliente para mostrar el saludo, pero la intención es ilustrar la mezcla de código de cliente y de servidor que existe en este tipo de aplicaciones.

Las principales desventajas de este tipo de codificación son las siguientes:

1. **No existe separación entre el diseño y la lógica de las aplicaciones.** Si queremos cambiar sustancialmente la apariencia de la aplicación Web lo tendremos bastante complicado puesto que el código del servidor está mezclado entre el HTML.
2. **En ASP clásico no existe el concepto de control** para la interfaz de usuario. Lo único que hay es HTML y JavaScript que se deben generar desde el servidor. En el ejemplo de la figura para generar un control de lista con unos elementos no podemos asignar una propiedad de la lista (porque no existe tal lista), sino que tenemos que crear un bucle que genere los elementos HTML necesarios para generarla. Tampoco disponemos de un diseñador visual que nos permita gestionar los controles y elementos HTML existentes, y menos cuando éstos se encuentran mezclados con el código del servidor.
3. **No disponemos de forma de detectar en el servidor que se ha realizado algo en el cliente.** El cliente se encuentra desconectado desde el momento en que se termina de devolver la página. Sólo se recibe información en el servidor cuando se solicita una nueva página o cuando se envía un formulario tal y como se hace en el ejemplo, debiéndonos encargar nosotros de averiguar si la petición es la primera vez que se hace o no, y de dar la respuesta adecuada. En cualquier caso es mucho menos intuitivo que el modelo de respuesta a eventos de una aplicación de escritorio.

4. **No existe constancia del estado de los controles de cada página entre las llamadas.** En cada ejecución de la página tendremos que recrear completamente la salida. Por ejemplo si presionamos en el botón "Di Hola" tenemos que escribir además de la etiqueta "Hola, nombre" el resto de la pantalla, incluyendo la lista con todos los nombres dejando seleccionado el mismo que hubiese antes. Si estos nombres viniesen de una base de datos esto puede ser todavía más ineficiente y tendremos que buscar métodos alternativos para generarlos ya que en ASP tampoco se deben almacenar en los objetos de sesión y/o aplicación Recordsets resultado de consultas.
5. **No existe el concepto de Propiedad de los controles.** En una aplicación Windows asignamos el texto de un campo usando una propiedad (por ejemplo Text1.Text = "Hola") y ésta se asigna y permanece en la interfaz sin que tengamos que hacer nada. En una aplicación Web clásica tenemos que almacenarlas en algún sitio (una variable de sesión o un campo oculto) para conservarlas entre diferentes peticiones de una misma página.
6. **Los controles complejos no tienen forma de enviar sus valores al servidor.** Si intentamos crear una interfaz avanzada que utilice tablas y otros elementos que no son controles de entrada de datos de formularios de HTML tendremos que inventarnos mecanismos propios para recoger esos datos y enviarlos al servidor.

La principal aportación de ASP.NET al mundo de la programación es que ha llevado a la Web el paradigma de la programación orientada a eventos propia de aplicaciones de escritorio, ofreciendo:

- Separación entre diseño y lógica.
- Componentes de interfaz de usuario, tanto estándar como de terceras empresas o propios.
- Diseñadores gráficos.
- Eventos.
- Estado.
- Enlazado a datos desde la interfaz.

Silverlight

Al igual que para el desarrollo de aplicaciones de escritorio o Windows Forms teníamos la posibilidad de crear aplicaciones de escritorio de una forma natural o con WPF, en las aplicaciones Web tenemos la posibilidad de abordar aplicaciones Web de forma tradicional (la que conocemos hasta ahora) y también con lo que se ha denominado Silverlight.

Silverlight, conocido en fase de diseño como WPF/E o WPF extendido, es al igual que ocurría con WPF, "*otra forma*" de crear aplicaciones Web aportando al usuario un enriquecimiento mucho mayor en lo que a la experiencia del usuario se refiere.

La posibilidad de que el usuario interactúe con la aplicación Software aportándole una experiencia de uso y funcionalidad mucho más agradable y rica, aporta ventajas a las aplicaciones Web de Silverlight sobre las aplicaciones Web tradicionales, y más aún cuando se tratan aspectos como por ejemplo la famosa Web 2.0 o SOA.

El origen de Silverlight es el mismo que WPF, y hay que encontrarlo por lo tanto en Microsoft .NET Framework 3.0. Sin embargo, al igual que ocurría con WPF, es a partir de Microsoft .NET Framework 3.5 sobre todo cuando Silverlight está empezando a ser tenida en cuenta.

Silverlight como tal ha evolucionado rápidamente en poco tiempo, pasando de Silverlight 1.0 a Silverlight 1.1, y de Silverlight 1.1 a una gran evolución con Silverlight 2.0. Sin embargo, Microsoft ha continuado evolucionando a Silverlight de las características demandadas por la comunidad de desarrollo, y así, ha sacado una nueva actualización con importantes mejoras, hablamos de Silverlight 3.0.

¿Quiere aprender más acerca de los Servicios Web?

Recuerde que existe un curso gratuito complementario a este, que le enseñará aspectos relacionados únicamente con los Servicios Web y SOA en general.

Módulo 2: Características del lenguaje

- El sistema de tipos
- Clases y estructuras
- Manejo de excepciones
- Eventos y delegados
- Atributos

Continuación de línea implícita

Visual Basic 2010 introduce una novedad aplicable en todos los desarrollos Software que realicemos y en cualquier momento, por ese motivo y antes de empezar a desarrollar el curso, creo conveniente que lo tenga en consideración.

Esa novedad, recibe el nombre de continuación de línea implícita.

En el siguiente ejemplo, podemos ver como usar la antigua forma de realizar una continuación de línea en Visual Basic:

```
Dim sample As String = "Ejemplo con Visual Basic 2010" & _  
    Environment.NewLine & _  
    "Con salto de linea"
```

Esta antigua forma, que consiste en continuar la línea utilizando el carácter `_` al final de la misma, continua siendo válido. Sin embargo, a partir de Visual Basic 2010, también podremos continuar la línea sin necesidad de utilizar dicho carácter.

Un ejemplo de código compatible con lo que estamos diciendo, sería el siguiente:

```
Dim sample As String = "Ejemplo con Visual Basic 2010" &  
    Environment.NewLine &  
    "Con salto de linea"
```

Como vemos, la única diferencia reside en utilizar u omitir el carácter de salto de línea `_` que ha sido usado desde que apareciera la primera versión de Visual Basic antes incluso de la plataforma .Net.

Contenido

- [Lección 1: El sistema de tipos](#)

- Tipos primitivos
- Variables y constantes
- Enumeraciones
- Arrays (matrices)

- [Lección 2: Clases y estructuras](#)

- Clases
- Definir una clase
- Instanciar una clase
- Estructuras
- Accesibilidad
- Propiedades
- Interfaces

- [Lección 3: Manejo de excepciones](#)

- Manejo de excepciones

- [Lección 4: Eventos y delegados](#)

- Eventos
- Definir y producir eventos en una clase
- Delegados
- Definir un evento bien informado

- [Lección 5: Atributos](#)

- Atributos
-



Lección 1: El sistema de tipos

- **Tipos primitivos**
- **Variables y constantes**
- **Enumeraciones**
- **Arrays (matrices)**

Introducción

En esta primera lección veremos los tipos de datos que .NET Framework pone a nuestra disposición y cómo tendremos que usarlos desde Visual Basic 2010.

A continuación daremos un repaso a conceptos básicos o elementales sobre los tipos de datos, que si bien nos serán familiares, es importante que lo veamos para poder comprender mejor cómo están definidos y organizados los tipos de datos en .NET.

Tipos de datos de .NET

Visual Basic 2010 está totalmente integrado con .NET Framework, por lo tanto, los tipos de datos que podremos usar con este lenguaje serán los definidos en este "marco de trabajo", por este motivo vamos a empezar usando algunas de las definiciones que nos encontraremos al recorrer la documentación que acompaña a este lenguaje de programación.

Los tipos de datos que podemos usar en Visual Basic 2010 son los mismo tipos de datos definidos en .NET Framework y por tanto están soportados por todos los lenguajes que usan esta tecnología. Estos tipos comunes se conocen como el *Common Type System*, (CTS), que traducido viene a significar el sistema de tipos comunes de .NET. El hecho de que los tipos de datos usados en todos los lenguajes .NET estén definidos por el propio Framework nos asegura que independientemente del lenguaje que estemos usando, siempre utilizaremos el mismo tipo interno de .NET, si bien cada lenguaje puede usar un nombre (o alias) para referirse a ellos, aunque lo importante es que siempre serán los mismos datos, independientemente de cómo se llame en cada lenguaje. Esto es una gran ventaja, ya que nos permite usarlos sin ningún tipo de problemas para acceder a ensamblados creados con otros lenguajes, siempre que esos lenguajes sean compatibles con los tipos de datos de .NET.

En los siguientes enlaces tenemos los temas a tratar en esta primera lección del módulo sobre las características del lenguaje Visual Basic 2010.

- **Tipos primitivos**

- Sufijos o caracteres y símbolos identificadores para los tipos
- Tipos por valor y tipos por referencia
- Inferencia de tipos

- **Variables y constantes**

- Consejo para usar las constantes
- Declarar variables
- Declarar variables y asignar el valor inicial
- El tipo de datos Char
- Obligar a declarar las variables con el tipo de datos
- Aplicar Option Strict On a un fichero en particular
- Aplicar Option Strict On a todo el proyecto
- Más opciones aplicables a los proyectos
- Tipos Nullables
- Tipos anónimos
- Propiedades autoimplementadas
- Inicialización de colecciones

- **Enumeraciones: Constantes agrupadas**

- El nombre de los miembros de las enumeraciones
- Los valores de una enumeración no son simples números

- **Arrays (matrices)**

- Declarar arrays
 - Declarar e inicializar un array
 - Cambiar el tamaño de un array
 - Eliminar el contenido de un array
 - Los arrays son tipos por referencia
 - Literales de arrays (o matrices)
-

Lección 1: El sistema de tipos

Tipos primitivos

- Variables y constantes
- Enumeraciones
- Arrays (matrices)

Tipos primitivos

Veamos en la siguiente tabla los tipos de datos definidos en .NET Framework y los alias utilizados en Visual Basic 2010.

.NET Framework	VB 2010
System.Boolean	Boolean
System.Byte	Byte
System.Int16	Short
System.Int32	Integer
System.Int64	Long
System.Single	Single
System.Double	Double
System.Decimal	Decimal
System.Char	Char
System.String	String
System.Object	Object
System.DateTime	Date
System.SByte	SByte
System.UInt16	UShort
System.UInt32	UInteger
System.UInt64	ULong

Tabla 2.1. Tipos de datos y equivalencia entre lenguajes

Debemos tener en cuenta, al menos si el rendimiento es una de nuestra prioridades, que las cadenas en .NET son inmutables, es decir, una vez que se han creado no se pueden modificar y en caso de que queramos cambiar el contenido, .NET se encarga de desechar la anterior y crear una nueva cadena, por tanto si usamos las cadenas para realizar concatenaciones (unión de cadenas para crear una nueva), el rendimiento será muy bajo, si bien existe una clase en .NET que es ideal para estos casos y cuyo rendimiento es superior al tipo *String*: la clase *StringBuilder*.

Las últimas filas mostradas en la tabla son tipos especiales que si bien son parte del sistema de tipos comunes (CTS) no forman parte de la *Common Language Specification* (CLS), es

decir la especificación común para los lenguajes "compatibles" con .NET, por tanto, si queremos crear aplicaciones que puedan interoperar con todos los lenguajes de .NET, esos tipos no debemos usarlos como valores de devolución de funciones ni como tipo de datos usado en los argumentos de las funciones, propiedades o procedimientos.

Los tipos mostrados en la tabla 2.1 son los tipos primitivos de .NET y por extensión de Visual Basic 2010, es decir son tipos "elementales" para los cuales cada lenguaje define su propia palabra clave equivalente con el tipo definido en el CTS de .NET Framework. Todos estos tipos primitivos podemos usarlos tanto por medio de los tipos propios de Visual Basic, los tipos definidos en .NET o bien como literales. Por ejemplo, podemos definir un número entero literal indicándolo con el sufijo **I**: **12345I** o bien asignándolo a un valor de tipo *Integer* o a un tipo *System.Int32* de .NET. La única excepción de los tipos mostrados en la tabla 1 es el tipo de datos *Object*, este es un caso especial del que nos ocuparemos en la próxima lección.

Sufijos o caracteres y símbolos identificadores para los tipos

Cuando usamos valores literales numéricos en Visual Basic 2010, el tipo de datos que le asigna el compilador es el tipo *Double*, por tanto si nuestra intención es indicar un tipo de datos diferente podemos indicarlos añadiendo una letra como sufijo al tipo, esto es algo que los más veteranos de VB6 ya estarán acostumbrados, e incluso los más noveles también, en Visual Basic 2010 algunos de ellos se siguen usando, pero el tipo asociado es el equivalente al de este nuevo lenguaje (tal como se muestra en la tabla 1), por ejemplo para indicar un valor entero podemos usar la letra **I** o el signo **%**, de igual forma, un valor de tipo entero largo (*Long*) lo podemos indicar usando **L** o **&**, en la siguiente tabla podemos ver los caracteres o letra que podemos usar como sufijo en un literal numérico para que el compilador lo identifique sin ningún lugar a dudas.

Tipo de datos	Símbolo	Carácter
Short	N.A.	S
Integer	%	I
Long	&	L
Single	!	F
Double	#	R
Decimal	@	D
UShort	N.A.	US
UInteger	N.A.	UI
ULong	N.A.	UL

Tabla 2.2. Sufijos para identificar los tipos de datos

El uso de estos caracteres nos puede resultar de utilidad particularmente para los tipos de datos que no se pueden convertir en un valor doble.

Nota:

Los sufijos pueden indicarse en minúsculas, mayúsculas o cualquier combinación de mayúscula y minúscula.

Por ejemplo, el sufijo de un tipo *ULong* puede ser: **UL**, **UI**, **ul**, **uL**, **LU**, **Lu**, **IU** o **lu**.

Para evitar confusiones, se recomienda siempre indicarlos en mayúsculas, independientemente de que Visual Basic no haga ese tipo de distinción.

Por ejemplo, si queremos asignar este valor literal a un tipo *Decimal*:

12345678901234567890, tal como vemos en la figura 1, el IDE de Visual Basic 2010 nos indicará que existe un error de desbordamiento (*Overflow*) ya que esa cifra es muy grande para usarlo como valor *Double*, pero si le agregamos el sufijo **D** o **@** ya no habrá dudas de

que estamos tratando con un valor *Decimal*.

```
Option Strict On

Module Module1
    Sub Main()
        Dim m As Decimal = 12345678901234567890
    End Sub
End Module
```

Figura 2.1. Error de desbordamiento al intentar asignar un valor Double a una variable Decimal

Tipos por valor y tipos por referencia

Los tipos de datos de .NET los podemos definir en dos grupos:

- Tipos por valor
- Tipos por referencia

Los tipos por valor son tipos de datos cuyo valor se almacena en la pila o en la memoria "cercana", como los numéricos que hemos visto. Podemos decir que el acceso al valor contenido en uno de estos tipos es directo, es decir se almacena directamente en la memoria reservada para ese tipo y cualquier cambio que hagamos loaremos directamente sobre dicho valor, de igual forma cuando copiamos valores de un tipo por valor a otro, estaremos haciendo copias independientes.

Por otro lado, los tipos por referencia se almacenan en el "monto" (*heap*) o memoria "lejana", a diferencia de los tipos por valor, los tipos por referencia lo único que almacenan es una referencia (o puntero) al valor asignado. Si hacemos copias de tipos por referencia, realmente lo que copiamos es la referencia propiamente dicha, pero no el contenido.

Estos dos casos los veremos en breve con más detalle.

Inferencia de tipos

Una de las características nuevas en Visual Basic 2008 y Visual Basic 2010 es la inferencia de tipos.

Se conoce como inferencia de tipos a la característica de Visual Basic para inferir el tipo de un dato al ser inicializado.

Para que la inferencia de tipos sea efectiva, deberemos activar la opción **Option Infer** a **True**, aunque por defecto, ese es el valor que tiene el compilador de Visual Basic. Sin embargo, si se hace una migración de una aplicación de Visual Basic a Visual Basic 2010, el valor de esta opción será **False**.

Supongamos por lo tanto, la siguiente declaración:

```
Dim datoDeclarado = 2010
```

En este ejemplo, la variable *datoDeclarado*, será una variable de tipo **Integer (Int32)**.

Si deseamos cambiar el tipo de dato a un tipo **Int64** por ejemplo, el compilador nos devolverá un error. Así, el siguiente ejemplo **no** será válido en Visual Basic 2010 con la

opción de inferencia activada:

```
Dim datoDeclarado = 2010  
datoDeclarado = Int64.MaxValue
```

Ahora bien, si cambiamos el valor de **Option Infer** a **False**, el mismo ejemplo será correcto.

¿Dónde está la diferencia?

En este último caso, el caso de tener desactivada la opción de inferencia, la declaración de la variable *datoDeclarado* nos indica que es un tipo de dato **Object** en su origen, y que al darle un valor **Integer**, ésta funciona como una variable entera. Al cambiar su valor a **Long**, esta variable que es de tipo **Object**, cambia sin problemas a valor **Long**.

En todo este proceso, hay un problema claro de rendimiento con acciones de boxing y unboxing que no serían necesarias si tipáramos la variable con un tipo concreto.

Eso es justamente lo que hace la opción **Option Infer** por nosotros. Nos permite declarar una variable con el tipo inferido, y ese tipo de datos se mantiene dentro de la aplicación, por lo que nos da la seguridad de que ese es su tipo de dato, y que ese tipo de dato no va a variar.

Lección 1: El sistema de tipos

- Tipos primitivos
- Variables y constantes**
- Enumeraciones
- Arrays (matrices)

Variables y constantes

Disponer de todos estos tipos de datos no tendría ningún sentido si no pudiéramos usarlos de alguna otra forma que de forma literal. Y aquí es donde entran en juego las variables y constantes, no vamos a contarte qué son y para qué sirven, salvo en el caso de las constantes, ya que no todos los desarrolladores las utilizamos de la forma adecuada.

Consejo para usar las constantes

Siempre que tengamos que indicar un valor constante, ya sea para indicar el máximo o mínimo permitido en un rango de valores o para comprobar el término de un bucle, deberíamos usar una constante en lugar de un valor literal, de esta forma si ese valor lo usamos en varias partes de nuestro código, si en un futuro decidimos que dicho valor debe ser diferente, nos resultará más fácil realizar un solo cambio que cambiarlo en todos los sitios en los que lo hemos usado, además de que de esta forma nos aseguramos de que el cambio se realiza adecuadamente y no tendremos que preocuparnos de las consecuencias derivadas de no haber hecho el cambio en todos los sitios que deberíamos.

Las constantes se definen utilizando la instrucción *Const* seguida del nombre, opcionalmente podemos indicar el tipo de datos y por último una asignación con el valor que tendrá. Como veremos en la siguiente sección, podemos obligar a Visual Basic 2010 a que en todas las constantes (y variables) que declaremos, tengamos que indicar el tipo de datos.

Para declarar una constante lo haremos de la siguiente forma:

```
Const maximo As Integer = 12345678
```

Declarar variables

La declaración de las variables en Visual Basic 2010 se hace por medio de la instrucción *Dim* seguida del nombre de la constante y del tipo de datos que esta contendrá. Con una misma instrucción *Dim* podemos declarar más de una variable, incluso de tipos diferentes, tal y como veremos a continuación.

La siguiente línea de código declara una variable de tipo entero:

```
Dim i As Integer
```

Tal y como hemos comentado, también podemos declarar en una misma línea más de una variable:

```
Dim a, b, c As Integer
```

En este caso, las tres variables las estamos definiendo del mismo tipo, que es el indicado al final de

la declaración.

Nota:

Como hemos comentado, en Visual Basic 2010 se pueden declarar las constantes y variables sin necesidad de indicar el tipo de datos que contendrán, pero debido a que eso no es una buena práctica, a lo largo de este curso siempre declararemos las variables y constantes con el tipo de datos adecuado a su uso.

Declarar variables y asignar el valor inicial

En Visual Basic 2010 también podemos inicializar una variable con un valor distinto al predeterminado, que en los tipos numéricos es un cero, en las fechas es el 1 de enero del año 1 a las doce de la madrugada (#01/01/0001 12:00:00AM#) y en las cadenas es un valor nulo (*Nothing*), para hacerlo, simplemente tenemos que indicar ese valor, tal como veremos es muy parecido a como se declaran las constantes. Por ejemplo:

```
Dim a As Integer = 10
```

En esa misma línea podemos declarar y asignar más variables, pero todas deben estar indicadas con el tipo de datos:

```
Dim a As Integer = 10, b As Integer = 25
```

Por supuesto, el tipo de datos puede ser cualquiera de los tipos primitivos:

```
Dim a As Integer = 10, b As Integer = 25, s As String = "Hola"
```

Aunque para que el código sea más legible, y fácil de depurar, no deberíamos mezclar en una misma instrucción *Dim* más de un tipo de datos.

Nota:

Es importante saber que en las cadenas de Visual Basic 2010 el valor de una variable de tipo String no inicializada NO es una cadena vacía, sino un valor nulo (*Nothing*).

El tipo de datos Char

En Visual Basic 2010 podemos declarar valores de tipo *Char*, este tipo de datos es un carácter Unicode y podemos declararlo y asignarlo a un mismo tiempo. El problema con el que nos podemos encontrar es a la hora de indicar un carácter literal.

Podemos convertir un valor numérico en un carácter o bien podemos convertir un carácter en su correspondiente valor numérico.

```
Dim c As Char  
c = Chr(65)  
Dim n As Byte  
n = Asc(c)
```

En Visual Basic 2010 los tipos *Char* se pueden asignar a variables de tipo *String* y se hará una conversión automática sin necesidad de utilizar funciones de conversión.

Si nuestra intención es asignar un valor *Char* a una variable, además de la función *Chr*, podemos hacerlo con un literal, ese valor literal estará encerrado entre comillas dobles, (al igual que una cadena), aunque para que realmente sea un carácter debemos agregarle una **c** justo después del cierre de las comillas dobles:

```
Dim c As Char = "A"c
```

Obligar a declarar las variables con el tipo de datos

Visual Basic 2010 nos permite, (lamentablemente de forma predeterminada), utilizar las variables

y constantes sin necesidad de indicar el tipo de datos de estas, pero, como comentábamos al principio, podemos obligar a que nos avise cuando no lo estamos haciendo, ya que como decíamos en la nota, es una buena costumbre indicar **siempre** el tipo de datos que tendrán nuestras variables y constantes.

Esa obligatoriedad la podemos aplicar a todo el proyecto o a un módulo en particular, para ello tenemos que usar la instrucción *Option Strict On*, una vez indicado, se aplicará a todo el código, no solo a las declaraciones de variables, constantes o al tipo de datos devuelto por las funciones y propiedades, sino también a las conversiones y asignaciones entre diferentes tipos de datos.

No debemos confundir *Option Strict* con *Option Explicit*, este último, sirve para que siempre tengamos que declarar todas las variables, mientras que el primero lo que hace es obligarnos a que esas declaraciones tengan un tipo de datos.

Tanto una como la otra tienen dos estados: conectado o desconectado dependiendo de que agreguemos *On* u *Off* respectivamente.

Insistimos en la recomendación de que siempre debemos "conectar" estas dos opciones, si bien *Option Explicit On* ya viene como valor por defecto, cosa que no ocurre con *Option Strict*, que por defecto está desconectado.

Aplicar Option Strict On a un fichero en particular

Cuando agregábamos un nuevo fichero a nuestro proyecto de Visual Basic 2010 si ya tenemos predefinida las opciones "estrictas", como es el caso de *Option Explicit On*, estas no se añadirán a dicho fichero, (en un momento veremos cómo hacerlo para que siempre estén predefinidas), pero eso no significa que no se aplique, aunque siempre podemos escribir esas instrucciones (con el valor *On* al final) en cada uno de los ficheros de código que agreguemos a nuestro proyecto. Si nos decidimos a añadirlas a los ficheros, esas líneas de código deben aparecer al principio del fichero y solamente pueden estar precedidas de comentarios.

En la figura 2.1 mostrada en la lección anterior, tenemos una captura del editor de Visual Basic 2010 en la que hemos indicado que queremos tener comprobación estricta.

Aplicar Option Strict On a todo el proyecto

También podemos hacer que *Option Strict* funcione igual que *Option Explicit*, es decir, que esté activado a todo el proyecto, en este caso no tendríamos que indicarlo en cada uno de los ficheros de código que formen parte de nuestro proyecto, si bien solamente será aplicable a los que no tengan esas instrucciones, aclaremos esto último: si *Option Strict* (u *Option Explicit*) está definido de forma global al proyecto, podemos desactivarlo en cualquiera de los ficheros, para ello simplemente habría que usar esas declaraciones pero usando *Off* en lugar de *On*. De igual forma, si ya está definido globalmente y lo indicamos expresamente, no se producirá ningún error. Lo importante aquí es saber que siempre se usará el estado indicado en cada fichero, independientemente de cómo lo tengamos definido a nivel de proyecto.

Para que siempre se usen estas asignaciones en todo el proyecto, vamos a ver cómo indicarlo en el entorno de Visual Basic 2010.

Abrimos Visual Studio 2010 y una vez que se haya cargado, (no hace falta crear ningún nuevo proyecto, de este detalle nos ocuparemos en breve), seleccionamos la opción **Herramientas>Opciones...** se mostrará un cuadro de diálogo y del panel izquierdo seleccionamos la opción **Proyectos y soluciones**, la expandimos y seleccionamos **Valores predeterminados de VB** y veremos ciertas opciones, tal como podemos comprobar en la figura 2.2:

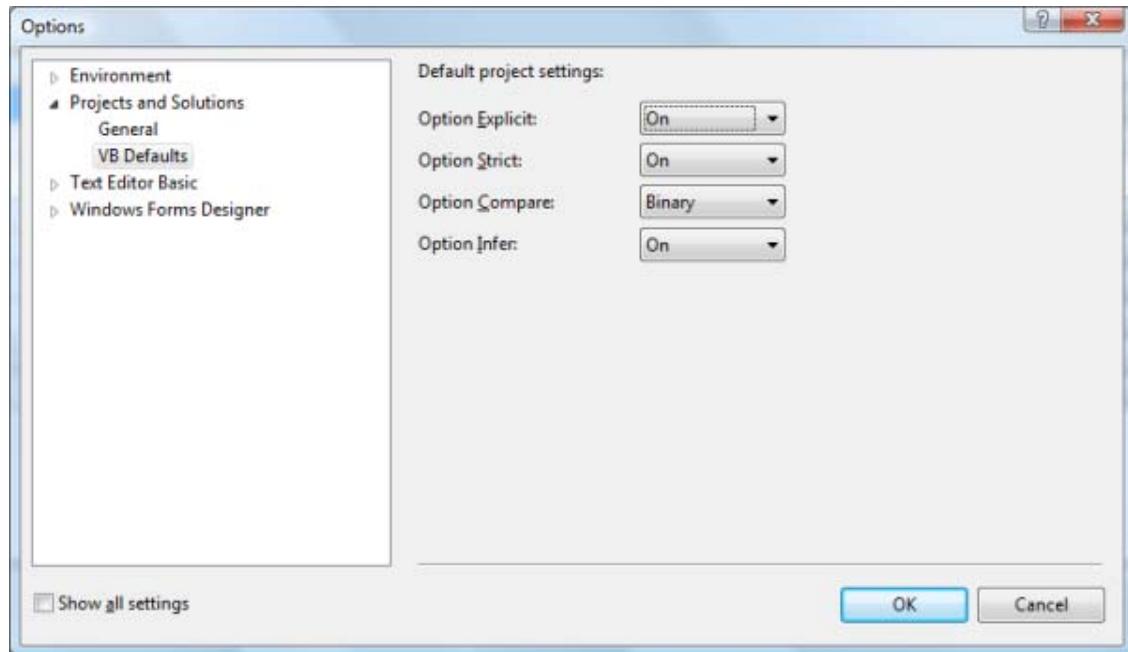


Figura 2.2. Opciones de proyectos (opciones mínimas)

De la lista desplegable **Option Strict**, seleccionamos **On**. Por defecto ya estarán seleccionadas las opciones **On** de **Option Explicit** y **Binary** de **Option Compare**, por tanto no es necesario realizar ningún cambio más, para aceptar los cambios y cerrar el cuadro de diálogo, presionamos en el botón **Aceptar**.

Si en la ventana de opciones no aparece toda la configuración podemos hacer que se muestren todas las disponibles. Para hacerlo, debemos marcar la casilla que está en la parte inferior izquierda en la que podemos leer: **Mostrar todas las configuraciones**, al seleccionar esa opción nos mostrará un número mayor de opciones, tal como podemos ver en la figura 2.3:

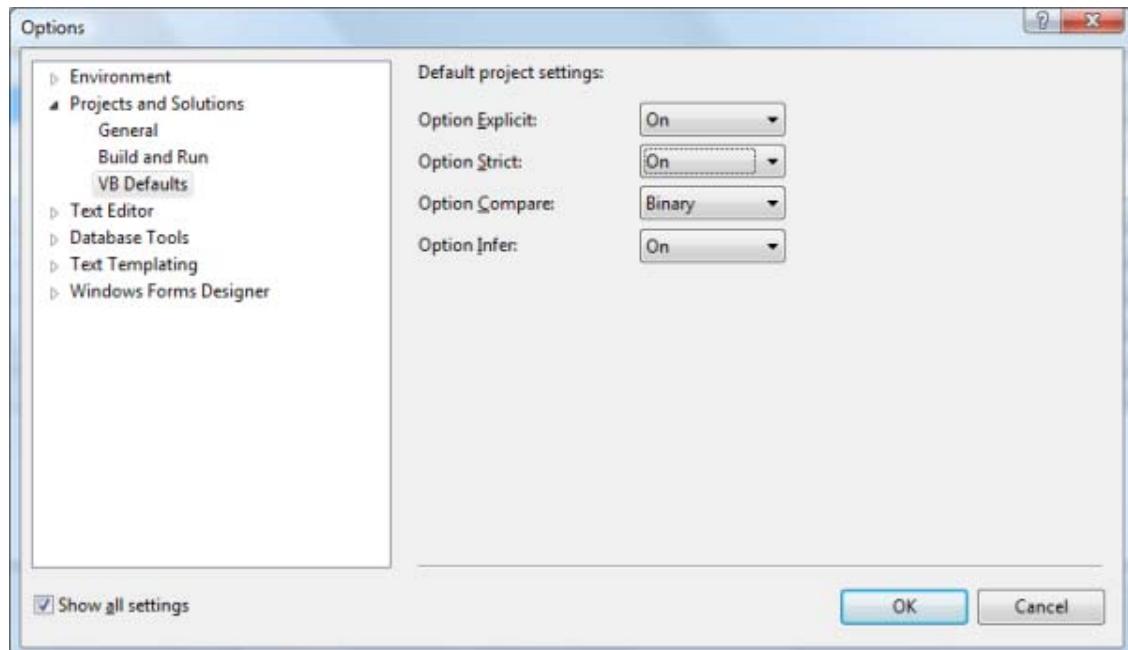


Figura 2.3. Opciones de proyectos (todas las opciones)

Desde este momento el compilador de Visual Basic se volverá estricto en todo lo relacionado a las declaraciones de variables y conversiones, tal como vemos en la figura 2.4 al intentar declarar una variable sin indicar el tipo de datos.

```
Option Strict On

Module Module1

    Sub Main()
        Dim m As Decimal = 12345678901234567890D

        Dim a
        Option Strict On requiere que todas las declaraciones de variables tengan una cláusula 'As'.

    End Sub

End Module
```

Figura 2.4. Aviso de Option Strict al declarar una variable sin tipo

Nota:

Una de las ventajas del IDE (*Integrated Development Environment*, entorno de desarrollo integrado) de Visual Basic 2010 es que nos avisa al momento de cualquier fallo que cometamos al escribir el código, este "pequeño" detalle, aunque alguna veces puede llegar a parecer fastidioso, nos facilita la escritura de código, ya que no tenemos que esperar a realizar la compilación para que tengamos constancia de esos fallos.

Más opciones aplicables a los proyectos

Aunque en estos módulos no trataremos a fondo el entorno de desarrollo, ya que la finalidad de este curso online es tratar más en el código propiamente dicho, vamos a mostrar otro de los sitios en los que podemos indicar dónde indicar que se haga una comprobación estricta de tipos y, como veremos, también podremos indicar algunas "nuevas peculiaridades" de Visual Basic 2010, todas ellas relacionadas con el tema que estamos tratando.

Cuando tengamos un proyecto cargado en el IDE de Visual Studio 2010, (pronto veremos cómo crear uno), podemos mostrar las propiedades del proyecto, para ello seleccionaremos del menú **Proyecto** la opción **Propiedades de <NombreDelProyecto>** y tendremos un cuadro de diálogo como el mostrado en la figura 2.5.

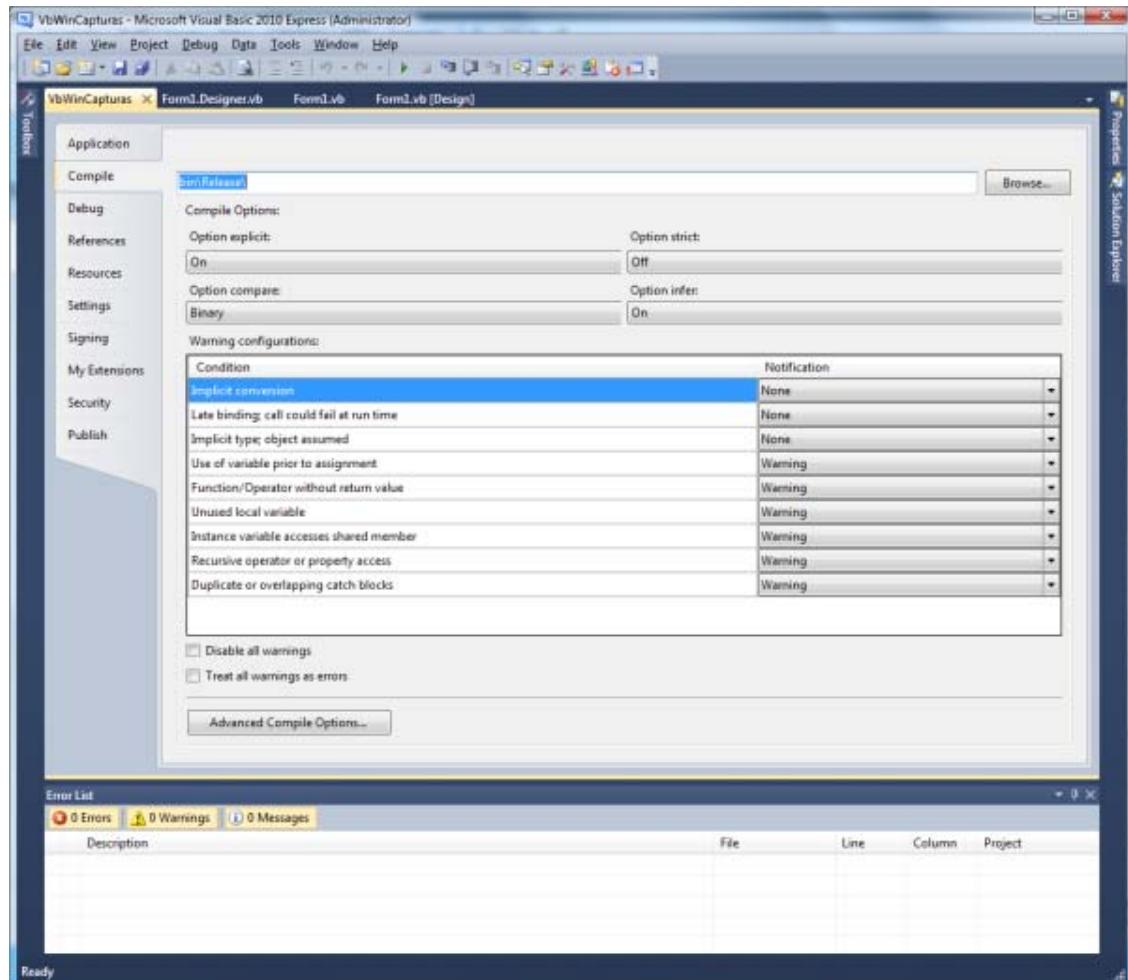


Figura 2.5. Ficha Compilar de las opciones del proyecto actual

Seleccionando la ficha **Compilar**, además de las típicas opciones de **Option Strict**, **Option Explicit**, **Option Compare** y **Option Infer**, (estas asignaciones solo serán efectivas para el proyecto actual), tendremos cómo queremos que reaccione el compilador si se cumple algunas de las condiciones indicadas. Entre esas condiciones, tenemos algo que muchos desarrolladores de Visual Basic siempre hemos querido tener: Que nos avise cuando una variable la hemos declarado pero no la utilizamos (**Variable local no utilizada**). Al tener marcada esta opción (normalmente como una **Advertencia**), si hemos declarado una variable y no la usamos en el código, (siempre que no le hayamos asignado un valor al declararla), nos avisará, tal como podemos ver en la figura 2.6:

```

    Option Strict On

    Module Module1
        Sub Main()
            Dim m As Decimal = 12345678901234567890D
            Dim a As Integer
            |   Variable local sin utilizar: 'a'.
        End Sub
    End Module

```

Figura 2.6. Aviso de variable no usada

Tipos Nullables

Otra interesantísima característica introducida en Visual Basic 2008 y utilizable en Visual Basic 2010 que conviene conocer, es lo que se denominan tipos Nullables o anulables.

Los tipos anulables no son nuevos en Visual Basic, de hecho su origen lo encontramos en Visual Studio 2005, aunque eso sí, implementando la clase *Nullable(Of T)*.

Con Visual Basic 2008 y Visual Basic 2010 no es necesario declarar ninguna clase para implementar tipos anulables dentro de nuestras aplicaciones, y podemos declararlos de forma directa.

Un tipo de dato anulable nos permitirá declarar una variable que podrá tener un tipo de dato nulo.

Si hemos estado atentos hasta ahora, hemos podido ver que las variables numéricas por ejemplo, se inicializan a 0.

Si quisieramos por la razón que fuera, declarar esa variable como nula para saber si en un determinado momento ha cambiado de valor o cualquier otra acción, deberíamos utilizar los tipos de datos anulables, o bien, utilizar técnicas más rudimentarias como una variable de tipo *Boolean* que nos permitiera saber si ha habido un cambio de valor en una variable, sin embargo, coincidiría conmigo en que el uso de un tipo de datos anulable es más natural y directo.

De esta forma, podríamos trabajar con tipos de datos anulables o que los declararemos como nulos. A continuación veremos un ejemplo:

```
Dim valor As Integer?
```

Para acceder al valor de un tipo anulable, podríamos hacerlo de la forma habitual, ahora bien, si no sabemos si el valor es nulo o no, podríamos acceder a su valor preguntando por él mediante la propiedad *HasValue*. La propiedad *Value* nos indicará también, el valor de esa variable.

Un ejemplo que aclare esta explicación es el que podemos ver a continuación:

```
Dim valor As Integer?  
If Valor.HasValue Then  
    MessageBox.Show(valor.Value)  
End If
```

Otra característica de los tipos anulables es la posibilidad de utilizar la función *GetValueOrDefault*. Esta función nos permitirá acceder al valor de la variable si no es nulo, y al valor que le indiquemos si es nulo.

Un breve ejemplo de este uso es el que se indica a continuación:

```
Dim valor As Integer?  
valor = 2010  
MessageBox.Show(valor.GetValueOrDefault(2012))  
End If
```

En este ejemplo, el compilador nos devolvería el valor *2010*, ya que *GetValueOrDefault* sabe que la variable no posee un valor nulo y que por lo tanto, debe obtener el valor no nulo de la variable anulable.

En el caso de que no hubiéramos dado ningún valor a la variable, la aplicación obtendría el valor *2012*.

Tipos anónimos

Esta característica de Visual Basic 2008 y Visual Basic 2010, nos permite declarar los tipos de datos de forma implícita desde el código de nuestras aplicaciones.

Un ejemplo práctico de declaración de tipos anónimos es el siguiente:

```
Dim declaracion = New With { .Nombre = "Carlos", .Edad = 27}  
MessageBox.Show(String.Format("{0} tiene {1} años", _  
    declaracion.Nombre, declaracion.Edad))
```

Como podemos ver, en el ejemplo anterior hemos declarado un objeto al que no hemos indicado ningún tipo de dato concreto, pero a la hora de crear ese objeto, hemos creado implícitamente un miembro *Nombre* y un miembro *Edad*.

Propiedades autoimplementadas

Esta característica de Visual Basic 2010 nos permite declarar propiedades y autoimplementarlas con un valor determinado.

De esta manera, podremos inicializar una propiedad con un valor determinado.

Un sencillo ejemplo de uso de esta característica es el siguiente:

```
Property Id As Integer = 0
Property NamePerson As String = "<sin nombre>" 
Property AgePerson As Byte = 0
```

En este ejemplo, se han creado tres propiedades, una de ellas de tipo Int32, otra de tipo String, y una tercera de tipo Byte.

Estas propiedades están inicializadas con valores por defecto.

Inicialización de colecciones

Otra característica muy útil en Visual Basic es la inicialización de colecciones, que nos permite inicializar una colección de datos con valores de forma directa.

La mejor forma de entender esto es viendo un ejemplo.

Supongamos la siguiente interfaz en Visual Basic:

```
Interface IPerson

    Property Id As Integer
    Property Name As String
    Property Age As Byte

End Interface
```

Ahora, crearemos una clase que implemente la interfaz *IPerson*:

```
Public Class Estudiante
    Implements IPerson

    <DefaultValue("0")>
    Property Id As Integer Implements IPerson.Id

    Property Name As String Implements IPerson.Name

    Property Age As Byte Implements IPerson.Age

End Class
```

Ya tenemos la interfaz y la clase que implementa la interfaz, por lo que ahora, lo tenemos que hacer es consumir la clase *Estudiante* e inicializar los elementos de la colección.

Un ejemplo práctico de inicialización es el que se indica a continuación:

```
Property StudentCollection As New List(Of Estudiante)

    ...

    Me.StudentCollection.Add(New Estudiante With {.Id = 1, .Name = "Luisa", .Age = 25})
    Me.StudentCollection.Add(New Estudiante With {.Id = 2, .Name = "Antonio", .Age = 27})
    Me.StudentCollection.Add(New Estudiante With {.Id = 3, .Name = "Juan", .Age = 26})
```

Como podemos apreciar, al mismo tiempo que declaramos un nuevo objeto de tipo Estudiante, inicializamos sus propiedades con valores, de forma que asignamos esos elementos a la colección con valores directos.

Lección 1: El sistema de tipos

- Tipos primitivos
- Variables y constantes
- Enumeraciones**
- Arrays (matrices)

Enumeraciones: Constantes agrupadas

Una enumeración es una serie de constantes que están relacionadas entre sí. La utilidad de las enumeraciones es más manifiesta cuando queremos manejar una serie de valores constantes con nombre, es decir, podemos indicar un valor, pero en lugar de usar un literal numérico, usamos un nombre, ese nombre es al fin y al cabo, una constante que tiene un valor numérico.

En Visual Basic 2010 las enumeraciones pueden ser de cualquier tipo numérico integral, incluso enteros sin signo, aunque el valor predefinido es el tipo *Integer*. Podemos declarar una enumeración de varias formas:

1- Sin indicar el tipo de datos, por tanto serán de tipo *Integer*:

```
Enum Colores
    Rojo
    Verde
    Azul
End Enum
```

2- Concretando explícitamente el tipo de datos que realmente tendrá:

```
Enum Colores As Long
    Rojo
    Verde
    Azul
End Enum
```

En este segundo caso, el valor máximo que podemos asignar a los miembros de una enumeración será el que pueda contener un tipo de datos *Long*.

3- Indicando el atributo *FlagsAttribute*, (realmente no hace falta indicar el sufijo *Attribute* cuando usamos los atributos) de esta forma podremos usar los valores de la enumeración para indicar valores que se pueden "sumar" o complementar entre sí, pero sin perder el nombre, en breve veremos qué significa esto de "no perder el nombre".

```
<Flags()> _
Enum Colores As Byte
    Rojo = 1
    Verde = 2
```

```
Azul = 4  
End Enum
```

Nota:

Los atributos los veremos con más detalle en otra lección de este mismo módulo.

El nombre de los miembros de las enumeraciones

Tanto si indicamos o no el atributo *Flags* a una enumeración, la podemos usar de esta forma:

```
Dim c As Colores = Colores.Azul Or Colores.Rojo
```

Es decir, podemos "sumar" los valores definidos en la enumeración. Antes de explicar con detalle que beneficios nos puede traer el uso de este atributo, veamos una característica de las enumeraciones.

Como hemos comentado, las enumeraciones son constantes con nombres, pero en Visual Basic 2010 esta definición llega más lejos, de hecho, podemos saber "el nombre" de un valor de una enumeración, para ello tendremos que usar el método *ToString*, el cual se usa para convertir en una cadena cualquier valor numérico.

Por ejemplo, si tenemos la siguiente asignación:

```
Dim s As String = Colores.Azul.ToString
```

La variable **s** contendrá la palabra "**Azul**" no el valor 4.

Esto es aplicable a cualquier tipo de enumeración, se haya o no usado el atributo *FlagsAttribute*.

Una vez aclarado este comportamiento de las enumeraciones en Visual Basic 2010, veamos que es lo que ocurre cuando sumamos valores de enumeraciones a las que hemos aplicado el atributo *Flags* y a las que no se lo hemos aplicado. Empecemos por este último caso.

Si tenemos este código:

```
Enum Colores As Byte  
    Rojo = 1  
    Verde = 2  
    Azul = 4  
End Enum  
  
Dim c As Colores = Colores.Azul Or Colores.Rojo  
  
Dim s As String = c.ToString
```

El contenido de la variable **s** será "**5**", es decir, la representación numérica del valor contenido: **4 + 1**, ya que el valor de la constante **Azul** es **4** y el de la constante **Rojo** es **1**.

Pero si ese mismo código lo usamos de esta forma (aplicando el atributo *Flags* a la enumeración):

```
<Flags()> _  
Enum Colores As Byte  
    Rojo = 1  
    Verde = 2  
    Azul = 4  
End Enum  
  
Dim c As Colores = Colores.Azul Or Colores.Rojo
```

```
Dim s As String = c.ToString
```

El contenido de la variable **s** será: "**Rojo, Azul**", es decir, se asignan los nombres de los miembros de la enumeración que intervienen en ese valor, no el valor "interno".

Los valores de una enumeración no son simples números

Como hemos comentado, los miembros de las enumeraciones realmente son valores de un tipo de datos entero (en cualquiera de sus variedades) tal como podemos comprobar en la figura 2.7:

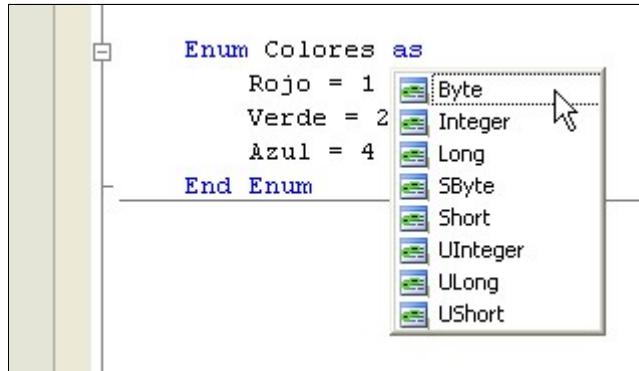


Figura 2.7. Los tipos subyacentes posibles de una enumeración

Por tanto, podemos pensar que podemos usar cualquier valor para asignar a una variable declarada como una enumeración, al menos si ese valor está dentro del rango adecuado. En Visual Basic 2010 esto no es posible, al menos si lo hacemos de forma "directa" y con **Option Strict** conectado, ya que recibiremos un error indicándonos que no podemos convertir, por ejemplo, un valor entero en un valor del tipo de la enumeración. En la figura 2.8 podemos ver ese error al intentar asignar el valor **3** a una variable del tipo **Colores** (definida con el tipo predeterminado *Integer*).

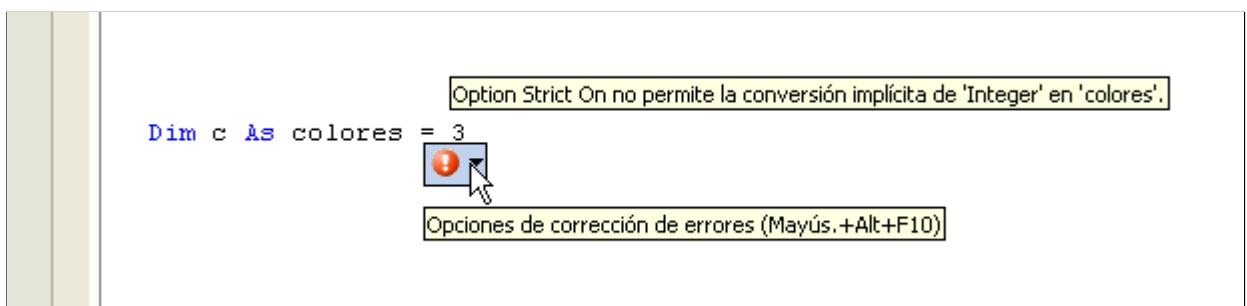


Figura 2.8. Error al asignar un valor "normal" a una variable del tipo Colores

El error nos indica que no podemos realizar esa asignación, pero el entorno integrado de Visual Studio 2010 también nos ofrece alternativas para que ese error no se produzca, esa ayuda se obtiene presionando en el signo de admiración que tenemos justo donde está el cursor del mouse, pero no solo nos dice cómo corregirlo, sino que también nos da la posibilidad de que el propio IDE se encargue de corregirlo, tal como podemos apreciar en la figura 2.9.

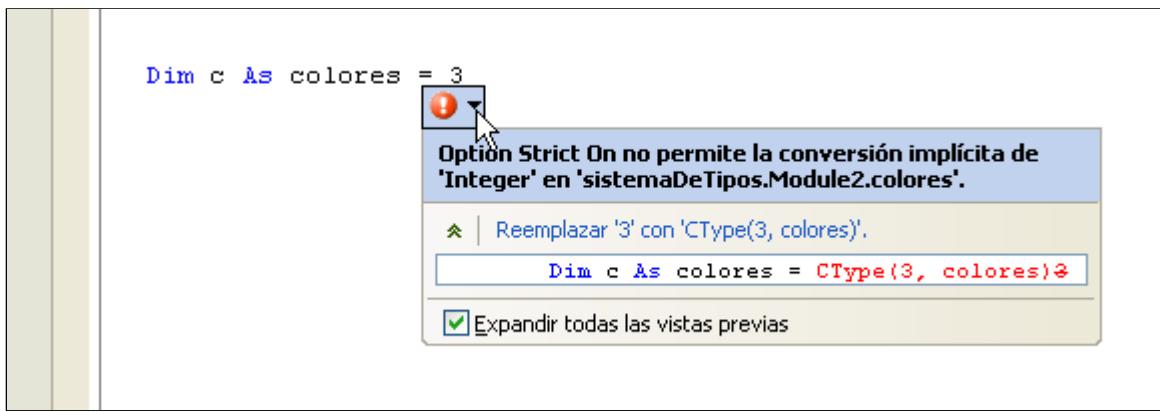


Figura 2.9. Opciones de corrección de errores

Lo único que tendríamos que hacer es presionar en la sugerencia de corrección, que en este caso es la única que hay, pero en otros casos pueden ser varias las opciones y tendríamos que elegir la que creamos adecuada.

El código final (una vez corregido) quedaría de la siguiente forma:

```
Dim c As Colores = CType(3, Colores)
```

CType es una de las formas que nos ofrece Visual Basic 2010 de hacer conversiones entre diferentes tipos de datos, en este caso convertimos un valor entero en uno del tipo **Colores**.

Si compilamos y ejecutamos la aplicación, ésta funcionará correctamente.

Aunque sabemos que es posible que usando *CType* no asignemos un valor dentro del rango permitido. En este caso, el valor 3 podríamos darlo por bueno, ya que es la suma de 1 y 2 (**Rojo** y **Verde**), pero ¿que pasaría si el valor asignado es, por ejemplo, 15? En teoría no deberíamos permitirlo.

Estas validaciones podemos hacerlas de dos formas:

- 1- Con la clásica solución de comprobar el valor indicado con todos los valores posibles.
- 2- Usando funciones específicas del tipo *Enum*. Aunque en este último caso, solo podremos comprobar los valores definidos en la enumeración.

En el siguiente ejemplo podemos hacer esa comprobación.

```
Sub mostrarColor(ByVal c As Colores)
    ' comprobar si el valor indicado es correcto
    ' si no está; definido, usar el valor Azul
    If [Enum].IsDefined(GetType(Colores), c) = False Then
        c = Colores.Azul
    End If
    Console.WriteLine("El color es {0}", c)
End Sub
```

Este código lo que hace es comprobar si el tipo de datos **Colores** tiene definido el valor contenido en la variable **c**, en caso de que no sea así, usamos un valor predeterminado.

Nota:

La función *IsDefined* sólo comprueba los valores que se han definido en la enumeración, no las posibles combinaciones que podemos conseguir sumando cada uno de sus miembros, incluso aunque hayamos usado el atributo *FlagsAttribute*.

Lección 1: El sistema de tipos

- Tipos primitivos
- Variables y constantes
- Enumeraciones
- Arrays (matrices)**

Arrays (matrices)

Los arrays (o matrices) nos permitirán agrupar valores que de alguna forma queremos que estén relacionados entre sí.

Nota:

Esta es la definición usada en la documentación de Visual Studio sobre qué es una matriz:

"Una matriz es una estructura de datos que contiene una serie de variables denominadas elementos de la matriz."

Aclaramos este punto, porque la traducción en castellano de Array puede variar dependiendo del país (arreglos, matrices, etc). Aquí utilizaremos la usada a lo largo de la documentación de Visual Studio.

Declarar arrays

En C# los arrays se definen indicando un par de corchetes en el tipo de datos.

En Visual Basic 2010 la declaración de un array la haremos usando un par de paréntesis en el nombre de la variable o del tipo, en el siguiente ejemplo declaramos un array de tipo *String* llamado **nombres**:

```
Dim nombres() As String  
Dim nombres As String()
```

Estas dos formas son equivalentes.

También podemos indicar el número de elementos que contendrá el array o matriz:

```
Dim nombres(10) As String
```

Pero solo podemos hacerlo en el nombre, si esa cantidad de elementos lo indicamos en el tipo, recibiremos un error indicándonos que *"los límites de la matriz no pueden aparecer en los especificadores del tipo"*.

Al declarar un array indicando el número de elementos, como es el caso anterior, lo que estamos definiendo es un array de 11 elementos: desde cero hasta 10, ya que en Visual

Basic 2010, al igual que en el resto de lenguajes de .NET, **todos los arrays deben tener como índice inferior el valor cero**.

Para que quede claro que el límite inferior debe ser cero, en Visual Basic 2010 podemos usar la instrucción **0 To** para indicar el valor máximo del índice superior, ya que, tal como podemos comprobar si vemos 0 To 10, quedará claro que nuestra intención es declarar un array con 11 elementos, o al menos nuestro código resultará más legible:

```
Dim nombres(0 To 10) As String
```

Declarar e inicializar un array

En Visual Basic 2010 también podemos inicializar un array al declararlo, para ello debemos poner los valores a asignar dentro de un par de llaves, tal como vemos en el siguiente ejemplo:

```
Dim nombres() As String = {"Pepe", "Juan", "Luisa"}
```

Con el código anterior estamos creando un array de tipo *String* con tres valores cuyos índices van de cero a dos.

En este caso, cuando iniciamos el array al declararlo, no debemos indicar el número de elementos que tendrá ese array, ya que ese valor lo averiguará el compilador cuando haga la asignación. Tampoco es válido indicar el número de elementos que queremos que tenga y solo asignarle unos cuantos menos (o más), ya que se producirá un error en tiempo de compilación.

Si el array es bidimensional (o con más dimensiones), también podemos inicializarlos al declararlo, pero en este caso debemos usar doble juego de llaves:

```
Dim nombres(,) As String = {{"Juan", "Pepe"}, {"Ana", "Eva"}}
```

En este código tendríamos un array bidimensional con los siguientes valores:

```
nombres(0,0)= Juan  
nombres(0,1)= Pepe  
nombres(1,0)= Ana  
nombres(1,1)= Eva
```

Como podemos ver en la declaración anterior, si definimos arrays con más de una dimensión, debemos indicarlas usando una coma para separar cada dimensión, o lo que es más fácil de recordar: usando una coma menos del número de dimensiones que tendrá el array. En los valores a asignar, usaremos las llaves encerradas en otras llaves, según el número de dimensiones.

Aunque, la verdad, es que hay algunas veces hay que hacer un gran esfuerzo mental para asociar los elementos con los índices que tendrán en el array, por tanto, algunas veces puede que resulte más legible si indentamos o agrupamos esas asignaciones, tal como vemos en el siguiente código:

```
Dim nomTri,,, As String = _  
    { _  
        { {"Juan", "Pepe"}, {"Luisa", "Eva"}}, _  
        { {"A", "B"}, {"C", "D"} } _  
    }  
  
Console.WriteLine(nomTri(0, 0, 0)) ' Juan  
Console.WriteLine(nomTri(0, 0, 1)) ' Pepe  
Console.WriteLine(nomTri(0, 1, 0)) ' Luisa  
Console.WriteLine(nomTri(0, 1, 1)) ' Eva
```

```
Console.WriteLine(nomTri(1, 0, 0)) ' A
Console.WriteLine(nomTri(1, 0, 1)) ' B
Console.WriteLine(nomTri(1, 1, 0)) ' C
Console.WriteLine(nomTri(1, 1, 1)) ' D
```

Tal como podemos comprobar, así es más legible. Por suerte tenemos el carácter del guión bajo para continuar líneas de código, aunque esto ya no es necesario como hemos podido ver al principio de este tutorial.

Cambiar el tamaño de un array

Para cambiar el tamaño de un array, usaremos la instrucción *ReDim*, esta instrucción solo la podemos usar para cambiar el tamaño de un array previamente declarado, no para declarar un array, ya que siempre hay que declarar previamente los arrays antes de cambiarles el tamaño.

```
Dim nombres() As String
...
ReDim nombres(3)
nombres(0) = "Juan"
nombres(1) = "Pepe"
```

La mayor utilidad de esta instrucción, es que podemos cambiar el tamaño de un array y mantener los valores que tuviera anteriormente, para lograrlo debemos usar *ReDim Preserve*.

```
ReDim Preserve nombres(3)
nombres(2) = "Ana"
nombres(3) = "Eva"
```

En este ejemplo, los valores que ya tuviera el array **nombres**, se seguirían manteniendo, y se asignarían los nuevos.

Si bien tanto *ReDim* como *ReDim Preserve* se pueden usar en arrays de cualquier número de dimensiones, en los arrays de más de una dimensión solamente podemos cambiar el tamaño de la última dimensión.

Eliminar el contenido de un array

Una vez que hemos declarado un array y le hemos asignado valores, es posible que nos interese eliminar esos valores de la memoria, para lograrlo, podemos hacerlo de tres formas:

1. Redimensionando el array indicando que tiene cero elementos, aunque en el mejor de los casos, si no estamos trabajando con arrays de más de una dimensión, tendríamos un array de un elemento, ya que, como hemos comentado anteriormente, los arrays de .NET el índice inferior es cero.
2. Usar la instrucción *Erase*. La instrucción *Erase* elimina totalmente el array de la memoria.
3. Asignar un valor *Nothing* al array. Esto funciona en Visual Basic 2010 porque los arrays realmente son tipos por referencia.

Los arrays son tipos por referencia

Como acabamos de ver, en Visual Basic 2010 los arrays son tipos por referencia, y tal como comentamos anteriormente, los tipos por referencia realmente lo que contienen son una referencia a los datos reales no los datos propiamente dichos.

¿Cuál es el problema?

Veámoslo con un ejemplo y así lo tendremos más claro.

```
Dim nombres() As String = {"Juan", "Pepe", "Ana", "Eva"}  
Dim otros() As String  
  
otros = nombres  
  
nombres(0) = "Antonio"
```

En este ejemplo definimos el array **nombres** y le asignamos cuatro valores. A continuación definimos otro array llamado **otros** y le asignamos lo que tiene **nombres**. Por último asignamos un nuevo valor al elemento cero del array **nombres**.

Si mostramos el contenido de ambos arrays nos daremos cuenta de que realmente solo existe una copia de los datos en la memoria, y tanto **nombres(0)** como **otros(0)** contienen el nombre "**Antonio**".

¿Qué ha ocurrido?

Que debido a que los arrays son tipos por referencia, solamente existe una copia de los datos y tanto la variable **nombres** como la variable **otros** lo que contienen es una referencia (o puntero) a los datos.

Si realmente queremos tener copias independientes, debemos hacer una copia del array **nombres** en el array **otros**, esto es fácil de hacer si usamos el método *CopyTo*. Éste método existe en todos los arrays y nos permite copiar un array en otro empezando por el índice que indiquemos.

El único requisito es que el array de destino debe estar inicializado y tener espacio suficiente para contener los elementos que queremos copiar.

En el siguiente código de ejemplo hacemos una copia del contenido del array **nombres** en el array **otros**, de esta forma, el cambio realizado en el elemento cero de **nombres** no afecta al del array **otros**.

```
Dim nombres() As String = {"Juan", "Pepe", "Ana", "Eva"}  
Dim otros() As String  
  
ReDim otros(nombres.Length)  
  
nombres.CopyTo(otros, 0)  
  
nombres(0) = "Antonio"
```

Además del método *CopyTo*, los arrays tienen otros miembros que nos pueden ser de utilidad, como por ejemplo la propiedad *Length* usada en el ejemplo para saber cuantos elementos tiene el array **nombres**.

Para averiguar el número de elementos de un array, (realmente el índice superior), podemos usar la función *UBound*.

También podemos usar la función *LBound*, (que sirve para averiguar el índice inferior de un array), aunque no tiene ningún sentido en Visual Basic 2010, ya que, como hemos comentado, todos los arrays siempre tienen un valor cero como índice inferior.

Para finalizar este tema, solo nos queda por decir, que los arrays de Visual Basic 2010 realmente son tipos de datos derivados de la clase *Array* y por tanto disponen de todos los miembros definidos en esa clase, aunque de esto hablaremos en la próxima lección, en la que también tendremos la oportunidad de profundizar un poco más en los tipos por referencia y en como podemos definir nuestros propios tipos de datos, tanto por referencia como por valor.

Literales de arrays (o matrices)

Otra característica de Visual Basic 2010 es la aplicación de los denominados literales de arrays o matrices.

La idea es que Visual Basic infiera por nosotros el tipo de dato de forma directa.

Esto representa una buena práctica aunque nos impide hacer cosas que antes podíamos hacer sin complicaciones.

Por ejemplo, pensemos en el siguiente ejemplo de Visual Basic 2008:

```
Dim ejemploMatriz() = {2009, 2010, 2011, 2012}
```

En este ejemplo, habremos creado una matriz de 4 elementos. Sin embargo, podríamos cambiar uno de sus elementos a otro valor de diferente tipo, como por ejemplo:

```
ejemploMatriz(1) = "s"
```

De esta manera, tendremos un resultado: 2009, "s", 2011, 2012.

Esto es así, porque la matriz *ejemploMatriz* es de tipo *Object*.

Sin embargo, esto ha cambiado. Ahora, podemos declarar literales de matrices, haciendo que estas matrices se declaren de acuerdo al tipo de dato declarado, es decir, infiriendo el tipo de dato en la declaración.

Un ejemplo de este uso es el siguiente:

```
Dim ejemploInteger = {2009, 2010, 2011, 2012}
```

En este ejemplo, si tratamos de hacer lo que antes intentábamos mediante la siguiente intrucción:

```
ejemploInteger(1) = "s"
```

La aplicación nos devolverá un error en tiempo de ejecución, ya que la matriz estará inferida por defecto como tipo *Integer*, y estaremos intentando asignar un valor String a una matriz de tipo Integer.

Lección 2: Clases y estructuras

- Clases
- Definir una clase
- Instanciar una clase
- Estructuras
- Accesibilidad
- Propiedades
- Interfaces

Introducción

En [la lección anterior](#) vimos los tipos de datos predefinidos en .NET Framework, en esta lección veremos cómo podemos crear nuestros propios tipos de datos, tanto por valor como por referencia.

También tendremos ocasión de ver los distintos niveles de accesibilidad que podemos aplicar a los tipos, así como a los distintos miembros de esos tipos de datos. De los distintos miembros que podemos definir en nuestros tipos, nos centraremos en las propiedades para ver en detalle los cambios que han sufrido con respecto a VB6. También veremos temas relacionados con la programación orientada a objetos (POO) en general y de forma particular los que atañen a las interfaces.

Clases y estructuras

• Clases: Tipos por referencia definidos por el usuario

- Las clases: El corazón de .NET Framework
- La herencia: Característica principal de la Programación Orientada a Objetos
- Encapsulación y Polimorfismo
- Object: La clase base de todas las clases de .NET

• Definir una clase

- Una clase especial: Module
- Los miembros de una clase
- Características de los métodos y propiedades
 - Accesibilidad, ámbito y miembros compartidos
 - Parámetros y parámetros opcionales
 - Array de parámetros opcionales (ParamArray)
 - Sobre carga de métodos y propiedades
 - Parámetros por valor y parámetros por referencia
 - Parámetros opcionales de tipo Nullable

• Instanciar: Crear un objeto en la memoria

- Declarar primero la variable y después instanciarla
- Declarar y asignar en un solo paso
- El constructor: El punto de inicio de una clase
- Constructores parametrizados

- Cuando Visual Basic 2010 no crea un constructor automáticamente
- El destructor: El punto final de la vida de una clase
- Inicialización directa de objetos

- **Estructuras: Tipos por valor definidos por el usuario**

- Definir una estructura
- Constructores de las estructuras
- Destructores de las estructuras
- Los miembros de una estructura
- Cómo usar las estructuras

- **Accesibilidad y ámbito**

- Ámbito
 - Ámbito de bloque
 - Ámbito de procedimiento
 - Ámbito de módulo
 - Ámbito de espacio de nombres
- La palabra clave Global
- Accesibilidad
 - Accesibilidad de las variables en los procedimientos
- Las accesibilidades predeterminadas
- Anidación de tipos
 - Los tipos anidables
 - El nombre completo de un tipo
 - Importación de espacios de nombres
 - Alias de espacios de nombres

- **Propiedades**

- Definir una propiedad
- Propiedades de solo lectura
- Propiedades de solo escritura
- Diferente accesibilidad para los bloques Get y Set
- Propiedades predeterminadas
 - Sobrecarga de propiedades predeterminadas

- **Interfaces**

- ¿Qué es una interfaz?
- ¿Qué contiene una interfaz?
- Una interfaz es un contrato
- Las interfaces y el polimorfismo
- Usar una interfaz en una clase
- Acceder a los miembros implementados
- Saber si un objeto implementa una interfaz
- Implementación de múltiples interfaces
- Múltiple implementación de un mismo miembro
- ¿Dónde podemos implementar las interfaces?
- Un ejemplo práctico usando una interfaz de .NET

Lección 2: Clases y estructuras

Clases

- Definir una clase
- Instanciar una clase
- Estructuras
- Accesibilidad
- Propiedades
- Interfaces

Clases: Tipos por referencia definidos por el usuario

Tal como vimos en la lección anterior, los tipos de datos se dividen en dos grupos: tipos por valor y tipos por referencia. Los tipos por referencia realmente son clases, de las cuales debemos crear una instancia para poder usarlas, esa instancia o copia, se crea siempre en la memoria lejana (*heap*) y las variables lo único que contienen es una referencia a la dirección de memoria en la que el CLR (*Common Language Runtime*, motor en tiempo de ejecución de .NET), ha almacenado el objeto recién creado.

En .NET Framework todo es de una forma u otra una clase, por tanto Visual Basic 2010 también depende de la creación de clases para su funcionamiento, ya que todo el código que escribamos debemos hacerlo dentro de una clase.

Antes de entrar en detalles sintácticos, veamos la importancia que tienen las clases en .NET Framework y como repercuten en las que podamos definir nosotros usando Visual Basic 2010.

Las clases: el corazón de .NET Framework

Prácticamente todo lo que podemos hacer en .NET Framework lo hacemos mediante clases. La librería de clases de .NET Framework es precisamente el corazón del propio .NET, en esa librería de clases está todo lo que podemos hacer dentro de este marco de programación; para prácticamente cualquier tarea que queramos realizar existen clases, y si no existen, las podemos definir nosotros mismos, bien ampliando la funcionalidad de alguna clase existente mediante la herencia, bien implementando algún tipo de funcionalidad previamente definida o simplemente creándolas desde cero.

La herencia: Característica principal de la Programación Orientada a Objetos

El concepto de Programación Orientada a Objetos (POO) es algo intrínseco al propio .NET Framework, por tanto es una característica que todos los lenguajes basados en este "marco de trabajo" tienen de forma predeterminada, entre ellos el Visual Basic 2010. De las características principales de la POO tenemos que destacar la herencia, que en breve podemos definir como una característica que nos permite ampliar la funcionalidad de una clase existente sin perder la que ya tuviera previamente. Gracias a la herencia, podemos

crear una nueva clase que se derive de otra, esta nueva clase puede cambiar el comportamiento de la clase base y/o ampliarlo, de esta forma podemos adaptar la clase, llamémosla, original para adaptarla a nuestras necesidades.

El tipo de herencia que .NET Framework soporta es la herencia simple, es decir, solo podemos usar una clase como base de la nueva, si bien, como veremos más adelante, podemos agregar múltiple funcionalidad a nuestra nueva clase. Esta funcionalidad nos servirá para aprovechar la funcionalidad de muchas de las clases existentes en .NET Framework, funcionalidad que solamente podremos aplicar si previamente hemos firmado un contrato que asegure a la clase de .NET que la nuestra está preparada para soportar esa funcionalidad, esto lo veremos dentro de poco con más detalle.

Encapsulación y Polimorfismo

La encapsulación y el polimorfismo son otras dos características de la programación orientada a objetos.

La encapsulación nos permite abstraer la forma que tiene de actuar una clase sobre los datos que contiene o manipula, para poder lograrlo se exponen como parte de la clase los métodos y propiedades necesarios para que podamos manejar esos datos sin tener que preocuparnos cómo se realiza dicha manipulación.

El polimorfismo es una característica que nos permite realizar ciertas acciones o acceder a la información de los datos contenidos en una clase de forma semi-anónima, al menos en el sentido de que no tenemos porqué saber sobre qué tipo objeto realizamos la acción, ya que lo único que nos debe preocupar es que podemos hacerlo, por la sencilla razón de que estamos usando ciertos mecanismos que siguen unas normas que están adoptadas por la clase.

El ejemplo clásico del polimorfismo es que si tengo un objeto que "sabe" cómo morder, da igual que lo aplique a un ratón o a un dinosaurio, siempre y cuando esas dos "clases" expongan un método que pueda realizar esa acción... y como decía la documentación de Visual Basic 5.0, siempre será preferible que nos muerda un ratón antes que un dinosaurio.

Object: La clase base de todas las clases de .NET

Todas las clases de .NET se derivan de la clase *Object*, es decir, lo indiquemos o no, cualquier clase que definamos tendrá el comportamiento heredado de esa clase. El uso de la clase *Object* como base del resto de las clases de .NET es la única excepción a la herencia simple soportada por .NET, ya que de forma implícita, todas las clases de .NET se derivan de la clase *Object* independientemente de que estén derivadas de cualquier otra.

Esta característica nos asegura que siempre podremos usar un objeto del tipo *Object* para acceder a cualquier clase de .NET, aunque no debemos abrumarnos todavía, ya que en el texto que sigue veremos con más detalle qué significado tiene esta afirmación.

De los miembros que tiene la clase *Object* debemos resaltar el método *ToString*, el cual ya lo vimos en la lección anterior cuando queríamos convertir un tipo primitivo en una cadena. Este método está pensado para devolver una representación en formato cadena de un objeto. El valor que obtengamos al usar este método dependerá de cómo esté definido en cada clase y por defecto lo que devuelve es el nombre completo de la clase, si bien en la mayoría de los casos el valor que obtendremos al usar este método debería ser más intuitivo, por ejemplo los tipos de datos primitivos tienen definido este método para devuelva el valor que contienen, de igual forma, nuestras clases también deberían devolver un valor adecuado al contenido almacenado. De cómo hacerlo, nos ocuparemos en breve.

Nota:

Todos los tipos de datos de .NET, ya sean por valor o por referencia siempre están derivados de la clase *Object*, por tanto podremos llamar a cualquiera de los métodos que están definidos en esa clase.

Aunque en el caso de los tipos de datos por valor, cuando queremos acceder a la clase Object que contienen, .NET Framework primero debe convertirla en un objeto por referencia (boxing) y cuando hemos dejado de usarla y queremos volver a asignar el dato a la variable por valor, tiene que volver a hacer la conversión inversa (unboxing).

Lección 2: Clases y estructuras

- Clases
- Definir una clase
- Instanciar una clase
- Estructuras
- Accesibilidad
- Propiedades
- Interfaces

Definir una clase

En Visual Basic 2010, todo el código que queramos escribir, lo tendremos que hacer en un fichero con la extensión **.vb**, dentro de ese fichero es donde escribiremos nuestro código, el cual, tal como dijimos anteriormente siempre estará incluido dentro de una clase, aunque un fichero de código de VB puede contener una o más clases, es decir, no está limitado a una clase por fichero.

En Visual Basic 2010 las clases se definen usando la palabra clave *Class* seguida del nombre de la clase, esa definición acaba indicándolo con *End Class*.

En el siguiente ejemplo definimos una clase llamada Cliente que tiene dos campos públicos.

```
Class Cliente
    Public Nombre As String
    Public Apellidos As String
End Class
```

Una vez definida la clase podemos agregar los elementos (o miembros) que creamos conveniente.

En el ejemplo anterior, para simplificar, hemos agregado dos campos públicos, aunque también podríamos haber definido cualquiera de los miembros permitidos en las clases.

Una clase especial: Module

En Visual Basic 2010 también podemos definir una clase especial llamada *Module*, este tipo de clase, como veremos, tiene un tratamiento especial.

La definición se hace usando la instrucción *Module* seguida del nombre a usar y acaba con *End Module*.

Cualquier miembro definido en un *Module* siempre estará accesible en todo el proyecto y para usarlos no tendremos que crear ningún objeto en memoria.

Las clases definidas con la palabra clave *Module* realmente equivalen a las clases en las que todos los miembros están compartidos y por tanto siempre disponibles a toda la aplicación.

De todos estos conceptos nos ocuparemos en las siguientes lecciones, pero es necesario explicar que existe este tipo de clase ya que será el tipo de datos que el IDE de Visual Basic 2010 usará al crear aplicaciones del tipo consola, ya que ese será el tipo de proyecto que crearemos para practicar con el código mostrado en este primer módulo.

Los miembros de una clase

Una clase puede contener cualquiera de estos elementos (miembros):

- Enumeraciones
- Campos
- Métodos (funciones o procedimientos)
- Propiedades
- Eventos

Las enumeraciones, como vimos en la lección anterior, podemos usarlas para definir valores constantes relacionados, por ejemplo para indicar los valores posibles de cualquier "característica" de la clase.

Los campos son variables usadas para mantener los datos que la clase manipulará.

Los métodos son las acciones que la clase puede realizar, normalmente esas acciones serán sobre los datos que contiene. Dependiendo de que el método devuelva o no un valor, podemos usar métodos de tipo *Function* o de tipo *Sub* respectivamente.

Las propiedades son las "características" de las clases y la forma de acceder "públicamente" a los datos que contiene. Por ejemplo, podemos considerar que el nombre y los apellidos de un cliente son dos características del cliente.

Los eventos son mensajes que la clase puede enviar para informar que algo está ocurriendo en la clase.

Características de los métodos y propiedades

Accesibilidad, ámbito y miembros compartidos

Aunque estos temas los veremos en breve con más detalle, para poder comprender mejor las características de los miembros de una clase (o cualquier tipo que definamos), daremos un pequeño adelanto sobre estas características que podemos aplicar a los elementos que definamos.

Accesibilidad y ámbito son dos conceptos que están estrechamente relacionados. Aunque en la práctica tienen el mismo significado, ya que lo que representan es la "cobertura" o alcance que tienen los miembros de las clases e incluso de las mismas clases que definamos.

Si bien cada uno de ellos tienen su propia "semántica", tal como podemos ver a continuación:

Ámbito

Es el alcance que la definición de un miembro o tipo puede tener. Es decir, cómo podemos acceder a ese elemento y desde dónde podemos accederlo.

El ámbito de un elemento de código está restringido por el "sitio" en el que lo hemos declarado. Estos *sitios* pueden ser:

- **Ámbito de bloque:** Disponible únicamente en el bloque de código en el que se ha declarado.
- **Ámbito de procedimiento:** Disponible únicamente dentro del procedimiento en el que se ha declarado.
- **Ámbito de módulo:** Disponible en todo el código del módulo, la clase o la estructura donde se ha declarado.
- **Ámbito de espacio de nombres:** Disponible en todo el código del espacio de nombres.

Accesibilidad

A los distintos elementos de nuestro código (ya sean clases o miembros de las clases) podemos darle diferentes tipos de accesibilidad. Estos tipos de "acceso" dependerán del ámbito que queramos que tengan, es decir, desde dónde podremos accederlos.

Los modificadores de accesibilidad son:

- **Public:** Acceso no restringido.
- **Protected:** Acceso limitado a la clase contenedora o a los tipos derivados de esta clase.
- **Friend:** Acceso limitado al proyecto actual.
- **Protected Friend:** Acceso limitado al proyecto actual o a los tipos derivados de la clase contenedora.
- **Private:** Acceso limitado al tipo contenedor.

Por ejemplo, podemos declarar miembros privados a una clase, en ese caso, dichos miembros solamente los podremos acceder desde la propia clase, pero no desde fuera de ella.

Nota:

Al declarar una variable con **Dim**, el ámbito que le estamos aplicando por regla general, es privado, pero como veremos, en Visual Basic 2010 el ámbito puede ser diferente a privado dependiendo del tipo en el que declaremos esa variable.

Miembros compartidos

Por otro lado, los miembros compartidos de una clase o tipo, son elementos que no pertenecen a una instancia o copia en memoria particular, sino que pertenecen al propio tipo y por tanto siempre están accesibles o disponibles, dentro del nivel del ámbito y accesibilidad que les hayamos aplicado, y su *tiempo de vida* es el mismo que el de la aplicación.

Del ámbito, la accesibilidad y los miembros compartidos nos ocuparemos con más detalle en una [lección posterior](#), donde veremos ciertas peculiaridades, como puede ser la limitación del ámbito de un miembro que *aparentemente* tiene una accesibilidad no restringida.

Parámetros específicos y parámetros opcionales

En Visual Basic 2010, tanto los miembros de una clase, (funciones y métodos Sub), como las propiedades pueden recibir parámetros. Esos parámetros pueden estar explícitamente declarados, de forma que podemos indicar cuantos argumentos tendremos que pasar al método, también podemos declarar parámetrosopcionales, que son parámetros que no hace falta indicar al llamar a la función o propiedad, y que siempre tendrán un valor predeterminado, el cual se usará en caso de que no lo indiquemos.

Además de los parámetros específicos y opcionales, podemos usar un array de parámetrosopcionales, en los que se puede indicar un número variable de argumentos al llamar a la función que los define, esto lo veremos en la siguiente sección.

Nota: Sobre parámetros y argumentos

Para clarificar las cosas, queremos decir que los parámetros son los definidos en la función, mientras que los argumentos son los "parámetros" que pasamos a esa función cuando la utilizamos desde nuestro código.

Con los parámetrosopcionales lo que conseguimos es permitir que el usuario no tenga que introducir todos los parámetros, sino solo los que realmente necesite, del resto, (los no especificados), se usará el valor predeterminado que hayamos indicado, porque una de las "restricciones" de este tipo de parámetros, es que siempre debemos indicar también el valor por defecto que debemos usar en caso de que no se especifique.

Veamos unos ejemplo para aclarar nuestras ideas:

```
Function Suma(n1 As Integer, Optional n2 As Integer = 15) As Integer
    Suma = n1 + n2
```

```
End Function
```

En este primer ejemplo, el primer parámetro es obligatorio (siempre debemos indicarlo) y el segundo es opcional, si no se indica al llamar a esta función, se usará el valor 15 que es el predeterminado.

Para llamar a esta función, lo podemos hacer de estas tres formas:

```
' 1- indicando los dos parámetros (el resultado será 4= 1 + 3)
t = Suma(1, 3)
' 2- Indicando solamente el primer parámetro (el resultado será 16= 1 + 15)
t = Suma(1)
' 3- Indicando los dos parámetros, pero en el opcional usamos el nombre
t = Suma(1, n2:= 9)
```

El tercer ejemplo solamente tiene utilidad si hay más de un parámetro opcional.

Nota:

Los parámetros opcionales deben aparecer en la lista de parámetros del método, después de los "obligatorios"

Nota:

En el caso hipotético y no recomendable de que no estemos usando Option Strict On, tanto los parámetros normales como los opcionales los podemos indicar sin el tipo de datos, pero si en alguno de ellos especificamos el tipo, debemos hacerlo en todos.

Array de parámetros opcionales (ParamArray)

En cuanto al uso de *ParamArray*, este tipo de parámetro opcional nos permite indicar un número indeterminado de parámetros, en Visual Basic 2010 **siempre** debe ser una array e internamente se trata como un array normal y corriente, es decir, dentro del método o propiedad se accede a él como si de un array se tratara... entre otras cosas, ¡porque es un array!

Si tenemos activado *Option Strict*, el array usado con *ParamArray* debe ser de un tipo en concreto, si queremos que acepte cualquier tipo de datos, lo podemos declarar con el tipo *Object*.

Si por cualquier razón necesitamos pasar otros valores que no sean de un tipo en concreto, por ejemplo, además de valores "normales" queremos pasar un array, podemos desactivar *Option Strict* para hacerlo fácil.

Por ejemplo, si queremos hacer algo como esto:

```
Function Suma(ParamArray n() As Object) As Integer
    Dim total As Integer
    '
    For i As Integer = 0 To n.Length - 1
        If IsArray(n(i)) Then
            For j As Integer = 0 To n(i).Length - 1
                total += n(i)(j)
            Next
        Else
            total += n(i)
        End If
    Next
    Return total
End Function

' Para usarlo:
Dim t As Integer
Dim a(2) As Integer = {1, 2, 3}
t = Suma(a, 4, 5, 6)

Console.WriteLine(t)
```

Tal como vemos, el primer argumento que pasamos a la función Suma es un array, después pasamos tres valores enteros normales. El resultado de ejecutar ese código será el valor **21**, como es de esperar.

Nota:

Como Visual Basic 2010 no sabe que tipo contiene *n(i)*, al escribir este código, no nos informará de que ese "objeto" tiene una propiedad llamada *Length*.

Pero como **no** debemos "acostumbrarnos" a desactivar *Option Strict*, vamos a ver el código que tendríamos que usar para que también acepte un array como parte de los argumentos indicados al llamar a la función.

```
Function Suma(ByVal ParamArray n() As Object) As Integer
    Dim total As Integer
    '
    For i As Integer = 0 To n.Length - 1
        If IsArray(n(i)) Then
            For j As Integer = 0 To CType(n(i), Integer()).Length - 1
                total += CType(n(i), Integer())(j)
            Next
        Else
            total += CInt(n(i))
        End If
    Next
    Return total
End Function

' Para usarlo:
```

```

Dim t As Integer
Dim a(2) As Integer = {1, 2, 3}
t = Suma(a, 4, 5, 6)
Console.WriteLine(t)

```

Como podemos comprobar, al tener *Option Strict* activado, debemos hacer una conversión explícita del array a uno de un tipo en concreto, como es natural, en este código estamos suponiendo que el array pasado como argumento a la función es de tipo *Integer*, en caso de que no lo fuera, recibiríamos un error en tiempo de ejecución.

Para comprobar si el contenido de n(i) es un array, hemos usado la función *IsArray* definida en el espacio de nombres *Microsoft.VisualBasic* (que por defecto utilizan todas las aplicaciones de Visual Basic 2010), pero también lo podríamos haber hecho más al "estilo .NET", ya que, como sabemos todos los arrays de .NET realmente se derivan de la clase *Array*:

```
If TypeOf n(i) Is Array Then
```

Nota:

Cuando queramos usar *ParamArray* para recibir un array de parámetros opcionales, esta instrucción debe ser la última de la lista de parámetros de la función (método).

Tampoco se permite tener parámetros opcionales y *ParamArray* en la misma función.

Sobrecarga de métodos y propiedades

La sobrecarga de funciones (realmente de métodos y propiedades), es una característica que nos permite tener una misma función con diferentes tipos de parámetros, ya sea en número o en tipo.

Supongamos que queremos tener dos funciones (o más) que nos permitan hacer operaciones con diferentes tipos de datos, y que, según el tipo de datos usado, el valor que devuelva sea de ese mismo tipo.

En este ejemplo, tenemos dos funciones que se llaman igual pero una recibe valores de tipo entero y la otra de tipo decimal:

```

Function Suma(n1 As Integer, n2 As Integer) As Integer
    Return n1 + n2
End Function

Function Suma(n1 As Double, n2 As Double) As Double
    Return n1 + n2
End Function

```

Como podemos comprobar las dos funciones tienen el mismo nombre, pero tanto una como otra reciben parámetros de tipos diferentes.

Con Visual Basic 2010 podemos sobrecargar funciones, pero lo interesante no es que podamos hacerlo, sino cómo podemos usar esas funciones. En el código anterior tenemos dos funciones llamadas **Suma**, la primera acepta dos parámetros de tipo *Integer* y la segunda de tipo *Double*. Lo interesante es que cuando queramos usarlas, no tenemos que preocuparnos de cual vamos a usar, ya que será el compilador el que decida la más adecuada al código que usemos, por ejemplo:

```

' En este caso, se usará la que recibe dos valores enteros
Console.WriteLine(Suma(10, 22))

' En este caso se usará la que recibe valores de tipo Double
Console.WriteLine(Suma(10.5, 22))

```

El compilador de Visual Basic 2010 es el que decide que función usar, esa decisión la toma a partir de los tipos de parámetros que hayamos indicado. En el segundo ejemplo de uso, el que mejor coincide es el de los dos parámetros de tipo *Double*.

También podemos tener sobrecarga usando una cantidad diferente de parámetros, aunque éstos sean del mismo tipo. Por ejemplo, podemos añadir esta declaración al código anterior sin que exista ningún tipo de error, ya que esta nueva función recibe tres parámetros en lugar de dos:

```

Function Suma(n1 As Integer, n2 As Integer, n3 As Integer) As Integer
    Return n1 + n2 + n3
End Function

```

Por tanto, cuando el compilador se encuentre con una llamada a la función **Suma** en la que se usen tres parámetros, intentará usar esta última.

Nota:

Para que exista sobrecarga, la diferencia debe estar en el número o en el tipo de los parámetros, no en el tipo del valor devuelto.

Cuando, desde el IDE de Visual Basic 2010, queremos usar los métodos sobrecargados, nos mostrará una lista de opciones indicándonos las posibilidades de sobrecarga que existen y entre las que podemos elegir, tal como vemos en la figura 2.10:

```

Dim t As Integer
t = suma()

```

1 of 3 Suma(n1 As Integer, n2 As Integer) As Integer

Figura 2.10. Lista de parámetros soportados por un método

Cuando utilizemos parámetros opcionales debemos tener en cuenta que puede que el compilador nos muestre un error, ya que es posible que esa función que declara parámetros opcionales entre en conflicto con una "sobrecargada".

Por ejemplo:

```

Function Suma(n1 As Integer, Optional n2 As Integer = 33) As Integer
    Return n1 + n2
End Function

```

Si tenemos esta declaración además de las anteriores, el programa no compilará, ya que si hacemos una llamada a la función **Suma** con dos parámetros enteros, el compilador no sabrá si usar esta última o la primera que declaramos, por tanto producirá un error.

Parámetros por valor y parámetros por referencia

Al igual que tenemos dos tipos de datos diferentes, en los parámetros de las funciones también podemos tenerlos, para ello tendremos que usar *ByVal* o *ByRef* para indicar al compilador cómo debe tratar a los parámetros.

Cuando un parámetro es por valor (*ByVal*), el *runtime* antes de llamar a la función hace una copia de ese parámetro y pasa la copia a la función, por tanto cualquier cambio que hagamos a ese parámetro dentro de la función no afectará al valor usado "externamente".

En el caso de que el parámetro sea por referencia (*ByRef*), el compilador pasa una referencia que apunta a la dirección de memoria en la que están los datos, por tanto si realizamos cambios dentro de la función, ese cambio sí que se verá reflejado en el parámetro usado al llamar a la función.

Nota:

Hay que tener en cuenta que si pasamos un objeto a una función, da igual que lo declaremos por valor o por referencia, ya que en ambos casos se pasa una referencia a la dirección de memoria en la que están los datos, porque, como sabemos, las variables de los tipos por referencia siempre contienen una referencia a los datos, no los datos en sí.

Cuando en Visual Basic 2010 usamos parámetros en los que no se indica si es por valor (*ByVal*) o por referencia (*ByRef*), serán tratados como parámetros por valor.

Nota:

Si usamos el IDE de Visual Studio 2010 para escribir el código, no debemos preocuparnos de este detalle, ya que si no indicamos si es por valor o por referencia, automáticamente le añadirá la palabra clave *ByVal*, para que no haya ningún tipo de dudas.

Parámetros opcionales de tipo Nullable

En el tradicional Visual Basic, podíamos crear métodos y funciones con parámetros opcionales. Esta característica fue agregada en Visual Basic, y demandada por la comunidad de desarrolladores de C#, ya que en determinadas situaciones, puede resultar realmente útil.

De esta manera, hacer lo siguiente en Visual Basic 2008 es completamente válido:

```

Private Function Sumar(Optional ByVal x As Integer = Nothing, Optional ByVal y As Integer = Nothing) As Integer
    x = IIf(IsNothing(x), 0, x)
    y = IIf(IsNothing(y), 0, y)
    Return x + y
End Function

```

Podríamos entonces llamar a esa función de la siguiente manera:

```
MessageBox.Show(Sumar(3).ToString())
```

Función que nos devolvería el valor 3, siendo la suma, 3 + 0, y siendo 3 el valor de x.

Ahora bien, en Visual Basic 2010 podemos indicar parámetros opcionales, pero de tipo Nullable.

Su funcionamiento es el mismo, con la salvedad de utilizar la variable como variable de tipo Nullable (en el apartado de tipos Nullable se explicaba como funcionaban).

Un ejemplo práctico de como funcionan y como utilizar este tipo de parámetros es el que se indica a continuación:

```

Private Function Sumar(Optional ByVal x As Integer? = Nothing, Optional ByVal y As Integer? = Nothing) As Integer
    x = IIf(IsNothing(x), 0, x)
    y = IIf(IsNothing(y), 0, y)

```

```
    Return x + y  
End Function
```

Para ejecutar el método, lanzaremos el mismo procedimiento de ejecución:

```
MessageBox.Show(Sumar(3).ToString())
```

El resultado que obtenemos en este caso, será el mismo que vimos anteriormente, es decir, el valor 3, pero intente pensar que este es un ejemplo muy básico, y que los tipos Nullable nos permiten indicar valores que no están inicializados y en base a ese valor, realizar una u otra acción.

Lección 2: Clases y estructuras

- Clases
- Definir una clase
- Instanciar una clase**
- Estructuras
- Accesibilidad
- Propiedades
- Interfaces

Instanciar: Crear un objeto en la memoria

Una vez que tenemos una clase definida, lo único de lo que disponemos es de una especie de plantilla o molde a partir del cual podemos crear objetos en memoria.

Para crear esos objetos en Visual Basic 2010 lo podemos hacer de dos formas, pero como veremos siempre será mediante la instrucción *New* que es la encargada de crear el objeto en la memoria y asignar la dirección del mismo a la variable usada en la parte izquierda de la asignación.

Declarar primero la variable y después instanciarla

Lo primero que tenemos que hacer es declarar una variable del tipo que queremos instanciar, esto lo hacemos de la misma forma que con cualquier otro tipo de datos:

```
Dim c As Cliente
```

Con esta línea de código lo que estamos indicando a Visual Basic es que tenemos intención de usar una variable llamada **c** para acceder a una clase de tipo **Cliente**. Esa variable, cuando llegue el momento de usarla, sabrá todo lo que hay que saber sobre una clase **Cliente**, pero hasta que no tenga una "referencia" a un objeto de ese tipo no podremos usarla.

La asignación de una referencia a un objeto **Cliente** la haremos usando la instrucción *New* seguida del nombre de la clase:

```
c = New Cliente
```

A partir de este momento, la variable **c** tiene acceso a un nuevo objeto del tipo **Cliente**, por tanto podremos usarla para asignarle valores y usar cualquiera de los miembros que ese tipo de datos contenga:

```
c.Nombre = "Antonio"  
c.Apellidos = "Ruiz Rodríguez"
```

Declarar y asignar en un solo paso

Con las clases o tipos por referencia también podemos declarar una variable y al mismo tiempo asignarle un nuevo objeto:

```
Dim c As New Cliente
```

O también:

```
Dim c As Cliente = New Cliente
```

Las dos formas producen el mismo resultado, por tanto es recomendable usar la primera.

El constructor: El punto de inicio de una clase

Cada vez que creamos un nuevo objeto en memoria estamos llamando al constructor de la clase. En Visual Basic 2010 el constructor es un método de tipo *Sub* llamado *New*.

En el constructor de una clase podemos incluir el código que creamos conveniente, pero realmente solamente deberíamos incluir el que realice algún tipo de inicialización, en caso de que no necesitemos realizar ningún tipo de inicialización, no es necesario definir el constructor, ya que el propio compilador lo hará por nosotros. Esto es así porque todas las clases deben implementar un constructor, por tanto si nosotros no lo definimos, lo hará el compilador de Visual Basic 2010.

Si nuestra clase **Cliente** tiene un campo para almacenar la fecha de creación del objeto podemos hacer algo como esto:

```
Class Cliente
    Public Nombre As String
    Public Apellidos As String
    Public FechaCreacion As Date
    '
    Public Sub New()
        FechaCreacion = Date.Now
    End Sub
End Class
```

De esta forma podemos crear un nuevo **Cliente** y acto seguido comprobar el valor del campo **FechaCreacion** para saber la fecha de creación del objeto.

En los constructores también podemos hacer las inicializaciones que, por ejemplo permitan a la clase a conectarse con una base de datos, abrir un fichero o cargar una imagen gráfica, etc.

Constructores parametrizados

De la misma forma que podemos tener métodos y propiedades sobrecargadas, también podemos tener constructores sobrecargados, ya que debemos recordar que en Visual Basic 2010, un constructor realmente es un método de tipo *Sub*, y como todos los métodos y propiedades de Visual Basic 2010, también admite la sobrecarga.

La ventaja de tener constructores que admitan parámetros es que podemos crear nuevos objetos indicando algún parámetro, por ejemplo un fichero a abrir o, en el caso de la clase **Cliente**, podemos indicar el nombre y apellidos del cliente o cualquier otro dato que creamos conveniente.

Para comprobarlo, podemos ampliar la clase definida anteriormente para que también acepte la creación de nuevos objetos indicando el nombre y los apellidos del cliente.

```
Class Cliente
    Public Nombre As String
```

```

Public Apellidos As String
Public FechaCreacion As Date
'
Public Sub New()
    FechaCreacion = Date.Now
End Sub
'
Public Sub New(elNombre As String, losApellidos As String)
    Nombre = elNombre
    Apellidos = losApellidos
    FechaCreacion = Date.Now
End Sub
End Class

```

Teniendo esta declaración de la clase **Cliente**, podemos crear nuevos clientes de dos formas:

```

Dim c1 As New Cliente
Dim c2 As New Cliente("Jose", "Sánchez")

```

Como podemos comprobar, en ciertos casos es más intuitiva la segunda forma de crear objetos del tipo **Cliente**, además de que así nos ahorraremos de tener que asignar individualmente los campos **Nombre** y **Apellidos**.

Esta declaración de la clase **Cliente** la podríamos haber hecho de una forma diferente. Por un lado tenemos un constructor "normal" (no recibe parámetros) en el que asignamos la fecha de creación y por otro el constructor que recibe los datos del nombre y apellidos. En ese segundo constructor también asignamos la fecha de creación, ya que, se instancie como se instancie la clase, nos interesa saber siempre la fecha de creación. En este ejemplo, por su simpleza no es realmente un problema repetir la asignación de la fecha, pero si en lugar de una inicialización necesitáramos hacer varias, la verdad es que nos encontraríamos con mucha "duplicidad" de código. Por tanto, en lugar de asignar los datos en dos lugares diferentes, podemos hacer esto otro:

```

Class Cliente
    Public Nombre As String
    Public Apellidos As String
    Public FechaCreacion As Date
'
    Public Sub New()
        FechaCreacion = Date.Now
    End Sub
'
    Public Sub New(elNombre As String, losApellidos As String)
        Nombre = elNombre
        Apellidos = losApellidos
    End Sub
End Class

```

Es decir, desde el constructor con argumentos llamamos al constructor que no los tiene, consiguiendo que también se asigne la fecha.

Esta declaración de la clase **Cliente** realmente no compilará. Y no compila por la razón tan "simple" de que aquí el compilador de Visual Basic 2010 no sabe cual es nuestra intención, ya que *New* es una palabra reservada que sirve para crear nuevos objetos y, siempre, una instrucción tiene preferencia sobre el nombre de un método, por tanto, podemos recurrir al objeto especial *Me* el cual nos sirve, para representar al objeto que actualmente está en la memoria, así que, para que esa declaración de la clase Cliente funcione, debemos usar *Me.New()* para llamar al constructor sin parámetros.

Nota:

El IDE de Visual Basic 2010 colorea las instrucciones y tipos propios del

lenguaje, en el caso de Me, lo que no debe confundirnos es que cuando declaramos Sub New, tanto Sub como New se muestran coloreadas, cuando solo se debería colorear Sub; sin embargo cuando usamos Me.New, solo se colorea Me y no New que es correcto, tal como vemos en la figura 2.11, ya que en este caso New es el nombre de un procedimiento y los procedimientos no son parte de las instrucciones y tipos de .NET.

```
Public Sub New()
    FechaCreacion = Date.Now
End Sub

Public Sub New(ByVal elNombre As String)
    Me.New()
    Nombre = elNombre
    Apellidos = losApellidos
End Sub

End Class
```

Figura 2.11. Coloreo erróneo de New

Cuando Visual Basic 2010 no crea un constructor automáticamente

Tal como hemos comentado, si nosotros no definimos un constructor (*Sub New*), lo hará el propio compilador de Visual Basic 2010, y cuando lo hace automáticamente, siempre es un constructor sin parámetros.

Pero hay ocasiones en las que nos puede interesar que no exista un constructor sin parámetros, por ejemplo, podemos crear una clase **Cliente** que solo se pueda instanciar si le pasamos, por ejemplo el número de identificación fiscal, (NIF), en caso de que no se indique ese dato, no podremos crear un nuevo objeto **Cliente**, de esta forma, nos aseguramos siempre de que el NIF siempre esté especificado. Seguramente por ese motivo, si nosotros definimos un constructor con parámetros, Visual Basic 2010 no crea uno automáticamente sin parámetros. Por tanto, si definimos un constructor con parámetros en una clase y queremos que también tenga uno sin parámetros, lo tenemos que definir nosotros mismos.

El destructor: El punto final de la vida de una clase

De la misma forma que una clase tiene su punto de entrada o momento de nacimiento en el constructor, también tienen un sitio que se ejecutará cuando el objeto creado en la memoria ya no sea necesario, es decir, cuando acabe la vida del objeto creado.

El destructor de Visual Basic 2010 es un método llamado *Finalize*, el cual se hereda de la clase *Object*, por tanto no es necesario que escribamos nada en él. El propio CLR se encargará de llamar a ese método cuando el objeto ya no sea necesario.

La forma en que se destruyen los objetos pueden ser de dos formas:
Porque los destruyamos nosotros asignando un valor *Nothing* a la variable que lo referencia, o bien porque el objeto esté fuera de ámbito, es decir, haya salido de su espacio de ejecución, por ejemplo, si declaramos un objeto dentro de un método, cuando ese método termina, el objeto se destruye, (hay algunas excepciones a esta última regla, como puede ser que ese mismo objeto también esté referenciado por otra variable externa al método.)

Lo que debemos tener muy presente es que en .NET los objetos no se destruyen inmediatamente. Esto es así debido a que en .NET existe un "sistema" que se encarga de realizar esta gestión de limpieza: El recolector de basura o de objetos no usados (*Garbage Collector*, GC). Este recolector de objetos no usados se encarga de comprobar constantemente cuando un objeto no se está usando y es el que decide cuando hay que

llamar al destructor.

Debido a esta característica de .NET, si nuestra clase hace uso de recursos externos que necesiten ser eliminados cuando el objeto ya no se vaya a seguir usando, debemos definir un método que sea el encargado de realizar esa liberación, pero ese método debemos llamarlo de forma manual, ya que, aunque en .NET existen formas de hacer que esa llamada sea automática, nunca tenderemos la seguridad de que se llame en el momento oportuno, y esto es algo que, según que casos, puede ser un inconveniente.

Recomendación:

Si nuestra clase utiliza recursos externos, por ejemplo un fichero o una base de datos, debemos definir un método que se encargue de liberarlos y a ese método debemos encargarnos de llamarlo cuando ya no lo necesitemos.

Por definición a este tipo de métodos se les suele dar el nombre Close o Dispose, aunque este último tiene un significado especial y por convención solo debemos usarlo siguiendo las indicaciones de la documentación.

Inicialización directa de objetos

De todos los modos, en Visual Basic 2010 se ha agregado una característica adicional que tiene que ver con la inicialización de objetos.

Hasta ahora hemos visto como instanciar una clase utilizando su constructor, pero quizás lo que no sabíamos es que en Visual Basic 2010 podemos inicializar la clase a través de su constructor e incluso inicializar directamente las propiedades de la clase.

Supongamos por lo tanto el siguiente código fuente:

```
Class Cliente
    Private m_Nombre As String
    Public Property Nombre() As String
        Get
            Return m_Nombre
        End Get
        Set(ByVal value As String)
            m_Nombre = value
        End Set
    End Property

    Public Sub New()
    End Sub
End Class
```

A continuación instanciaremos la clase y con ello, inicializaremos adicionalmente la propiedad que hemos declarado.

El código quedará de la siguiente forma:

```
Dim claseCliente As New Cliente() With {.Nombre = "Pedro"}
```

Esta forma de inicializar objetos nos permite ser más productivos y prácticos y ahorrar tiempo y líneas de código en nuestras aplicaciones.

Como podemos apreciar, la forma de inicializar las propiedades o las variables de una clase es utilizando la palabra clave *With* seguida de llaves, y dentro de las llaves, un punto seguido de la propiedad o variable que queremos inicializar con sus valor de inicialización. Si tenemos más de una variable a inicializar, deberemos separarlas por comas.

Intellisense nos ayudará enormemente a inicializar la propiedad o variable que deseemos, ya que vincula de forma directa el código de la clase haciéndonos más fácil el trabajo con los

miembros de la clase.

Lección 2: Clases y estructuras

- Clases
- Definir una clase
- Instanciar una clase

Estructuras

- Accesibilidad
- Propiedades
- Interfaces

Estructuras: Tipos por valor definidos por el usuario

De la misma forma que podemos definir nuestros propios tipos de datos por referencia, Visual Basic 2010 nos permite crear nuestros propios tipos por valor.

Para crear nuestros tipos de datos por referencia, usamos la "instrucción" *Class*, por tanto es de esperar que también exista una instrucción para crear nuestros tipos por valor, y esa instrucción es: *Structure*, por eso en Visual Basic 2010 a los tipos por valor definidos por el usuario se llaman estructuras.

Las estructuras pueden contener los mismos miembros que las clases, aunque algunos de ellos se comporten de forma diferente o al menos tengan algunas restricciones, como que los campos definidos en las estructuras no se pueden inicializar al mismo tiempo que se declaran o no pueden contener constructores "simples", ya que el propio compilador siempre se encarga de crearlo, para así poder inicializar todos los campos definidos.

Otra de las características de las estructuras es que no es necesario crear una instancia para poder usarlas, ya que es un tipo por valor y los tipos por valor no necesitan ser instanciados para que existan.

Definir una estructura

Las estructuras se definen usando la palabra *Structure* seguida del nombre y acaba usando las instrucciones *End Structure*.

El siguiente código define una estructura llamada Punto en la que tenemos dos campos públicos.

```
Structure Punto
    Public X As Integer
    Public Y As Integer
End Structure
```

Para usarla podemos hacer algo como esto:

```
Dim p As Punto
p.X = 100
p.Y = 75
```

También podemos usar *New* al declarar el objeto:

```
Dim p As New Punto
```

Aunque en las estructuras, usar *New*, sería algo redundante y por tanto no necesario.

Las estructuras siempre se almacenan en la pila, por tanto deberíamos tener la precaución de no crear estructuras con muchos campos o con muchos miembros, ya que esto implicaría un mayor consumo del "preciado" espacio de la pila.

Constructores de las estructuras

Tal y como hemos comentado, las estructuras siempre definen un constructor sin parámetros, este constructor no lo podemos definir nosotros, es decir, siempre existe y es el que el propio compilador de Visual Basic 2010 define.

Por tanto, si queremos agregar algún constructor a una estructura, este debe tener parámetros y, tal como ocurre con cualquier método o como ocurre en las clases, podemos tener varias sobrecargas de constructores parametrizados en las estructuras. La forma de definir esos constructores es como vimos en las clases: usando distintas sobrecargas de un método llamado *New*, en el caso de las estructuras también podemos usar la palabra clave *Me* para referirnos a la instancia actual.

Esto es particularmente práctico cuando los parámetros del constructor se llaman de la misma forma que los campos declarados en la estructura, tal como ocurre en el constructor mostrado en la siguiente definición de la estructura Punto.

```
Structure Punto
    Public X As Integer
    Public Y As Integer
    '
    Sub New(ByVal x As Integer, ByVal y As Integer)
        Me.X = x
        Me.Y = y
    End Sub
End Structure
```

Nota:

Tanto en las estructuras como en las clases podemos tener constructores compartidos, (*Shared*), en el caso de las estructuras, este tipo de constructor es el único que podemos declarar sin parámetros.

Destructores de las estructuras

Debido a que las estructuras son tipos por valor y por tanto una variable declarada con un tipo por valor "contiene el valor en si misma", no podemos destruir este tipo de datos, lo más que conseguiríamos al asignarle un valor nulo (*Nothing*) sería eliminar el contenido de la variable, pero nunca podemos destruir ese valor. Por tanto, en las estructuras no podemos definir destructores.

Los miembros de una estructura

Como hemos comentado, los miembros o elementos que podemos definir en una estructura son los mismos que ya vimos en [las clases](#). Por tanto aquí veremos las diferencias que existen al usarlos en las estructuras.

Campos

Como vimos, las variables declaradas a nivel del tipo, son los campos, la principal diferencia con respecto a las clases, es que los campos de una estructura no pueden inicializarse en la

declaración y el valor que tendrán inicialmente es un valor "nulo", que en el caso de los campos de tipo numéricos es un cero.

Por tanto, si necesitamos que los campos tengan algún valor inicial antes de usarlos, deberíamos indicarlo a los usuarios de nuestra estructura y proveer un constructor que realice las inicializaciones correspondientes, pero debemos recordar que ese constructor debe tener algún parámetro, ya que el predeterminado sin parámetros no podemos "reescribirlo".

Los únicos campos que podemos inicializar al declararlos son los campos compartidos, pero como tendremos oportunidad de ver, estos campos serán accesibles por cualquier variable declarada y cualquier cambio que realicemos en ellos se verá reflejado en el resto de "instancias" de nuestro tipo.

Métodos y otros elementos

El resto de los miembros de una estructura se declaran y usan de la misma forma que en las clases, si bien debemos tener en cuenta que el modificador de accesibilidad predeterminado para los miembros de una estructura es *Public*, (incluso si son campos declarados con *Dim*).

Otro detalle a tener en cuenta es que en una estructura siempre debe existir al menos un evento o un campo no compartido, no se permiten estructuras en las que solo tienen constantes, métodos y/o propiedades, estén o no compartidos.

Cómo usar las estructuras

Tal como hemos comentado las estructuras son tipos por valor, para usar los tipos por valor no es necesario instanciarlos explícitamente, ya que el mero hecho de declararlos indica que estamos creando un nuevo objeto en memoria. Por tanto, a diferencia de las clases o tipos por referencia, cada variable definida como un tipo de estructura será independiente de otras variables declaradas, aunque no las hayamos instanciado.

Esta característica de las estructuras nos permite hacer copias "reales" no copia de la referencia (o puntero) al objeto en memoria como ocurre con los tipos por referencia. Veámoslos con un ejemplo.

```
Dim p As New Punto(100, 75)
Dim p1 As Punto
p1 = p
p1.X = 200
' p.X vale 100 y p1.X vale 200
```

En este trozo de código definimos e *instanciamos* una variable del tipo **Punto**, a continuación declaramos otra variable del mismo tipo y le asignamos la primera, si estos tipos fuesen por referencia, tanto una como la otra estarían haciendo referencia al mismo objeto en memoria, y cualquier cambio realizado a cualquiera de las dos variables afectaría al mismo objeto, pero en el caso de las estructuras (y de los tipos por valor), cada cambio que realicemos se hará sobre un *objeto* diferente, por tanto la asignación del valor **200** al campo **X** de la variable **p1** solo afecta a esa variable, dejando intacto el valor original de la variable **p**.

Lección 2: Clases y estructuras

- Clases
 - Definir una clase
 - Instanciar una clase
 - Estructuras
- Accesibilidad**
- Propiedades
 - Interfaces

Accesibilidad y ámbito

Tal y como comentamos anteriormente, dependiendo de dónde y cómo estén declarados los tipos de datos y los miembros definidos en ellos, tendremos o no acceso a esos elementos.

Recordemos que el ámbito es el alcance con el que podemos acceder a un elemento y depende de dónde esté declarado, por otro lado, la accesibilidad depende de cómo declaremos cada uno de esos elementos.

Ámbito

Dependiendo de donde declaremos un miembro o un tipo, éste tendrá mayor alcance o cobertura, o lo que es lo mismo, dependiendo del ámbito en el que usemos un elemento, podremos acceder a él desde otros puntos de nuestro código.

A continuación veremos con detalle los ámbitos en los que podemos declarar los distintos elementos de Visual Basic 2010.

- **Ámbito de bloque:** Disponible únicamente en el bloque de código en el que se ha declarado.
Por ejemplo, si declaramos una variable dentro de un bucle *For* o un *If Then*, esa variable solo estará accesible dentro de ese bloque de código.
- **Ámbito de procedimiento:** Disponible únicamente dentro del procedimiento en el que se ha declarado. Cualquier variable declarada dentro de un procedimiento (método o propiedad) solo estará accesible en ese procedimiento y en cualquiera de los bloques internos a ese procedimiento.
- **Ámbito de módulo:** Disponible en todo el código del módulo, la clase o la estructura donde se ha declarado. Las variables con ámbito a nivel de módulo, también estarán disponibles en los procedimientos declarados en el módulo (clase o estructura) y por extensión a cualquier bloque dentro de cada procedimiento.
- **Ámbito de espacio de nombres:** Disponible en todo el código del espacio de nombres. Este es el nivel mayor de cobertura o alcance, aunque en este nivel solo podemos declarar tipos como clases, estructuras y enumeraciones, ya que los procedimientos solamente se pueden declarar dentro de un tipo.

Nota:

Por regla general, cuando declaramos una variable en un ámbito, dicha variable "ocultará" a otra que tenga el mismo nombre y esté definida en un bloque con mayor alcance, aunque veremos que en Visual Basic 2010 existen ciertas restricciones dependiendo de dónde declaremos esas variables.

En Visual Basic 2010 podemos definir una variable dentro de un bloque de código, en ese caso dicha variable solo será accesible dentro de ese bloque. Aunque, como veremos a continuación, en un procedimiento solamente podremos definir variables que no se oculten entre sí, estén o no dentro de un bloque de código.

Ámbito de bloque

En los siguientes ejemplos veremos cómo podemos definir variables para usar solamente en el bloque en el que están definidas.

Los bloques de código en los que podemos declarar variables son los bucles, (*For*, *Do*, *While*), y los bloques condicionales, (*If*, *Select*).

Por ejemplo, dentro de un procedimiento podemos tener varios de estos bloques y por tanto podemos definir variables "internas" a esos bloques:

```
Dim n As Integer = 3
'
For i As Integer = 1 To 10
    Dim j As Integer
    j += 1
    If j < n Then
        ...
    End If
Next
'
If n < 5 Then
    Dim j As Integer = n * 3
End If
'
Do
    Dim j As Integer
    For i As Integer = 1 To n
        j += i
    Next
    If j > 10 Then Exit Do
Loop
```

La variable **n** estará disponible en todo el procedimiento, por tanto podemos acceder a ella desde cualquiera de los bloques.

En el primer bucle *For*, definimos la variable **i** como la variable a usar de contador, esta variable solamente estará accesible dentro de este bucle *For*. Lo mismo ocurre con la variable **j**.

En el primer *If* definimos otra variable **j**, pero esa solo será accesible dentro de este bloque *If* y por tanto no tiene ninguna relación con la definida en el bucle *For* anterior.

En el bucle *Do* volvemos a definir nuevamente una variable **j**, a esa variable la podemos acceder solo desde el propio bucle *Do* y cualquier otro bloque de código interno, como es el caso del bucle *For*, en el que nuevamente declaramos una variable llamada **i**, que nada tiene que ver con el resto de variables declaradas con el mismo nombre en los otros bloques.

Lo único que no podemos hacer en cualquiera de esos bloques, es declarar una variable llamada **n**, ya que al estar declarada en el procedimiento, el compilador de Visual Basic 2010 nos indicará que no podemos ocultar una variable previamente definida fuera del bloque, tal como podemos ver en la figura 2.12.

```

    Dim n As Integer = 3
    '
    For i As Integer = 0 To 9
        Dim n As Integer
        La variable 'n' oculta una variable en un bloque de inclusión.
    Next

```

Figura 2.12. Error al ocultar una variable definida en un procedimiento

Esta restricción solo es aplicable a las variables declaradas en el procedimiento, ya que si declaramos una variable a nivel de módulo, no habrá ningún problema para usarla dentro de un bloque, esto es así porque en un procedimiento podemos declarar variables que se llamen de la misma forma que las declaradas a nivel de módulo, aunque éstas ocultarán a las del "nivel" superior.

Ámbito de procedimiento

Las variables declaradas en un procedimiento tendrán un ámbito o cobertura que será el procedimiento en el que están declaradas, y como hemos visto, ese ámbito incluye también cualquier bloque de código declarado dentro del procedimiento.

Estas variables ocultarán a las que se hayan declarado fuera del procedimiento, si bien, dependiendo del tipo de módulo, podremos acceder a esas variables "externas" indicando el nombre completo del módulo o bien usando la instrucción *Me*, tal como vimos en el código del constructor parametrizado de [la estructura Punto](#).

Pero mejor veámoslo con un ejemplo. En el siguiente código, definimos una clase en la que tenemos un campo llamado **Nombre**, también definimos un método en el que internamente se utiliza una variable llamada *nombre*, para acceder a la variable declarada en la clase, tendremos que usar la instrucción o palabra clave *Me*.

```

Public Class Cliente
    Public Nombre As String = "Juan"
    Function Mostrar() As String
        Dim nombre As String = "Pepita"
        Return "Externo= " & Me.Nombre & ", interno= " & nombre
    End Function
End Class

```

En este ejemplo, el hecho de que una variable esté declarada con la letra **ENE** en mayúscula o en minúscula no implica ninguna diferencia, ya que Visual Basic 2010 no hace distinciones de este tipo; aún así, Visual Basic 2010 respetará el nombre según lo hemos escrito y no cambiará automáticamente el "case" de las variables, salvo cuando están en el mismo nivel de ámbito, es decir, si la variable **nombre** que hemos definido en la función **Mostrar** la volvemos a usar dentro de esa función, Visual Basic 2010 la seguirá escribiendo en minúsculas, pero si escribimos "nombre" fuera de esa función, VB se dará cuenta de que hay una variable declarada con la ENE en mayúsculas y automáticamente la cambiará a **Nombre**, aunque nosotros la escribamos de otra forma.

Ámbito de módulo

Cuando hablamos de módulos, nos estamos refiriendo a un "tipo" de datos, ya sea una clase, una estructura o cualquier otro tipo de datos que nos permita definir .NET.

En estos casos, las variables declaradas dentro de un tipo de datos serán visibles desde cualquier parte de ese tipo, siempre teniendo en cuenta las restricciones mencionadas en los casos anteriores.

Ámbito de espacio de nombres

Los espacios de nombres son los contenedores de tipos de datos de mayor nivel, y sirven para contener definiciones de clases, estructuras, enumeraciones y delegados. Cualquier tipo definido a nivel de espacio de nombres estará disponible para cualquier otro elemento definido en el mismo espacio de nombres.

Al igual que ocurre en el resto de ámbitos "inferiores", si definimos un tipo en un espacio de nombres, podemos usar ese mismo nombre para nombrar a un procedimiento o a una variable, en cada caso se aplicará el ámbito correspondiente y, tal como vimos anteriormente, tendremos que usar nombres únicos para poder acceder a los nombres definidos en niveles diferentes.

La palabra clave Global

En Visual Basic 2010 podemos definir espacios de nombres cuyos nombres sean los mismos que los definidos en el propio .NET Framework, para evitar conflictos de ámbitos, podemos usar la palabra clave *Global* para acceder a los que se han definido de forma "global" en .NET.

Por ejemplo, si tenemos el siguiente código en el que definimos una clase dentro de un espacio de nombres llamado *System* y queremos acceder a uno de los tipos definidos en el espacio de nombres *System* de .NET, tendremos un problema:

```
Namespace System
    Class Cliente
        Public Nombre As String
        Public Edad As System.Int32
    End Class
End Namespace
```

El problema es que el compilador de Visual Basic 2010 nos indicará que el tipo *Int32* no está definido, ya que intentará buscarlo dentro del ámbito que actualmente tiene, es decir, la declaración que nosotros hemos hecho de *System*, por tanto para poder acceder al tipo *Int32* definido en el espacio de nombres "global" *System* de .NET tendremos que usar la instrucción *Global*, por suerte el IDE de Visual Studio 2010 reconoce este tipo de error y nos ofrece ayuda para poder solventar el conflicto, tal como vemos en la figura 2.13:

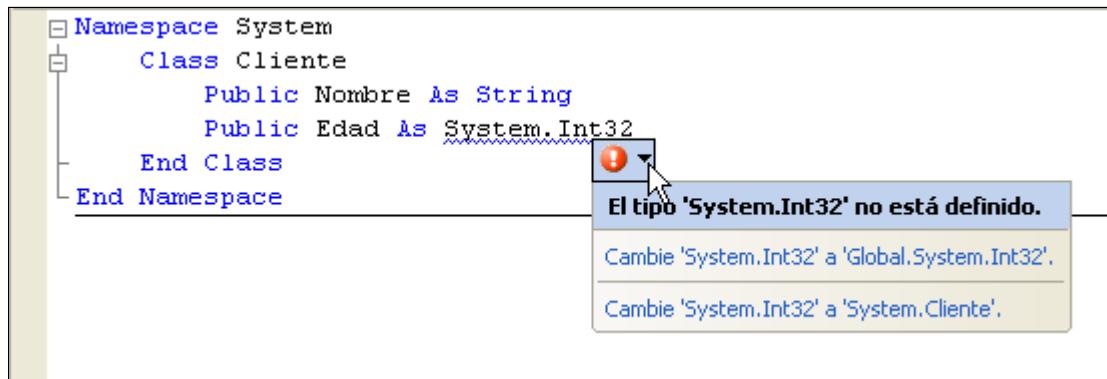


Figura 2.13. Ayuda del IDE en los conflictos de espacios nombres globales

Nota:

Afortunadamente este conflicto con los espacios de nombres no será muy habitual para los desarrolladores que usemos el idioma de Cervantes, por la sencilla razón de que los espacios de nombres de .NET Framework suelen estar definidos usando palabras en inglés.

Accesibilidad

La accesibilidad es la característica que podemos aplicar a cualquiera de los elementos que definamos en nuestro código. Dependiendo de la accesibilidad declarada tendremos distintos tipos de accesos a esos elementos.

Los modificadores de accesibilidad que podemos aplicar a los tipos y elementos definidos en nuestro código pueden ser cualquiera de los mostrados en la siguiente lista:

- **Public:** Acceso no restringido. Este es modificador de accesibilidad con mayor "cobertura", podemos acceder a cualquier miembro público desde cualquier parte de nuestro código. Aunque, como veremos, este acceso no restringido puede verse reducido dependiendo de dónde lo usemos.
- **Protected:** Acceso limitado a la clase contenedora o a los tipos derivados de esta clase. Este modificador solamente se usa con clases que se deriven de otras.
- **Friend:** Acceso limitado al proyecto actual. Visual Basic 2010 aplica este modificador de forma predeterminada a los procedimientos declarados en las clases.
- **Protected Friend:** Acceso limitado al proyecto actual o a los tipos derivados de la clase contenedora. Una mezcla de los dos modificadores anteriores.
- **Private:** Acceso limitado al tipo contenedor. Es el más restrictivos de todos los modificadores de accesibilidad y en el caso de los campos declarados en las clases (*Class*) equivale a usar *Dim*.

Estos modificadores de accesibilidad los podemos usar tanto en clases, estructuras, interfaces, enumeraciones, delegados, eventos, métodos, propiedades y campos. Aunque no serán aplicables en espacios de nombres (*Namespace*) ni clases de tipo *Module*, en estos dos casos siempre tendrán cobertura pública, si bien no se permite el uso de ningún modificador.

Accesibilidad de las variables en los procedimientos

Las variables declaradas dentro de un procedimiento solo son accesibles dentro de ese procedimiento, en este caso solo se puede aplicar el ámbito privado, aunque no podremos usar la instrucción *Private*, sino *Dim* o *Static*.

Nota:

La palabra clave *Static*, nos permite definir una variable privada (o local) al procedimiento para que mantenga el valor entre diferentes llamadas a ese procedimiento; esto contrasta con el resto de variables declaradas en un procedimiento cuya duración es la misma que la vida del propio procedimiento, por tanto, las variables no estáticas pierden el valor al terminar la ejecución del procedimiento.

Las accesibilidades predeterminadas

La accesibilidad de una variable o procedimiento en la que no hemos indicado el modificador de accesibilidad dependerá del sitio en el que la hemos declarado.

Por ejemplo, en las estructuras si definimos los campos usando *Dim*, estos tendrán un ámbito igual que si le hubiésemos aplicado el modificador *Public*; sin embargo, esa misma variable declarada en una clase (*Class* o *Module*) tendrá una accesibilidad *Private*. Así mismo, si el elemento que declaramos es un procedimiento y no indicamos el modificador de ámbito, éste tendrá un ámbito de tipo *Public* si lo definimos en una estructura y si el lugar en el que lo declaramos es una clase (o *Module*), éste será *Friend*.

En la siguiente tabla tenemos la accesibilidad predeterminada de cada tipo (clase, estructura, etc.), así como de las variables declaradas con *Dim* y de los procedimientos en los que no se indican el modificador de accesibilidad.

Tipo	del tipo	de las variables	de los
------	----------	------------------	--------

		declaradas con Dim	procedimientos
Class	Friend	Private	Friend
Module	Friend	Private	Friend
Structure	Friend	Public	Public
Enum	Public Friend	N.A. (los miembros siempre son públicos)	N.A.
Interface	Friend	N.A. (no se pueden declarar variables)	Public (no se permite indicarlo)

Tabla 2.3. La accesibilidad predeterminada de los tipos

Tal como podemos ver en la tabla 2.3, la accesibilidad predeterminada, (la que tienen cuando no se indica expresamente con un modificador), de todos los tipos es *Friend*, es decir, accesible a todo el proyecto, aunque en el caso de las enumeraciones el modificador depende de dónde se declare dicha enumeración, si está declarada a nivel de espacio de nombres será *Friend*, en el resto de los casos será *Public*.

En la tercera columna tenemos la accesibilidad predeterminada cuando declaramos las variables con *Dim*, aunque en las interfaces y en las enumeraciones no se permiten declarar variables.

La última columna es la correspondiente a los procedimientos, en el caso de las interfaces no se puede aplicar ningún modificador de accesibilidad y de forma predeterminada son públicos.

En esta otra tabla tenemos la accesibilidad permitida en cada tipo así como las que podemos indicar en los miembros de esos tipos.

Tipo	del tipo	de los miembros
Class	Public Friend Private Protected Protected Friend	Public Friend Private Protected Protected Friend
Module	Public Friend	Public Friend Private
Structure	Public Friend Private	Public Friend Private
Enum	Public Friend Private	N.A.
Interface	Public Friend Private Protected Protected Friend	N.A. Siempre son públicos

Tabla 2.4. Accesibilidades permitidas en los tipos

Algunos de los modificadores que podemos indicar en los tipos dependen de dónde declaremos esos tipos, por ejemplo, tan solo podremos indicar el modificador privado de las enumeraciones cuando estas se declaren dentro de un tipo. En el caso de las clases e

interfaces, los modificadores *Protected* y *Protected Friend* solo podremos aplicarlos cuando están declaradas dentro de una clase (*Class*).

Anidación de tipos

Tal como hemos comentado en el párrafo anterior, podemos declarar tipos dentro de otros tipos, por tanto el ámbito y accesibilidad de esos tipos dependen del ámbito y accesibilidad del tipo que los contiene. Por ejemplo, si declaramos una clase con acceso *Friend*, cualquier tipo que esta clase contenga siempre estará supeditado al ámbito de esa clase, por tanto si declaramos otro tipo interno, aunque lo declaremos como *Public*, nunca estará más accesible que la clase contenedora, aunque en estos casos no habrá ningún tipo de confusión, ya que para acceder a los tipos declarados dentro de otros tipos siempre tendremos que indicar la clase que los contiene.

En el siguiente código podemos ver cómo declarar dos clases "anidadas". Tal como podemos comprobar, para acceder a la clase *Salario* debemos indicar la clase *Cliente*, ya que la única forma de acceder a una clase anidada es mediante la clase contenedora.

```
Friend Class Cliente
    Public Nombre As String

    Public Class Salario
        Public Importe As Decimal
    End Class
End Class

' Para usar la clase Salario debemos declararla de esta forma:
Dim s As New Cliente.Salario
s.Importe = 2200
```

Los tipos anidables

Cualquiera de los tipos mostrados en la tabla 2.4, excepto las enumeraciones, pueden contener a su vez otros tipos. La excepción es el tipo *Module* que aunque puede contener a otros tipos, no puede usarse como tipo anidado. Una enumeración siempre puede usarse como tipo anidado.

Nota:

Los espacios de nombres también pueden anidarse y contener a su vez cualquiera de los tipos mostrados en la tabla 2.4, incluso tipos *Module*.

El nombre completo de un tipo

Tal como hemos visto, al poder declarar tipos dentro de otros tipos y estos a su vez pueden estar definidos en espacios de nombres, podemos decir que el nombre "completo" de un tipo cualquiera estará formado por el/los espacios de nombres y el/los tipos que los contiene, por ejemplo si la clase **Cliente** definida anteriormente está a su vez dentro del espacio de nombres **Ambitos**, el nombre completo será: **Ambitos.Cliente** y el nombre completo de la clase **Salario** será: **Ambitos.Cliente.Salario**.

Aunque para acceder a la clase **Cliente** no es necesario indicar el espacio de nombres, al menos si la queremos usar desde cualquier otro tipo declarado dentro de ese espacio de nombres, pero si nuestra intención es usarla desde otro espacio de nombre externo a **Ambitos**, en ese caso si que tendremos que usar el nombre completo.

Por ejemplo, en el siguiente código tenemos dos espacios de nombres que no están anidados, cada uno de ellos declara una clase y desde una de ellas queremos acceder a la otra clase, para poder hacerlo debemos indicar el nombre completo, ya que en caso contrario, el compilador de Visual Basic 2010 sería incapaz de saber a que clase queremos acceder.

```

Namespace Uno
    Public Class Clase1
        Public Nombre As String
    End Class
End Namespace

Namespace Dos
    Public Class Clase2
        Public Nombre As String
        Sub Main()
            Dim c1 As New Uno.Clase1
            c1.Nombre = "Pepe"
        End Sub
    End Class
End Namespace

```

Esto mismo lo podemos aplicar en el caso de que tengamos dos clases con el mismo nombre en espacios de nombres distintos.

Nota:

En el mismo proyecto podemos tener más de una declaración de un espacio de nombres con el mismo nombre, en estos casos el compilador lo tomará como si todas las clases definidas estuvieran dentro del mismo espacio de nombres, aunque estos estén definidos en ficheros diferentes.

Importación de espacios de nombres

Tal como hemos comentado, los espacios de nombres pueden contener otros espacios de nombres y estos a su vez también pueden contener otros espacios de nombres o clases, y como hemos visto, para poder acceder a una clase que no esté dentro del mismo espacio de nombres debemos indicar el "nombre completo".

Para evitar estar escribiendo todos los espacios de nombres en los que está la clase que nos interesa declarar, podemos usar una especie de acceso directo o para que lo entendamos mejor, podemos crear una especie de "Path", de forma que al declarar una variable, si esta no está definida en el espacio de nombres actual, el compilador busque en todos los espacios de nombres incluidos en esas rutas (paths).

Esto lo conseguimos usando la instrucción *Imports* seguida del espacio de nombres que queremos importar o incluir en el path de los espacios de nombres.

Podemos usar tantas importaciones de espacios de nombres como necesitemos y estas siempre deben aparecer al principio del fichero, justo después de las instrucciones *Options*.

Por ejemplo, si tenemos el código anterior y hacemos la importación del espacio de nombres en el que está definida la clase **Clase1**:

```
Imports Uno
```

podremos acceder a esa clase de cualquiera de estas dos formas:

```
Dim c1 As New Uno.Clase1
Dim c1 As New Clase1
```

Alias de espacios de nombres

Si hacemos demasiadas importaciones de nombres, el problema con el que nos podemos encontrar es que el *IntelliSense* de Visual Basic 2010 no sea de gran ayuda, ya que mostrará una gran cantidad de clases, y seguramente nos resultará más difícil encontrar la clase a la que queremos acceder, o también podemos encontrarnos en ocasiones en las que nos interese usar un nombre corto para acceder a las clases contenidas en un espacio de

nombres, por ejemplo, si queremos indicar de forma explícita las clases de un espacio de nombres como el de *Microsoft.VisualBasic*, podemos hacerlo de esta forma:

```
Imports vb = Microsoft.VisualBasic
```

De esta forma podemos usar el "alias" **vb** para acceder a las clases y demás tipos definidos en ese espacio de nombres.

En las figuras 2.14 2.15 podemos ver las dos formas de acceder a las clases del espacio de ese espacio de nombres, en el primer caso sin usar un alias y en el segundo usando el alias **vb**.

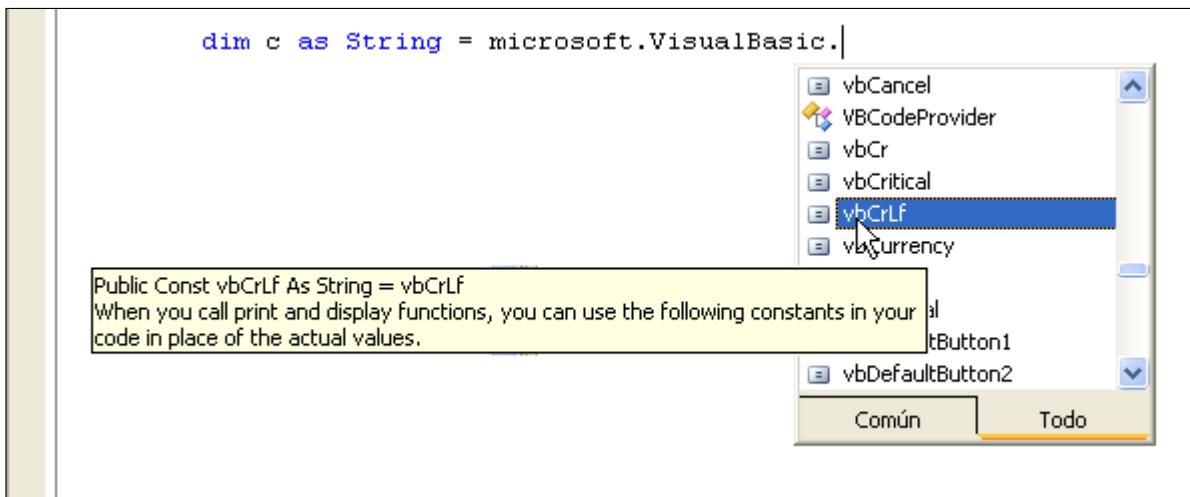


Figura 2.14. Los miembros de un espacio de nombres usando el nombre completo

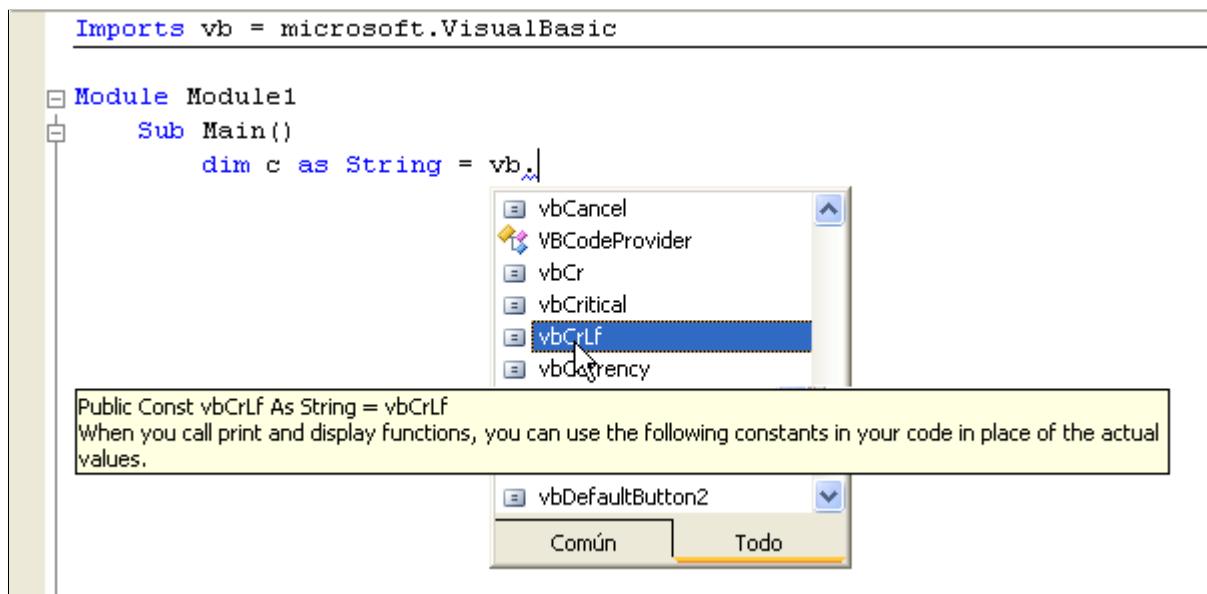


Figura 2.15. Acceder a los miembros de un espacio de nombres usando un alias

Lección 2: Clases y estructuras

- Clases
- Definir una clase
- Instanciar una clase
- Estructuras
- Accesibilidad
- Propiedades**
- Interfaces

Propiedades

Las propiedades son los miembros de los tipos que nos permiten acceder a los datos que dicho tipo manipula. Normalmente una propiedad está relacionada con un campo, de forma que el campo sea el que realmente contenga el valor y la propiedad simplemente sea una especie de método a través del cual podemos acceder a ese valor.

Debido a que el uso de las propiedades realmente nos permite acceder a los valores de una clase (o tipo), se suelen confundir los campos con las propiedades, de hecho si definimos una variable pública en una clase, ésta se comporta de manera similar, pero realmente un campo (o variable) público no es una propiedad, al menos en el sentido de que el propio .NET Framework no lo interpreta como tal, aunque en la práctica nos puede parecer que es así, ya que se utilizan de la misma forma. Pero no debemos dejarnos llevar por la comodidad y si no queremos perder funcionalidad, debemos diferenciar en nuestro código las propiedades de los campos.

Lo primero que debemos tener presente es que gracias a esta diferenciación que hace .NET Framework, podemos poner en práctica una de las características de la programación orientada a objetos: la encapsulación, de forma, que la manipulación de los datos que una clase contiene siempre se deben hacer de forma "interna" o privada a la clase, dejando a las propiedades la posibilidad de que externamente se manipulen, de forma controlada, esos datos. De esta forma tendremos mayor control sobre cómo se acceden o se asignan los valores a esos datos, ya que al definir una propiedad, tal como hemos comentado, realmente estamos definiendo un procedimiento con el cual podemos controlar cómo se acceden a esos datos.

Definir una propiedad

Debido a que una propiedad realmente nos permite acceder a un dato que la clase (o estructura) manipula, siempre tendremos un campo relacionado con una propiedad. El campo será el que contenga el valor y la propiedad será la que nos permita manipular ese valor.

En Visual Basic 2010, las propiedades las declaramos usando la instrucción *Property* y la definición de la misma termina con *End Property*, dentro de ese bloque de código tenemos que definir otros dos bloques: uno se usará a la hora de leer el valor de la propiedad

(bloque *Get*), y el otro cuando queremos asignar un valor a la propiedad (bloque *Set*).

El bloque que nos permite acceder al valor de la propiedad estará indicado por la instrucción *Get* y acaba con *End Get*, por otra parte, el bloque usado para asignar un valor a la propiedad se define mediante la instrucción *Set* y acaba con *End Set*.

Veamos como definir una propiedad en Visual Basic 2010:

```
Public Class Cliente
    Private _nombre As String
    Public Property Nombre() As String
        Get
            Return _nombre
        End Get
        Set(ByVal value As String)
            _nombre = value
        End Set
    End Property
End Class
```

Como podemos comprobar tenemos dos bloques de código, el bloque *Get* que es el que se usa cuando queremos acceder al valor de la propiedad, por tanto devolvemos el valor del campo privado usado para almacenar ese dato. El bloque *Set* es el usado cuando asignamos un valor a la propiedad, este bloque tiene definido un parámetro (*value*) que representa al valor que queremos asignar a la propiedad.

Propiedades de solo lectura

En Visual Basic 2010 podemos hacer que una propiedad sea de solo lectura, de forma que el valor que representa no pueda ser cambiado.

Para definir este tipo de propiedades solo debemos indicar el bloque *Get* de la propiedad, además de indicar de forma expresa que esa es nuestra intención, para ello debemos usar el modificador *ReadOnly* para que el compilador de Visual Basic 2010 acepte la declaración:

```
Public ReadOnly Property Hoy() As Date
    Get
        Return Date.Now
    End Get
End Property
```

Propiedades de solo escritura

De igual forma, si queremos definir una propiedad que sea de solo escritura, solo definiremos el bloque *Set*, pero al igual que ocurre con las propiedades de solo lectura, debemos indicar expresamente que esa es nuestra intención, para ello usaremos la palabra clave *WriteOnly*:

```
Public WriteOnly Property Password() As String
    Set(ByVal value As String)
        If value = "blablabla" Then
            ' ok
        End If
    End Set
End Property
```

Diferente accesibilidad para los bloques *Get* y *Set*

En las propiedades normales (de lectura y escritura), podemos definir diferentes niveles de accesibilidad a cada uno de los dos bloques que forman una propiedad. Por ejemplo, podríamos definir el bloque *Get* como público, (siempre accesible), y el bloque *Set* como *Private*, de forma que solo se puedan realizar asignaciones desde dentro de la propia clase.

Por ejemplo, el salario de un empleado podríamos declararlo para que desde cualquier punto se pueda saber el importe, pero la asignación de dicho importe solo estará accesible para los procedimientos definidos en la propia clase:

```
Public Class Empleado
    Private _salario As Decimal
    Public Property Salario() As Decimal
        Get
            Return _salario
        End Get
        Private Set(ByVal value As Decimal)
            _salario = value
        End Set
    End Property
End Class
```

Para hacer que el bloque *Set* sea privado, lo indicamos con el modificador de accesibilidad *Private*, al no indicar ningún modificador en el bloque *Get*, éste será el mismo que el de la propiedad.

Nota:

El nivel de accesibilidad de los bloques *Get* o *Set* debe ser igual o inferior que el de la propiedad, por tanto si la propiedad la declaramos como *Private*, no podemos definir como público los bloques *Get* o *Set*.

Propiedades predeterminadas

Las propiedades predeterminadas son aquellas en las que no hay que indicar el nombre de la propiedad para poder acceder a ellas, pero en Visual Basic 2010 no podemos definir como predeterminada cualquier propiedad, ya que debido a como se realizan las asignaciones de objetos en .NET, siempre debemos indicar la propiedad a la que queremos asignar el valor, porque en caso de que no se indique ninguna, el compilador interpretará que lo que queremos asignar es un objeto y no un valor a una propiedad.

Para evitar conflictos o tener que usar alguna instrucción "extra" para que se sepa si lo que queremos asignar es un valor o un objeto, en Visual Basic 2010 las propiedades predeterminadas siempre deben ser parametrizadas, es decir, tener como mínimo un parámetro.

Para indicar que una propiedad es la propiedad por defecto lo debemos hacer usando la instrucción *Default*:

```
Default Public ReadOnly Property Item(ByVal index As Integer) As Empleado
    Get
        ' ...
    End Get
End Property
```

Como vemos en este ejemplo, una propiedad por defecto puede ser de solo lectura y también de solo escritura o de lectura/escritura.

Para usar esta propiedad, al ser la propiedad por defecto, no es necesario indicar el nombre de la propiedad, aunque si así lo deseamos podemos indicarla, aunque en este caso no tendría mucha utilidad el haberla definido como propiedad por defecto:

```
Dim e As New Empleado
Dim e1 As Empleado = e(2)
' También podemos usarla indicando el nombre de la propiedad:
Dim e2 As Empleado = e.Item(2)
```

Sobrecarga de propiedades predeterminadas

Debido a que las propiedades predeterminadas de Visual Basic 2010 deben recibir un

parámetro, podemos crear sobrecargas de una propiedad predeterminada, aunque debemos recordar que para que esa sobrecarga pueda ser posible, el tipo o número de argumentos deben ser distintos entre las distintas sobrecargas, por ejemplo podríamos tener una sobrecarga que reciba un parámetro de tipo entero y otra que lo reciba de tipo cadena:

```
Default Public ReadOnly Property Item(ByVal index As Integer) As Empleado
    Get
        ' ...
    End Get
End Property

Default Public Property Item(ByVal index As String) As Empleado
    Get
        ' ...
    End Get
    Set(ByVal value As Empleado)
        ' ...
    End Set
End Property
```

Incluso como vemos en este código una de las sobrecargas puede ser de solo lectura y la otra de lectura/escritura. Lo que realmente importa es que el número o tipo de parámetros de cada sobrecarga sea diferente.

Las propiedades predeterminadas tienen sentido en Visual Basic 2010 cuando queremos que su uso sea parecido al de un array. Por tanto es habitual que las clases de tipo colección sean las más indicadas para definir propiedades por defecto. Aunque no siempre el valor devuelto debe ser un elemento de una colección o array, ya que podemos usar las propiedades predeterminadas para acceder a los miembros de una clase "normal", de forma que se devuelva un valor según el parámetro indicado, esto nos permitiría, por ejemplo, acceder a los miembros de la clase desde un bucle *For*. Si definimos una propiedad predeterminada como en el siguiente código:

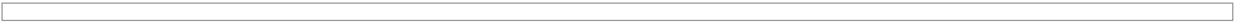
```
Public Class Articulo
    Public Descripción As String
    Public PrecioVenta As Decimal
    Public Existencias As Decimal
    Default Public ReadOnly Property Item(ByVal index As Integer) As String
        Get
            Select Case index
                Case 0
                    Return Descripción
                Case 1
                    Return PrecioVenta.ToString
                Case 2
                    Return Existencias.ToString
                Case Else
                    Return ""
            End Select
        End Get
    End Property
End Class
```

La podemos usar de esta forma:

```
For i As Integer = 0 To 2
    Console.WriteLine( art(i) )
Next
```

Resumiendo:

Las propiedades predeterminadas en Visual Basic 2010 siempre deben tener un parámetro, para que su uso se asemeje a un array, es decir, se use como indizador de la clase. Por convención, cuando se usan como indizador, el nombre de la propiedad predeterminada suele ser **Item**.



Lección 2: Clases y estructuras

- Clases
 - Definir una clase
 - Instanciar una clase
 - Estructuras
 - Accesibilidad
 - Propiedades
- Interfaces**

Interfaces

Las interfaces son un elemento bastante importante en .NET Framework, ya que de hecho se utiliza con bastante frecuencia, en esta lección veremos que son las interfaces y como utilizarlas en nuestros proyectos, también veremos que papel juegan en .NET y cómo aplicar algunas de las definidas en la biblioteca base.

¿Qué es una interfaz?

Las interfaces son una forma especial de una clase, aunque la diferencia principal con las clases es que las interfaces no contienen código ejecutable, solo definen los miembros.

Las interfaces se utilizan para indicar el "comportamiento" que tendrá una clase, o al menos qué miembros debe definir esa clase.

Para definir una interfaz en Visual Basic 2010 tenemos que usar la instrucción *Interface* seguida del nombre de la interfaz y terminar la declaración con *End Interface*:

```
Public Interface IAnimal  
    ...  
End Interface
```

Nota:

Según las indicaciones de nomenclatura de .NET Framework, se recomienda que todas las interfaces empiecen con una I mayúscula seguida del nombre al que hacer referencia la interfaz.

¿Qué contiene una interfaz?

Al principio de esta lección hemos comentado que las interfaces no contienen código, solo define los miembros que contiene. Esta definición la haremos como cualquier otra, con la diferencia de que no incluimos ningún código, solo la "firma" o el prototipo de cada uno de esos miembros.

En el siguiente código definimos una interfaz que contiene los cuatro tipos de miembros típicos de cualquier clase:

```
Public Interface IPrueba
    Sub Mostrar()
    Function Saludo(ByVal nombre As String) As String
    Property Nombre() As String
    Event DatosCambiados()
End Interface
```

El primer miembro de esta interfaz, es un método de tipo *Sub* que no recibe parámetros. El siguiente método es una función que devuelve un valor de tipo *String* y recibe un parámetro también de tipo cadena.

A continuación definimos una propiedad que devuelve una cadena.

Por último, definimos un evento.

Como podemos observar, lo único que tenemos que hacer es indicar el tipo de miembro y si recibe o no algún parámetro o argumento.

Dos cosas importantes sobre las interfaces:

- 1- No se pueden definir campos.
- 2- Los miembros de las interfaces siempre son públicos, tal como indicábamos en la tabla 2.3.

Una interfaz es un contrato

Siempre que leemos sobre las interfaces, lo primero con lo que nos solemos encontrar es que *una interfaz es un contrato*. Veamos que nos quieren decir con esa frase.

Tal como acabamos de ver, las interfaces solo definen los miembros, pero no el código a usar en cada uno de ellos, esto es así precisamente porque el papel que juegan las interfaces es el de solo indicar que es lo que una clase o estructura puede, o mejor dicho, debe implementar.

Si en una clase indicamos que queremos "implementar" una interfaz, esa clase debe definir cada uno de los miembros que la interfaz expone. De esta forma nos aseguramos de que si una clase implementa una interfaz, también implementa todos los miembros definidos en dicha interfaz.

Cuando una clase implementa una interfaz está firmando un contrato con el que se compromete a definir todos los miembros que la clase define, de hecho el propio compilador nos obliga a hacerlo.

Las interfaces y el polimorfismo

Como comentamos anteriormente, el polimorfismo es una característica que nos permite acceder a los miembros de un objeto sin necesidad de tener un conocimiento exacto de ese objeto (o de la clase a partir del que se ha instanciado), lo único que tenemos que saber es que ese objeto tiene ciertos métodos (u otros miembros) a los que podemos acceder.

También hemos comentado que las interfaces representan un contrato entre las clases que las implementan, por tanto las interfaces pueden ser, (de hecho lo son), un medio para poner en práctica esta característica de la programación orientada a objetos. Si una clase implementa una interfaz, esa clase tiene todos los miembros de la interfaz, por tanto podemos acceder a esa clase, que en principio pude sernos desconocida, desde un objeto del mismo tipo que la interfaz.

Usar una interfaz en una clase

Para poder utilizar una interfaz en una clase, o dicho de otra forma: para "implementar" los miembros expuestos por una interfaz en una clase debemos hacerlo mediante la instrucción *Implements* seguida del nombre de la interfaz:

```
Public Class Prueba  
    Implements IPrueba
```

Y como comentábamos, cualquier clase que implemente una interfaz debe definir cada uno de los miembros de esa interfaz, por eso es el propio Visual Basic el encargado de crear automáticamente los métodos y propiedades que la interfaz implementa, aunque solo inserta el "prototipo" de cada uno de esos miembros, dejando para nosotros el trabajo de escribir el código.

Usando la definición de la interfaz **IPrueba** que vimos antes, el código que añadirá VB será el siguiente:

```
Public Class Prueba  
    Implements IPrueba  
  
    Public Event DatosCambiados() Implements IPrueba.DatosCambiados  
  
    Public Sub Mostrar() Implements IPrueba.Mostrar  
  
    End Sub  
  
    Public Property Nombre() As String Implements IPrueba.Nombre  
        Get  
  
        End Get  
        Set(ByVal value As String)  
  
        End Set  
    End Property  
  
    Public Function Saludo(ByVal nombre As String) As String _  
        Implements IPrueba.Saludo  
  
    End Function  
End Class
```

Como podemos apreciar, no solo ha añadido las definiciones de cada miembro de la interfaz, sino que también añade código extra a cada uno de esos miembros: la instrucción **Implements** seguida del nombre de la interfaz y el miembro al que se hará referencia.

La utilidad de que en cada uno de los miembros se indique expresamente el método al que se hace referencia, es que podemos usar nombres diferentes al indicado en la interfaz. Por ejemplo, si implementamos esta interfaz en una clase que solo utilizará la impresora, al método **Mostrar** lo podríamos llamar **Imprimir** que sería más adecuado, en ese caso simplemente cambiamos el nombre del método de la clase para que implemente el método **Mostrar** de la interfaz:

```
Public Sub Imprimir() Implements IPrueba.Mostrar  
End Sub
```

De esta forma, aunque en la clase se llame de forma diferente, realmente hace referencia al método de la interfaz.

Acceder a los miembros implementados

Una vez que tenemos implementada una interfaz en nuestra clase, podemos acceder a esos miembros de forma directa, es decir, usando un objeto creado a partir de la clase:

```
Dim pruebal As New Prueba  
pruebal.Mostrar()
```

O bien de forma indirecta, por medio de una variable del mismo tipo que la interfaz:

```
Dim prueba1 As New Prueba  
Dim interfaz1 As IPrueba  
  
interfaz1 = prueba1  
  
interfaz1.Mostrar()
```

¿Qué ha ocurrido aquí?

Como ya comentamos anteriormente, cuando asignamos variables por referencia, realmente lo que asignamos son referencias a los objetos creados en la memoria, por tanto la variable **interfaz1** está haciendo referencia al mismo objeto que **prueba1**, aunque esa variable solo tendrá acceso a los miembros de la clase Prueba que conoce, es decir, los miembros definidos en **IPrueba**.

Si la clase define otros miembros que no están en la interfaz, la variable **interfaz1** no podrá acceder a ellos.

Saber si un objeto implementa una interfaz

Si las interfaces sirven para acceder de forma anónima a los métodos de un objeto, es normal que en Visual Basic tengamos algún mecanismo para descubrir si un objeto implementa una interfaz.

Para realizar esta comprobación podemos usar en una expresión *If/Then* la instrucción *TypeOf... Is*, de forma que si la variable indicada después de *TypeOf* contiene el tipo especificado después de *Is*, la condición se cumple:

```
If TypeOf prueba1 Is IPrueba Then  
    interfaz1 = prueba1  
  
    interfaz1.Mostrar()  
End If
```

De esta forma nos aseguramos de que el código se ejecutará solamente si la variable **prueba1** contiene una definición de la interfaz **IPrueba**.

Implementación de múltiples interfaces

En Visual Basic 2010, una misma clase puede implementar más de una interfaz. Para indicar que implementamos más de una interfaz podemos hacerlo de dos formas:

1- Usando nuevamente la instrucción *Implements* seguida del nombre de la interfaz:

```
Public Class Prueba  
    Implements IPrueba  
    Implements IComparable
```

2- Indicando las otras interfaces en la misma instrucción *Implements*, pero separándolas con comas:

```
Public Class Prueba  
    Implements IPrueba, IComparable
```

De cualquiera de las dos formas es válido implementar más de una interfaz, aunque en ambos casos siempre debemos definir los miembros de cada una de esas interfaces.

Múltiple implementación de un mismo miembro

Como acabamos de comprobar, una misma clase puede implementar más de una interfaz, y

esto nos puede causar una duda:

¿Qué ocurre si dos interfaces definen un método que es idéntico en ambas?

En principio, no habría problemas, ya que el propio Visual Basic crearía dos métodos con nombres diferentes y a cada uno le asignaría la implementación de ese método definido en cada interfaz.

Por ejemplo, si tenemos otra interfaz que define el método **Mostrar** y la implementamos en la clase **Prueba**, la declaración podría quedar de esta forma:

```
Public Interface IMostrar
    Sub Mostrar()
End Interface
```

```
Public Sub Mostrar1() Implements IMostrar.Mostrar
    End Sub
```

Aunque si ambos métodos hacen lo mismo, en este ejemplo mostrar algo, podríamos hacer que el mismo método de la clase sirva para implementar el de las dos interfaces:

```
Public Sub Mostrar() Implements IPrueba.Mostrar, IMostrar.Mostrar
    End Sub
```

Es decir, lo único que tendríamos que hacer es indicar la otra implementación separándola con una coma.

¿Dónde podemos implementar las interfaces?

Para ir acabando este tema nos queda por saber, entre otras cosas, dónde podemos implementar las interfaces, es decir, en qué tipos de datos podemos usar *Implements*.

La implementación de interfaces la podemos hacer en las clases (*Class*), estructuras (*Structure*) y en otras interfaces (*Interface*).

Debido a que una interfaz puede implementar otras interfaces, si en una clase implementamos una interfaz que a su vez implementa otras, esa clase tendrá definidas cada una de las interfaces, lo mismo ocurre con una clase que "se derive" de otra clase que implementa alguna interfaz, la nueva clase también incorporará esa interfaz.

Nota:

Cuando una interfaz implementa otras interfaces, éstas no se pueden indicar mediante *Implements*, en lugar de usar esa instrucción debemos usar *Inherits*.

```
Public Interface IPrueba2
    Inherits IMostrar
```

Si en una clase implementamos una interfaz que a su vez implementa otras interfaces, esa clase tendrá definiciones de todos los miembros de todas las interfaces, por ejemplo, si tenemos la siguiente definición de la interfaz **IPrueba2** que "implementa" la interfaz **IMostrar**:

```
Public Interface IPrueba2
    Inherits IMostrar

        Function Saludo(ByVal nombre As String) As String
        Property Nombre() As String
        Event DatosCambiados()
    End Interface
```

Y la clase **Prueba2** implementa **IPrueba2**, la definición de los miembros quedaría de la siguiente forma:

```
Public Class Prueba2
    Implements IPrueba2

    Public Sub Mostrar() Implements IMostrar.Mostrar
        End Sub

    Public Event DatosCambiados() Implements IPrueba2.DatosCambiados

    Public Property Nombre() As String Implements IPrueba2.Nombre
        Get
            End Get
            Set(ByVal value As String)
                End Set
            End Property

    Public Function Saludo(ByVal nombre As String) As String _
        Implements IPrueba2.Saludo
        End Function
    End Class
```

En este código, el método **Mostrar** se indica mediante la interfaz **IMostrar**, pero también se puede hacer por medio de **IPrueba2.Mostrar**, ya que **IPrueba2** también lo implementa (o hereda).

Si dejamos que Visual Basic cree los miembros, no tendremos problemas a la hora de definirlos. Pero si lo hacemos manualmente, aunque dentro del IDE de Visual Basic, éste nos ayuda indicándonos que interfaces implementamos y qué miembros son los que se adecuan a la declaración que estamos usando, tal como podemos comprobar en la figura 2.16:

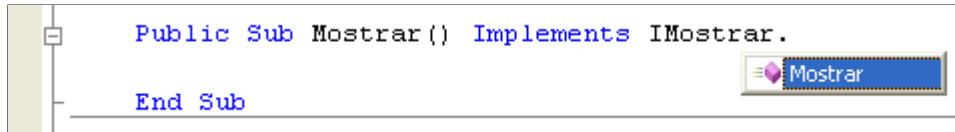


Figura 2.16. IntelliSense solo muestra los métodos que mejor se adecuan a la declaración

Un ejemplo práctico usando una interfaz de .NET

Tal como comentamos al principio, el propio .NET está "plagado" de interfaces, cada una de ellas tiene un fin concreto, por ejemplo, si queremos definir una clase que pueda ser clasificada por el propio .NET, esa clase debe implementar la interfaz *IComparable*, ya que el método *Sort*, (de la clase que contiene los elementos del tipo definido por nosotros), que es el encargado de clasificar los elementos, hará una llamada al método *IComparable.CompareTo* de cada uno de los objetos que queremos clasificar, por tanto, si la clase no ha definido esa interfaz, no podremos clasificar los elementos que contenga.

En el siguiente código tenemos la definición de una clase llamada Empleado que implementa la interfaz *IComparable* y en el método *CompareTo* hace la comprobación de que objeto es mayor o menor, si el de la propia clase o el indicado en el parámetro de esa función:

```
Public Class Empleado
    Implements IComparable

    Public Nombre As String
```

```

Public Sub New(ByVal nombre As String)
    Me.Nombre = nombre
End Sub

' Si el objeto es del tipo Empleado, comparamos los nombres.
' Si no es del tipo Empleado, devolvemos un cero
' que significa que los dos objetos son iguales.
Public Function CompareTo(ByVal obj As Object) As Integer _
    Implements System.IComparable.CompareTo
    If TypeOf obj Is Empleado Then
        Dim el As Empleado = CType(obj, Empleado)

        Return String.Compare(Me.Nombre, el.Nombre)
    Else
        Return 0
    End If
End Function
End Class

```

En el método *CompareTo* hacemos una comprobación de que el objeto con el que debemos realizar la comparación es del tipo **Empleado**, en ese caso convertimos el objeto pasado en uno del tipo **Empleado** y comparamos los nombres.

Si el objeto que recibe el método no es del tipo **Empleado**, devolvemos un cero, para indicar que no haga ninguna clasificación, ya que ese valor indica que los dos objetos son iguales.

Esta comparación no es estrictamente necesaria, ya que si no indicamos el valor que debe devolver una función, devolverá un valor cero, al menos en este caso, ya que el tipo a devolver es un número entero.

Esta clase la podemos usar de esta forma:

```

' Una colección de datos del tipo Empleado.
Dim empleados As New System.Collections.Generic.List(Of Empleado)

' Añadimos varios empleados a la colección.
empleados.Add(New Empleado("Pepe"))
empleados.Add(New Empleado("Bernardo"))
empleados.Add(New Empleado("Juan"))
empleados.Add(New Empleado("Ana"))

' Clasificamos los empleados de la colección.
empleados.Sort()

' Mostramos los datos una vez clasificados.
For Each el As Empleado In empleados
    Console.WriteLine(el.Nombre)
Next

```

Lección 3: Manejo de excepciones

- Manejo de excepciones

Introducción

Es indiscutible que por mucho que nos lo propongamos, nuestras aplicaciones no estarán libres de errores, y no nos referimos a errores sintácticos, ya que, afortunadamente, el IDE (*Integrated Development Environment*, entorno de desarrollo integrado) de Visual Basic 2010 nos avisará de cualquier error sintáctico e incluso de cualquier asignación no válida (al menos si tenemos activado *Option Strict On*), pero de lo que no nos avisará, como es lógico, será de los errores que se produzcan en tiempo de ejecución. Para estos casos, Visual Basic pone a nuestra disposición el manejo de excepciones, veamos pues cómo utilizarlo, sobre todo el sistema de excepciones estructuradas que es el recomendable para cualquier desarrollo con .NET Framework.

Manejo de excepciones

- Manejo de excepciones
 - Manejo de excepciones no estructuradas
 - Manejo de excepciones estructuradas
 - Bloque Try
 - Bloque Catch
 - Varias capturas de errores en un mismo bloque Try/Catch
 - Evaluación condicional en un bloque Catch
 - Bloque Finally
 - Captura de errores no controlados

Lección 3: Manejo de excepciones

Manejo de excepciones

Manejo de excepciones

En Visual Basic 2010 el tratamiento de errores (excepciones) podemos hacerlo de dos formas distintas, dependiendo de que utilicemos el tratamiento de errores heredado de versiones anteriores o el recomendado para la plataforma .NET: el control de errores estructurado, con idea de hacerlo de una forma más "ordenada".

En esta lección solamente veremos cómo tratar los errores de forma estructurada, porque, como hemos comentado es la forma recomendada de hacerlo en .NET Framework.

Manejo de excepciones no estructuradas

Como acabamos de comentar, no trataremos o explicaremos cómo trabajar con el tratamiento de errores no estructurados, pero al menos queremos hacer una aclaración para que no nos llevemos una sorpresa si nos decidimos a usarlo:

No podemos usar los dos sistemas de tratamiento de errores al mismo tiempo, por lo menos en un mismo método o propiedad, o utilizamos *On Error* o utilizamos *Try/Catch*. De todas formas, si escribimos nuestro código con el IDE (entorno integrado) de Visual Studio 2010, éste nos avisará de que no podemos hacer esa mezcla.

Manejo de excepciones estructuradas

Las excepciones en Visual Basic 2010 las podemos controlar usando las instrucciones *Try / Catch / Finally*. Estas instrucciones realmente son bloques de instrucciones, al estilo de *If / Else*.

- Cuando queramos controlar una parte del código que puede producir un error lo incluimos dentro del bloque *Try*.
- Si se produce un error, éste lo podemos detectar en el bloque *Catch*.
- Por último, independientemente de que se produzca o no una excepción, podemos ejecutar el código que incluyamos en el bloque *Finally*, para indicar el final del bloque de control de excepciones lo haremos con *End Try*.

Cuando creamos una estructura de control de excepciones no estamos obligados a usar los tres bloques, aunque el primero: *Try* si es necesario, ya que es el que le indica al compilador que tenemos intención de controlar los errores que se produzcan. Por tanto podemos crear un "manejador" de excepciones usando los tres bloques, usando *Try* y *Catch* o usando *Try* y *Finally*.

Veamos ahora con más detalle cada uno de estos bloques y que es lo que podemos hacer en cada uno de ellos.

Bloque Try

En este bloque incluiremos el código en el que queremos comprobar los errores. El código a usar será un código normal, es decir, no tenemos que hacer nada en especial, ya que en el momento que se produzca el error se usará (si hay) el código del bloque *Catch*.

Bloque Catch

Si se produce una excepción, ésta la capturamos en un bloque *Catch*.

En el bloque *Catch* podemos indicar que tipo de excepción queremos capturar, para ello usaremos una variable de tipo *Exception*, la cual puede ser del tipo de error específico que queremos controlar o de un tipo genérico.

Por ejemplo, si sabemos que nuestro código puede producir un error al trabajar con ficheros, podemos usar un código como este:

```
Try
    ' código para trabajar con ficheros, etc.
Catch ex As System.IO.IOException
    ' el código a ejecutar cuando se produzca ese error
End Try
```

Si nuestra intención es capturar todos los errores que se produzcan, es decir, no queremos hacer un filtro con errores específicos, podemos usar la clase *Exception* como tipo de excepción a capturar. La clase *Exception* es la más genérica de todas las clases para manejo de excepciones, por tanto capturará todas las excepciones que se produzcan.

```
Try
    ' código que queremos controlar
Catch ex As Exception
    ' el código a ejecutar cuando se produzca cualquier error
End Try
```

Aunque si no vamos usar la variable indicada en el bloque *Catch*, pero queremos que no se detenga la aplicación cuando se produzca un error, podemos hacerlo de esta forma:

```
Try
    ' código que queremos controlar
Catch
    ' el código a ejecutar cuando se produzca cualquier error
End Try
```

Varias capturas de errores en un mismo bloque Try/Catch

En un mismo *Try* podemos capturar diferentes tipos de errores, para ello podemos incluir varios bloques *Catch*, cada uno de ellos con un tipo de excepción diferente.

Es importante tener en cuenta que cuando se produce un error y usamos varios bloques *Catch*, Visual Basic buscará la captura que mejor se adapte al error que se ha producido, pero siempre lo hará examinando los diferentes bloques *Catch* que hayamos indicado empezando por el indicado después del bloque *Try*, por tanto deberíamos poner las más

genéricas al final, de forma que siempre nos aseguremos de que las capturas de errores más específicas se intercepten antes que las genéricas.

Evaluación condicional en un bloque Catch

Además de indicar la excepción que queremos controlar, en un bloque *Catch* podemos añadir la cláusula *When* para evaluar una expresión. Si la evaluación de la expresión indicada después de *When* devuelve un valor verdadero, se procesará el bloque *Catch*, en caso de que devuelva un valor falso, se ignorará esa captura de error.

Esto nos permite poder indicar varios bloques *Catch* que detecten el mismo error, pero cada una de ellas pueden tener diferentes expresiones indicadas con *When*.

En el siguiente ejemplo, se evalúa el bloque *Catch* solo cuando el valor de la variable *y* es cero, en otro caso se utilizará el que no tiene la cláusula *When*:

```
Dim x, y, r As Integer
Try
    x = CInt(Console.ReadLine())
    y = CInt(Console.ReadLine())
    r = x \ y
    Console.WriteLine("El resultado es: {0}", r)
Catch ex As Exception When y = 0
    Console.WriteLine("No se puede dividir por cero.")
Catch ex As Exception
    Console.WriteLine(ex.Message)
End Try
```

Bloque Finally

En este bloque podemos indicar las instrucciones que queremos que se ejecuten, se produzca o no una excepción. De esta forma nos aseguramos de que siempre se ejecutará un código, por ejemplo para liberar recursos, se haya producido un error o no.

Nota:

Hay que tener en cuenta de que incluso si usamos *Exit Try* para salir del bloque de control de errores, se ejecutará el código indicado en el bloque *Finally*.

Captura de errores no controlados

Como es lógico, si no controlamos las excepciones que se puedan producir en nuestras aplicaciones, estas serán inicialmente controladas por el propio runtime de .NET, en estos casos la aplicación se detiene y se muestra el error al usuario. Pero esto es algo que no deberíamos consentir, por tanto siempre deberíamos detectar todos los errores que se produzcan en nuestras aplicaciones, pero a pesar de que lo intentemos, es muy probable que no siempre podamos conseguirlo.

Por suerte, en Visual Basic 2010 tenemos dos formas de interceptar los errores no controlados:

La primera es iniciando nuestra aplicación dentro de un bloque *Try/Catch*, de esta forma, cuando se produzca el error, se capturará en el bloque *Catch*.

La segunda forma de interceptar los errores no controlados es mediante el evento: *UnhandledException*, disponible por medio del objeto *My.Application*.

Nota:

De los eventos nos ocuparemos en la siguiente lección, pero como el evento *UnhandledException* está directamente relacionado con la captura de errores, lo mostramos en esta, aunque recomendamos al lector que esta sección la vuelva a leer después de ver todo lo relacionado con los eventos.

Este evento se "dispara" cuando se produce un error que no hemos interceptado, por tanto podríamos usarlo para prevenir que nuestra aplicación se detenga o bien para guardar en un fichero **.log** la causa de dicho error para posteriormente actualizar el código y prevenirla. Ya que cuando se produce el evento *UnhandledException*, podemos averiguar el error que se ha producido e incluso evitar que la aplicación finalice. Esta información la obtenemos mediante propiedades expuestas por el segundo parámetro del evento, en particular la propiedad *Exception* nos indicará el error que se ha producido y por medio de la propiedad *ExitApplication* podemos indicar si terminamos o no la aplicación.

Nota:

Cuando ejecutamos una aplicación desde el IDE, los errores no controlados siempre se producen, independientemente de que tengamos o no definida la captura de errores desde el evento *UnhandledException*. Ese evento solo se producirá cuando ejecutemos la aplicación fuera del IDE de Visual Basic.

Lección 4: Eventos y delegados

- Eventos
- Definir y producir eventos en una clase
- Delegados
- Definir un evento bien informado

Introducción

La forma que tienen nuestras clases y estructuras de comunicar que algo está ocurriendo, es por medio de eventos. Los eventos son mensajes que se lanzan desde una clase para informar al "cliente" que los utiliza de que está pasando algo.

Seguramente estaremos acostumbrados a usarlos, incluso sin tener una noción consciente de que se tratan de eventos, o bien porque es algo tan habitual que no le prestamos mayor atención, es el caso de las aplicaciones de escritorio, cada vez que presionamos un botón, escribimos algo o movemos el mouse se están produciendo eventos.

El compilador de Visual Basic 2010 nos facilita mucho la creación de los eventos y "esconde" todo el proceso que .NET realmente hace "en la sombra". Ese trabajo al que nos referimos está relacionado con los delegados, una palabra que suele aparecer en cualquier documentación que trate sobre los eventos.

Y es que, aunque Visual Basic 2010 nos oculte, o facilite, el trabajo con los eventos, éstos están estrechamente relacionados con los delegados. En esta lección veremos que son los delegados y que relación tienen con los eventos, también veremos que podemos tener mayor control sobre cómo se interceptan los eventos e incluso cómo y cuando se asocian los eventos en la aplicación cliente, aunque primero empezaremos viendo cómo declarar y utilizar eventos en nuestros tipos de datos.

Eventos y delegados

• Eventos

- Interceptar los eventos de los controles de un formulario
 - Interceptar eventos en Visual Basic 2010
- Asociar un evento con un control
- Formas de asociar los eventos con un control
 - 1- Asociar el evento manualmente por medio de Handles
 - 2- Asociar el evento desde la ventana de código
 - 3- Asociar el evento desde el diseñador de formularios
- Asociar varios eventos a un mismo procedimiento
- Declarar una variable para asociar eventos con Handles

- [Definir y producir eventos en una clase](#)

- Definir eventos en una clase
- Producir un evento en nuestra clase
- Otra forma de asociar los eventos de una clase con un método
 - Asociar eventos mediante AddHandler
 - Desasociar eventos mediante RemoveHandler

- [Delegados](#)

- ¿Qué ocurre cuando se asigna y se produce un evento?
- ¿Qué papel juegan los delegados en todo este proceso?
- Definición "formal" de delegado
- Utilizar un delegado para acceder a un método

- [Definir un evento bien informado con Custom Event](#)

Lección 4: Eventos y delegados

Eventos

- Definir y producir eventos en una clase
- Delegados
- Definir un evento bien informado

Eventos

Como hemos comentado en la introducción, en Visual Basic 2010 podemos usar los eventos de una forma bastante sencilla, al menos si la comparamos con otros lenguajes de la familia .NET.

En las secciones que siguen veremos cómo utilizar los eventos que producen las clases de .NET, empezaremos con ver cómo utilizar los eventos de los formularios, que con toda seguridad serán los que estamos más acostumbrados a usar.

También veremos las distintas formas que tenemos de asociar un método con el evento producido por un control (que al fin y al cabo es una clase).

Interceptar los eventos de los controles de un formulario

Debido a que aún no hemos visto el módulo dedicado a las aplicaciones de Windows, en las que se utilizan los "clásicos" formularios, no vamos a entrar en detalles sobre cómo crear un formulario ni como añadir controles, etc., todo eso lo veremos en el siguiente módulo. Para simplificar las cosas, veremos cómo se interceptan en un formulario de la nueva versión de Visual Basic, aunque sin entrar en demasiados detalles.

La forma más sencilla de asociar el evento de un control con el código que se usará, es haciendo doble pulsación en el control cuando estamos en modo de diseño; por ejemplo, si en nuestro formulario tenemos un botón, al hacer doble pulsación sobre él tendremos asociado el evento *Click* del botón, ya que ese es el evento predeterminado de los controles *Button*, y desde el diseñador de formularios de Windows, al realizar esa doble pulsación, siempre se muestra el evento predeterminado del control en cuestión.

Interceptar eventos en Visual Basic 2010

Aunque en Visual Basic 2010 la forma de asignar los eventos predeterminados de los controles es como hemos comentado anteriormente, es decir, haciendo doble pulsación en el control, la declaración del código usado para interceptar el evento es como el mostrado en el siguiente código:

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

Lo primero que podemos notar es que en Visual Basic 2010 se utilizan dos argumentos, esto siempre es así en todos los eventos producidos por los controles. El primero indica el control que produce el evento, (en nuestro ejemplo sería una referencia al control **Button1**), y el segundo normalmente contiene información sobre el evento que se produce, si el evento en cuestión no proporciona información extra, como es el caso del evento *Click*, ese parámetro será del tipo *EventArgs*. Sin embargo en otros eventos, por ejemplo, los relacionados con el mouse, el segundo argumento tendrá información que nos puede resultar útil, por ejemplo para saber que botón se ha usado o cual es la posición del cursor, en la figura 2.17 podemos ver las propiedades relacionadas con el evento *MouseEventArgs*:

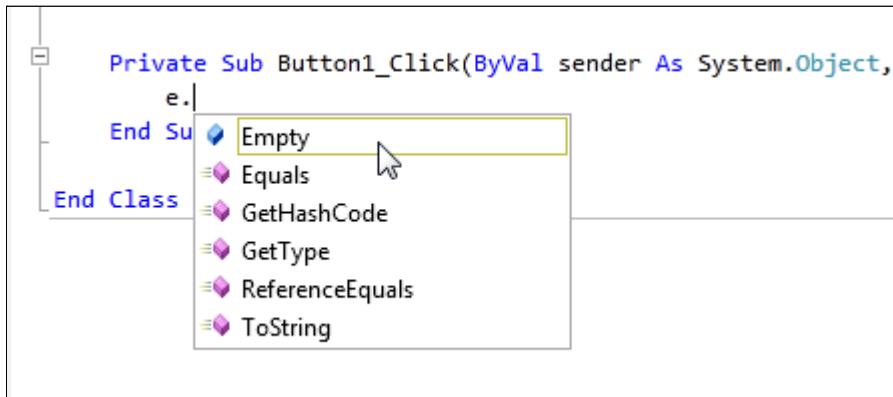


Figura 2.17. Propiedades relacionadas con un evento del mouse

Asociar un evento con un control

Siguiendo con el código que intercepta el evento *Click* de un botón, podemos apreciar que el IDE de Visual Basic 2010 añade al final de la declaración del procedimiento de evento la instrucción *Handles* seguida del control y el evento que queremos interceptar: **Handles Button1.Click**, esta la forma habitual de hacerlo en Visual Basic 2010, aunque también hay otras formas de "ligar" un método con un evento, como tendremos ocasión de comprobar más adelante.

Nota:

En Visual Basic 2010 el nombre del procedimiento de evento no tiene porqué estar relacionado con el evento, aunque por defecto, el nombre usado es el que habitualmente se ha utilizado por años en otros entornos de desarrollo, y que se forma usando el nombre del control seguido de un guión bajo y el nombre del evento, pero que en cualquier momento lo podemos cambiar, ya que en Visual Basic 2010 no hay ninguna relación directa entre ese nombre y el evento que queremos interceptar.

Tal como resaltamos en la nota anterior, en Visual Basic 2010, el nombre asociado a un evento puede ser el que queramos, lo realmente importante es que indiquemos la instrucción *Handles* seguida del evento que queremos interceptar. Aunque, como veremos a continuación, también hay otras formas de "relacionar" los eventos con el método usado para recibir la notificación.

Formas de asociar los eventos con un control

Cuando estamos trabajando con el diseñador de formularios, tenemos tres formas de asociar un evento con el código correspondiente:

La forma más sencilla es la expuesta anteriormente, es decir, haciendo doble click en el control, esto hará que se muestre el evento predeterminado del control. En el caso del

control *Button*, el evento predeterminado es el evento *Click*.

Si queremos escribir código para otros eventos podemos hacerlo de tres formas, aunque la primera que explicaremos no será la más habitual, ya que debemos saber exactamente qué parámetros utiliza el evento.

1- Asociar el evento manualmente por medio de Handles

Con esta forma, simplemente escribimos el nombre del procedimiento (puede ser cualquier nombre), indicamos los dos argumentos que recibe: el primero siempre es de tipo *Object* y el segundo dependerá del tipo de evento, y finalmente por medio de la cláusula *Handles* lo asociamos con el control y el evento en cuestión.

2- Asociar el evento desde la ventana de código

En Visual Basic 2010 también podemos seleccionar los eventos disponibles de una lista desplegable. Esto lo haremos desde la ventana de código. En la parte superior derecha tenemos una lista con los controles que hemos añadido al formulario, seleccionamos el control que nos interese y en la lista que hay a su izquierda tenemos los eventos que ese control produce. Por tanto podemos seleccionar de esa lista de eventos el que nos interese interceptar, tal como podemos ver en la figura 2.18

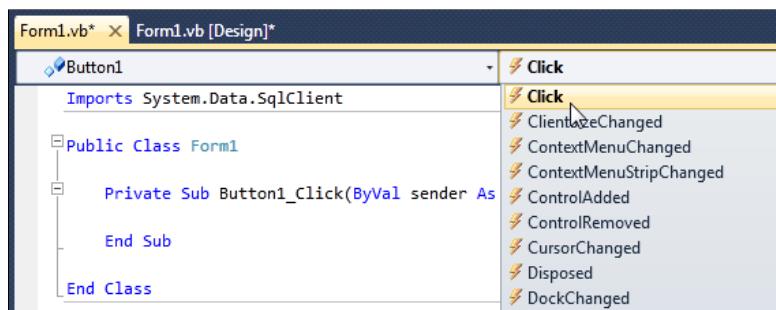


Figura 2.18. Lista de eventos de un control

De esta forma el propio IDE será el que cree el "esqueleto" del procedimiento de evento usando los parámetros adecuados.

3- Asociar el evento desde el diseñador de formularios

La tercera forma de asociar un evento con un control, es hacerlo desde el diseñador de formularios. En la ventana de propiedades del control, (tal como podemos apreciar en la figura 2.19), tenemos una lista con los eventos más importantes de cada control, seleccionando el evento de esa lista y haciendo doble-click en el que nos interese, conseguiremos exactamente el mismo resultado que con el paso anterior.

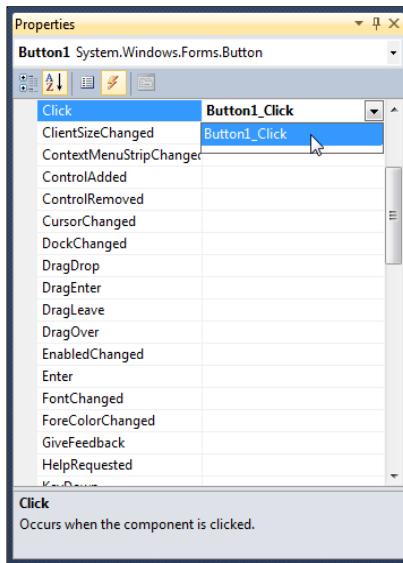


Figura 2.19. Ventana de propiedades con los eventos del control seleccionado

Asociar varios eventos a un mismo procedimiento

Cuando utilizamos *Handles* para asociar eventos, podemos indicar que un mismo procedimiento sirva para interceptar varios eventos. Veamos un caso práctico en el que tenemos varios controles de tipo *TextBox* y queremos que cuando reciban el foco siempre se ejecute el mismo código, por ejemplo, que se seleccione todo el texto que contiene, podemos hacerlo indicando después de la cláusula *Handles* todos los controles que queremos asociar con ese procedimiento.

En el siguiente código indicamos tres controles *TextBox*:

```
Private Sub TextBox1_Enter( _
    ByVal sender As System.Object, _
    ByVal e As System.EventArgs) _
Handles TextBox1.Enter, TextBox2.Enter, TextBox3.Enter

End Sub
```

Esta asociación la podemos hacer manualmente, simplemente indicando en la cláusula *Handles* cada uno de los eventos a continuación del anterior separándolos por comas. O bien desde el diseñador de formularios. En este segundo caso, cuando seleccionamos un control y desde la ventana de propiedades, seleccionamos un evento, nos muestra los procedimientos que tenemos definidos en nuestro código que utilizan los mismos parámetros que el evento en cuestión, si seleccionamos uno de esos procedimientos, el propio IDE añadirá ese control a la lista *Handles*.

Como vemos en la figura 2.20 podemos usar el procedimiento **TextBox1_Enter** para asociarlo al evento *Enter* del control **TextBox2**:

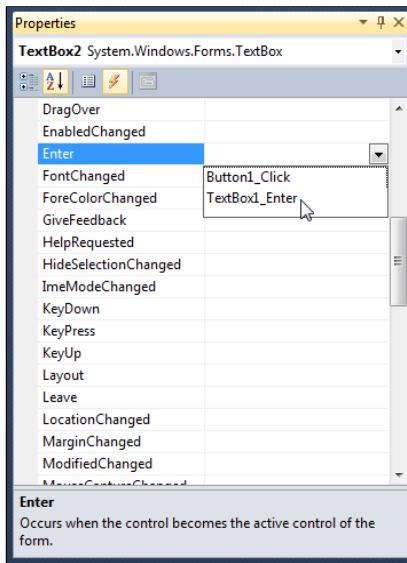


Figura 2.20. Lista de procedimientos "compatibles" con un evento

Declarar una variable para asociar eventos con Handles

Para que podamos usar la instrucción *Handles* para asociar manualmente un procedimiento con un evento, o para que el diseñador de Visual Basic 2010 pueda hacerlo, la variable del control o clase que tiene los eventos que queremos interceptar tenemos que declararla con la instrucción *WithEvents*.

De estos detalles se encarga el propio IDE de Visual Basic 2010, por tanto no debemos preocuparnos de cómo declarar los controles para que se pueda usar *Handles* en el método del procedimiento que recibe la notificación del evento, pero como los controles de .NET realmente son clases, veamos cómo declara el VB los controles, (en este caso un control llamado **Button1**), para a continuación compararlo con una clase definida por nosotros.

```
Friend WithEvents Button1 As System.Windows.Forms.Button
```

Si en lugar de estar trabajando con formularios y controles, lo hacemos con clases "normales", la forma de declarar una variable que tiene eventos es por medio de la instrucción *WithEvents*. Por ejemplo, en esta declaración indicamos que tenemos intención de usar los eventos que la clase **Empleado** exponga:

```
Private WithEvents unEmpleado As Empleado
```

Y posteriormente podremos definir los métodos de eventos usando la instrucción *Handles*:

```
Private Sub unEmpleado_DatosCambiados() Handles unEmpleado.DatosCambiados  
End Sub
```

Nota:

Usar *WithEvents* y *Handles* es la forma más sencilla de declarar y usar una variable que accede a una clase que produce eventos, pero como ya hemos comentado, no es la única forma que tenemos de hacerlo en Visual Basic 2010, tal como tendremos oportunidad de comprobar.

Lección 4: Eventos y delegados

- Eventos

Definir y producir eventos en una clase

- Delegados

- Definir un evento bien informado

Definir y producir eventos en una clase

En la lección anterior hemos visto cómo interceptar eventos, en esta veremos cómo definirlos en nuestras clases y cómo producirlos, para que el cliente que los intercepte sepa que algo ha ocurrido en nuestra clase.

Definir eventos en una clase

Para definir un evento en una clase usamos la instrucción *Event* seguida del nombre del evento y opcionalmente indicamos los parámetros que dicho evento recibirá.

En el siguiente trozo de código definimos un evento llamado **DatosModificados** que no utiliza ningún argumento:

```
Public Event DatosModificados()
```

Esto es todo lo que necesitamos hacer en Visual Basic 2010 para definir un evento.

Como podemos comprobar es muy sencillo, ya que solo tenemos que usar la instrucción *Event*. Aunque *detrás del telón* ocurren otras cosas de las que, al menos en principio, no debemos preocuparnos, ya que es el propio compilador de Visual Basic 2010 el que se encarga de esos "pequeños detalles".

Producir un evento en nuestra clase

Para producir un evento en nuestra clase, y de esta forma notificar a quién quiera interceptarlo, simplemente usaremos la instrucción *RaiseEvent* seguida del evento que queremos producir. Cuando escribimos esa instrucción en el IDE de Visual Basic 2010, nos mostrará los distintos eventos que podemos producir, tal como vemos en la figura 2.21:

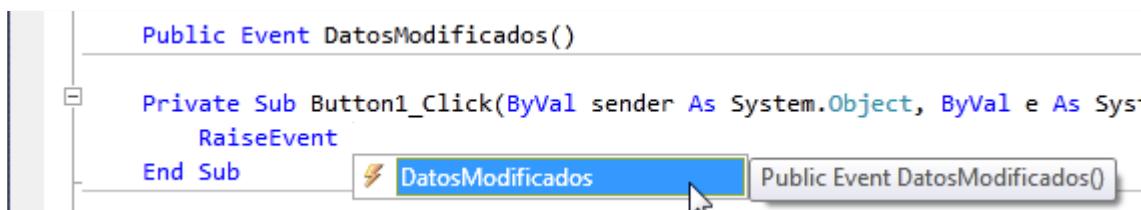


Figura 2.21. Lista de eventos que podemos producir

Esta es la forma más sencilla de definir y lanzar eventos en Visual Basic 2010.

Otra forma de asociar los eventos de una clase con un método

Tal como hemos estado comentando, la forma más sencilla de declarar una variable para interceptar eventos es declarándola usando *WithEvents* y para interceptar los eventos lo hacemos por medio de la instrucción *Handles*.

Esta forma, es a todas luces la más recomendada, no solo por la facilidad de hacerlo, sino porque también tenemos la ventaja de que todas las variables declaradas con *WithEvents* se muestran en la lista desplegable de la ventana de código, tal como podemos apreciar en la figura 2.22:

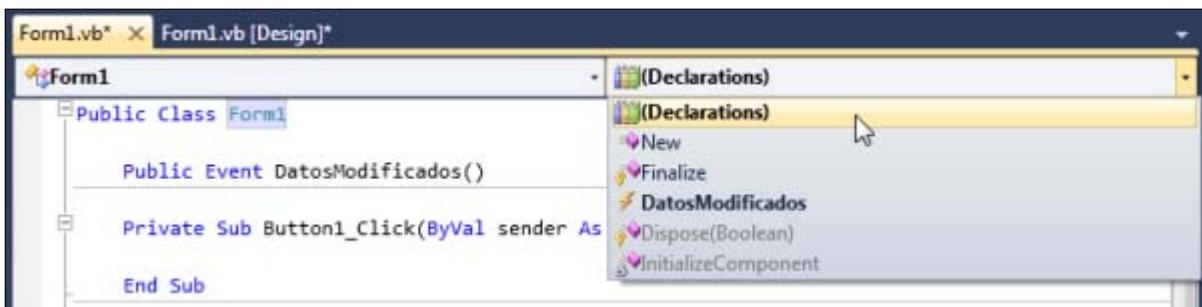


Figura 2.22. Lista de objetos que producen eventos

Y de esta forma podemos seleccionar la variable y posteriormente elegir el evento a interceptar, tal como vimos en la figura 2.18.

Asociar eventos mediante AddHandler

Pero Visual Basic 2010 también proporciona otra forma de asociar un procedimiento con un evento. Aunque en este caso es algo más manual que todo lo que hemos visto y, de alguna forma está más ligado con los delegados, y como los delegados los veremos dentro de poco, ahora solamente mostraremos la forma de hacerlo y después veremos con algo de más detalle cómo funciona.

La forma de de asociar eventos con su correspondiente método es por medio de la instrucción *AddHandler*. A esta instrucción le pasamos dos argumentos, el primero es el evento a asociar y el segundo es el procedimiento que usaremos cuando se produzca dicho evento. Este último parámetro tendremos que indicarlo mediante la instrucción *AddressOf*, que sirve para pasar una referencia a una función o procedimiento, y precisamente eso es lo que queremos hacer: indicarle que procedimiento debe usar cuando se produzca el evento:

```
AddHandler unEmpleado.DatosCambiados, AddressOf unEmpleado_DatosCambiados
```

En este caso, el uso de *AddressOf* es una forma "fácil" que tiene Visual Basic 2010 de asociar un procedimiento de evento con el evento. Aunque por el fondo, (y sin que nos enteremos), realmente lo que estamos usando es un constructor a un delegado.

La ventaja de usar esta forma de asociar eventos con el procedimiento, es que podemos hacerlo con variables que no están declaradas con *WithEvents*, realmente esta sería la única forma de asociar un procedimiento de evento con una variable que no hemos declarado con *WithEvents*.

Desasociar eventos mediante RemoveHandler

De la misma forma que por medio de *AddHandler* podemos asociar un procedimiento con un

evento, usando la instrucción *RemoveHandler* podemos hacer el proceso contrario: desligar un procedimiento del evento al que previamente estaba asociado.

Los parámetros a usar con *RemoveHandler* son los mismos que con *AddHandler*.

Podemos usar *RemoveHandler* tanto con variables y eventos definidos con *AddHandler* como con variables declaradas con *WithEvents* y ligadas por medio de *Handles*.

Esto último es así porque cuando nosotros definimos los procedimientos de eventos usando la instrucción *Handles*, es el propio Visual Basic el que internamente utiliza *AddHandler* para ligar ese procedimiento con el evento en cuestión. Saber esto nos facilitará comprender mejor cómo funciona la declaración de eventos mediante la instrucción *Custom*, aunque de este detalle nos ocuparemos después de ver que son los delegados.

Lección 4: Eventos y delegados

- Eventos
 - Definir y producir eventos en una clase
- Delegados**
- Definir un evento bien informado

Delegados

Como hemos comentado anteriormente, los eventos son acciones que una clase puede producir cuando ocurre algo. De esta forma podemos notificar a las aplicaciones que hayan decidido interceptar esos mensajes para que tomen las acciones que crean conveniente.

Visual Basic 2010 esconde al desarrollador prácticamente todo lo que ocurre cada vez que definimos, lanzamos o interceptamos un evento, nosotros solo vemos una pequeña parte de todo el trabajo que en realidad se produce, y el que no lo veamos no quiere decir que no esté ocurriendo. También es cierto que no debe preocuparnos demasiado si no sabemos lo que está pasando, pero si somos consciente de que es lo que ocurre, puede que nos ayude a comprender mejor todo lo relacionado con los eventos.

¿Qué ocurre cuando se asigna y se produce un evento?

Intentemos ver de forma sencilla lo que ocurre "por dentro" cada vez que definimos un método que intercepta un evento y cómo hace el Visual Basic para comunicarse con el receptor de dicho evento.

1. Cuando Visual Basic se encuentra con el código que le indica que un método debe interceptar un evento, ya sea mediante *AddHandler* o mediante el uso de *Handles*, lo que hace es añadir la dirección de memoria de ese método a una especie de array. En la figura 2.23 podemos ver un diagrama en el que un mismo evento lo interceptan tres clientes, cuando decimos que un cliente intercepta un evento, realmente nos referimos a que hay un método que lo intercepta y el evento realmente guarda la dirección de memoria de ese método.

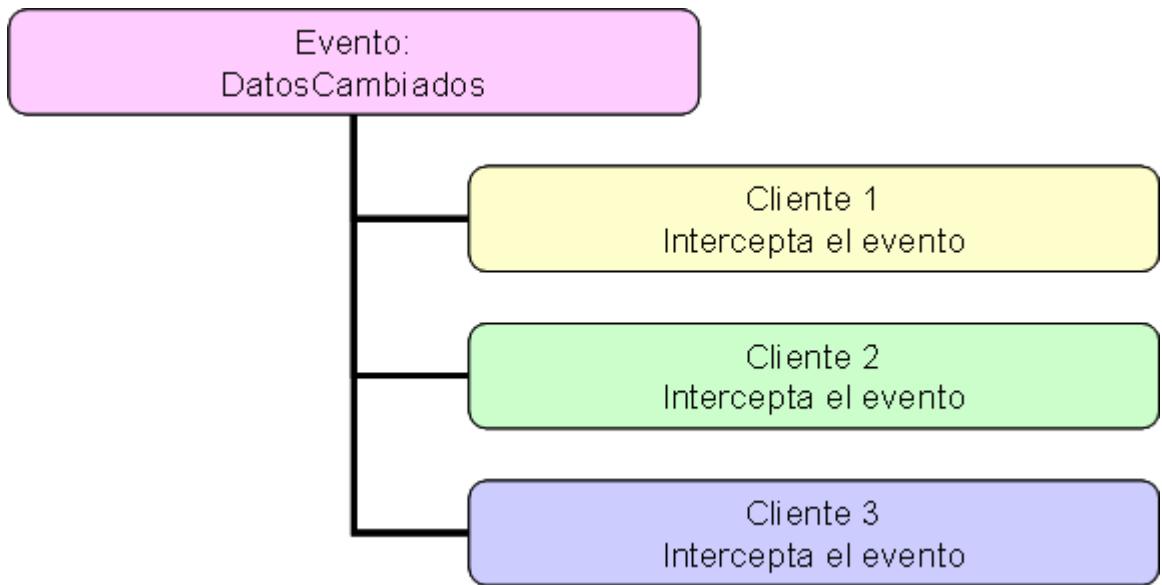


Figura 2.23. El evento guarda la dirección de memoria de cada método que lo intercepta

2. Cuando usamos la instrucción *RaiseEvent* para producir el evento, se examina esa lista de direcciones y se manda el mensaje a cada uno de los métodos que tenemos en el "array".
En este caso, lo que realmente ocurre es que se hace una llamada a cada uno de los métodos, de forma que se ejecute el código al que tenemos acceso mediante la dirección de memoria almacenada en la lista.
3. Cuando usamos la instrucción *RemoveHandler*, le estamos indicando al evento que elimine de la lista el método indicado en esa instrucción, de esta forma, la próxima vez que se produzca el evento, solo se llamará a los métodos que actualmente estén en la lista.

Tanto el agregar nuevos métodos a esa lista como quitarlos, lo podemos hacer en tiempo de ejecución, por medio de *AddHandler* y *RemoveHandler* respectivamente. Ya que la instrucción *Handles* solo la podemos usar en tiempo de diseño.

Es más, podemos incluso indicar que un mismo evento procese más de un método en una misma aplicación o que un mismo método sea llamado por más de un evento. Ya que lo que realmente necesita cada evento es que exista un método que tenga una "firma" concreta: la indicada al declarar el evento.

¿Qué papel juegan los delegados en todo este proceso?

Veamos primero que papel tienen los delegados en todo este proceso y después veremos con más detalle lo que "realmente" es un delegado.

1. Cuando definimos un evento, realmente estamos definiendo un delegado, (que en el fondo es una clase con un tratamiento especial), y un método del mismo tipo que el delegado.
2. Cuando indicamos que un método intercepte un evento, realmente estamos llamando al constructor del delegado, al que le pasamos la dirección de memoria del método. El delegado almacena cada una de esas direcciones de memoria para posteriormente usarlas.
3. Cuando se produce el evento, (por medio de *RaiseEvent*), realmente estamos llamando al delegado para que acceda a todas las "direcciones" de memoria que tiene almacenadas y ejecute el código que hayamos definido en cada uno de esos métodos.

Como podemos comprobar, y para decirlo de forma simple, un delegado realmente es la forma que tiene .NET para definir un puntero. La diferencia principal es que los punteros, (no vamos a entrar en demasiados detalles sobre los punteros, ya que no estamos en un curso de C/C++), no tienen forma de comprobar si están accediendo a una dirección de memoria correcta o, para decirlo de otra forma, a una dirección de memoria "adecuada". En .NET, los "punteros" solo se pueden usar mediante delegados, y éstos solamente pueden acceder a direcciones de memoria que tienen la misma "firma" con el que se han definido. Para que lo entendamos un poco mejor, es como si los delegados solo pudieran acceder a sitios en la memoria que contienen un método con la misma "interfaz" que el que ha definido el propio delegado.

Seguramente es difícil de entender, y la principal razón es que hemos empezado la casa por el techo.

Por tanto, veamos a continuación una definición "formal" de qué es un delegado y veamos varios ejemplos para que lo comprendamos mejor.

Definición "formal" de delegado

Veamos que nos dice la documentación de Visual Basic sobre los delegados:

"Un delegado es una clase que puede contener una referencia a un método. A diferencia de otras clases, los delegados tienen un prototipo (firma) y pueden guardar referencias únicamente a los métodos que coinciden con su prototipo."

Esta definición, al menos en lo que respecta a su relación con los eventos, viene a decir que los delegados definen la forma en que debemos declarar los métodos que queramos usar para interceptar un evento.

Si seguimos buscando más información sobre los delegados en la documentación de Visual Basic, también nos encontramos con esta definición:

"Los delegados habilitan escenarios que en otros lenguajes se han resuelto con punteros a función. No obstante, a diferencia de los punteros a función, los delegados están orientados a objetos y proporcionan seguridad de tipos."

Que es lo que comentamos en la sección anterior: los delegados nos facilitan el acceso a "punteros" (o direcciones de memoria) de funciones, pero hecho de una forma "controlada", en este caso por el propio .NET framework.

Por ejemplo, el evento **DatosCambiados** definido anteriormente, también lo podemos definir de la siguiente forma:

```
Public Delegate Sub DatosCambiadosEventHandler()  
Public Event DatosCambiados As DatosCambiadosEventHandler
```

Es decir, el método que intercepte este evento debe ser del tipo *Sub* y no recibir ningún parámetro.

Si nuestro evento utiliza, por ejemplo, un parámetro de tipo *String*, la definición del delegado quedaría de la siguiente forma:

```
Public Delegate Sub NombreCambiadoEventHandler(ByVal nuevoNombre As String)
```

Y la definición del evento quedaría de esta otra:

```
Public Event NombreCambiado As NombreCambiadoEventHandler
```

Como vemos al definir el evento ya no tenemos que indicar si recibe o no algún parámetro,

ya que esa definición la hemos hecho en el delegado.

Si nos decidimos a definir este evento de la forma "normal" de Visual Basic, lo haríamos así:

```
Public Event NombreCambiado(ByVal nuevoNombre As String)
```

Como podemos comprobar, Visual Basic nos permite definir los eventos de dos formas distintas: definiendo un delegado y un evento que sea del tipo de ese delegado o definiendo el evento con los argumentos que debemos usar.

Declaremos como daremos los eventos, los podemos seguir usando de la misma forma, tanto para producirlo mediante *RaiseEvent* como para definir el método que reciba ese evento.

Utilizar un delegado para acceder a un método

Ahora veamos brevemente cómo usar los delegados, en este caso sin necesidad de que defina un evento.

Como hemos comentado, un delegado realmente es una clase que puede contener una referencia a un método, además define el prototipo del método que podemos usar como referencia. Sabiendo esto, podemos declarar una variable del tipo del delegado y por medio de esa variable acceder al método que indiquemos, siempre que ese método tenga la misma "firma" que el delegado. Parece complicado ¿verdad? Y no solo lo parece, es que realmente lo es. Comprobaremos esta "complicación" por medio de un ejemplo. En este código, que iremos mostrando poco a poco, vamos a definir un delegado, un método con la misma firma para que podamos usarlo desde una variable definida con el mismo tipo del delegado.

Definimos un delegado de tipo *Sub* que recibe un valor de tipo cadena:

```
Delegate Sub Saludo(ByVal nombre As String)
```

Definimos un método con la misma firma del delegado:

```
Private Sub mostrarSaludo(ByVal elNombre As String)
    Console.WriteLine("Hola, " & elNombre)
End Sub
```

Ahora vamos a declarar una variable para que acceda a ese método.

Para ello debemos declararla con el mismo tipo del delegado:

```
Dim saludando As Saludo
```

La variable *saludando* es del mismo tipo que el delegado **Saludo**. La cuestión es ¿cómo o que asignamos a esta variable?

Primer intento:

Como hemos comentado, los delegados realmente son clases, por tanto podemos usar **New Saludo** y, según parece, deberíamos pasarle un nombre como argumento. Algo así:

```
saludando = New Saludo("Pepe")
```

Pero esto no funciona, entre otras cosas, porque hemos comentado que un delegado contiene (o puede contener) una referencia a un método, y "**Pepe**" no es un método ni una referencia a un método.

Segundo intento:

Por lógica y, sobre todo, por sentido común, máxime cuando hemos declarado un método con la misma "firma" que el delegado, deberíamos pensar que lo que debemos pasar a esa

variable es el método, ya que un delegado puede contener una referencia a un método.

```
saludando = New Saludo(mostrarSaludo)
```

Esto tampoco funciona, ¿seguramente porque le falta el parámetro?

```
saludando = New Saludo(mostrarSaludo("Pepe"))
```

Pues tampoco.

Para usar un delegado debemos indicarle la dirección de memoria de un método, a eso se refiere la definición que vimos antes, una referencia a un método no es ni más ni menos que la dirección de memoria de ese método. Y en Visual Basic, desde la versión 5.0, tenemos una instrucción para obtener la dirección de memoria de cualquier método: *AddressOf*.

Por tanto, para indicarle al delegado dónde está ese método tendremos que usar cualquiera de estas dos formas:

```
saludando = New Saludo(AddressOf mostrarSaludo)
```

Es decir, le pasamos al constructor la dirección de memoria del método que queremos "asociar" al delegado.

En Visual Basic esa misma asignación la podemos simplificar de esta forma:

```
saludando = AddressOf mostrarSaludo
```

Ya que el compilador "sabe" que **saludando** es una variable de tipo delegado y lo que esa variable puede contener es una referencia a un método que tenga la misma firma que la definición del delegado, en nuestro caso, que sea de tipo *Sub* y reciba una cadena.

Si queremos, también podemos declarar la variable y asignarle directamente el método al que hará referencia:

```
Dim saludando As New Saludo(AddressOf mostrarSaludo)
```

Y ahora... ¿cómo podemos usar esa variable?

La variable **saludando** realmente está apuntando a un método y ese método recibe un valor de tipo cadena, por tanto si queremos llamar a ese método (para que se ejecute el código que contiene), tendremos que indicarle el valor del argumento, sabiendo esto, la llamada podría ser de esta forma:

```
saludando("Pepe")
```

Y efectivamente, así se mostraría por la consola el saludo (Hola) y el valor indicado como argumento.

Realmente lo que hacemos con esa llamada es acceder al método al que apunta la variable y como ese método recibe un parámetro, debemos pasárselo, en cuanto lo hacemos, el runtime de .NET se encarga de localizar el método y pasarle el argumento, de forma que se ejecute de la misma forma que si lo llamásemos directamente:

```
mostrarSaludo("Pepe")
```

Con la diferencia de que la variable "**saludando**" no tiene porqué saber a qué método está llamando, y lo más importante, no sabe dónde está definido ese método, solo sabe que el método recibe un parámetro de tipo cadena y aparte de esa información, no tiene porqué

saber nada más.

Así es como funcionan los eventos, un evento solo tiene la dirección de memoria de un método, ese método recibe los mismos parámetros que los definidos por el evento (realmente por el delegado), cuando producimos el evento con *RaiseEvent* es como si llamáramos a cada uno de los métodos que se han ido agregando al delegado, si es que se ha agregado alguno, ya que en caso de que no haya ningún método asociado a ese evento, éste no se producirá, por la sencilla razón de que no habrá ningún código al que llamar.

Realmente es complicado y, salvo que lo necesitemos para casos especiales, no será muy habitual que usemos los delegados de esta forma, aunque no está de más saberlo, sobre todo si de así comprendemos mejor cómo funcionan los eventos.

Lección 4: Eventos y delegados

- Eventos
- Definir y producir eventos en una clase
- Delegados

Definir un evento bien informado

Definir un evento bien informado con Custom Event

Para terminar con esta lección sobre los eventos y los delegados, vamos a ver otra forma de definir un evento. Esta no es exclusiva de Visual Basic, ya que el lenguaje C#, compañero inseparable en los entornos de Visual Studio 2010, también tiene esta característica, pero aunque pueda parecer extraño, es menos potente que la de Visual Basic; por medio de esta declaración, tal como indicamos en el título de la sección, tendremos mayor información sobre cómo se declara el evento, cómo se destruye e incluso cómo se produce, es lo que la documentación de Visual Basic llama evento personalizado (*Custom Event*).

Cuando declaramos un evento usando la instrucción *Custom* estamos definiendo tres bloques de código que nos permite interceptar el momento en que se produce cualquiera de las tres acciones posibles con un evento:

1. Cuando se "liga" el evento con un método, ya sea por medio de *AddHandler* o mediante *Handles*
2. Cuando se desliga el evento de un método, por medio de *RemoveHandler*
3. Cuando se produce el evento, al llamar a *RaiseEvent*

Para declarar este tipo de evento, siempre debemos hacerlo por medio de un delegado.

Veamos un ejemplo de una declaración de un evento usando *Custom Event*:

```
Public Delegate Sub ApellidosCambiadosEventHandler(ByVal nuevo As String)

Public Custom Event ApellidosCambiados As ApellidosCambiadosEventHandler
    AddHandler(ByVal value As ApellidosCambiadosEventHandler)
        ' este bloque se ejecutará cada vez que asignemos un método al evento
    End AddHandler

    RemoveHandler(ByVal value As ApellidosCambiadosEventHandler)
        ' este bloque se ejecutará cuando se desligue el evento de un método
    End RemoveHandler

    RaiseEvent(ByVal nuevo As String)
        ' este bloque se ejecutará cada vez que se produzca el evento
    End RaiseEvent
End Event
```

Como podemos apreciar, debemos definir un delegado con la "firma" del método a usar con el evento.

Después definimos el evento por medio de las instrucciones *Custom Event*, utilizando el mismo formato que al definir un evento con un delegado.

Dentro de la definición tenemos tres bloques, cada uno de los cuales realizará la acción que ya hemos indicado en la lista numerada.

Nota:

Los eventos Custom Event solamente podemos definirlos e interceptarlos en el mismo ensamblado.

Lección 5: Atributos

- **Atributos**

Introducción

Esta es la definición que nos da la ayuda de Visual Basic sobre lo que es un atributo.

Los atributos son etiquetas descriptivas que proporcionan información adicional sobre elementos de programación como tipos, campos, métodos y propiedades. Otras aplicaciones, como el compilador de Visual Basic, pueden hacer referencia a la información adicional en atributos para determinar cómo pueden utilizarse estos elementos.

En esta lección veremos algunos ejemplos de cómo usarlos en nuestras propias aplicaciones y, aunque sea de forma general, cómo usar y aplicar algunos de los atributos definidos en el propio .NET Framework, al menos los que más directamente nos pueden interesar a los desarrolladores de Visual Basic.

Atributos

- **Atributos**

- Atributos para representar información de nuestra aplicación
 - Mostrar los ficheros ocultos del proyecto
- Tipos de atributos que podemos usar en una aplicación
 - Atributos globales a la aplicación
 - Atributos particulares a las clases o miembros de las clases
- Atributos personalizados
 - Acceder a los atributos personalizados en tiempo de ejecución
- Atributos específicos de Visual Basic
- Marcar ciertos miembros de una clase como obsoletos

Lección 5: Atributos

Atributos

Atributos

Como hemos comentado en la introducción, los atributos son etiquetas que podemos aplicar a nuestro código para que el compilador y, por extensión, el propio .NET Framework los pueda usar para realizar ciertas tareas o para obtener información extra sobre nuestro código.

De hecho en cualquier aplicación que creemos con Visual Basic 2010 estaremos tratando con atributos, aunque nosotros ni nos enteremos, ya que el propio compilador los utiliza para generar los metadatos del ensamblado, es decir, la información sobre todo lo que contiene el ejecutable o librería que hemos creado con Visual Basic 2010.

Por otra parte, el uso de los atributos nos sirve para ofrecer cierta funcionalidad extra a nuestro código, por ejemplo, cuando creamos nuestros propios controles, mediante atributos podemos indicarle al diseñador de formularios si debe mostrar ciertos miembros del control en la ventana de propiedades, etc.

Atributos para representar información de nuestra aplicación

De forma más genérica podemos usar los atributos para indicar ciertas características de nuestra aplicación, por ejemplo, el título, la versión, etc. Todos estos atributos los indicaremos como "características" de nuestra aplicación, y lo haremos sin ser demasiados conscientes de que realmente estamos usando atributos, ya que el propio Visual Basic los controla mediante propiedades de la aplicación.

Por ejemplo, si mostramos la ventana de propiedades de nuestro proyecto, ver figura 2.24:

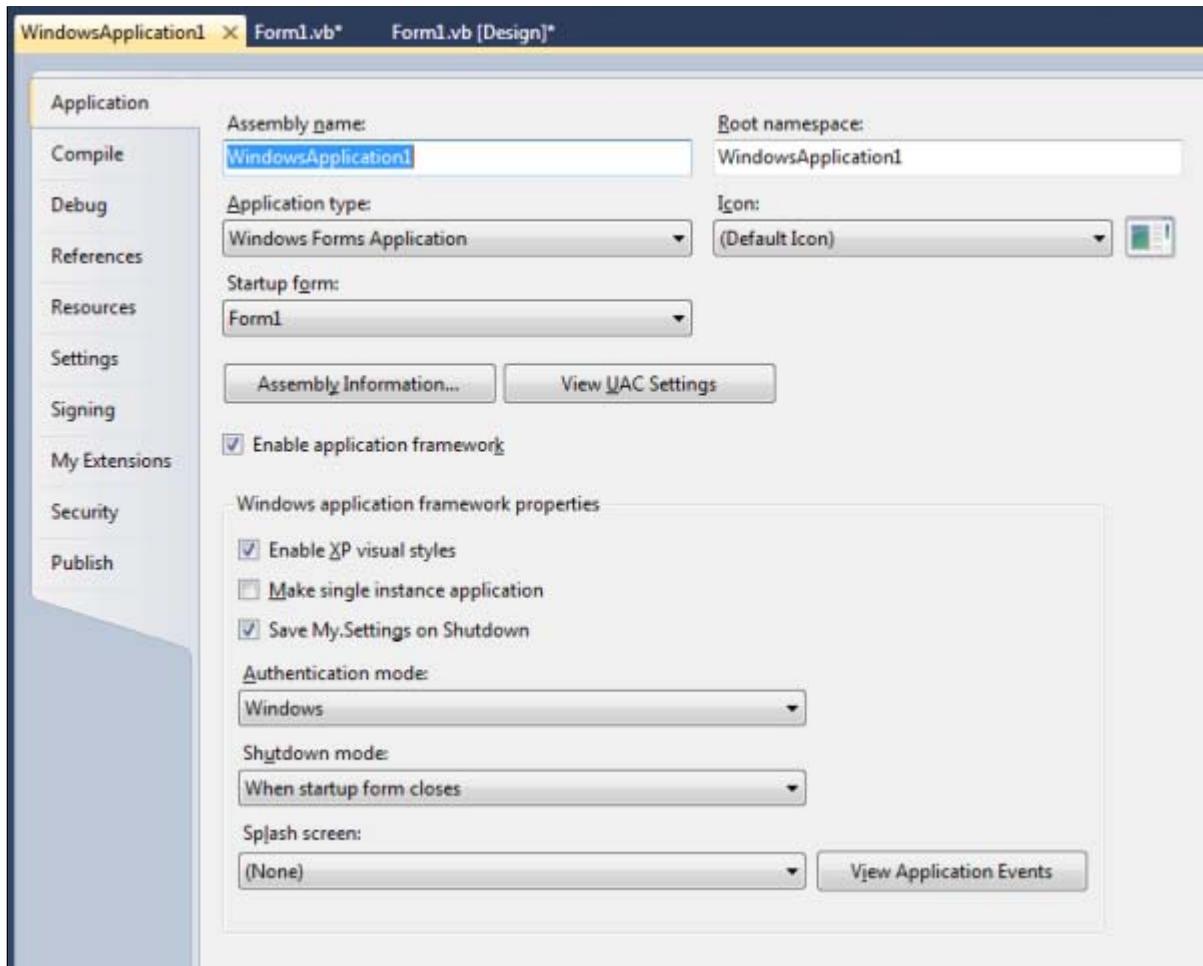


Figura 2.24. Propiedades de la aplicación

Tendremos acceso a las propiedades de la aplicación, como el nombre del ensamblado, el espacio de nombres, etc. Si queremos agregar información extra, como la versión, el copyright, etc. podemos presionar el botón "Assembly Information", al hacerlo, se mostrará una nueva ventana en la que podemos escribir esa información, tal como mostramos en la figura 2.25:

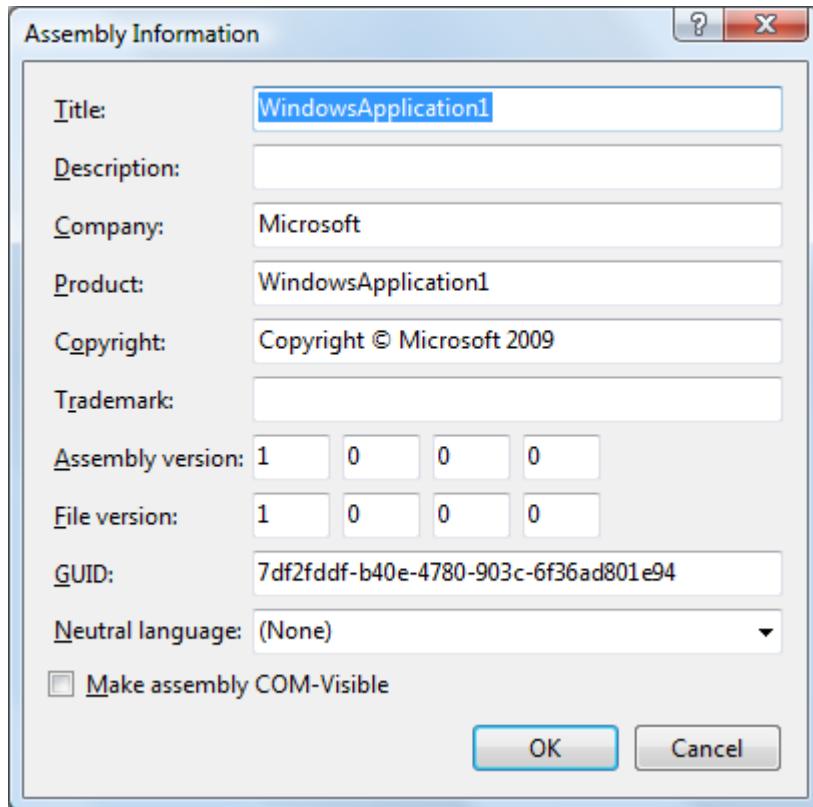
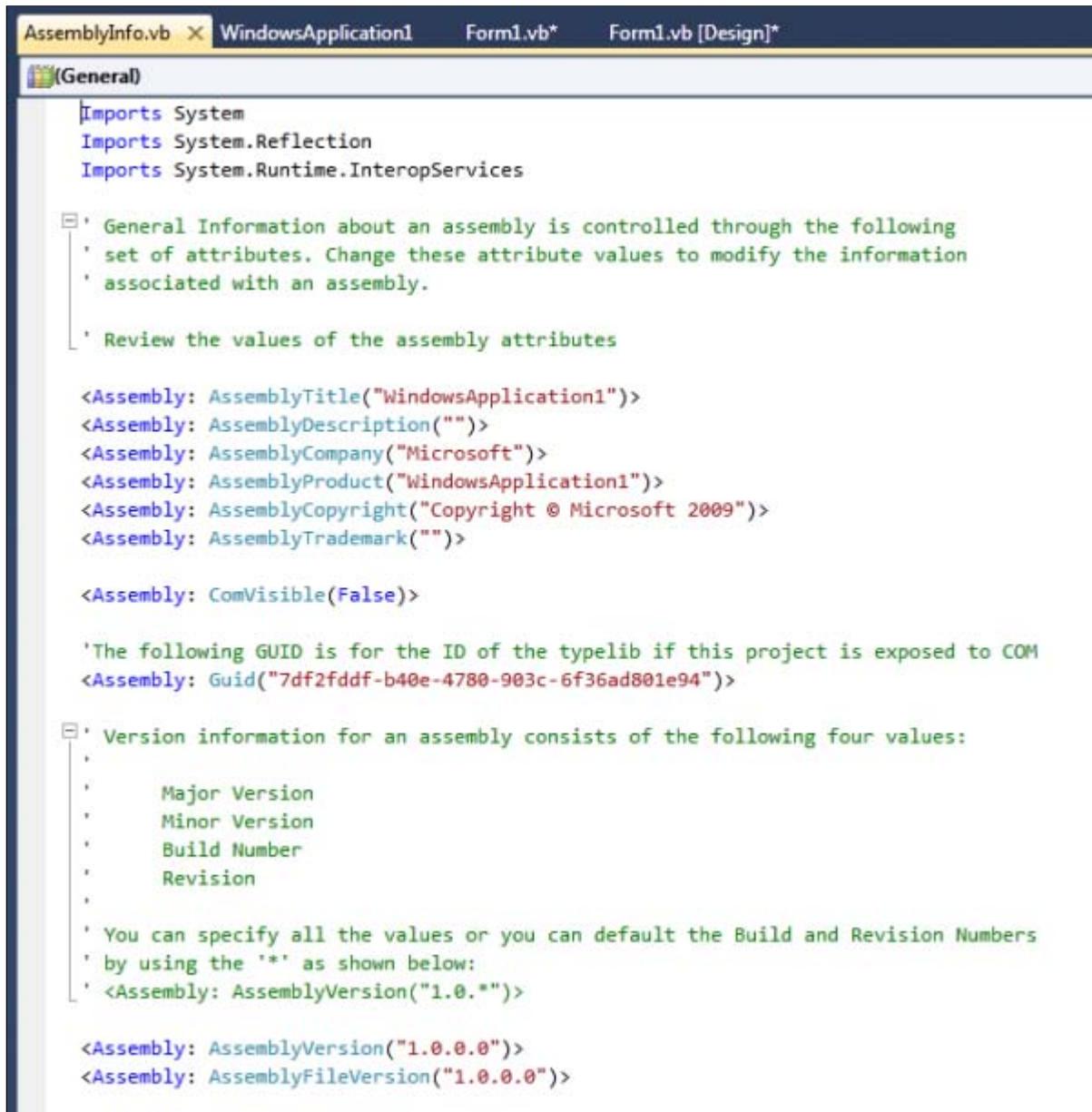


Figura 2.25. Información del ensamblado

Esa información realmente está definida en un fichero del proyecto llamado AssembluInfo.vb, el cual de forma predeterminada está oculto, si lo mostramos, veremos que esa información la contiene en formato de atributos.

Parte del código de ese fichero lo podemos ver en la figura 2.26:



```
AssemblyInfo.vb X WindowsApplication1 Form1.vb* Form1.vb [Design]*

(General)

Imports System
Imports System.Reflection
Imports System.Runtime.InteropServices

' General Information about an assembly is controlled through the following
' set of attributes. Change these attribute values to modify the information
' associated with an assembly.

' Review the values of the assembly attributes

<Assembly: AssemblyTitle("WindowsApplication1")>
<Assembly: AssemblyDescription("")>
<Assembly: AssemblyCompany("Microsoft")>
<Assembly: AssemblyProduct("WindowsApplication1")>
<Assembly: AssemblyCopyright("Copyright © Microsoft 2009")>
<Assembly: AssemblyTrademark("")>

<Assembly: ComVisible(False)>

'The following GUID is for the ID of the typelib if this project is exposed to COM
<Assembly: Guid("7df2fddf-b40e-4780-903c-6f36ad801e94")>

' Version information for an assembly consists of the following four values:
'
'     Major Version
'     Minor Version
'     Build Number
'     Revision
'
' You can specify all the values or you can default the Build and Revision Numbers
' by using the '*' as shown below:
' <Assembly: AssemblyVersion("1.0.*")>

<Assembly: AssemblyVersion("1.0.0.0")>
<Assembly: AssemblyFileVersion("1.0.0.0")>
```

Figura 2.26. Contenido del fichero AssemblyInfo

En este código podemos resaltar tres cosas:

La primera es que tenemos una importación al espacio de nombres **System.Reflection**, este espacio de nombres contiene la definición de las clases/atributos utilizados para indicar los atributos de la aplicación, como el título, etc.

La segunda es la forma de usar los atributos, estos deben ir encerrados entre signos de menor y mayor: **<Assembly: ComVisible(False)>**.

La tercera es que, en este caso, los atributos están definidos a nivel de ensamblado, para ellos se añade la instrucción **Assembly**: al atributo.

Como veremos a continuación, los atributos también pueden definirse a nivel local, es decir, solo aplicable al elemento en el que se utiliza, por ejemplo, una clase o un método, etc.

Mostrar los ficheros ocultos del proyecto

Como acabamos de comentar, el fichero AssemblyInfo.vb que es el que contiene la información sobre la aplicación (o ensamblado), está oculto. Para mostrar los ficheros ocultos, debemos hacer lo siguiente:

En la ventana del explorador de soluciones, presionamos el segundo botón, (si pasamos el

cursor por encima, mostrará un mensaje que indica "Mostrar todos los ficheros"), de esta forma tendremos a la vista todos los ficheros de la aplicación, incluso el de los directorios en el que se crea el ejecutable, tal como podemos apreciar en la figura 2.27:

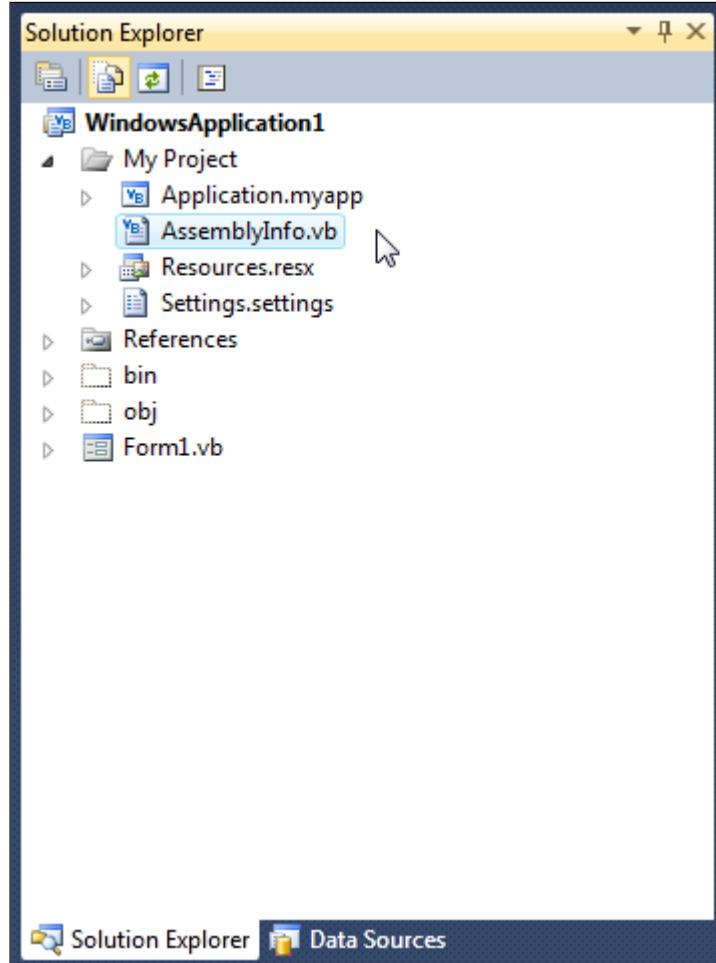


Figura 2.27. Mostrar todos los ficheros de la solución

Tipos de atributos que podemos usar en una aplicación

Como hemos comentado, existen atributos que son globales a toda la aplicación y otros que podremos aplicar a elementos particulares, como una clase o un método.

Atributos globales a la aplicación

Estos se indican usando *Assembly*: en el atributo y los podremos usar en cualquier parte de nuestro código, aunque lo habitual es usarlos en el fichero AssemblyInfo.vb.

Nota:

La palabra o instrucción *Assembly*: lo que indica es que el atributo tiene un ámbito de ensamblado.

Atributos particulares a las clases o miembros de las clases

Estos atributos solo se aplican a la clase o al miembro de la clase que creamos conveniente, el formato es parecido a los atributos globales, ya que se utilizan los signos de menor y mayor para encerrarlo, con la diferencia de que en este tipo de atributos no debemos usar *Assembly*: ya que esta instrucción indica que el atributo es a nivel del ensamblado.

Cuando aplicamos un atributo "particular", este debe estar en la misma línea del elemento

al que se aplica, aunque si queremos darle mayor legibilidad al código podemos usar un guión bajo para que el código continúe en otra línea:

```
<Microsoft.VisualBasic.HideModuleName()> _
Module MyResources
```

Atributos personalizados

Además de los atributos que ya están predefinidos en el propio .NET o Visual Basic, podemos crear nuestros propios atributos, de forma que en tiempo de ejecución podamos acceder a ellos mediante las clases del espacio de nombres *Reflection*, aunque debido a que este tema se sale un poco de la intención de este curso, simplemente indicar que los atributos personalizados son clases que se derivan de la clase *System.Attribute* y que podemos definir las propiedades que creamos conveniente utilizar en ese atributo para indicar cierta información a la que podemos acceder en tiempo de ejecución.

En el siguiente código tenemos la declaración de una clase que se utilizará como atributo personalizado, notamos que, por definición las clases para usarlas como atributos deben terminar con la palabra *Attribute* después del nombre "real" de la clase, que como veremos en el código que utiliza ese atributo, esa "extensión" al nombre de la clase no se utiliza.

Veamos primero el código del atributo personalizado:

```
<AttributeUsage(AttributeTargets.All)> _
Public Class AutorAttribute
    Inherits System.Attribute
    '
    Private _ModificadoPor As String
    Private _Version As String
    Private _Fecha As String
    '
    Public Property ModificadoPor() As String
        Get
            Return _ModificadoPor
        End Get
        Set(ByVal value As String)
            _ModificadoPor = value
        End Set
    End Property
    '
    Public Property Version() As String
        Get
            Return _Version
        End Get
        Set(ByVal value As String)
            _Version = value
        End Set
    End Property
    '
    Public Property Fecha() As String
        Get
            Return _Fecha
        End Get
        Set(ByVal value As String)
            _Fecha = value
        End Set
    End Property
End Class
```

Para usar este atributo lo podemos hacer de la siguiente forma:

```
<Autor(ModificadoPor:="Guillermo 'guille'", _
        Version:="1.0.0.0", Fecha:="13/Abr/2005")> _
Public Class PruebaAtributos
```

Nota:

Cuando utilizamos el atributo, en el constructor que de forma predeterminada crea Visual Basic, los parámetros se indican por el orden alfabético de las propiedades, pero que nosotros podemos alterar usando directamente los nombres de las propiedades, tal como podemos ver en el código de ejemplo anterior.

Acceder a los atributos personalizados en tiempo de ejecución

Para acceder a los atributos personalizados podemos hacer algo como esto, (suponiendo que tenemos la clase **AutorAttribute** y una clase llamada **PruebaAtributos** a la que hemos aplicado ese atributo personalizado):

```
Sub Main()
    Dim tipo As Type
    tipo = GetType(PruebaAtributos)

    Dim atributos() As Attribute
    atributos = Attribute.GetCustomAttributes(tipo)

    For Each atr As Attribute In atributos
        If TypeOf atr Is AutorAttribute Then
            Dim aut As AutorAttribute
            aut = CType(atr, AutorAttribute)
            Console.WriteLine("Modificado por: " & aut.ModificadoPor)
            Console.WriteLine("Fecha: " & aut.Fecha)
            Console.WriteLine("Versión: " & aut.Version)
        End If
    Next

End Sub
```

Atributos específicos de Visual Basic

Visual Basic utiliza una serie de atributos para indicar ciertas características de nuestro código, en particular son tres:

- **COMClassAttribute**
 - Este atributo se utiliza para simplificar la creación de componentes COM desde Visual Basic.
- **VBFixedStringAttribute**
 - Este atributo se utiliza para crear cadenas de longitud fija en Visual Basic 2010. Habitualmente se aplica a campos o miembros de una estructura, principalmente cuando queremos acceder al API de Windows o cuando queremos usar esa estructura para guardar información en un fichero, pero utilizando una cantidad fija de caracteres.
- **VBFixedArrayAttribute**
 - Este atributo, al igual que el anterior, lo podremos usar para declarar arrays de tamaño fijo, al menos si las declaramos en una estructura, ya que por defecto, los arrays de Visual Basic son de tamaño variable.

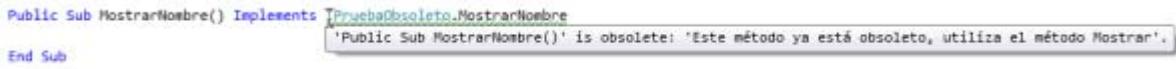
Marcar ciertos miembros de una clase como obsoletos

En ocasiones nos encontraremos que escribimos cierto código que posteriormente no queremos que se utilice, por ejemplo porque hemos creado una versión optimizada. Si ese código lo hemos declarado en una interfaz, no deberíamos eliminarlo de ella, ya que así romperíamos el contrato contraído por las clases que implementan esa interfaz. En estos casos nos puede venir muy bien el uso del atributo `<Obsolete>`, ya que así podemos informar al usuario de que ese atributo no debería usarlo. En el constructor de este atributo podemos indicar la cadena que se mostrará al usuario. En el siguiente código se declara un

método con el atributo *Obsolete*:

```
Public Interface IPruebaObsoleto
    <Obsolete("Este método ya está obsoleto, utiliza el método Mostrar")> _
        Sub MostrarNombre()
        Sub Mostrar()
    End Interface
```

Si trabajamos con el IDE de Visual Basic, ese mensaje se mostrará al compilar o utilizar los atributos marcados como obsoletos, tal como podemos apreciar en la figura 2.28:



A screenshot of the Visual Studio IDE showing a tooltip for an obsolete method. The tooltip contains the message: "'Public Sub MostrarNombre()' is obsolete: 'Este método ya está obsoleto, utiliza el método Mostrar'".

Figura 2.28. Mensaje de que un método está obsoleto

Módulo 3: Desarrollo de aplicaciones Windows

- Uso del diseñador de Visual Studio
- Controles de Windows Forms
- Trabajo con controles
- Trabajo con imágenes y gráficos
- Despliegue de aplicaciones

Módulo 3 - Desarrollo de aplicaciones Windows

En este módulo, conoceremos las partes generales y más importantes del entorno de desarrollo rápido Visual Studio 2010 para la programación de aplicaciones con este lenguaje de la familia .NET.

Veremos las partes principales del entorno, también veremos como desarrollar nuestros propios controles Windows, aprenderemos a trabajar con imágenes y gráficos con Visual Basic y finalmente, conoceremos como desplegar nuestras aplicaciones desarrolladas en Visual Basic 2010.

FAQ:

[¿Qué tipo de aplicaciones puedo desarrollar si estoy utilizando Visual Basic 2010 Express?](#)

Podrá desarrollar principalmente, **Aplicaciones para Windows, Bibliotecas de Clases y Aplicaciones de Consola**. Además, podrá añadir sus propias plantillas para obtener un mayor rendimiento en el desarrollo de aplicaciones.

Si su interés es el desarrollo de aplicaciones Web, puede utilizar **Visual Web Developer 2010 Express**.

► [Productos Visual Studio Express](#)

Las partes que forman parte de este módulo son las siguientes:

Capítulo 1

- [Uso del diseñador de Visual Studio](#)

Capítulo 2

- [Controles de Windows Forms](#)

Capítulo 3

- [Desarrollo de controles](#)

Capítulo 4

- [Trabajo con imágenes y gráficos](#)

Capítulo 5

- [Despliegue de aplicaciones](#)



Lección 1: Diseñador de Visual Studio

- Cuadro de herramientas
- Explorador de base de datos
- Explorador de soluciones
- Propiedades
- Menús y barras de botones
- Otras consideraciones

Introducción

Cuando nos encontramos con Visual Studio 2010 por primera vez, saltan a la vista, algunos de los cambios más importantes de este novedoso entorno de desarrollo de aplicaciones Windows.

Para un desarrollador, familiarizarse con el entorno de Visual Studio 2010 es una tarea que no debe entrañar una complejidad excesivamente grande. Como nos ocurre a todos los que nos encontramos delante de un nuevo entorno de trabajo, lo único que se requiere es constancia y práctica, mucha práctica. Sin embargo, si usted es ya un desarrollador habitual de otros entornos de desarrollo, notará que sus avances van a ser significativos en muy poco tiempo.

Sobre Visual Studio 2010, cabe destacar que el entorno de desarrollo ha sido reprogramado por completo utilizando las bondades de WPF (Windows Presentation Foundation).

El entorno en sí no ha sufrido grandes cambios salvo las posibilidades de WPF que son mayores de cara al usuario, aunque sus funcionalidades siguen siendo las mismas.

Nota:

Si está utilizando Visual Basic 2010 Express para seguir este curso debe saber que este entorno está especializado en desarrollar aplicaciones Windows con Visual Basic 2010, aunque podrá usar controles y librerías escritas en otros lenguajes de la plataforma .NET .

Módulo 3 - Capítulo 1

- 1. [Cuadro de herramientas](#)
- 2. [Explorador de base de datos](#)
- 3. [Explorador de soluciones](#)
- 4. [Propiedades](#)
- 5. [Menus y barra de botones](#)
- 6. [Otras consideraciones](#)

Lección 1: Diseñador de Visual Studio

Cuadro de herramientas

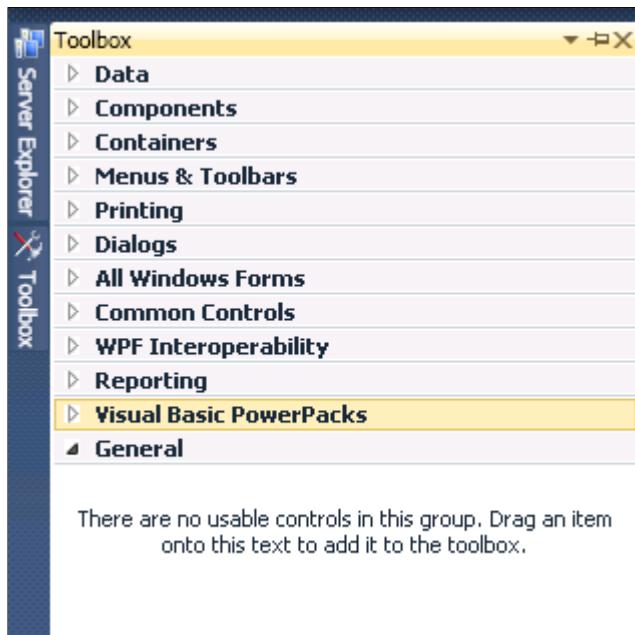
- Explorador de base de datos
- Explorador de soluciones
- Propiedades
- Menús y barras de botones
- Otras consideraciones

Módulo 3 - Capítulo 1

1. Cuadro de herramientas

El cuadro o barra de herramientas de Visual Studio 2010, nos permite utilizar los distintos componentes que .NET Framework pone a nuestra disposición. Dentro de Visual Studio 2010 tenemos una gran cantidad de controles dispuestos en diferentes categorías.

En la figura 1 podemos ver la barra de herramientas de Visual Studio 2010.

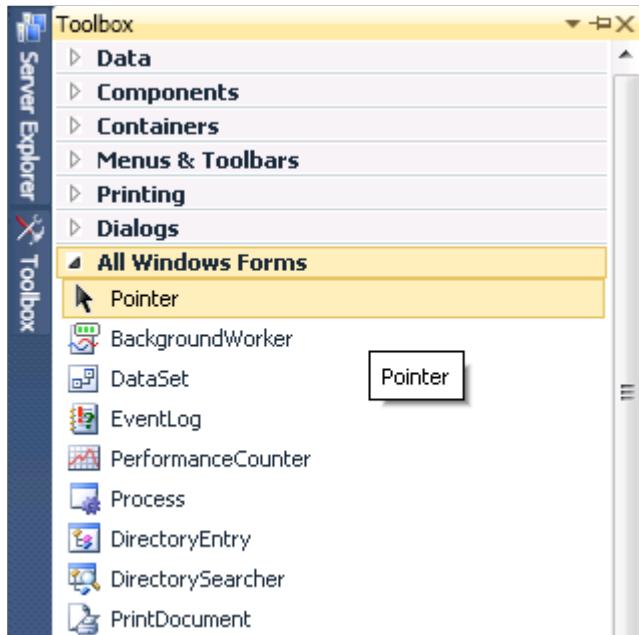


Visual Studio 2010

Figura 1

El *Cuadro de herramientas*, lo localizará en la parte izquierda del entorno *Visual Studio 2010*.

Cuando iniciamos un nuevo proyecto con Visual Studio 2010, el cuadro de herramientas queda relleno con los controles que podemos utilizar en el proyecto. Si abrimos un formulario Windows, los controles quedan habilitados para que los podamos insertar en el formulario Windows. En la figura 2 se muestra la barra de herramientas con los controles preparados para ser insertados en el formulario Windows.



Toolbox de Visual Studio 2010 con controles Windows preparados para ser insertados en el formulario Windows

Figura 2

Nota:

Para insertar un control en un formulario Windows, se requiere que el formulario Windows sobre el que deseamos insertar un control, esté abierto. Una vez que está abierto, bastará con realizar una de las tres siguientes acciones para insertar un control al formulario:

- *Hacer doble clic sobre un control del cuadro de herramientas*
- *Hacer clic sobre un control del cuadro de herramientas, y sin soltar el botón del mouse, arrastrarlo sobre el formulario*
- *Hacer clic sobre un control del cuadro de herramientas, y luego hacer clic sobre el formulario y arrastrar para marcar una zona que cubrirá nuestro control y soltar el ratón*

El control quedará entonces insertado dentro del formulario.

Lección 1: Diseñador de Visual Studio

- Cuadro de herramientas
- Explorador de base de datos
- Explorador de soluciones
- Propiedades
- Menús y barras de botones
- Otras consideraciones

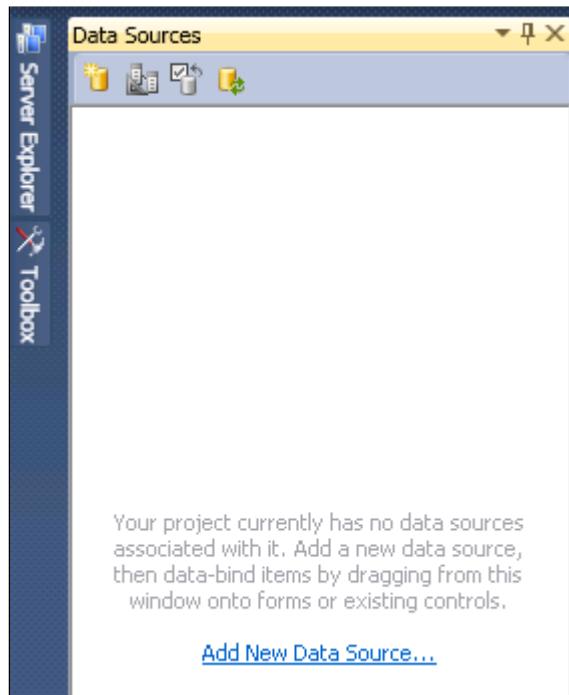
Módulo 3 - Capítulo 1

2. Explorador de base de datos

Si ha sido lo suficientemente observador cuando se explicaban los detalles del cuadro o barra de herramientas, y ha prestado especial atención a las figuras o a las ventanas del entorno de desarrollo de Visual Studio 2010, quizás haya notado que en la parte izquierda además de la solapa *cuadro de herramientas*, aparece otra solapa de nombre *explorador de base de datos*.

Desde esta solapa, un programador puede acceder a diferentes recursos del sistema. El principal y más importante recurso, es el que tiene que ver con las conexiones con bases de datos, ya sean Microsoft Access, Microsoft SQL Server o cualquier otra fuente de datos.

En la figura 1 puede observar la solapa *Explorador de base de datos* extendida con parte de sus opciones.



La solapa del Explorador de base de datos desplegada de Visual Studio 2010

Figura 1

Conectando con una base de datos Microsoft Access a través de SQL Server

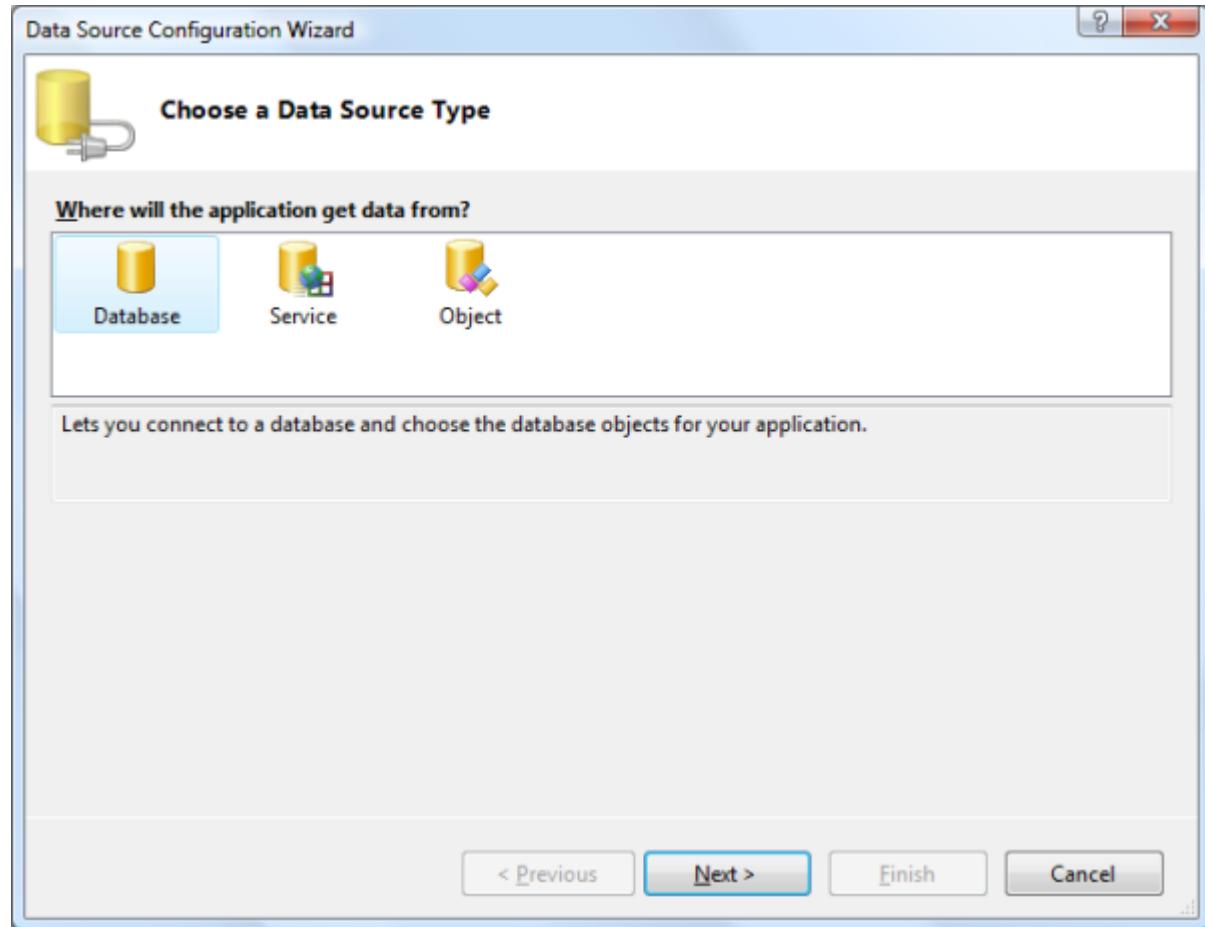
Para muestra un botón, y dado el carácter práctico de este tutorial, aprenderá a crear una conexión con cualquier base de datos, en nuestro caso de ejemplo una base de datos Microsoft SQL Server, para poder utilizarla fácilmente en nuestra aplicación Windows.

Haga clic sobre el botón representado por la siguiente imagen . En este instante, se abrirá una nueva ventana como la que se muestra en la figura 2.

De esta manera se iniciará un asistente para configurar el acceso a datos.

La primera pantalla del asistente nos permite seleccionar la fuente de datos, ya sea desde un objeto de datos, desde un servicio o desde una fuente de datos determinada, que será lo más habitual.

En nuestro caso, seleccionaremos esta opción.



La siguiente ventana del asistente nos indica si queremos seleccionar los datos desde un DataSet o desde un modelo de entidad relación, algo que se verá más adelante.

Vamos a presuponer que obtenemos los datos desde un DataSet.



Choose a Database Model

What type of database model do you want to use?



Dataset



Entity Data Model

The database model you choose determines the types of data objects your application code uses. A dataset file will be added to your project.

[< Previous](#)[Next >](#)[Finish](#)[Cancel](#)

Si continuamos con el asistente, éste nos indica en la siguiente ventana si queremos seleccionar una conexión existente en el proyecto, o si queremos crear una nueva conexión.



Choose Your Data Connection

Which data connection should your application use to connect to the database?

New Connection...

This connection string appears to contain sensitive data (for example, a password), which is required to connect to the database. However, storing sensitive data in the connection string can be a security risk. Do you want to include this sensitive data in the connection string?

- No, exclude sensitive data from the connection string. I will set this information in my application code.
- Yes, include sensitive data in the connection string.

+ Connection string

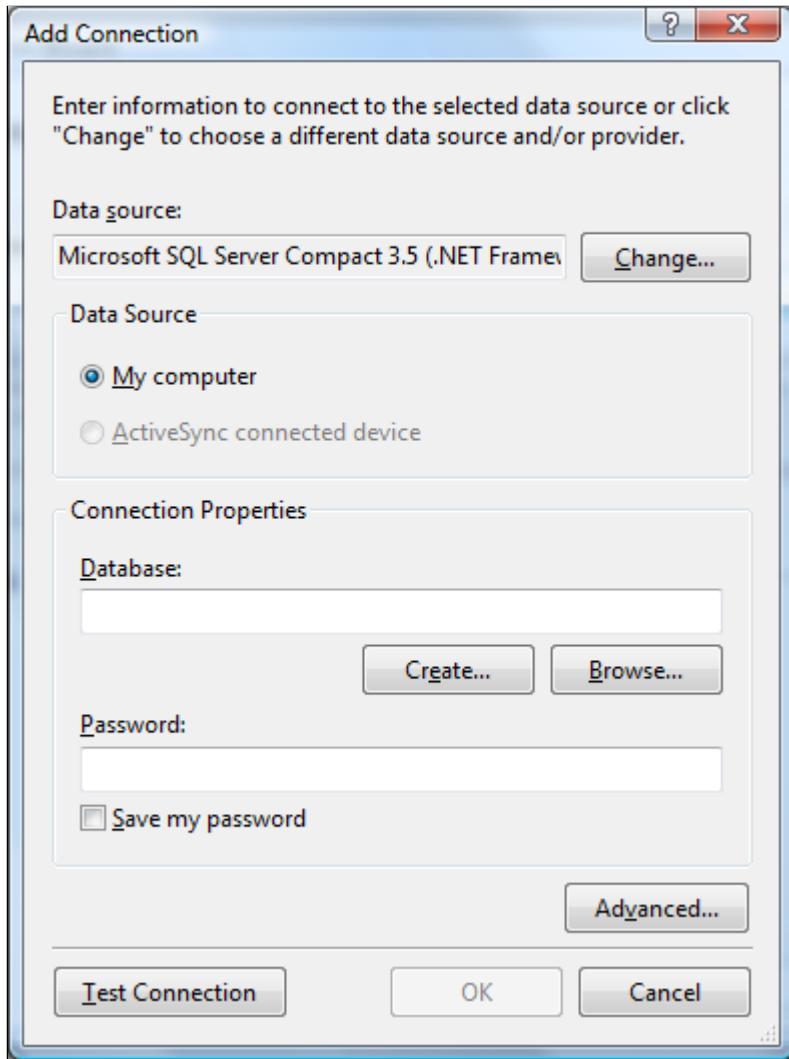
< Previous

Next >

Finish

Cancel

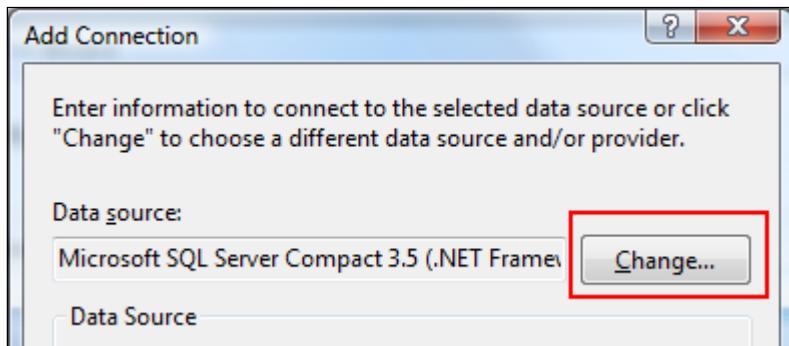
Si pulsamos sobre el botón de nueva conexión, accederemos a una ventana nueva para crear una nueva conexión.



Ventana Agregar conexión, para establecer las propiedades de conexión con una fuente de datos

Figura 2

Por defecto, la ventana **Agregar conexión** no está preparada para establecer una conexión con una fuente de datos de origen de datos *OLE DB* o con *SQL Server*, por lo que si nuestra intención es establecer una conexión con esta fuente de datos, entonces deberemos hacer clic sobre el botón **Cambiar...** que se indica en la figura 3.

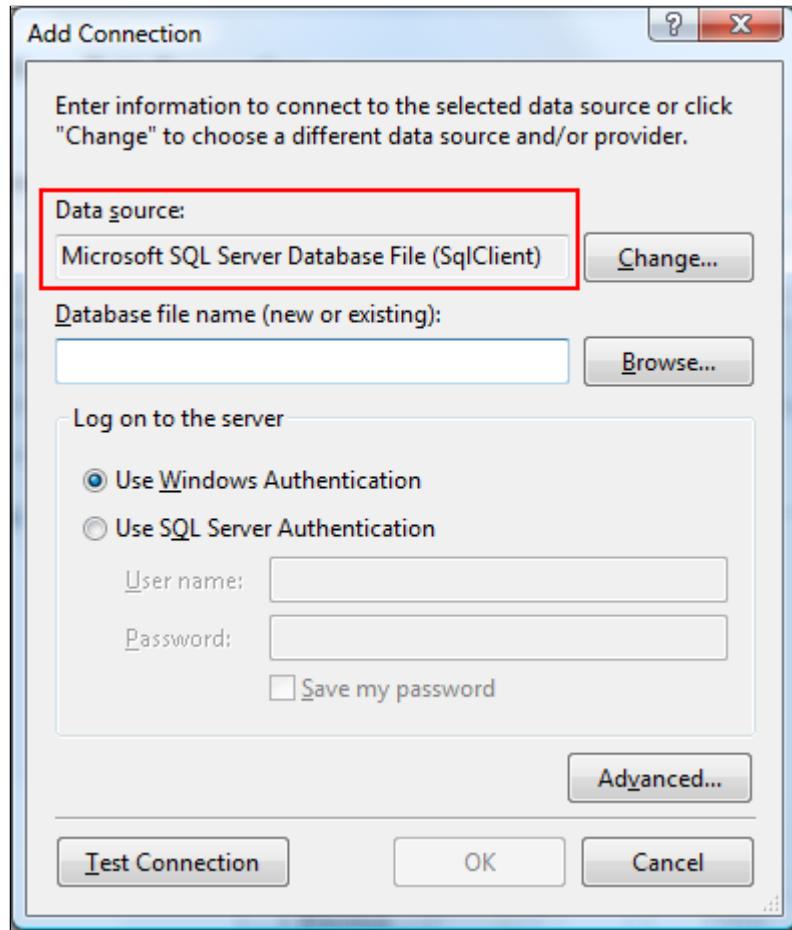


Botón Cambiar... para seleccionar otro proveedor de origen de datos

Figura 3

De esta manera, podemos indicar el origen de acceso a datos que necesitamos para establecer la conexión con nuestra fuente de datos, y que en nuestro ejemplo será un proveedor de *SQL Server*.

Una vez que hemos hecho clic sobre el botón **Cambiar...**, nos aseguramos por lo tanto, que nuestro origen de datos es *base de datos de Microsoft SQL Server Database File (SqlClient)*, como se indica en la figura 4.



Ventana de selección del proveedor u origen de datos

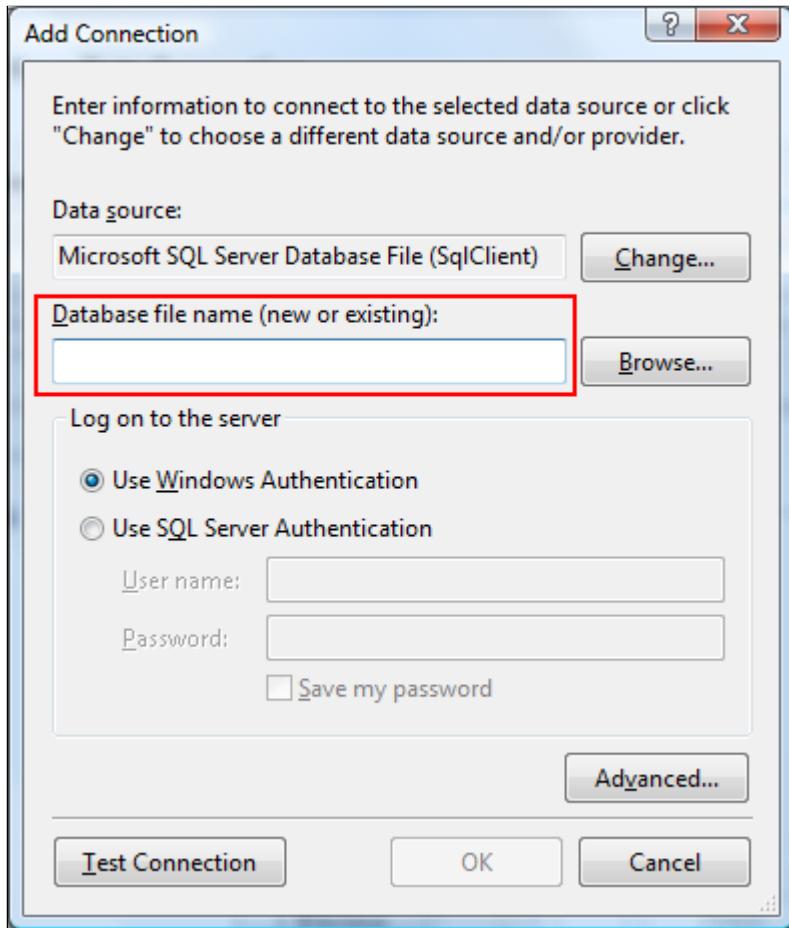
Figura 4

FAQ:

¿Puedo utilizar el proveedor OLE DB en lugar del proveedor de SQL Server para conectar con una base de datos SQL Server?

Con OLE DB, puede acceder a fuentes de datos SQL Server u otras fuentes de datos como Microsoft Access, sin embargo, si utiliza SQL Server 7.0, SQL Server 2000, SQL Server 2005 ó superior, se recomienda el uso del proveedor de SQL Server, que es un proveedor de acceso a datos nativo que aumenta el rendimiento de nuestras aplicaciones con SQL Server. Sólo si utiliza una versión de SQL Server anterior a SQL Server 7.0, deberá utilizar necesariamente el proveedor de acceso a datos OLE DB.

Una vez que hemos seleccionado el proveedor de acceso a datos, nos centraremos en la opción **Nombre de archivo base de datos** como se muestra en la figura 5.

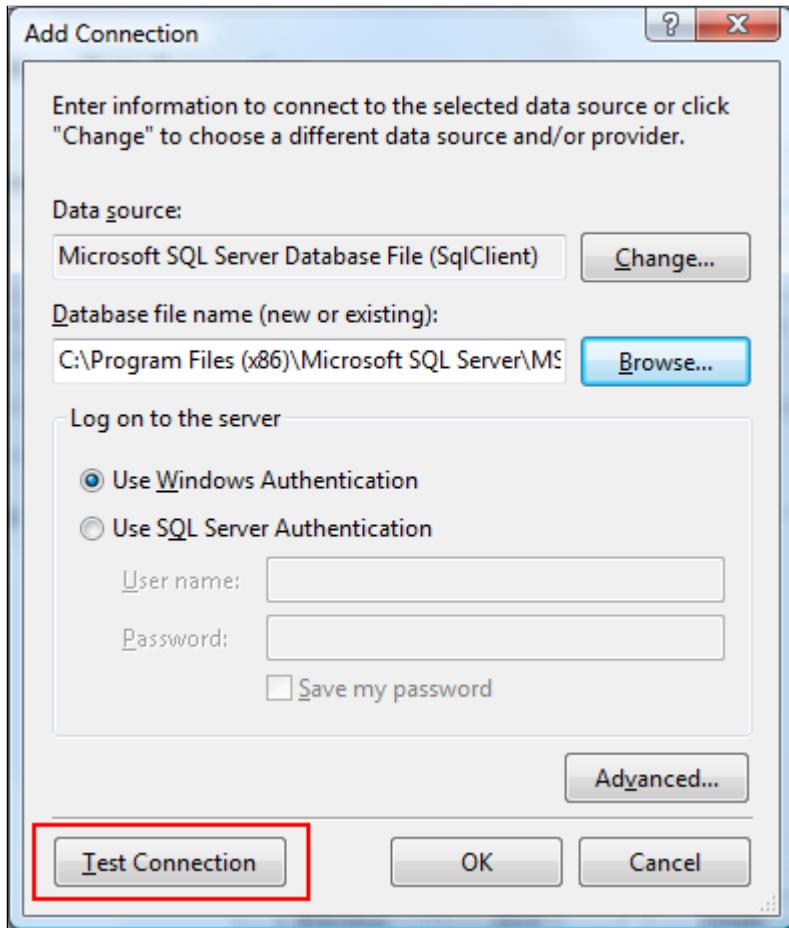


En este lugar indicaremos el fichero de base de datos con el que estableceremos la conexión

Figura 5

Para agregar el fichero de base de datos a la conexión, presionaremos el botón **Examinar...** y seleccionaremos el fichero de base de datos de nuestro disco duro.

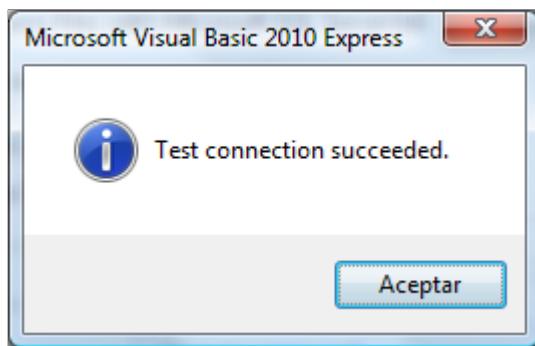
De esta manera, la base de datos quedará indicada en la conexión y tan sólo deberemos probar nuestra conexión pulsando el botón **Probar conexión** como se indica en la figura 6.



Es recomendable probar la conexión antes de agregarla al entorno

Figura 6

Si la prueba de conexión se ha realizado satisfactoriamente, recibiremos un mensaje en pantalla afirmativo como el que se indica en la figura 7.



Prueba de la conexión realizada con éxito

Figura 7

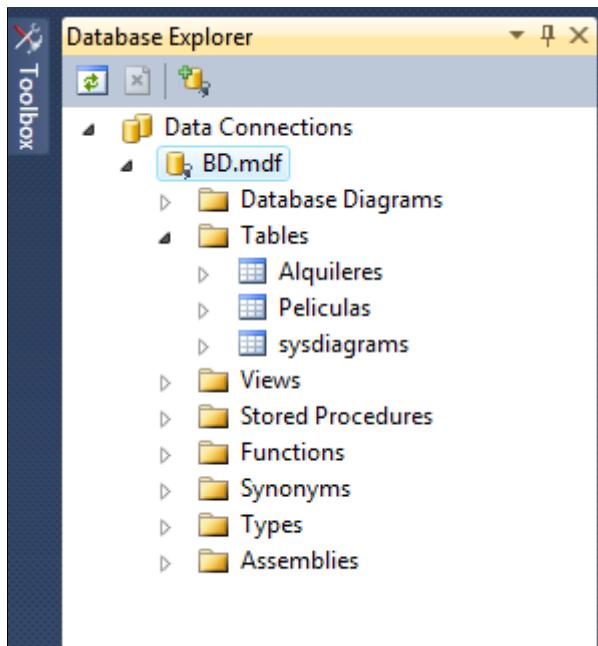
A tener en cuenta:

En este ejemplo, la conexión con la base de datos SQL Server no tiene ningún tipo de usuario y contraseña. Tenga en cuenta que en la parte identificada como **Conexión con la base de datos**, podríamos indicar el usuario y contraseña si fuera necesario.

En este punto, el asistente podría pedirnos las tablas, vistas y objetos de la fuente de datos para que los seleccionaríamos según nuestras necesidades.

Cuando terminemos la selección de todas las opciones, deberemos presionar sobre el botón **Aceptar** para que la base de datos con la que hemos establecido la conexión, quede ahora insertada en la ventana del

Explorador de base de datos como se muestra en la figura 8.



Base de datos Microsoft Access insertada en la ventana del Explorador de base de datos

Figura 8

¡Advertencia!

Tener demasiadas conexiones activas en el entorno o configuradas en él, puede incidir negativamente en el rendimiento de Visual Studio 2010 cuando se trabaja con él. Tenga configuradas solamente, aquellas conexiones que va a utilizar, o aquellas conexiones de uso más habitual y frecuente.

Lección 1: Diseñador de Visual Studio

- Cuadro de herramientas
- Explorador de base de datos
- Explorador de soluciones**
- Propiedades
- Menús y barras de botones
- Otras consideraciones

Módulo 3 - Capítulo 1

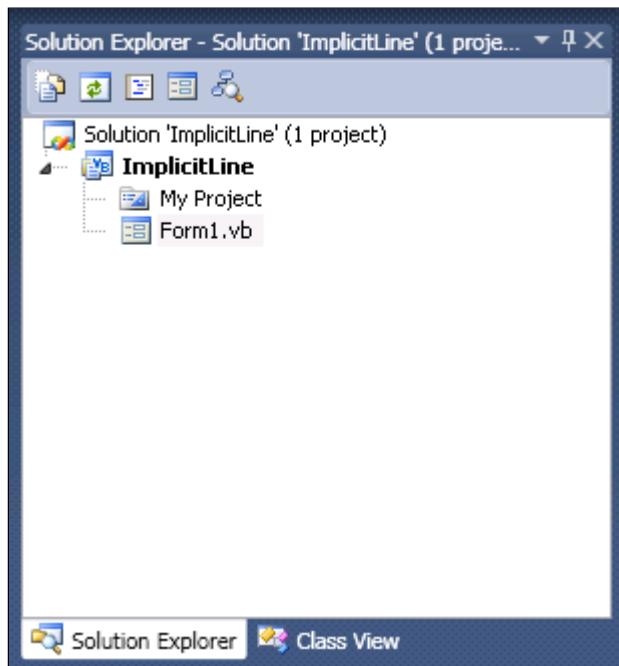
3. Explorador de soluciones

El *Explorador de soluciones* lo podemos encontrar en la parte derecha de nuestro entorno de desarrollo.

Una solución se compone de proyectos y éstos, de recursos y objetos. Por lo general, una solución contendrá un proyecto, pero podemos encontrarnos con más de un proyecto dentro de una misma solución.

Sin embargo, estos conceptos son muy sencillos de comprender y controlar, y para nada debe hacernos pensar que esto es algo complejo que nos costará mucho tiempo dominar.

En la figura 1, podemos observar el explorador de soluciones de *Visual Studio 2010*.



La opción Explorador de soluciones desplegada en Visual Studio 2010

Figura 1

Si queremos añadir un nuevo formulario al proyecto, lo haremos presionando con el botón secundario en cualquier parte de la ventana del explorador de soluciones, pero si esa pulsación la hacemos en alguno de los objetos que contiene el proyecto, no podremos hacerlo, ya que el IDE de Visual Studio 2010 muestra un

menú diferente según el objeto presionado, por ejemplo si queremos añadir un nuevo proyecto, podemos hacerlo presionando con el botón secundario del mouse sobre la "solución".

Nota:

Para abrir un recurso de la solución, basta con situarnos en el recurso determinado, por ejemplo un formulario Windows de nombre Form1.vb y hacer doble clic sobre él. El recurso se abrirá automáticamente en Visual Studio 2010.

Además, en Visual Studio 2010 sabremos en todo momento sobre qué recurso estamos trabajando en un momento dado.

Lección 1: Diseñador de Visual Studio

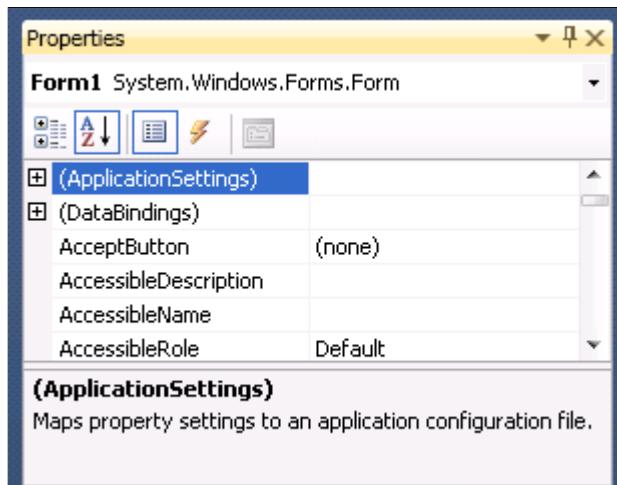
- Cuadro de herramientas
 - Explorador de base de datos
 - Explorador de soluciones
- Propiedades**
- Menús y barras de botones
 - Otras consideraciones

Módulo 3 - Capítulo 1

4. Propiedades

La ventana de propiedades la encontraremos en la parte derecha y más abajo de la ventana **Explorador de soluciones** en nuestro entorno de desarrollo.

Esta ventana nos permitirá acceder a las propiedades de los objetos insertados en nuestros formularios Windows, como se muestra en la figura 1.



Ventana de Propiedades de Visual Studio 2010

Figura 1

Para acceder a las propiedades de un determinado control, deberemos seleccionar el control en el formulario Windows y acudir a la ventana **Propiedades**, o bien, seleccionar el control en el formulario Windows y presionar la tecla **F4**.

Lección 1: Diseñador de Visual Studio

- Cuadro de herramientas
 - Explorador de base de datos
 - Explorador de soluciones
 - Propiedades
- Menús y barras de botones**
- Otras consideraciones

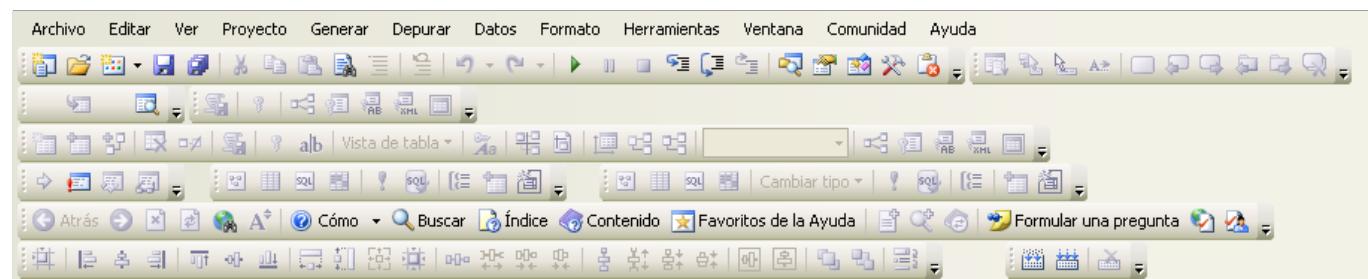
Módulo 3 - Capítulo 1

5. Menús y barra de botones

Respecto a los menús y barra de botones, son muchas las opciones que tenemos disponibles, tal como podemos comprobar en la figura 1. Las barras de botones son configurables, además de que podemos elegir las que queremos que se muestren de forma permanente en el entorno de desarrollo de Visual Studio 2010.

Algunas de las barras de botones se mostrarán automáticamente según las tareas que estemos realizando, por ejemplo, cuando estamos en modo depuración o diseñando las tablas de una base de datos.

Con el contenido de los menús ocurre lo mismo, según el elemento que tengamos seleccionado se mostrarán ciertas opciones que sea relevantes para ese elemento del IDE de Visual Studio 2010.



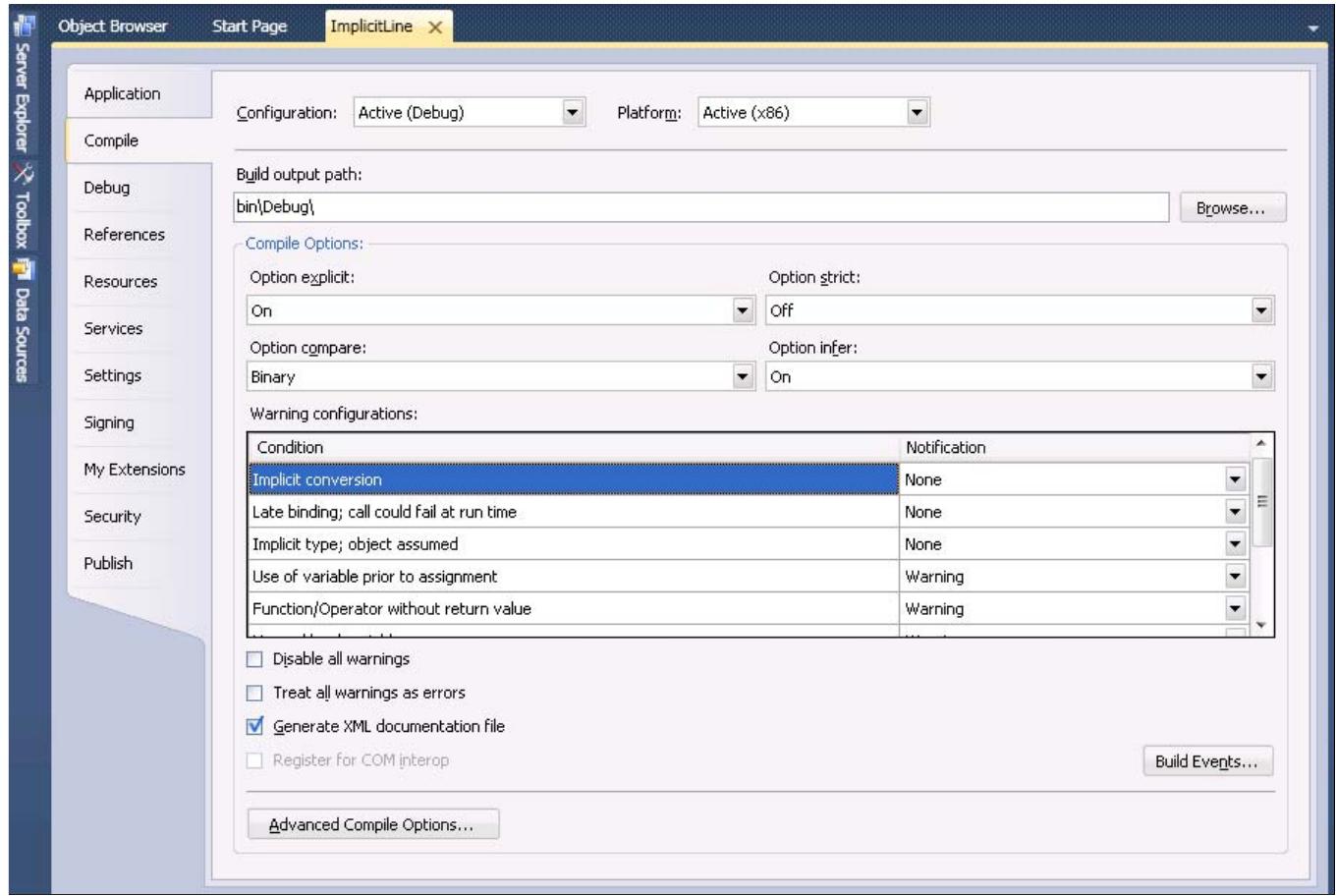
Los menús y barras de botones de Visual Studio 2010

Figura 1

Algunas de las opciones que tenemos en los menús también las podemos conseguir usando los menús contextuales (el mostrado al presionar con el botón secundario del mouse), y como es de esperar, también serán diferentes según el elemento sobre el que hemos presionado.

Por ejemplo, para configurar el proyecto actual, podemos elegir la opción **Propiedades** del menú **Proyecto** o bien presionar con el botón secundario del mouse sobre el proyecto mostrado en el Explorador de soluciones.

Al seleccionar las propiedades del proyecto, tendremos una nueva ventana desde la que podemos configurar algunas de las características de nuestro proyecto. En la figura 2, tenemos esa ventana de propiedades del proyecto, en la que podemos apreciar que está dividida según el tipo de configuración que queremos realizar, en este caso concreto las opciones de generación o compilación del proyecto.



Propiedades del proyecto sobre la que se trabaja en Visual Basic 2010

Figura 2

Como vemos en la figura 2, existen sin embargo multitud de opciones y apartados diferentes relacionados todos ellos con nuestra solución. Otro de los apartados destacables, es el apartado denominado **Publicar**.

Aún así, éste es el corazón o parte fundamental que debemos controlar a la hora de desarrollar una aplicación o a la hora de gestionar una solución, porque dentro de esta ventana, se resume buena parte de los menús y barra de botones del entorno de *Visual Studio 2010*.

De todos los modos, tendremos la oportunidad de ver más adelante, algunos usos de algunas de las opciones de la barra de botones del entorno.

Lección 1: Diseñador de Visual Studio

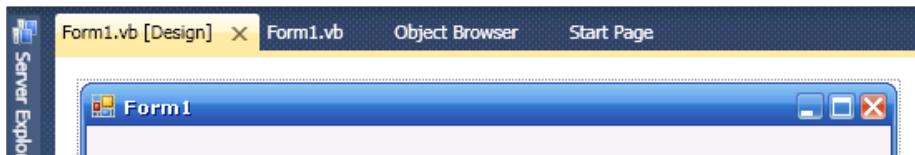
- Cuadro de herramientas
 - Explorador de base de datos
 - Explorador de soluciones
 - Propiedades
 - Menús y barras de botones
- Otras consideraciones

Módulo 3 - Capítulo 1

6. Otras consideraciones

El desarrollador que haya utilizado previamente otros entornos de desarrollo distinto a los de la familia de Visual Studio .NET, encontrará muy interesantes algunos de los cambios incorporados en *Visual Studio 2010*. Al principio, quizás se encuentre un poco desorientado, pero rápidamente y gracias a su experiencia en otros entornos de desarrollo, se acostumbrará al cambio. Entre algunos de estos cambios, destacaría los siguientes:

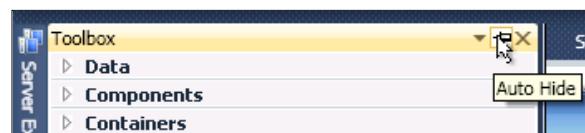
- En *Visual Studio 2010*, acceder a los objetos de nuestra aplicación es mucho más fácil. Dentro del entorno, observaremos que se van creando diferentes solapas que nos permite acceder y localizar los recursos con los que estamos trabajando de forma rápida. En la figura 1 podemos observar justamente esto que comenté.



Solapas de los objetos abiertos en Visual Studio 2010

Figura 1

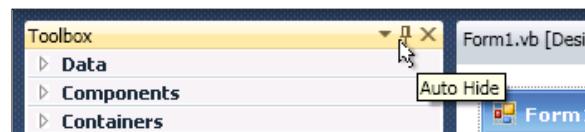
- *Visual Basic 2010* permite, hacer un **Stop & Go** (editar y continuar), de nuestras aplicaciones, es decir, pausar la ejecución de una aplicación en modo depuración y modificar los valores o propiedades que deseemos y continuar ejecutándola. Esta opción que los programadores de *Visual Basic 6* utilizan con mucha frecuencia en el desarrollo de sus aplicaciones, se ha mantenido en *Visual Basic 2010*, pero no en *Visual Studio .NET 2002* y *Visual Studio .NET 2003*. Si por alguna razón, debe trabajar con alguno de estos entornos, debe saber que esta opción no está disponible para las versiones comentadas.
- Otra característica que debemos conocer de nuestro entorno de desarrollo, es la capacidad de anclar o fijar una ventana de las comentadas anteriormente o de permitir que se haga visible cuando acercamos el puntero del mouse sobre ella. Esta opción es la que puede verse en la figura 2.



Opción de ocultar o mostrar la ventana seleccionada en Visual Studio 2010

Figura 2

Nótese que al presionar el ícono indicado en la figura 2, haremos que esta ventana quede fija en el entorno de desarrollo. Cuando pulsamos este ícono, la ventana queda fija y queda representado por un ícono como el que se muestra en la figura 3.



Icono para ocultar o mostrar la ventana seleccionada cuando se encuentra en modo anclado

Figura 3

- Algo que *oculta* el entorno de *Visual Studio 2010* por defecto, son las denominadas *clases parciales*. Se trata de una característica añadida a .NET 2.0, que permite *separar o partir* una clase en varias porciones de código.

La explicación ruda de esto, es que el programador puede tener dos ficheros de código fuente independientes, que posean el mismo nombre de clase.

Para indicar que pertenece a la misma clase, ésta debe tener la palabra clave **Partial** como parte de su definición para indicar que es una clase parcial.

Un ejemplo que aclare esto es el siguiente:

```
Código  
Public Class Class1  
  
    Public Function Accion1() As Integer  
        Return 1  
    End Function  
  
End Class
```

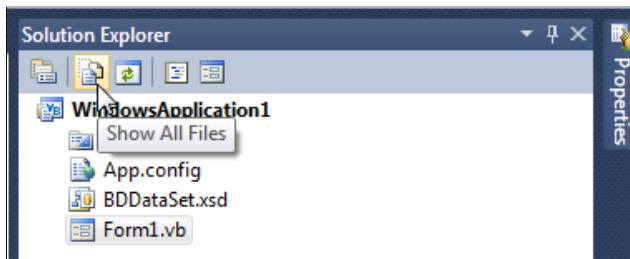
```
Código  
Partial Public Class Class1  
  
    Public Function Accion2() As Integer  
        Return 2  
    End Function  
  
End Class
```

El comportamiento de la clase es el de una única clase, por lo que su declaración y uso es como el de cualquier clase normal, tal y como se indica en el siguiente código:

```
Código  
Public Class Form1  
  
    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button1.Click  
        Dim MiClase As New Class1  
        MessageBox.Show(MiClase.Accion2.ToString() & vbCrLf & MiClase.Accion1.ToString())  
    End Sub  
End Class
```

De todas las maneras, el entorno nos oculta muchas veces las clases parciales de un aplicación.

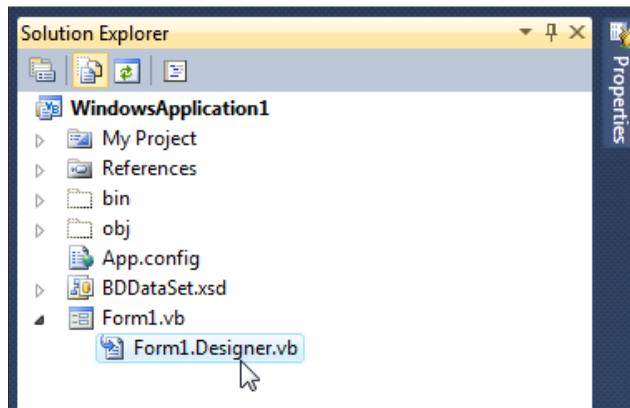
Para ello, presionaremos la opción **Mostrar todos los archivos** de la ventana *Explorador de soluciones* como se indica en la figura 4.



Icono u opción para mostrar todos los archivos del proyecto

Figura 4

De esta manera, podremos acceder a los archivos y recursos del proyecto, incluidas las clases parciales como se indica en la figura 5. En el archivo Form1.Designer.vb estará el código utilizado por el diseñador de formularios de Windows Forms, en el que se incluye la declaración de todos los controles y controladores de eventos que hemos definido en nuestro proyecto.



Clase parcial mostrada en los archivos del proyecto

Figura 5

A tener en cuenta:

Cuando se genera un proyecto con Visual Studio 2010, el entorno genera diferentes clases parciales, como por ejemplo la que se genera para un formulario.

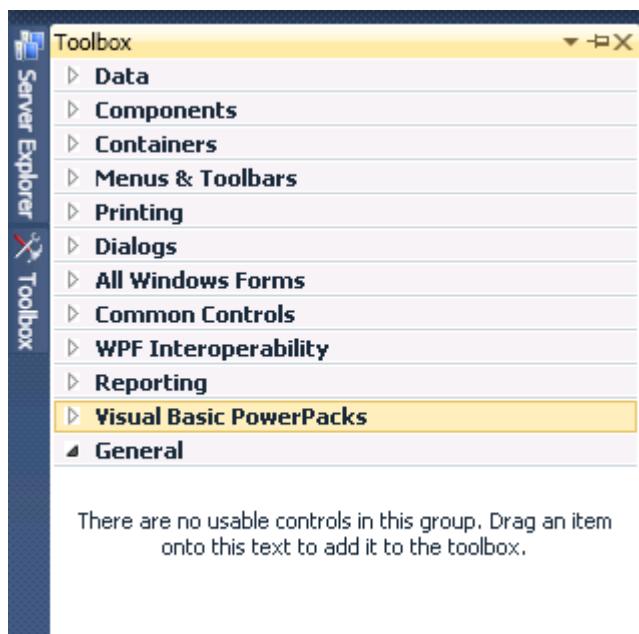
Lección 2: Controles de Windows Forms

- [Datos](#)
- [Componentes](#)
- [Controles Comunes](#)
- [General](#)
- [Otras consideraciones](#)

Introducción

Dentro del entorno de desarrollo de *Visual Studio 2010*, nos encontramos un enorme conjunto de librerías y controles que podemos utilizar en nuestras aplicaciones Windows. Dependiendo del tipo de aplicación que llevemos a cabo, el entorno habilitará los controles correspondientes para cada tipo de aplicación. En nuestro caso, nos centraremos en los controles más habituales de Windows, e indicaremos como utilizarlos en nuestros desarrollos.

En nuestro entorno de desarrollo, encontraremos diferentes grupos de controles o componentes dispuestos de ser utilizados. En la figura 1 encontraremos los grupos de controles y componentes más habituales.



Grupos de controles en Visual Studio 2010

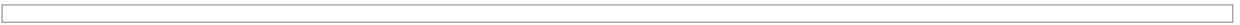
Figura 1

Estos controles se dividen en los grupos representados en la figura anterior. A continuación veremos los más representativos.

Módulo 3 - Capítulo 2

- 1. [Datos](#)

- 2. [Componentes](#)
- 3. [Controles comunes](#)
- 4. [General](#)
- 5. [Otras consideraciones](#)



Lección 2: Controles de Windows Forms

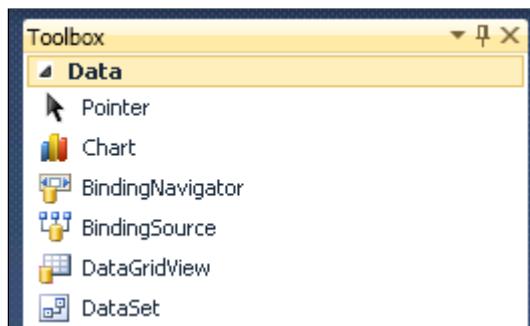
Datos

- Componentes
- Controles Comunes
- General
- Otras consideraciones

Módulo 3 - Capítulo 2

1. Datos

El grupo *Datos* corresponde con el grupo que tiene relación directa con los componentes de acceso a datos, como se muestra en la figura 1.

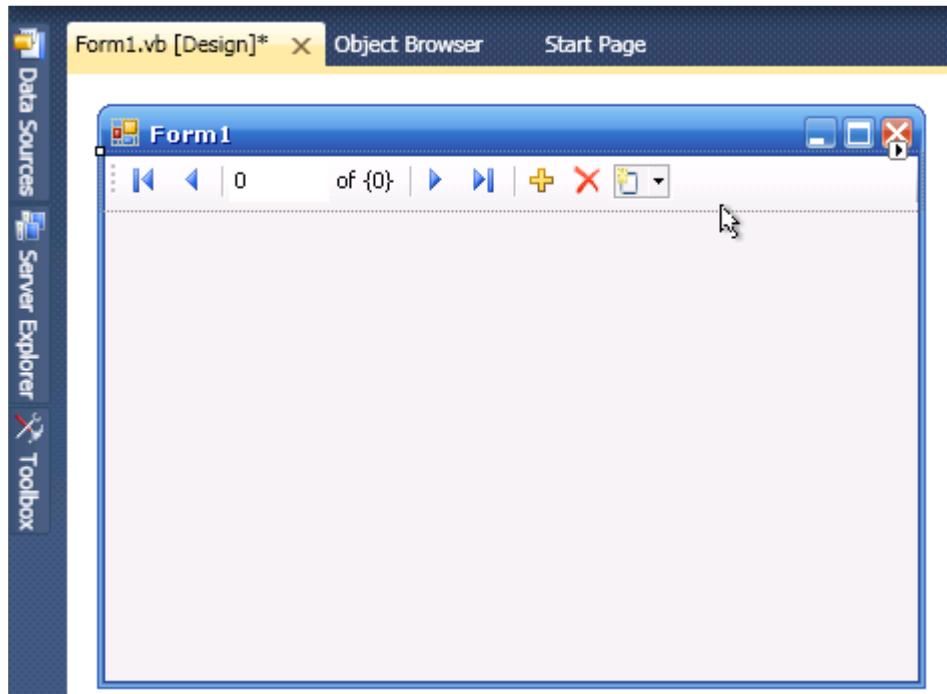


Controles de Datos en Visual Studio 2010

Figura 1

Para muchos desarrolladores, los controles, componentes y métodos de acceso a datos, contiene dentro de sí un especial misterio, es como el *Santo Grial* de la programación. Casi siempre nos atascamos ahí, siempre en el mismo sitio. Pero no se preocupe ni lo más mínimo por ello, aprenderemos a utilizarlos a base de práctica, y lo que es más importante, los dominaremos rápidamente. Solo como curiosidad y por ahora, le presentaré uno de los componentes más destacables en *Visual Studio 2010*, por su semejanza con otro muy utilizado en "otros" entornos de desarrollo, estoy hablando del control y componente **BindingNavigator** que usaremos frecuentemente en nuestras aplicaciones con acceso a fuentes de datos.

Este control insertado en un formulario Windows, es el que se puede ver en la figura 2.



Control **BindingNavigator** insertado en un formulario Windows en Visual Studio 2010

Figura 2

Como puede observar, este control, tiene un aspecto muy similar al del famoso *Recordset* de Visual Basic 6 o al *DataNavigator* de Borland. Lógicamente, este control tiene un aspecto mucho más vistoso y moderno, pero es uno de los controles estrella de *Visual Basic 2005* y que también están incluidos en *Visual Studio 2010*, ya que en *Visual Studio .NET 2002* y *Visual Studio .NET 2003* no existía este control en el entorno. Desde *Visual Studio 2005* y por lo tanto, desde *Visual Basic 2005*, tenemos la posibilidad de trabajar con el control **BindingNavigator**.

Comunidad dotNet:

Visual Studio 2010 le proporciona un amplio conjunto de controles y componentes así como un no menos completo conjunto de clases que le facilita al desarrollador las tareas de programación requeridas. Sin embargo, existen contribuciones gratuitas y otras de pago, que el programador puede utilizar según lo requiera. A continuación le indico el que a mi modo de ver es el lugar más representativo de este tipo de contribuciones a la Comunidad de desarrolladores .NET.

► [Microsoft CodePlex](#)

Lección 2: Controles de Windows Forms

- Datos
- Componentes**
- Controles Comunes
- General
- Otras consideraciones

Módulo 3 - Capítulo 2

2. Componentes

Windows Forms incluye un conjunto de componentes muy nutrido y variado. Algunos de estos componentes, han sido mejorados y otros ampliados. En la figura 1 podemos observar estos componentes.

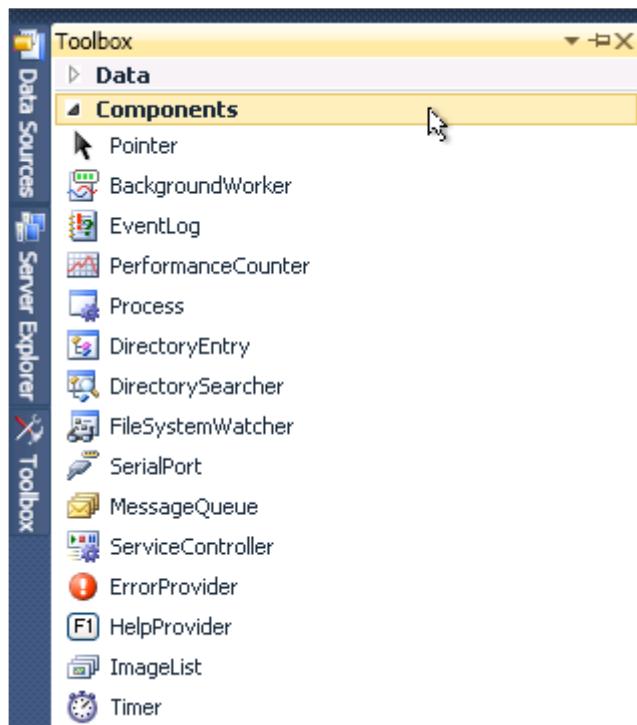


Figura 1

Los componentes son *como controles no visibles*, o dicho de otra forma, son controles que realizan ciertas tareas, pero no tienen un interfaz que mostrar, como puede ser el caso de un botón o una caja de textos.

Por ejemplo, el componente *Timer* nos permite recibir una notificación cada x tiempo, pero no muestra nada

al usuario de nuestra aplicación. Si hacemos doble clic sobre el componente *Timer* para insertarlo en el formulario, éste quedará dispuesto en la parte inferior del formulario como se indica en la figura 2.



Este tipo de componentes no son visibles en tiempo de ejecución.

Lección 2: Controles de Windows Forms

- Datos
- Componentes
- Controles Comunes**
- General
- Otras consideraciones

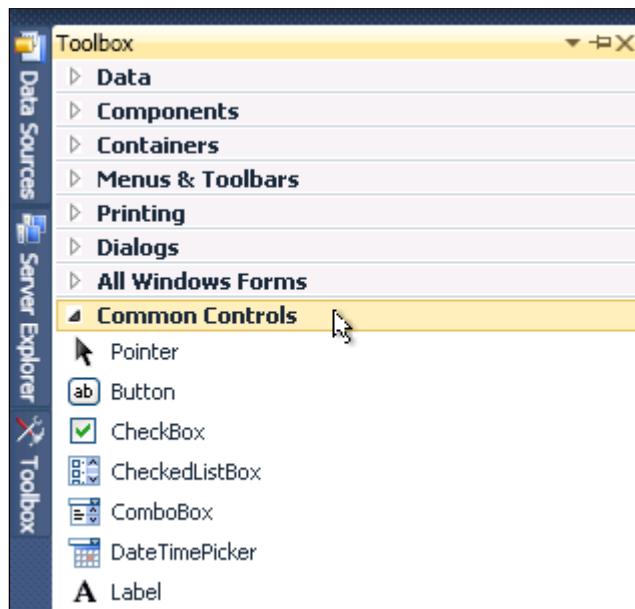
Módulo 3 - Capítulo 2

3. Controles comunes

Con este nombre, se agrutan los controles más generales y variados que podemos utilizar en nuestras aplicaciones Windows. Sería algo así, como el resto de controles y componentes no contenidos en ninguna de las secciones que hemos visto anteriormente, aunque esto no es siempre así. Por esa razón, si encuentra dos controles o componentes iguales en dos o más secciones, no lo tenga en consideración.

Digamos que en esta solapa se agrutan por lo tanto, los controles que utilizaremos con más frecuencia.

En la figura 1 podemos observar los controles y componentes citados.



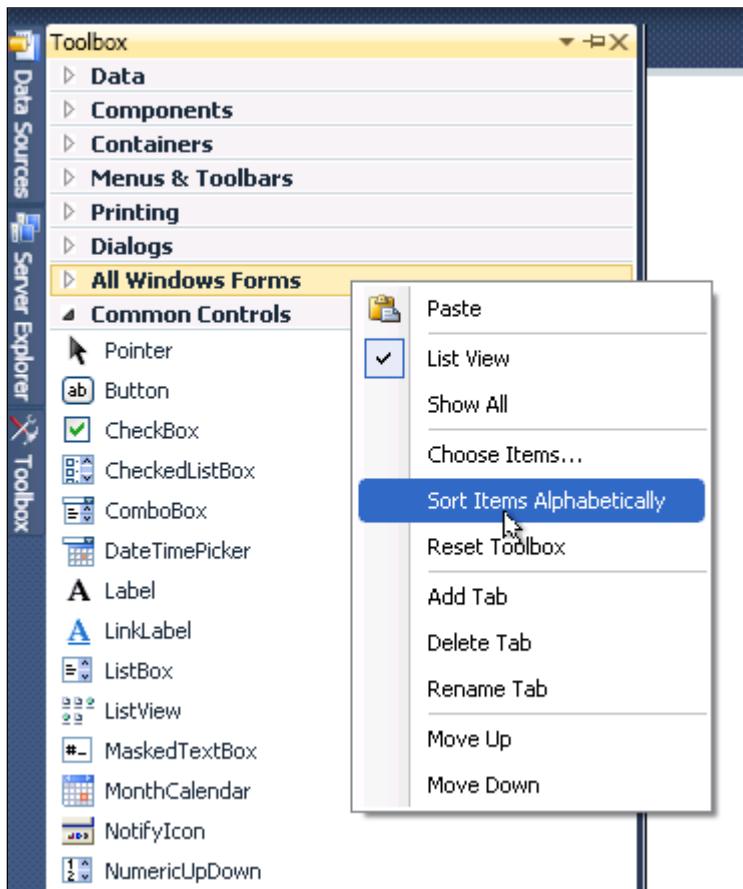
Controles Windows Forms en Visual Studio 2010

Figura 1

Debido a la cantidad de controles y componentes de los distintos grupos del Cuadro de herramientas, podemos usar el siguiente truco para que nos resulte más fácil su localización.

Truco:

Como puede observar, a veces cuesta localizar un control debido a la enorme cantidad de controles que hay. Para ordenarlos, puede arrastrar y soltar los controles y componentes en la barra de herramientas o bien, si quiere hacer una ordenación por orden alfabético, puede hacer clic con el botón secundario del mouse sobre una determinada sección de controles y seleccionar la opción **Ordenar elementos alfabéticamente** como se indica en la siguiente figura siguiente:



Los controles y componentes de esa sección quedarán ordenados alfabéticamente.

Lo más destacable para el desarrollador habituado a otros entornos, es que aquí veremos una gran cantidad de controles que nos resultarán muy familiares.

Controles como: **Label**, **PictureBox**, **TextBox**, **Frame** que ahora pasa a llamarse **GroupBox**, **CommandButton** que ahora pasa a llamarse **Button**, **CheckBox**, **OptionButton** que ahora pasa a llamarse **RadioButton**, **ComboBox**, **ListBox**, **HScrollBar**, **VScrollBar**, **Timer**, etc.

Pero además tenemos muchos otros que no son tan habituales en todos los entornos de desarrollo diferentes de Visual Studio .NET. Controles que proporcionan nuevas y ventajosas características a la hora de desarrollar aplicaciones con Visual Basic 2010.

Entre estos controles, podemos encontrar el control **PrintDocument** y **PrintPreviewControl**, para imprimir y realizar vistas preliminares, **ErrorProvider**, **WebBrowser**, **FolderBrowserDialog**, **ToolTip** para aportar *tooltips* a nuestros controles, **TrackBar**, **NumericUpDown**, **SplitContainer**, **MonthCalendar**, **DateTimePicker**, etc.

Cada uno de los controles, tiene unas características y cualidades determinadas.

Sólo a base de práctica, aprenderemos a utilizarlos y lo único que debemos saber, es cuál de ellos utilizar en un momento dado.

El abanico de controles y componentes es lo suficientemente amplio como para poder abordar con ellos, cualquier tipo de proyecto y aplicación Windows que nos sea demandada.

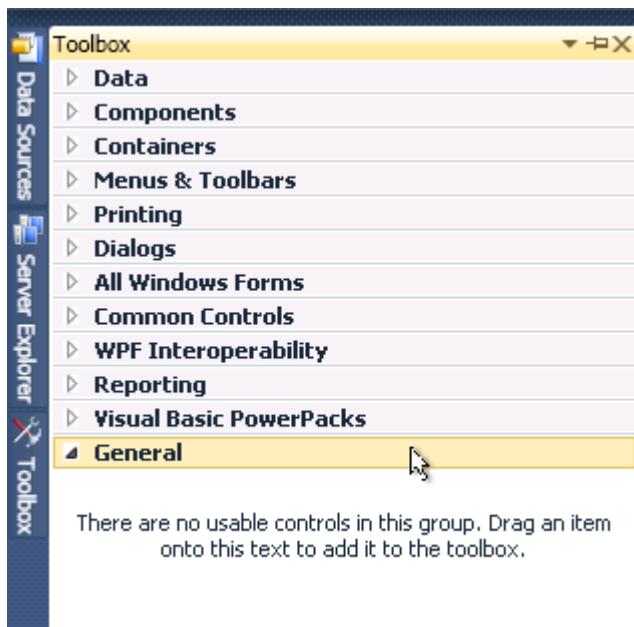
Lección 2: Controles de Windows Forms

- Datos
- Componentes
- Controles Comunes
- General**
- Otras consideraciones

Módulo 3 - Capítulo 2

4. General

Esta sección es como el cajón desastre, un lugar dónde podemos insertar otros controles o componentes desarrollados por terceros, por ejemplo.



Sección General en Visual Studio 2010

Figura 1

Esta sección de todos los modos, la puede utilizar un desarrollador en muchos casos.

Por ejemplo, los desarrolladores que desean arrastrar y soltar aquí los controles y componentes que más utiliza o los que utiliza en un determinado proyecto.

Otro caso de ejemplo es cuando se trabaja con controles o componentes similares desarrollados por dos empresas diferentes que queremos tener localizados o separados para no mezclarlos.

En otras circunstancias, tampoco es raro encontrarse con controles o componentes con iconos similares, por

lo que aclararse cuál es el que nos interesa puede ser una tarea obligada.

Aún así, otra de las posibilidades con la que nos podemos encontrar para utilizar esta sección es la de tener que utilizar un control o componente circunstancialmente en un momento dado, y por eso, que no deseemos añadir este control o componente a otra sección como la de *Controles comunes* por ejemplo.

Utilice por tanto esta sección como lo considere oportuno.

Lección 2: Controles de Windows Forms

- Datos
 - Componentes
 - Controles Comunes
 - General
- Otras consideraciones

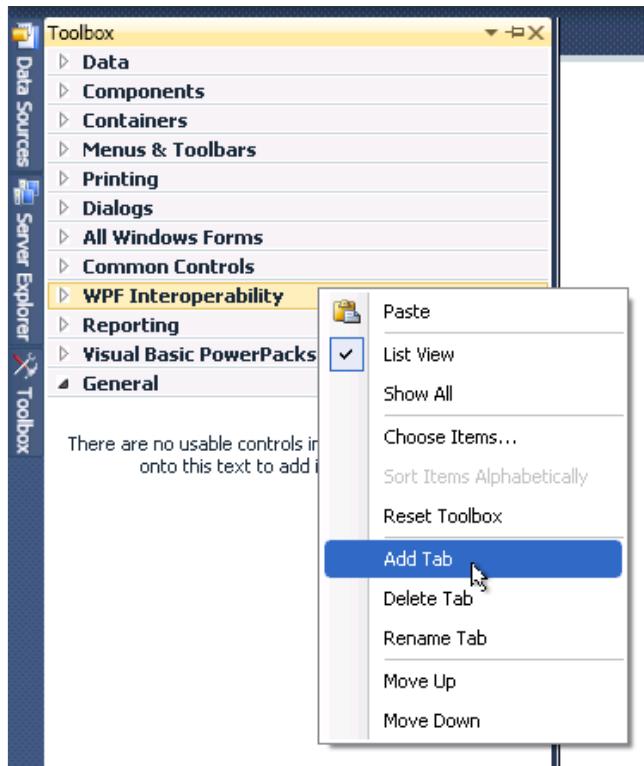
Módulo 3 - Capítulo 2

5. Otras consideraciones

La sección *General* nos indica un repositorio de ámbito y carácter general, sin embargo, el desarrollador puede querer ordenar su propio repositorio o sección de controles y componentes.

Manipulando el Cuadro de herramientas

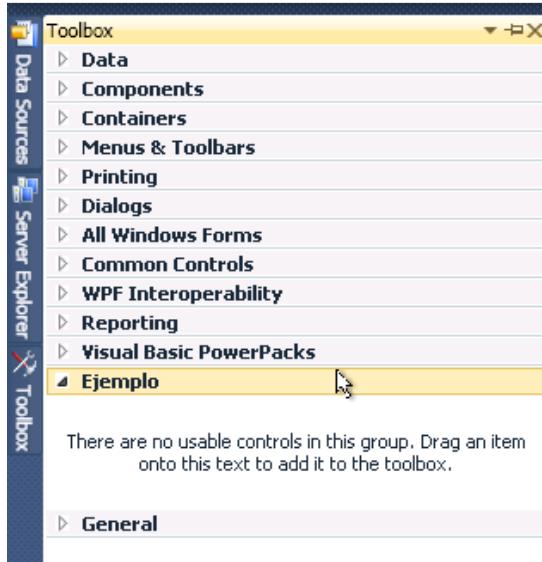
Para ello, nos posicionaremos en la barra de herramientas y presionaremos el botón secundario del mouse sobre la parte gris de la barra de herramientas desplegada y seleccionaremos la opción *Agregar ficha* del menú emergente, como se muestra en la figura 1.



Opción de personalización de nuestros propios grupos de controles y componentes

Figura 1

Cuando seleccionamos esta opción, aparecerá una caja de texto en la barra de herramientas dónde podremos escribir el nombre que consideremos oportuno, como se muestra en la figura 2.



Personalización de un grupo de controles y componentes en Visual Studio 2010

Figura 2

Si se hace la siguiente pregunta, ¿cómo cambiar el nombre de una sección ya creada o una existente?, sepa que deberá realizar los siguientes pasos.

Haga clic con el botón secundario del mouse sobre la sección sobre la que desea cambiar el nombre y seleccione la opción **Cambiar nombre de ficha** como se muestra en la figura 3.

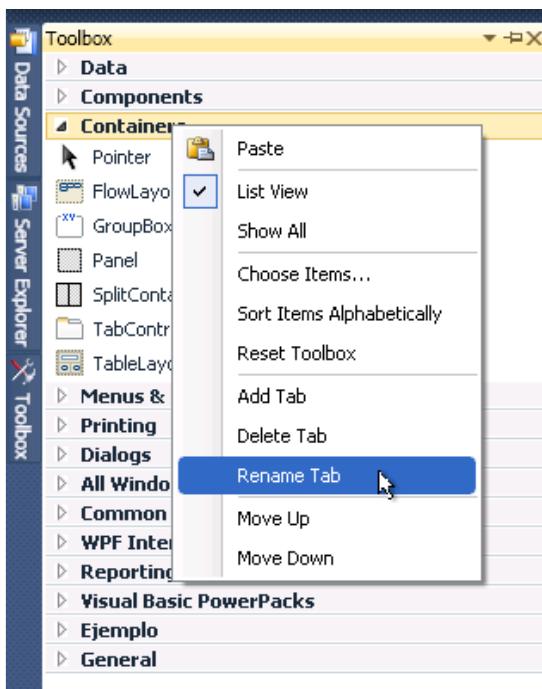


Figura 3

De igual forma, puede cambiar también el nombre de los controles o componentes insertados.

Para hacer eso, haga clic con el botón secundario del mouse sobre un control o componente y seleccione la opción **Cambiar nombre de elemento** como se muestra en la figura 4.

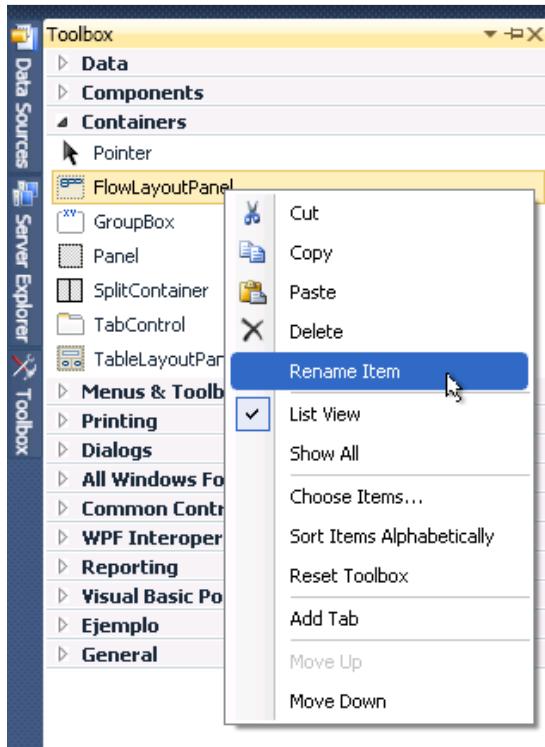


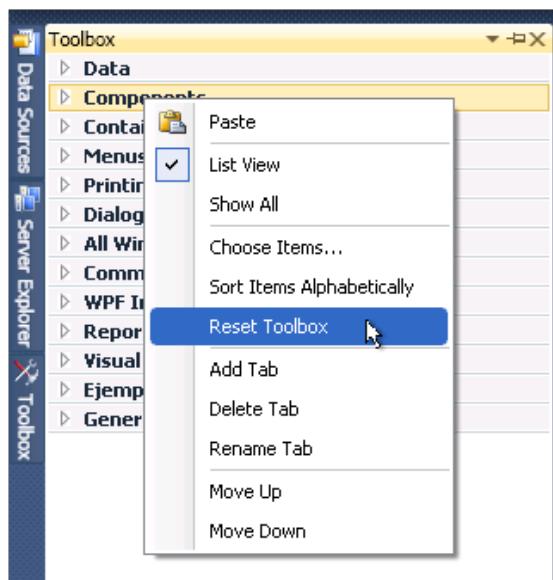
Figura 4

Visual Basic 2010, nos proporciona un amplio conjunto de opciones de personalización del entorno de trabajo, para que se ajuste a las exigencias de los desarrolladores.

FAQ:

¿Qué ocurre si me equivoco personalizando mi barra de herramientas?

Visual Studio 2010 nos proporciona la posibilidad de resetear o restaurar el estado inicial de la barra de herramientas en el entorno de desarrollo. Para hacer esto, haremos clic con el botón secundario del mouse la barra de herramientas y seleccionaremos la opción **Restablecer cuadro de herramientas** del menú emergente, como se muestra en la siguiente figura.



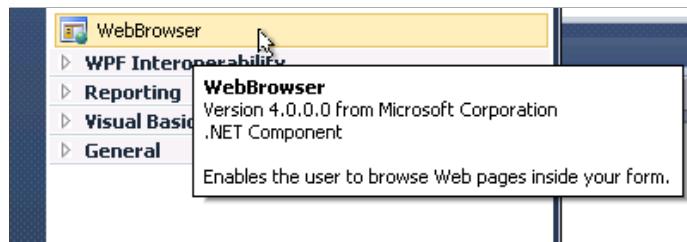
Ojo!, al seleccionar esta opción, perderemos todas las modificaciones que hayamos realizado sobre la barra de herramientas.

Otros controles a tener en cuenta

Dentro del entorno de Visual Studio 2010 y en .NET en general, tenemos la posibilidad de utilizar diferentes controles muy

interesantes que conviene comentar.

Uno de estos controles, se llama *WebBrowser*, tal y como se indica en la figura 5.



Control WebBrowser en el Cuadro de herramientas

Figura 5

Este control es la representación de un control específico para mostrar contenido XML o contenido HTML, como si de una página Web se tratara.

Sirva el siguiente ejemplo de código fuente para demostrar como usar el control y como se muestra dicho control en una aplicación Windows.

El código de la aplicación quedaría como se detalla a continuación:

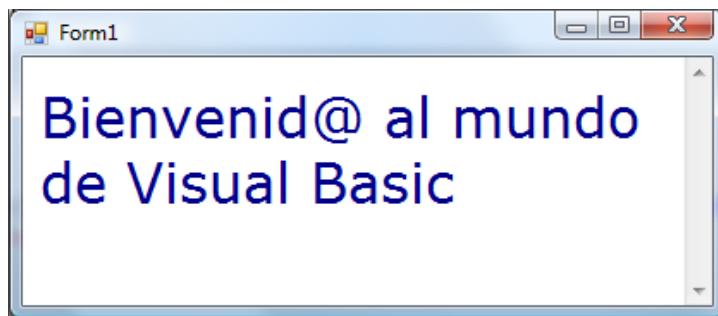
```
Código

Public Class Form1

    Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.Load
        Me.webBrowser1.Navigate("http://localhost/Bienvenido.aspx")
    End Sub

End Class Form1
```

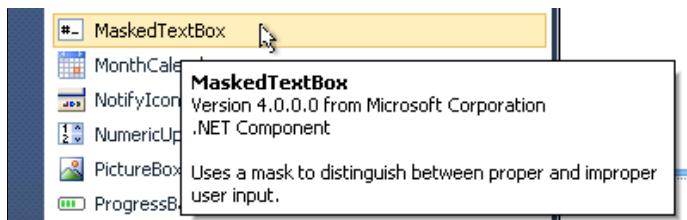
Nuestro ejemplo en ejecución es el que se muestra en la figura 6.



Control WebBrowser en ejecución

Figura 6

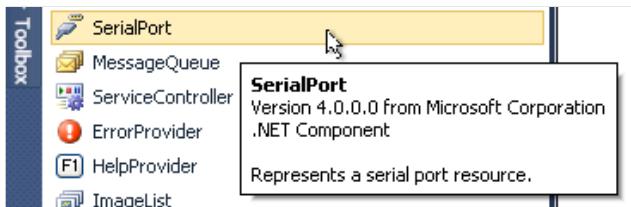
Hay más controles que representan una novedad para el desarrollador de .NET, como puede ser por ejemplo, el control **MaskedTextBox**, como se muestra en la figura 7.



Control MaskedTextBox en Visual Basic 2010

Figura 7

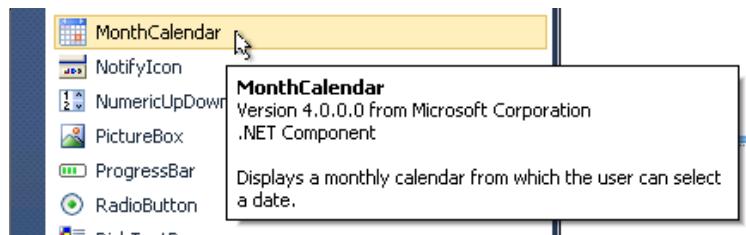
Sin embargo, hay otros controles clásicamente demandados por los desarrolladores, como los controles de accesos a puertos COM y puertos serie, como es el caso del control **SerialPort** que se muestra en la figura 8.



Control SerialPort en Visual Basic 2010

Figura 8

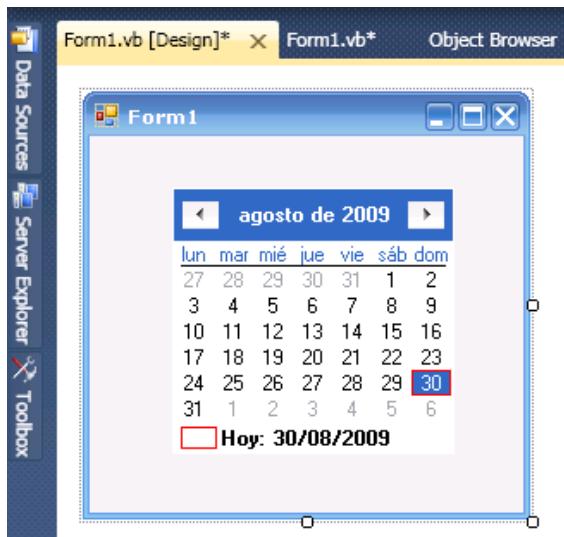
No es cuestión de repasar cada uno de los controles que el programador puede encontrar en *Visual Studio 2010*, sin embargo, no me gustaría dejar de comentar, uno de los controles más usados y útiles para las aplicaciones Windows, que tiene a su vez su equivalente para el desarrollo de aplicaciones Web en ASP.NET. Me refiero al control **MonthCalendar** que se muestra en la figura 9.



Control MonthCalendar en Visual Basic 2010

Figura 9

Este control, que se muestra en la figura 10 cuando lo insertamos en un formulario, es un control que nos facilita la entrada de fechas en el sistema y permite asegurarnos, que la fecha seleccionada es una fecha válida.

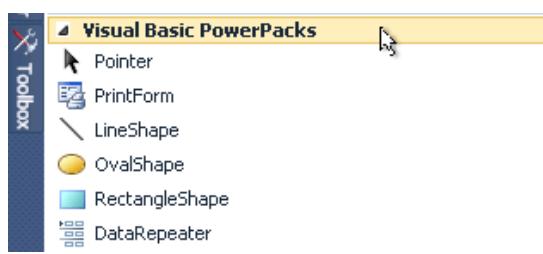


Control MonthCalendar insertado en un formulario Windows

Figura 10

Otros controles añadidos en Visual Studio 2010 y en concreto para Visual Basic 2010, son los controles del *PowerPack*.

Estos controles diseñados por el equipo de trabajo de Visual Basic, se podían descargar e instalar de forma independiente en el sistema, sin embargo, en Visual Studio 2010 estos controles se han incorporado directamente sin necesidad de descargarlos de forma separada.



Controles PowerPack de Windows en Visual Basic

Figura 11

Lección 3: Trabajo con controles

- Dominando los controles en el entorno de trabajo
- Creación de controles en tiempo de ejecución
- Creación de una matriz de controles
- Creación de controles nuevos
- Otras consideraciones

Introducción

Hasta ahora, hemos aprendido a identificar las partes más importantes del entorno de desarrollo de *Visual Studio 2010*, hemos visto igualmente como se separan los controles y componentes, y hemos visto también que existe un grupo de solapas donde podemos añadir nuestros propios controles y componentes, incluso hemos aprendido a crear nuestro propio grupo o sección de controles y componentes, en el que podemos también incluir los controles y componentes que por ejemplo hayamos desarrollado nosotros mismos, sin embargo, para poder insertar ahí un control o componente desarrollado por nosotros, deberíamos aprender a crearlos.

Eso es justamente lo que veremos a continuación además de ver otras técnicas que conviene repasar antes de aprender a desarrollar nuestros propios controles y componentes.

Por eso, lo primero que haremos será aprender a desenvolvernos adecuadamente en el entorno de desarrollo con los controles que ya conocemos.

Módulo 3 - Capítulo 3

- 1. [Dominando los controles en el entorno de trabajo](#)
- 2. [Creación de controles en tiempo de ejecución](#)
- 3. [Creación de una matriz de controles](#)
- 4. [Creación de controles nuevos](#)
- 5. [Otras consideraciones](#)

Lección 3: Trabajo con controles

Dominando los controles en el entorno de trabajo

- Creación de controles en tiempo de ejecución
- Creación de una matriz de controles
- Creación de controles nuevos
- Otras consideraciones

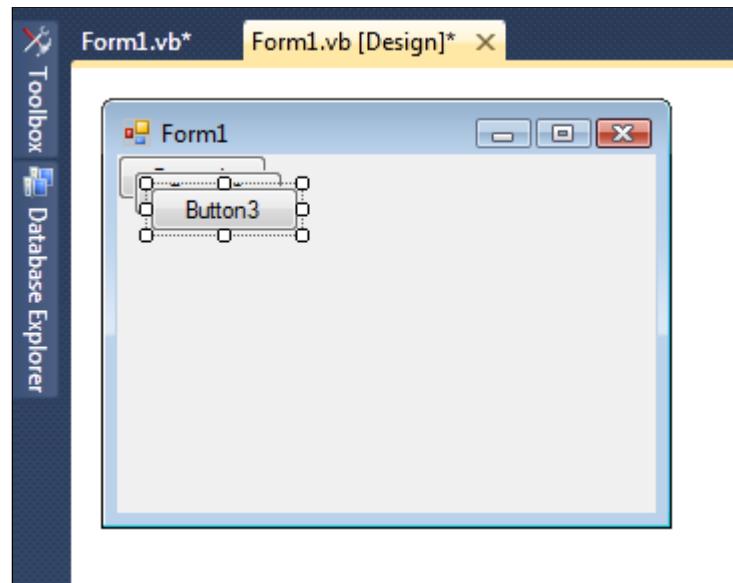
Módulo 3 - Capítulo 3

1. Dominando los controles en el entorno de trabajo

Si queremos aprender a crear nuestros propios controles para poder distribuirlos y utilizarlos en nuestro entorno, lo mejor es dominar el uso de los controles en nuestro entorno de desarrollo.

Ya hemos visto anteriormente como insertar un control a nuestro formulario, pero quizás no sepamos como manejarlos de forma eficiente en ese formulario.

Inserte en un formulario Windows por ejemplo, tres controles **Button** como se muestra en la figura 1.

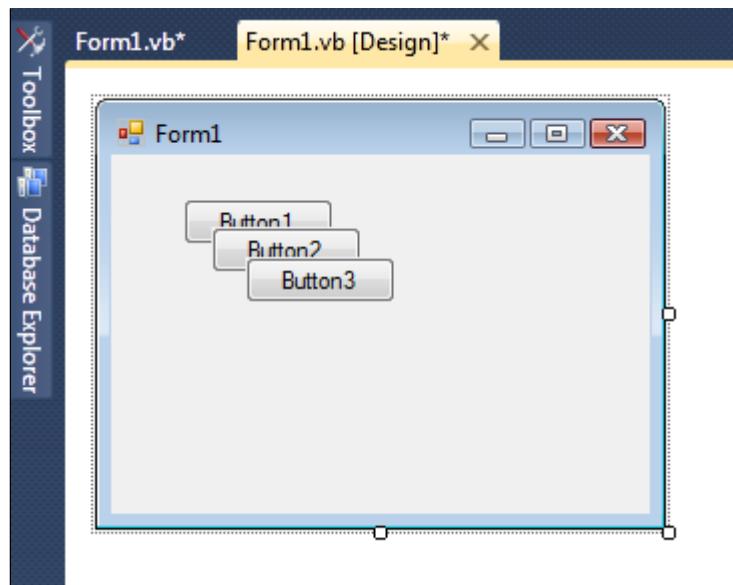


Controles Button insertados en un formulario Windows

Figura 1

Como podemos observar, los controles están dispuestos de una forma desordenada, ya que al insertarlos por ejemplo haciendo doble clic tres veces sobre un control **Button**, éstos quedan dispuestos anárquicamente en el formulario.

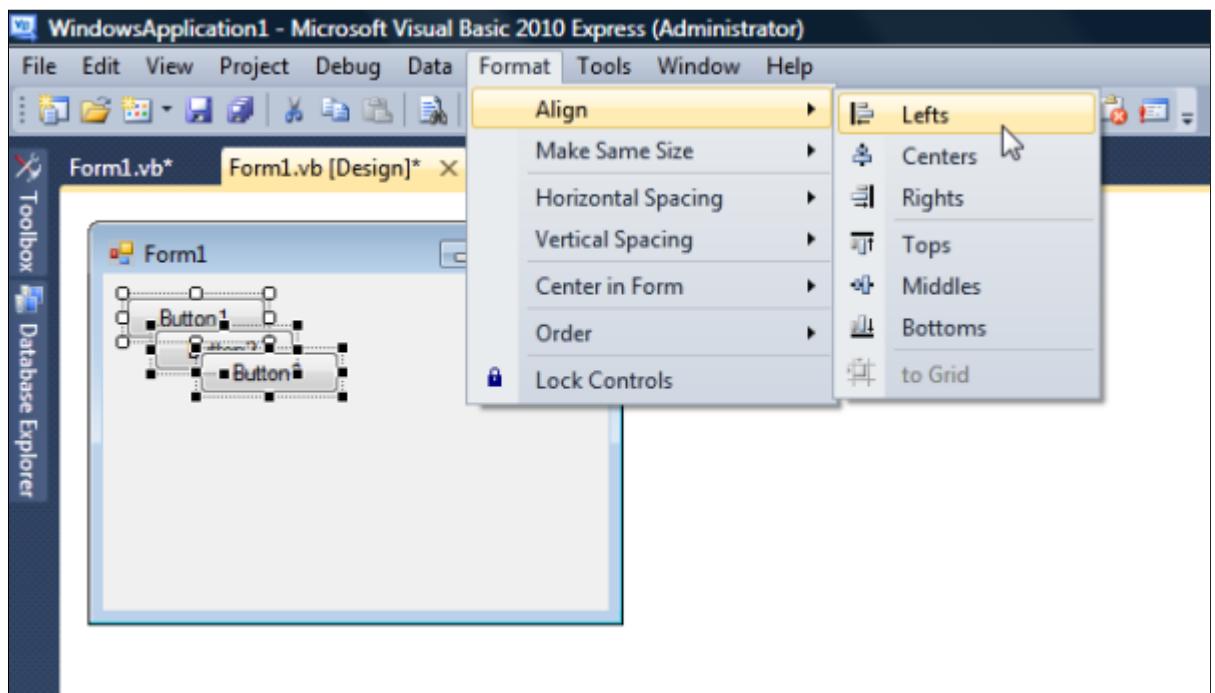
Separé los controles **Button** de forma que queden ahora esparcidos en el formulario de alguna manera tal y como se muestra en la figura 2.



Controles Button separados en el formulario

Figura 2

Seleccione todos los controles **Button** como se mostraba en la figura 2 y seleccione del menú las opciones **Formato > Alinear > Lados izquierdos** como se indica en la figura 3.

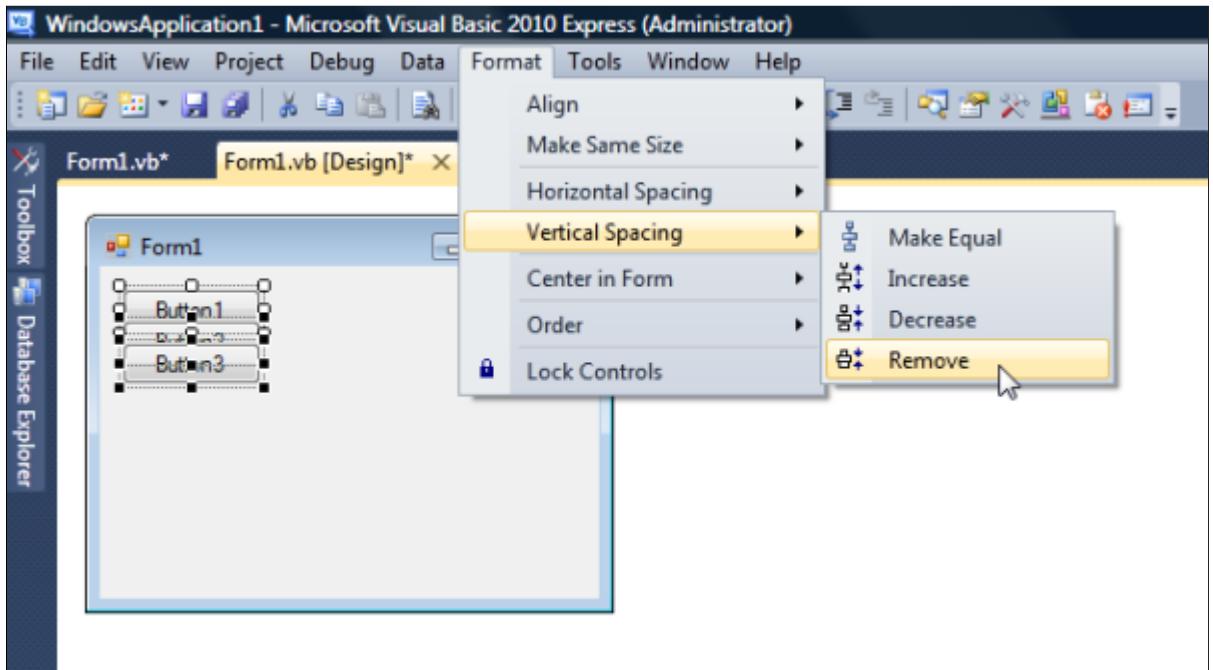


Los controles Button quedarán ordenados correctamente dentro del formulario

Figura 3

Sin embargo, podemos extender el uso de las propiedades especiales para alinear los controles en un formulario.

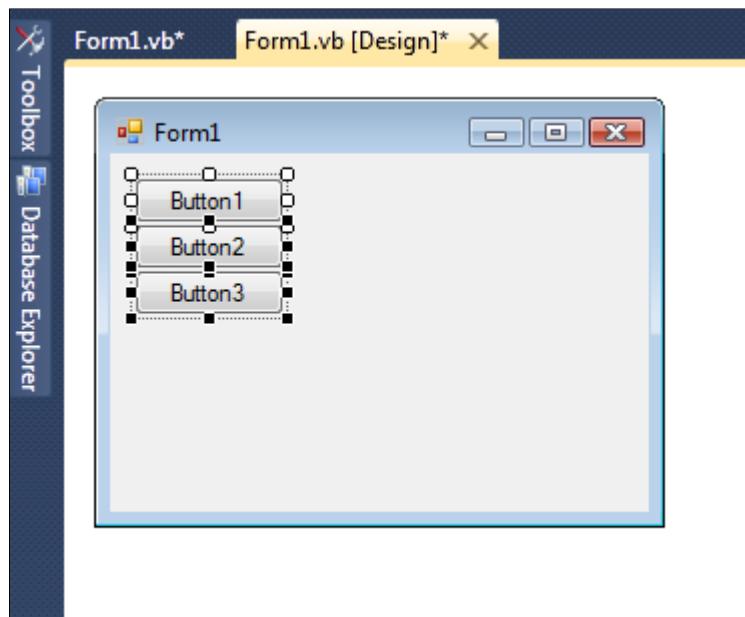
Por ejemplo, ahora que tenemos los controles **Button** alienados correctamente, podemos hacer uso de la opción de menú **Formato > Espaciado vertical > Quitar** como se indica en la figura 4.



Otras opciones especiales, nos permiten alinear o trabajar con los controles de forma rápida y segura

Figura 4

En este caso, los controles quedarán dispuestos en el formulario como se indica en la figura 5.



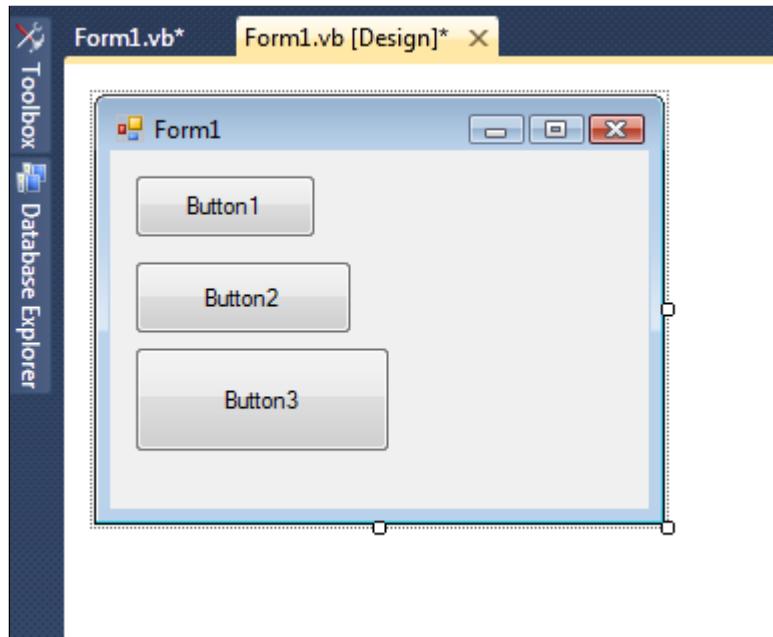
Controles alineados y espaciados según la elección de opciones del entorno

Figura 5

Como podemos apreciar, alinear los controles en el entorno es realmente sencillo.

Visual Studio 2010 nos proporciona una gran cantidad de opciones para llevar a cabo este tipo de tareas.

Incluso si nos encontramos con un controles de diferente tamaño entre sí como se muestra en la figura 6, podemos hacer uso de la opción del menú **Formato > Igualar tamaño** permitiéndonos cambiar el tamaño de los controles seleccionados dentro del formulario Windows.

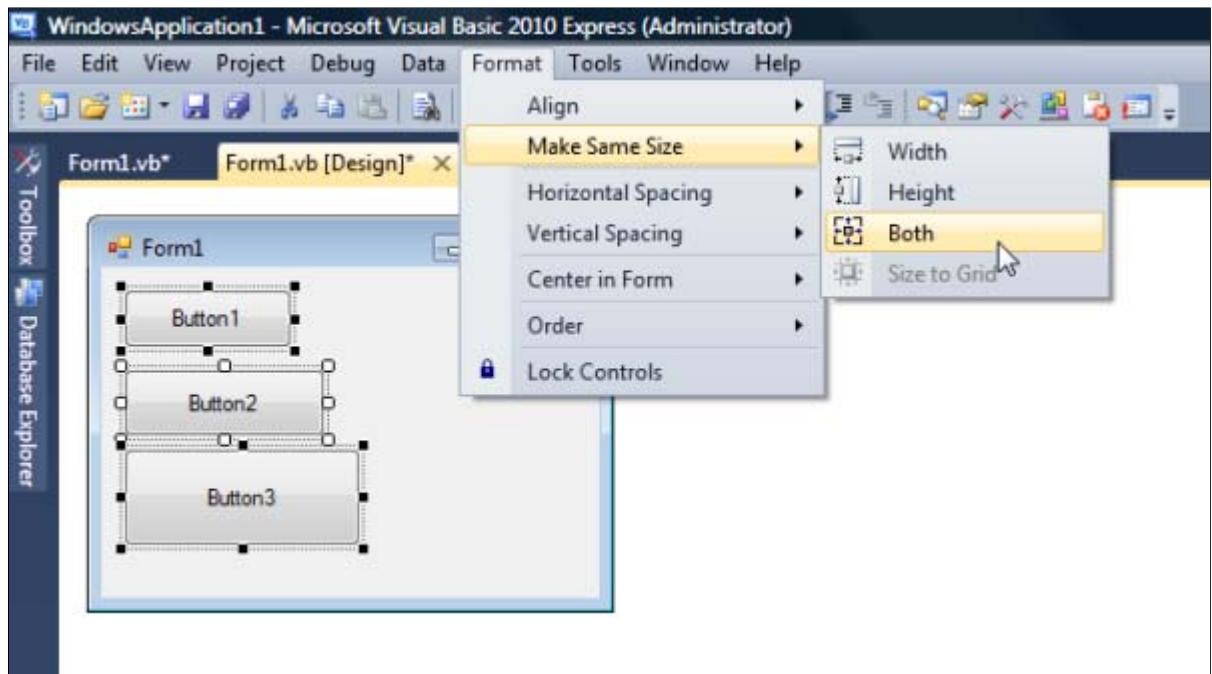


Controles de diferentes tamaños dispuestos en el formulario Windows

Figura 6

El menú que nos permite cambiar los tamaños de los controles insertados en un formulario posee diferentes posibilidades.

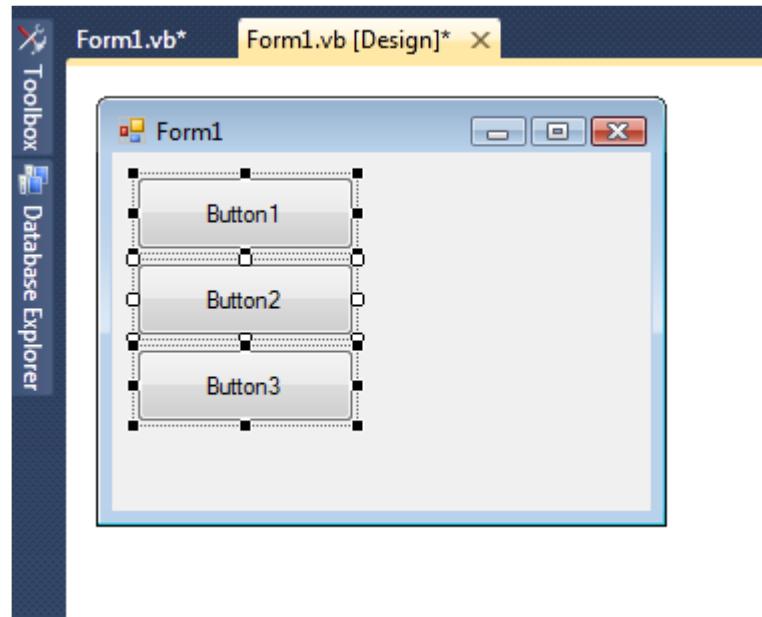
En nuestro caso, seleccionaremos del menú, la opción **Formato > Igualar tamaño > Ambos** tal y como se muestra en la figura 7.



Cambiando los tamaños ancho y alto de los controles seleccionados de un formulario

Figura 7

Una vez seleccionada esta opción, los controles se modificarán con el mismo tamaño tal y como se muestra en la figura 8.



Controles del formulario con su tamaño modificado

Figura 8

Una vez seleccionada esta opción, los controles se modificarán con el mismo tamaño tal y como se muestra en la figura 8.

Truco:

*Suponiendo que tengamos tres controles **Button** de diferentes tamaños y que queramos que todos tengan el mismo tamaño que el segundo de sus controles, seleccionaremos **siempre** como primer control, el control que queremos como base de tamaño para el resto de controles, y posteriormente con la tecla **Ctrl** seleccionaremos uno a uno el resto de controles.*

Sin embargo, para alinear los controles en un formulario tenemos más opciones.

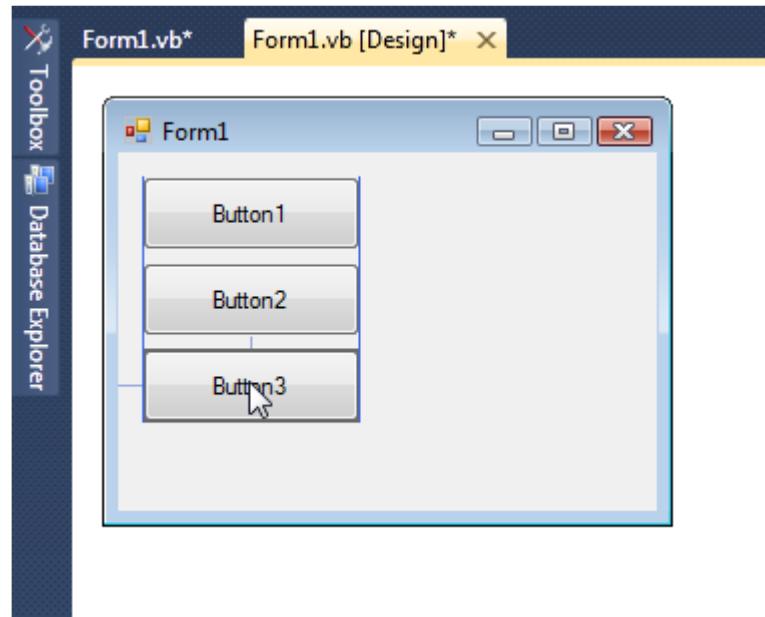
Hemos visto algunas de ellas, quizás las más habituales, pero como podemos deducir del menú **Formato**, podremos alinear los controles, espaciarlos entre sí horizontal o verticalmente, modificar su tamaño, centrarlos en el formulario horizontal o verticalmente, etc.

Por otro lado, Visual Studio 2010, nos proporciona una utilidad en tiempo de diseño muy útil.

Se trata de las guías de representación que veremos en tono azul claro, y que aparecen cuando movemos un control en el formulario.

Estas guías indican la situación y posición de un control respecto a otro próximo.

La representación de estas guías que se muestran en la figura 9, nos facilita enormemente la disposición de los controles en un formulario.



Guías o reglas de dirección o separación entre controles

Figura 9

Lección 3: Trabajo con controles

- Dominando los controles en el entorno de trabajo

Creación de controles en tiempo de ejecución

- Creación de una matriz de controles
- Creación de controles nuevos
- Otras consideraciones

Módulo 3 - Capítulo 3

2. Creación de controles en tiempo de ejecución

Prácticamente todo en .NET son objetos. Los controles también, así que para añadir un control a un formulario en tiempo de ejecución, deberemos hacerlo tratándolo como un objeto.

La declaración principal de un objeto, se realiza con la instrucción **New**.

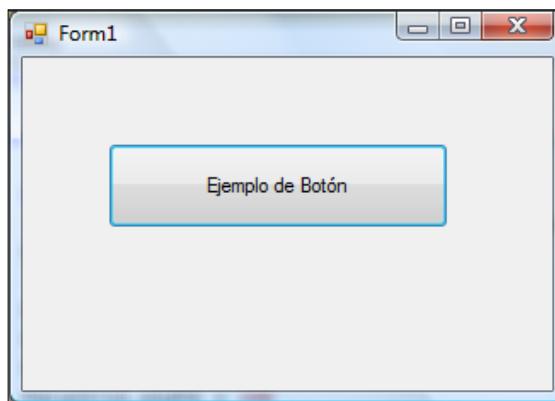
El siguiente ejemplo de código, crea un control **Button** en tiempo de ejecución.

Código

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.Load
    ' Declaramos el objeto Button
    Dim MiControl As New Button
    ' Declaramos un nombre al control (si queremos)
    MiControl.Name = "btn1"
    ' Cambiamos algunas de sus propiedades
    MiControl.Text = "Ejemplo de Botón"
    MiControl.Top = 50
    MiControl.Left = 50
    MiControl.Width = 200
    MiControl.Height = 50
    ' Añadimos el control al Formulario
    Me.Controls.Add(MiControl)
End Sub
```

En la figura 1 podemos ver el resultado en ejecución del código escrito anteriormente.

Para ejecutar nuestra aplicación, presionaremos el botón **F5**, que es la forma habitual de ejecutar una aplicación.



Creación de un control en tiempo de ejecución en Visual Basic 2010

Figura 1

Otra de las características de los controles, es la posibilidad de manipular los controles en tiempo de ejecución. Sin embargo, en

nuestro caso, vamos a modificar la propiedad **Text** del control **Button** que hemos insertado en tiempo de ejecución.

Para hacer esto, lo más habitual es poner el nombre del control, su propiedad y el valor correspondiente. En nuestro caso, el código quedaría como el que se indica a continuación:

Código

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.Load
    ' Declaramos el objeto Button
    Dim MiControl As New Button
    ' Declaramos un nombre al control (si queremos)
    MiControl.Name = "btn1"
    ' Cambiamos algunas de sus propiedades
    MiControl.Text = "Ejemplo de Botón"
    MiControl.Top = 50
    MiControl.Left = 50
    MiControl.Width = 200
    MiControl.Height = 50
    ' Añadimos el control al Formulario
    Me.Controls.Add(MiControl)
    ' Modificamos la propiedad Text del control insertado
    btn1.Text = "Otro texto"
End Sub
```

Analizando este código, parece estar bien escrito, pero al presionar **F5** para ejecutar nuestro proyecto, nos encontramos con que Visual Basic 2010 nos muestra un error. Nos indica que **btn1 no está declarado**. ¿Qué ocurre?. Al buscar la clase de ensamblados de la aplicación, el control **Button** de nombre **btn1** no existe, por lo que Visual Basic 2010 detecta que debemos declarar el control, sin embargo y en nuestro caso, esta declaración la hacemos en tiempo de ejecución. ¿Cómo acceder a la propiedad **Text** del control **Button** creado en tiempo de ejecución?

El siguiente código resuelve esta duda:

Código

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.Load
    ' Declaramos el objeto Button
    Dim MiControl As New Button
    ' Declaramos un nombre al control (si queremos)
    MiControl.Name = "btn1"
    ' Cambiamos algunas de sus propiedades
    MiControl.Text = "Ejemplo de Botón"
    MiControl.Top = 50
    MiControl.Left = 50
    MiControl.Width = 200
    MiControl.Height = 50
    ' Añadimos el control al Formulario
    Me.Controls.Add(MiControl)
    ' Modificamos la propiedad Text del control insertado
    CType(Me.FindForm.Controls("btn1"), Button).Text = "Otro texto"
End Sub
```

Básicamente, utilizamos una conversión explícita del objeto devuelto por la búsqueda realizada en los controles del formulario principal, que será un control **Button** de nombre **btn1**, para poder así, cambiar la propiedad **Text** de este control.

Aún así, también podríamos haber accedido a la propiedad **Text** del control mediante otra acción complementaria, como se muestra en el siguiente código:

Código

```
' Declaramos el objeto Button
Dim MiControl As New Button

Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.Load
    ' Declaramos un nombre al control
    MiControl.Name = "btn1"
    ' Cambiamos algunas de sus propiedades
    MiControl.Text = "Ejemplo de Botón"
    MiControl.Location = New Point(50, 50)
    MiControl.Size = New Size(200, 50)
    ' Añadimos el control al Formulario
    Me.Controls.Add(MiControl)
End Sub

Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button1.Click
    MiControl.Text = "Otro texto"
End Sub
```

¡Ojo!

Tenga en cuenta que esta acción se puede realizar si la declaración del objeto **Button** está dentro del mismo ámbito de llamada de la propiedad **Text**. Por esa razón, hemos sacado la declaración **MiControl** del objeto **Button** fuera del procedimiento de creación dinámica del control, pero tenga en cuenta también, que en este caso, tendremos declarada siempre en memoria la variable **MiControl**. El uso de la conversión (**Button**) es siempre más seguro.

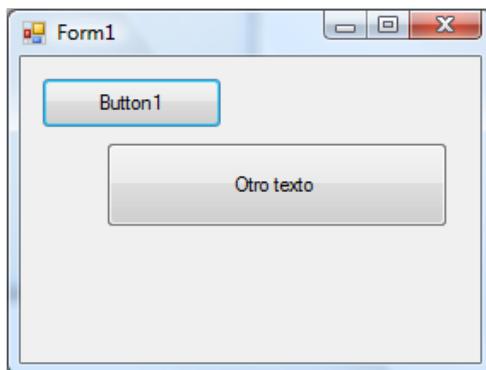
El caso anterior utilizando únicamente la conversión explícita, (**Button**), quedaría como se detalla a continuación (para estos dos ejemplos, he añadido además un control **Button** al formulario *Windows*, desde el cuál cambiaremos la propiedad **Text** del control **Button** creado en tiempo de ejecución):

Código

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.Load
    ' Declaramos el objeto Button
    Dim MiControl As New Button
    ' Declaramos un nombre al control (si queremos)
    MiControl.Name = "btn1"
    ' Cambiamos algunas de sus propiedades
    MiControl.Text = "Ejemplo de Botón"
    MiControl.Location = New Point(50, 50)
    MiControl.Size = New Size(200, 50)
    ' Añadimos el control al Formulario
    Me.Controls.Add(MiControl)
End Sub

Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button1.Click
    CType(Me.FindForm.Controls("btn1"), Button).Text = "Otro texto"
End Sub
```

Este ejemplo en ejecución es el que se muestra en la figura 2.



Ejecución del ejemplo anterior en Visual Basic 2010

Figura 2

Antes de continuar con el siguiente apartado en el que aprenderemos a crear un array de controles, comentaré que a la hora de posicionar un determinado control en un formulario *Windows*, lo podemos hacer con las propiedades **Top** y **Left**, o bien, utilizando la propiedad **Location**. Lo mismo ocurre con las propiedades **Width** y **Height** que pueden ser sustituidas por la propiedad **Size**.

El mismo ejemplo de creación de controles en tiempo de ejecución utilizando el método **Location** y **Size**, quedaría como se detalla a continuación:

Código

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.Load
    ' Declaramos el objeto Button
    Dim MiControl As New Button
    ' Declaramos un nombre al control (si queremos)
    MiControl.Name = "btn1"
    ' Cambiamos algunas de sus propiedades
    MiControl.Text = "Ejemplo de Botón"
    ' Propiedad Location
    MiControl.Location = New Point(50, 50)
    ' Propiedad Size
    MiControl.Size = New Size(200, 50)
    ' Añadimos el control al Formulario
    Me.Controls.Add(MiControl)
End Sub
```

Hasta ahora hemos visto como crear controles en tiempo de ejecución, pero en muchas ocasiones, nos es útil no sólo crear un control en tiempo de ejecución, sino relacionarlo con un evento.

En nuestro ejemplo del control **Button**, cuando hacemos clic sobre el control, no ocurre nada, y sería interesante mostrar un

mensaje o realizar una acción determinada.

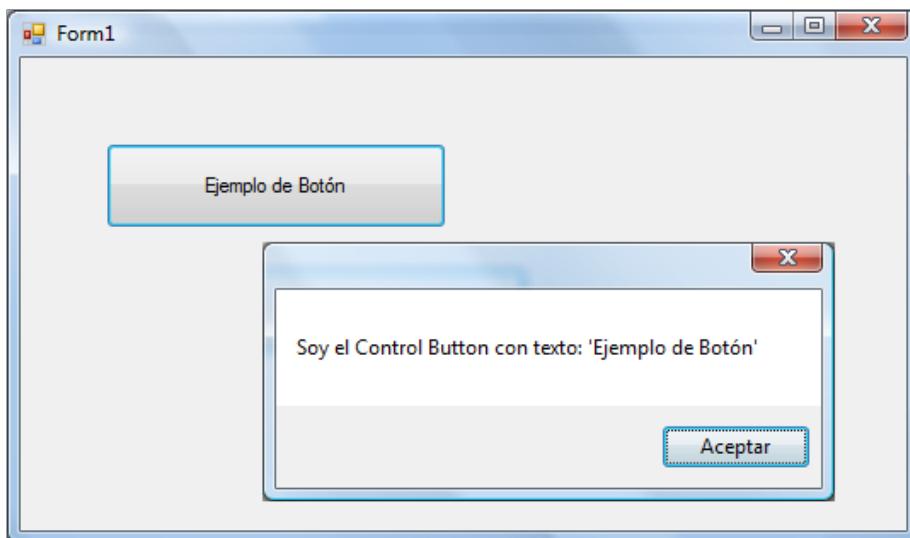
Para hacer esto, deberemos utilizar el método **AddHandler** para asignar un evento al control creado dinámicamente. El código de nuestro ejemplo *mejorado* es el que se detalla a continuación:

```
Código

Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.Load
    ' Declaramos el objeto Button
    Dim MiControl As New Button
    ' Declaramos un nombre al control (si queremos)
    MiControl.Name = "btn1"
    ' Cambiamos algunas de sus propiedades
    MiControl.Text = "Ejemplo de Botón"
    MiControl.Location = New Point(50, 50)
    MiControl.Size = New Size(200, 50)
    ' Añadimos el control al Formulario
    Me.Controls.Add(MiControl)
    ' Añadimos el evento Click al control creado dinámicamente
    AddHandler MiControl.Click, AddressOf btn1Click
End Sub

Public Sub btn1Click(ByVal Sender As Object, ByVal e As System.EventArgs)
    ' Mostramos un Mensaje
    MessageBox.Show("Soy el Control Button con texto: '" + CType(CType(Sender, Button).Text, String) + "'")
End Sub
```

Nuestro ejemplo en ejecución es el que se muestra en la figura 3.



Ejecución del ejemplo con asociación de evento desarrollado en Visual Basic 2010

Figura 3

Lección 3: Trabajo con controles

- Dominando los controles en el entorno de trabajo
 - Creación de controles en tiempo de ejecución
- Creación de una matriz de controles**
- Creación de controles nuevos
 - Otras consideraciones

Módulo 3 - Capítulo 3

3. Creación de una matriz de controles

En el capítulo anterior, hemos visto como crear controles en tiempo de ejecución e incluso como asociar un evento a un control creado dinámicamente en Visual Basic 2010, pero, ¿qué ocurre si queremos crear un array o matriz de controles?.

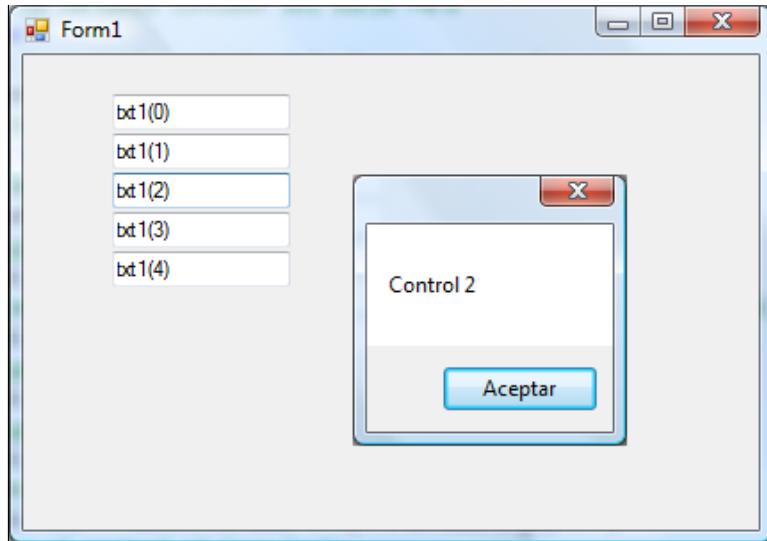
Podemos simular una matriz de controles de muchas formas. Yo en este ejemplo, aplicaré la siguiente forma de llevar a cabo esta tarea:

Código

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.Load
    ' Declaramos la variable contador del bucle Para
    Dim I As Byte
    ' Declaramos la variable contador del número de controles a crear
    Dim intNumControles As Byte = 5
    ' Iniciamos el bucle Para
    For I = 0 To CByte(intNumControles - 1)
        ' Declaramos el objeto TextBox
        Dim MiControl As New TextBox
        ' Le asignamos un nombre al control
        MiControl.Name = "txt1"
        ' Utilizamos la propiedad Tag para almacenar ahí el valor del control de la matriz virtual
        MiControl.Tag = I
        ' Le asignamos un tamaño en el Formulario Windows
        MiControl.Size = New Size(100, 20)
        ' Le asignamos una posición en el formulario Windows
        MiControl.Location = New Point(50, 22 * (I + 1))
        ' Le cambiamos la propiedad Text
        MiControl.Text = MiControl.Name + "(" + I.ToString() + ")"
        ' Añadimos el control al Formulario
        Me.Controls.Add(MiControl)
        ' Añadimos el evento Click al control creado dinámicamente
        AddHandler MiControl.Click, AddressOf txt1Click
    Next
End Sub

Public Sub txt1Click(ByVal Sender As Object, ByVal e As System.EventArgs)
    ' Mostramos un Mensaje
    MessageBox.Show("Control " + CType(Sender, TextBox).Tag.ToString())
End Sub
```

Nuestro ejemplo de demostración en ejecución es el que se puede ver en la figura 1.



Ejecución de la simulación de una matriz de controles

Figura 1

Obviamente, existen diferentes formas y técnicas de simular un array o matriz de controles. Sirva esta que hemos visto, como ejemplo de lo que se puede hacer, pero para nada se trata de una norma o regla fija.

Debemos destacar que en .NET no existe el concepto de array o matriz de controles ya que ésta, es una característica propia del lenguaje, pero que para ciertas ocasiones conviene conocer.

Lección 3: Trabajo con controles

- Dominando los controles en el entorno de trabajo
 - Creación de controles en tiempo de ejecución
 - Creación de una matriz de controles
- Creación de controles nuevos**
- Otras consideraciones

Módulo 3 - Capítulo 3

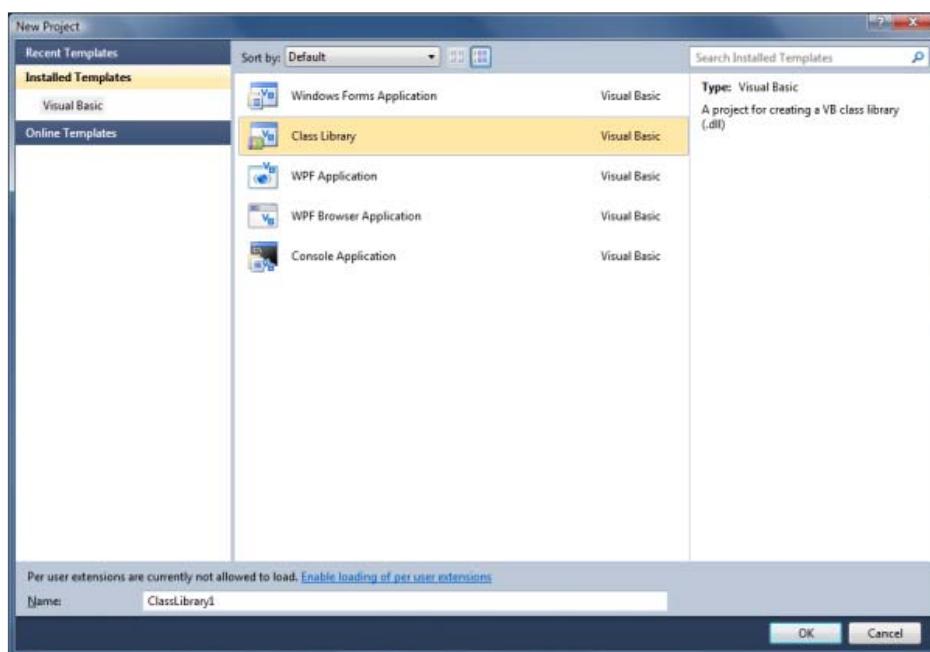
4. Creación de controles nuevos

Ya hemos visto la diferencia más genérica entre un componente y un control, pero aún no sabemos como desarrollar nuestros propios controles en Visual Basic 2010.

En primer lugar y antes de adentrarnos en la creación de nuestros propios controles con Visual Basic 2010, debo indicarle que debemos obviar todo lo relacionado con los ActiveX OCX y ActiveX en general.

En Visual Basic 2010, la palabra ActiveX ya no existe. El modelo de programación ha cambiado y por eso, los componentes y controles se generan ahora siguiendo otras normas que aprenderemos a utilizar de forma inmediata.

Iniciaremos *Visual Studio 2010* y seleccionaremos un proyecto de tipo **Biblioteca de clases**. En el nombre de proyecto, podemos indicarle el nombre que deseemos tal y como se muestra en la figura 1, y a continuación presionaremos el botón **OK**.



Selección de nuevo proyecto Biblioteca de clases en Visual Basic 2010

Figura 1

La diferencia mayor que reside entre el desarrollo de componentes y controles en .NET, es que en lugar de heredar de la clase **Component** como en el caso de la creación de los componentes, se ha de heredar de la clase **Control** o **System.Windows.Forms.UserControl**.

El tipo de proyecto seleccionado no posee por defecto como ocurre con los controles ActiveX, de la superficie contenedora sobre la cuál podremos insertar otros controles o realizar las acciones que consideremos pertinentes para crear así nuestro control personalizado. En este caso, la superficie contenedora la deberemos crear añadiendo las referencias necesarias a nuestro programa.

Haga clic con el botón secundario del ratón sobre el proyecto o solución de la ventana **Explorador de soluciones** y a continuación, seleccione la opción **Propiedades** del menú emergente.

A continuación, agregue las referencias a las librerías de clases **System.Drawing** y **System.Windows.Forms**.

Por último, escriba las siguientes instrucciones básicas.

Código

```

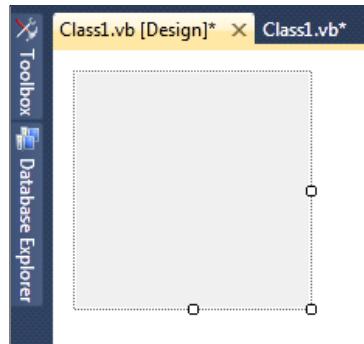
Imports System.ComponentModel
Imports System.Windows.Forms
Imports System.Drawing

Public Class Class1
    Inherits UserControl

End Class

```

En este punto, nuestra clase habrá sido transformada en la clase contenedora de un control que es la que puede verse en la figura 2.

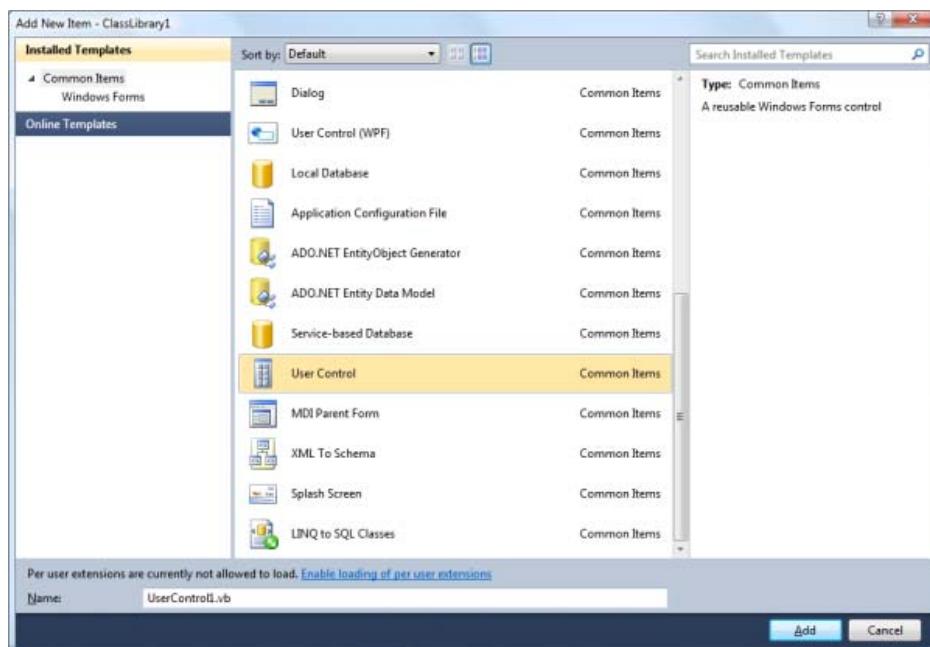


Superficie contenedora por defecto de un control

Figura 2

Aún así, si no quiere realizar esta acción, otra forma de tener lista la superficie contenedora del control, es la de eliminar la clase del proyecto y presionar el botón secundario del ratón sobre la ventana del *Explorador de soluciones* y seleccionar la opción de **Agregar > Nuevo elemento...** del menú emergente.

De las opciones que salen, (ver figura 2B), deberíamos seleccionar entonces la plantilla **Control de usuario**.



Seleccionar un control de usuario

Figura 2B

Sin más dilación, añadiremos sobre la superficie del control contenedor, (al que habremos cambiado el nombre a **MiControl**), dos controles **Label** y dos controles **TextBox**.

Debido a que al añadirle los controles tendremos mucho código que simplemente sirve para el diseño del interfaz de usuario, podemos hacer lo que el propio diseñador hace: dividir la clase en dos partes o clases parciales, de esta forma conseguiremos el propósito de las clases parciales, (al menos en cuanto al diseño de formularios y controles de usuario se refiere), que es, como ya comentamos anteriormente, separar el código de diseño del código que nosotros vamos a escribir.

Para ello, añadiremos un nuevo elemento del tipo Class, al que le daremos el nombre **MiControl.Designer.vb**, y en esa clase pegaremos el código que actualmente hay en la clase, pero añadiendo la partícula Partial antes de Class, con idea de que el compilador de Visual Basic 2010 sepa que nuestra intención es crear una clase parcial.

Realmente es casi lo mismo que tenemos actualmente, tan solo tendremos que añadir la instrucción *partial*, (no hace falta que esté declarada como *public*, ya que en el otro "trozo" tendremos declarada esta clase como pública), además del constructor de la clase, desde el que llamaremos al método que se encarga de toda la inicialización: **InitializeComponent()**.

A continuación, escribiremos el siguiente código, en el archivo que teníamos originalmente (MiControl.vb):

Código

```
Imports System.ComponentModel
Imports System.Windows.Forms

Public Class MiControl

    Private _Acceso As Boolean

    <Category("Acceso"), Description("Indica si se permite o no el acceso"), DefaultValue("False"), [ReadOnly](True)> _
    Public Property Acceso() As Boolean
        Get
            Return _Acceso
        End Get
        Set(ByVal Val As Boolean)
            _Acceso = Val
        End Set
    End Property

    Private Sub UserControl1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.Load
        _Acceso = False
    End Sub

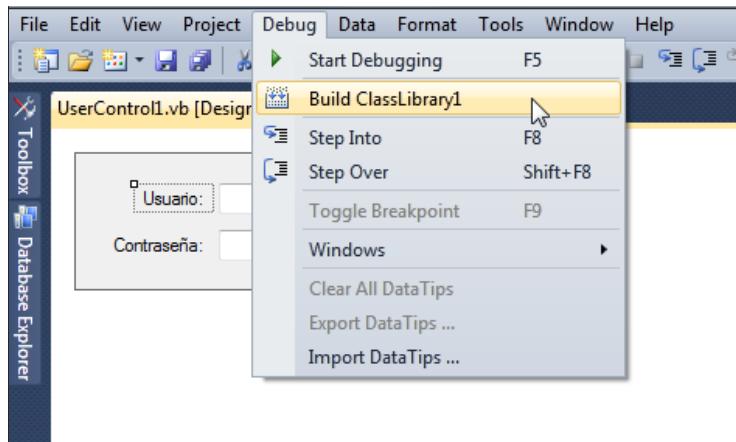
    Public Sub Validar()
        If TextBox1.Text = "ejemplo" And TextBox2.Text = "ejemplo" Then
            _Acceso = True
        Else
            _Acceso = False
        End If
    End Sub

End Class
```

Observando el código, vemos que hemos creado una propiedad **Acceso** que nos permitirá saber si un usuario y contraseña han sido validadas o no.

Ahora nuestro control, está listo ya para ser compilado y probado en una aplicación Windows.

Para compilar nuestro control, haga clic en el menú **Generar > Generar Solución** como se muestra en la figura 3.



Opción para compilar nuestro control

Figura 3

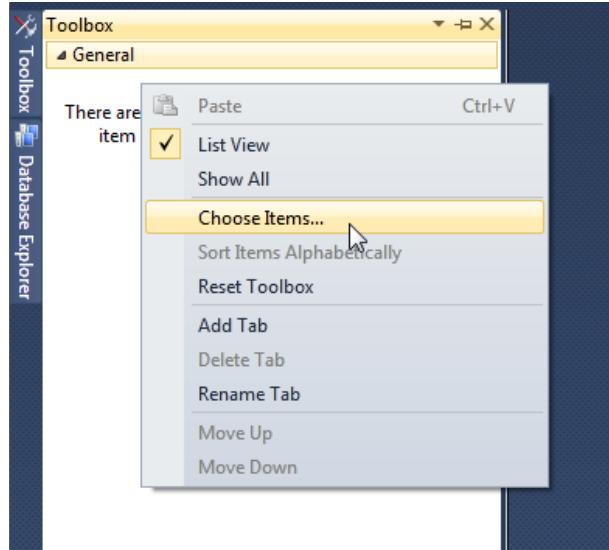
A continuación añada un nuevo proyecto **Aplicación para Windows** a la solución. Para añadir un proyecto a la solución actual, tendremos que ir al menú **Archivo** y seleccionar **Nuevo proyecto**, cuando nos muestre el cuadro de diálogo, indicaremos que lo añada a la misma solución.

Establézcalo como proyecto inicial de la solución.

Acuda a la barra de herramientas y busque el control que hemos creado y compilado para insertarlo en el formulario Windows.

Sino aparece en la barra de herramientas, (que debería aparecer en el grupo **ClassLibrary1**), deberá añadirla de la siguiente manera.

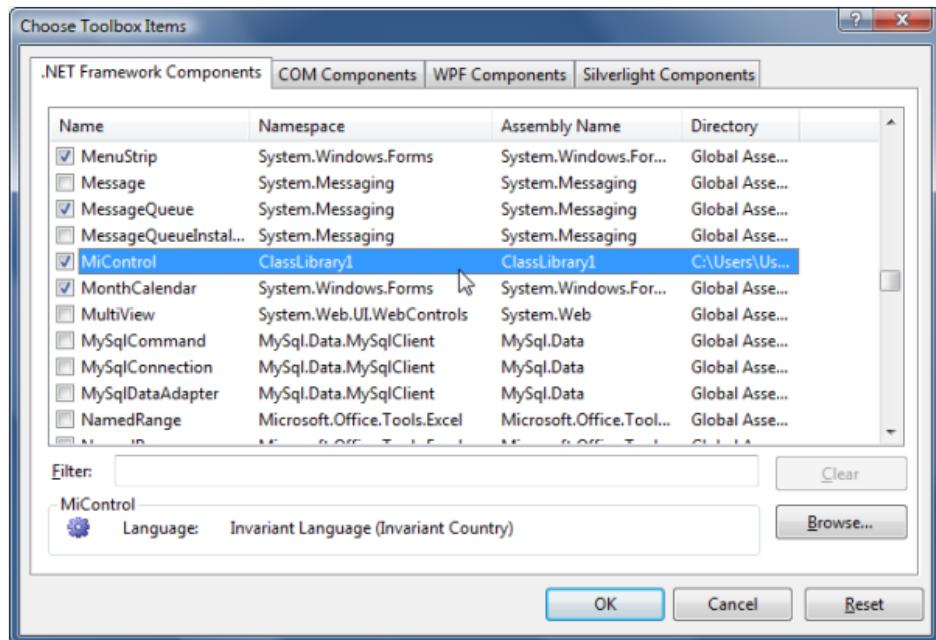
Haga clic sobre la barra de herramientas y seleccione la opción **Elegir Elementos...** del menú emergente que aparece en la figura 4.



Opción para añadir un control o componente a la barra de herramientas

Figura 4

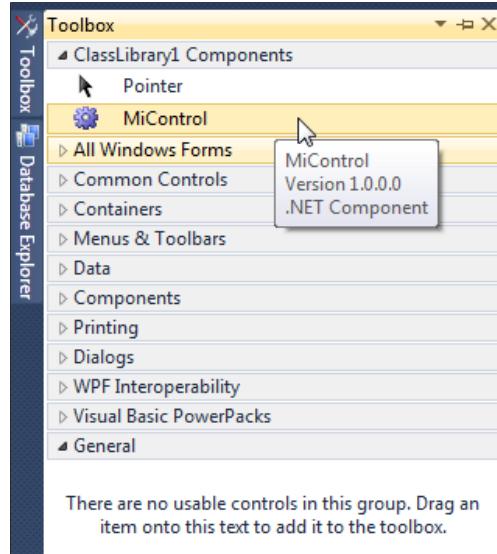
Aparecerá una ventana para buscar el control ya compilado en el disco duro. presionaremos el botón **Examinar...** y buscaremos nuestro control para seleccionarlo. Una vez hecho esto, tal y como se indica en la figura 5, haremos clic sobre el botón **OK**.



Selección del control compilado anteriormente

Figura 5

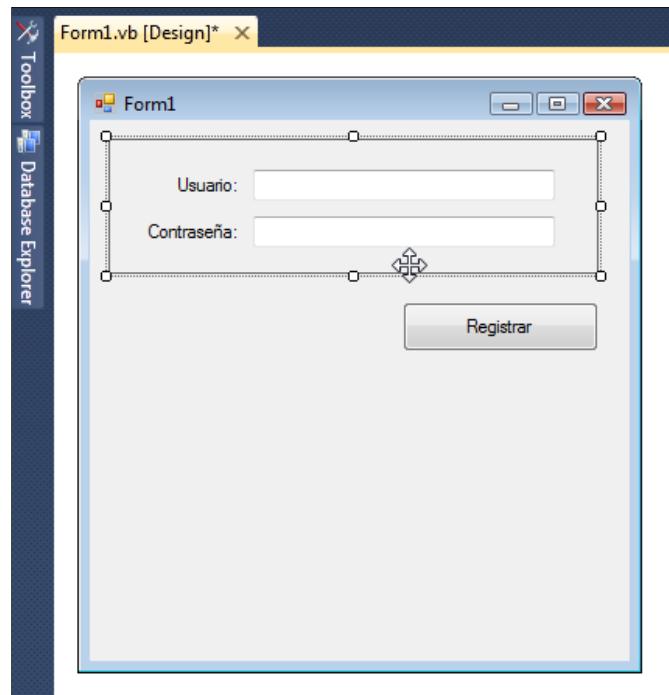
Nuestro control quedará insertado en la barra de herramientas como se muestra en la figura 6.



Control insertado en la barra de herramientas

Figura 6

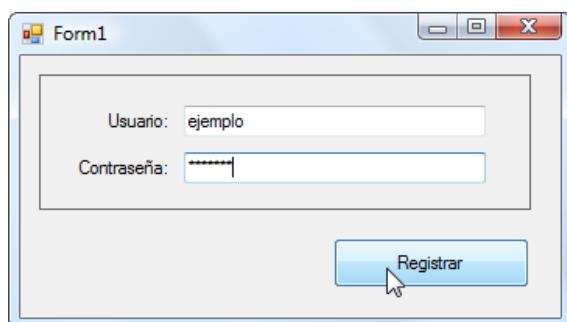
Para insertar el control en el formulario, haremos doble clic sobre él. Este quedará dispuesto en el formulario Windows como se indica en la figura 7.



Control insertado en el formulario Windows de prueba

Figura 7

Nuestro formulario Windows de prueba en ejecución con el control insertado en él, es el que puede verse en la figura 8.



Formulario Windows de prueba en ejecución con el control insertado

Figura 8

Como ha podido comprobar, la creación de controles en Visual Basic 2010, tampoco requiere de una gran habilidad o destreza, y su similitud con la creación de componentes es enorme. De hecho, todo se reduce en el uso y programación de una clase con la salvedad de que dentro de esa clase, indicamos si se trata de una clase como tal, la clase de un componente o la clase de un control.



Lección 3: Trabajo con controles

- Dominando los controles en el entorno de trabajo
 - Creación de controles en tiempo de ejecución
 - Creación de una matriz de controles
 - Creación de controles nuevos
- Otras consideraciones**

Módulo 3 - Capítulo 3

5. Otras consideraciones

Controles contenedores

Cuando trabajamos con controles, surgen muchas veces muchas dudas de carácter habitual. Definamos que hay dos tipos de controles, los controles contenedores y los controles no contenedores.

La diferencia entre ambos es que los controles contenedores, pueden como su propia palabra dice, contener otros controles dentro de éste. Los controles no contenedores no pueden.

Un claro ejemplo de control contenedor, es el control **TabControl**, el cuál permite incluir dentro de él otros controles, incluso controles contenedores a su vez.

Un ejemplo de control no contenedor es el control **Button** por ejemplo. Si intenta poner un control **TextBox** encima de un control **Button**, observará que no puede.

Cuando un control se inserta dentro de un contenedor (no olvidemos que el formulario o Windows es otro contenedor), éste control queda alojado dentro de su contenedor, de tal manera que si movemos el contenedor, estaremos moviendo también el *contenido* de éste.

El problema sin embargo cuando trabajamos con controles, viene cuando deseamos recorrer los controles de un formulario. Esto es muy habitual cuando deseamos borrar el contenido de todos los controles **TextBox** de un formulario Windows por ejemplo.

Dentro de un formulario o supuesto que tengamos 4 controles **TextBox** insertados dentro de él, y luego un botón que nos permita eliminar el contenido de cada caja de texto, tal y como se indica en la figura 1, deberíamos escribir un código similar al que veremos a continuación:

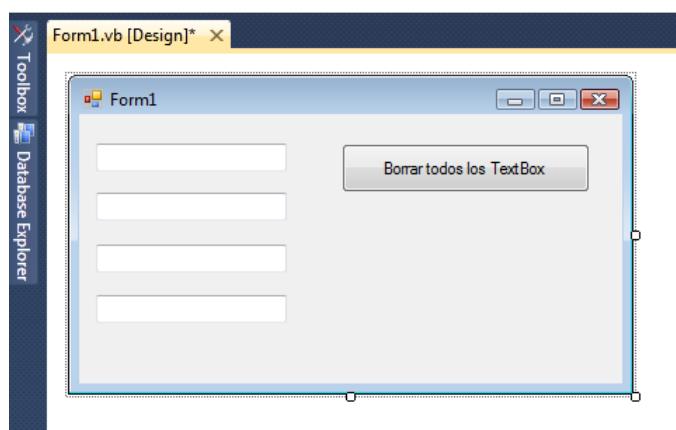


Figura 1

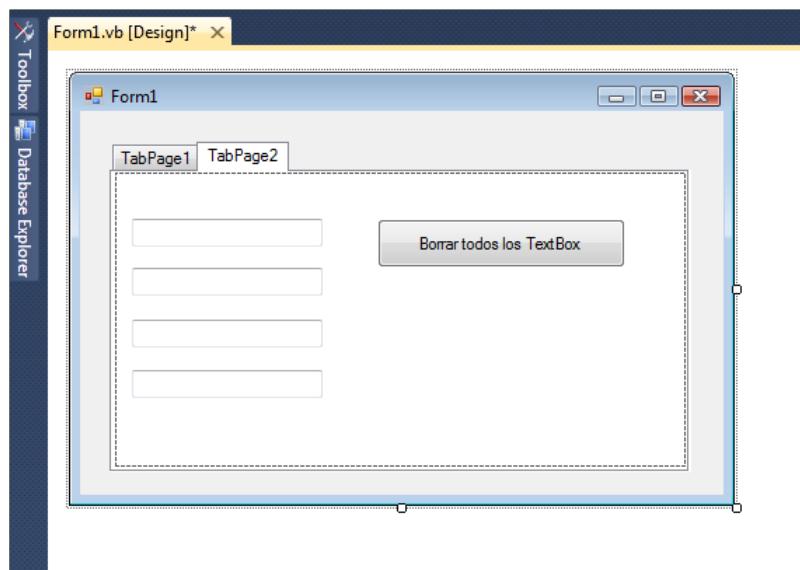
Código

```
Public Class Form1
    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button1.Click
        Dim obj As Object
        For Each obj In Controls
            If TypeOf obj Is TextBox Then
                CType(obj, TextBox).Text = ""
            End If
        Next
    End Sub
End Class
```

Observando este código, vemos que lo que hacemos no es otra cosa que recorrer los objetos como controles dentro del formulario y miramos si se trata de un control de tipo **TextBox**. En este caso, modificamos la propiedad **Text** para dejarla en blanco. Ejecute este ejemplo con la tecla de función **F5**, escriba algo en las cajas de texto y pulse el botón. Observará que logramos conseguir nuestro objetivo de borrar el contenido de todos los controles **TextBox**.

A continuación, lo que haremos será añadir un control **TabControl** a nuestro formulario Windows, y dentro de él, añadiremos los cuatro controles **TextBox** y el

control **Button** anteriormente comentados, tal y como se muestra en la figura 2.



Controles insertados dentro de un control contenedor como el control **TabControl**

Figura 2

El código anterior no hará falta modificarlo, por lo que ejecutaremos la aplicación nuevamente presionando el botón **F5**, escribiremos algo de texto en las cajas de texto y pulsaremos el botón como hicimos antes. Observaremos que en este caso, los controles **TextBox** no se han quedado en blanco como antes.

El contenedor no es el formulario Windows, sino el control **TabControl**. Dicho control es tratado en el bucle anterior del control como control dependiente del formulario Windows, pero los controles **TextBox** y el propio control **Button**, quedan encerrados dentro de un ámbito contenedor diferente.

Para solventar este problema, deberemos recorrer los controles del contenedor. Una forma de solventar esto es de la siguiente manera:

```
Código

Public Class Form1
    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button1.Click
        For I As Byte = 0 To TabControl1.TabPages.Count - 1
            For Each obj As Object In TabControl1.TabPages(I).Controls
                If TypeOf obj Is TextBox Then
                    CType(obj, TextBox).Text = ""
                End If
            Next
        Next
    End Sub
End Class
```

De esta manera, recorreremos todos los controles que residen en las páginas de un control **TabControl** y si se trata de un control **TextBox**, modificaremos la propiedad **Text** correspondiente.

Por esta razón, cuando trabajamos con controles dentro de un formulario y queremos actuar sobre un conjunto de controles determinado, debemos tener en cuenta entre otras cosas, si se trata de un conjunto de controles o un control simplemente, se encuentra dentro de un control contenedor o fuera de él.

Sin embargo, para resolver el problema de recorrer todos los controles de un formulario, estén o no dentro de un control contenedor, lo mejor es ampliar la función anterior y hacerla recursiva, de modo que permite recorrer todos los controles del formulario, estén o no dentro de un contenedor.

El siguiente código, refleja un ejemplo de como hacer esto posible.

En el mismo proyecto, podemos añadir nuevos **TextBox** a la otra ficha del **TabControl**, además uno en el propio formulario, (para que no esté dentro del **TabControl**).

Rellene algo de contenido en los controles **TextBox** y ejecute el código:

```
Código

Public Class Form1
    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button1.Click
        VaciarTextBox(Me)
    End Sub

    Private Sub VaciarTextBox(ByVal Parent As Control)
        For Each obj As Control In Parent.Controls
            If obj.Controls.Count > 0 Then
                VaciarTextBox(obj)
            End If
            If TypeOf obj Is TextBox Then
                CType(obj, TextBox).Text = ""
            End If
        Next
    End Sub
End Class
```

Smart Tags

Otras de las novedades del diseñador de formularios de Visual Studio 2010, es el hecho de poder utilizar las denominadas *Smart Tags*.

Si usted es usuario de *Office* en sus últimas versiones, sabrá a qué me refiero. Las **Smart Tags** nos indican trucos y consejos directamente a la cual se puede acceder

desde Visual Basic 2010 de forma rápida, ahorrándonos tiempo y aportándonos productividad en nuestros desarrollos.

Las **Smart Tags** aparecen en Visual Studio 2010 tanto en el lado del diseño del formulario Windows como en la parte correspondiente del código fuente escrito.

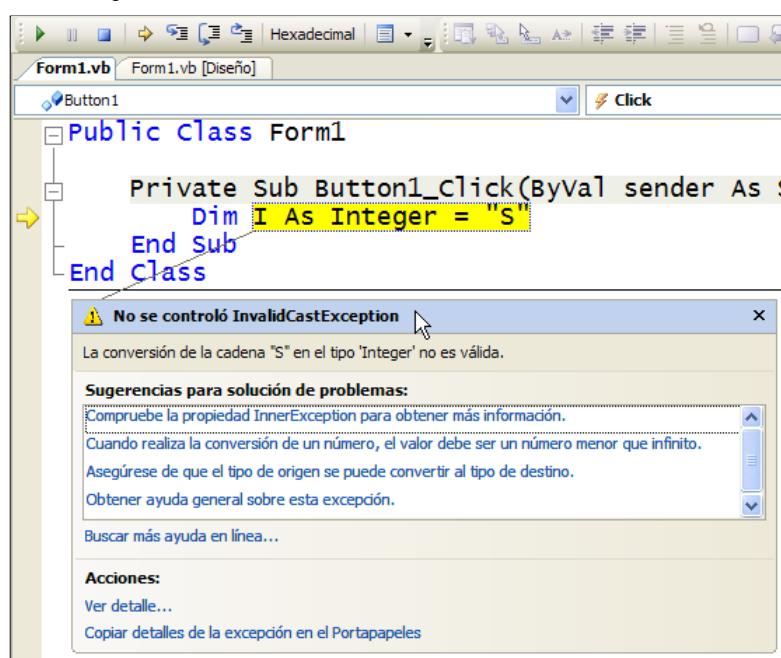
En el caso de los controles insertados en un formulario Windows, estas **Smart Tags** aparecerán por lo general en el borde superior derecho del control y se representará con un icono parecido a este

Si hacemos clic sobre ese ícono, aparecerá una ayuda gráfica con varias opciones que permite realizar las acciones más comunes sobre el control seleccionado, tal y como se indica en la figura 3 para el caso del control **TabControl**.



Figura 3

En el caso del código, las **Smart Tags** poseen un comportamiento ligeramente diferente dependiendo del caso. Un ejemplo habitual es el hecho de declarar una variable incorrecta tratando de trabajar con ella. Cuando ejecutamos nuestra aplicación, el depurador se detiene en la declaración incorrecta y nos muestra una ventana de depuración, como se indica en la figura 4.

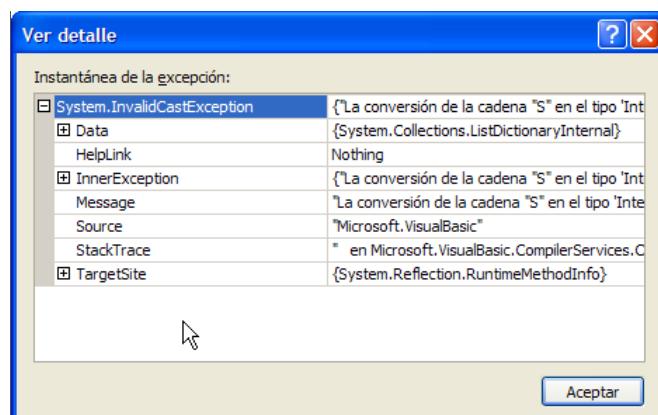


Ventana de depuración detallada

Figura 4

La ventana emergente de depuración nos indica no sólo el tipo de error encontrado, sino que además, nos indica las posibles soluciones para solventar el problema. Estos consejos, sugerencias o *Tips*, nos ofrecen una ventaja sobre nuestros desarrollos, proporcionándonos una rápida solución a un problema y permitiéndonos ahorrar mucho tiempo resolviendo el porqué del problema.

Si en la ventana de depuración emergente presionamos sobre la opción **Ver detalle...**, obtendremos una nueva ventana como la que se muestra en la figura 5.



Ventana de detalles del error

Figura 5

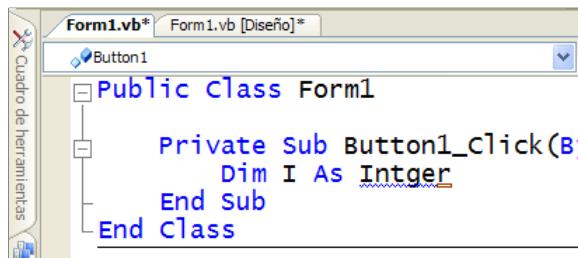
De esta manera, conoceremos los detalles del problema detectado por Studio 2010 permitiéndonos corregirlo cómodamente.

Otra posibilidad de las **Smart Tags** es la detección de una declaración de variable o cualquier parte de código, no esperada. Como ejemplo, declararemos una variable de la forma:



```
Código  
Dim I As intger
```

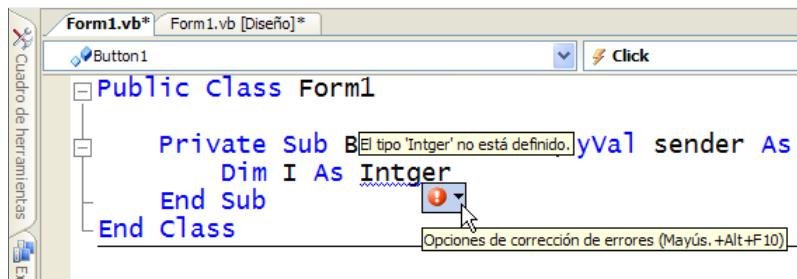
Queríamos decir *Integer*, pero con las prisas he escrito *intger*. En ese instante, Visual Studio 2010 se ha percatado de que la declaración no es correcta y antes de que me pueda dar cuenta de que lo he escrito incorrectamente, el entorno me indica el error marcándolo con un subrayado característico como se indica en la figura 6.



Subrayado de Visual Studio 2010 indicando un error en la instrucción de código escrita

Figura 6

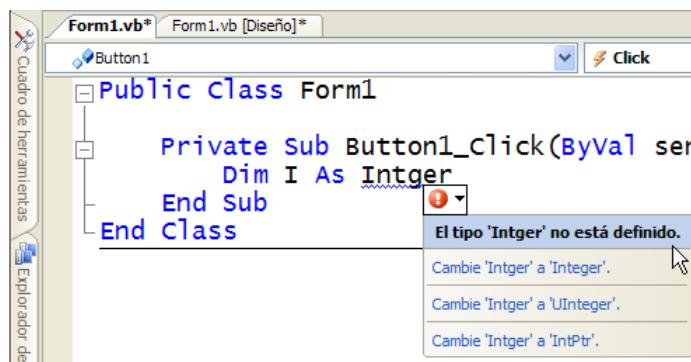
Cuando acercamos el puntero del mouse sobre el recuadro de color rojo de la declaración errónea, aparece una información adicional junto a un icono parecido a este  y una información emergente que se muestra en la figura 7.



Información sobre el detalle del error encontrado

Figura 7

Si hacemos clic sobre el icono , accederemos a una información adicional que nos muestra un conjunto de sugerencias. En nuestro caso es la primera de ellas como se indica en la figura 8, por lo que al hacer clic sobre ella, la declaración se actualiza directamente por la declaración correcta:



Opciones o sugerencias de resolución del error

Figura 8

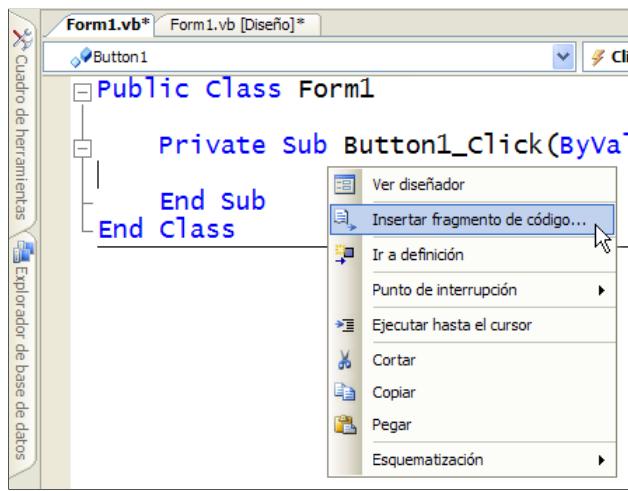


```
Código  
Dim I As Integer
```

Generación de código rápido

Otra consideración a tener en cuenta a la hora de trabajar con código y a la hora por lo tanto, de desarrollar nuestros propios controles, componentes, clases, etc., es la ayuda que nos proporciona el entorno cuando deseamos escribir determinadas funciones rápidas de código que son bastante habituales.

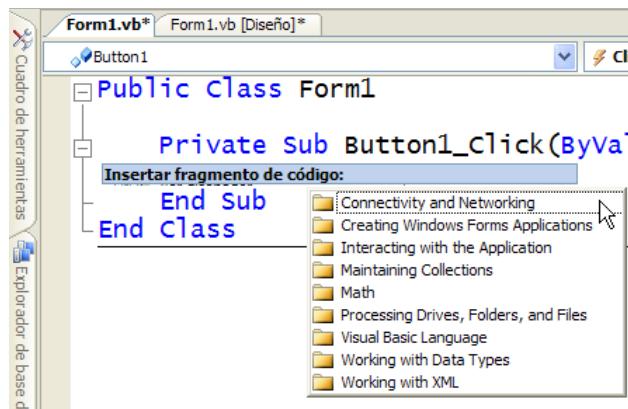
Póngase sobre el código de Visual Basic 2010 y haga clic con el botón secundario del mouse y seleccione la opción **Insertar fragmento de código...** como se indica en la figura 9.



Opción de recortes de código de Visual Studio 2010

Figura 9

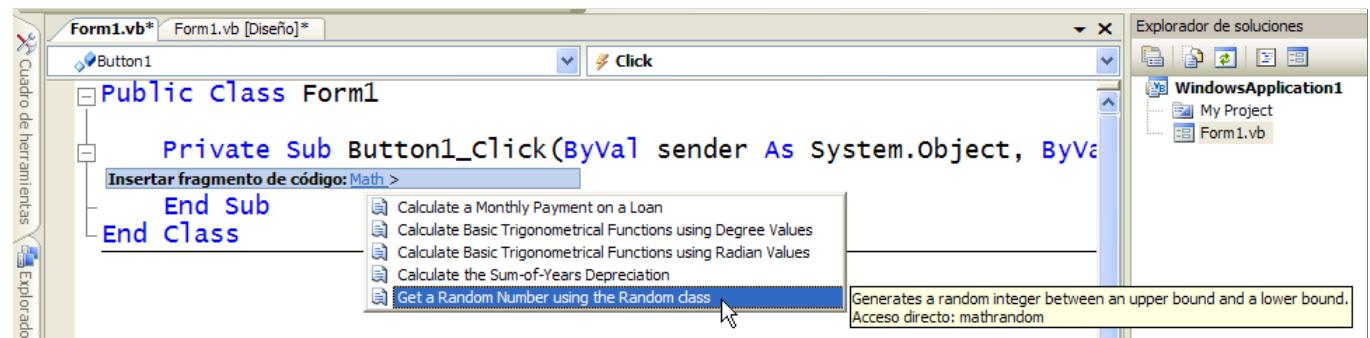
Aparecerá una ventana emergente como la que se muestra en la figura 10 dentro de la cuál, podremos seleccionar la carpeta que contiene un amplio conjunto de funciones, para insertarla rápidamente a nuestro código.



Ventana emergente de carpetas de recortes de código

Figura 10

Como ejemplo, si se pulsa con los cursos sobre **Math**, pulse *Enter*, seleccione *Get a Random Number using the Random class* como se indica en la figura 11 y vuelva a presionar *Enter*. El código quedará insertado a nuestro proyecto de Visual Basic 2010 como se muestra a continuación:



Selección de un recorte de código

Figura 11

```

Código
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button1.Click
    Dim generator As New Random
    Dim randomValue As Integer
    ' Generates numbers between 1 and 5, inclusive.
    randomValue = generator.Next(1, 6)
End Sub

```

Estos atajos, aumentan la productividad en el desarrollo de aplicaciones de forma abismal. Tengamos en cuenta, que muchas de las operaciones que realizamos los desarrolladores, son acciones repetitivas, funciones generales o funciones habituales que reducen nuestro rendimiento y productividad y nos permite cometer errores. Con estas acciones, evitamos todo esto.

FAQ:

¿Cómo añadir nuestros propios recortes o *Snippets*?

Indudablemente, una cosa es usar los recortes que nos proporciona Visual Basic 2010 y otra muy diferente añadir los nuestros al entorno. Para hacer esto último, primero debemos escribir nuestros recortes de código, la mejor forma es fijarse cómo están hechos los que se incluyen con Visual Studio 2010, para localizarlos, abra una ventana de Windows y sitúese sobre la carpeta de instalación del entorno de Visual Studio 2010 y concretamente en la carpeta de Snippets. En mi caso, la ruta es C:\Archivos de programa\Microsoft Visual Studio 10.0\VB\Snippets\3082.

Para añadir nuevos recortes, lo podemos hacer desde el menú Herramientas > Administrador de fragmentos de código

Lección 4: Trabajo con imágenes y gráficos

- Gráficos 3D
- Gráficos 2D
- Dibujando líneas con GDI+
- Dibujando curvas con GDI+
- Dibujando cadenas de texto con GDI+
- Otras consideraciones

Introducción

En este capítulo veremos las partes más generales en el uso y generación de imágenes y gráficos con Visual Basic 2010.

Pasaremos de puntillas sobre el uso de DirectX para la generación de gráficos 3D y nos adentraremos un poco más profundamente, en el desarrollo de gráficos e imágenes 2D con GDI+.

Módulo 3 - Capítulo 4

- 1. [Gráficos 3D](#)
- 2. [Gráficos 2D](#)
- 3. [Dibujando líneas con GDI+](#)
- 4. [Dibujando curvas con GDI+](#)
- 5. [Dibujando cadenas de texto con GDI+](#)
- 6. [Otras consideraciones](#)

Lección 4: Trabajo con imágenes y gráficos

Gráficos 3D

- Gráficos 2D
- Dibujando líneas con GDI+
- Dibujando curvas con GDI+
- Dibujando cadenas de texto con GDI+
- Otras consideraciones

Módulo 3 - Capítulo 4

1. Gráficos 3D

Para trabajar con imágenes y gráficos en 3D, deberemos utilizar DirectX, ya que dentro de la plataforma .NET de Microsoft, no tenemos la posibilidad de crear representaciones 3D.

Esto significa por otra parte, que si desarrollamos una aplicación con representación 3D con DirectX, deberemos distribuir también las librerías DirectX en la máquina en la que se ejecute nuestra aplicación. Es decir, no bastará con distribuir Microsoft .NET Framework.

Pero DirectX no nos permite sólo crear gráficos e imágenes en 3D, también podemos crear imágenes en 2D, pero lo más normal en este último caso y salvo que no requeramos un uso continuado de DirectX, será utilizar en su lugar GDI+, como veremos más adelante.

Para trabajar con DirectX en Visual Basic 2010, deberemos añadir una referencia al proyecto con la librería o librerías COM de DirectX, eso sí trabajamos con DirectX 8 o anterior, ya que a partir de DirectX 9, Microsoft ha proporcionado un conjunto de clases y librerías que interactúan con .NET directamente, permitiéndonos ejecutar aplicaciones .NET con DirectX administrado.

En nuestro caso, lo mejor es realizar un ejemplo, no sin antes recordar, que debería descargar Microsoft DirectX SDK 9 e instalarlo en su sistema.

El SDK contiene ficheros de ayuda y documentación que le enseñará a crear aplicaciones con DirectX 9 en Visual Basic 2010.

FAQ:

¿Dónde puedo descargar Microsoft DirectX 9.0 SDK?

La última versión de DirectX SDK la encontrará en la página Web de MSDN de DirectX, especialmente creada para desarrolladores de aplicaciones y juegos.

[Descargas de Microsoft DirectX](#)

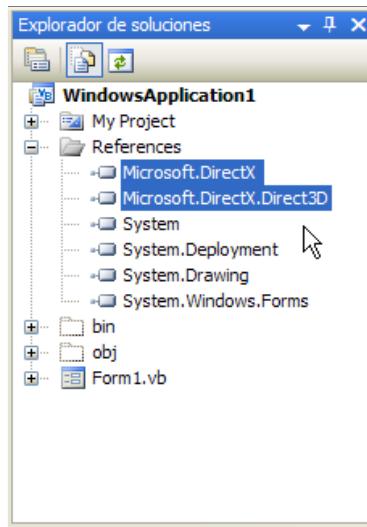
Sobre el soporte de DirectX y .NET:

Desde la versión de DirectX 10.0

no existe soporte directo para Microsoft .NET.

Es de esperar que Microsoft proporcione librerías con soporte en .NET en un futuro. Los ejemplos que aquí aparecen, son adecuados para DirectX 9.0.

Una vez que tenemos instalado en nuestro sistema las librerías de desarrollo de DirectX para .NET, iniciaremos una nueva aplicación Windows, añadiremos las referencias a las librerías Microsoft DirectX y Microsoft DirectX Direct3D tal y como se muestra en la figura 1 y las cuáles encontraremos normalmente en el directorio c:\windows\system32\



Referencias a DirectX añadidas a nuestro proyecto

Figura 1

A continuación, escribiremos el siguiente código:

Código

```

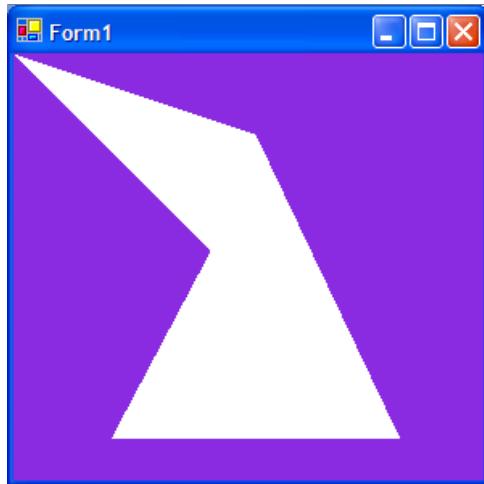
Imports Microsoft.DirectX
Imports Microsoft.DirectX.Direct3D

Public Class Form1
    Public Sub CreateDevice()
        Dim pp As New PresentParameters()
        pp.Windowed = True
        pp.SwapEffect = SwapEffect.Discard
        Dim dv As New Device(0, DeviceType.Hardware, Me, CreateFlags.SoftwareVertexProcessing, pp)
        Dim vertices(6) As CustomVertex.TransformedColored
        vertices(0).Position = New Vector4(Me.Width / 2.0F, 70.0F, 0.5F, 1.0F)
        vertices(1).Position = New Vector4(Me.Width - (Me.Width / 5.0F), Me.Height - (Me.Height / 5.0F), 0.5F, 1.0F)
        vertices(2).Position = New Vector4(Me.Width / 5.0F, Me.Height - (Me.Height / 5.0F), 0.5F, 1.0F)
        vertices(3).Position = New Vector4(Me.Width / 2.0F, 50.0F, 0.5F, 1.0F)
        vertices(4).Position = New Vector4(Me.Width - (Me.Width / 5.0F), Me.Height - (Me.Height / 5.0F), 0.5F, 1.0F)
        vertices(5).Position = New Vector4(Me.Width / 5.0F, Me.Height - (Me.Height / 5.0F), 0.5F, 1.0F)
        dv.Clear(ClearFlags.Target, System.Drawing.Color.BlueViolet, 2.0F, 0)
        dv.BeginScene()
        dv.VertexFormat = VertexFormats.Transformed
        dv.DrawUserPrimitives(PrimitiveType.TriangleList, 2, vertices)
        dv.EndScene()
        dv.Present()
    End Sub

    Protected Overrides Sub OnPaint(ByVal e As System.Windows.Forms.PaintEventArgs)
        MyBase.OnPaint(e)
        CreateDevice()
    End Sub
End Class

```

Este pequeño ejemplo demostrativo en ejecución del uso de DirectX desde nuestras aplicaciones Visual Basic 2010, es el que puede verse en la figura 2



Ejemplo de DirectX con Visual Basic 2010 en ejecución

Figura 2

Lección 4: Trabajo con imágenes y gráficos

• Gráficos 3D

Gráficos 2D

- Dibujando líneas con GDI+
- Dibujando curvas con GDI+
- Dibujando cadenas de texto con GDI+
- Otras consideraciones

Módulo 3 - Capítulo 4

2. Gráficos 2D

Las APIs de GDI+ corresponden a la evolución natural de las APIs y librerías GDI que se utilizaban en otros lenguajes de desarrollo. GDI+ no es otra cosa que un conjunto de clases desarrolladas para el entorno .NET y por esa razón, podemos entonces dibujar y crear representaciones gráficas en Visual Basic 2010.

Todas las clases GDI+, pueden ser localizadas a través del nombre de espacio **System.Drawing**. Así, podemos acceder a todas las posibilidades que nos ofrece GDI+ y las cuales veremos más adelante.

GDI+, no accede al hardware directamente, interactúa con los *drivers* de los dispositivos gráficos. Como particularidad, debemos saber que GDI+ está soportado por Win32 y Win64.

Respecto a los sistemas operativos y el soporte de estos para GDI+, Windows XP contiene la librería **gdiplus.dll** que encontraremos normalmente en *c:\windows\system32* y la cuál nos permite trabajar con GDI+. Microsoft .NET por otra parte, nos proporciona el acceso directo a esta librería para poder desarrollar nuestras aplicaciones de forma mucho más cómoda y sencilla.

FAQ:

¿Dispone mi sistema operativo de GDI+?

Microsoft .NET Framework instala automáticamente en su sistema las librerías GDI+ para que las pueda utilizar en sus aplicaciones.

Lección 4: Trabajo con imágenes y gráficos

- Gráficos 3D
 - Gráficos 2D
- Dibujando líneas con GDI+**
- Dibujando curvas con GDI+
 - Dibujando cadenas de texto con GDI+
 - Otras consideraciones

Módulo 3 - Capítulo 4

3. Dibujando líneas con GDI+

Lo primero que aprenderemos a representar con GDI+ y Visual Basic 2010, son líneas muy sencillas.

Líneas simples

Cuando representamos líneas, debemos tener en cuenta varios aspectos.

Una línea está representada por dos puntos. Cuando la representamos en un plano, La representación de una línea tiene dos pares de puntos (esto me recuerda a mis tiempos de estudiante con el álgebra y el cálculo).

Para representar una línea por lo tanto, debemos indicar un par de puntos (x, y) cuyas coordenadas (horizontal, vertical), representa en este orden, el lugar o punto de inicio indicado por el margen superior del formulario o superficie sobre la cuál deseamos dibujar nuestra línea, y un segundo par de puntos que representan la dirección final de nuestra línea.

Un ejemplo práctico de este tipo de representación es la que se detalla en el siguiente código fuente de ejemplo:

Código

```
Imports System.Drawing

Public Class Form1

    Protected Overrides Sub OnPaint(ByVal e As System.Windows.Forms.PaintEventArgs)
        MyBase.OnPaint(e)
        e.Graphics.DrawLine(New System.Drawing.Pen(Color.DarkBlue, 2), 1, 1, 50, 50)
    End Sub

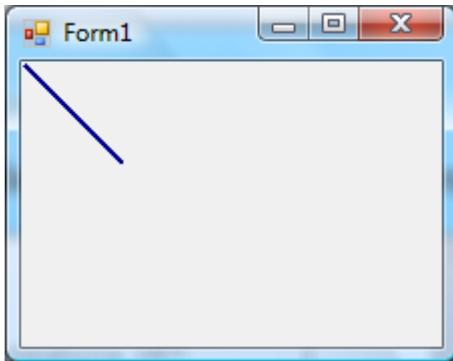
End Class
```

Atendiendo al código, observamos que hemos representado la gráfica indicando que queremos dibujar una línea *e.Graphics.DrawLine* indicando posteriormente una serie de atributos como el color del lápiz y tamaño o grosor de éste *Pen(Color.DarkBlue, 2)* y un conjunto de parámetros (x,y) que representan las coordenadas de la línea (1,1) y (50,50).

Otra representación similar sería *e.Graphics.DrawLine(Pens.DarkBlue, 1, 1, 50, 50)*, con la salvedad de que en este caso, el grosor del lápiz es el grosor por defecto, que es 1.

De todos los modos, la declaración `e.Graphics.DrawLine(New System.Drawing.Pen(Color.DarkBlue, 2), 1, 1, 50, 50)` y `e.Graphics.DrawLine(New System.Drawing.Pen(Color.DarkBlue, 2), 50, 50, 1, 1)` en el caso de la representación de líneas es igualmente compatible.

Este ejemplo de prueba en ejecución es el que se puede ver en la figura 1.



Ejemplo de dibujo con GDI+ de una línea recta en un formulario Windows

Figura 1

Líneas personalizadas

Sin embargo, la representación de líneas con GDI+ tiene diferentes particularidades que nos permiten sacar un alto grado de personalización a la hora de pintar o representar las imágenes y gráficos en nuestras aplicaciones.

La representación de líneas con GDI+, nos permite entre otras cosas, personalizar no sólo el color y el grosor de una línea, sino otras características de ésta como por ejemplo los extremos a dibujar.

Así, podemos dibujar unos extremos más o menos redondeados, puntiagudos, o personalizados.

Esto lo conseguimos hacer mediante las propiedades **StartCap** y **EndCap** de la clase **Pen** que es lo que vulgarmente he denominado como lápiz.

A estas propiedades, las podemos dar diferentes valores. Sirva el siguiente ejemplo de muestra de lo que podemos llegar a hacer.

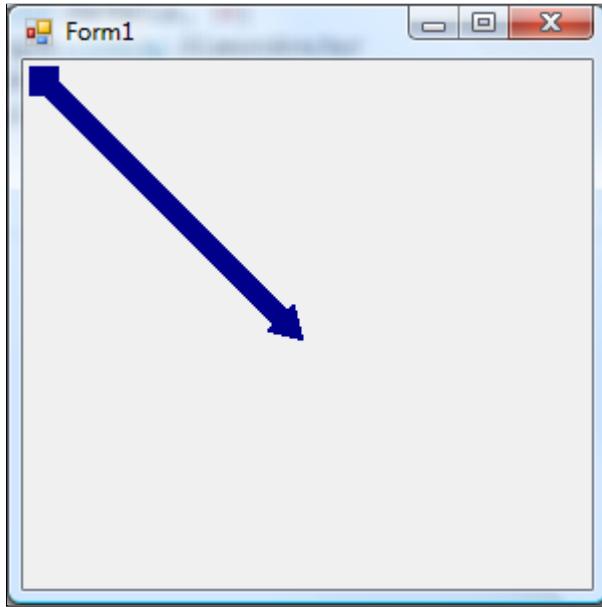
Código

```
Imports System.Drawing

Public Class Form1

    Protected Overrides Sub OnPaint(ByVal e As System.Windows.Forms.PaintEventArgs)
        MyBase.OnPaint(e)
        Dim Lapiz As New Pen(Color.DarkBlue, 10)
        Lapiz.StartCap = Drawing2D.LineCap.DiamondAnchor
        Lapiz.EndCap = Drawing2D.LineCap.ArrowAnchor
        e.Graphics.DrawLine(Lapiz, 10, 10, 140, 140)
    End Sub

End Class
```



Demostración de como representar diferentes extremos en una línea con GDI +

Figura 2

Trazando caminos o rutas de líneas

Otra posibilidad de GDI+ es la de crear líneas entrelazadas sin llegar a cerrarlas.

Esto se hace con el método **AddLine**.

De esta manera, podemos dibujar diferentes líneas para representarlas en el marco de trabajo.

El siguiente ejemplo, nos enseña a utilizar el método **AddLine**.

Código

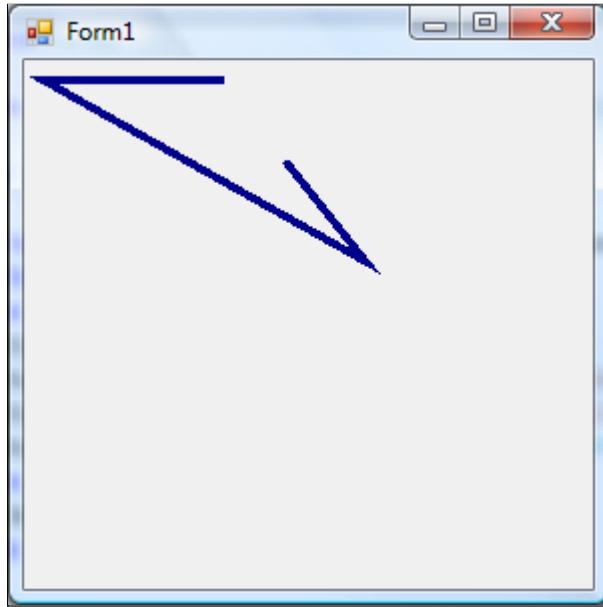
```
Imports System.Drawing

Public Class Form1

    Protected Overrides Sub OnPaint(ByVal e As System.Windows.Forms.PaintEventArgs)
        MyBase.OnPaint(e)
        Dim Ruta As New Drawing2D.GraphicsPath()
        Ruta.StartFigure()
        Ruta.AddLine(New PointF(10, 10), New PointF(100, 10))
        Ruta.AddLine(New PointF(10, 10), New PointF(170, 100))
        Ruta.AddLine(New PointF(170, 100), New PointF(130, 50))
        Dim Lapiz As New Pen(Color.DarkBlue, 4)
        e.Graphics.DrawPath(Lapiz, Ruta)
    End Sub

End Class
```

En la figura 3 podemos ver el resultado de crear diferentes líneas en Visual Basic 2010 con GDI+.



Ejecución del ejemplo de uso del método **AddLine** con Visual Basic 2010

Figura 3

Observando el código, vemos que hemos declarado el método **StartFigure** y el método **AddLine** de la clase **GraphicsPath**.

Finalmente y para representar la gráfica correspondiente, hemos llamado al método **DrawPath**.

Líneas con texturas

Otra de las características de la representación gráfica de líneas con Visual Basic 2010 y GDI+, es la posibilidad de representar líneas aplicando a esas líneas una determinada textura.

El siguiente ejemplo, aplica una textura de la bandera de la Comunidad Económica Europea a un camino de líneas.

Código

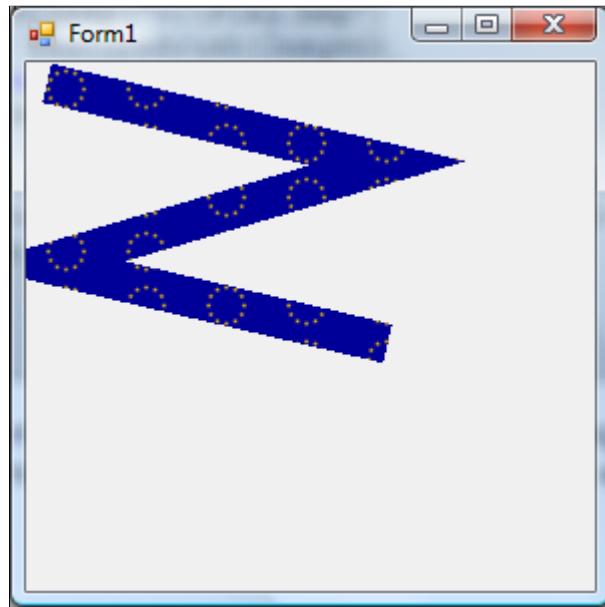
```
Imports System.Drawing

Public Class Form1

    Protected Overrides Sub OnPaint(ByVal e As System.Windows.Forms.PaintEventArgs)
        MyBase.OnPaint(e)
        Dim Imagen As New Bitmap("c:\Flag.bmp")
        Dim Cepillo As New TextureBrush(Imagen)
        Dim TexturaPincel As New Pen(Cepillo, 20)
        Dim Ruta As New Drawing2D.GraphicsPath()
        Ruta.StartFigure()
        Ruta.AddLine(New PointF(10, 10), New PointF(180, 50))
        Ruta.AddLine(New PointF(10, 100), New PointF(180, 140))
        e.Graphics.DrawPath(TexturaPincel, Ruta)
    End Sub

End Class
```

Como podemos observar, lo primero que hacemos es cargar una imagen que será la textura que utilizaremos, dentro de un objeto **Bitmap**, para posteriormente, preparar el trazo con su textura y tamaño. Luego creamos una ruta o camino que marcaremos para dibujarla en el formulario Windows en nuestro caso, tal y como puede verse en la figura 4.



Ejemplo de un trazo de línea aplicando texturas

Figura 4

Lección 4: Trabajo con imágenes y gráficos

- Gráficos 3D
- Gráficos 2D
- Dibujando líneas con GDI+
- Dibujando curvas con GDI+**
- Dibujando cadenas de texto con GDI+
- Otras consideraciones

Módulo 3 - Capítulo 4

4. Dibujando curvas con GDI+

La representación de curvas con GDI+ tiene cierta similitud con la representación de líneas. A continuación veremos las partes más destacables en la creación de trazos curvos con Visual Basic 2010 y GDI+.

Trazando curvas simples

Lo más sencillo de todo es siempre el trazo de una línea curva genérica con Visual Basic 2010. Esto lo conseguiremos con el siguiente código:

Código

```
Imports System.Drawing

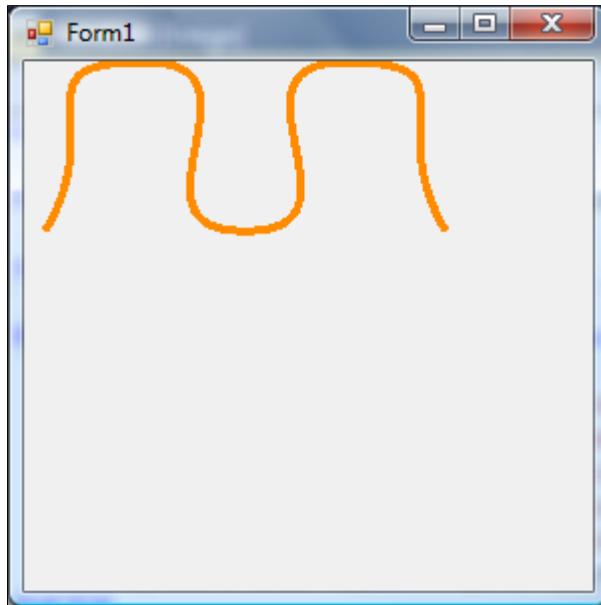
Public Class Form1

    Protected Overrides Sub OnPaint(ByVal e As System.Windows.Forms.PaintEventArgs)
        MyBase.OnPaint(e)
        Dim Puntos As PointF() = {New PointF(10, Convert.ToSingle(Math.Sin(1) * 100)), _
                                  New PointF(60, Convert.ToSingle(Math.Sin(0) * 100)), _
                                  New PointF(110, Convert.ToSingle(Math.Sin(1) * 100)), _
                                  New PointF(160, Convert.ToSingle(Math.Sin(0) * 100)), _
                                  New PointF(210, Convert.ToSingle(Math.Sin(1) * 100))}
        e.Graphics.DrawCurve(New Pen(Color.DarkOrange, 4), Puntos, 2.0F)
    End Sub

End Class
```

Si observamos este código, veremos que lo que hacemos es crear un conjunto de puntos para representar los trazos o líneas rectas que unan los puntos. Esto lo conseguimos utilizando la clase **PointF**. Posteriormente utilizamos el método **DrawCurve** con la salvedad de que el tercer parámetro, indica la tensión de la curva. Si este valor recibe un *0.0F*, la curva no tendrá tensión y por lo tanto, la representación será en trazos rectos, mientras que si ese valor aumenta, la tensión aumenta produciéndose el efecto curvo que deseamos conseguir.

El ejemplo anterior en ejecución es el que se muestra en la figura 1



Ejemplo en ejecución de la representación gráfica de trazos curvos

Figura 1

Curvas de Bézier

Otra posibilidad que nos ofrece GDI+ es la representación de curvas de Bézier, algo que conseguiremos gracias al método **DrawBezier** de GDI+.

La representación de las curvas de Bézier pueden hacerse mediante dos pares de puntos (x,y) o mediante cuatro coordenadas de puntos que definen la asignación de las curvas a representar.

El siguiente ejemplo nos muestra como representar una curva de Bézier con Visual Basic 2010 y GDI+.

Código

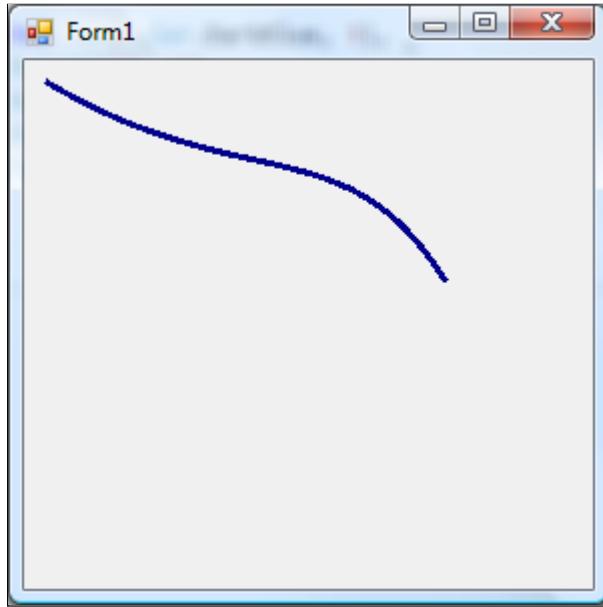
```
Imports System.Drawing

Public Class Form1

    Protected Overrides Sub OnPaint(ByVal e As System.Windows.Forms.PaintEventArgs)
        MyBase.OnPaint(e)
        e.Graphics.DrawBezier(New Pen(Color.DarkBlue, 3), _
            New PointF(10, 10), _
            New PointF(110, 70), _
            New PointF(160, 30), _
            New PointF(210, 110))
    End Sub

End Class
```

Este código en ejecución es el que puede verse en la figura 2.



Representación de las curvas de Bézier con GDI+ en Visual Basic 2010

Figura 2

Rellenando curvas

En algunas ocasiones, nos podemos ver con la necesidad de crear curvas y de llenar su interior para destacarlo de alguna manera. Esto es lo que veremos a continuación.

El método **AddArc** nos permite añadir una serie de puntos para representar una circunferencia. En realidad, se representan puntos, los cuales son el punto (x,y) inicial, el ancho y el alto de la representación, el ángulo de comienzo de representación, y el ángulo final de la representación. En ejemplo que veremos a continuación, utilizaremos también los métodos **FillPath** y **DrawPath**, para representar la curva en el formulario.

El código del ejemplo es el que se detalla a continuación:

Código

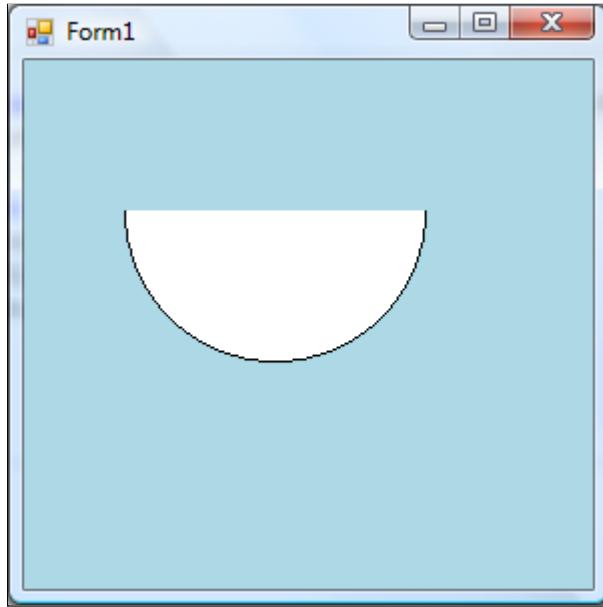
```
Imports System.Drawing

Public Class Form1

    Protected Overrides Sub OnPaint(ByVal e As System.Windows.Forms.PaintEventArgs)
        MyBase.OnPaint(e)
        Me.BackColor = Color.LightBlue
        Dim Camino As New Drawing2D.GraphicsPath()
        Camino.AddArc(50, 0, 150, 150, 0, 180)
        e.Graphics.FillPath(Brushes.White, Camino)
        e.Graphics.DrawPath(Pens.Black, Camino)
    End Sub

End Class
```

En la figura 3, podemos ver el ejemplo en ejecución.



Ejemplo de un dibujo de curvas cerradas llenando su interior

Figura 3

Dibujando tartas

Otra posibilidad que nos ofrece GDI+ es la representación de las conocidas tartas.

Todas estas representaciones son representaciones 2D, pero siempre se puede emular una representación 3D, superponiendo tartas una encima de otra.

El siguiente ejemplo, utiliza los métodos **FillPie** y **DrawPie** para generar un gráfico de tarta y rellenarlo de un determinado color.

El código de nuestro ejemplo, quedaría de la siguiente manera:

Código

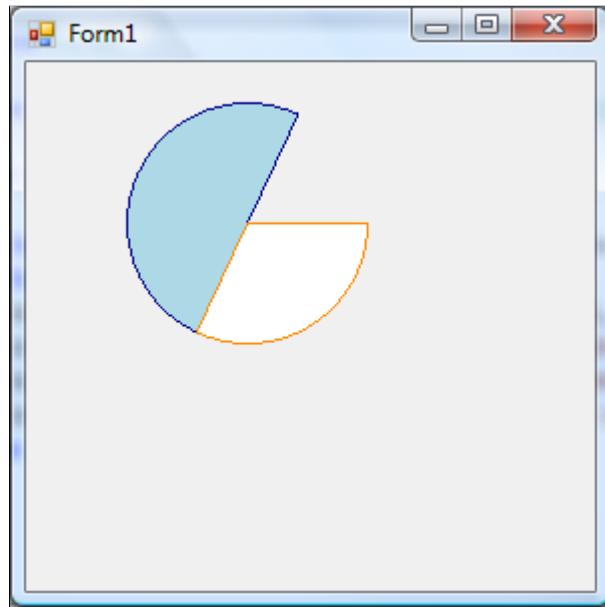
```
Imports System.Drawing

Public Class Form1

    Protected Overrides Sub OnPaint(ByVal e As System.Windows.Forms.PaintEventArgs)
        MyBase.OnPaint(e)
        e.Graphics.FillPie(Brushes.LightBlue, 50, 20, 120.0F, 120.0F, 115.0F, 180.0F)
        e.Graphics.DrawPie(Pens.DarkBlue, 50, 20, 120.0F, 120.0F, 115.0F, 180.0F)
        e.Graphics.FillPie(Brushes.White, 50, 20, 120.0F, 120.0F, 0.0F, 115.0F)
        e.Graphics.DrawPie(Pens.DarkOrange, 50, 20, 120.0F, 120.0F, 0.0F, 115.0F)
    End Sub

End Class
```

Nuestro ejemplo en ejecución, es el que se muestra en la figura 4.

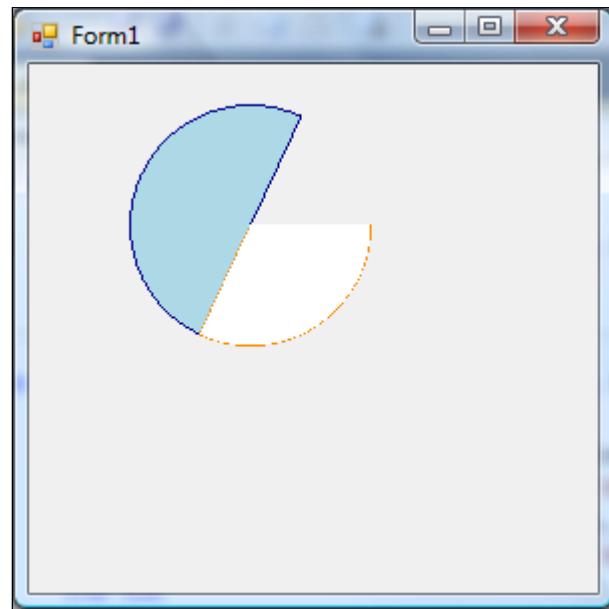


Demostración de cómo crear gráficos de tartas y como llenar su interior

Figura 4

Consejo:

Cuando represente gráficas de tartas por ejemplo y desee llenar una parte de esta de un color y marcar el borde en otro color, use primero el método FillPie y después el método DrawPie, en caso contrario, la representación gráfica perderá nitidez como se indica en la siguiente imagen:



Lección 4: Trabajo con imágenes y gráficos

- Gráficos 3D
- Gráficos 2D
- Dibujando líneas con GDI+
- Dibujando curvas con GDI+
- Dibujando cadenas de texto con GDI+**
- Otras consideraciones

Módulo 3 - Capítulo 4

5. Dibujando cadenas de texto con GDI+

Como hemos visto ya en el capítulo referente a la creación de nuestros propios controles, desde Visual Basic 2010, podemos utilizar GDI+ para crear cadenas de texto y manipularlas como deseemos.

En los siguientes apartados veremos como representar cadenas de texto desde Visual Basic 2010

Dibujando cadenas de texto

El método **DrawString** nos permite representar cadenas de texto de forma gráfica.

El funcionamiento en Visual Basic 2010 de estas instrucciones es realmente simple.

El siguiente ejemplo, ilustra en cómo abordar un pequeño proyecto para representar una cadena de texto en pantalla.

Código

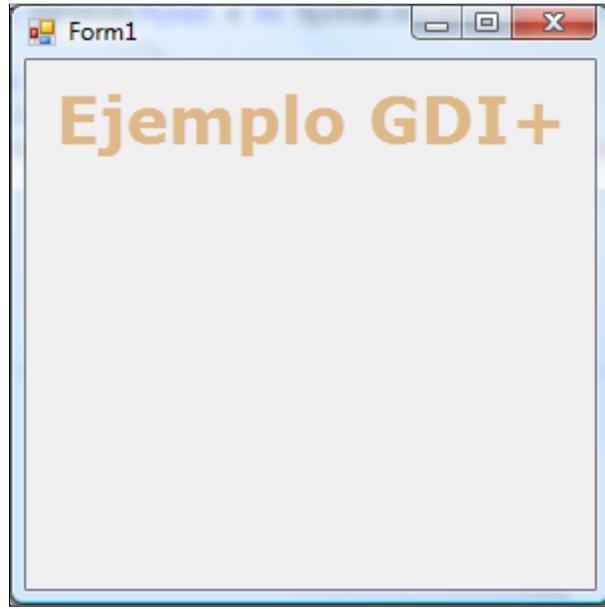
```
Imports System.Drawing

Public Class Form1

    Protected Overrides Sub OnPaint(ByVal e As System.Windows.Forms.PaintEventArgs)
        MyBase.OnPaint(e)
        Dim MiFuente As New Font("Verdana", 24, FontStyle.Bold)
        Dim Brocha As New SolidBrush(Color.BurlyWood)
        e.Graphics.DrawString("Ejemplo GDI+", MiFuente, Brocha, 10, 10)
    End Sub

End Class
```

Este ejemplo en ejecución es el que puede verse en la figura 1.



Ejemplo de cómo insertar una cadena de texto gráficamente en un formulario

Figura 1

Dibujando cadenas de texto con textura

Otra particularidad que a estas alturas ya no lo es tanto, es la posibilidad de trabajar con texturas dentro de cadenas de texto.

Como si estuviéramos dibujando una cadena de texto, la única variación es que asignamos como brocha de dibujo, una textura determinada.

El siguiente código, aclarará suficientemente esto que comento.

Código

```
Imports System.Drawing

Public Class Form1

    Protected Overrides Sub OnPaint(ByVal e As System.Windows.Forms.PaintEventArgs)
        MyBase.OnPaint(e)
        Dim Imagen As New Bitmap("c:\Flag.bmp")
        Dim TexturaDefondo As New TextureBrush(Imagen)
        Dim MiFuente As New Font("Arial", 30, FontStyle.Bold)
        e.Graphics.DrawString("Ejemplo GDI+", MiFuente, TexturaDefondo, 4, 10)
    End Sub

End Class
```

El resultado de este ejemplo, es el que puede verse en la figura 2.



Demostración del efecto de añadir una textura a la representación de una cadena de texto

Figura 2

Lección 4: Trabajo con imágenes y gráficos

- Gráficos 3D
 - Gráficos 2D
 - Dibujando líneas con GDI+
 - Dibujando curvas con GDI+
 - Dibujando cadenas de texto con GDI+
- Otras consideraciones**

Módulo 3 - Capítulo 4

6. Otras consideraciones

Uso de degradados con GDI +

GDI+ proporciona un variadísimo juego de brochas que nos permiten dibujar degradados en un formulario o control que permita el trabajo con gráficos.

El siguiente ejemplo de código, nos muestra como dibujar un degradado en un formulario Windows.

Código

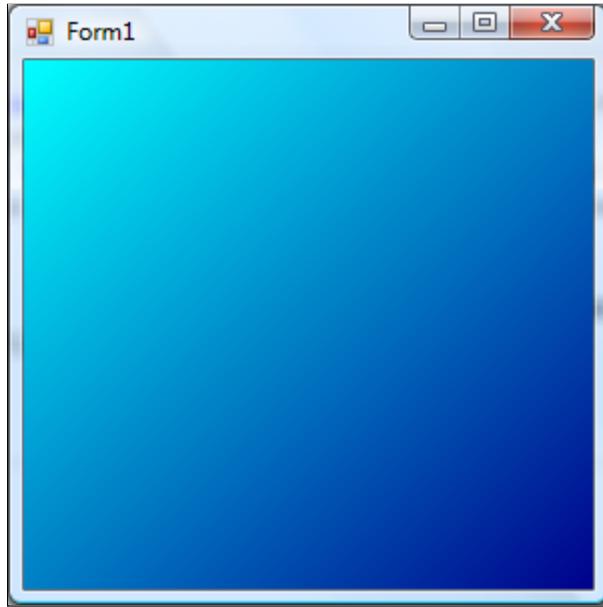
```
Imports System.Drawing

Public Class Form1

    Protected Overrides Sub OnPaint(ByVal e As System.Windows.Forms.PaintEventArgs)
        MyBase.OnPaint(e)
        Dim Forma As New Rectangle(New Point(0, 0), Me.ClientSize)
        Dim Gradiente As New Drawing2D.LinearGradientBrush(Forma, _
            Color.Cyan, _
            Color.DarkBlue, _
            Drawing2D.LinearGradientMode.ForwardDiagonal)
        e.Graphics.FillRegion(Gradiente, New Region(Forma))
    End Sub

End Class
```

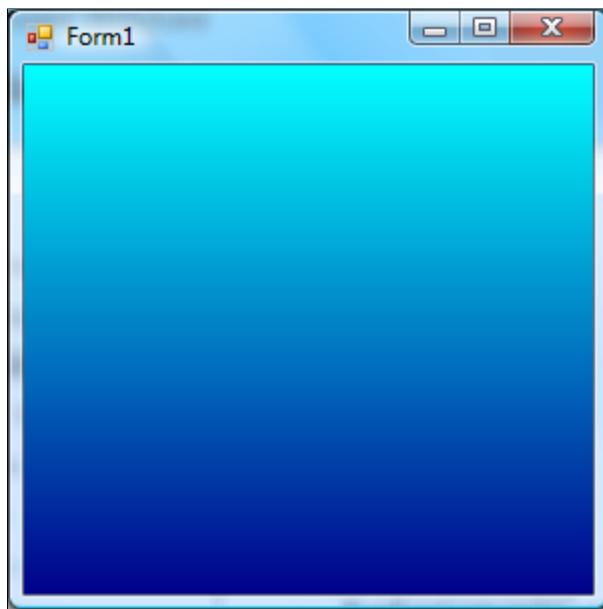
La figura 1 representa el ejemplo anterior en ejecución



Representación gráfica de un degradado en una ventana Windows

Figura 1

Evidentemente, podemos jugar con la clase **LinearGradientBrush** y con la lista enumerada **LinearGradientMode** para dar una aspecto o un toque ligeramente diferente al degradado, como el que se indica en la figura 2.



Otro ejemplo de representación degradada en un formulario Windows

Figura 2

Insertando y trabajando con imágenes con System.Drawing

Sirva como detalle general, que GDI+ nos permite también, trabajar con imágenes.

Para cargar una imagen en un formulario con GDI+, utilizaremos el método **DrawImage**.

Un ejemplo del uso de este método para cargar una imagen en un formulario es el siguiente:

Código

```
Imports System.Drawing
```

```

Public Class Form1

    Protected Overrides Sub OnPaint(ByVal e As System.Windows.Forms.PaintEventArgs)
        MyBase.OnPaint(e)
        e.Graphics.DrawImage(New Bitmap("c:\159.jpg"), 1, 1)
    End Sub

End Class

```

La figura 3 nos muestra el resultado final de insertar una imagen en el formulario Windows.

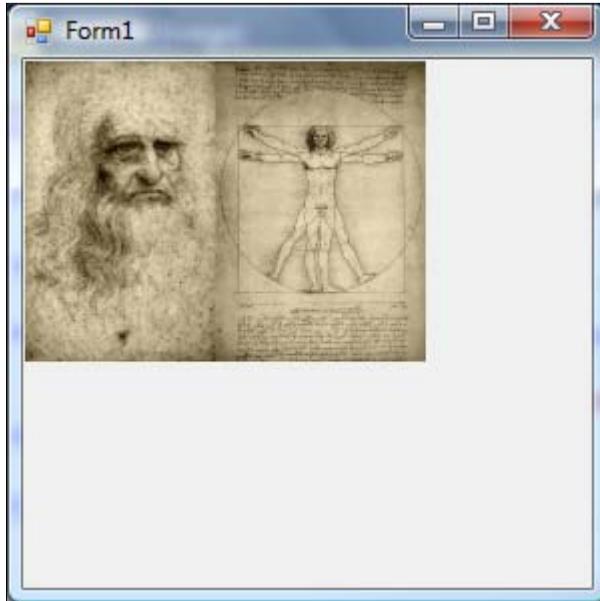


Imagen insertada con GDI + dentro del formulario Windows

Figura 3

Aplicando transparencias a una imagen

Otra de las características que nos ofrece GDI+, es el trabajo con imágenes aplicando transparencias. Para realizar esto, deberemos indicar el color o colores que queremos utilizar para provocar en él un efecto transparente.

El siguiente ejemplo, nos muestra como hacer esto utilizando para ello el método ***MakeTransparent***.

Código

```

Imports System.Drawing

Public Class Form1

    Protected Overrides Sub OnPaint(ByVal e As System.Windows.Forms.PaintEventArgs)
        MyBase.OnPaint(e)
        Dim Imagen As New Bitmap("c:\Flag.bmp")
        e.Graphics.DrawImage(New Bitmap(Imagen), 1, 1)
        Imagen.MakeTransparent(Color.FromArgb(255, 0, 51, 153))
        e.Graphics.DrawImage(New Bitmap(Imagen), 100, 1)
    End Sub

End Class

```

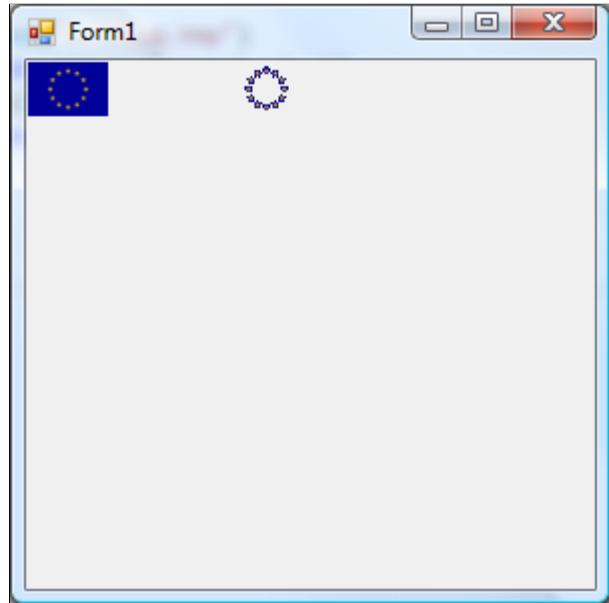


Imagen insertada con GDI+ dentro del formulario Windows

Figura 4

Lección 5: Despliegue de aplicaciones

- Desmitificando los ensamblados
- Desplegando con XCOPY
- GAC y Strong Names
- Creando un paquete de instalación
- Otras consideraciones

Introducción

En este capítulo, aprenderemos lo que son los ensamblados o *assemblies*, un término completamente nuevo para los desarrolladores de otros entornos distintos de .NET, y aprenderemos a desplegar o instalar nuestras aplicaciones.

Siempre que terminamos de desarrollar un proyecto, nos aborda la duda y pregunta casi obligada de *¿Y ahora qué?*.

Eso es lo que veremos en este capítulo... *el qué*.

Módulo 3 - Capítulo 5

- 1. [Desmitificando los ensamblados](#)
- 2. [Desplegando con XCOPY](#)
- 3. [GAC y Strong Names](#)
- 4. [Creando un paquete de instalación](#)
- 5. [Otras consideraciones](#)

Lección 5: Despliegue de aplicaciones

Desmitificando los ensamblados

- Desplegando con XCOPY
- GAC y Strong Names
- Creando un paquete de instalación
- Otras consideraciones

Módulo 3 - Capítulo 5

1. Desmitificando los ensamblados

Un concepto completamente nuevo para los que nunca han desarrollado con .NET, es la palabra **Assembly**, denominada *ensamblado*.

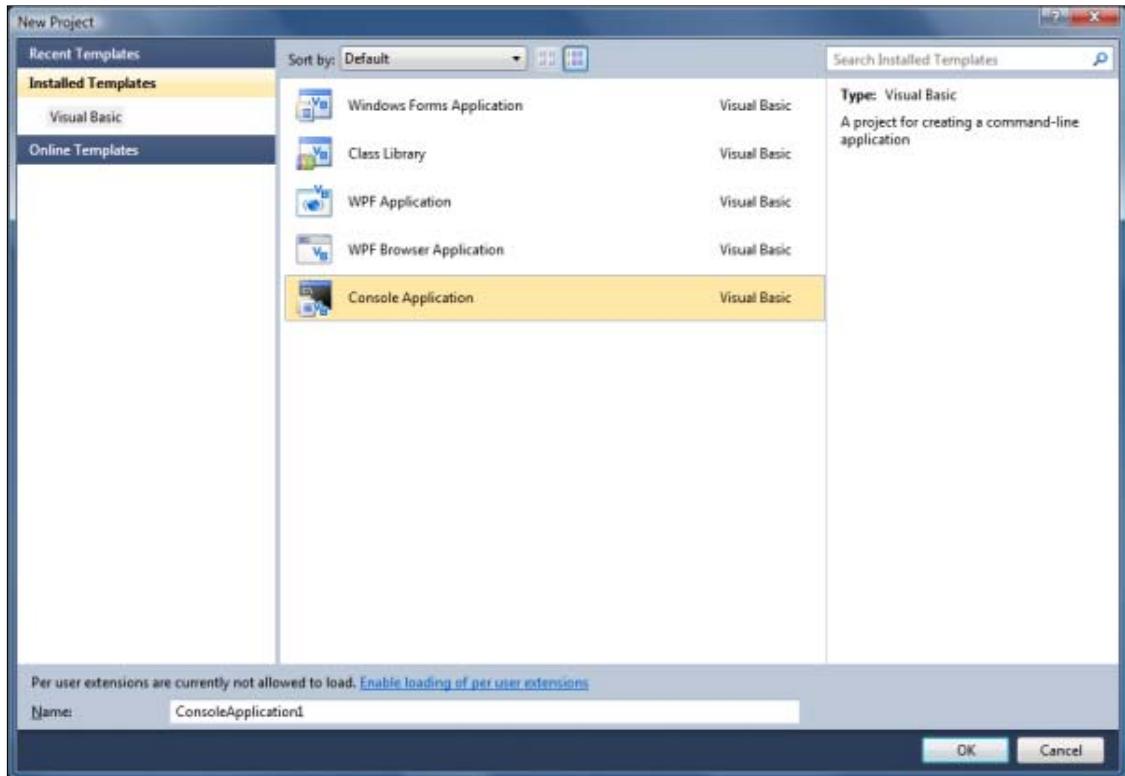
Los ensamblados, son para entenderlo muy rápidamente, como los ejecutables de una aplicación.

La única diferencia notable, es que en .NET, un proyecto o aplicación, se compila en código intermedio, el conocido como *Intermediate Language* o lenguaje intermedio, que luego interpretará el CLR o *Common Language Runtime* para ejecutarla en el sistema operativo correspondiente.

Ese código intermedio, es el ensamblado de nuestra aplicación que a su vez puede contener uno o más ficheros, y a su vez, un proyecto, puede estar contenido por uno o más ensamblados.

Por lo tanto, dentro de un ensamblado, se encierran algunas partes importantes que debemos conocer.

Lo mejor para entender bien lo que hay en un ensamblado y que contiene es que abramos Visual Studio 2010 y seleccionemos una plantilla de proyecto de tipo **Console Application** como se muestra en la figura 1.



Seleccionando un proyecto de tipo Console Application

Figura 1

A continuación, escribiremos un ejemplo simple de consola, para que estudiemos el resultado de éste.

El código fuente de nuestra pequeña aplicación de ejemplo, es el que se detalla a continuación:

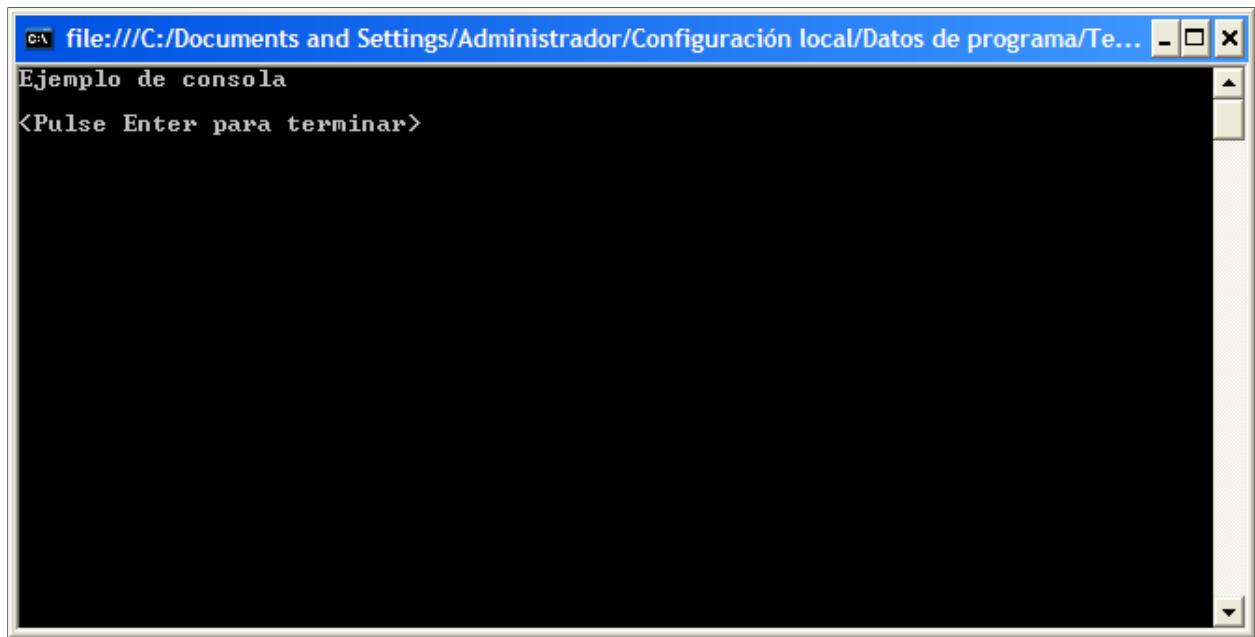
Código

```
Module Module1

    Sub Main()
        Console.WriteLine("Ejemplo de consola")
        Console.WriteLine("")
        Console.Write("<Pulse Enter para terminar>")
        Console.ReadLine()
    End Sub

End Module
```

Nuestro ejemplo de prueba en ejecución, es el que puede verse en la figura 2.



Ejecución del ejemplo de Consola

Figura 2

Ahora bien, lo que tenemos una vez compilamos nuestra aplicación de consola, no es un ejecutable como tal o como lo entenderíamos en otros compiladores, por ejemplo en Visual C++.

En Visual C++, generábamos un ejecutable nativo al sistema en el cuál compilábamos el proyecto y si ese ejecutable lo llevábamos a otra máquina con otro sistema operativo diferente a Windows, ese programa no iba a funcionar.

Con .NET y en nuestro caso con Visual Basic 2010, este concepto ha cambiado.

El proyecto que hemos desarrollado no se compila a un ejecutable nativo, sino que el sistema .NET lo compila a un lenguaje intermedio, que luego el **CLR** de la máquina en la cuál lanzamos nuestra aplicación, será capaz de interpretar adecuadamente.

Puede que estos conceptos le puedan desorientar un poco, pero es muy fácil de comprender.

Adicionalmente a esto, lo suyo es destripar un poco lo que hay dentro del ensamblado que hemos convertido a código intermedio.

Para llevar a cabo nuestra tarea, haremos uso de una herramienta externa de Microsoft y del entorno Visual Studio 2010, que nos permite analizar el ensamblado de un proyecto convertido a código intermedio.

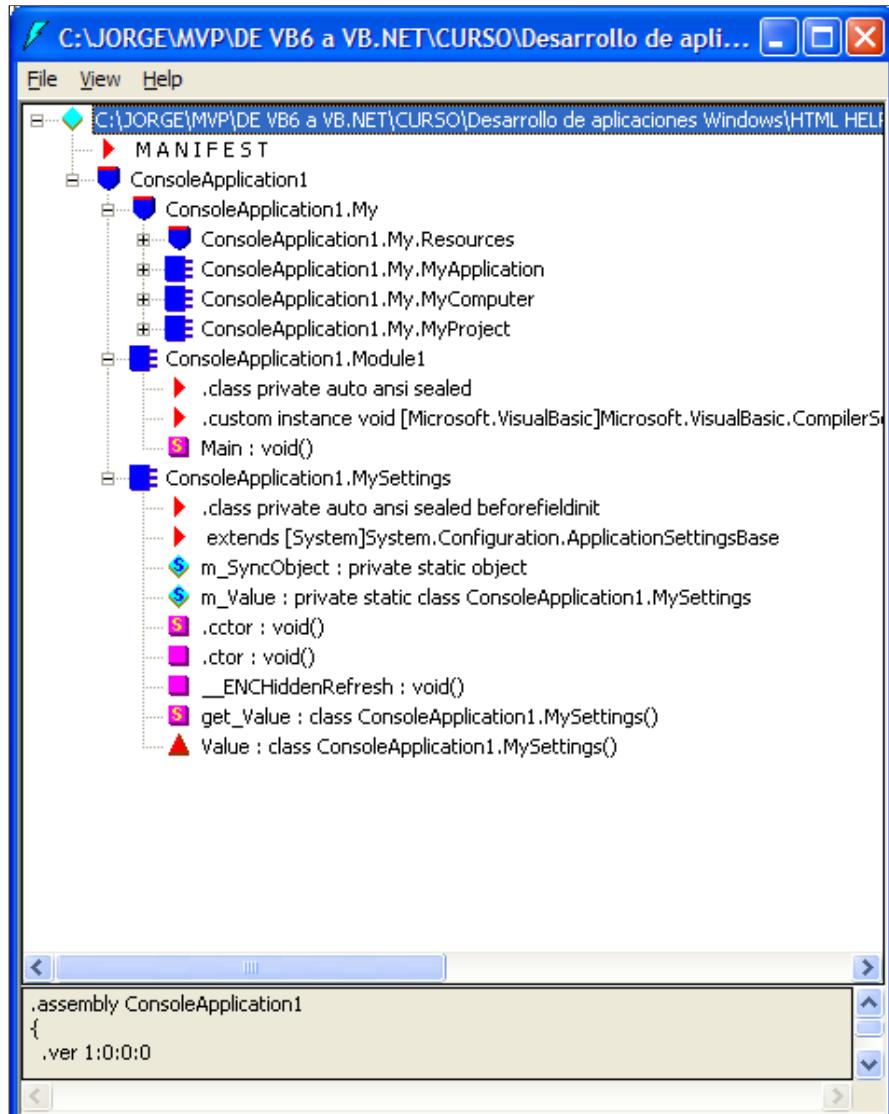
Aclaración:

*Un ensamblado o el código intermedio, (**IL** a partir de ahora), no es ni el código fuente de nuestra aplicación ni el programa ejecutable.*

*El **IL** se puede analizar con una herramienta de Microsoft denominada **ildasm**.*

Si está usando una versión Express de Visual Studio debe saber que esta herramienta no se incluye con estas versiones, sólo forma parte del SDK de .NET Framework que se incluye con las versiones de Visual Studio 2010.

Cuando abrimos el fichero ejecutable de nuestra aplicación con la herramienta *ildasm.exe*, observamos que esta tiene una gran cantidad de información, como se indica en la figura 3.



ildasm con el fichero ejecutable de nuestra aplicación de ejemplo abierto

Figura 3

Antes de adentrarnos más en los entresijos de un ensamblado, piense en él como si fuera una colección de elementos. Esos elementos, pueden ser recursos, tipos, funcionalidades, que todas juntas, forman nuestra aplicación.

Así, lo primero que vemos en la figura 3, es la palabra **MANIFEST**.

Referencias a las librerías utilizadas que el **CLR** deberá interpretar posteriormente.

Luego encontramos otras el ámbito de la aplicación y las clases de ésta, con sus método tanto estáticos como no estáticos.

No es cuestión de entrar mucho más en detalle del código **IL** que contiene un proyecto compilado en .NET como éste, pero es conveniente a verlo para entender los conceptos que estamos tratando. En la figura 4, tenemos el código **IL** del método **Main**.

The screenshot shows a debugger window with the title bar "ConsoleApplication1.Module1::Main : void()". The menu bar includes "Buscar" and "Buscar siguiente". The main pane displays the IL code for the Main method:

```
.method public static void Main() cil managed
{
    .entrypoint
    .custom instance void [mscorlib]System.STAThreadAttribute::.ctor() = ( 01 00 00 00 )
    // Code size       42 (0x2a)
    .maxstack 8
    IL_0000: nop
    IL_0001: ldstr     "Ejemplo de consola"
    IL_0006: call      void [mscorlib]System.Console::WriteLine(string)
    IL_000b: nop
    IL_000c: ldstr     ....
    IL_0011: call      void [mscorlib]System.Console::WriteLine(string)
    IL_0016: nop
    IL_0017: ldstr     "<Pulse Enter para terminar>"
    IL_001c: call      void [mscorlib]System.Console::Write(string)
    IL_0021: nop
    IL_0022: call      string [mscorlib]System.Console::ReadLine()
    IL_0027: pop
    IL_0028: nop
    IL_0029: ret
} // end of method Module1::Main
```

El código IL del método Main de la aplicación de ejemplo

Figura 4

Los ensamblados como podemos ver, contiene más información de la que propiamente tendría un ejecutable "normal".

Un ensamblado en .NET, contiene también como hemos podido ver, datos e información que encontraríamos en cualquier tipo de librería, lo cuál representa además, toda la información necesaria del **CLR** en cualquier momento.

Con todo esto, podemos resumir por lo tanto, que un ensamblado contiene código, recursos y metadatos.

El código en **IL** es el código que será ejecutado por el **CLR**.

Además, cualquier recurso (imágenes, metadatos, etc.) son accesibles en el ensamblado.

Los metadatos por su parte, contiene información sobre las clases, interfaces, métodos y propiedades, que posibilitan toda la información necesaria por el **CLR** para poder ejecutar nuestra aplicación correctamente.

Lección 5: Despliegue de aplicaciones

- Desmitificando los ensamblados
- Desplegando con XCOPY
 - GAC y Strong Names
 - Creando un paquete de instalación
 - Otras consideraciones

Módulo 3 - Capítulo 5

2. Desplegando con XCOPY

Notas previas

Todas las aplicaciones desarrolladas con .NET están aisladas, no como ocurre con los compilados pre-.NET, de manera tal que los conflictos con las DLL se han visto reducidos enormemente, por no decir que han desaparecido.

El famoso *infierno de las DLLs* que tanto hemos sufrido los desarrolladores, ha pasado ya a la historia.

También podemos usar componentes privados.

Basta con copiarlos al mismo directorio en el que se encuentra nuestra aplicación ejecutable.

Además, .NET nos permite tener más de un componente versionado (diferentes versiones de un mismo componente) dentro de un mismo ordenador, por lo que los problemas de compatibilidad estarían resueltos.

Con estos detalles, repasamos algunas de las mejoras a la hora de desplegar o instalar una aplicación desarrollada con Visual Basic 2010 en un sistema.

A continuación, veremos algunas anotaciones adicionales que nos ayudarán a realizar estas y otras acciones.

XCOPY

Lo que nos ofrece **XCOPY** a los desarrolladores e ingenieros, es la posibilidad de instalar e implementar nuestra solución y proyecto a un sistema de una manera rápida, fiable y sin apenas impacto.

El método **XCOPY** para desplegar aplicaciones, pasa por alto la posibilidad de implementar los ensamblados en la **GAC**, algo que según determinadas circunstancias, resulta más que provechoso.

Hay que tener en cuenta, que si hacemos un mal uso de la **GAC**, ésta puede convertirse en un desván difícil de gestionar.

Utilice la **GAC** con criterio y si no quiere complicaciones, despliegue sus aplicaciones con **XCOPY**.

Como habrá podido ya observar, dentro de un proyecto de Visual Basic 2010, nos podemos encontrar con una extensa estructura de directorios.

Esta estructura, lejos de ser una molestia, constituye las características más importantes a la hora de desplegar nuestras aplicaciones a través de **XCOPY**.

Como vemos, todo en .NET tiene su sentido y tiene un porqué.

Para instalar nuestra aplicación desarrollada en Visual Basic 2010 en un sistema cliente, bastará por lo tanto, con realizar una acción similar al **XCOPY** de *DOS*, es decir, copiaremos en el ordenador cliente, los ficheros o ensamblados necesarios (ejecutables, recursos, dlls, etc.) de la estructura de nuestro proyecto.

Debemos tener en cuenta, que en otros lenguajes basados en COM, como Visual Basic 6, podíamos hacer esto igualmente, pero debíamos además registrar las DLL con aquel famosísimo comando *regsvr32* para que no hubiera problemas, aún así, algunas veces nos encontrábamos con algunos contratiempos, sin embargo, con Visual Basic 2010, esta forma de trabajar ha desaparecido y ahora el despliegue de una aplicación es mucho más sencilla.

¡Ojo!

Cuando desarrollemos una aplicación, tenga en cuenta otros recursos que utiliza en la misma.

Crystal Reports, bases de datos SQL Server, que la máquina cliente disponga de .NET Framework o de la versión mínima de MDAC necesaria, etc.

En caso contrario, la ejecución de nuestra aplicación, podría no funcionar o provocar algún tipo de excepción o error.

Por último, y aunque parezca de perogrullo, no olvide que debe tener los permisos necesarios para instalar y ejecutar los ensamblados y otras partes de Software, necesarios para ejecutar su aplicación.

Lección 5: Despliegue de aplicaciones

- Desmitificando los ensamblados
- Desplegando con XCOPY
- GAC y Strong Names**
- Creando un paquete de instalación
- Otras consideraciones

Módulo 3 - Capítulo 5

3. GAC y Strong Names

GAC

El **GAC** o *Global Assembly Cache* no es otra cosa que un repositorio de ensamblados globales.

Imaginemos que usted todos los días cuando llega a casa de trabajar, lo primero que hace es quitarse los zapatos y ponerse una zapatilla cómoda para estar por casa.

Lo lógico en este caso, será situar un zapatero o un pequeño armario para que ponga ahí las zapatillas, ya que todos los días, hace repetitivamente esta operación.

Obviamente, las zapatillas deben de estar ahí y no en la otra punta de la casa, pero para estar en ese zapatero, deberá cumplir una serie de requisitos que usted mismo exige, por lo que no todos los zapatos o zapatillas deberían estar ahí.

El **GAC** funciona de una manera realmente semejante.

En el **GAC** se incluyen aquellas librerías o ensamblados que son utilizados frecuentemente.

Si usted va a desarrollar y distribuir su propio ensamblado, convendría ponerlo en el **GAC**.

Una aplicación .NET, lo primero que hace cuando se ejecuta, es revisar los ensamblados que va a necesitar, y el primer sitio dónde va a ir a buscarlo es en el **GAC**.

Sino lo encuentra, se pondrá a buscarlo en el directorio en el que se encuentra el fichero ejecutable, pero esto repercute en el rendimiento.

Si nuestras aplicaciones utilizan frecuentemente unos ensamblados, sería lógico y conveniente ponerlos en el **GAC**.

¿Y cómo se añade un ensamblado al **GAC**?

La tarea no es sencilla, ya que para añadirlo debemos realizar algunos pasos, entre los que está el crear un nombre fuerte o **Strong Name**.

Strong Names

Con los **Strong Names**, aseguramos un uso seguro de los componentes contenidos en el ensamblado.

Hay que tener en cuenta que un ensamblado declarado en la **GAC** que es llamado por varias aplicaciones, crea una única instancia.

De ahí, que crear un **Strong Name** para el ensamblado, está más que justificado.

Un **Strong Name**, nos asegura por otro lado, que el nombre de un ensamblado es único y que por lo tanto,

al ser único, no puede ser utilizado por otros ensamblados.

Los **Strong Names** se generan con un par de claves, una de ellas de carácter público y otra de carácter privado, claves que se introducen en fichero de ensamblado de la aplicación, y que luego al compilar nuestra aplicación, queda registrada con ese **Strong Name**.

Para indicar un **Strong Names** a un ensamblado, debe crear primero el par de claves (clave pública y clave privada), que se generará en un fichero con extensión **snk**, y modificar posteriormente el fichero *AssemblyInfo.vb* de su proyecto.

En ese archivo, debe añadir una instrucción similar a la siguiente:

```
<Assembly: AssemblyKeyFile("KeyFile.snk")>
```

Lección 5: Despliegue de aplicaciones

- Desmitificando los ensamblados
 - Desplegando con XCOPY
 - GAC y Strong Names
- Creando un paquete de instalación**
- Otras consideraciones

Módulo 3 - Capítulo 5

4. Creando un paquete de instalación

Setup Project

Otra acción habitual, es que cuando desarrollemos nuestras aplicaciones, generemos diferentes dependencias con otros ensamblados o componentes, para lo cuál, la forma más sencilla de desplegar nuestra aplicación, es generando un paquete de distribución que contenga esas dependencias y relaciones.

Nota:

Si está utilizando Visual Basic 2010 Express no tendrá disponible la opción de proyecto Setup.

Con la plantilla *Setup Project* crearemos un nuevo proyecto para generar el paquete de distribución e instalación de nuestra aplicación y proyecto.

Cuando generamos el paquete de instalación, debemos tener en cuenta que se generan dos archivos que por lo general tendrán los nombres de *Setup.exe* y *Setup.msi*.

La diferencia entre ambos ficheros es que el fichero con extensión **exe** instalará *Windows Installer* si es necesario, mientras que el fichero con extensión **msi**, no instalará *Windows Installer*, por lo que si *Windows Installer* no está presente en la máquina en la que se ejecuta el programa de instalación, dará un error.

Otra consideración a tener en cuenta cuando generamos el paquete de instalación, es en el momento de la distribución, el asegurar que el sistema destino tenga el .NET Framework correspondiente.

Sugerencia:

[¿Cómo usar Visual Studio 2010 para distribuir Microsoft .NET Framework?](#)

Más que interesante artículo donde obtendrá información adicional sobre cómo generar un paquete de instalación para distribuir .NET Framework.

[Using Visual Studio .NET to Redistribute the .NET Framework \(en inglés\)](#)

Respecto al comportamiento de *Windows Installer* sobre aplicaciones ya instaladas que deseamos desinstalar, el entorno se comporta de manera tal, que si detecta componentes compartidos, estos no son desinstalados del sistema.

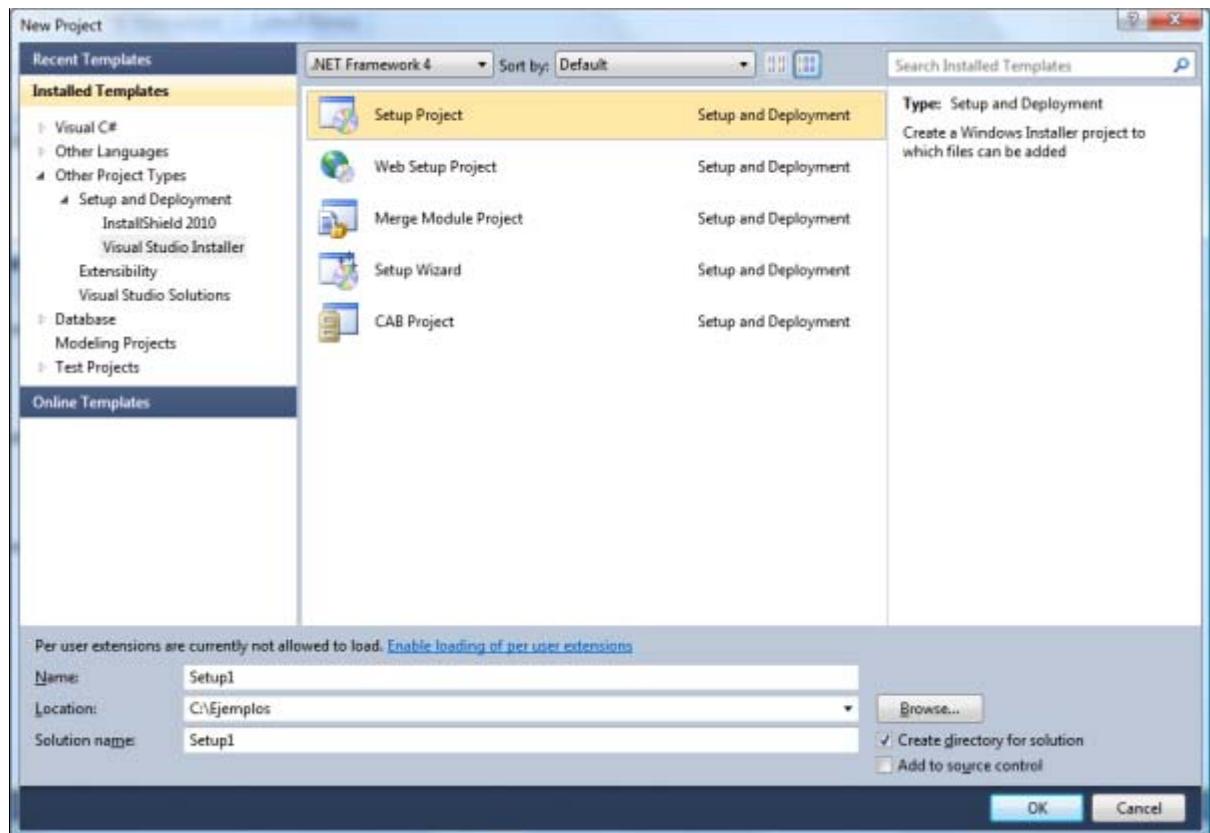
Otro comportamiento de *Windows Installer* es el que nos permite echar marcha atrás si el proceso de instalación se cancela o falla por algún motivo.

Windows Installer dejará el sistema en el mismo estado que tenía justo antes de realizar la instalación.

Tipos de despliegues de proyectos

Cuando generamos un paquete de instalación tenemos dentro del entorno de desarrollo Visual Studio varias opciones, como se indica en la imagen 1.

Así, nos podemos encontrar por lo general con diferentes tipos de despliegues de proyectos.



Tipos de despliegue de proyectos en Visual Studio

Figura 1

La plantilla ***Cab Project*** El fichero con extensión **CAB** es un fichero comprimido que contiene todos los ficheros y recursos necesarios para instalar nuestra aplicación.

Por lo general, este tipo de ficheros son usados para descargarlos de Servidores Web.

Por lo general y en nuestro caso, crearemos aplicaciones Windows, y deberíamos entonces utilizar la plantilla ***Setup Project***.

Si utilizamos la plantilla, ***Merge Module Project***, crearemos un paquete instalación para componentes compartidos.

El uso de la plantilla, ***Web Setup Project*** por su parte, nos permitirá generar un paquete de instalación para aplicaciones basadas en Internet o aplicaciones Web.

La plantilla, ***Setup Wizard*** genera un asistente para generar uno de los cuatro anteriores tipos de proyectos de despliegue.

Es una forma rápida de generar el proyecto de despliegue.

Lección 5: Despliegue de aplicaciones

- Desmitificando los ensamblados
 - Desplegando con XCOPY
 - GAC y Strong Names
 - Creando un paquete de instalación
- Otras consideraciones**

Módulo 3 - Capítulo 5

5. Otras consideraciones

Setup Project

Ya lo hemos comentado por encima en los capítulos anteriores sobre el *Despliegue de aplicaciones*, pero cuando se procede a instalar y distribuir una aplicación en otros sistemas, se deben tener en cuenta diferentes aspectos que no podemos pasar por alto.

Debemos conocer en primer lugar, la naturaleza del sistema o sistemas destino dónde vamos a implantar nuestro programa. Entendiendo y conociendo bien esto, podremos saber qué aplicaciones Software adicionales necesitaremos para abordar nuestro trabajo adecuadamente.

Entre otros, debemos tener en cuenta que el sistema destino dónde se va a ejecutar nuestra aplicación dispone de los drivers de acceso a datos adecuados, *MDAC* (versión 2.6 ó superior), y por supuesto de Microsoft .NET Framework.

FAQ:

[¿Cuando instalar con Windows Installer y cuando a través de XCOPY?](#)

Interesante artículo que debate sobre cuando instalar con Windows Installer y cuando con XCOPY.

[▶ Determining When to Use Windows Installer Versus XCOPY \(en inglés\)](#)

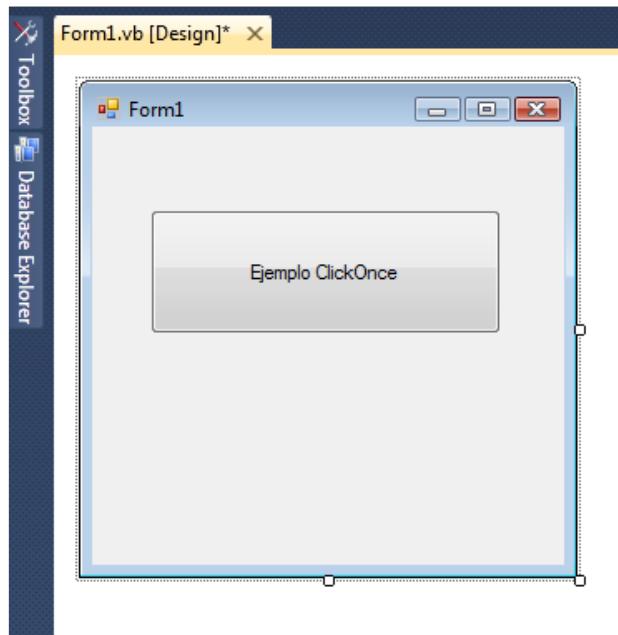
El anterior artículo, nos da un enfoque sobre qué tipo de instalación o despliegue establecer de nuestra aplicación desarrollada con Visual Basic 2010.

El concepto ClickOnce

En *Visual Studio 2010* se introduce un concepto nuevo denominado *ClickOnce*.

Este concepto representa la tecnología que permite instalar y ejecutar aplicaciones Windows desde un servidor Web con una escasa acción por parte del usuario.

Inicie un nuevo proyecto Windows. Dentro del formulario Windows inserte un control *Button* dentro del cuál, inserte un texto identificativo como el que se muestra en la figura 1.



Aplicación Windows de ejemplo para explicar el concepto de ClickOnce

Figura 1

A continuación, escriba el siguiente código fuente:

```
Código

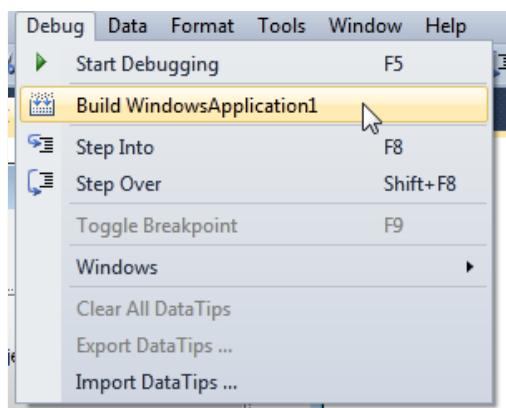
Public Class Form1

    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button1.Click
        MessageBox.Show("Ejemplo ClickOnce ejecutado a las:" & vbCrLf & Date.Now.ToString)
    End Sub
End Class
```

Una vez que ha escrito el código fuente de la aplicación, ejecútela y compruebe que funciona como se espera.

Si lo desea, abra el fichero *AssemblyInfo.vb* e indique la versión de la aplicación que deseé. En mi caso he dejado la versión 1.0.0.0 que es la versión que aparece por defecto.

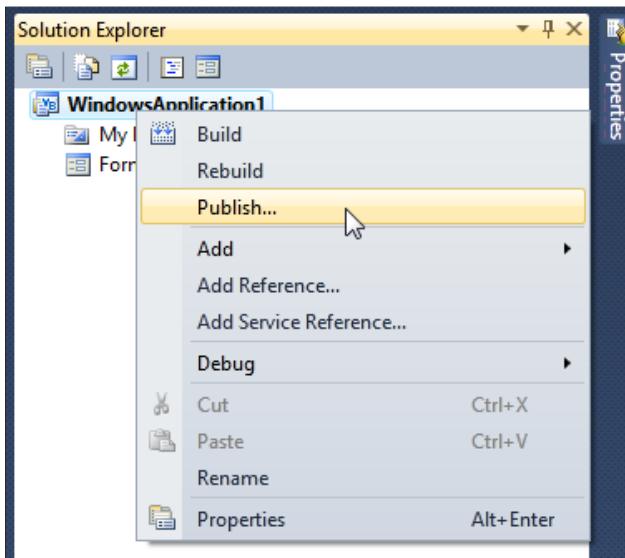
Después de esto, genere la aplicación como se indica en la figura 2.



Generamos la aplicación para comprobar entre otras cosas que todo está preparado

Figura 2

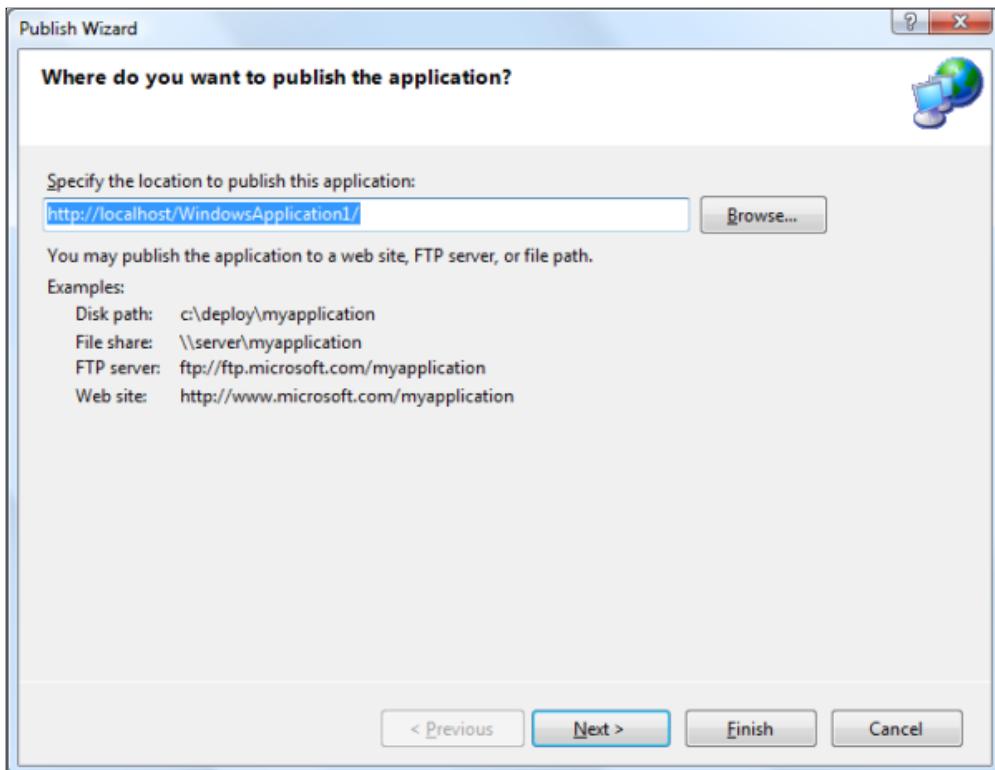
Por último, pulse el botón secundario del mouse sobre el proyecto y seleccione la opción **Publicar...** como se indica en la figura 3.



Opción de Publicar la aplicación para ejecutar la tecnología de distribución ClickOnce

Figura 3

Al ejecutar la opción **Publicar...**, el entorno nos muestra una ventana similar a la que se muestra en la figura 4.

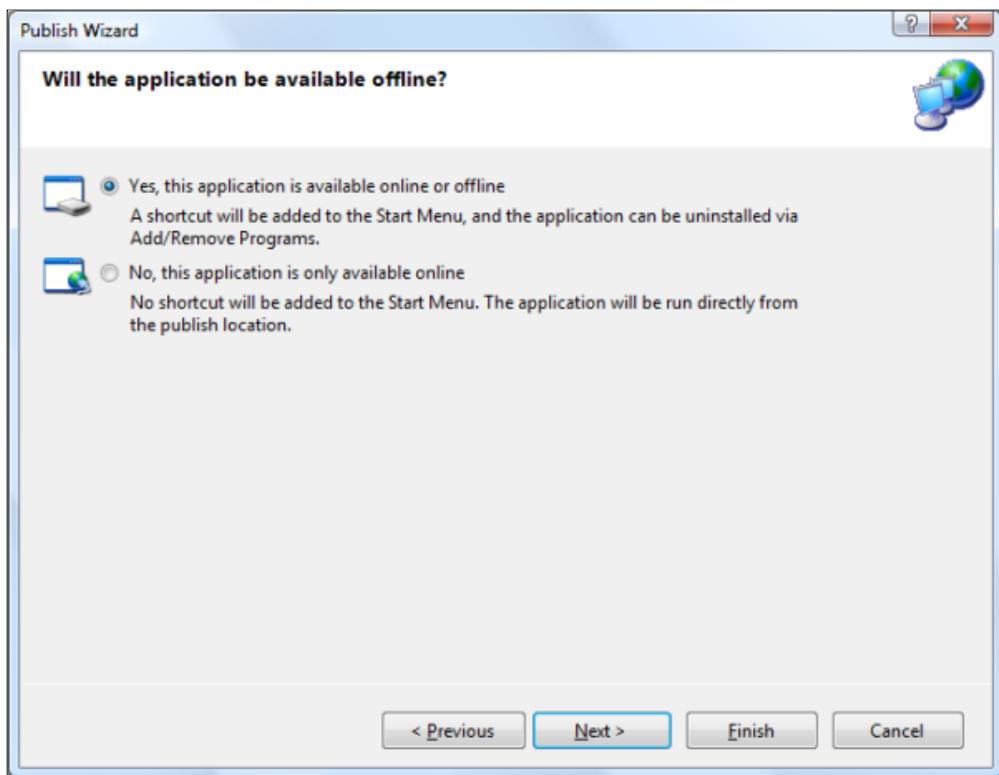


Asistente de la publicación de la aplicación

Figura 4

Seleccione una ubicación para publicar la aplicación y haga clic en el botón **Siguiente**.

Aparecerá en el asistente entonces, una ventana similar a la que se presenta en la figura 5.

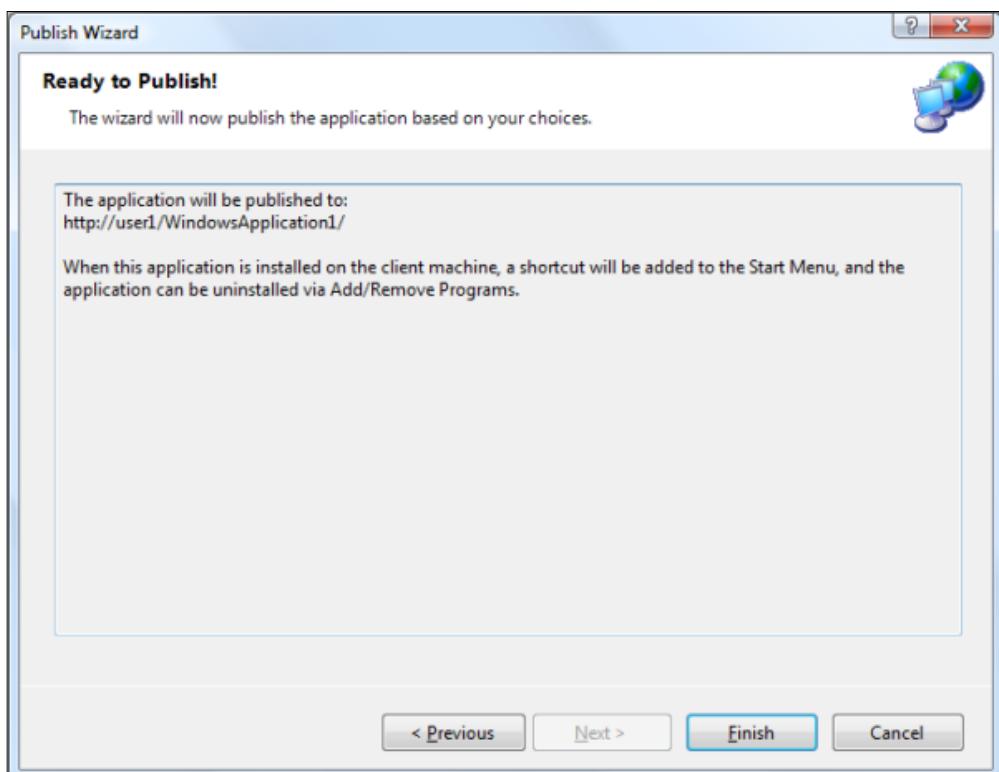


Ventana del asistente para indicar dónde estará disponible la aplicación

Figura 5

Pulse el botón **Siguiente**.

Nuevamente, el asistente mostrará una última ventana similar a la que se muestra en la figura 6.



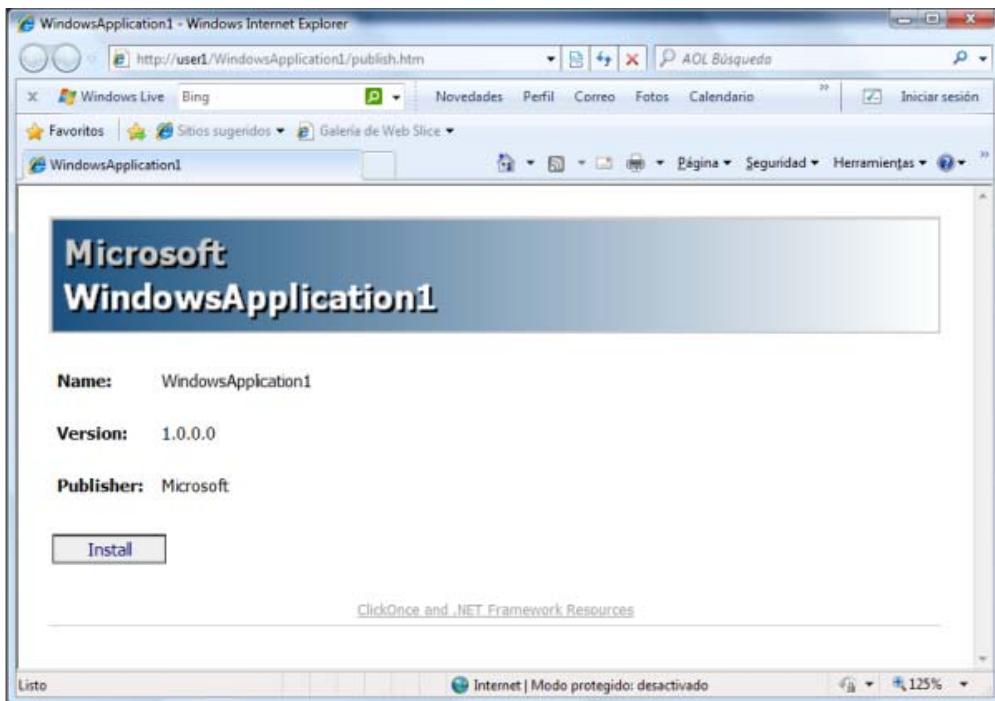
Ventana final del asistente con un resumen de las opciones seleccionadas

Figura 6

El asistente muestra en este caso, una información de resumen. Lo que haremos a continuación, será presionar el botón **Finalizar**.

Con esta acción, nuestra aplicación está ya publicada en el servidor Web, por lo que si abrimos una ventana de nuestro explorador Web y escribimos la dirección en la cuál hemos publicado nuestra aplicación, ésta se ejecutará de forma correcta como se indica en la

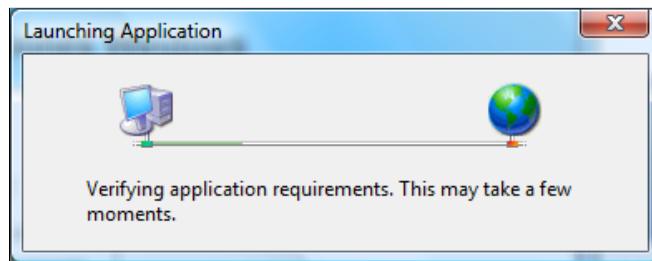
figura 7.



Aplicación Web del lanzamiento de la aplicación Windows a través del Servidor Web

Figura 7

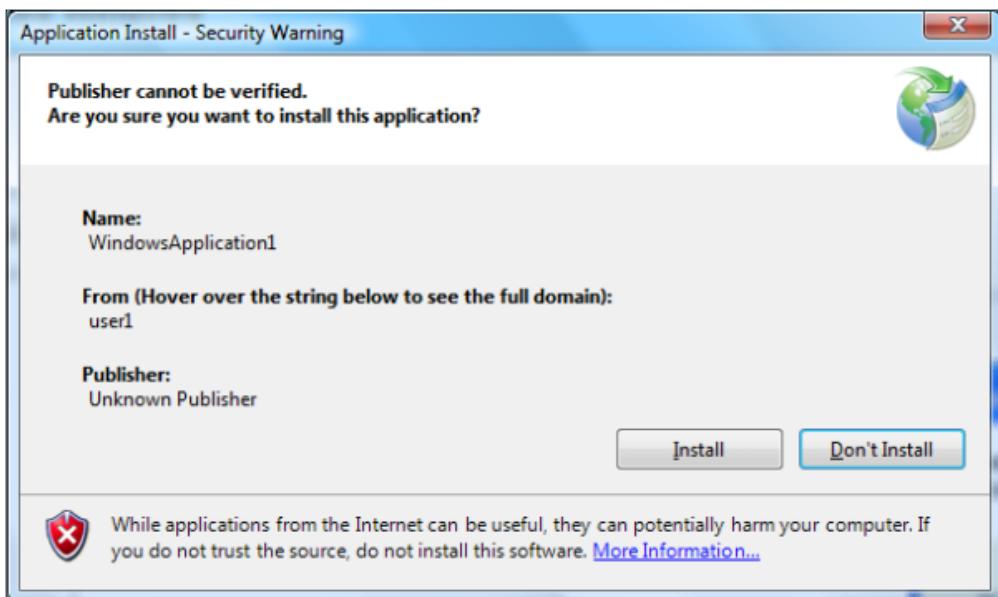
Si presionamos sobre el botón **Instalar** del navegador Web, observaremos que el Servidor Web realiza diferentes acciones de verificación. La primera acción es una acción de conexión como la que se muestra en la figura 8.



La primera acción que se realiza es una acción de conexión con el Servidor Web

Figura 8

Posteriormente, puede aparecer una ventana de seguridad como la que se indica en la figura 9, siempre y cuando no hayamos realizado un proceso de generación segura o confiable de la aplicación, como ha sido el caso.

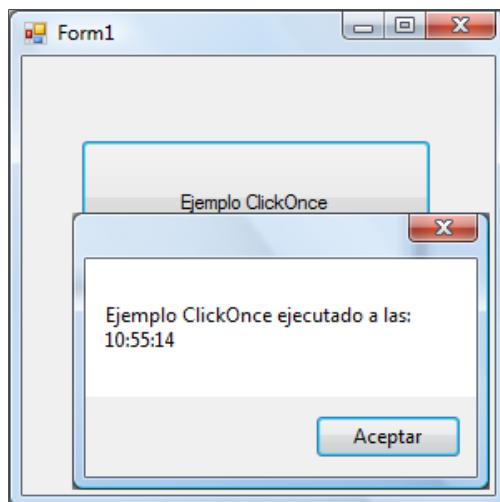


La aplicación Windows ejecutada a través del Servidor Web, debe ser confiable y segura

Figura 9

En nuestro caso, como sabemos que no hay problema en ejecutar la aplicación, presionaremos el botón **Instalar**.

Nuestra aplicación en ejecución es la que se muestra en la figura 10.



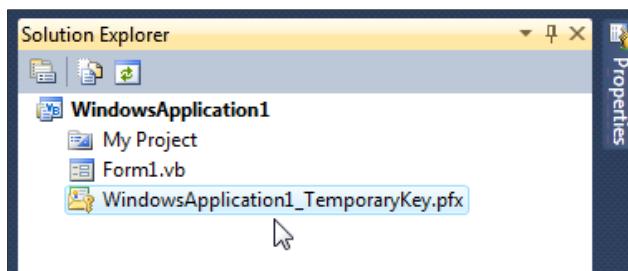
Aplicación Windows en ejecución

Figura 10

Cuando hacemos esto, siempre que ejecutemos la aplicación, el sistema detectará que aceptamos una vez su seguridad, por lo que siempre se ejecutará sin indicarnos ningún mensaje de seguridad.

Ahora bien, supongamos que decidimos modificar parte del código de nuestra aplicación y que por supuesto, cambiamos la versión de la misma.

Acuda antes a la ventana del *Explorador de soluciones* y observe que se ha añadido en la ventana un fichero de nombre *WindowsApplication1_TemporaryKey.pfx* que corresponde a una llave o clave temporal relacionada con el proyecto publicado. Esto es lo que se muestra en la figura 11.



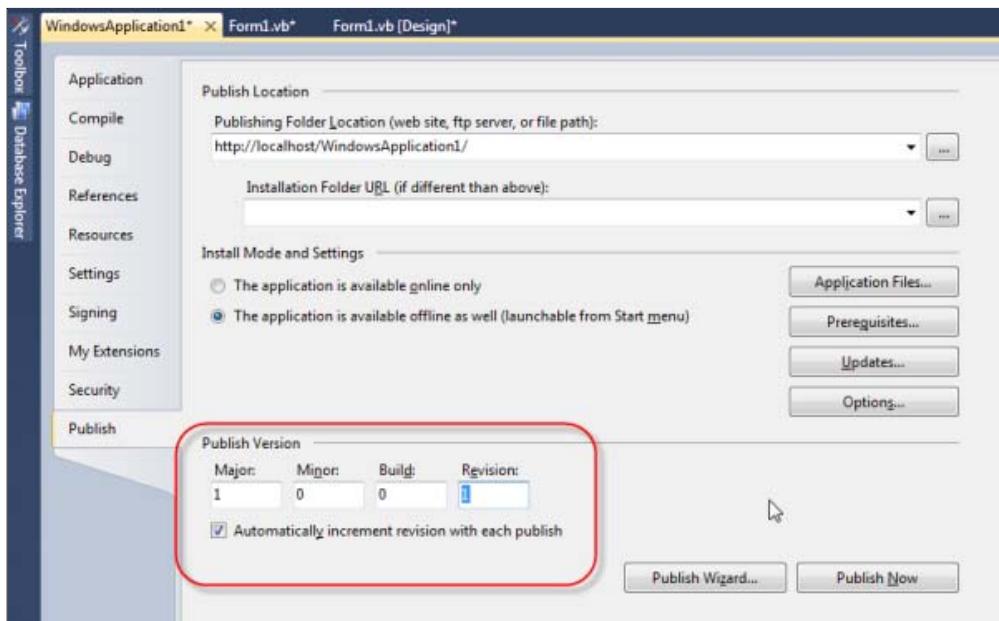
Ventana del Explorador de soluciones con el fichero WindowsApplication1_TemporaryKey.pfx añadido a él

Figura 11

Vamos a actualizar nuestra aplicación y la vamos a cambiar la versión, por lo que ahora escribiremos el siguiente código:

```
Código  
Public Class Form1  
  
    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button1.Click  
        MessageBox.Show("Ejemplo ClickOnce ejecutado a las:" & vbCrLf & _  
            Date.Now.ToShortDateString() & vbCrLf & _  
            Date.Now.ToString("yyyy/MM/dd"))  
    End Sub  
End Class
```

La versión de la aplicación, no tiene relación con la versión de publicación, por ejemplo, la versión de la aplicación la he modificado a 1 1 0 0, pero el instalador autogenera su propio número de versión, el cual podemos cambiar en las propiedades del proyecto, ficha **Publish**. En la figura 12 podemos ver la ficha **Publish** de las propiedades del proyecto.

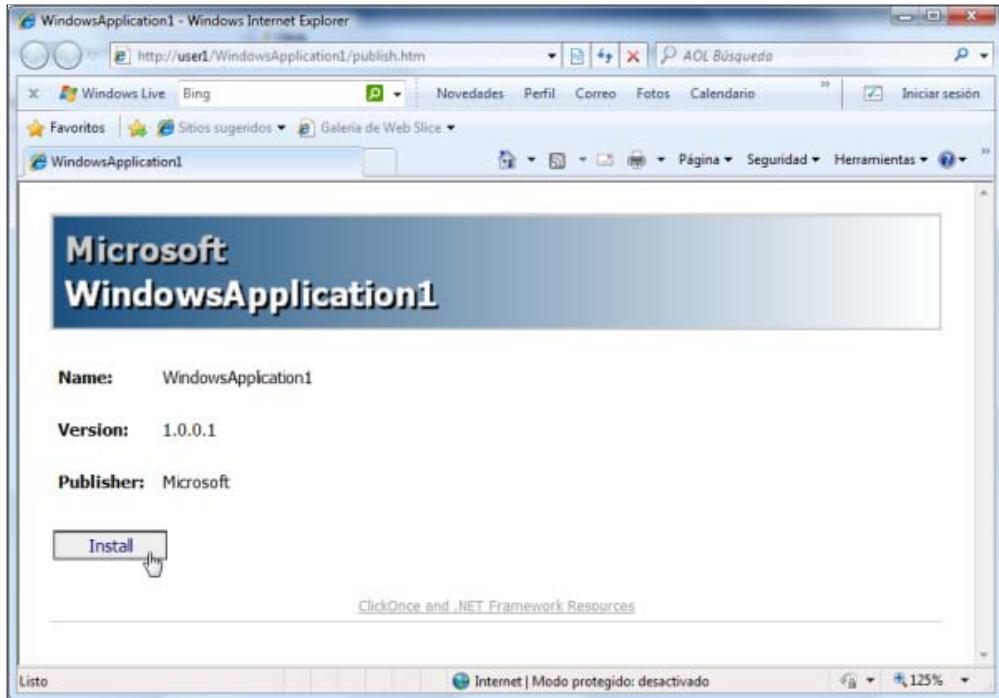


Ventana Web para ejecutar la aplicación

Figura 12

El siguiente paso que he hecho es compilar la aplicación y publicarla nuevamente.

Una vez hecho esto, acudimos a la página Web de la aplicación y presionamos nuevamente el botón **Instalar** como se indica en la figura 13.



Ventana Web para ejecutar la aplicación

Figura 13

La ejecución de la aplicación se realizará sin problemas de manera sencilla y controlada.

Como vemos, la publicación de aplicaciones Windows a través de un Servidor Web, lo que se denomina tecnología de publicación *ClickOnce*, es un proceso de publicación rápido y sencillo, que aporta grandes ventajas y que en otras versiones de .NET trae consigo algunos inconvenientes superados en esta versión de .NET.

Comunidad dotNet:

Si quiere saber más sobre ClickOnce, le recomiendo la lectura del siguiente artículo, escrito en español que complementará y ampliará la información de este tutorial.

► [\(Diciembre 2004\) Implementación de aplicaciones de Windows Forms con ClickOnce](#)

Módulo 4: La librería de clases de .NET

- Colecciones de datos
- Streams en .NET
- Acceso al sistema de archivos
- Acceso a Internet

Contenido

- **Lección 1: Colecciones de datos**

- Los tipos de colecciones de .NET
- Las clases base para crear colecciones
- Colecciones de tipo generic (en breve)

- **Lección 2: Streams en .NET**

- Las clases basadas en Stream
- Manejar un fichero usando FileStream
- Manejar ficheros con StreamReader y StreamWriter
- Cifrar y descifrar un fichero

- **Lección 3: Acceso al sistema de archivos**

- Las clases de System.IO
- Clases para manipular unidades, directorios y ficheros
- Clases para leer o escribir en streams
- El objeto My: My.Computer.FileSystem

- **Lección 4: Acceso a Internet**

- Las clases de System.Net
- Acceder a una página Web
- Acceder a un servidor FTP
- Acceso rápido a la red con My.Computer.Network
- Obtener información de la red con las clases de .NET

Lección 1: Colecciones de datos

- Los tipos de colecciones de .NET
- Las clases base para crear colecciones
- Colecciones de tipo Generic

Introducción

Cuando necesitamos agrupar una serie de datos que de alguna forma están relacionados, en .NET (y en otros "marcos" de desarrollo) tenemos dos formas de hacerlo: usando los arrays (matrices) o usando las colecciones.

De [los arrays](#) ya nos ocupamos anteriormente, así que en esta lección (o capítulo) vamos a ver cómo usar las colecciones que .NET nos ofrece. También veremos, aunque sea de una manera más "sutil", el nuevo tipo de colecciones que la versión 2.0 de .NET Framework introduce y que ha sido arrastrado a las diferentes versiones de .NET Framework: las colecciones *generic*, también conocidas como colecciones genéricas.

Colecciones de datos

- [Los tipos de colecciones de .NET](#)
 - Las colecciones basadas en ICollection
 - Las colecciones basadas en IList
 - La colección ArrayList
 - El tipo de datos de almacenamiento de las colecciones
 - Las colecciones basadas en IDictionary
 - Almacenar valores en una colección tipo IDictionary
 - Cómo se almacenan los elementos de las colecciones IDictionary
 - Obtener todas las claves y valores de una colección IDictionary
- [Las clases base para crear colecciones personalizadas](#)
 - Crear una colección basada en CollectionBase
 - Crear una colección basada en DictionaryBase
 - Crear colecciones personalizadas usando colecciones generic
 - La colección Clientes en versión generic
 - La colección Articulos en versión generic
- [Colecciones de tipo generic](#)
 - Restricciones en los tipos generic

Lección 1: Diseñador de Visual Studio 2008

Los tipos de colecciones de .NET

- Las clases base para crear colecciones
- Colecciones de tipo Generic

Colecciones de datos

En la versión 2010 de Visual Basic, tenemos un amplio abanico de tipos de colecciones, desde colecciones genéricas (o de uso común, para no confundir el término con las colecciones "*generic*"), hasta colecciones especializadas, es decir, colecciones que están pensadas para usarlas de forma muy concreta y que por tanto no nos sirven para usarlas en la mayoría de las ocasiones.

Empecemos viendo los tres tipos básicos de colecciones que podemos utilizar en nuestras aplicaciones de .NET.

Nota:

Las colecciones que vamos a ver a continuación son las colecciones "clásicas" de .NET, (cuyo tipo interno es Object), pero debemos saber que también existen colecciones casi con las mismas características pero que están definidas en el espacio de nombres System.Collections.Generic, las cuales utilizan la nueva "tecnología" de los tipos genéricos (generic) para almacenar los elementos, (cuyo tipo interno puede ser de cualquier tipo).

Los tipos de colecciones de .NET

En .NET Framework existen tres tipos principales de colecciones, éstas dependen del tipo de interfaz que implementan:

- Las colecciones basadas en ICollection
- Las colecciones basadas en la interfaz IList
- Las colecciones basadas en la interfaz IDictionary

Como podemos imaginar, dependiendo del ["contrato" firmado](#) por cada una de estos tipos de colecciones, podremos hacer ciertas operaciones con ellas.

A continuación veremos con más detalle estos tipos de colecciones y cuales son las que podemos usar dependiendo del interfaz que cada una de ellas implemente.

Nota:

La diferencia básica entre estos tipos de colecciones es cómo están almacenados los elementos que contienen, por ejemplo, las colecciones de tipo *IList* (y las directamente derivadas de *ICollection*) solo almacenan un valor, mientras que las colecciones de tipo *IDictionary* guardan un valor y una clave relacionada con dicho valor.

También veremos unas clases base que implementan cada una de estas dos interfaces, las cuales las podemos usar como *base* de nuestras propias colecciones personalizadas.

Las colecciones basadas en *ICollection*

La interfaz *ICollection* es un caso aparte, ya que realmente todas las colecciones de .NET implementan esta interfaz, de hecho, esta interfaz se deriva de *IEnumerable* que es la que nos permite recorrer las colecciones usando bucles *For Each*.

Esta interfaz no la tendremos que usar de forma habitual, ya que realmente el resto de colecciones (e interfaces) útiles ya se derivan de ella, por tanto vamos a centrarnos en las otras dos.

Aunque es importante que tengamos en cuenta que el resto de colecciones implementan *ICollection*, por tanto siempre podremos usar un objeto de este tipo para acceder a cualquier colección.

Independientemente de que todas las colecciones de .NET estén basadas en esta interfaz, hay ciertos tipos de colecciones que solo implementan esta interfaz, por ejemplo las colecciones de tipo *Queue*, *Stack* o *BitArray*, por tanto esas colecciones estarán limitadas a los métodos expuestos por la interfaz *ICollection* y los que esas colecciones implementen de forma independiente.

Por regla general, los tipos que solo se basan en *ICollection* suelen ser colecciones que no necesitan de las características que proporcionan las otras interfaces y, por regla general, nos permiten la manipulación de los datos de una forma básica o elemental, de forma que su uso sea para casos muy concretos.

Por ejemplo, la clase *Queue* nos permite crear fácilmente una colección de tipo FIFO (primero en entrar, primero en salir); por otra parte, con la clase *Stack* podemos crear colecciones del tipo LIFO (último en entrar, el primero en salir), de forma que sean muy útiles para crear "pilas" de datos.

El caso de la otra clase que hemos comentado: *BitArray*, nos sirve para almacenar valores de tipo "bit", en el que cada valor se almacena como un cero o un uno, de forma que podemos tener una colección muy compacta, pero, también muy específica y no de uso general, ya que en este caso particular, los métodos que implementa esta clase están enfocados en la manipulación de valores de tipo *Boolean*, (*False* y *True*), aunque internamente se almacenen como valores cero y uno respectivamente.

Nota:

Realmente la clase *BitArray* no se comporta como una colección "normal", ya que el tamaño de la misma debemos controlarlo nosotros, al igual que ocurre con los arrays, aunque de forma más "fácil", mediante la propiedad *Length*.

Las colecciones basadas en *IList*

La interfaz *IList* se utiliza en las colecciones a las que queremos acceder mediante un índice, por ejemplo, los arrays realmente están basados en esta interfaz, y tal como pudimos comprobar, la única forma que tenemos de acceder a los elementos de un array, (y por extensión a los elementos de las colecciones basadas en *IList*), es mediante un índice numérico.

Existen tres tipos principales de colecciones que implementan esta interfaz:

- Las de solo lectura, colecciones que no se pueden modificar. Este tipo de colecciones suelen basarse en la clase abstracta *ReadOnlyCollectionBase*.
- Las colecciones de tamaño fijo, no se pueden quitar ni añadir elementos, pero si modificarlos. Por ejemplo, las colecciones basadas en *Array* son de tamaño fijo.
- Las de tamaño variable permiten cualquier tipo de adición, eliminación y modificación. La mayoría de las colecciones suelen ser de este tipo, es decir, nos permiten dinámicamente añadir o eliminar elementos.

Existe un gran número de colecciones en .NET que implementan esta interfaz, (sobre todo las colecciones basadas en controles), entre las que podemos destacar las siguientes:

- *ArrayList*, la colección "clásica" para este tipo de interfaz. Contiene todos los miembros habituales en este tipo de colecciones.
- *CollectionBase*, una clase abstracta para poder crear nuestras propias colecciones basadas en *IList*.
- *StringCollection*, una colección especializada que solo puede contener valores de tipo cadena.

La colección *ArrayList*

Tal como hemos comentado, el tipo de colección que se usa como referencia a la hora de hablar de las colecciones basadas en la interfaz *IList*, es *ArrayList*.

Esta colección permite añadir, eliminar y modificar fácilmente los elementos que contiene. También podemos recorrerlos mediante un bucle *For* accediendo a los elementos por medio de un índice e incluso mediante un bucle del tipo *For Each*.

Al igual que ocurre con los arrays, el índice inferior es siempre el cero y los elementos se almacenan de forma consecutiva, es decir, si añadimos dos elementos a una colección de tipo *ArrayList* (y a las que implementen la interfaz *IList*), el primero ocupará la posición cero y el segundo la posición uno. La ventaja de trabajar con las colecciones es que no debemos preocuparnos de reservar memoria cada vez que vayamos a añadir un nuevo elemento, simplemente usamos el método *Add* y asunto arreglado.

Lo mismo ocurre a la hora de quitar elementos de una colección, no tenemos que preocuparnos demasiado por el espacio dejado al quitar elementos, de eso se encarga el propio .NET, nosotros simplemente debemos llamar al método *Remove* o *RemoveAt* indicando respectivamente el elemento a eliminar o el índice en el que se encuentra almacenado.

Truco:

Si decidimos eliminar varios elementos de una colección de tipo *IList* (o de un array), lo normal es que lo hagamos usando un bucle *For*; si este es el caso, para evitar una posible excepción, (realmente no es posible, sino con toda certeza segura), debemos recorrer el bucle desde el final hacia adelante, con idea de que al cambiar el número de elementos no falle al intentar a acceder a un elemento que ya no existe.

El tipo de datos de almacenamiento de las colecciones

Estos elementos internamente están almacenados como objetos del tipo *Object*, por tanto podemos añadir cualquier tipo de datos a una colección de este tipo, ya que todos los tipos de datos de .NET están basado en la clase *Object*.

El problema con este tipo de colecciones es que siempre que queramos acceder a uno de los elementos que contiene, debemos hacer una conversión al tipo adecuado, es decir, si en una colección de este tipo guardamos objetos de tipo Cliente y queremos acceder a uno de ellos, debemos hacer una conversión (*cast*) del tipo *Object* al tipo Cliente, ya que si no lo hacemos y tenemos activada *Option Strict* (la opción para las comprobaciones estrictas), se

producirá un error si hacemos algo como esto:

```
Dim lista As New ArrayList
lista.Add(New Cliente("Pepe"))
lista.Add(New Cliente("Lola"))

Dim unCliente As Cliente
' ¡Error!
unCliente = lista(0)
...
```

Por tanto, la última línea deberíamos escribirla de una de estas dos formas:

```
' Usando CType para hacer la conversión
unCliente = CType(lista(0), Cliente)
...
' Usando DirectCast para hacer la conversión (más recomendable)
unCliente = DirectCast(lista(0), Cliente)
```

Otro de los problemas que tienen las colecciones "normales" es que en algunos casos, particularmente cuando almacenamos tipos por valor, el rendimiento se ve bastante mermado, ya que el runtime de .NET (el CLR) debe hacer lo que en inglés se conoce como boxing/unboxing, es decir, convertir un tipo por valor en uno por referencia cuando va a guardarlo en la colección (boxing), y el proceso inverso cuando lo queremos recuperar (unboxing).

Nota:

Por suerte, en Visual Basic 2010 tenemos otra forma de mejorar el rendimiento de las colecciones, y es mediante las colecciones "generic", de esto, nos ocuparemos más adelante.

Otras de las ventajas de las colecciones de .NET, no solo las basadas en la interfaz *IList*, es que proporcionan una gran cantidad de métodos que nos facilitan la manipulación de ese tipo de datos. Por ejemplo, tenemos métodos para clasificar el contenido de las colecciones, (aunque esos objetos deben implementar la interfaz *IComparable*, tal como vimos en [el último ejemplo del capítulo de las interfaces](#)), además tienen métodos para hacer copias, buscar elementos y muchos etcéteras más.

Las colecciones basadas en *IDictionary*

El otro grupo de colecciones que podemos encontrar en .NET son las colecciones basadas en la interfaz *IDictionary*. Éstas, a diferencia de las colecciones *IList*, siempre mantienen el par clave/valor, ya que la forma de acceder a los elementos es mediante una clave única. Por tanto, cada vez que añadimos un elemento a una colección de este tipo tendremos que indicar una clave y un valor. Cada valor estará relacionado con su correspondiente clave.

Sabiendo esto, es fácil adivinar que si queremos acceder a un elemento, lo normal es que lo hagamos usando la clave indicada al añadirlo. Los que hayan trabajado anteriormente con Visual Basic 6.0, (o lo estén haciendo actualmente), puede que piensen que también se podrá acceder a cada elemento mediante un índice numérico, ya que el objeto *Collection* de VB6, (que aún sigue existiendo en Visual Basic 2010), nos permite indicar una clave para cada elemento, y además de acceder a esos elementos mediante la clave, podemos hacerlo mediante un valor numérico (índice). Pero en las colecciones basadas en *IDictionary*, salvo casos muy especiales, siempre accederemos a los valores contenidos mediante la clave y "nunca" mediante un índice que haga referencia a la posición dentro de la colección, entre otras cosas porque cuando almacenamos valores en este tipo de colecciones, éstos no se guardan en el mismo orden en que fueron añadidos.

Nota:

Si bien la clase Collection está disponible en la nueva versión de Visual Basic, ésta tiene algunas mejoras con respecto a la que tiene VB6, entre esas mejoras están dos nuevos métodos que nos facilitarán su uso: el método Clear con el que podemos eliminar todos los elementos de la colección y el método Contains, con el que podemos averiguar si un determinado elemento está en la colección.

Entre las colecciones basadas en la interfaz *IDictionary* podemos destacar:

- *Hashtable*, es la colección por excelencia de las basadas en *IDictionary*. Los elementos se organizan basándose en el código *hash* de las claves.
- *DictionaryBase*, es una clase abstracta que podemos usar como base de nuestras propias colecciones de tipo diccionario.
- *ListDictionary*, es una colección con mayor rendimiento que *Hashtable* pensada para trabajar con 10 o menos elementos.
- *HybridDictionary*, es una colección especial en la que si hay 10 o menos elementos, se utiliza una colección *ListDictionary* y si contiene más elementos se utiliza una colección *Hashtable*.
- *SortedList*, es una colección en la que los elementos están clasificados por las claves. Internamente utiliza una mezcla entre *Hashtable* y *Array*, según la forma en que se accedan a esos elementos.

Almacenar valores en una colección tipo *IDictionary*

Para añadir nuevos elementos a una colección de tipo *IDictionary* siempre tendremos que indicar la clave y el valor, la clave no puede ser un valor nulo, (*Nothing*), pero puede ser de cualquier tipo. El valor también puede ser de cualquier tipo y en este caso si que se admiten valores nulos.

Los elementos los añadiremos usando el método *Add*, al que habrá que indicar primero la clave y después el valor:

```
Dim valores() As String = {"uno", "dos", "tres"}  
  
Dim dic As New System.Collections.Hashtable  
  
For i As Integer = 0 To valores.Length - 1  
    dic.Add(valores(i), "El valor de " & valores(i))  
Next
```

Cómo se almacenan los elementos de las colecciones *IDictionary*

Tal como hemos comentado, las colecciones que implementan la interfaz *IDictionary* siempre almacenan un par de datos: la clave y el valor propiamente dicho, por tanto cada vez que queramos acceder a un valor, debemos usar la clave asociada con dicho valor. Al menos esa es la forma habitual, ya que como veremos, también podremos acceder a esos valores directamente.

Debido a esta característica, para acceder a los elementos de este tipo de colecciones por medio de un bucle del tipo *For Each*, debemos usar una clase llamada *DictionaryEntry*, esta clase tiene dos propiedades, una contiene la clave y otra el valor. Por tanto, cuando usemos un bucle *For Each*, el tipo de objeto usado para acceder a los elementos de la colección será *DictionaryEntry*, tal como vemos en el siguiente código:

```
For Each de As DictionaryEntry In dic  
    Console.WriteLine("{0} = {1}", de.Key, de.Value)  
Next
```

Obtener todas las claves y valores de una colección *IDictionary*

Independientemente de que podamos recorrer los valores contenidos en una colección de tipo *IDictionary* usando un objeto de tipo *DictionaryEntry*, habrá ocasiones en las que realmente nos interesen tener solo los valores e incluso solo las claves, en estos casos, podemos usar dos propiedades que la interfaz *IDictionary* define: *Keys* y *Values*. Estas propiedades devuelven un objeto del tipo *ICollection* con las claves y valores respectivamente.

Al ser objetos *ICollection*, solo podremos usarlos para recorrerlos por medio de un bucle *For Each*, ya que las colecciones *ICollection* no tienen ningún método que nos permita acceder a los elementos que contiene usando un índice. En el siguiente código mostramos todas las claves de la colección creada en el ejemplo anterior:

```
For Each clave As String In dic.Keys  
    Console.WriteLine(clave)  
Next
```

Nota:

Siempre que usemos un bucle For Each para recorrer los elementos (o datos) de una colección, solo tendremos acceso de solo lectura a esos datos, es decir, no podremos modificarlos usando la variable por medio de la que accedemos a ellos.

Lección 1: Diseñador de Visual Studio 2008

- Los tipos de colecciones de .NET

Las clases base para crear colecciones

- Colecciones de tipo Generic

Las clases base para crear colecciones personalizadas

Tal como hemos visto, en el espacio de nombres *System.Collections* tenemos dos clases abstractas que podemos usar como clases base para crear nuestras propias colecciones.

Dependiendo que queramos crear una colección basada en *IList*, por ejemplo para acceder a los elementos mediante un índice numérico, o bien una colección basada en *IDictionary*, para almacenar los elementos usando el par clave/valor, tendremos que usar la clase *CollectionBase* o *DictionaryBase*.

Estas clases ya tienen cierta funcionalidad que podremos aprovechar para no tener que reinventar la rueda, (ésa es la "gracia" de la herencia), y lo único que tendremos que hacer es definir nuestros propios métodos o propiedades para que la colección actúe como nosotros decidamos y, lo más importante, para que solo acepte los tipos de datos que realmente queramos.

Por ejemplo, si queremos almacenar datos de tipo Cliente y queremos acceder a esos datos solo por un índice numérico, podríamos basar nuestra colección en *CollectionBase*, pero si lo que necesitamos es una colección que contenga, por ejemplo, objetos de tipo Artículo, nos podría interesar crear una colección basada en *DictionaryBase* para que de esta forma podamos acceder a cada uno de los elementos por medio del código del artículo.

A continuación veremos el código (reducido) para crear estos dos tipos de colecciones personalizadas.

Nota:

En las definiciones de las colecciones que vamos a mostrar, no hemos añadido ninguna funcionalidad extra, sino que hemos creado las clases/colecciones para que tengan un funcionamiento parecido al de las colecciones "normales". La diferencia principal con esas colecciones "normales" es que estas dos clases/colecciones que vamos a mostrar, solo admitirán elementos de un tipo concreto.

Esto lo hemos hecho así para que podamos comparar y comprobar la facilidad que ahora tenemos si usamos colecciones del espacio de nombres

Crear una colección basada en CollectionBase

A continuación vamos a ver un ejemplo de una colección personalizada basada en *CollectionBase* y cómo usarla. Esta colección almacenará elementos de un tipo definido por nosotros: **Cliente**.

Primero veamos una clase **Cliente** muy simple, pero que implementa la interfaz *IComparable*, de forma que se puedan clasificar sus elementos por el campo Apellidos. También define el método *ToString*, ya que esta es una recomendación que siempre deberíamos seguir, ya que muchas de las clases de punto NET utilizan este método para mostrar el contenido de los objetos.

```

''' <summary>
''' Clase Cliente
''' </summary>
''' <remarks>
''' Esta clase se puede clasificar por el campo Apellidos
''' </remarks>
Public Class Cliente
    Implements System.IComparable
    '
    Public Nombre As String
    Public Apellidos As String
    '
    Public Sub New(ByVal nombre As String, ByVal apellidos As String)
        Me.Nombre = nombre
        Me.Apellidos = apellidos
    End Sub
    '
    Public Overrides Function ToString() As String
        Return Apellidos & ", " & Nombre
    End Function
    '
    Public Function CompareTo(ByVal obj As Object) As Integer _
        Implements System.IComparable.CompareTo
        If TypeOf obj Is Cliente Then
            Dim cli As Cliente = DirectCast(obj, Cliente)
            Return String.Compare(Me.Apellidos, cli.Apellidos)
        Else
            Return 0
        End If
    End Function
End Class

```

En el siguiente código tenemos la definición de la clase/colección **Clients**, que al estar derivada de *CollectionBase* tendrá todos los miembros definidos en esa clase abstracta, (que solo se puede usar para crear clases derivadas); y en la que hemos definido los métodos más habituales, así como una propiedad por defecto que nos permite acceder a los elementos mediante un índice numérico.

Como podemos comprobar, en los métodos que hemos definido, realmente no tenemos que hacer demasiadas cosas, ya que en el código que hemos escrito en esos nuevos miembros nos apoyamos en las colecciones internas proporcionadas por la clase base: *List*, que es una colección basada en *IList* que contiene los elementos, e *InnerList* que es una colección de tipo *ArrayList* que también hace referencia a la colección *List*.

En la propiedad *Item*, que es la propiedad predeterminada o indizador, cuando devolvemos el valor indicado por el índice numérico, tenemos que hacer una conversión para que se devuelva un objeto de tipo *Cliente* en lugar de uno de tipo *Object* que es como realmente se almacena en la colección.

```
''' <summary>
```

```

''' Colección de tipo Cliente basada en IList
''' </summary>
''' <remarks></remarks>
Public Class Clientes
    Inherits System.Collections.CollectionBase

    Public Function Add(ByVal value As Cliente) As Integer
        Return List.Add(value)
    End Function

    Public Function Contains(ByVal value As Cliente) As Boolean
        Return List.Contains(value)
    End Function

    Public Function IndexOf(ByVal value As Cliente) As Integer
        Return List.IndexOf(value)
    End Function

    Public Sub Insert(ByVal index As Integer, ByVal value As Cliente)
        List.Insert(index, value)
    End Sub

    Default Public Property Item(ByVal index As Integer) As Cliente
        Get
            Return DirectCast(List(index), Cliente)
        End Get
        Set(ByVal value As Cliente)
            List(index) = value
        End Set
    End Property

    Public Sub Remove(ByVal value As Cliente)
        List.Remove(value)
    End Sub

    Public Sub Sort()
        InnerList.Sort()
    End Sub
End Class

```

Para usar esta colección, lo haremos como es costumbre en las colecciones de tipo *IList*:

```

Sub Main()
    Dim col As New Clientes

    col.Add(New Cliente("Pepe", "López"))
    col.Add(New Cliente("Loli", "Pérez"))
    col.Add(New Cliente("Eva", "Kelo"))
    col.Add(New Cliente("Juan", "Salvador"))
    col.Add(New Cliente("Miguel", "Andrade"))

    col.Sort()

    For i As Integer = 0 To col.Count - 1
        Console.WriteLine(col(i).Apellidos)
    Next

    col.RemoveAt(2)

    For Each cli As Cliente In col
        Console.WriteLine(cli.ToString())
    Next
End Sub

```

Crear una colección basada en DictionaryBase

En el siguiente código veremos cómo definir una colección personalizada basada en la clase abstracta *DictionaryBase*. Esta colección almacenará objetos del tipo **Artículo**. Esos objetos se almacenarán indicando como clave el código del artículo.

Veamos el código y comentaremos las cosas dignas de resaltar.

La clase **Articulo** no tiene nada que resaltar, es una clase "normalita".

```
''' <summary>
''' Clase artículo
''' </summary>
''' <remarks>
''' Clase para el ejemplo de colección derivada de DictionaryBase
''' </remarks>
Public Class Articulo
    Public Código As String
    Public Descripción As String
    Public PVP As Decimal

    Sub New(    ByVal código As String, _
              ByVal descripción As String, _
              ByVal precio As Decimal)
        Me.Código = código
        Me.Descripción = descripción
        Me.PVP = precio
    End Sub

    Public Overrides Function ToString() As String
        Return Código & ", " & Descripción
    End Function
End Class
```

La clase/colección la derivamos de *DictionaryBase* para que tenga todas las "características" expuestas por esa clase abstracta, a la que le añadimos nuevos métodos y propiedades para darle funcionalidad. Tal como hicimos en la colección **Clientes**, nos apoyamos en las colecciones internas de la clase base para realizar el trabajo de esos nuevos miembros.

```
''' <summary>
''' Colección Clientes basada en IDictionary
''' </summary>
''' <remarks>
''' </remarks>
Public Class Articulos
    Inherits System.Collections.DictionaryBase

    Default Public Property Item(ByVal key As String) As Articulo
        Get
            Return DirectCast(Dictionary(key), Articulo)
        End Get
        Set(ByVal value As Articulo)
            Dictionary(key) = value
        End Set
    End Property

    Public ReadOnly Property Keys() As ICollection
        Get
            Return Dictionary.Keys
        End Get
    End Property

    Public ReadOnly Property Values() As ICollection
        Get
            Return Dictionary.Values
        End Get
    End Property

    Public Sub Add(ByVal key As String, ByVal value As Articulo)
        Dictionary.Add(key, value)
    End Sub

    Public Function Contains(ByVal key As String) As Boolean
        Return Dictionary.Contains(key)
    End Function
```

```

    Public Sub Remove(ByVal key As String)
        Dictionary.Remove(key)
    End Sub
End Class

```

La forma de usar esta colección es la misma que cualquier colección basada en *IDictionary*.

```

Dim col As New Articulos

col.Add("uno", New Articulo("Uno", "Art. Uno", 10.6D))
col.Add("dos", New Articulo("Dos", "Art. Dos", 22))
col.Add("tres", New Articulo("Tres", "Art. Tres", 45.55D))

For Each de As DictionaryEntry In col
    Dim art As Articulo
    art = DirectCast(de.Value, Articulo)
    Console.WriteLine("{0}, {1}, {2}", de.Key, art.Descripcion, art.PVP)
Next

col.Remove("dos")

For Each s As String In col.Keys
    Console.WriteLine("{0}, {1}", s, col(s).Descripcion)
Next

```

Crear colecciones personalizadas usando colecciones generic

Hasta esta versión de Visual Basic, si queríamos crear colecciones "fuertemente tipadas", es decir, colecciones que solo admitieran datos del tipo que nosotros quisiéramos, teníamos que hacerlo con un código parecido al que hemos visto.

Pero si nuestra intención es crear colecciones que "simplemente" contengan elementos de un tipo determinado, por ejemplo objetos de tipo **Cliente** o **Articulo**, pero que no tengan ninguna funcionalidad extra a las que de forma predeterminada tienen las clases base para crear colecciones, no es necesario que creemos nuestros propias clases/colección, ya que Visual Basic 2010 puede crear colecciones con esas características sin necesidad de crear un clase específica.

Veamos primero el código equivalente usando colecciones del espacio de nombres *Generic* con respecto a los dos tipos de colecciones anteriores, y después explicaremos un poco de que va todo esto de los *generics*.

La colección Clientes en versión generic

La colección **Clientes** es una colección que solo acepta elementos del tipo **Cliente** y a la que podemos acceder mediante un índice numérico, por tanto debemos buscar una colección del espacio de nombres *System.Collections.Generic* que nos ofrezca esa misma funcionalidad, y esa colección es: *List*.

Debido a que las colecciones *generic* necesitan saber el tipo de datos que van a almacenar, no necesitamos crear una clase/colección para almacenar los elementos del tipo **Cliente**, simplemente tendremos que indicar en el constructor de esa colección que tipo debe contener.

En el siguiente código tenemos la forma de declarar la colección de tipo *List* y cómo acceder a los elementos que tienen, como podrá comprobar es prácticamente el mismo que el mostrado en el ejemplo de la colección basada en *CollectionBase*.

```

' Colección generic equivalente a ArrayList
Console.WriteLine("Ejemplo usando Generic.List")
Dim colCli As New System.Collections.Generic.List(Of Cliente)

colCli.Add(New Cliente("Pepe", "López"))

```

```

colCli.Add(New Cliente("Loli", "Pérez"))
colCli.Add(New Cliente("Eva", "Kelo"))
colCli.Add(New Cliente("Juan", "Salvador"))
colCli.Add(New Cliente("Miguel", "Andrade"))

colCli.Sort()

For i As Integer = 0 To colCli.Count - 1
    Console.WriteLine(colCli(i).Apellidos)
Next

Console.WriteLine()

' Elimina el elemento de la posición 3
' (pero después de haberlo clasificado)
colCli.RemoveAt(2)

For Each cli As Cliente In colCli
    Console.WriteLine(cli.ToString())
Next

```

El "quid" de la cuestión está en la forma de declarar la variable **colCli**, en la que le indicamos que el tipo de datos que contendrá la colección es "de" **Cliente**:

```
Dim colCli As New List(Of Cliente)
```

Por lo demás, el código a usar para acceder a los elementos, eliminarlos, etc., es el mismo que con cualquier otra colección basada en *IList*, pero con el "detalle" de que dicha colección "sepa" manejar elementos del tipo **Cliente**.

Si no fuera así, esta línea produciría un error, ya que estamos accediendo a un tipo de datos que define una propiedad llamada **Apellidos**:

```
Console.WriteLine(colCli(i).Apellidos)
```

La colección Articulos en versión generic

Como vimos, la colección **Articulos** solo acepta elementos del tipo **Articulo**, pero como es una colección de tipo *IDictionary* cada vez que añadimos algún elemento o queremos acceder a cualquiera de los contenidos en ella, debemos indicar también una clave. Por tanto necesitamos una colección *generic* que tenga esas mismas "características" y la que nos puede servir es la clase *System.Collections.Generic.Dictionary*.

A continuación tenemos el código para manejar objetos de tipo **Articulo** en una colección tipo *IDictionary*, pero como veremos, en este caso hay que usar otra de las clases del espacio de nombres *Generic* que nos sea útil para hacer el bucle *For Each*, ya que para usar la clase *Generic.Dictionary* debemos indicar tanto el tipo del valor a almacenar como el de la clave.

Veamos el código y después entraremos en más detalles:

```

' Colección generic equivalente a Hashtable (IDictionary)
Console.WriteLine("Ejemplo usando Generic.Dictionary")

Dim colArt As New System.Collections.Generic.Dictionary(Of String, Articulo)

colArt.Add("uno", New Articulo("Uno", "Art. Uno", 10.6D))
colArt.Add("dos", New Articulo("Dos", "Art. Dos", 22))
colArt.Add("tres", New Articulo("Tres", "Art. Tres", 45.55D))

For Each de As KeyValuePair(Of String, Articulo) In colArt
    Dim art As Articulo = de.Value
    Console.WriteLine("{0}, {1}, {2}", de.Key, art.Descripcion, art.PVP)
Next

Console.WriteLine()

```

```
colArt.Remove( "dos" )

For Each s As String In colArt.Keys
    Console.WriteLine("{0}, {1}", s, colArt(s).Descripcion)
Next
```

Nuevamente "el truco" está en la forma de declarar la variable **colArt**, en la que le decimos que la colección *Dictionary* usará claves de tipo *String* y valores del tipo **Articulo**:

```
Dim colArt As New Dictionary(Of String, Articulo)
```

Para acceder a los elementos de esta colección por medio de un bucle *For Each*, en lugar de usar una variable de tipo *DictionaryEntry* debemos usar una del tipo *generic KeyValuePair* en la que debemos especificar los tipos de datos que contiene la colección:

```
For Each de As KeyValuePair(Of String, Articulo) In colArt
```

Podemos comprobar que los tipos de datos que esta colección contiene son de tipo *String* para las claves y de tipo **Articulo** para los valores. Dentro del bucle hacemos una asignación a la variable **art**, pero en este caso, a diferencia de cuando usamos la colección "**Articulos**" basada en *DictionaryBase*, no es necesario hacer una conversión explícita del tipo que tiene **de.Value** a un tipo **Articulo**, ya que **de.Value** es del tipo **Articulo**.

Esto mismo lo podemos comprobar al acceder a un elemento dentro del segundo bucle, esos son los tipos de datos que deben tener, ya que de lo contrario, no podríamos acceder a la propiedad **Descripcion** del objeto almacenado:

```
Console.WriteLine("{0}, {1}", s, colArt(s).Descripcion)
```

Lección 1: Diseñador de Visual Studio 2008

- Los tipos de colecciones de .NET
- Las clases base para crear colecciones

Colecciones de tipo Generic

Colecciones de tipo generic

En los ejemplos que acabamos de ver podemos apreciar que las colecciones del espacio de nombres *Generic* son bastante "potentes" o al menos nos pueden facilitar la tarea de crear colecciones fuertemente tipadas, ya que podremos indicar que tipo de datos son los que queremos que contenga.

En la siguiente imagen podemos ver cómo al declarar una colección *Generic.Dictionary* nos pide tanto el tipo de datos de la clave como del valor:

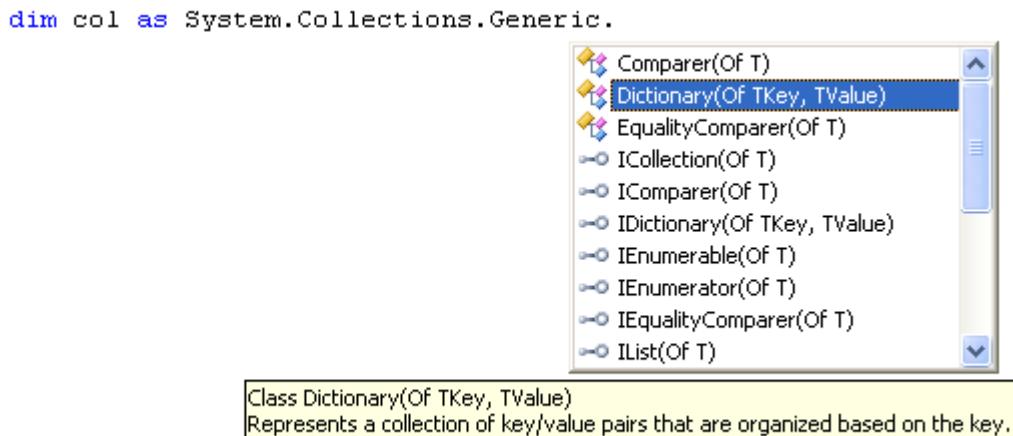


Figura 4.1. IntelliSense en las colecciones generics

La principal ventaja de estas colecciones es que debemos indicar siempre el tipo de datos que va a contener y en el caso de las colecciones *Dictionary* también tenemos que indicar el tipo de datos para las claves.

Estos tipos de datos pueden ser cualquiera de los que están definidos en el propio .NET Framework o los que hayamos definido nosotros.

Como podemos comprobar, siempre que vamos a declarar una colección *Generic* vemos la "palabra clave" *Of*, esta instrucción lo que hace es indicar cual será el tipo de datos que contendrá la colección. Y ese será el único tipo de datos que la colección podrá contener.

Restricciones en los tipos generic

La ventaja de este tipo de "restricción" del tipo que puede contener una colección de tipo *Generic* es que en lugar de indicar un tipo concreto, podemos indicar también un tipo algo más "genérico", (genérico en el sentido de no tan estricto), por ejemplo, si tenemos un tipo de datos que implementa una interfaz o que se deriva de una clase, podemos indicar ese tipo después de *Of* y en ese caso solamente se admitirán objetos que implemente o se derive de ese tipo.

Nota:

Para facilitar la lectura de los tipos generics, se recomienda leer las declaraciones genéricas usando "de" y el tipo cuando nos encontramos con un tipo después de *Of*. Por ejemplo: *List(Of Cliente)* lo leeremos como una colección *List de Cliente*. Precisamente esa es la razón de que en Visual Basic 2010 se haya optado por el uso de la instrucción *Of* para los tipos generics.

Por ejemplo, si tenemos una clase de tipo **Persona** y la utilizamos como clase base de un tipo **Cliente** y otro de tipo **Empleado**, podríamos crear una colección *Generic.List* que admita solo elementos del tipo **Persona**, con lo cual podemos añadir tanto elementos de tipo **Cliente** y/o de tipo **Empleado**, ya que estas dos clases se derivan de la clase **Persona**. En el siguiente código tenemos las definiciones de estas tres clases y el código para crear la colección *generic*:

La clase Persona:

```
Public Class Persona
    Private _Nombre As String
    Public Property Nombre() As String
        Get
            Return _Nombre
        End Get
        Set(ByVal value As String)
            _Nombre = value
        End Set
    End Property

    Private _Apellidos As String
    Public Property Apellidos() As String
        Get
            Return _Apellidos
        End Get
        Set(ByVal value As String)
            _Apellidos = value
        End Set
    End Property
End Class
```

La clase Cliente

```
Public Class Cliente
    Inherits Persona

    Public Sub New(ByVal empresa As String)
        Me.Nombre = empresa
    End Sub

    Public Overrides Function ToString() As String
        Return Nombre
    End Function
End Class
```

```
    End Function  
End Class
```

La clase Empleado

```
Public Class Empleado  
    Inherits Persona  
  
    Public Sub New(ByVal nombre As String, ByVal apellidos As String)  
        Me.Nombre = nombre  
        Me.Apellidos = apellidos  
    End Sub  
  
    Public Overrides Function ToString() As String  
        Return Apellidos & ", " & Nombre  
    End Function  
End Class
```

El código para usar estas clases

```
Dim col As New System.Collections.Generic.List(Of Persona)  
  
col.Add(New Cliente("Construcciones Pepe"))  
col.Add(New Empleado("Juan", "Pérez"))  
  
For Each p As Persona In col  
    Console.WriteLine(p.Nombre)  
Next
```

Las colecciones y clases *generic* son bastante "potentes" y en principio fáciles de utilizar, con el valor añadido (o ventaja) es que nosotros también podemos crear nuestros propios tipos de datos que utilicen esta nueva "tecnología", pero su tratamiento en profundidad sobrepasa el objetivo de este curso, aunque con lo aquí expuesto creemos que el lector está preparado para investigar por su cuenta, con la ayuda de la documentación de Visual Studio 2010 y de los muchos recursos que están disponibles tanto en formato "artículo Web" y en libros especializados en las novedades de Visual Basic 2010.

Lección 2: Streams en .NET

- Las clases basadas en Stream
- Manejar un fichero usando FileStream
- Manejar ficheros con StreamReader y StreamWriter
- Cifrar y descifrar un fichero

Introducción

Al empezar a trabajar con .NET, uno de los "grandes" cambios que notaremos los que estamos habituados a desarrollar con otros lenguajes es todo lo relacionado con el acceso al contenido de los ficheros.

En .NET, y por tanto en Visual Basic 2010, el acceso a ficheros se realiza por medio de los *streams* (o usando la traducción de la documentación: las secuencias).

Con los *streams* no solo podemos acceder a los ficheros de disco, sino que también podemos acceder a otros tipos de "secuencias" o flujos de datos, desde *streams* de memoria a *streams* para enviar información a través de Internet.

Toda esta transmisión o flujo de información se realiza mediante una serie de métodos de lectura y escritura que están básicamente encapsulados en la clase abstracta *Stream*. Esta clase será la clase base de todas aquellas que de alguna forma tienen que transmitir cierta información entre la fuente de datos y nuestra aplicación.

En este capítulo trataremos de las clases basadas en *Stream* que con más frecuencia utilizaremos, sobre todo en lo relacionado con el acceso a ficheros, que es al fin y al cabo la utilidad principal de este tipo de "secuencias".

Streams en .NET

- [Streams en .NET](#)
- [Las clases basadas en Stream](#)
 - [Manejar un fichero usando FileStream](#)
 - [Manejar un fichero usando StreamReader y StreamWriter](#)
 - Asegurarnos que el fichero se cierra
 - Liberar recursos: Using... End Using
 - [Ejemplo de para cifrar y descifrar un fichero](#)

Lección 2: Streams en .NET

- Las clases basadas en Stream
- Manejar un fichero usando FileStream
- Manejar ficheros con StreamReader y StreamWriter
- Cifrar y descifrar un fichero

Streams en .NET

Según hemos comentado en la introducción, los *streams* (o secuencias o flujos) nos permiten abstraernos de la forma en que están implementados los procesos de acceso a los ficheros u otros recursos. Todas las clases que manejan este tipo de flujos de datos están basadas (directa o indirectamente) en la clase *Stream*, la cual nos ofrece ciertos métodos que nos permiten, entre otras cosas, leer y escribir en esos flujos de información.

Además de poder leer o escribir, también podemos realizar búsquedas o, dicho de otro modo, podemos movernos a través de esa secuencia de datos. Pero debido a que no todos los flujos de datos nos permiten realizar todas las operaciones de la clase *Stream*, existen ciertas propiedades por medio de las cuales podemos saber si se permiten todas las operaciones "básicas" que normalmente podremos hacer con los *streams*. Por ejemplo, es posible que no podamos leer o escribir en una secuencia o que no podamos cambiar la posición del "puntero" de lectura o escritura. Para todas estas comprobaciones podremos usar las propiedades *CanRead*, *CanWrite* o *CanSeek*, pero creo que antes de entrar en detalles, deberíamos ver algunos de las clases que .NET pone a nuestra disposición para poder realizar este tipo de operaciones con "las secuencias" de datos.

Las clases basadas en Stream

Entre las clases que están basadas en esta clase abstracta tenemos las siguientes:

- *BufferedStream*, clase abstracta que representa un buffer de almacenamiento para operaciones de lectura y escritura de otro *stream*.
- *DeflateStream*, permite la compresión y descompresión de *streams* usando el algoritmo *Deflat*.
- *GZipStream*, usada para comprimir y descomprimir *streams*.
- *FileStream*, nos permite una forma básica de acceder y manipular ficheros.
- *MemoryStream*, crear un *stream* que se almacena en la memoria como una secuencia de bytes.
- *NetworkStream*, proporciona una secuencia de datos para el acceso a la red.
- *CryptoStream*, un *stream* usado para encriptar otros streams.

Nota:

Las clases *DeflateStream* y *GZipSteam* están incluidas en el espacio de

nombres System.IO.Compression.

La clase CryptoStream está incluida en el espacio de nombres

System.Security.Cryptography.

La clase NetworkStream está incluida en el espacio de nombres

System.Net.Sockets.

Además de estas clases que se derivan directamente de la clase *Stream*, y que normalmente se usan como "secuencias" a usar por otras clases de entrada/salida, tenemos otras que nos permitirán acceder a esas secuencias de datos de una forma más directa, (algunas de estas las veremos con algo de más detalle en el próximo capítulo dedicado al sistema de archivos de .NET), por ejemplo:

- *BinaryReader / BinaryWriter*, lee o escribe tipos primitivos como valores binarios utilizando una codificación específica.
- *StreamReader / StreamWriter*, clases para leer y escribir caracteres en ficheros utilizando una codificación determinada.
- *StringReader / StringWriter*, implementa *TextReader* o *TextWriter* para leer o escribir en una cadena.
- *TextReader / TextWriter*, clases abstractas para leer o escribir en una secuencia de caracteres.

Cuando trabajamos con los *streams* debemos olvidarnos de las "cosas simples" y debemos tener en cuenta que trataremos casi siempre con secuencias de bytes, ya que al fin y al cabo esa es la forma de almacenar la información en los *streams*. Por tanto cuando veamos los ejemplos que la documentación de Visual Basic 2010 nos proporciona no debemos extrañarnos de que haya que hacer tantas "cosas" para acceder o manipular la información almacenada en esas "secuencias" de datos. Si bien, esa "complicación" nos da mayor control sobre el formato de la información contenida en los *streams*. Por suerte, para los que nos gustan las cosas "simples" las clases específicas nos facilitan mucho las cosas.

A continuación veremos un par de ejemplos en los que manipularemos cierta información tanto en la memoria usando un objeto del tipo *MemoryStream*, como en un fichero de disco usando *FileStream* y las clases que casi con seguridad usaremos habitualmente para acceder al contenido de los ficheros: *StreamReader* y *StreamWriter*.

Manejar un fichero usando *FileStream*

En este primer ejemplo veremos lo complicado que puede parecer acceder a un fichero usando la clase *FileStream* y por extensión cualquier método de otras clases que devuelvan este tipo de secuencia, como por ejemplo los métodos *OpenRead*, *OpenWrite*, etc. de la clase *File*.

La "complejidad" de esta clase es que realmente obtiene o guarda la información por medio de un array de tipo *Byte*, cuando a lo que estamos acostumbrados es a usar cadenas.

Nota:

Realmente, esta forma "binaria" de acceder a la información de un fichero no la tenemos que ver como un inconveniente, ya que nos puede servir para acceder de forma "binaria" a ese fichero, en caso de que nuestra intención sea acceder de forma "normal" para, por ejemplo, leer solo texto, deberíamos usar otras clases más especializadas para esa tarea, como lo es *StreamReader*.

En el siguiente código tenemos dos métodos, uno que guarda una cadena en el fichero indicado:

```
Private Sub guardarDatos(ByVal fichero As String, ByVal cadena As String)
    ' Abrimos o creamos el fichero, para escribir en él
```

```

Dim fs As New System.IO.FileStream(fichero, _
    System.IO.FileMode.OpenOrCreate, _
    System.IO.FileAccess.Write)
' Escribimos algunas cadenas,
' el problema es que solo podemos escribir arrays de bytes,
' por tanto debemos convertir la cadena en un array de bytes
Dim datos() As Byte
' pero usando la codificación que creamos conveniente
' de forma predeterminada es UTF-8,
' aunque la codificación de Windows es ANSI (Encoding.Default)
Dim enc As New System.Text.UTF8Encoding
' convertimos la cadena en un array de bytes
datos = enc.GetBytes(cadena)
' lo escribimos en el stream
fs.Write(datos, 0, datos.Length)
' nos aseguramos que se escriben todos los datos
fs.Flush()
' cerramos el stream
fs.Close()
End Sub

```

- En el constructor de la clase *FileStream* indicamos el fichero en el que queremos guardar la información, también le indicamos que queremos crearlo o abrirlo, es decir, si ya existe lo abre y si no existe lo crea, de cualquiera de las formas, en el siguiente parámetro del constructor le indicamos que nuestra intención es escribir en ese fichero.
- Como hemos comentado, la clase *FileStream* (y en general todos los *streams*) trabaja con bytes, por tanto para poder almacenar algo en ese fichero debemos hacerlo mediante un array de tipo *Byte*.

En el caso de las cadenas, éstas siempre deben estar codificadas, es decir, deben usar el juego de caracteres que creamos conveniente, en el mundo de .NET ese juego de caracteres es Unicode, más concretamente usando la codificación UTF-8, la cual permite trabajar con cualquier carácter de cualquier cultura.

Como lo que nos interesa es convertir una cadena en un array de bytes, usamos el método *GetBytes* de un objeto *UTF8Encoding*, el cual convierte la cadena en una "ristra" de bytes con el formato adecuado, en este caso UTF-8.

Si en lugar de usar UTF-8 quisiéramos usar otro "codificador", por ejemplo el predeterminado de Windows, con idea de que los ficheros sean compatibles con otras aplicaciones que utilizan el formato predeterminado de Windows, tendremos que declarar la variable **enc** de la siguiente forma:

```
Dim enc As System.Text.Encoding = System.Text.Encoding.Default
```

- A continuación, simplemente le pasamos el array de bytes al método *Write* del *FileStream* indicando desde qué posición de dicho array debe escribir y cuantos bytes.
- Por último nos aseguramos de que todos los bytes del "buffer" se guarden en el fichero y lo cerramos.

Y otra función que devuelve el contenido de un fichero en formato cadena:

```

Private Function leerDatos(ByVal fichero As String) As String
    ' Los bloques leídos los almacenaremos en un StringBuilder
    Dim res As New System.Text.StringBuilder
    ' Abrimos el fichero para leer de él
    Dim fs As New System.IO.FileStream(fichero, _
        System.IO.FileMode.Open, _
        System.IO.FileAccess.Read)
    ' los datos se leerán en bloques de 1024 bytes (1 KB)
    Dim datos(1024) As Byte
    Dim enc As New System.Text.UTF8Encoding()
    ' leemos mientras hay algo en el fichero
    While fs.Read(datos, 0, 1024) > 0

```

```

    ' agregamos al stringBuilder los bytes leídos
    ' (convertidos en una cadena)
    res.Append(enc.GetString(datos))
End While
' cerramos el buffer
fs.Close()
' devolvemos todo lo leído
Return res.ToString
End Function

```

- En esta función vamos a leer los datos del fichero indicado, como ya hemos visto, la clase *FileStream* trabaja con bytes y esos bytes los convertimos a caracteres por medio de las clases de codificación especializadas.
Por regla general, esas lecturas las haremos de forma parcial, es decir leyendo bloques de bytes y como tenemos que convertir esos bytes en caracteres, y puede ser que el fichero sea muy grande, en lugar de concatenar una cadena para almacenar las lecturas parciales, vamos a usar un objeto del tipo *StringBuilder* en el que iremos "agregando" cada trozo leído, de forma que el rendimiento no se vea penalizado por la forma de ser de las cadenas, ya que cada vez que hacemos una concatenación en una variable de tipo *String*, realmente estamos creando nuevos objetos en la memoria y si son muchos, pues la verdad es que tendremos al recolector de basura (GC) trabajando a tope, y si usamos un objeto *StringBuilder* el rendimiento mejora una barbaridad.
- La lectura de cada bloque de bytes lo hacemos en un bucle *While*, también podríamos haberlo hecho en un bucle *Do While*, pero en estos casos, el rendimiento de *While* es un "poquitín" mayor.
- Los datos leídos los agregamos al objeto *StringBuilder* por medio del método *Append* que se encarga de agregarlo a la cadena interna.
- Finalmente cerramos el *stream* y devolvemos la cadena leída.

Para usar estas dos funciones lo podemos hacer de la siguiente forma:

```

' un fichero de ejemplo (el directorio debe existir)
Const fichero As String = "E:\Pruebas\prueba.txt"
' guardamos una cadena en el fichero
guardarDatos(fichero, "Hola, Mundo de FileStream")
'
' Leemos el contenido del fichero y lo mostramos
Console.WriteLine(leerDatos(fichero))

```

Este código no necesita mayor explicación.

Manejar un fichero usando StreamReader y StreamWriter

A continuación veremos cómo crear las dos funciones del ejemplo anterior para que utilicen las clases "especializadas" para leer y escribir cadenas en un fichero.

Como podremos comprobar, esta es una forma muchísimo más simple y, por tanto recomendada para este tipo de acceso a los ficheros. Aunque debemos recordar que solo servirá para leer la información de forma secuencial y en formato cadena.

Nota:

El código para usar estas dos funciones será el mismo que el usado para las funciones que utilizan la clase *FileStream*.

La función para guardar una cadena en un fichero:

```

Private Sub guardarDatos(ByVal fichero As String, ByVal cadena As String)
    ' Abrimos el fichero para escribir, (no añadir),
    ' usando la codificación predeterminada: UTF-8
    Dim sw As New System.IO.StreamWriter(fichero, False)
    ' guardamos toda la cadena
    sw.WriteLine(cadena)

```

```

' Cerramos el fichero
sw.Close()
End Sub

```

Como podemos apreciar, esta es una forma mucho más "compacta" que la anterior, ya que solo tenemos que indicar en el constructor lo que queremos hacer y usar el método *Write* o *WriteLine* para guardar lo que queramos en el fichero.

Para guardarlo usando la codificación predeterminada del Sistema Operativo en el que se utilice la aplicación, (en Windows será ANSI), simplemente usamos este constructor:

```
Dim sw As New System.IO.StreamWriter(fichero, False, System.Text.Encoding.Default)
```

Para leer los datos podemos hacerlo de dos formas: línea a línea o todo el contenido de una sola vez.

En el código siguiente se muestra línea a línea, y al final, (comentado), cómo hacerlo en una sola pasada.

```

Private Function leerDatos(ByVal fichero As String) As String
    ' Abrimos el fichero para leer
    ' usando la codificación UTF-8 (la predeterminada de .NET)
    Dim sr As New System.IO.StreamReader(fichero, True)
    ' si queremos usar la predeterminada de Windows
    'Dim sr As New System.IO.StreamReader(fichero, System.Text.Encoding.Default)

    ' Podemos leer cada una de las líneas del fichero o todo el contenido
    '     Forma larga:
    ' si vamos a leer el fichero línea por línea, mejor usar un StringBuilder
    Dim ret As New System.Text.StringBuilder
    ' recorremos el fichero hasta que no haya nada que leer
    While sr.Peek > -1
        ret.Append(sr.ReadLine)
    End While
    ' cerramos el fichero
    sr.Close()
    ' devolvemos lo leído
    Return ret.ToString
    '

    ''     Forma corta:
    '' leemos todo el contenido del fichero
    'Dim ret As String = sr.ReadToEnd()
    '' lo cerramos
    'sr.Close()
    '' devolvemos lo leído
    'Return ret
End Function

```

Si nos decidimos a leer el contenido del fichero línea a línea, podemos usar el método *Peek*, el cual devolverá el siguiente carácter del buffer del *stream*, o -1 si no hay nada que leer. *Peek* no "consume" el carácter, simplemente comprueba si hay algo que leer.

Si hay algo que leer, leemos la línea completa y la añadimos al objeto *StringBuilder*, el bucle se repetirá mientras haya información pendiente de leer.

Pero si optamos por la vía rápida, porque realmente no nos interese procesar cada línea, podemos usar el método *ReadToEnd*, que en nuestro ejemplo, el valor devuelto será todo el contenido del fichero, el cual asignamos a una variable de tipo *String* para usarla como valor devuelto por la función, después de cerrar el fichero.

Asegurarnos que el fichero se cierra

Si queremos ahorrarnos el paso intermedio de asignar el valor en una variable y después devolverlo, también podemos hacerlo de esta forma:

```
Try
```

```

        Return sr.ReadToEnd()
    Finally
        sr.Close()
    End Try

```

Ya que el bloque **Finally** siempre se ejecutará, se produzca o no un error, por tanto nos aseguramos de que el fichero se cierra.

Liberar recursos: Using... End Using

O si lo preferimos, podemos usar la nueva forma de asegurarnos de que los recursos usados se liberan:

```

' Podemos usar Using para asegurarnos de que el recurso se libera
Private Function leerDatos(ByVal fichero As String) As String
    Dim sr As New System.IO.StreamReader(fichero, True)
    Using sr
        Return sr.ReadToEnd()
    End Using
End Function

```

En este código, cuando usamos **Using sr**, al ejecutarse **End Using**, el CLR se encargará de llamar al método *Dispose* de la clase, de forma que se liberen los recursos que estemos usando, en este ejemplo: el fichero abierto.

Estos dos últimos ejemplos serían equivalentes a los anteriores, más seguros, pero también con más "trabajo" para el CLR.

Nota:

Using... End Using solo se puede usar con clases que implementen la interfaz **IDisposable**, que es la que asegura que el objeto implementa el método *Dispose*.

Por tanto, si implementamos el método **IDisposable.Dispose** en nuestras clases, en ese método nos tenemos que asegurar que liberamos los recursos que nuestra clase esté utilizando.

Ejemplo de para cifrar y descifrar un fichero

En el siguiente ejemplo (adaptado de uno de la documentación), veremos cómo usar algunas de las clases basadas en *Stream*, particularmente las clase *MemoryStream*, *FileStream*, *CryptoStream* además de las clases *StreamReader* y *StreamWriter*.

Este código tiene dos funciones:

La primera encripta (cifra) una cadena y la guarda en un fichero.

La segunda desencripta (descifra) el contenido de un fichero y lo guarda en otro.

Ambas funciones devuelven la cadena cifrada o descifrada respectivamente.

```

Imports System
Imports System.IO
Imports System.Text
Imports System.Security.Cryptography

Module Module1
    Const fic1 As String = "E:\Pruebas\Prueba CryptoStream.txt"
    Const fic3 As String = "E:\Pruebas\Prueba CryptoStream des.txt"
    Const sKey As String = "El password a usar"
    '

    Sub Main()
        Dim ret As String
        '

        ret = cifrarFichero("Hola, Mundo encriptado", fic1)
        Console.WriteLine("Cadena encriptada : {0}", ret)
    End Sub
End Module

```

```

    '
    ret = descifrarFichero(fic1, fic3)
    Console.WriteLine("Cadena desencriptada: {0}", ret)
    '
    Console.ReadLine()
End Sub

Function cifrarFichero( _
    ByVal texto As String, _
    ByVal ficSalida As String) As String
    ' Creamos un MemoryStream con el texto a cifrar
    Dim enc As New UTF8Encoding
    Dim datos() As Byte = enc.GetBytes(texto)
    Dim ms As New MemoryStream(datos)
    ' El fichero de salida
    Dim fs As New FileStream(ficSalida, FileMode.Create, FileAccess.Write)
    ' El proveedor criptográfico
    Dim r As New DESCryptoServiceProvider
    '
    ' Establecer la clave secreta
    r.Key = Encoding.Default.GetBytes(sKey.Substring(0, 8))
    r.IV = Encoding.Default.GetBytes(sKey.Substring(0, 8))
    '
    ' Crear una secuencia de cifrado
    Dim cs As New CryptoStream(fs, _
        r.CreateEncryptor(), _
        CryptoStreamMode.Write)
    '
    ' Escribir el fichero cifrado
    cs.Write(datos, 0, datos.Length)
    cs.Close()
    '
    ' devolver el texto cifrado
    Return Convert.ToBase64String(ms.ToArray())
End Function

Function descifrarFichero( _
    ByVal fichero As String, _
    ByVal ficSalida As String) As String
    ' el proveedor del cifrado y las claves usadas para cifrar
    Dim r As New DESCryptoServiceProvider
    r.Key() = Encoding.Default.GetBytes(sKey.Substring(0, 8))
    r.IV = Encoding.Default.GetBytes(sKey.Substring(0, 8))
    '
    ' crear la secuencia para leer el fichero cifrado
    Dim fs As New FileStream(fichero, FileMode.Open, FileAccess.Read)
    '
    Dim cs As New CryptoStream(fs, _
        r.CreateDecryptor(), _
        CryptoStreamMode.Read)
    '
    ' guardar el contenido de fichero descifrado
    Dim sw As New StreamWriter(ficSalida)
    Dim sr As New StreamReader(cs)
    sw.Write(sr.ReadToEnd())
    sw.Flush()
    sw.Close()
    '
    ' devolver el texto
    sr = New StreamReader(fic3)
    Dim ret As String = sr.ReadToEnd()
    sr.Close()
    '
    Return ret
End Function
End Module

```

Lección 3: Acceso al sistema de archivos

- Las clases del espacio de nombres System.IO
- Clases para manipular unidades, directorios y ficheros
- Las clases para leer o escribir los streams
- El objeto My

Introducción

En la lección anterior vimos cómo podemos acceder al contenido de los ficheros, es decir, cómo leer lo que hay dentro de un fichero, e incluso cómo haremos para escribir en ellos. En esta lección trataremos del acceso al sistema de archivos, es decir, veremos las clases que .NET pone a nuestra disposición para que podamos manejar tanto los ficheros (archivos) como los directorios.

Los que hayan utilizado el objeto *File System Objects* desde Visual Basic 6.0 seguramente encontrarán muchas similitudes con lo que .NET ofrece, pero, como comprobaremos, en .NET tenemos más variedad de clases y, por supuesto, podemos hacer muchas más cosas, y de forma más fácil, que desde VB6.

Acceso al sistema de archivos

- [Las clases del espacio de nombres System.IO](#)
 - Clases para manipular unidades, directorios y ficheros
 - Las clases para crear streams
 - Las clases para leer o escribir en los streams
- [Clases para manipular unidades, directorios y ficheros](#)
 - Las clases Directory y DirectoryInfo
 - Los métodos de las clases Directory y DirectoryInfo
 - Las clases File y FileInfo
 - Cifrar y descifrar un fichero usando File o FileInfo
 - Abrir ficheros para agregar o leer el contenido
 - Manipular cadenas relacionadas con ficheros y directorios usando Path
- [Las clases para leer o escribir en los streams](#)
 - Las clases StreamReader y StreamWriter
 - La codificación de los ficheros de .NET
- [El objeto My](#)
 - El objeto My.Computer.FileSystem
 - Buscar texto en ficheros con FindInFiles

- Examinar el contenido de un fichero con formato con OpenTextFieldParse
-

Lección 3: Acceso al sistema de archivos

Las clases del espacio de nombres System.IO

- Clases para manipular unidades, directorios y ficheros
- Las clases para leer o escribir los streams
- El objeto My

Acceso al sistema de archivos

Empezaremos viendo las las clases elementales o más usadas para manejar el sistema de archivos desde .NET.

El espacio de nombres *System.IO* es el que contiene todas las clases relacionadas con los ficheros y directorios, en ese mismo espacio de nombres están las que vimos en la lección anterior dedicada a los *streams* de .NET, ya que al fin y al cabo, cuando tratamos con los ficheros, estaremos tratando con una secuencia de caracteres o bytes, secuencia a la que accederemos con las clases especializadas que ya vimos.

Aquí nos centraremos principalmente en la forma de acceder a esos ficheros, a copiarlos, a comprobar si existen, etc., en definitiva: a manipularlos.

Las clases del espacio de nombres System.IO

Entre las clases que nos podemos encontrar en este espacio de nombres, tenemos clases que podemos agrupar dependiendo de las tareas que podemos hacer con ellas en:

- Las clases para manipular las unidades, directorios y ficheros
- Las clases para crear *streams*
- Las clases para leer o escribir en los *streams*

Veamos primero una relación de esas clases y enumeraciones y después veremos algunas de ellas con más detalle.

Clases para manipular unidades, directorios y ficheros

Entre las clases que nos permiten trabajar con los ficheros, directorios y las unidades de disco, podemos destacar las siguientes:

- *Directory*, proporciona métodos estáticos para crear, mover y enumerar los ficheros de directorios y subdirectorios.
- *DirectoryInfo*, al igual que la clase *Directory*, pero los métodos son de instancia, dicha instancia se creará a partir de un directorio determinado.
- *DriveInfo*, proporciona métodos de instancia para crear, mover y enumerar el

contenido de unidades.

- *File*, proporciona métodos estáticos para crear, copiar, eliminar, mover y abrir ficheros, además de ayudar a la creación de objetos *FileStream*.
- *FileInfo*, igual que la clase *File*, pero los métodos son de instancia, dicha instancia se creará a partir de un fichero determinado.
- *Path*, proporciona métodos y propiedades para procesar cadenas relacionadas con los directorios.

También tenemos las siguientes enumeraciones:

- *FileAccess*, define constantes para el tipo de acceso a los ficheros: lectura, escritura o lectura/escritura.
- *FileAttributes*, define constantes para el atributo de los ficheros y directorios: archivo, oculto, solo lectura, etc.
- *FileMode*, define las constantes que podemos usar para controlar como abrimos un fichero.
- *FileShare*, define las constantes para controlar el tipo de acceso que otros objetos *FileStream* pueden tener al mismo fichero.

Las clases para crear streams

La siguiente relación son clases que nos permiten la creación de streams, las cuales ya vimos en la lección anterior, por tanto solo las mencionaremos.

- *BufferedStream*, clase abstracta que representa un buffer de almacenamiento para operaciones de lectura y escritura de otro stream.
- *DeflateStream*, permite la compresión y descompresión de streams usando el algoritmo *Deflat*.
- *GZipStream*, usada para comprimir y descomprimir streams.
- *FileStream*, nos permite una forma básica de acceder y manipular ficheros.
- *MemoryStream*, crear un stream que se almacena en la memoria como una secuencia de bytes.
- *NetworkStream*, proporciona una secuencia de datos para el acceso a la red.
- *CryptoStream*, un stream usado para encriptar otros streams.

Las clases para leer o escribir en los streams

- *BinaryReader*, lee tipos primitivos o cadenas codificadas desde un *FileStream*.
 - *BinaryWriter*, escribe tipos primitivos o cadenas codificadas en un *FileStream*.
 - *StreamReader*, lee caracteres desde un *FileStream*, usando codificación para convertir los caracteres a/desde bytes.
 - *StreamWriter*, escribe caracteres a un *FileStream*, usando codificación para convertir los caracteres en bytes.
 - *StringReader*, lee caracteres desde un *String*. La salida puede ser a un stream en cualquier codificación o a una cadena.
 - *StringWriter*, escribe caracteres a un *String*. Al igual que *StringReader*, la salida puede ser a un stream usando cualquier codificación o a una cadena.
-
- *TextReader*, es la clase abstracta base para *StreamReader* y *StringReader*.
 - *TextWriter*, es la clase abstracta base para *StreamWriter* y *StringWriter*.

Nota:

La clase abstracta Stream está diseñada para entrada y salida de bytes, las clases abstractas *TextReader* y *TextWriter* están diseñadas para la entrada/salida de caracteres Unicode.

Lección 3: Acceso al sistema de archivos

- Las clases del espacio de nombres System.IO
- Clases para manipular unidades, directorios y ficheros
- Las clases para leer o escribir los streams
- El objeto My

Clases para manipular unidades, directorios y ficheros

Entre las clases que nos permiten trabajar con los ficheros, directorios y las unidades de disco, podemos destacar las siguientes:

- *Directory*, proporciona métodos estáticos para crear, mover y enumerar los ficheros de directorios y subdirectorios.
- *DirectoryInfo*, al igual que la clase *Directory*, pero los métodos son de instancia, dicha instancia se creará a partir de un directorio determinado.
- *DriveInfo*, proporciona métodos de instancia para crear, mover y enumerar el contenido de unidades.
- *File*, proporciona métodos estáticos para crear, copiar, eliminar, mover y abrir ficheros, además de ayudar a la creación de objetos *FileStream*.
- *FileInfo*, igual que la clase *File*, pero los métodos son de instancia, dicha instancia se creará a partir de un fichero determinado.
- *Path*, proporciona métodos y propiedades para procesar cadenas relacionadas con los directorios.

También tenemos las siguientes enumeraciones:

- *FileAccess*, define constantes para el tipo de acceso a los ficheros: lectura, escritura o lectura/escritura.
- *FileAttributes*, define constantes para el atributo de los ficheros y directorios: archivo, oculto, solo lectura, etc.
- *FileMode*, define las constantes que podemos usar para controlar como abrimos un fichero.
- *FileShare*, define las constantes para controlar el tipo de acceso que otros objetos *FileStream* pueden tener al mismo fichero.

Las clases Directory y DirectoryInfo

Cuando necesitemos acceder a un directorio, por ejemplo, para saber que subdirectorios y ficheros contiene o para obtener otro tipo de información, incluso para saber si existe o no, en esos casos podemos utilizar las clases *Directory* o *DirectoryInfo*.

La primera de ellas: *Directory*, proporciona métodos estáticos, es decir, los métodos que contienen siempre estarán disponibles sin necesidad de crear una nueva instancia de la clase. Por tanto, podremos usar esos métodos simplemente usando la propia clase.

La segunda: *DirectoryInfo*, proporciona los mismos métodos que *Directory*, pero en lugar de ser métodos estáticos, son métodos de instancia, es decir, esos métodos solamente se podrán usar después de haber creado una instancia (u objeto en memoria) de esta clase. Cada una de las instancias de *DirectoryInfo*, hará referencia a un directorio en particular, ese directorio se indicará al crear el objeto, aunque también puede ser que esté referenciado al obtenerlo mediante otros métodos que devuelven este tipo de objetos.

Nota:

El directorio asignado al crear la instancia de *DirectoryInfo* no tiene porqué existir.

De esa forma podemos comprobar si existe, y en caso de que no exista, crearlo, etc.

Por ejemplo, si queremos saber si un directorio existe, utilizaremos el método *Exists*, en el caso de la clase *Directory*, como parámetro le indicaremos el directorio del que queremos comprobar su existencia, sin embargo, si usamos un objeto del tipo *DirectoryInfo*, no necesitamos indicar ningún parámetro, ya que el directorio que está asociado con esa clase lo habremos indicado al crear el objeto.

En el siguiente código vemos cómo usar estas dos clases:

```
Const dir1 As String = "E:\Pruebas"
Const dir2 As String = "E:\Pruebas2"

Sub Main()
    claseDirectory()
    Console.WriteLine()
    claseDirectoryInfo()
    Console.WriteLine()
    '
    Console.ReadLine()
End Sub

Sub claseDirectory()
    ' Los métodos de la clase Directory son estáticos,
    ' por tanto no necesitamos crear una instancia de esta clase
    '
    ' Comprobar si existe un directorio:
    If System.IO.Directory.Exists(dir1) Then
        Console.WriteLine("El directorio '{0}' SI existe", dir1)
    Else
        Console.WriteLine("El directorio '{0}' NO existe", dir1)
    End If
End Sub

Sub claseDirectoryInfo()
    ' Los métodos de la clase DirectoryInfo son de instancia,
    ' por tanto tenemos que crear una instancia de la clase
    ' para poder usarla
    '
    Dim di As New System.IO.DirectoryInfo(dir2)
    '
    ' Comprobar si existe un directorio:
    If di.Exists Then
        Console.WriteLine("El directorio '{0}' SI existe", dir2)
    Else
        Console.WriteLine("El directorio '{0}' NO existe", dir2)
    End If
End Sub
```

El resto de métodos de estas dos clases funcionan de forma similar, al menos en el sentido de que si usamos la clase *Directory*, siempre habrá un parámetro que haga referencia al directorio que queremos usar, mientras que con *DirectoryInfo* siempre estará haciendo referencia al directorio usado al crear la instancia.

Nota:

No queremos parecer repetitivos, pero **nos interesa que quede claro** como funcionan este tipo de clases, que por otro lado, será similar al de las clases File y FileInfo, al menos en el sentido de que la clase con los métodos estáticos siempre necesitará saber a qué elemento del sistema de archivos estamos refiriéndonos, mientras que las clases que debemos instanciar, siempre sabrán con qué elemento trabajarán.

Los métodos de las clases Directory y DirectoryInfo

Estas clases siempre manejarán directorios y entre los métodos que podemos utilizar, destacamos los siguientes, empezamos por la clase *Directory*:

- *CreateDirectory*, crear los directorios indicados en el parámetro. Si algunos de los directorios intermedios no existen, los creará.
- *Exists*, comprueba si el directorio indicado existe.
- *GetCreationTime*, devuelve la fecha y hora de creación del directorio indicado.
- *GetCreationTimeUtc*, devuelve la fecha y hora universal (UTC/GMT) de creación del directorio.
- *GetCurrentDirectory*, devuelve el directorio de trabajo de la aplicación.
- *GetDirectories*, devuelve un array de *String* con todos los subdirectorios del directorio indicado. Se puede indicar un "pattern" de búsqueda y si queremos incluir los subdirectorios que tenga el directorio indicado.
- *GetDirectoryRoot*, devuelve el directorio raíz del directorio indicado.
- *GetFiles*, devuelve un array de tipo *String* con los ficheros del directorio indicado. Se puede indicar un filtro de búsqueda.
- *GetLastAccessTime*, devuelve la fecha y hora del último acceso al directorio.
- *GetLastAccessTimeUtc*, ídem que el anterior, pero la fecha y hora en formato UTC.
- *GetLastWriteTime*, devuelve la fecha y hora de la última escritura realizada en el directorio.
- *GetLastWriteTimeUtc*, como el anterior, pero usando fechas UTC.
- *GetLogicalDrives*, devuelve un array con los nombres de las unidades lógicas en el formato: "<letra>:\\".
- *GetParent*, devuelve el directorio de nivel superior del indicado.
- *Move*, mueve un fichero o directorio y su contenido a una nueva localización.
- *SetCreationTime*, asigna la fecha y hora de creación de un fichero o directorio.
- *SetCreationTimeUtc*, como el anterior, pero la fecha/hora es UTC.
- *SetCurrentDirectory*, indica el directorio de trabajo de la aplicación.
- *SetLastAccessTime*, asigna la fecha y hora del último acceso.
- *SetLastAccessTimeUtc*, como el anterior, pero la fecha/hora en formato UTC.
- *SetLastWriteTime*, asigna la fecha y hora de la última escritura.
- *SetLastWriteTimeUtc*, como el anterior, pero usando la fecha y hora UTC.

Como hemos comentado, la clase *DirectoryInfo* siempre hará referencia al directorio utilizado para instanciarla, por tanto algunos de los métodos de la clase *Directory* se convierten en propiedades de esta clase y otros cambian de nombre para adecuarlos mejor a la acción a realizar con ellos.

Por ejemplo, en la clase *Directory* tenemos el método *Exists*, que en *DirectoryInfo* es una propiedad. De igual forma, el método *Move* de la clase *Directory* es el método *MoveTo* de *DirectoryInfo*.

En cuanto a los métodos de *Directory* para obtener o asignar la fecha de creación, acceso, etc., en *DirectoryInfo* son propiedades de lectura/escritura.

Dicho esto, no vamos a relacionar todos estos miembros de *DirectoryInfo*, ya que en la documentación están bien detallados y no nos resultará difícil de saber cómo usarlos.

Lo que si queremos resaltar es que algunos de los métodos de la clase *DirectoryInfo*, concretamente *GetDirectories* y *GetFiles*, no devuelven un array de tipo *String*, sino un array de objetos *DirectoryInfo* o *FileInfo* respectivamente.

También debemos indicar que la clase *DirectoryInfo* tiene ciertas propiedades, como *FullName* o *Name*, que nos dan información del nombre de la ruta completa o del nombre del directorio.

Y como nota final sobre *Directory* y *DirectoryInfo*, aclarar que el "filtro" (o *pattern*) que podemos usar para filtrar los directorios o ficheros obtenidos con *GetDirectories* o *GetFiles* solo pueden tener un filtro o especificación, queremos aclarar este punto, ya que en Visual Basic 6.0, si indicábamos un filtro en los controles File o *Directory*, podíamos indicar varios filtros separados por punto y coma (;), en las clases de .NET esto no se puede hacer.

Para dejar claro estos puntos, veremos un ejemplo de cómo usar *GetDirectories* y *GetFiles*, con estas dos clases.

```
Imports System
Imports System.IO

Module Module1
    Const dir1 As String = "E:\Pruebas"

    Sub Main()
        Console.WriteLine("Ejemplo de la clase Directory:")
        claseDirectory()
        Console.WriteLine()
        Console.WriteLine("Ejemplo de la clase DirectoryInfo:")
        clase DirectoryInfo()
        Console.WriteLine()
        Console.ReadLine()
    End Sub

    Sub claseDirectory()
        ' Recorrer los subdirectorios de un directorio
        Dim dirs() As String
        dirs = Directory.GetDirectories(dir1)
        mostrarFicheros(dir1, "*.*")
        ' recorrer los ficheros de cada uno de estos directorios
        For Each dir As String In dirs
            mostrarFicheros(dir, "*.*")
        Next
    End Sub

    Sub mostrarFicheros(ByVal dir As String, ByVal filtro As String)
        Console.WriteLine("Los ficheros {0} del directorio: {1}", _
            filtro, dir)
        Dim fics() As String
        fics = Directory.GetFiles(dir, filtro)
        For Each fic As String In fics
            Console.WriteLine(" Fichero: {0}", fic)
        Next
    End Sub

    Sub clase DirectoryInfo()
        Dim di As New DirectoryInfo(dir1)
        Dim dirs() As DirectoryInfo
        dirs = di.GetDirectories()
        mostrarFicheros(di, "*.*")
        For Each dir As DirectoryInfo In dirs
            mostrarFicheros(dir, "*.*")
        Next
    End Sub

    Sub mostrarFicheros(ByVal dir As DirectoryInfo, ByVal filtro As String)
        Console.WriteLine("Los ficheros {0} del directorio: {1}", _
            filtro, dir.FullName)
        Dim fics() As FileInfo
        fics = dir.GetFiles(filtro)
        For Each fic As FileInfo In fics
            Console.WriteLine(" Fichero: {0}", fic.Name)
        Next
    End Sub
End Module
```

Las clases File y FileInfo

Al igual que ocurre con las clases para manejar los directorios, tenemos dos clases diferentes para manejar los ficheros, una de ellas (*File*) todos los métodos que tiene son estáticos, por tanto podemos usarlos directamente, sin crear una nueva instancia de la clase. Por otro lado la clase *FileInfo* es una clase de la que tenemos que crear un nuevo objeto para poder usarla, al crear ese objeto (o instancia), tenemos que indicar el fichero al que hará referencia, ese fichero no tiene porqué existir previamente.

En el siguiente código podemos ver cómo usar estas dos clases.

```
Imports System
Imports System.IO

Module Module1
    Const fic1 As String = "E:\Pruebas\Prueba.txt"
    Const fic2 As String = "E:\Pruebas\SubDir\Prueba3.txt"

    Sub Main()
        Console.WriteLine("Ejemplo de la clase File:")
        claseFile()
        Console.WriteLine()
        Console.WriteLine("Ejemplo de la clase FileInfo:")
        claseFileInfo()
        Console.WriteLine()
        Console.ReadLine()
    End Sub

    Sub claseFile()
        ' comprobar si el fichero existe
        If File.Exists(fic1) Then
            Console.WriteLine("El fichero '{0}' SI existe", fic1)
            Console.WriteLine("Fecha creación: {0}", File.GetCreationTime(fic1))
        Else
            Console.WriteLine("El fichero '{0}' NO existe", fic1)
        End If
    End Sub

    Sub claseFileInfo()
        Dim fi As New FileInfo(fic2)
        '
        ' Comprobar si existe el fichero:
        If fi.Exists Then
            Console.WriteLine("El fichero '{0}' SI existe", fic2)
            Console.WriteLine("Fecha creación: {0}", fi.CreationTime)
            agregarTexto(fi)
        Else
            Console.WriteLine("El fichero '{0}' NO existe", fic2)
            ' lo creamos
            fi.Create()
        End If
    End Sub

    Sub agregarTexto(ByVal fi As FileInfo)
        Dim fs As StreamWriter
        fs = fi.CreateText
        fs.WriteLine("Hola, Mundo")
        fs.Flush()
        fs.Close()
        '
        fi.Refresh()
        Console.WriteLine("Tamaño : {0}", fi.Length)
    End Sub
End Module
```

Cifrar y descifrar un fichero usando File o FileInfo

En los sistemas operativos Windows XP y Windows 2003, al mostrar el cuadro de diálogo de las propiedades avanzadas de un fichero, nos permite cifrarlo, de forma que el usuario actual sólo tenga acceso al contenido del mismo.



Figura 4.2. Windows XP/2003 nos permite cifrar un fichero

Tanto la clase *File* como *FileInfo* tienen métodos para realizar esta operación, así como la inversa: descifrarlo. Cuando ciframos un fichero, solo el usuario actual podrá acceder a su contenido (y el resto de administradores), la forma de hacerlo es bien simple: solo tenemos que llamar al método *Encrypt* o *Decrypt* para cifrarlo o descifrarlo. Cuando está cifrado, el nombre del fichero se mostrará en un color diferente al del resto de ficheros (de forma predeterminada en verde).

Esta forma de cifrado es diferente a la que vimos en la lección anterior, ya que si ciframos el contenido de un fichero de forma "manual", siempre estará cifrado, independientemente de quién acceda al fichero.

Cuando utilizamos los métodos para cifrar o descifrar un fichero, no recibiremos ninguna excepción si aplicamos el método de cifrar sobre un fichero ya cifrado. Pero si queremos saber si un determinado fichero está o no cifrado, lo podemos averiguar mediante la propiedad *Attribute* de la clase *FileInfo*, particularmente comprobando si tiene activado el atributo *FileAttributes.Encrypted*.

En el siguiente código tenemos ejemplos de cómo usar las clases *File* y *FileInfo* para comprobar si un fichero ya está cifrado, en cuyo caso lo desciframos, y si resulta que no está cifrado, lo ciframos.

```
Imports System
Imports System.IO

Module Module1
    Const fic1 As String = "E:\Pruebas\Prueba.txt"

    Sub Main()
        cifradoFile(fic1)
        Console.WriteLine()
        cifradoFileInfo(fic2)
        Console.WriteLine()
        '
        Console.ReadLine()
    End Sub

    ' Cifrar / Descifrar ficheros
    ' y comprobar si ya están cifrados...
    Sub cifradoFile(ByVal fichero As String)
        ' comprobar si está cifrado,
        Dim atrib As FileAttributes
        atrib = File.GetAttributes(fichero)
        If (atrib And FileAttributes.Encrypted) = FileAttributes.Encrypted Then
```

```

        Console.WriteLine("El fichero {0} ya estaba cifrado.", fichero)
        File.Decrypt(fichero)
    Else
        File.Encrypt(fichero)
        Console.WriteLine("El fichero {0} no estaba cifrado.", fichero)
    End If
End Sub

Sub cifradoFileInfo(ByVal fichero As String)
    ' comprobar si está cifrado,
    Dim fi As New FileInfo(fichero)
    Dim atrib As FileAttributes
    atrib = fi.Attributes
    If (atrib And FileAttributes.Encrypted) = FileAttributes.Encrypted Then
        Console.WriteLine("El fichero {0} ya estaba cifrado.", fi.FullName)
        fi.Decrypt()
    Else
        fi.Encrypt()
        Console.WriteLine("El fichero {0} no estaba cifrado.", fi.FullName)
    End If
End Sub
End Module

```

Abrir ficheros para agregar o leer el contenido

Estas dos clases de manipulación de ficheros exponen una serie de métodos que nos permiten abrir el fichero al que hace referencia la clase para leer el contenido del mismo o para escribir en él.

Dependiendo del método usado, éste devolverá un objeto de tipo *FileStream* o *StreamReader* o *StreamWriter*. Esto nos permite de una forma fácil poder acceder al contenido que tiene sin necesidad de usar otros constructores. Lo que si debemos tener en cuenta es que en el caso de que leamos o escribamos texto, éste se guardará usando la codificación UTF-8, por tanto si queremos usar otro tipo de codificación, tendremos que abrirlos usando los constructores correspondientes, normalmente los de las clases *StreamReader* y *StreamWriter*. De estas dos clases nos ocuparemos en el siguiente capítulo.

Veamos un par de ejemplos que nos permitan leer el contenido o agregar nuevo texto a los ficheros.

```

Imports System
Imports System.IO

Module Module1
    Const fic3 As String = "E:\Pruebas\Prueba4.txt"
    Const fic4 As String = "E:\Pruebas\SubDir\Prueba4.txt"

    Sub Main()
        abrirFile(fic3)
        Console.WriteLine()
        abrirFileInfo(fic4)
        Console.WriteLine()
        '
        Console.ReadLine()
    End Sub

    ' Abrir ficheros para leer el contenido o escribir en él
    Sub abrirFile(ByVal fichero As String)
        If File.Exists(fichero) Then
            Dim fs As FileStream = File.OpenRead(fichero)
            Dim enc As New System.Text.UTF8Encoding()
            Dim datos(1024) As Byte
            fs.Read(datos, 0, datos.Length)
            fs.Close()
            Dim s As String = enc.GetString(datos, 0, datos.Length)
            Console.WriteLine("El contenido de: {0}{2} es:{2}{1}", _
                fichero, s, vbCrLf)
        Else
            Console.WriteLine("Guardando algo en el fichero {0}", fichero)
            Dim fs As FileStream = File.OpenWrite(fichero)

```

```

        Dim enc As New System.Text.UTF8Encoding()
        Dim datos() As Byte = enc.GetBytes("¡Hola bytes!")
        fs.Write(datos, 0, datos.Length)
        fs.Close()
    End If
End Sub

Sub abrirFileInfo(ByVal fichero As String)
    Dim fi As New FileInfo(fichero)
    If fi.Exists Then
        ' abrir el fichero como texto y leemos el contenido
        Dim sr As StreamReader = fi.OpenText
        Dim s As String = sr.ReadToEnd
        Console.WriteLine("El contenido de: {0}{2} es:{2}{1}", _
                          fichero, s, vbCrLf)
    Else
        Console.WriteLine("Guardando algo en el fichero {0}", fichero)
        Dim sw As New StreamWriter(fi.OpenWrite)
        sw.WriteLine("Hello bytes!")
        sw.Close()
    End If
End Sub
End Module

```

En el método **abrirFile** utilizamos la clase *File* para abrir, leer y escribir en un fichero. En este caso estamos usando un objeto del tipo *FileStream* para acceder al contenido de dicho fichero, como sabemos, esta clase solo trabaja con bytes, por tanto si queremos leer el contenido del mismo, debemos hacerlo mediante un array de bytes y para mostrar en formato texto ese contenido, tenemos que aplicarle la codificación correspondiente para extraer el contenido en forma de una cadena. Lo mismo hacemos para guardar una cadena en el fichero, (en caso de que no exista previamente).

En el método **abrirFileInfo**, en lugar de crear un objeto *FileStream* para acceder al contenido, usamos uno del tipo *StreamReader*, ya que esta clase si que nos devolverá una cadena sin necesidad de hacer conversiones extras. Lo mismo ocurre a la hora de escribir algo en el fichero, aunque en esta ocasión utilizamos un objeto del tipo *StreamWriter*.

En el primer caso, cuando leemos el contenido, utilizamos el método *OpenText*, el cual devuelve un objeto del tipo *StreamReader*. Por otra parte, cuando queremos guardar texto, lo que hacemos es pasarle al constructor de la clase *StreamWriter* el objeto *FileStream* devuelto por el método *OpenWrite*.

Nota:

Como hemos podido comprobar, cuando tratamos con las clases *DirectoryInfo* o *FileInfo*, siempre estamos tratando con instancias que pueden ser diferentes y por tanto múltiples, esas instancias siempre manejan el directorio o fichero usado en el constructor, por tanto, será preferible usar estas clases cuando tengamos que hacer varias operaciones sobre el mismo directorio o fichero. Por otro lado, si solo queremos hacer pocas operaciones, podemos usar las clases *Directory* o *File*, ya que no necesitaremos crear un objeto en memoria para hacer las tareas que necesitemos hacer.

Manipular cadenas relacionadas con ficheros y directorios usando Path

La clase *Path* es una clase especial, en la que todos los métodos son estáticos, por tanto para usarla no necesitamos crear una instancia. Lo de "especial" es porque esta clase realmente no realiza ninguna acción "física" sobre los directorios (o ficheros), simplemente nos permite manipular los nombres de esos directorios o ficheros.

Por ejemplo, podemos usar esta clase para obtener información sobre un fichero, como el nombre completo, el nombre del directorio o la extensión.

En el siguiente código vemos cómo usar algunos de esos métodos:

```
Const f1 As String = "E:\Pruebas\Prueba.txt"
```

```
Sub clasePath()
    Console.WriteLine("Nombre completo: {0}", fic1)
    Console.WriteLine("El nombre: {0}", Path.GetFileName(fic1))
    Console.WriteLine("Sin extensión: {0}", Path.GetFileNameWithoutExtension(fic1))
    Console.WriteLine("La extensión: {0}", Path.GetExtension(fic1))
    Console.WriteLine("El directorio: {0}", Path.GetDirectoryName(fic1))
End Sub
```

Nota:

Debido a que la clase `Path` solo maneja cadenas, podemos usar los métodos de esta clase con nombres de ficheros o directorios que no existen, y por tanto, cualquier cambio que hagamos en esos parámetros, no afectarán a ningún fichero ni directorio.

Debido a que los ensamblados de .NET podemos usarlos con diferentes sistemas operativos (al menos en teoría), esta clase proporciona ciertas propiedades que nos permiten averiguar los caracteres especiales de cada sistema, por ejemplo para saber el separador usado por los directorios.

Con esta clase también podemos crear ficheros con nombres temporales, de forma que nos aseguremos de que no habrá otro fichero "temporal" con el mismo nombre. Igualmente, por medio del método `GetTempPath` podemos averiguar el directorio temporal del sistema.

Incluso podemos generar nombres aleatorios "seguros" para ficheros o directorios mediante el método `GetRandomFileName`.

Veamos un ejemplo para generar estos nombres temporales y aleatorios:

```
Dim dir As String
dir = Path.GetTempPath()
Console.WriteLine("Directorio temporal: {0}", dir)
Dim fic As String
fic = Path.GetTempFileName()
Console.WriteLine("Fichero temporal: {0}", fic)
'
fic = Path.GetRandomFileName()
Console.WriteLine("Fichero random: {0}", fic)
```

En los dos primeros casos, se usa el directorio **%TEMP%** del sistema. Y en el caso del fichero temporal, la extensión es **.tmp**

En el nombre aleatorio, tanto el nombre como la extensión están formados por caracteres "imprimibles" aleatorios.

Esta sería una " posible" salida para un usuario llamado **Alguien**:

```
Directorio temporal: C:\Documents and Settings\Alguien\Local Settings\Temp\
Fichero temporal: C:\Documents and Settings\Alguien\Local Settings\Temp\tmp257.tmp
Fichero random: rmbpgxv1.14g
```

Lección 3: Acceso al sistema de archivos

- Las clases del espacio de nombres `System.IO`
 - Clases para manipular unidades, directorios y ficheros
- Las clases para leer o escribir los streams**
- El objeto `My`

Las clases para leer o escribir en los streams

Las clases que podemos usar para leer o escribir en los streams, (y por extensión en los ficheros a los que apuntan esos streams), son:

- `BinaryReader`, lee tipos primitivos o cadenas codificadas desde un `FileStream`.
- `BinaryWriter`, escribe tipos primitivos o cadenas codificadas en un `FileStream`.
- `StreamReader`, lee caracteres desde un `FileStream`, usando codificación para convertir los caracteres a/desde bytes.
- `StreamWriter`, escribe caracteres a un `FileStream`, usando codificación para convertir los caracteres en bytes.
- `StringReader`, lee caracteres desde un `String`. La salida puede ser a un stream en cualquier codificación o a una cadena.
- `StringWriter`, escribe caracteres a un `String`. Al igual que `StringReader`, la salida puede ser a un stream usando cualquier codificación o a una cadena.
- `TextReader`, es la clase abstracta base para `StreamReader` y `StringReader`.
- `TextWriter`, es la clase abstracta base para `StreamWriter` y `StringWriter`.

Las clases `StreamReader` y `StreamWriter`

Aunque ya hemos utilizado algunas de estas clases en los ejemplos que hemos estado viendo, nos vamos a centrar en las dos que con más asiduidad usaremos: `StreamReader` y `StreamWriter`, aunque solo trataremos algunos temas, ya que en líneas generales, ya "deberíamos" saber cómo usarlas.

Estas dos clases las podemos crear directamente, indicando el fichero al que accederemos o bien, podemos usar el valor devuelto por alguna otra clase o método de las clases `File` o `FileInfo`.

La codificación de los ficheros de .NET

En lo que resta de este capítulo nos centraremos en los formatos de codificación usados con estas dos clases, y por regla general con los ficheros de .NET.

La principal ventaja de estas clases es que nos permiten acceder al contenido de los ficheros en modo texto, que es lo que necesitaremos en la mayoría de los casos.

También podremos indicar en el constructor de estas clases la codificación a usar. Como ya hemos comentado en otras ocasiones, por defecto esa codificación es UTF-8, es decir caracteres Unicode de 8 bits. Pero si queremos leer y escribir ficheros que sean compatibles con otras aplicaciones de Windows, por ejemplo, nuestras aplicaciones de Visual Basic 6.0, deberíamos usar la codificación ANSI, que es la predeterminada de los sistemas Windows. De esta forma podremos leer y escribir caracteres "raros" como la letra eñe o las vocales con tilde.

Para indicar la codificación a usar, lo tendremos que indicar como parámetro del constructor, por ejemplo, para abrir un fichero con la codificación predeterminada de Windows, lo haremos de esta forma:

```
Dim sr As New StreamReader(fic1, Encoding.Default)
```

Es decir, usamos la codificación *Default*, que es la predeterminada de Windows.

Nota:

Si no indicamos la codificación que queremos usar, ésta será UTF-8. Pero debemos tener en cuenta que esta codificación solamente leerá correctamente los ficheros que se hayan guardado previamente usando esa misma codificación, por tanto, si queremos leer ficheros creados con VB6, no deberíamos usar UTF-8, (*Encoding.UTF8*), sino *Encoding.Default*.

Algunos tipos de ficheros (según la codificación usada), guardarán al principio del mismo una marca: BOM (*Byte Order Mark*) indicando la codificación usada para escribir en ellos. El propio .NET puede usar dicha marca a la hora de leer en los ficheros, para que así lo haga, podemos indicarlo en el constructor de la clase *StreamReader*:

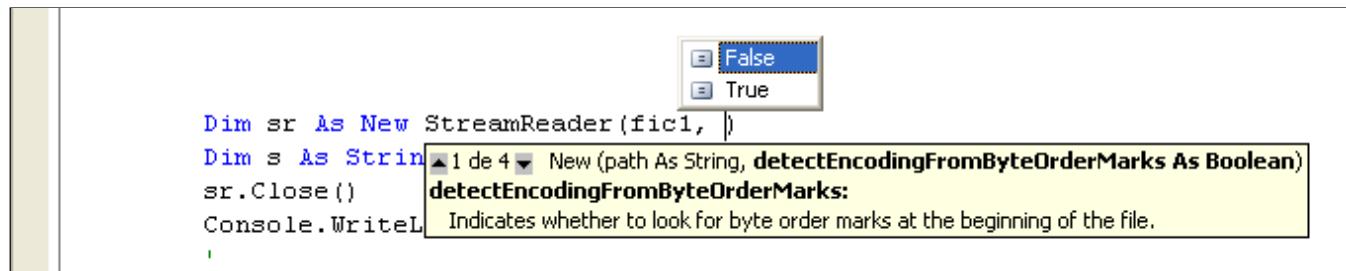


Figura 4.3. Detectar automáticamente la codificación usada

Aunque esta detección automática solo funcionará con los tipos de ficheros que guardan dicha información, que son los tipos: UTF-8, UTF-16 y UTF-32, de estos dos últimos existen dos marcas según se utilice "big-endian" o "little-endian", es decir, si el byte más significativo está al principio o al final respectivamente. Estas marcas, conocidas en .NET como "preámbulos", se pueden averiguar también por medio del método *GetPreamble* de la clase *Encoding*.

En la siguiente tabla podemos ver los valores hexadecimales de las marcas usadas en los ficheros.

Codificación	Número de bytes	Secuencia de bytes (hex)
UTF-8	3	EF BB BF
UTF-16 Little-Endian	2	FF FE
UTF-16 Big-Endian	2	FE FF
UTF-32 Little-Endian	4	00 00 FF FE

Tabla 4.1. Marcas (BOM) de los ficheros UTF

Cuando leemos la información de un fichero abierto por medio de *StreamReader*, podemos hacerlo línea a línea o bien todo el contenido de una sola vez. Como vimos en [la lección de los streams](#), podemos usar el método *ReadLine* o *ReadToEnd* para realizar estas dos operaciones de lectura.

Al usar la clase *StreamWriter*, también podemos indicar la codificación a usar para guardar el texto del fichero:

```
Dim sw As New StreamWriter(fic2, False, Encoding.Default)
```

El segundo parámetro del constructor lo usaremos para indicar si queremos añadir texto al contenido del fichero (*True*) o simplemente queremos sobrescribirlo, (*False*), de forma que cualquier contenido anterior desaparezca.

Como recomendación final, y siguiendo con el tema de la codificación, (que aunque parezca algo "trivial" a muchos usuarios les da algún que otro dolor de cabeza), decir que debemos tener precaución a la hora de elegir la más adecuada.

- Si queremos que los ficheros que manejemos con nuestra aplicación de Visual Basic 2010 sean "compatibles", por ejemplo, con los de VB6, deberíamos usar la codificación *Encoding.Default*.
 - Si los ficheros solo se usarán desde aplicaciones de .NET, podemos usar la codificación predeterminada, es decir: UTF-8.
 - Si vamos a guardar ficheros XML, deberíamos usar siempre codificación UTF-8, que es la predeterminada para ese formato de ficheros.
-

Lección 3: Acceso al sistema de archivos

- Las clases del espacio de nombres System.IO
- Clases para manipular unidades, directorios y ficheros
- Las clases para leer o escribir los streams

El objeto My

El objeto My

Entre los objetivos de este curso no está hablar del objeto *My* y de los objetos que contiene, los cuales nos facilitan un acceso rápido a la mayoría de recursos con los que habitualmente trabajaremos en nuestras aplicaciones.

Pero no nos gustaría dejar un "mal sabor de boca", el cual se quedaría si no habláramos, aunque sea un poco de este objeto.

También es cierto que el objeto *My* (y el resto de objetos que lo compone) será uno de los que con más frecuencia encontraremos documentación y ejemplos, tanto en la Web como en la propia documentación de Visual Basic 2010, por tanto quisiéramos que el lector no deje de leer sobre este objeto, ya que en muchas ocasiones nos ayudará a hacer muchas de las tareas habituales en cualquier aplicación, e incluso los desarrolladores de VB6 encontrarán muchos de los objetos a los que están acostumbrados a utilizar, como el objeto *App* o la forma de acceder a la colección de los formularios cargados en la memoria.

El objeto My.Computer.FileSystem

En cuanto a lo que respecta al sistema de archivos, la mayoría de la funcionalidad está incluida en el objeto *My.Computer.FileSystem*. Si bien, debemos aclarar que esos objetos o propiedades, realmente son una especie de acceso directo a las clases del propio .NET Framework, y esa es la excusa de no haber tratado este objeto con más profundidad, ya que si sabemos manejar el resto de clases de .NET, sabremos trabajar con el objeto *My*.

Entre las propiedades del objeto *My.Computer.FileSystem*, podemos encontrar muchos de los métodos que exponen las clases *File* y *Directory*, aunque estos métodos tendrán nombres diferentes, por ejemplo, en lugar de tener un método *Exists*, nos encontramos con uno para averiguar si existe un directorio: *DirectoryExists*, y otro para averiguar si existe un fichero: *FileExists*.

También quisiéramos resaltar dos métodos que pueden ser de bastante utilidad cuando estemos buscando texto en los ficheros o cuando queramos abrir un fichero con una estructura determinada, nos estamos refiriendo a los métodos: *FindInFiles* y *OpenTextFieldParse* respectivamente.

Buscar texto en ficheros con FindInFiles

FindInFiles devuelve una colección "generic" de tipo *String* y de solo lectura con los ficheros que contienen el texto indicado.

En el siguiente ejemplo buscamos ficheros que contenga el texto "hola", se ignore las mayúsculas y minúsculas y se haga la búsqueda también en los subdirectorios:

```
Dim list As System.Collections.ObjectModel.ReadOnlyCollection(Of String)
' Buscar texto en los ficheros.
list = My.Computer.FileSystem.FindInFiles( _
    "E:\Pruebas", "hola", _
    True, FileIO.SearchOption.SearchAllSubDirectories)
' Mostrar los ficheros con el texto indicado.
For Each name As String In list
    Console.WriteLine(name)
Next
```

Examinar el contenido de un fichero con formato con OpenTextFieldParse

En el siguiente ejemplo vamos a abrir un fichero con una estructura, realmente delimitado con tabuladores, leeremos el contenido y lo mostraremos examinando cada uno de los campos.

En este ejemplo, podríamos llamar al método *TextParser* de la siguiente forma:
textParser("E:\Pruebas\Prueba parse.txt", vbTab)

```
Sub textParser(ByVal fic As String, ByVal sep As String)
    ' Creamos el "parser".
    Dim reader As FileIO.TextFieldParser
    reader = My.Computer.FileSystem.OpenTextFieldParser(fic)
    ' Le indicamos que buscaremos delimitadores.
    reader.TextFieldType = FileIO.FieldType.Delimited
    ' Un array con los delimitadores.
    reader.Delimiters = New String() {sep}
    Dim fila() As String
    ' Mientras haya datos...
    While Not reader.EndOfData
        Try
            ' Leemos todos los campos.
            fila = reader.ReadFields()
            Console.WriteLine("Los campos de la fila son:")
            ' Los mostramos.
            For Each campo As String In fila
                Console.Write("{0}, ", campo)
            Next
            Console.WriteLine()
        Catch ex As Exception
            Console.WriteLine("Error: " & ex.Message)
        End Try
    End While
End Sub
```

Lección 4: Acceso a Internet

- Las clases de System.Net
- Acceder a una página Web
- Acceder a un servidor FTP
- Acceso rápido a la red con My.Computer.Network

Introducción

Esta es otra de las áreas con la que los desarrolladores de VB6 se verán beneficiados, al menos si nuestra intención es la acceder a la red de redes: Internet.

Aunque también tendremos clases para interactuar con nuestra red local o empresarial, además de clases para comunicación por "sockets" y acceso a FTP, etc.

Las mayoría de las clases que necesitemos para acceder a la red, las encontraremos en el espacio de nombres *System.Net*.

Acceso a Internet

- [System.Net: Las clases para acceder a la red](#)
 - [Las clases de System.Net](#)
 - [Acceder a una página Web](#)
 - [Acceder a un servidor FTP](#)
 - [Acceso rápido a la red con My.Computer.Network](#)
 - Averiguar si tenemos red disponible
 - Hacer Ping a un equipo
 - Bajar un fichero desde una dirección Web
 - [Obtener información de la red con las clases de .NET](#)

Lección 4: Acceso a Internet

- Las clases de System.Net
- Acceder a una página Web
- Acceder a un servidor FTP
- Acceso rápido a la red con My.Computer.Network

System.Net: Las clases para acceder a la red

En el espacio de nombres *System.Net* tenemos todo lo que necesitamos para acceder a Internet y/o a otros recursos que estén en una red local (intranet).

Debido a que son muchas las clases, y algunas solo las necesitaremos en muy contadas ocasiones, vamos a intentar centrarnos en las que consideramos más importantes.

Las clases de System.Net

En este espacio de nombres tenemos una gran cantidad de clases, entre las que podemos destacar las siguientes.

- *AuthenticationManager*, administra los módulos de autenticación durante el proceso de autenticación del cliente.
- *Authorization*, contiene un mensaje de autenticación de un servidor de Internet.
- *Cookie*, proporciona métodos y propiedades para administrar los cookies.
- *CredentialCache*, proporciona almacenamiento para múltiples credenciales.
- *Dns*, proporciona funcionalidad simple para resolución de nombres de dominios.
- *DnsPermission*, controla los permisos de acceso a servidores DNS en la red.
- *EndPoint*, clase abstracta que identifica una dirección de red.
- *FileWebRequest*, proporciona una implementación del sistema de archivos de la clase *WebRequest*.
- *FileWebResponse*, proporciona una implementación del sistema de archivos de la clase *WebResponse*.
- *FtpWebRequest*, implementa un cliente FTP.
- *FtpWebResponse*, encapsula una respuesta desde una petición a un servidor FTP.
- *HttpListener*, proporciona un protocolo de escucha HTTP simple.
- *HttpListenerBasicIdentity*, proporciona la identidad para la clase *HttpListener*.
- *HttpListenerContext*, proporciona acceso a las peticiones y respuestas utilizadas por la clase *HttpListener*.
- *HttpListenerRequest*, describe una petición HTTP a un objeto *HttpListener*.
- *HttpListenerResponse*, representa una respuesta a una petición administrada por un objeto *HttpListener*.
- *HttpVersion*, define las versiones HTTP soportadas por las clases *HttpWebRequest* y *HttpWebResponse*.
- *HttpWebRequest*, proporciona una implementación HTTP específica de la clase *WebRequest*.
- *HttpWebResponse*, proporciona una implementación HTTP específica de la clase *WebResponse*.
- *IPAddress*, proporciona una dirección IP (*Internet Protocol*).
- *IPEndPoint*, representa un punto final de red como una dirección IP y un número de puerto.

- *IPHostEntry*, proporciona una clase contenedora para la información de dirección de host de Internet.
- *NetworkCredential*, proporciona credenciales para autenticación basada en contraseña.
- *ServicePoint*, proporciona administración de conexiones para las conexiones HTTP.
- *ServicePointManager*, administra la colección de objetos *ServicePoint*.
- *SocketAddress*, almacena información serializada de las clases derivadas de *EndPoint*.
- *SocketPermission*, controla los derechos para realizar o aceptar conexiones en una dirección de transporte.
- *WebClient*, proporciona métodos comunes para enviar o recibir datos desde un recurso identificado por una URI (*Uniform Resource Identifier*).
- *WebPermission*, controla los derechos de acceso a un recurso HTTP de Internet.
- *WebProxy*, contiene la configuración del proxy HTTP de la clase *WebRequest*.
- *WebRequest*, realiza una petición a una URI.
- *WebRequestMethods*, clase contenedora para las clases *WebRequestMethods.Ftp*, *WebRequestMethods.File*, y *WebRequestMethods.Http*.
- *WebRequestMethods.File*, representa los tipos del protocolo de fichero que se pueden usar con una petición FILE.
- *WebRequestMethods.Ftp*, representa los tipos del protocolo FTP que se pueden usar con una petición FTP.
- *WebRequestMethods.Http*, representa los tipos del protocolo HTTP que se pueden usar con una petición HTTP.
- *WebResponse*, proporciona una respuesta desde una URI.

Acceder a una página Web

Empezaremos viendo cómo acceder a una página Web. Para este tipo de acceso vamos a usar la clase abstracta *WebRequest* y de paso algunas otras clases como *WebResponse* y otras para manejar el stream recibido con el contenido de la página solicitada.

Veamos un pequeño ejemplo, (basado y simplificado de uno de la documentación), para hacerlo:

```
Sub leerPaginaWeb(ByVal laUrl As String)
    ' Cear la solicitud de la URL.
    Dim request As WebRequest = WebRequest.Create(laUrl)
    ' Obtener la respuesta.
    Dim response As WebResponse = request.GetResponse()
    ' Abrir el stream de la respuesta recibida.
    Dim reader As New StreamReader(response.GetResponseStream())
    ' Leer el contenido.
    Dim res As String = reader.ReadToEnd()
    ' Mostrarlo.
    Console.WriteLine(res)
    ' Cerrar los streams abiertos.
    reader.Close()
    response.Close()
End Sub
```

Si la dirección que estamos solicitando es una página "activa", el valor que recibiremos es el código "cliente", es decir, el código que se enviará al navegador.

Si necesitamos hacer algunas peticiones más específicas o necesitamos obtener alguna otra información podemos usar un objeto del tipo *HttpWebRequest* el cual está basado en la clase *WebRequest*, pero que la amplía para ofrecer otros tipos de información y acciones más adecuadas a una petición HTTP.

La forma de usar esta clase sería igual que *WebRequest*, además de que la documentación recomienda usar el método *Create* de *WebRequest* para crear una nueva instancia de *HttpWebRequest*.

En el siguiente ejemplo, le indicamos que estamos usando un navegador desde un "Smartphone":

```
Sub leerPaginaWeb2(ByVal laUrl As String)
    ' Cear la solicitud de la URL.
    Dim hRequest As HttpWebRequest =
        CType(WebRequest.Create(laUrl), HttpWebRequest)
    ' para que lo devuelva como si accediéramos con un Smartphone
    hRequest.UserAgent =
        "Mozilla/4.0 (compatible; MSIE 4.01; Windows CE; Smartphone; 176x220)"
```

```

' Obtener la respuesta y abrir el stream de la respuesta recibida.
Dim reader As New StreamReader(hRequest.GetResponse.GetResponseStream)
Dim res As String = reader.ReadToEnd()
' Mostrarlo.
Console.WriteLine(res)
' Cerrar el stream abierto.
reader.Close()
End Sub

```

Acceder a un servidor FTP

Por medio de la clase *FtpWebRequest* podemos acceder a una dirección FTP de forma bastante fácil, como es habitual en este tipo de clases, este objeto se crea mediante el método compartido *Create*, al que le debemos pasar la dirección FTP a la que queremos acceder.

Debido a que los sitios FTP pueden estar protegidos por contraseña, es posible que necesitemos asignar la propiedad *Credentials* de este objeto, esto lo podemos hacer asignando el objeto creado a partir de la clase *NetworkCredential*.

Los distintos comandos que necesitemos enviar al FTP lo haremos mediante la propiedad *Method*.

En el siguiente ejemplo utilizamos la clase *FtpWebRequest* para listar el contenido de un directorio FTP público.

```

Sub Main()
    listarFTP("ftp://ftp.rediris.es", "anonymous@nadie.com", "")
    '
    Console.ReadLine()
End Sub

Sub listarFTP(ByVal dir As String, ByVal user As String, ByVal pass As String)
    Dim dirFtp As FtpWebRequest = CType(FtpWebRequest.Create(dir), FtpWebRequest)
    Dim cr As New NetworkCredential(user, pass)
    dirFtp.Credentials = cr
    dirFtp.Method = "LIST"
    Dim reader As New StreamReader(dirFtp.GetResponse.GetResponseStream())
    Dim res As String = reader.ReadToEnd()
    ' Mostrarlo.
    Console.WriteLine(res)
    ' Cerrar el stream abierto.
    reader.Close()
End Sub

```

Acceso rápido a la red con My.Computer.Network

El objeto *My* también contiene objetos con los que podemos acceder a ciertas características de la red, en particular mediante el objeto *My.Computer.Network* tenemos acceso a una propiedad y tres métodos con los que podemos hacer lo siguiente:

- *IsAvailable*, esta propiedad nos permite saber si la red está disponible.
- *Ping*, con este método podemos hacer un "ping" a un equipo remoto.
- *DownloadFile*, nos permite bajar un fichero desde una dirección Web.
- *UploadFile*, nos permite enviar un fichero a una dirección Web.

En el siguiente código tenemos ejemplos de estos métodos y propiedades:

```

If My.Computer.Network.IsAvailable = True Then
    Console.WriteLine("El ordenador está conectado.")
Else
    Console.WriteLine("El ordenador no está conectado.")
End If
'
If My.Computer.Network.Ping("192.168.1.26") Then
    Console.WriteLine("El equipo está activo.")
Else
    Console.WriteLine("Se ha sobrepasado el tiempo de espera.")
End If
'
Dim ficDest As String = "E:\Pruebas\robots.txt"
My.Computer.Network.DownloadFile(
    "http://www.elguille.info/robots.txt", _
    ficDest)

```

```

' 
If My.Computer.FileSystem.FileExists(ficDest) Then
    Console.WriteLine("El fichero se ha bajado")
Else
    Console.WriteLine("El fichero no se ha bajado")
End If

```

Obtener información de la red con las clases de .NET

Como acabamos de ver, el espacio de nombres System.Net incluye algunas clases y otros espacios de nombres que son parte de las novedades de .NET Framework 2.0, que es al fin y al cabo el entorno que proporciona la librería de clases en las que se apoya Visual Basic 2010 para obtener funcionalidad; entre ellos tenemos *NetworkInformation*, el cual incluye clases que nos permitirán comprobar si tenemos conexión a la red, hacer un ping, obtener la dirección MAC del adaptador de red y algunos etcéteras más, de esas clases es la que se sirve el objeto *My.Computer.Network* para obtener la funcionalidad, ya que en realidad todos los objetos de *My* se basan en clases existentes en .NET.

En el siguiente ejemplo, podemos ver algunas de esas "cosillas" que podemos hacer ahora sin necesidad de "rebuscar" en la funcionalidad del API de Windows.

```

Imports Microsoft.VisualBasic
Imports System
Imports System.Net
Imports System.Net.NetworkInformation

Module Module1

    Sub Main()
        probarConexionARed()
        Console.WriteLine()

        realizarPing()
        Console.WriteLine()

        adaptadoresRed()
        Console.ReadLine()

    End Sub

    ' Realizar un ping
    Sub realizarPing()
        Console.WriteLine("Loopback: {0}{1}", IPAddress.Loopback, vbCrLf)
        Dim ip As IPAddress = IPAddress.Parse("127.0.0.1")
        Dim ping As Ping = New Ping
        For i As Integer = 0 To 3
            Dim pr As PingReply = ping.Send(ip)
            Console.Write("Respuesta desde {0}: bytes: {1}, ", pr.Address, pr.Buffer.Length)
            Console.WriteLine("tiempo= {0} ({1})", pr.RoundtripTime, pr.Status)
        Next
    End Sub

    ' Comprobar si tenemos conexión de red
    Sub probarConexionARed()
        If NetworkInterface.GetIsNetworkAvailable Then
            Console.WriteLine("Conexion disponible")
        Else
            Console.WriteLine("Conexion NO disponible")
        End If
    End Sub

    ' Muestra información de los adaptadores de red que tenemos
    Sub adaptadoresRed()
        For Each ni As NetworkInterface In NetworkInterface.GetAllNetworkInterfaces()
            Console.WriteLine("Descripción: {0}", ni.Description)
            Console.WriteLine("Estado: {0}", ni.OperationalStatus)
            Console.WriteLine("Velocidad: {0:n} bps", ni.Speed)
            Console.WriteLine("Dirección MAC: {0}{1}", ni.GetPhysicalAddress, vbCrLf)
        Next
    End Sub

End Module

```


Módulo 5: Acceso a Datos

- Descripción ADO.NET
- Acceso conectado a base de datos
- Acceso desconectado: Datasets y DataAdapters
- Datasets tipados
- Enlace a formularios

Módulo 5 - Acceso a datos

En este módulo, aprenderemos a trabajar con datos y fuentes de datos en Visual Basic 2010. ADO.NET es la tecnología principal para conectarse a una base de datos , nos ofrece un alto nivel de abstracción, ocultando los detalles de bajo nivel de la implementación de la base de datos de un fabricante.

En este tutorial, encontrará las cosas más importantes que debe saber, para trabajar con fuentes de datos con Visual Basic 2010.

De esta manera, aprenderá en poco tiempo, a encontrarse cómodo en el entorno de clases de ADO.NET y podrá así, sacar el máximo provecho a sus desarrollos.

Las partes que forman parte de este módulo son las siguientes:

Capítulo 1

- [Descripción de ADO.NET](#)

Capítulo 2

- [Acceso conectado a bases de datos](#)

Capítulo 3

- [Acceso desconectado: DataSets y DataAdapters](#)

Capítulo 4

- [DataSets tipados](#)

Capítulo 5

- [Enlace a formularios](#)

Lección 1: Descripción ADO.NET

- **Acercándonos a ADO.NET**
- **System.Data**
- **Los proveedores de acceso a datos**
- **El concepto de DataBinding**
- **Otras consideraciones**

Introducción

A continuación veremos todo lo que debe saber sobre ADO.NET para crear aplicaciones que accedan a fuentes de datos desde Visual Basic 2010.

Comprobará que ahora, es incluso mucho más fácil y rápido, si bien, es necesario conocer el modelo con el que se trabaja para poder saber lo que deberemos hacer en un momento dado.

Módulo 5 - Capítulo 1

- 1. [Acercándonos a ADO.NET](#)
- 2. [System.Data](#)
- 3. [Los proveedores de acceso a datos](#)
- 4. [El concepto DataBinding](#)
- 5. [Otras consideraciones](#)

Lección 1: Descripción ADO.NET

Acercándonos a ADO.NET

- **System.Data**
- Los proveedores de acceso a datos
- El concepto de DataBinding
- Otras consideraciones

Módulo 5 - Capítulo 1

1. Acercándonos a ADO.NET

ADO.NET ha sufrido a lo largo de los últimos años diferentes mejoras y actualizaciones, desde que .NET apareció.

El resumen de las diferentes versiones de ADO.NET podría quedar de la siguiente forma.

ADO.NET 1.0 apareció con Microsoft .NET Framework 1.0. Posteriormente, ADO.NET 1.1 sufrió una pequeñas y casi inapreciables actualizaciones con la aparición de Microsoft .NET Framework 1.1. En el caso del entorno Visual Studio 2010, éste trabaja con el CLR de Microsoft .NET Framework 2.0 y por lo tanto, utiliza ADO.NET 2.0, el cuál añade algunas características nuevas adicionales.

En nuestro caso, nos centraremos única y exclusivamente en ADO.NET como modelo de objetos de acceso a datos para la plataforma .NET de Microsoft, ya que es el mismo para cualquier tipo de versión de ADO.NET.

¿Qué es ADO.NET?

ADO.NET es la tecnología principal para conectarse a un gestor de bases de datos, con un alto nivel de abstracción, lo que nos permite olvidarnos de los detalles de bajo nivel de las bases de datos. Además ADO.NET es una tecnología interoperativa. Aparte del almacenamiento y recuperación de datos, ADO.NET introduce la posibilidad de integrarse con el estándar XML, los datos pueden 'Serializarse' directamente a y desde XML lo que favorece el intercambio de información.

ADO.NET proporciona diferentes clases del nombre de espacio **System.Data** dentro de las cuales, destacaremos por encima de todas, la clase **DataView**, la clase **DataSet** y la clase **DataTable**.

Este conjunto de clases de carácter armónico, funcionan de igual forma con la capa inferior que es la que corresponde a los proveedores de acceso a datos con los que podemos trabajar.

Esto facilita el trabajo en n-capas y la posible migración de aplicaciones que utilicen una determinada fuente de datos y deseemos en un momento dado, hacer uso de otra fuente de datos.

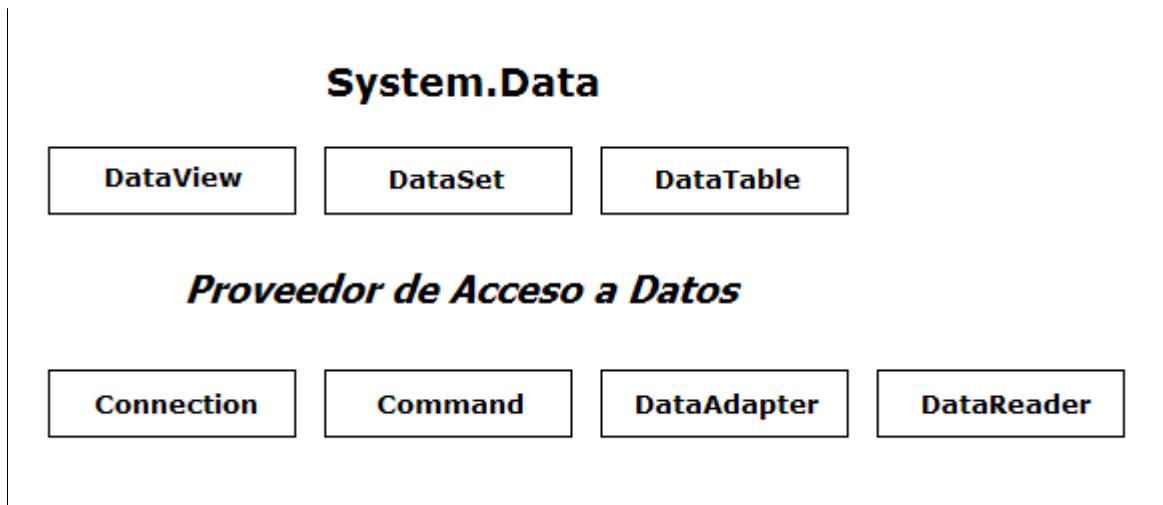
¿Qué capas o qué partes hay dentro de ADO.NET?

Dentro de ADO.NET tenemos dos partes importantes.

La primera de ellas es la que corresponde con el nombre de espacio **System.Data** y que constituye los objetos y clases globales de ADO.NET.

La otra parte es la que corresponde con los objetos que permiten el acceso a datos a una determinada fuente de datos desde ADO.NET y que utilizan así mismo, las clases del nombre de espacio **System.Data**.

Esta última parte, queda constituida por las clases y objetos de los diferentes proveedores de acceso a datos como se muestra en la figura 1.



Visión general de las clases de ADO.NET

Figura 1

Para resumir de alguna forma lo que estamos comentando, diremos que el trabajo de conexión con la base de datos, la ejecución de una instrucción SQL determinada, una vista, etc., la realiza el proveedor de acceso a datos.

Recuperar esos datos para tratarlos, manipularlos o volcarlos a un determinado control o dispositivo, es acción de la capa superior que corresponde con el nombre de espacio **System.Data**.

A continuación veremos todo esto con más detalle y comprenderemos de una forma más clara cada una de las partes que componen el modelo de trabajo con ADO.NET.

¿Qué nos permite realmente ADO.NET cuando trabajamos con XML?

El entorno de Microsoft .NET Framework nos proporciona el trabajo con estándares y con ello, la posibilidad de trabajar con diferentes tipos de aplicaciones, entornos, sistemas operativos y lenguajes sin necesidad de conocer lo que hay al otro lado de nuestra aplicación.

XML es sin lugar a dudas, el lenguaje de etiquetas por excelencia, válido para llevar a cabo esta tarea sin tener un impacto relevante cuando trabajamos con diferentes soluciones en entornos dispares.

Tanto la posibilidad de trabajar con Servicios Web XML como con documentos e información en XML, sobre todo al trabajar con fuentes de datos en ADO.NET, nos proporciona a los desarrolladores las posibilidades necesarias que nos permite hacer que la información con la que trabajamos, pueda ser tratada entre diferentes sistemas o entornos, sin que por ello nos preocupemos de lo que hay al otro lado.

Lección 1: Descripción ADO.NET

- Acercándonos a ADO.NET

System.Data

- Los proveedores de acceso a datos
- El concepto de DataBinding
- Otras consideraciones

Módulo 5 - Capítulo 1

2. System.Data

Las clases del nombre de espacio **System.Data** son bastante extensas y variadas.

Quizás las clases más importantes son la clase **DataView**, la clase **DataSet** y la clase **DataTable**.

La clase DataSet

El *DataSet* es una representación de datos residente en memoria que proporciona una modelo de programación relacional coherente independientemente del origen de datos que contiene. El *DataSet* contiene en sí, un conjunto de datos que han sido volcados desde el proveedor de datos.

Debemos tener en cuenta que cuando trabajamos con *Datasets*, el origen de datos no es lo más importante, ya que éste, puede ser cualquier tipo de origen de datos. No tiene porqué ser una base de datos.

Un *DataSet* contiene colecciones de *DataTables* y *DataRelations*.

El *DataTable* contiene una tabla o tablas, mientras que la *DataRelation* contiene las relaciones entre las *DataTables*.

Sin embargo, no es necesario especificar todo esto hasta el último detalle como veremos más adelante.

La clase DataView

Este objeto nos permite crear múltiples vistas de nuestros datos, además de permitirnos presentar los datos.

Es la clase que nos permite representar los datos de la clase *DataTable*, permitiéndonos editar, ordenar y filtrar, buscar y navegar por un conjunto de datos determinado.

La clase DataTable

Este objeto nos permite representar una determinada tabla en memoria, de modo que podamos interactuar con ella.

A la hora de trabajar con este objeto, debemos tener en cuenta el nombre con el cuál definimos una determinada tabla, ya que los objetos declarados en el *DataTable* es sensible a mayúsculas y minúsculas.

Un pequeño ejemplo práctico

El siguiente ejemplo práctico, nos enseña a utilizar un *DataSet* y nos muestra como podemos acceder a los objetos que dependen de un *DataSet* para recuperar por ejemplo, los campos y propiedades de una determinada tabla o tablas.

Código

```
Imports System.Data
Imports System.Data.SqlClient
Imports System.Xml

Public Class Form1

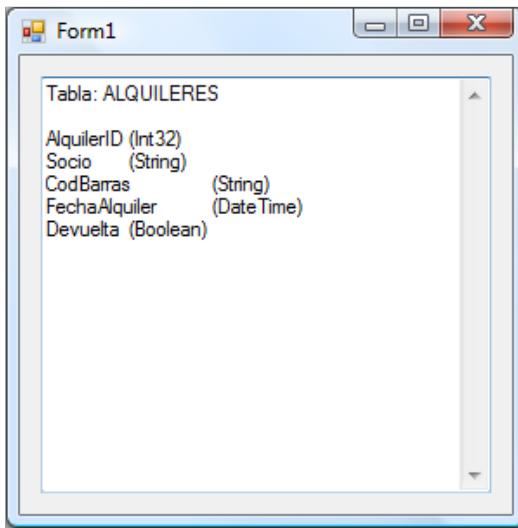
    Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.Load
```

```

Dim Conexion As String = "server=.;uid=sa;password=VisualBasic;database=MSDNVideo"
Dim MiTabla As DataTable
Dim MiColumna As DataColumn
Dim MiDataSet As New DataSet()
Dim Comando As New SqlDataAdapter("SELECT * FROM ALQUILERES", Conexion)
Comando.Fill(MiDataSet, "ALQUILERES")
'Recorremos las tablas
For Each MiTabla In MiDataSet.Tables
    TextBox1.Text += "Tabla: " & MiTabla.TableName & vbCrLf & vbCrLf
    'Recorremos las Columnas de cada Tabla
    For Each MiColumna In MiTabla.Columns
        TextBox1.Text += MiColumna.ColumnName & vbTab & _
                        "(" & MiColumna.DataType.Name & ")" & vbCrLf
    Next
Next
Comando = Nothing
End Sub
End Class

```

Nuestro ejemplo en ejecución es el que se muestra en la figura 1.



Ejemplo en ejecución del uso de DataSet, DataTable y DataColumn
Figura 1

Lección 1: Descripción ADO.NET

- Acercándonos a ADO.NET
- System.Data
- Los proveedores de acceso a datos**
- El concepto de DataBinding
- Otras consideraciones

Módulo 5 - Capítulo 1

3. Los proveedores de acceso a datos

Los proveedores de acceso a datos es la capa inferior de la parte correspondiente al acceso de datos y es la responsable de establecer la comunicación con las fuentes de datos.

En este conjunto de nombres de espacio, encontraremos casi siempre las clases **Connection**, **Command**, **DataAdapter** y **DataReader** como las clases más generales, las cuales nos permiten establecer la conexión con la fuente de datos.

Proveedores de acceso a datos de .NET Framework

Dentro del entorno .NET Framework, encontramos un nutrido conjunto de proveedores de acceso a datos.

Estos son los siguientes:

- *ODBC .NET Data Provider*
- *OLE DB .NET Data Provider*
- *Oracle Client .NET Data Provider*
- *SQL Server .NET Data Provider*

Estos proveedores de acceso a datos incluidos en Microsoft .NET Framework, los podemos encontrar en los nombres de espacio:

- *System.Data.Odbc*
- *System.Data.OleDb*
- *System.Data.OracleClient*
- *System.Data.SqlClient*

El proveedor ODBC .NET permite conectar nuestras aplicaciones a fuentes de datos a través de ODBC.

El proveedor OLE DB .NET permite conectar nuestras aplicaciones a fuentes de datos a través de OLE DB.

El proveedor Oracle Client .NET es un proveedor de acceso a datos especialmente diseñado para bases de datos Oracle.

Por último, el proveedor SQL Server .NET es un proveedor de acceso a datos nativo, que nos permite conectar nuestras aplicaciones a fuentes de datos Microsoft SQL Server 7.0 o posterior.

Se trata de un proveedor específico para bases de datos Microsoft SQL Server 7.0, Microsoft SQL Server 2000 y Microsoft SQL Server 2005 ó superior.

Consejo:

Siempre que pueda, utilice para acceder a fuentes de datos, un proveedor de acceso a datos nativo. Esto le permitirá aumentar considerablemente el rendimiento a la hora de establecer la conexión con una determinada fuente de datos

Los proveedores de acceso a datos que distribuye Microsoft en ADO.NET y algunos desarrollados por otras empresas o terceros, contienen los mismos objetos, aunque los nombres de éstos, sus propiedades y métodos, pueden ser diferentes.

Más adelante veremos algún ejemplo, y observará en la práctica cuáles son estas diferencias más destacables.

Otros proveedores de acceso a datos

Si bien el proveedor de acceso a datos es el *mecanismo* a través del cuál podemos establecer una comunicación nativa con una determinada fuente de datos, y dado que Microsoft proporciona los proveedores de acceso a datos más corrientes, es cierto que no los proporciona todos, si bien, con OLE DB y ODBC, podemos acceder a la inmensa totalidad de ellos.

Sin embargo, hay muchos motores de bases de datos de igual importancia como Oracle, MySql, AS/400, etc. En estos casos, si queremos utilizar un proveedor de acceso a datos nativo, deberemos acudir al fabricante o a empresas o iniciativas particulares para que nos proporcionen el conjunto de clases necesarias que nos permitan abordar esta acción.

El objeto Connection

Este objeto es el encargado de establecer una conexión física con una base de datos determinada.

Para establecer la conexión con una determinada fuente de datos, no sólo debemos establecer la cadena de conexión correctamente, sino que además deberemos usar los parámetros de conexión y el proveedor de acceso a datos adecuado.

Con este objeto, podremos además abrir y cerrar una conexión.

El objeto Command

Este objeto es el que representa una determinada sentencia SQL o un Stored Procedure.

Aunque no es obligatorio su uso, en caso de necesitarlo, lo utilizaremos conjuntamente con el objeto *DataAdapter* que es el encargado de ejecutar la instrucción indicada.

El objeto DataAdapter

Este objeto es quizás el objeto más complejo y a la vez complicado de todos los que forman parte de un proveedor de acceso a datos en .NET.

Cuando deseamos establecer una comunicación entre una fuente de datos y un *DataSet*, utilizamos como *intermediario* a un objeto *DataAdapter*.

A su vez, un *DataAdapter* contiene 4 objetos que debemos conocer:

- *SelectCommand* es el objeto encargado de realizar los trabajos de selección de datos con una fuente de datos dada.
En sí, es el que se encarga de devolver y llenar los datos de una fuente de datos a un *DataSet*.
- *DeleteCommand* es el objeto encargado de realizar las acciones de borrado de datos.
- *InsertCommand* es el objeto encargado de realizar las acciones de inserción de datos.
- *UpdateCommand* es el objeto encargado de realizar las acciones de actualización de datos.

Los objetos *DeleteCommand*, *InsertCommand* y *UpdateCommand* son los objetos que se utilizan para manipular y transmitir datos de una fuente de datos determinada, al contrario del objeto *SelectCommand* que tan sólo interactúa con la fuente de datos para recuperar una porción o todos los datos indicados en el objeto *Command* anteriormente comentado.

El objeto DataReader

Este objeto es el utilizado en una sola dirección de datos.

Se trata de un objeto de acceso a datos muy rápido.

Este objeto puede usar a su vez el objeto *Command* o el método *ExecuteReader*.

Lección 1: Descripción ADO.NET

- Acerca de ADO.NET
- System.Data
- Los proveedores de acceso a datos
- El concepto de DataBinding**
- Otras consideraciones

Módulo 5 - Capítulo 1

4. El concepto DataBinding

DataBinding es una expresión de enlace a datos. Como veremos a continuación es una forma rápida y sencilla de manejar la fuentes de datos mediante su enlace con controles o clases..

El uso de DataBind

El método *DataBind* se utiliza para llenar de datos un determinado control o clase.

Muchos controles y clases, posee este método al que le asignaremos un conjunto de datos para que se rellene con ellos, pudiendo después interactuar con los datos de forma directa.

En si, un *DataBinding* es un enlace a datos que se encarga de llenar de datos a un determinado control o clase.

Como ejemplo de esto, veremos como llenar un control **TextBox** con un dato utilizando este método.

Iniciaremos una nueva aplicación Windows y escribiremos el siguiente código fuente:

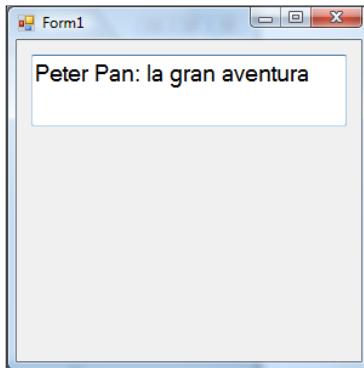
Código

```
Imports System.Data
Imports System.Data.SqlClient
Imports System.Xml

Public Class Form1

    Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.Load
        Dim Conexion As String = "server=.;uid=sa;password=VisualBasic;database=MSDNVideo"
        Dim MiDataSet As New DataSet()
        Dim Comando As New SqlDataAdapter("SELECT TITULO FROM ALQUILERES, PELICULAS WHERE PELICULACODBARRAS = CODBARRAS AND SOCIONIF = '111111'", Conexion)
        ' Rellenamos el DataSet con el contenido
        ' de la sentencia SELECT
        Comando.Fill(MiDataSet, "PELIS")
        ' Rellenamos el control TextBox1 con el
        ' dato correspondiente a la primera fila
        ' de la sentencia SELECT ejecutada
        TextBox1.DataBindings.Add("Text", MiDataSet, "PELIS.TITULO")
        Comando = Nothing
    End Sub
End Class
```

Nuestro ejemplo en ejecución es el que se muestra en la figura 1.



Ejemplo en ejecución del uso de DataBinding

Figura 1

Lección 1: Descripción ADO.NET

- Acercándonos a ADO.NET
 - System.Data
 - Los proveedores de acceso a datos
 - El concepto de DataBinding
- Otras consideraciones**

Módulo 5 - Capítulo 1

5. Otras consideraciones

Dentro de las conexiones a fuentes de datos, hay algunas partes de éstas que permanecen a veces en el olvido y su importancia sin embargo, es bastante grande.

La acción más pesada cuando realizamos un acceso a una fuente de datos, se encuentra en la conexión con la fuente de datos.

Esa tarea, simple tarea, es la que más recursos del sistema consume cuando accedemos a fuentes de datos.

Esto lo debemos tener en cuenta, y por lo tanto, variante de esto que comentamos son las siguientes premisas:

- La conexión debe realizarse siempre que se pueda, con los proveedores de acceso a datos nativos, que por lo general salvo raras excepciones, serán más rápidos que los accesos a fuentes de datos a través de proveedores del tipo OLE DB y ODBC.
- La conexión con la fuente de datos (apertura de la conexión), debe realizarse lo más tarde posible. Es recomendable definir todas las variables que podamos, antes de realizar la conexión.
- La conexión debe cerrarse lo antes posible, siempre y cuando no tengamos la necesidad de utilizar la conexión previamente abierta.

Hay más particularidades a tener en cuenta cuando trabajamos con fuentes de datos.

El hecho de que con un *DataSet* podamos trabajar con datos desconectados, no significa que dentro de él, podamos abrir una tabla con una cantidad de registros enormes, y trabajemos sobre ella creyendo que esto nos beneficiará.

Todo lo contrario.

Lección 2: Acceso conectado a bases de datos

- El paradigma de la conexión
- Conociendo el objeto DataReader
- Un primer contacto con el objeto DataReader
- ¿Trabaja DataReader en un ambiente conectado realmente?
- Usando DataSource con DataReader
- Usando los componentes de acceso a datos de .NET

Introducción

Ahora que ya tenemos un poco más clara la estructura del modelo de acceso a datos ADO.NET, podemos adentrarnos en todo lo que rodea al acceso conectado de base de datos y la manipulación y trabajo de datos conectados.

A continuación, encontrará el índice detallado de este capítulo.

Módulo 5 - Capítulo 2

- 1. [El paradigma de la conexión](#)
- 2. [Conociendo el objeto DataReader](#)
- 3. [Un primer contacto con el objeto DataReader](#)
- 4. [¿Trabaja DataReader en un ambiente conectado realmente?](#)
- 5. [Usando DataSource con DataReader](#)
- 6. [Usando los componentes de acceso a datos de .NET](#)

Lección 2: Acceso conectado a bases de datos

El paradigma de la conexión

- Conociendo el objeto DataReader
- Un primer contacto con el objeto DataReader
- ¿Trabaja DataReader en un ambiente conectado realmente?
- Usando DataSource con DataReader
- Usando los componentes de acceso a datos de .NET

Módulo 5 - Capítulo 2

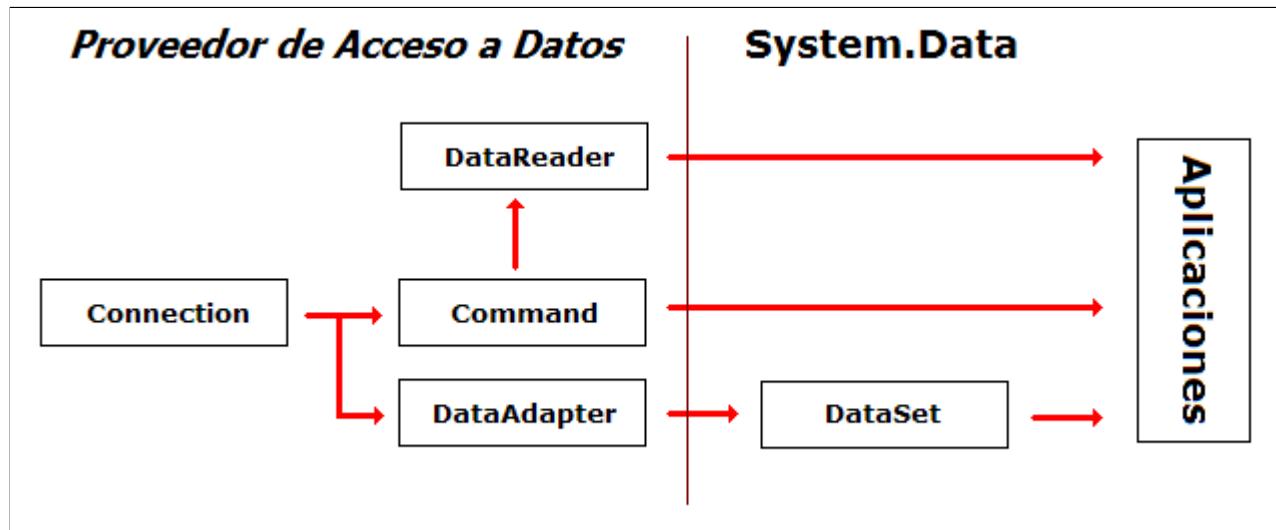
1. El paradigma de la conexión

Cuando abordamos un proyecto de acceso a fuentes de datos, siempre nos encontramos con una duda existencial.

¿Debemos crear una conexión con la base de datos al principio de nuestra aplicación y cerrarla cuando la aplicación se cierre?, ¿o debemos crear una conexión con la base de datos sólo cuando vayamos a trabajar con la fuente de datos?. ¿Y si estamos trabajando continuamente con una fuente de datos?, ¿cómo penalizarían todas estas acciones?.

Es difícil de asumir que acción tomar en cada caso, y es que dependiendo de lo que vayamos a realizar, a veces es más efectiva una acción que otra, y en otras ocasiones, no está del todo claro, ya que no existe en sí una regla clara que especifique qué acción tomar en un momento dado.

Lo que sí está claro es que el modelo de datos de ADO.NET que hemos visto, quedaría resumido en cuanto a la conectividad de la manera en la que se representa en la figura 1.



Visión general de ADO.NET respecto a la conectividad con bases de datos

Figura 1

El objeto *DataSet* nos ofrece la posibilidad de almacenar datos, tablas y bases de datos de una determinada

fuente de datos.

De esta manera, podemos trabajar con las aplicaciones estando desconectados de la fuente de datos.

Sin embargo, a veces necesitamos trabajar con la fuente de datos estando conectados a ella.

El objeto *DataReader* nos ofrece precisamente la posibilidad de trabajar con fuentes de datos conectadas.

Por otro lado, el objeto *DataReader* tiene algunas particularidades que conviene conocer y que veremos a continuación.

Lección 2: Acceso conectado a bases de datos

- El paradigma de la conexión
- Conociendo el objeto DataReader
- Un primer contacto con el objeto DataReader
- ¿Trabaja DataReader en un ambiente conectado realmente?
- Usando DataSource con DataReader
- Usando los componentes de acceso a datos de .NET

Módulo 5 - Capítulo 2

2. Conociendo el objeto DataReader

El objeto *DataReader* nos permite como hemos indicado anteriormente, establecer una conexión con una fuente de datos y trabajar con esta fuente de datos sin desconectarnos de ella, sin embargo, hay diferentes cualidades y particularidades que conviene conocer.

DataReader es de solo lectura

Lo que hemos dicho anteriormente, requiere sin embargo, que esta conexión se establezca en un modo de *sólo lectura*, al contrario de lo que se puede hacer con el objeto *DataSet*, con el que podemos interactuar con la fuente de datos en modo lectura y modo escritura.

DataReader se maneja en una sola dirección

El objeto *DataReader* sólo permite que nos desplazemos por los datos en una sola dirección, sin vuelta atrás. Por el contrario, el objeto *DataSet* nos permite movernos por los registros para adelante y para atrás.

Además, sólo podemos utilizar el objeto *DataReader* con conexiones establecidas en una sentencia SQL por ejemplo, pero no podemos variar esta.

Para hacerlo, debemos entonces modificar la conexión con el comando establecido.

DataReader es rápido

Debido a su naturaleza y características, este objeto es bastante rápido a la hora de trabajar con datos. Como es lógico, consume además menos memoria y recursos que un objeto *DataSet* por ejemplo. Sin embargo, dependiendo de las necesidades con las que nos encontremos, puede que este método de acceso y trabajo no sea el más idóneo.

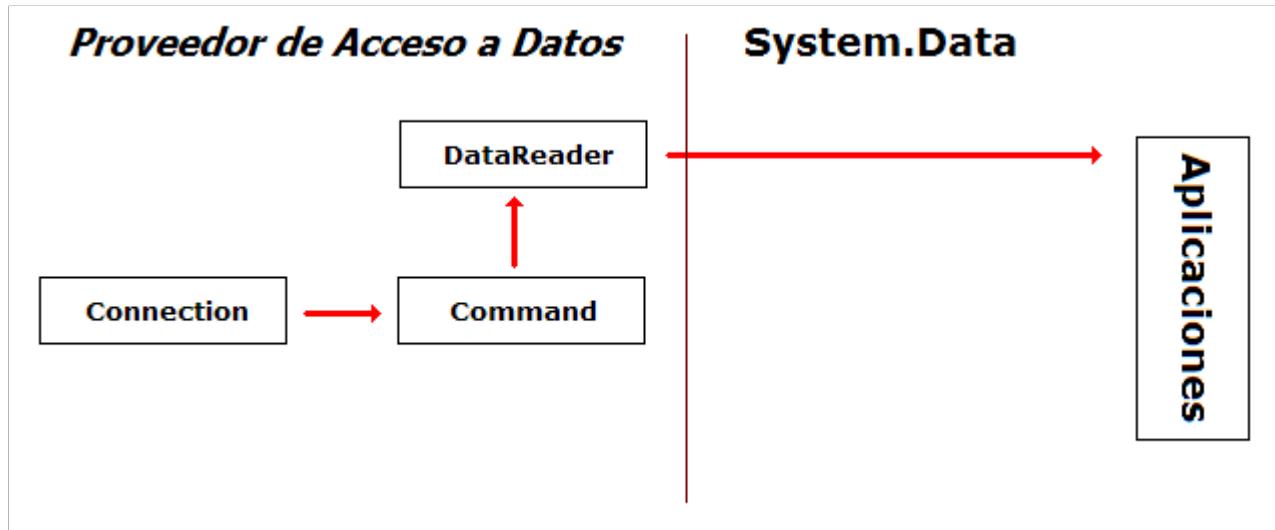
Analizando el flujo de trabajo de DataReader

Cuando trabajamos con fuentes de datos conectadas, trabajaremos con el objeto *DataReader*. Para trabajar con este objeto, utilizaremos los objetos siguientes del proveedor de acceso a datos:

- *Connection*

- *Command*
- *DataReader*

Un resumen gráfico de esto es lo que podemos ver en la figura 1.



El flujo de conectividad de *DataReader*

Figura 1

Lección 2: Acceso conectado a bases de datos

- El paradigma de la conexión
- Conociendo el objeto DataReader
- Un primer contacto con el objeto DataReader**
- ¿Trabaja DataReader en un ambiente conectado realmente?
- Usando DataSource con DataReader
- Usando los componentes de acceso a datos de .NET

Módulo 5 - Capítulo 2

3. Un primer contacto con el objeto DataReader

A continuación veremos un ejemplo sencillo sobre la forma de trabajar con *DataReader*

Un ejemplo simple para entenderlo mejor

Tenga en cuenta que en el siguiente ejemplo nos conectaremos a Microsoft SQL Server y recorreremos los registros uno a uno en un ambiente conectado y volcaremos estos registros en un control *TextBox*.

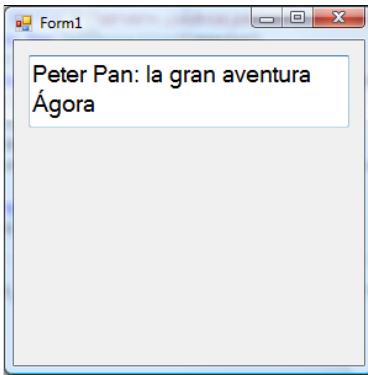
Código

```
Imports System.Data
Imports System.Data.SqlClient
Imports System.Xml

Public Class Form1

    Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.Load
        Dim Conexion As String = "server=.;uid=sa;password=VisualBasic;database=MSDNVideo"
        Dim MiConexion As New SqlConnection(Conexion)
        Dim MiDataReader As SqlDataReader
        Dim Comando As New SqlCommand("SELECT TITULO FROM ALQUILERES, PELICULAS WHERE PELICULACODBARRAS = CODBARRAS AND SOCIONIF = '1111111'", MiConexion)
        MiConexion.Open()
        Comando = Comando.ExecuteReader()
        While MiDataReader.Read()
            TextBox1.Text += MiDataReader("TITULO").ToString() & vbCrLf
        End While
        Comando = Nothing
        MiConexion.Close()
    End Sub
End Class
```

El código de ejemplo en ejecución es el que se muestra en la figura 1.



Ejemplo en ejecución del uso simple de DataReader

Figura 1

Este es un ejemplo simple del uso de *DataReader*.

Sin embargo, el objeto *DataReader* contiene un conjunto de propiedades y métodos que nos proporcionan acciones determinadas.

Por ejemplo, en el ejemplo anterior, hemos dado por hecho que la ejecución de la instrucción *Select* nos devolverá uno o más valores, pero podríamos también saber antes de manipular y trabajar con los posibles datos, si hay o no información.

Esto lo conseguimos con el método *HasRows*.

Lección 2: Acceso conectado a bases de datos

- El paradigma de la conexión
 - Conociendo el objeto DataReader
 - Un primer contacto con el objeto DataReader
- ¿Trabaja DataReader en un ambiente conectado realmente?**
- Usando DataSource con DataReader
 - Usando los componentes de acceso a datos de .NET

Módulo 5 - Capítulo 2

4. ¿Trabaja DataReader en un ambiente conectado realmente?

Pese a todo esto, ¿que ocurre si trabajando en un ambiente conectado se desconecta el servidor de acceso a datos?.

Imaginemos por un instante, que la conexión con SQL Server se establece correctamente y que en un momento dado se detiene el servicio del servidor de base de datos.

Esto es lo que veremos en el siguiente ejemplo.

Desenchufando la fuente de datos usando DataReader

Inicie un nuevo proyecto, inserte en el formulario de la aplicación un control **TextBox** y un control **Button**, y escriba el código que se detalla a continuación:

Código

```
Imports System.Data
Imports System.Data.SqlClient
Imports System.Xml

Public Class Form1

    Private Conexion As String = "server=.;uid=sa;password=VisualBasic;database=MSDNVideo"
    Private strSQL As String = "SELECT TITULO FROM ALQUILERES, PELICULAS WHERE PELICULACODBARRAS = CODBARRAS AND"
    Private MiConexion As New SqlConnection(Conexion)
    Private MiDataReader As SqlDataReader
    Private Contador As Long = 0
    Private Posicion As Long = 0

    Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.Load
        ' Establecemos la Conexión con la base de datos
        Establecer_Conexion(True)
        ' Si hay datos los mostramos, sino deshabilitamos
        ' la opción (botón) para recorrerlos
        If Not MiDataReader.HasRows Then
            Button1.Enabled = False
        Else
            Button1_Click(sender, e)
        End If
    End Sub

    Private Sub Establecer_Conexion(ByVal bolAccion As Boolean)
        Dim Comando As SqlCommand
        If bolAccion Then
            ' True => Establecemos la conexión
            Comando = New SqlCommand(strSQL, MiConexion)
            ' Abrimos la Conexión
            MiConexion.Open()
            ' Ejecutamos la sentencia SQL
            MiDataReader = Comando.ExecuteReader()
            ' Obtenemos la cantidad de registros obtenidos
            Contador = MiDataReader.VisibleFieldCount() + 1
        Else
            ' False => Finalizamos la conexión
            Button1.Enabled = False
            ' Cerramos la Conexión
        End If
    End Sub
```

```

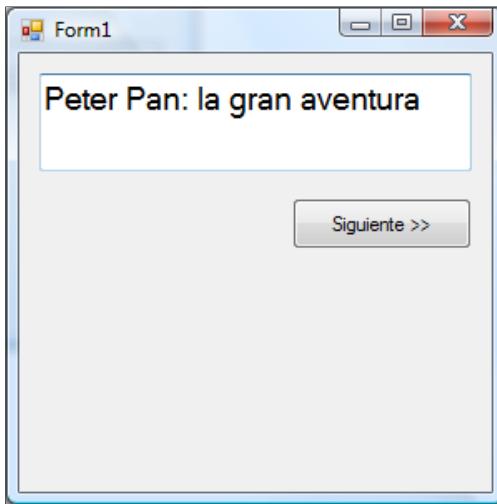
Comando = Nothing
MiConexion.Close()
End If
End Sub

Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button1.Click
    ' Recorremos los registros y los mostramos
    Posicion += 1
    MiDataReader.Read()
    TextBox1.Text = MiDataReader("TITULO")
    ' Si hemos recorrido el objeto por completo,
    ' finalizamos la Conexión y deshabilitamos el control
    ' Button que nos permite recuperar los registros
    If Posicion = Contador Then
        Establecer_Conexion(False)
    End If
End Sub
End Class

```

Suponiendo que tenemos en nuestra base de datos varios registros, ejecute la aplicación.

Si todo ha ido como se esperaba, observaremos que nuestra aplicación tiene un aspecto como el que se muestra en la figura 1.



Ejemplo en ejecución del uso de DataReader en un ambiente conectado, forzando la desconexión de la fuente de datos
Figura 1

En este punto, detenga el servicio de SQL Server y pulse el botón **Siguiente >>**.

Observará que la aplicación sigue funcionando.

En este punto se hará la pregunta que todos nos hemos hecho, ¿no es el objeto DataReader un objeto conectado?, ¿cómo es posible que funcione si hemos detenido el servicio de SQL Server?.

La respuesta es sencilla.

El objeto *DataReader* recupera un nutrido conjunto de valores llenando un pequeño buffer de datos e información.

Si el número de registros que hay en el buffer se acaban, el objeto *DataReader* regresará a la fuente de datos para recuperar más registros.

Si el servicio de SQL Server está detenido en ese momento o en su caso, la fuente de datos está parada, la aplicación generará un error a la hora de leer el siguiente registro.

En sí, *DataReader* es un objeto conectado, pero trabaja en *background* con un conjunto de datos, por lo que a veces nos puede resultar chocante su comportamiento como el ejemplo que comenté.

Lección 2: Acceso conectado a bases de datos

- El paradigma de la conexión
 - Conociendo el objeto DataReader
 - Un primer contacto con el objeto DataReader
 - ¿Trabaja DataReader en un ambiente conectado realmente?

Usando DataSource con DataReader

- Usando los componentes de acceso a datos de .NET

Módulo 5 - Capítulo 2

5. Usando DataSource con DataReader

¿Podemos usar el método *DataSource* con el objeto *DataReader*?

Demostración del uso de DataSource con DataReader

La respuesta es sí, en ADO.NET 2.0, se ha incorporado un nuevo método al objeto *DataTable* que le permite tener mayor independencia respecto al modo en el que nos conectemos y recuperemos datos de una fuente de datos.

Recuerde que podemos recuperar datos en modo conectado *DataReader* o en modo desconectado *DataSet*.

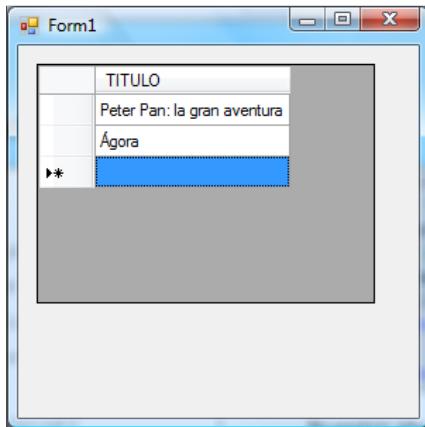
Este método que se ha incorporado a ADO.NET y que tiene por nombre **Load**, nos permite cargar un *DataReader* para volcarlo a continuación dentro de un control como por ejemplo el control *DataGridView*.

Lo mejor es que veamos como funciona esto con un ejemplo que nos ayude a comprender mejor la teoría.

Inserte en un formulario un control *DataGridView* y escriba el siguiente código:

Código

Nuestro ejemplo en ejecución es el que podemos ver en la figura 1.



Ejemplo en ejecución del uso de DataReader y DataSource en un control DataGridView

Figura 1

Con todo y con esto, lo que realmente es curioso, es que hemos olvidado por un instante que el objeto *DataReader* es un objeto de sólo lectura que funciona en una única dirección, hacia delante.

¿Qué significa esto o como puede influir o como podemos aprovechar esta circunstancia en nuestros desarrollos?

Carga segmentada de datos con DataSource y DataReader

Si recuperamos los datos de una fuente de datos con *DataReader* y leemos algunos de sus datos y posteriormente, ejecutamos el método *DataSource*, el resto de datos, aquellos datos que quedan en el *DataReader*, serán los que se vuelquen en el control que definamos como destino de los datos.

Imaginemos el ejemplo anterior, y el siguiente código fuente.

Código

En este caso, lo que ocurre como ya hemos comentado, es que los datos que se cargan son los que aún no han sido leídos en el objeto *DataReader*, por lo que se mostrarán todos los datos desde el último leído hasta llegar al final del objeto.

Lección 2: Acceso conectado a bases de datos

- El paradigma de la conexión
 - Conociendo el objeto DataReader
 - Un primer contacto con el objeto DataReader
 - ¿Trabaja DataReader en un ambiente conectado realmente?
 - Usando DataSource con DataReader

Módulo 5 - Capítulo 2

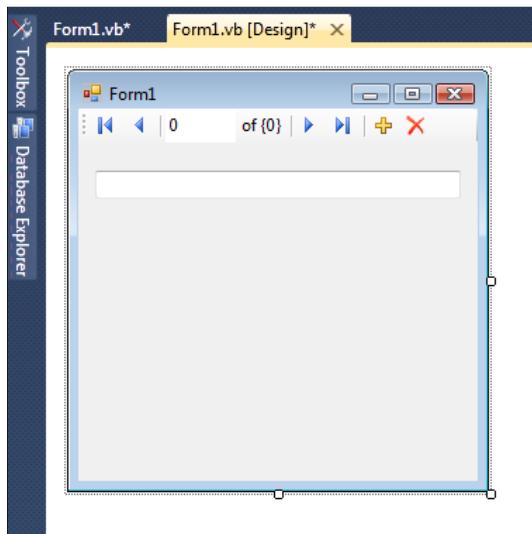
6. Usando los componentes de acceso a datos de .NET

Los componentes del entorno .NET nos proporcionan las características necesarias para poder acceder a fuentes de datos de forma rápida y sencilla. El mejor ejemplo de esto que comenté es el que veremos a continuación.

Demostración del uso de BindingSource y BindingNavigator

Para ello, crearemos un proyecto nuevo e insertaremos un control *BindingSource* y un control *BindingNavigator* dentro del formulario. También insertaremos un control *TextBox* al formulario, dónde presentaremos la información sobre la que navegaremos.

Nuestro formulario con los controles insertados en él, tendrá un aspecto similar al que se presenta en la figura 1.



Controles de navegación y acceso a datos dispuestos en el formulario

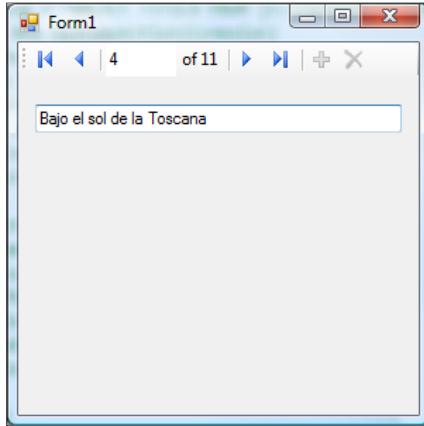
Figura 1

Una vez llegado a este punto, lo que tendremos que hacer a continuación será escribir el código fuente necesario para poder representar los datos de la sentencia SQL en el control *BingindNavigator*, para que a su vez los presente en el control *TextBox*.

A continuación se indica el código fuente de esta parte de demostración de la aplicación.

```
' Ejecutamos la sentencia SQL
MiDataReader = Comando.ExecuteReader()
' Cargamos los resultados en el objeto DataTable
MiDataTable.Load(MiDataReader, LoadOption.OverwriteChanges)
' Volcamos los datos en el control TextBox
BindingSource1.DataSource = MiDataTable
BindingNavigator1.BindingSource = BindingSource1
TextBox1.DataBindings.Add(New Binding("Text", BindingSource1, "TITULO", True))
' Cerramos la Conexión
Comando = Nothing
MiConexion.Close()
End Sub

End Class
```



Ejemplo anterior en ejecución

Figura 1

Lección 3: Acceso desconectado: DataSets y DataAdapters

- Esquema general de la estructura desconectada de acceso a datos
- Conociendo el objeto DataAdapter
- Insertando datos a través del objeto DataAdapter
- Actualizando datos a través del objeto DataAdapter
- Eliminando datos a través del objeto DataAdapter

Introducción

Ya tenemos claro el funcionamiento con fuentes de datos conectadas, sin embargo, trabajar con datos conectados sólo es necesario en algunos ámbitos, lo más habitual, será que nos encontremos trabajando con ambientes y accesos a datos desconectados, como ocurrirá en la inmensa mayoría de la veces.

A continuación, aprenderá a utilizar el *DataSet* y *DataAdapter* para sacar el máximo provecho a un ambiente desconectado de datos.

El índice detallado de este capítulo es el que se indica a continuación.

Módulo 5 - Capítulo 3

- 1. [Esquema general de la estructura desconectada de acceso a datos](#)
- 2. [Conociendo el objeto DataAdapter](#)
- 3. [Insertando datos a través del objeto DataAdapter](#)
- 4. [Actualizando datos a través del objeto DataAdapter](#)
- 5. [Eliminando datos a través del objeto DataAdapter](#)

Esquema general de la estructura desconectada de acceso a datos

- Conociendo el objeto DataAdapter
- Insertando datos a través del objeto DataAdapter
- Actualizando datos a través del objeto DataAdapter
- Eliminando datos a través del objeto DataAdapter

Módulo 5 - Capítulo 3

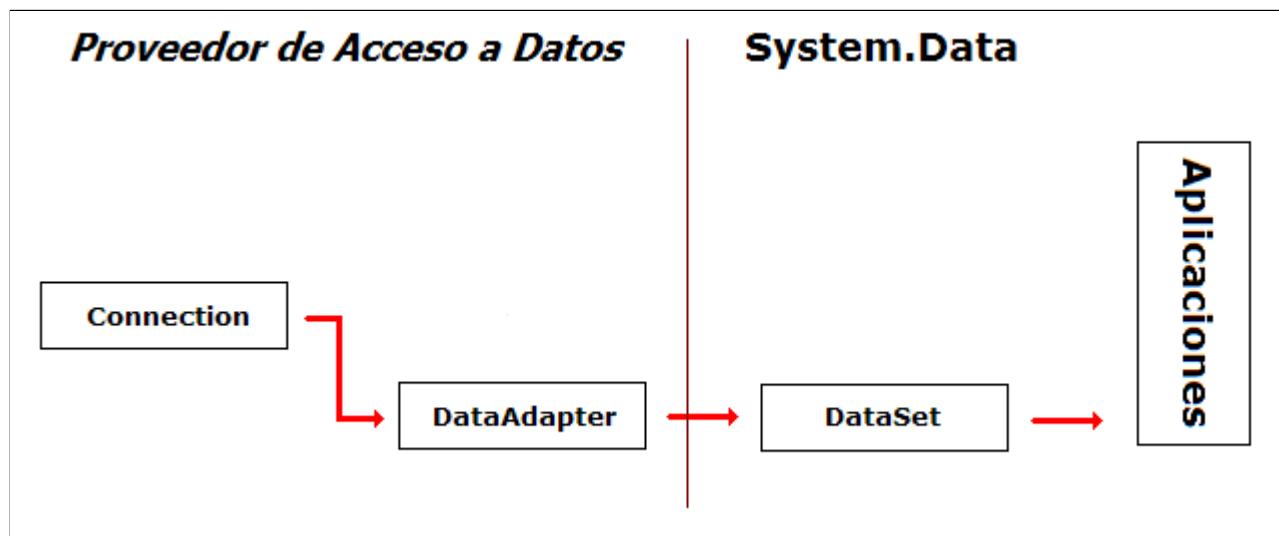
1. Esquema general de la estructura desconectada de acceso a datos

En los capítulos anteriores de este módulo, hemos visto ya el uso de la clase *DataSet*. Incluso lo hemos visto con algún ejemplo.

La clase *DataSet* está pensada y diseñada para trabajar con fuentes de datos desconectadas.

Indudablemente, en este punto, debemos tener clara la estructura general de cómo funciona el acceso desconectado con fuentes de datos.

En la figura 1, podemos observar el diagrama general de esta parte



Estructura general del uso de DataSet en el acceso desconectado a datos

Figura 1

Connection, DataAdapter y DataSet

Como podemos observar en la figura 1, para comunicarnos con una fuente de datos, siempre deberemos establecer una conexión, independientemente de si la conexión con la fuente de datos va a permanecer a lo largo del tiempo o no.

El objeto *Connection* nos permite por lo tanto, establecer la conexión con la fuente de datos. El objeto *DataSet* nos permite por otro lado, recoger los datos de la fuente de datos y mandárselos a la aplicación. Entre medias de estos dos objetos, encontramos el objeto *DataAdapter* que hace las funciones de puente o nexo de unión entre la conexión y el objeto *DataSet*. Esto es lo que veremos a continuación, como funciona el objeto *DataAdapter*, y como encaja todo esto en el acceso a fuentes de datos desconectadas.



Lección 3: Acceso desconectado: DataSets y DataAdapters

- Esquema general de la estructura desconectada de acceso a datos

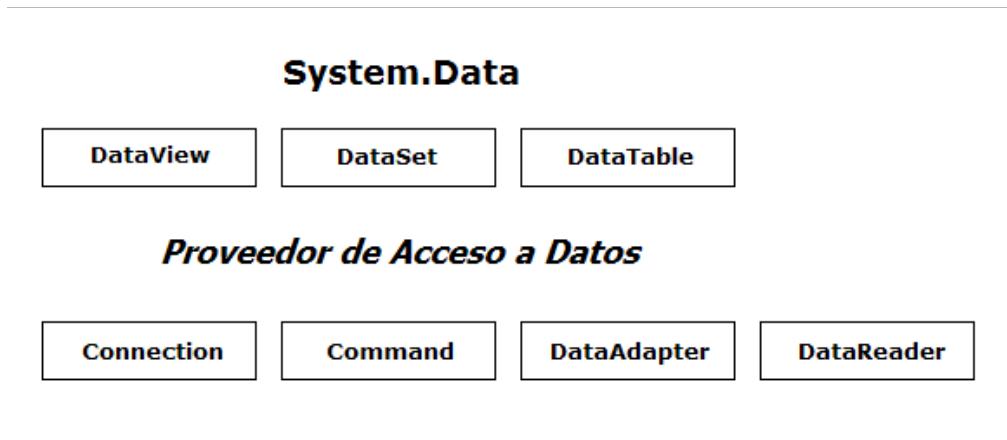
Conociendo el objeto DataAdapter

- Insertando datos a través del objeto DataAdapter
- Actualizando datos a través del objeto DataAdapter
- Eliminando datos a través del objeto DataAdapter

Módulo 5 - Capítulo 3

2. Conociendo el objeto DataAdapter

El objeto *DataAdapter* forma parte del proveedor de acceso a datos, tal y como se muestra en la figura 1.



Visión general de las clases de ADO.NET

Figura 1

Cada proveedor de acceso a datos posee su propio objeto *DataAdapter*.

Cuando realizamos alguna modificación o acción sobre la fuente de datos, utilizaremos siempre el objeto *DataAdapter* a caballo entre el objeto *DataSet* y la fuente de datos establecida a través de la conexión con el objeto *Connection*.

Con el objeto *DataAdapter*, podremos además realizar diferentes acciones sobre nuestras bases de datos, acciones como la ejecución general de sentencias de SQL no sólo para seleccionar un conjunto de datos, sino para alterar el contenido de una base de datos o de sus tablas.

Connection, DataAdapter y DataSet

Antes de entrar en materia más profundamente, diremos que en lo que respecta a los proveedores de acceso a datos que vienen integrados con .NET, encontramos dos formas de usar un *DataAdapter*.

La primera de ellas es utilizando los componentes del proveedor de acceso a datos.

La segunda de ellas es utilizando las clases del nombre de espacio del proveedor de acceso a datos.

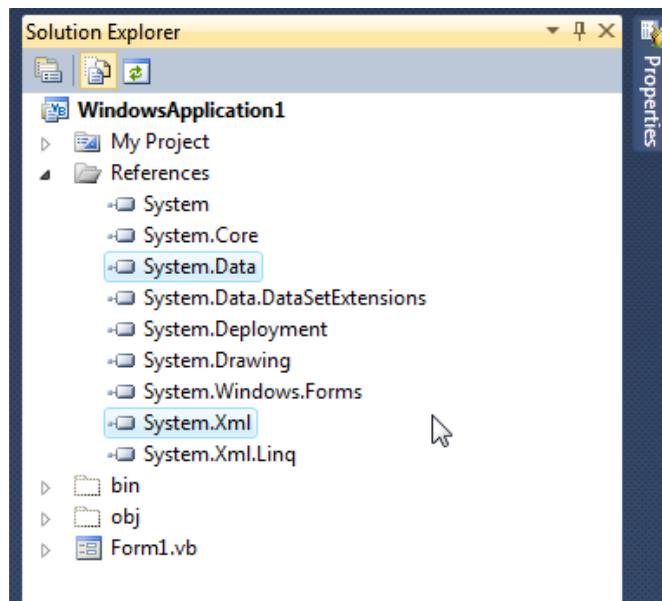
La mejor forma de entender todo esto que comentamos, es trabajando con un ejemplo práctico que nos enseñe a usar el objeto *DataAdapter* en Visual Studio 2010.

Utilizando las clases de .NET

En este primer ejemplo de demostración del uso de *DataAdapter* a través de código usando para ello las clases de .NET, estableceremos una conexión con SQL Server y mostraremos los datos recogidos en un control *TextBox*.

Iniciaremos Visual Studio 2010 y seleccionaremos un proyecto de formulario de Windows. Dentro del formulario, insertaremos un control *TextBox* y añadiremos dos referencias al proyecto.

Las referencias añadidas serán a las librerías *System.Data* y *System.XML*, como se muestra en la figura 2.



Referencias a las clases de acceso a datos de .NET

Figura 2

Una vez que hemos añadido las referencias necesarias para utilizar las clases que queremos en nuestro proyecto, iremos al código y escribiremos las siguientes instrucciones:

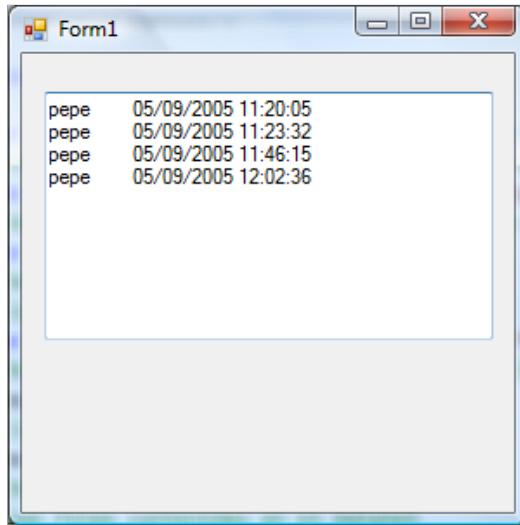
```
Código

Imports System.Data
Imports System.Data.SqlClient

Public Class Form1

    Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.Load
        ' Establecemos la cadena de conexión con la BBDD
        Dim Conexion As String = "server=.;uid=sa;password=VisualBasic;database=MSDNVideo"
        ' Declaramos el DataSet donde volcaremos los datos
        Dim MiDataSet As New DataSet()
        ' Declaramos el DataAdapter estableciendo
        ' la conexión con la fuente de datos
        Dim Comando As New SqlCommand("SELECT SocioNIF, FechaAlquiler FROM ALQUILERES", Conexion)
        ' Rellenamos el DataSet con el contenido de la instrucción SQL
        Comando.Fill(MiDataSet)
        ' Cerramos la conexión con la BBDD
        Comando = Nothing
        ' Declaramos la propiedad Row para recorrer
        ' las filas contenidas en el DataSet
        Dim Row As DataRow
        ' Recorremos todas las filas y las tratamos
        For Each Row In MiDataSet.Tables(0).Rows
            TextBox1.Text += Row("Socio").ToString() & vbTab & Row("FechaAlquiler").ToString() & vbCrLf
        Next
        ' Vaciamos el DataSet para liberar memoria
        MiDataSet = Nothing
    End Sub
End Class
```

El ejemplo en ejecución del uso de *DataAdapter* junto con las clases de .NET es el que se muestra en la figura 3.



Ejemplo del acceso a datos con DataAdapter a través de las clases de .NET

Figura 3

Utilizando los componentes de .NET

Sin embargo y como ya hemos comentado, existe otro método de acceso a fuentes de datos diferente a las clases de .NET, el acceso a través de componentes que nos faciliten esa tarea.

Si embargo, los componentes de acceso a datos, utilizan por detrás las clases de .NET que hemos visto, lo que ocurre, es que simplifica enormemente el trabajo y ahorra tiempo a la hora de desarrollar aplicaciones.

De todos los modos, todo depende de la utilidad o necesidades con las que nos encontramos en un momento dado.

Iniciaremos un proyecto Windows nuevamente, e insertaremos en él un control *TextBox* como hicimos en el caso anterior.

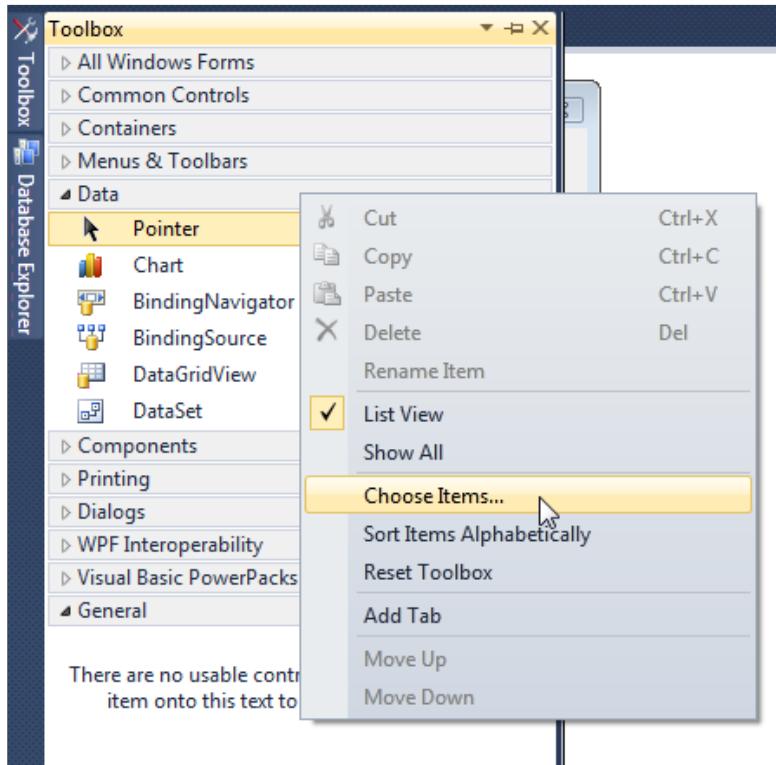
A continuación, añadiremos los componentes de acceso a fuentes de datos SQL Server que es la fuente de datos origen.

Como hemos visto, para conectar a fuentes de datos SQL Server, hemos utilizado el nombre de espacio *System.Data* y hemos importado en el proyecto los nombres de espacio *System.Data* y *System.Data.SqlClient*.

Los componentes .NET de acceso a fuentes de datos de SQL Server, se identifican por el nombre *Sqlxxx*, siendo xxx el tipo de componente a utilizar.

Para poder utilizarlos, deberemos añadirlo a la barra de herramientas.

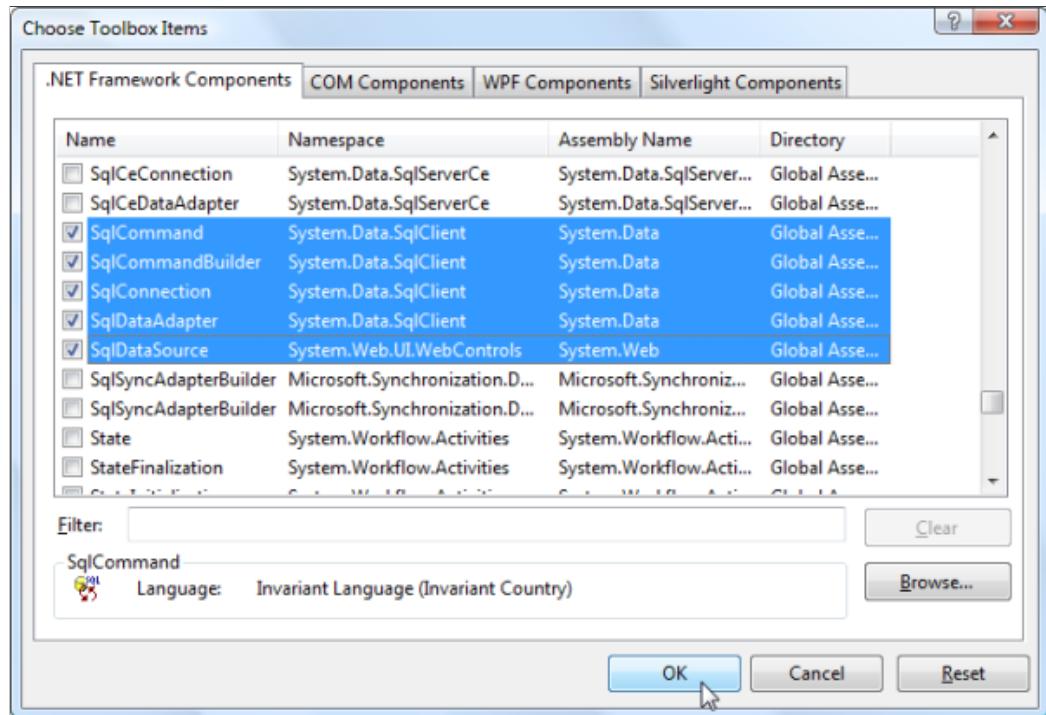
Para añadir los componentes a nuestro proyecto, haremos sobre clic sobre el formulario y posteriormente haremos clic con el botón secundario del mouse sobre la barra de herramientas y seleccionaremos la opción *Elegir elementos...* como se muestra en la figura 4.



Opción de la barra de herramientas para añadir componentes al entorno

Figura 4

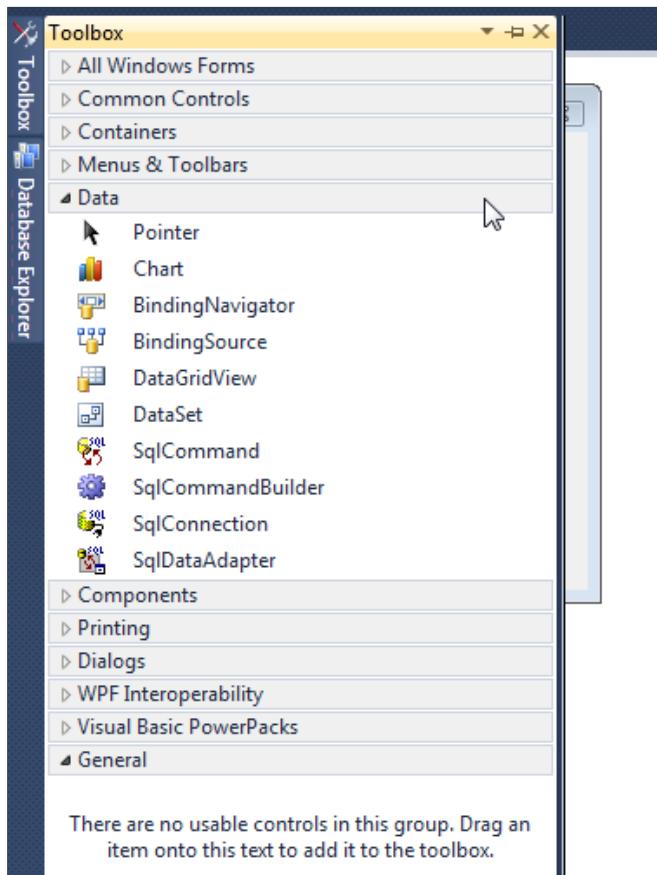
Una vez que hemos hecho esto, seleccionaremos los componentes *SqlCommand*, *SqlCommandBuilder*, *SqlConnection*, *SqlDataAdapter* y *SqlDataSource*, tal y como se muestra en la figura 5.



Componentes a añadir al entorno

Figura 5

Una vez que hemos añadido los componentes al entorno, estos quedarán dispuestos dentro de la barra de herramientas como se indica en la figura 6.



Componentes añadidos en la barra de herramientas

Figura 6

Lo primero que haremos será insertar un componente *SqlConnection* dentro del formulario.

Acudiremos a la ventana de propiedades del componente y modificaremos la propiedad **ConnectionString** dentro de la cuál escribiremos la instrucción:

server=.;uid=sa;password=VisualBasic;database=MSDNVideo

entendiendo que ésta, es la cadena de conexión válida con nuestra base de datos.

A continuación añadiremos el componente *SqlDataAdapter* a nuestro formulario.

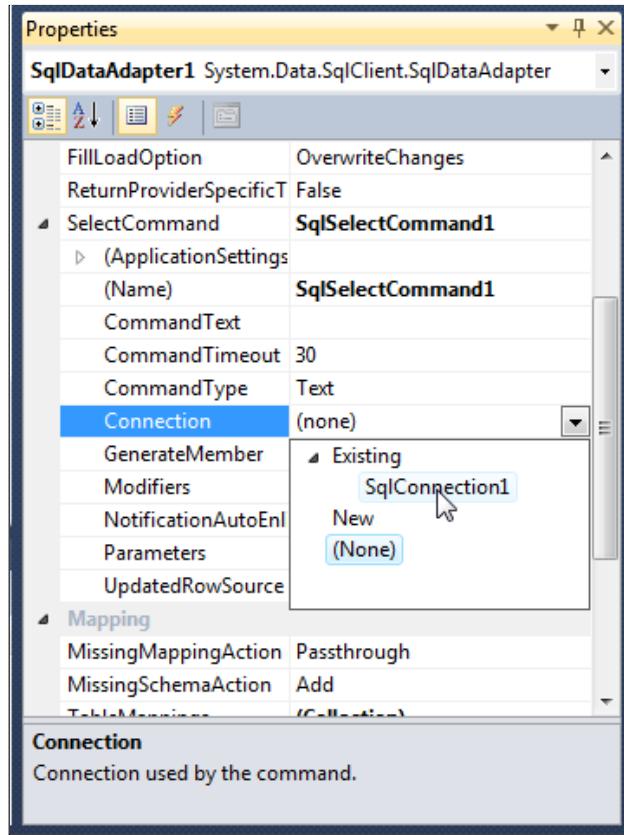
Si se abre alguna ventana, ciérrela.

Vamos a configurar el control con la ventana *Propiedades*. Podríamos haberlo hecho desde el asistente que se nos ha abierto, pero lo vamos a hacer de otra forma *menos sencilla*.

Sitúese sobre la propiedad *SelectCommand*, y dentro de ésta, en la propiedad *Connection*.

Lo que vamos a hacer, es asignar al componente *SqlDataAdapter* el componente de conexión que vamos a usar para establecer la comunicación entre la fuente de datos y nuestra aplicación.

Despliegue la propiedad *Connection* indicada, y seleccione el componente de conexión *SqlConnection1* anteriormente configurado, tal y como se muestra en la figura 7.

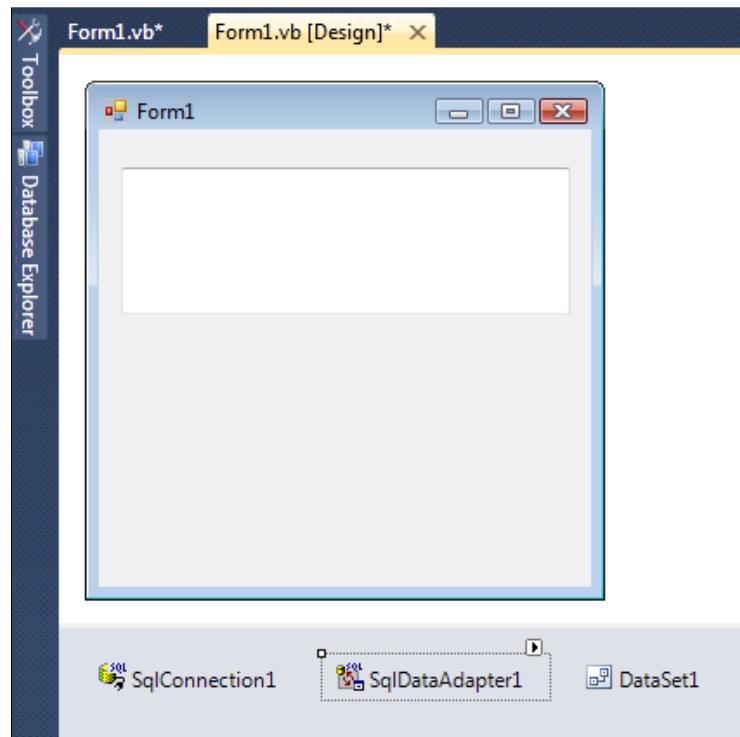


Componente **SqlDataAdapter** con la conexión establecida para ser usada en la ejecución de nuestra aplicación

Figura 7

Por último, inserte un componente *DataSet* al formulario.

Todos los componentes quedarán por lo tanto insertados, tal y como se indica en la figura 8.



Componentes añadidos en el formulario de nuestra aplicación

Figura 8

Una vez que tenemos todo preparado, tan sólo nos queda escribir la parte de código fuente necesario para poder realizar todas las operaciones y acciones que necesitamos.

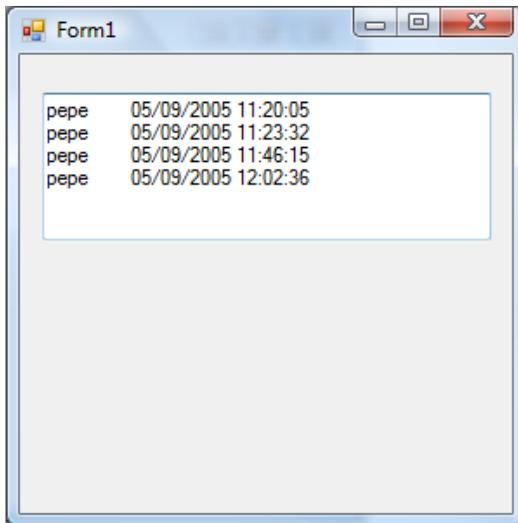
A continuación, se expone el código fuente de nuestra aplicación de demostración:

Código

```
Public Class Form1

    Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.Load
        ' Establecemos la cadena SQL a utilizar
        SqlDataAdapter1.SelectCommand.CommandText = "SELECT SocioNIF, FechaAlquiler FROM ALQUILERES"
        ' Abrimos la Conexión
        SqlConnection1.Open()
        ' Rellenamos el DataSet con el contenido de la instrucción SQL
        SqlDataAdapter1.Fill(DataSet1)
        ' Cerramos la Conexión
        SqlConnection1.Close()
        ' Declaramos la propiedad Row para recorrer
        ' las filas contenidas en el DataSet
        Dim Row As DataRow
        ' Recorremos todas las filas y las tratamos
        For Each Row In DataSet1.Tables(0).Rows
            TextBox1.Text += Row("SocioNIF").ToString() & vbTab & Row("FechaAlquiler").ToString() & vbCrLf
        Next
    End Sub
End Class
```

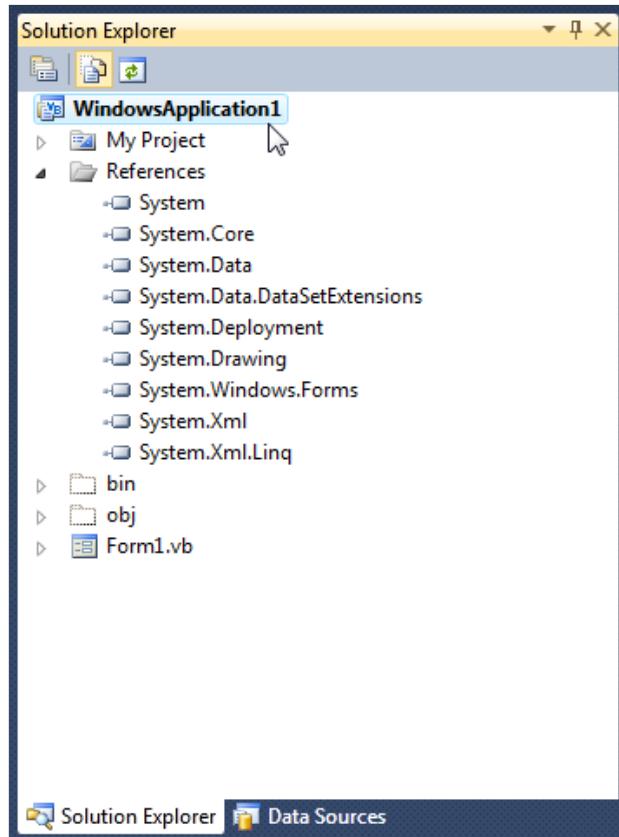
Ahora nos queda únicamente ejecutar nuestra aplicación para estudiar el resultado final. Este es el que se puede ver en la figura 9.



Ejemplo en ejecución del uso de componentes

Figura 9

Pese a todo lo que hemos visto, quizás se pregunte como es que en el caso del primer ejemplo que hemos visto y en el que hemos declarado el uso de un *DataAdapter* sin usar componentes, hemos tenido la obligatoriedad de añadir las referencias a los nombres de espacio *System.Data* y *System.Xml*, mientras que en este segundo ejemplo, no hemos hecho referencia a ellos. En realidad nosotros no hemos hecho referencia a ellos, pero al insertar los componentes dentro del formulario, el entorno Visual Studio 2010 se ha encargado por nosotros de añadir esas referencias al proyecto, tal y como puede verse en la figura 10.



Referencias añadidas automáticamente al trabajar con componentes de acceso a datos

Figura 10

- Esquema general de la estructura desconectada de acceso a datos
 - Conociendo el objeto DataAdapter
- Insertando datos a través del objeto DataAdapter**
- Actualizando datos a través del objeto DataAdapter
 - Eliminando datos a través del objeto DataAdapter

Módulo 5 - Capítulo 3

3. Insertando datos a través del objeto DataAdapter

Hasta ahora, todos los ejemplos que hemos visto del objeto *DataAdapter*, han sido ejemplos del uso de selección de datos, pero aún no hemos visto como debemos trabajar cuando realicemos otras acciones sobre la fuente de datos.

A continuación, veremos como realizar acciones de actualización de datos, utilizando para ello el objeto *DataAdapter*.

Recuerde:

El DataSet permanece desconectado de la fuente de datos y si realizamos una modificación o alteración de los datos de un DataSet, estos no son propagados a la fuente de datos. Para ello, el DataAdapter debe recibir la orden que queramos ejecutar.

¿Cómo se insertan datos con el objeto DataAdapter

Suponiendo que hemos recogido un conjunto de datos y que trabajando con el objeto *DataSet* hemos realizado una inserción de datos y que a continuación, queremos propagar dicha inserción a la base de datos, deberemos hacer uso del método *Insert* del objeto *DataAdapter*.

El objeto *DataAdapter* se encargará de llamar al comando apropiado para cada una de las filas que han sido modificadas en un determinado *DataSet*.
Esto lo realizará siempre a través del método *Update*.

Trabajando con un ejemplo

La mejor manera de ver esto es con un ejemplo que nos ayude a entender mejor como funciona la inserción de datos a través del objeto *DataAdapter*. Tenga en cuenta además, que la actualización y el borrado de datos funciona de la misma manera.

Iniciaremos un nuevo proyecto de formulario Windows y en él insertaremos los componentes *SqlConnection*, *SqlDataAdapter*, *DataSet* y *SqlCommand*.

Para el componente *SqlConnection*, estableceremos la propiedad *ConnectionString* con el valor:
server=.;uid=sa;password=VisualBasic;database=MSDNVideo

A continuación seleccionaremos el componente *SqlDataAdapter* y modificaremos la propiedad *SelectCommand > Connection* como vimos en el capítulo anterior.
De la lista de posibles conexiones que le aparezca, seleccione la conexión *SqlConnection1*.

Finalmente, inserte un control *Button* y un control *DataGridView* en el formulario.
Este quedará como se indica en la figura 1.

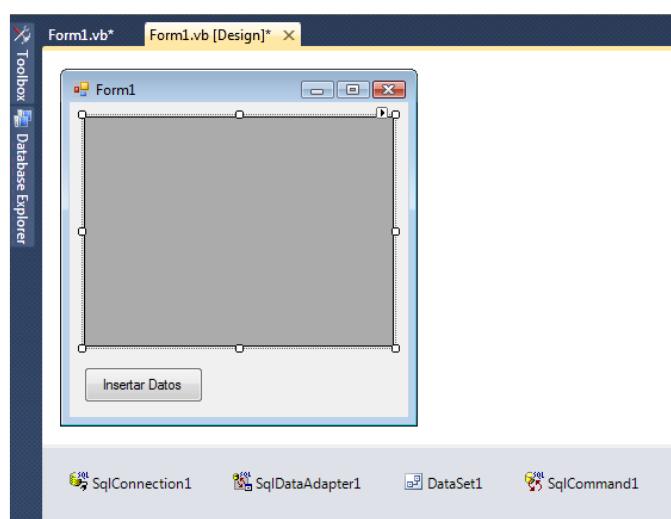


Figura 1

Finalmente, escribiremos el código necesario para ejecutar nuestra aplicación tal y como queremos. Este es el que se detalla a continuación:

Código

```
Public Class Form1

    Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.Load
        ' Establecemos la cadena SQL a utilizar
        SqlDataAdapter1.SelectCommand.CommandText = "SELECT NIF, Nombre, Apellido1, Apellido2, Telefono, Email, Direccion, Ciudad, Provincia, CP FROM SOCIOS"
        ' Abrimos la Conexión
        SqlConnection1.Open()
        ' Rellenamos el DataSet con el contenido de la instrucción SQL
        SqlDataAdapter1.Fill(DataSet1, "Ejemplo")
        ' Cerramos la Conexión
    End Sub
End Class
```

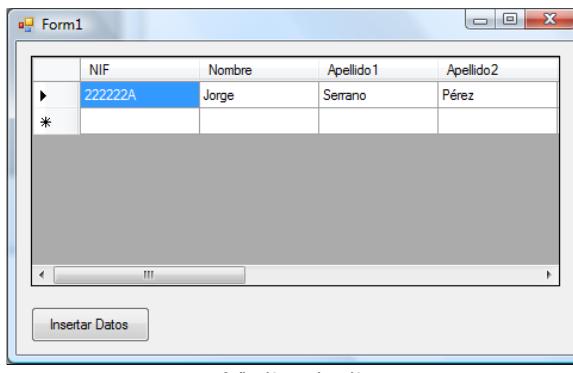
```

SqlConnection1.Close()
' Asociamos el control DataGridView al DataSet
DataGridView1.DataSource = DataSet1.Tables("Ejemplo")
End Sub

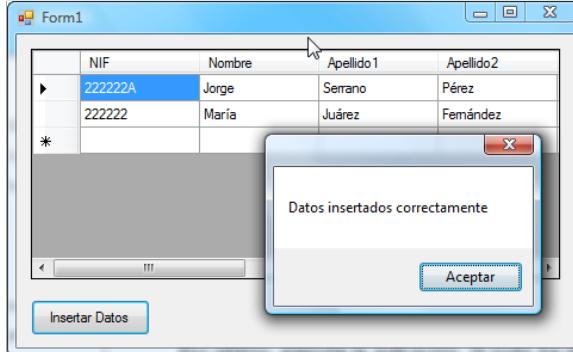
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button1.Click
    ' Declaramos un objeto DataRow para
    ' insertar en él los nuevos datos
    Dim MiDataRow As DataRow
    ' Creamos una nueva fila en el DataSet
    MiDataRow = DataSet1.Tables("Ejemplo").NewRow()
    ' Insertamos los datos en el DataSet
    MiDataRow("NIF") = "222222"
    MiDataRow("Nombre") = "María"
    MiDataRow("Apellido1") = "Juárez"
    MiDataRow("Apellido2") = "Fernández"
    MiDataRow("Telefono") = "1112333"
    MiDataRow("Email") = "maria@cuantademail.com"
    MiDataRow("Direccion") = "C\ Fernández de los Ríos, 77"
    MiDataRow("Ciudad") = "Valladolid"
    MiDataRow("Provincia") = "Valladolid"
    MiDataRow("CP") = "11111"
    DataSet1.Tables("Ejemplo").Rows.Add(MiDataRow)
    ' Si el DataSet tiene cambios ?
    If DataSet1.HasChanges Then
        ' Indicamos la instrucción SQL correspondiente
        SqlCommand1.CommandText = "INSERT INTO SOCIOS(NIF, Nombre, Apellido1, Apellido2, Telefono, Email, Direccion, Ciudad, Provincia, CP) VALUES(@NIF, @Nombre, @Apellido1, @Apellido2, @Telefono, @Email, @Direccion, @Ciudad, @Provincia, @CP)"
        ' Establecemos para el comando,
        ' la (conexión) que utilizaremos
        SqlCommand1.Connection = SqlConnection1
        ' Le indicamos al DataAdapter, cuál es el
        ' comando de inserción que usaremos
        SqlDataAdapter1.InsertCommand = SqlCommand1
        ' Añadimos los parámetros y comandos correspondientes
        ' para cada campo a añadir en la base de datos
        SqlCommand1.Parameters.Add("@NIF", Data.SqlDbType.NChar, 10, "NIF")
        SqlCommand1.Parameters.Add("@Nombre", Data.SqlDbType.NVarChar, 50, "Nombre")
        SqlCommand1.Parameters.Add("@Apellido1", Data.SqlDbType.NVarChar, 50, "Apellido1")
        SqlCommand1.Parameters.Add("@Apellido2", Data.SqlDbType.NVarChar, 50, "Apellido2")
        SqlCommand1.Parameters.Add("@Telefono", Data.SqlDbType.NVarChar, 13, "Telefono")
        SqlCommand1.Parameters.Add("@Email", Data.SqlDbType.NVarChar, 50, "Email")
        SqlCommand1.Parameters.Add("@Direccion", Data.SqlDbType.NVarChar, 100, "Direccion")
        SqlCommand1.Parameters.Add("@Ciudad", Data.SqlDbType.NVarChar, 50, "Ciudad")
        SqlCommand1.Parameters.Add("@Provincia", Data.SqlDbType.NVarChar, 50, "Provincia")
        SqlCommand1.Parameters.Add("@CP", Data.SqlDbType.NChar, 5, "CP")
        ' Abrimos la conexión
        SqlConnection1.Open()
        ' Realizamos la inserción de datos desde el DataSet
        ' a través del DataAdapter
        SqlDataAdapter1.Update(DataSet1, "Ejemplo")
        ' Cerramos la conexión
        SqlCommand1.Close()
        ' Indicamos con un mensaje que la inserción
        ' de datos se ha realizado con éxito
        MessageBox.Show("Datos insertados correctamente")
    End If
End Sub
End Class

```

Por último, ejecute la aplicación. Si todo ha ido correctamente, los datos habrán quedado correctamente insertados en la base de datos. Un ejemplo de nuestra aplicación en ejecución es la que puede verse en la figura 2.



Aplicación en ejecución
Figura 2



Aplicación de ejemplo de inserción de datos con DataAdapter y DataSet en ejecución
Figura 2

Como vemos, el uso de *DataAdapter* en el caso de manipular datos, varía ligeramente. Sin embargo, no es mucho más complicado con la actualización y borrado de datos.

De hecho, la forma de actuar es la misma como veremos a continuación.

Lección 3: Acceso desconectado: DataSets y DataAdapters

- Esquema general de la estructura desconectada de acceso a datos
- Conociendo el objeto DataAdapter
- Insertando datos a través del objeto DataAdapter
- Actualizando datos a través del objeto DataAdapter
- Eliminando datos a través del objeto DataAdapter

Módulo 5 - Capítulo 3

4. Actualizando datos a través del objeto DataAdapter

De la misma manera que hemos insertado datos en nuestra base de datos, debemos hacer a la hora de actualizar los mismos.

Sobre la base del ejemplo anterior (componentes y controles), escriba o modifique el siguiente código:

Código

```
Public Class Form1

    Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.Load
        ' Establecemos la cadena SQL a utilizar
        SqlDataAdapter1.SelectCommand.CommandText = "SELECT NIF, Nombre, Apellido1, Apellido2, Direccion, Telefono FROM SOCIOS"
        ' Abrimos la Conexión
        SqlConnection1.Open()
        ' Rellenamos el DataSet con el contenido de la instrucción SQL
        SqlDataAdapter1.Fill(DataSet1, "Ejemplo")
        ' Cerramos la Conexión
        SqlConnection1.Close()
        ' Asociamos el control DataGridView al DataSet
        DataGridView1.DataSource = DataSet1.Tables("Ejemplo")
    End Sub

    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button1.Click
        ' En nuestro ejemplo, sabemos que queremos modificar
        ' la fila 2, columna 4 (todos los elementos empiezan por 0)
        DataSet1.Tables("Ejemplo").Rows(1)(4) = "1112234"
        ' Si el DataSet tiene cambios ?
        If DataSet1.HasChanges Then
            ' Indicamos la instrucción SQL correspondiente
            SqlCommand1.CommandText = "UPDATE SOCIOS SET Telefono=@Telefono WHERE NIF=@NIF"
            ' Establecemos para el comando,
            ' la (conexión) que utilizaremos
            SqlCommand1.Connection = SqlConnection1
            ' Le indicamos al DataAdapter, cuál es el
            ' comando de actualización que usaremos
            SqlDataAdapter1.UpdateCommand = SqlCommand1
            ' Añadimos los parámetros y comandos correspondientes
            ' para cada campo a actualizar en la base de datos
            SqlCommand1.Parameters.Add("@NIF", Data.SqlDbType.NChar, 10, "NIF")
            SqlCommand1.Parameters.Add("@Nombre", Data.SqlDbType.NVarChar, 50, "Nombre")
        End If
    End Sub

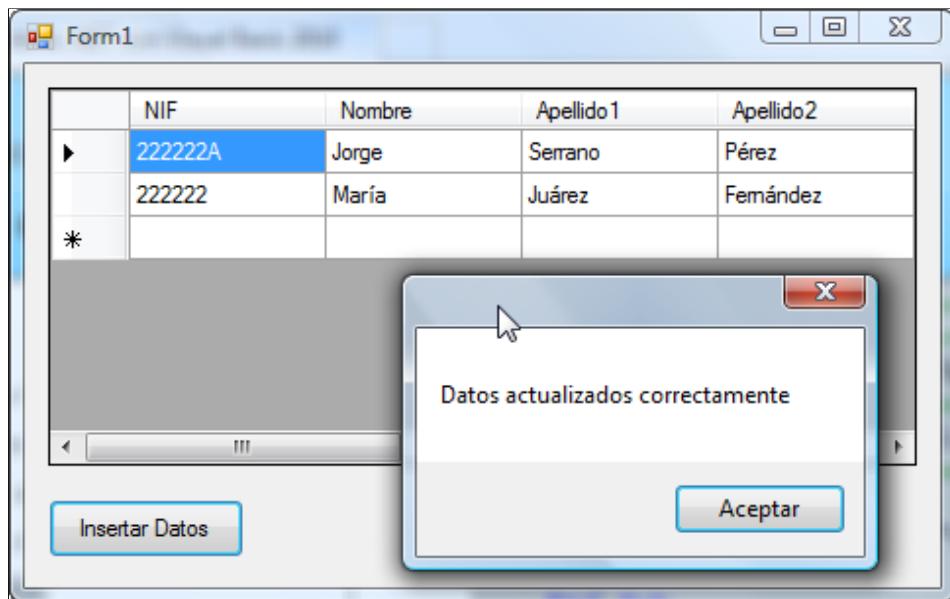
```

```

SqlCommand1.Parameters.Add("@Apellido1", Data.SqlDbType.NVarChar, 50, "Apellido1")
SqlCommand1.Parameters.Add("@Apellido2", Data.SqlDbType.NVarChar, 50, "Apellido2")
SqlCommand1.Parameters.Add("@Telefono", Data.SqlDbType.NVarChar, 13, "Telefono")
SqlCommand1.Parameters.Add("@Email", Data.SqlDbType.NVarChar, 50, "Email")
SqlCommand1.Parameters.Add("@Direccion", Data.SqlDbType.NVarChar, 100, "Direccion")
SqlCommand1.Parameters.Add("@Ciudad", Data.SqlDbType.NVarChar, 50, "Ciudad")
SqlCommand1.Parameters.Add("@Provincia", Data.SqlDbType.NVarChar, 50, "Provincia")
SqlCommand1.Parameters.Add("@CP", Data.SqlDbType.NChar, 5, "CP")
' Abrimos la conexión
SqlConnection1.Open()
' Realizamos la actualización de datos desde
' el DataSet a través del DataAdapter
SqlDataAdapter1.Update(DataSet1, "Ejemplo")
' Cerramos la conexión
SqlConnection1.Close()
' Indicamos con un mensaje que la actualización
' de datos se ha realizado con éxito
MessageBox.Show("Datos actualizados correctamente")
End If
End Sub
End Class

```

Nuestro ejemplo en ejecución es el que se puede ver en la figura 1.



Aplicación de ejemplo de actualización de datos con DataAdapter y DataSet

Figura 1

Lección 3: Acceso desconectado: DataSets y DataAdapters

- Esquema general de la estructura desconectada de acceso a datos
 - Conociendo el objeto DataAdapter
 - Insertando datos a través del objeto DataAdapter
 - Actualizando datos a través del objeto DataAdapter
- Eliminando datos a través del objeto DataAdapter**

Módulo 5 - Capítulo 3

5. Eliminando datos a través del objeto DataAdapter

De igual forma sucede con la eliminación de datos utilizando para ello el objeto *DataAdapter* junto al objeto *DataSet*.

Utilizaremos nuevamente en este caso, la base del ejemplo anterior (componentes y controles), y escribiremos el siguiente código:

Código

```
Public Class Form1

    Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.Load
        ' Establecemos la cadena SQL a utilizar
        SqlDataAdapter1.SelectCommand.CommandText = "SELECT NIF, Nombre, Apellido1, Apellido2,
        ' Abrimos la Conexión
        SqlConnection1.Open()
        ' Rellenamos el DataSet con el contenido de la instrucción SQL
        SqlDataAdapter1.Fill(DataSet1, "Ejemplo")
        ' Cerramos la Conexión
        SqlConnection1.Close()
        ' Asociamos el control DataGridView al DataSet
        DataGridView1.DataSource = DataSet1.Tables("Ejemplo")
    End Sub

    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button1.Click
        ' En nuestro ejemplo, sabemos que queremos eliminar
        ' la fila 2 (todos los elementos empiezan por 0)
        ' por lo que la fila 2 es aquí la 1
        DataSet1.Tables("Ejemplo").Rows(1).Delete()
        ' Si el DataSet tiene cambios ?
        If DataSet1.HasChanges Then
            ' Indicamos la instrucción SQL correspondiente
            SqlCommand1.CommandText = "DELETE SOCIOS WHERE NIF=@NIF"
            ' Establecemos para el comando,
            ' la (conexión) que utilizaremos
            SqlCommand1.Connection = SqlConnection1
            ' Le indicamos al DataAdapter, cuál es el
            ' comando de eliminación que usaremos
            SqlDataAdapter1.DeleteCommand = SqlCommand1
            ' Añadimos los parámetros y comandos correspondientes
            ' para cada campo a actualizar en la base de datos
        End If
    End Sub
End Class
```

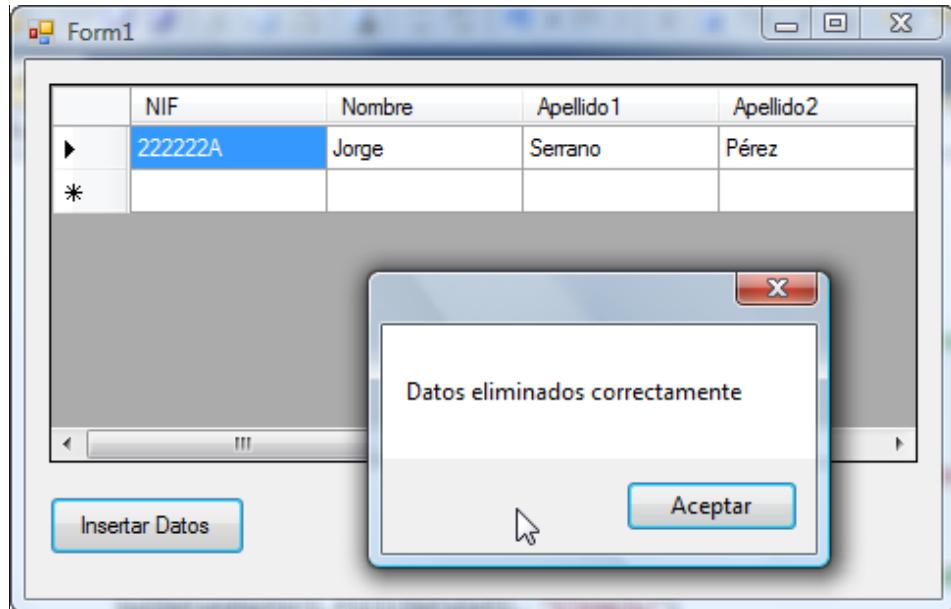
```

SqlCommand1.Parameters.Add("@NIF", Data.SqlDbType.NChar, 10, "NIF")
SqlCommand1.Parameters.Add("@Nombre", Data.SqlDbType.NVarChar, 50, "Nombre")
SqlCommand1.Parameters.Add("@Apellido1", Data.SqlDbType.NVarChar, 50, "Apellido1")
SqlCommand1.Parameters.Add("@Apellido2", Data.SqlDbType.NVarChar, 50, "Apellido2")
SqlCommand1.Parameters.Add("@Telefono", Data.SqlDbType.NVarChar, 13, "Telefono")
SqlCommand1.Parameters.Add("@Email", Data.SqlDbType.NVarChar, 50, "Email")
SqlCommand1.Parameters.Add("@Direccion", Data.SqlDbType.NVarChar, 100, "Direccion")
SqlCommand1.Parameters.Add("@Ciudad", Data.SqlDbType.NVarChar, 50, "Ciudad")
SqlCommand1.Parameters.Add("@Provincia", Data.SqlDbType.NVarChar, 50, "Provincia")
SqlCommand1.Parameters.Add("@CP", Data.SqlDbType.NChar, 5, "CP")

' Abrimos la conexión
SqlConnection1.Open()
' Realizamos la eliminación de datos desde
' el DataSet a través del DataAdapter
SqlDataAdapter1.Update(DataSet1, "Ejemplo")
' Cerramos la conexión
SqlConnection1.Close()
' Indicamos con un mensaje que la eliminación
' de datos se ha realizado con éxito
MessageBox.Show("Datos eliminados correctamente")
End If
End Sub
End Class

```

Nuestro ejemplo en ejecución es el que se puede ver en la figura 1.



Aplicación de ejemplo de eliminación de datos con DataAdapter y DataSet

Figura 1

Lección 4: DataSet tipados

- ¿Qué son los DataSets tipados?
- Generando nuestros DataSets tipados
- Generando un DataSet tipado con Visual Studio
- Generando un DataSet tipado con la línea de comandos
- Usando los DataSets tipados

Introducción

Otra particularidad de .NET a la hora de trabajar con fuentes de datos y con los componentes y controles que nos permiten acceder a ellas, es el trabajo con dos tipos de *DataSets*.

Sin quererlo ya hemos visto como trabajar con uno de ellos, me refiero a los *DataSets no tipados*, sin embargo, hay otro tipo de *DataSet* diferente denominado así, *DataSet tipado*.

En qué consiste un *DataSet tipado* y como utilizarlos, es lo que vamos a ver a continuación.

Módulo 5 - Capítulo 4

- 1. [¿Qué son los DataSets tipados?](#)
- 2. [Generando nuestros DataSets tipados](#)
- 3. [Generando un DataSet tipado con Visual Studio](#)
- 4. [Generando un DataSet tipado con la línea de comandos](#)
- 5. [Usando los DataSets tipados](#)

Lección 4: DataSet tipados

¿Qué son los DataSets tipados?

- Generando nuestros DataSets tipados
- Generando un DataSet tipado con Visual Studio
- Generando un DataSet tipado con la línea de comandos
- Usando los DataSets tipados

Módulo 5 - Capítulo 4

1. ¿Qué son los DataSets tipados?

De forma genérica, podemos definir como *DataSet tipado*, a aquel *DataSet* que posee un esquema de datos, a diferencia del *DataSet no tipado*, que no necesita de ese esquema.

Respecto a los *DataSet tipados*, diremos que en el entorno de desarrollo, encontramos muchas utilidades y herramientas que nos facilitan el trabajo de este tipo de *DataSets*.

Inclusive, el propio SDK posee herramientas de comandos que nos permite y nos facilita la preparación para trabajar con *DataSets tipados*.

El esquema al que un *DataSet tipado* hace referencia, es un documento XML.

Se trata de un documento XML con extensión *.xsd*.

Trabajar con un esquema en lugar de trabajar con una tabla directamente, es mucho más ágil, fácil, rápido y seguro, como veremos más adelante.

Cómo trabajar con un DataSet tipado

Evidentemente, lo que tenemos que tener claro y tomarlo así, como punto de partida, es que si queremos trabajar con un *DataSet tipado*, tenemos que crear su correspondiente esquema XML.

Esto lo podemos hacer con una herramienta externa, manualmente, o con las herramientas que el entorno de desarrollo de Visual Studio 2010 o el propio SDK nos ofrece.

En nuestro caso, será estas últimas acciones lo que usaremos. Obviamente, hacerlo manualmente es en mi opinión para *frikis*, máxime cuando sabemos que tenemos herramientas que nos facilitan su creación y nos ahoran mucho tiempo y recursos.

Cuando tengamos ya creado el esquema XML, deberíamos generar la clase del *DataSet*, algo que a estas alturas, ya lo tenemos dominado.

En uno de los métodos de creación del esquema, el entorno hace ese trabajo por nosotros, en el otro, que es un proceso más manual como veremos, deberíamos crear esa clase con posterioridad, pero aún así, esto último no lo veremos con excesiva profundidad.

¿Qué ventajas nos aportan los DataSets tipados?

Ya hemos enumerado algunas de ellas, rapidez, seguridad, agilidad, facilidad de trabajo, todo ello aplicable a los datos.

Aún así, existen muchas más razones para interesarnos por los *DataSets tipados*.

A través de los *DataSets tipados*, podemos realizar acciones que comúnmente son costosas, de manera rápida y sencilla.

Acciones como actualización de datos, modificación, inserción o eliminación de datos, o búsquedas de datos, son algunas de las acciones que en apenas un par de líneas de código, estarán resueltas gracias al uso de *DataSet tipados*.

Esto lo veremos más adelante con los ejemplos de este capítulo.

Otras acciones adicionales vinculadas con los *DataSets tipados* son la posibilidad de filtrar datos, ordenar los datos, y trabajar con datos jerárquicos y relaciones de datos.

Una diferencia notable entre los *DataSets no tipados* y los *DataSets tipados*, es que los primeros no saben ni tienen conocimiento alguno de lo que almacenan.

Los segundos tienen cierta dosis de *inteligencia* y conocen el tipo de datos que almacena dentro.

Además de todo esto, el acceso a los datos que guarda un *DataSet tipado*, así como la manipulación de los mismos, es mucho más simple y requiere menos código, haciendo que el acceso a los datos y el mantenimiento de la aplicación sea más cómoda y sencilla.



Lección 4: DataSet tipados

- ¿Qué son los DataSets tipados?

Generando nuestros DataSets tipados

- Generando un DataSet tipado con Visual Studio
- Generando un DataSet tipado con la línea de comandos
- Usando los DataSets tipados

Módulo 5 - Capítulo 4

2. Generando nuestros DataSets tipados

Dentro de Visual Studio 2010 y de .NET en general, tenemos varias formas de generar nuestros propios *DataSets tipados*.

Una de las formas de generar *DataSets tipados* es utilizando el entorno de desarrollo rápido Visual Studio 2010.

La otra es utilizando la herramienta **XSD.exe** que encontraremos en el directorio **SDK** del entorno.

Ambas formas, las veremos a continuación.

Diagrama de datos

Antes de continuar, repasaremos el diagrama de datos con el cuál vamos a trabajar. Este es el que se muestra en la figura 1.

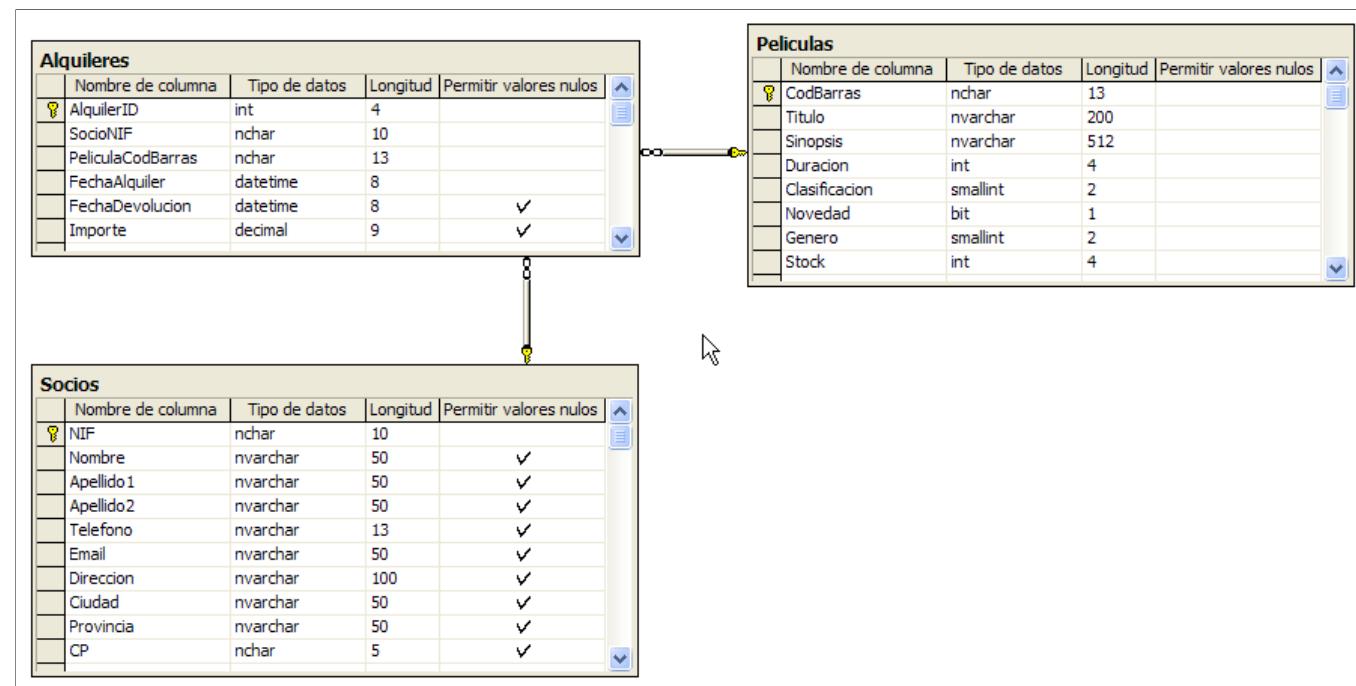


Diagrama de datos con el que vamos a trabajar

Figura 1

Este diagrama nos ayudará a interpretar los datos y a trabajar con ellos, en los ejemplos que veremos a continuación.

A continuación, veremos como generar *DataSets tipados* desde el entorno de trabajo rápido, es decir *Visual Studio 2010*, y desde la línea de comandos.

Lección 4: DataSet tipados

- ¿Qué son los DataSets tipados?
 - Generando nuestros DataSets tipados
- Generando un DataSet tipado con Visual Studio**
- Generando un DataSet tipado con la línea de comandos
 - Usando los DataSets tipados

Módulo 5 - Capítulo 4

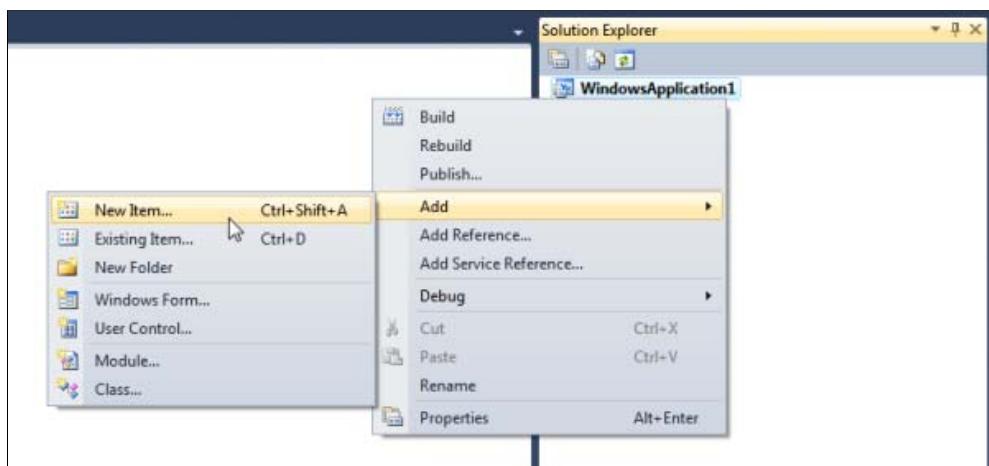
3. Generando un DataSet tipado con Visual Studio 2010

La forma más rápida para generar *DataSets tipados* es utilizando las herramientas automáticas del entorno Visual Studio 2010.

A continuación veremos como hacer esto de manera rápida y sencilla.

Usando el entorno de desarrollo rápido Visual Studio 2010

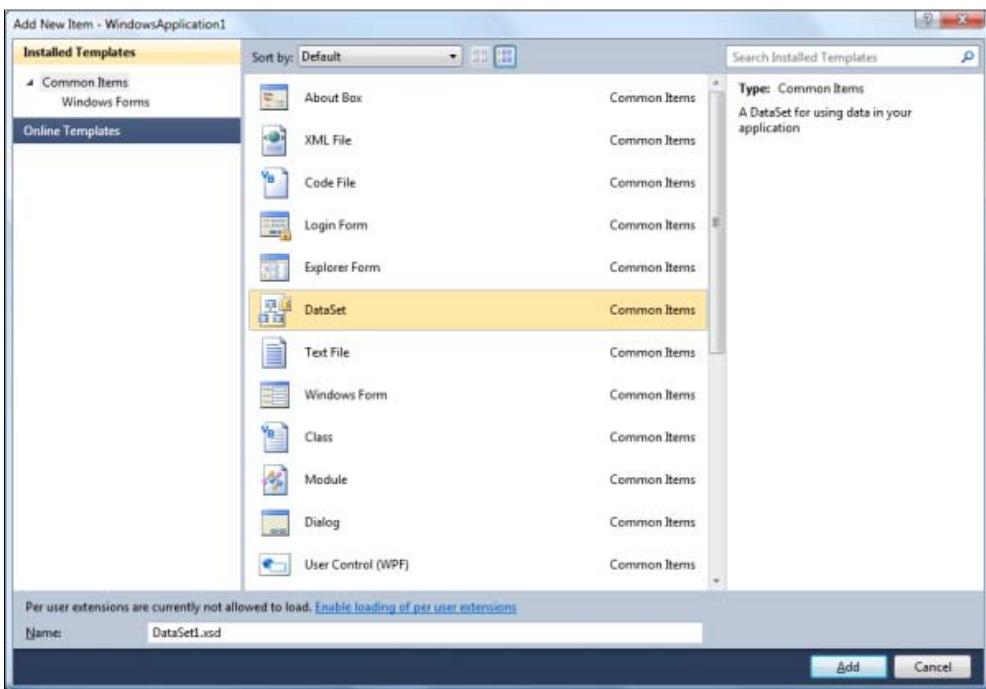
Cree un nuevo proyecto de *Aplicación para Windows* y haga clic con el botón secundario del mouse sobre la ventana *Explorador de soluciones*, y del menú emergente, seleccione la opción **Agregar > Nuevo elemento...**, tal y como se muestra en la figura 1.



Menú para agregar un elemento al proyecto

Figura 1

Aparecerá entonces una ventana de *Agregar nuevo elemento*, dentro de la cuál seleccionaremos la plantilla de **Conjunto de datos**, tal y como se muestra en la figura 2.



Ventana para agregar un nuevo elemento al proyecto

Figura 2

Haga clic en el botón **Agregar**.

La plantilla del esquema del *DataSet*, tendrá un aspecto similar al que se muestra en la figura 3.

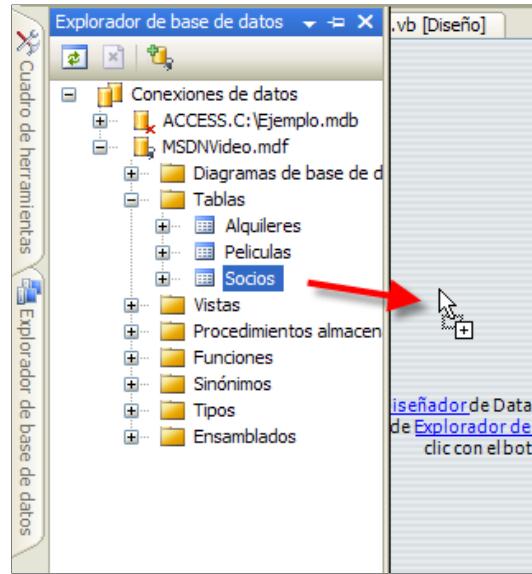


Esquema por defecto del *DataSet* en Visual Studio 2010

Figura 3

El siguiente paso que deberemos abordar, será la de añadir al esquema, las tablas que queremos que formen parte del *DataSet* tipado. Para hacer esto, abra la ventana **Explorador de base de datos** y seleccione la base de datos de prueba.

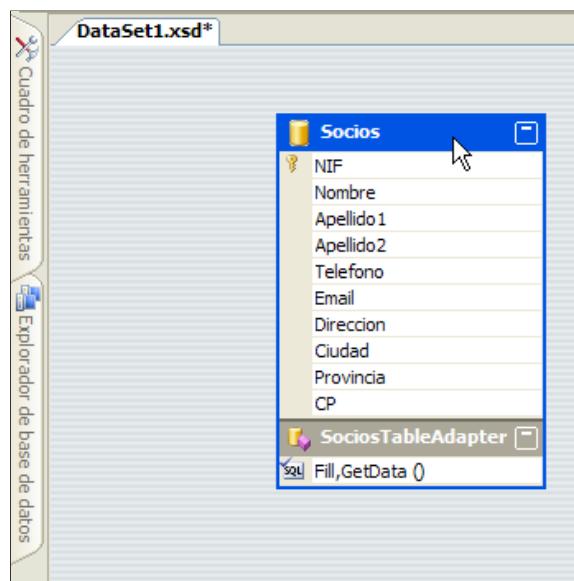
Pulse sobre la tabla **Socios** y arrástrela sobre el esquema del *DataSet* como se indica en la figura 4.



Arrastramos la tabla Socios sobre el esquema del DataSet

Figura 4

El esquema de la tabla quedará entonces añadido a la aplicación, tal y como se indica en la figura 5.



Esquema de la tabla Socios añadido al entorno de trabajo

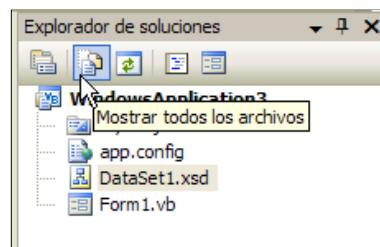
Figura 5

En este punto, tendremos listo nuestro *DataSet tipado*, pero aún no tendremos una clase o código asociado al *DataSet tipado*.

Aún no lo hemos dicho, pero en este punto, *Visual Studio 2010* ha generado para nosotros, el código relacionado con el *DataSet tipado* creado.

Este código, está dentro del directorio de la aplicación, y puede ser accedido a él presionando el botón de la ventana **Explorador de soluciones** y representado por el siguiente icono

Esto es lo que se representa en la figura 6.



Opción de mostrar todos los archivos

Figura 6

En este punto, veremos que los archivos relacionados con nuestro proyecto y en el caso del elemento *DataSet1 xsd*, son los que se detallan en la figura 7.

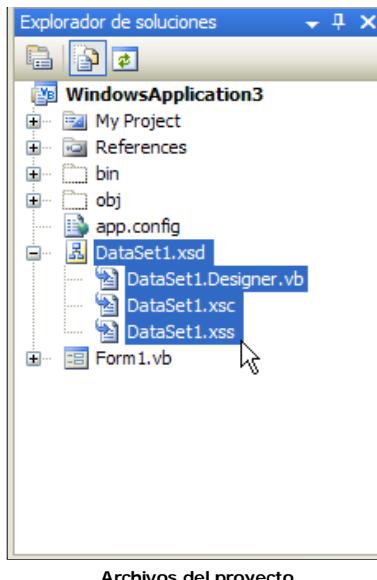


Figura 7

Observamos que el archivo *DataSet1 Designer.cs* depende del archivo *DataSet1 xsd*, y es el que contiene el código necesario para utilizar el *DataSet tipado*.

De hecho, si vemos el código generado por *Visual Studio 2010*, observaremos que coincide con el que se detalla a continuación:

Código

```
'-----
' <auto-generated>
' Una herramienta generó este código.
' Versión del motor en tiempo de ejecución:2.0.50215.44
'
' Los cambios en este archivo podrían causar un comportamiento incorrecto y se perderán si
' el código se vuelve a generar.
' </auto-generated>
'-----
```

```
Option Strict Off
Option Explicit On

Imports System

<Serializable(), _
System.ComponentModel.DesignerCategoryAttribute("code"), _
System.ComponentModel.ToolboxItem(true), _
System.Xml.Serialization.XmlSchemaProviderAttribute("GetTypedDataSetSchema"), _
System.Xml.Serialization.XmlRootAttribute("DataSet1"), _
System.ComponentModel.Design.HelpKeywordAttribute("vs.data.DataSet"), _
System.Diagnostics.CodeAnalysis.SuppressMessageAttribute("Microsoft.Usage", "CA2240:ImplementISerializableCorre
Partial Public Class DataSet1
    Inherits System.Data.DataSet

    Private tableSocios As SociosDataTable

    Private _schemaSerializationMode As System.Data.SchemaSerializationMode = System.Data.SchemaSerializationMode
    <System.Diagnostics.CodeAnalysis.SuppressMessageAttribute("Microsoft.Usage", "CA2214:DoNotCallOverridableMeth
    Public Sub New()
        MyBase.New
        Me.BeginInit
        Me.InitClass
        Dim schemaChangedHandler As System.ComponentModel.CollectionChangeEventHandler = AddressOf Me.SchemaChange
        AddHandler MyBase.Tables.CollectionChanged, schemaChangedHandler
        AddHandler MyBase.Relations.CollectionChanged, schemaChangedHandler
        Me.EndInit
    End Sub

    <System.Diagnostics.CodeAnalysis.SuppressMessageAttribute("Microsoft.Usage", "CA2214:DoNotCallOverridableMeth
    Protected Sub New(ByVal info As System.Runtime.Serialization.SerializationInfo, ByVal context As System.Runti
        MyBase.New(info, context)
        If (Me.IsBinarySerialized(info, context) = true) Then
            Me.InitVars(false)
```

```

        Dim schemaChangedHandler1 As System.ComponentModel.CollectionChangeEventHandler = AddressOf Me.SchemaCh
        AddHandler Me.Tables.CollectionChanged, schemaChangedHandler1
        AddHandler Me.Relations.CollectionChanged, schemaChangedHandler1
        Return
    End If
    Dim strSchema As String = CType(info.GetValue("XmlSchema", GetType(String)), String)
    If (Me.DetermineSchemaSerializationMode(info, context) = System.Data.SchemaSerializationMode.IncludeSchema)
        Dim ds As System.Data.DataSet = New System.Data.DataSet
        ds.ReadXmlSchema(New System.Xml.XmlTextReader(New System.IO.StringReader(strSchema)))
        If (Not (ds.Tables("Socios")) Is Nothing) Then
            MyBase.Tables.Add(New SociosDataTable(ds.Tables("Socios")))
        End If
        Me.DataSetName = ds.DataSetName
        Me.Prefix = ds.Prefix
        Me.Namespace = ds.Namespace
        Me.Locale = ds.Locale
        Me.CaseSensitive = ds.CaseSensitive
        Me.EnforceConstraints = ds.EnforceConstraints
        Me.Merge(ds, false, System.Data.MissingSchemaAction.Add)
        Me.InitVars
    Else
        Me.ReadXmlSchema(New System.Xml.XmlTextReader(New System.IO.StringReader(strSchema)))
    End If
    Me.GetSerializationData(info, context)
    Dim schemaChangedHandler As System.ComponentModel.CollectionChangeEventHandler = AddressOf Me.SchemaChange
    AddHandler MyBase.Tables.CollectionChanged, schemaChangedHandler
    AddHandler Me.Relations.CollectionChanged, schemaChangedHandler
End Sub

<System.ComponentModel.Browsable(false), _
System.ComponentModel.DesignerSerializationVisibility(System.ComponentModel.DesignerSerializationVisibility.
Public ReadOnly Property Socios() As SociosDataTable
    Get
        Return Me.tableSocios
    End Get
End Property

Public Overrides Property SchemaSerializationMode() As System.Data.SchemaSerializationMode
    Get
        Return Me._schemaSerializationMode
    End Get
    Set
        Me._schemaSerializationMode = value
    End Set
End Property

<System.ComponentModel.DesignerSerializationVisibilityAttribute(System.ComponentModel.DesignerSerializationVi
Public Shadows ReadOnly Property Tables() As System.Data.DataTableCollection
    Get
        Return MyBase.Tables
    End Get
End Property

<System.ComponentModel.DesignerSerializationVisibilityAttribute(System.ComponentModel.DesignerSerializationVi
Public Shadows ReadOnly Property Relations() As System.Data.DataRelationCollection
    Get
        Return MyBase.Relations
    End Get
End Property

<System.ComponentModel.DefaultValueAttribute(true)> -
Public Shadows Property EnforceConstraints() As Boolean
    Get
        Return MyBase.EnforceConstraints
    End Get
    Set
        MyBase.EnforceConstraints = value
    End Set
End Property

Protected Overrides Sub InitializeDerivedDataSet()
    Me.BeginInit
    Me.InitClass
    Me.EndInit
End Sub

Public Overrides Function Clone() As System.Data.DataSet
    Dim cln As DataSet1 = CType(MyBase.Clone, DataSet1)
    cln.InitVars
    Return cln
End Function

Protected Overrides Function ShouldSerializeTables() As Boolean
    Return false
End Function

Protected Overrides Function ShouldSerializeRelations() As Boolean

```

```

        Return false
End Function

Protected Overrides Sub ReadXmlSerializable(ByVal reader As System.Xml.XmlReader)
    If (Me.DetermineSchemaSerializationMode(reader) = System.Data.SchemaSerializationMode.IncludeSchema) Then
        Me.Reset
        Dim ds As System.Data.DataSet = New System.Data.DataSet
        ds.ReadXml(reader)
        If (Not (ds.Tables("Socios")) Is Nothing) Then
            MyBase.Tables.Add(New SociosDataTable(ds.Tables("Socios")))
        End If
        Me.DataSetName = ds.DataSetName
        Me.Prefix = ds.Prefix
        Me.Namespace = ds.Namespace
        Me.Locale = ds.Locale
        Me.CaseSensitive = ds.CaseSensitive
        Me.EnforceConstraints = ds.EnforceConstraints
        Me.Merge(ds, false, System.Data.MissingSchemaAction.Add)
        Me.InitVars
    Else
        Me.ReadXml(reader)
        Me.InitVars
    End If
End Sub

Protected Overrides Function GetSchemaSerializable() As System.Xml.Schema.XmlSchema
    Dim stream As System.IO.MemoryStream = New System.IO.MemoryStream
    Me.WriteXmlSchema(New System.Xml.XmlTextWriter(stream, Nothing))
    stream.Position = 0
    Return System.Xml.Schema.XmlSchema.Read(New System.Xml.XmlTextReader(stream), Nothing)
End Function

Friend Overloads Sub InitVars()
    Me.InitVars(true)
End Sub

Friend Overloads Sub InitVars(ByVal initTable As Boolean)
    Me.tableSocios = CType(MyBase.Tables("Socios"), SociosDataTable)
    If (initTable = true) Then
        If (Not (Me.tableSocios) Is Nothing) Then
            Me.tableSocios.InitVars
        End If
    End If
End Sub

Private Sub InitClass()
    Me.DataSetName = "DataSet1"
    Me.Prefix = ""
    Me.Namespace = "http://tempuri.org/DataSet1.xsd"
    Me.EnforceConstraints = true
    Me.tableSocios = New SociosDataTable
    MyBase.Tables.Add(Me.tableSocios)
End Sub

Private Function ShouldSerializeSocios() As Boolean
    Return false
End Function

Private Sub SchemaChanged(ByVal sender As Object, ByVal e As System.ComponentModel.CollectionChangeEventArgs)
    If (e.Action = System.ComponentModel.CollectionChangeAction.Remove) Then
        Me.InitVars
    End If
End Sub

Public Shared Function GetTypedDataSetSchema(ByVal xs As System.Xml.Schema.XmlSchemaSet) As System.Xml.Schema
    Dim ds As DataSet1 = New DataSet1
    Dim type As System.Xml.Schema.XmlSchemaComplexType = New System.Xml.Schema.XmlSchemaComplexType
    Dim sequence As System.Xml.Schema.XmlSchemaSequence = New System.Xml.Schema.XmlSchemaSequence
    xs.Add(ds.GetSchemaSerializable)
    Dim any As System.Xml.Schema.XmlSchemaAny = New System.Xml.Schema.XmlSchemaAny
    any.Namespace = ds.Namespace
    sequence.Items.Add(any)
    type.Particle = sequence
    Return type
End Function

Public Delegate Sub SociosRowChangeEventHandler(ByVal sender As Object, ByVal e As SociosRowChangeEvent)

<System.Serializable(), _
System.Xml.Serialization.XmlSchemaProviderAttribute("GetTypedTableSchema")> _
Partial Public Class SociosDataTable
    Inherits System.Data.DataTable
    Implements System.Collections.IEnumerable

    Private columnNIF As System.Data.DataColumn

    Private columnNameNombre As System.Data.DataColumn

```

```

Private columnApellido1 As System.Data.DataColumn
Private columnApellido2 As System.Data.DataColumn
Private columnTelefono As System.Data.DataColumn
Private columnEmail As System.Data.DataColumn
Private columnDireccion As System.Data.DataColumn
Private columnCiudad As System.Data.DataColumn
Private columnProvincia As System.Data.DataColumn
Private columnCP As System.Data.DataColumn

Public Sub New()
    MyBase.New
    Me.TableName = "Socios"
    Me.BeginInit
    Me.InitClass
    Me.EndInit
End Sub

Friend Sub New(ByVal table As System.Data.DataTable)
    MyBase.New
    Me.TableName = table.TableName
    If (table.CaseSensitive <> table.DataSet.CaseSensitive) Then
        Me.CaseSensitive = table.CaseSensitive
    End If
    If (table.Locale.ToString <> table.DataSet.Locale.ToString) Then
        Me.Locale = table.Locale
    End If
    If (table.Namespace <> table.DataSet.Namespace) Then
        Me.Namespace = table.Namespace
    End If
    Me.Prefix = table.Prefix
    Me.MinimumCapacity = table.MinimumCapacity
End Sub

Protected Sub New(ByVal info As System.Runtime.Serialization.SerializationInfo, ByVal context As System.Ru
    MyBase.New(info, context)
    Me.InitVars
End Sub

Public ReadOnly Property NIFColumn() As System.Data.DataColumn
    Get
        Return Me.columnNIF
    End Get
End Property

Public ReadOnly Property NombreColumn() As System.Data.DataColumn
    Get
        Return Me.columnNombre
    End Get
End Property

Public ReadOnly Property Apellido1Column() As System.Data.DataColumn
    Get
        Return Me.columnApellido1
    End Get
End Property

Public ReadOnly Property Apellido2Column() As System.Data.DataColumn
    Get
        Return Me.columnApellido2
    End Get
End Property

Public ReadOnly Property TelefonoColumn() As System.Data.DataColumn
    Get
        Return Me.columnTelefono
    End Get
End Property

Public ReadOnly Property EmailColumn() As System.Data.DataColumn
    Get
        Return Me.columnEmail
    End Get
End Property

Public ReadOnly Property DireccionColumn() As System.Data.DataColumn
    Get
        Return Me.columnDireccion
    End Get
End Property

```

```

Public ReadOnly Property CiudadColumn() As System.Data.DataColumn
    Get
        Return Me.columnCiudad
    End Get
End Property

Public ReadOnly Property ProvinciaColumn() As System.Data.DataColumn
    Get
        Return Me.columnProvincia
    End Get
End Property

Public ReadOnly Property CPCColumn() As System.Data.DataColumn
    Get
        Return Me.columnCP
    End Get
End Property

<System.ComponentModel.Browsable(false)> _
Public ReadOnly Property Count() As Integer
    Get
        Return Me.Rows.Count
    End Get
End Property

Public Default ReadOnly Property Item(ByVal index As Integer) As SociosRow
    Get
        Return CType(Me.Rows(index),SociosRow)
    End Get
End Property

Public Event SociosRowChanged As SociosRowChangeEventHandler
Public Event SociosRowChanging As SociosRowChangeEventHandler
Public Event SociosRowDeleted As SociosRowChangeEventHandler
Public Event SociosRowDeleting As SociosRowChangeEventHandler

Public Overloads Sub AddSociosRow(ByVal row As SociosRow)
    Me.Rows.Add(row)
End Sub

Public Overloads Function AddSociosRow(ByVal NIF As String, ByVal Nombre As String, ByVal Apellido1 As String, ByVal Apellido2 As String, ByVal Telefono As String, ByVal Email As String, ByVal Direccion As String) As SociosRow
    Dim rowSociosRow As SociosRow = CType(Me.NewRow,SociosRow)
    rowSociosRow.ItemArray = New Object() {NIF, Nombre, Apellido1, Apellido2, Telefono, Email, Direccion, C}
    Me.Rows.Add(rowSociosRow)
    Return rowSociosRow
End Function

Public Function FindByNIF(ByVal NIF As String) As SociosRow
    Return CType(Me.Rows.Find(New Object() {NIF}),SociosRow)
End Function

Public Overrides Function GetEnumerator() As System.Collections.IEnumerator Implements System.Collections.IEnumerable
    Return Me.Rows.GetEnumerator()
End Function

Public Overrides Function Clone() As System.Data.DataTable
    Dim cln As SociosDataTable = CType(MyBase.Clone,SociosDataTable)
    cln.InitVars()
    Return cln
End Function

Protected Overrides Function CreateInstance() As System.Data.DataTable
    Return New SociosDataTable()
End Function

Friend Sub InitVars()
    Me.columnNIF = MyBase.Columns("NIF")
    Me.columnNombre = MyBase.Columns("Nombre")
    Me.columnApellido1 = MyBase.Columns("Apellido1")
    Me.columnApellido2 = MyBase.Columns("Apellido2")
    Me.columnTelefono = MyBase.Columns("Telefono")
    Me.columnEmail = MyBase.Columns("Email")
    Me.columnDireccion = MyBase.Columns("Direccion")
    Me.columnCiudad = MyBase.Columns("Ciudad")
    Me.columnProvincia = MyBase.Columns("Provincia")
    Me.columnCP = MyBase.Columns("CP")
End Sub

Private Sub InitClass()
    Me.columnNIF = New System.Data.DataColumn("NIF", GetType(String), Nothing, System.Data.MappingType.Element)
    MyBase.Columns.Add(Me.columnNIF)
    Me.columnNombre = New System.Data.DataColumn("Nombre", GetType(String), Nothing, System.Data.MappingType.Element)
    MyBase.Columns.Add(Me.columnNombre)

```

```

Me.columnApellido1 = New System.Data.DataColumn("Apellido1", GetType(String), Nothing, System.Data.MappingType.String)
 MyBase.Columns.Add(Me.columnApellido1)
 Me.columnApellido2 = New System.Data.DataColumn("Apellido2", GetType(String), Nothing, System.Data.MappingType.String)
 MyBase.Columns.Add(Me.columnApellido2)
 Me.columnTelefono = New System.Data.DataColumn("Telefono", GetType(String), Nothing, System.Data.MappingType.String)
 MyBase.Columns.Add(Me.columnTelefono)
 Me.columnEmail = New System.Data.DataColumn("Email", GetType(String), Nothing, System.Data.MappingType.String)
 MyBase.Columns.Add(Me.columnEmail)
 Me.columnDireccion = New System.Data.DataColumn("Direccion", GetType(String), Nothing, System.Data.MappingType.String)
 MyBase.Columns.Add(Me.columnDireccion)
 Me.columnCiudad = New System.Data.DataColumn("Ciudad", GetType(String), Nothing, System.Data.MappingType.String)
 MyBase.Columns.Add(Me.columnCiudad)
 Me.columnProvincia = New System.Data.DataColumn("Provincia", GetType(String), Nothing, System.Data.MappingType.String)
 MyBase.Columns.Add(Me.columnProvincia)
 Me.columnCP = New System.Data.DataColumn("CP", GetType(String), Nothing, System.Data.MappingType.String)
 MyBase.Columns.Add(Me.columnCP)
 Me.Constraints.Add(New System.Data.UniqueConstraint("Constraint1", New System.Data.DataColumn() {Me.columnNIF}))
 Me.columnNIF.AllowDBNull = false
 Me.columnNIF.Unique = true
 Me.columnNIF.MaxLength = 10
 Me.columnNombre.MaxLength = 50
 Me.columnApellido1.MaxLength = 50
 Me.columnApellido2.MaxLength = 50
 Me.columnTelefono.MaxLength = 13
 Me.columnEmail.MaxLength = 50
 Me.columnDireccion.MaxLength = 100
 Me.columnCiudad.MaxLength = 50
 Me.columnProvincia.MaxLength = 50
 Me.columnCP.MaxLength = 5
 Me.Locale = New System.Globalization.CultureInfo("es-ES")
End Sub

Public Function NewSociosRow() As SociosRow
    Return CType(Me.NewRow, SociosRow)
End Function

Protected Overrides Function NewRowFromBuilder(ByVal builder As System.Data.DataRowBuilder) As System.Data.DataRow
    Return New SociosRow(builder)
End Function

Protected Overrides Function GetRowType() As System.Type
    Return GetType(SociosRow)
End Function

Protected Overrides Sub OnRowChanged(ByVal e As System.Data.DataRowChangeEventArgs)
    MyBase.OnRowChanged(e)
    If (Not (Me.SociosRowChangedEvent) Is Nothing) Then
        RaiseEvent SociosRowChanged(Me, New SociosRowChangeEvent(CType(e.Row, SociosRow), e.Action))
    End If
End Sub

Protected Overrides Sub OnRowChanging(ByVal e As System.Data.DataRowChangeEventArgs)
    MyBase.OnRowChanging(e)
    If (Not (Me.SociosRowChangingEvent) Is Nothing) Then
        RaiseEvent SociosRowChanging(Me, New SociosRowChangeEvent(CType(e.Row, SociosRow), e.Action))
    End If
End Sub

Protected Overrides Sub OnRowDeleted(ByVal e As System.Data.DataRowChangeEventArgs)
    MyBase.OnRowDeleted(e)
    If (Not (Me.SociosRowDeletedEvent) Is Nothing) Then
        RaiseEvent SociosRowDeleted(Me, New SociosRowChangeEvent(CType(e.Row, SociosRow), e.Action))
    End If
End Sub

Protected Overrides Sub OnRowDeleting(ByVal e As System.Data.DataRowChangeEventArgs)
    MyBase.OnRowDeleting(e)
    If (Not (Me.SociosRowDeletingEvent) Is Nothing) Then
        RaiseEvent SociosRowDeleting(Me, New SociosRowChangeEvent(CType(e.Row, SociosRow), e.Action))
    End If
End Sub

Public Sub RemoveSociosRow(ByVal row As SociosRow)
    Me.Rows.Remove(row)
End Sub

Public Shared Function GetTypedTableSchema(ByVal xs As System.Xml.Schema.XmlSchemaSet) As System.Xml.Schema.XmlSchemaComplexType
    Dim type As System.Xml.Schema.XmlSchemaComplexType = New System.Xml.Schema.XmlSchemaComplexType
    Dim sequence As System.Xml.Schema.XmlSchemaSequence = New System.Xml.Schema.XmlSchemaSequence
    Dim ds As DataSet1 = New DataSet1
    xs.Add(ds.GetSchemaSerializable)
    Dim any1 As System.Xml.Schema.XmlSchemaAny = New System.Xml.Schema.XmlSchemaAny
    any1.Namespace = "http://www.w3.org/2001/XMLSchema"
    any1.MinOccurs = New Decimal(0)
    any1.MaxOccurs = Decimal.MaxValue
    any1.ProcessContents = System.Xml.Schema.XmlSchemaContentProcessing.Lax
    sequence.Items.Add(any1)

```

```

    Dim any2 As System.Xml.Schema.XmlSchemaAny = New System.Xml.Schema.XmlSchemaAny
    any2.Namespace = "urn:schemas-microsoft-com:xml-diffgram-v1"
    any2.MinOccurs = New Decimal(1)
    any2.ProcessContents = System.Xml.Schema.XmlSchemaContentProcessing.Lax
    sequence.Items.Add(any2)
    Dim attribute1 As System.Xml.Schema.XmlSchemaAttribute = New System.Xml.Schema.XmlSchemaAttribute
    attribute1.Name = "namespace"
    attribute1.FixedValue = ds.Namespace
    type.Attributes.Add(attribute1)
    Dim attribute2 As System.Xml.Schema.XmlSchemaAttribute = New System.Xml.Schema.XmlSchemaAttribute
    attribute2.Name = "tableTypeName"
    attribute2.FixedValue = "SociosDataTable"
    type.Attributes.Add(attribute2)
    type.Particle = sequence
    Return type
End Function
End Class

Partial Public Class SociosRow
Inherits System.Data.DataRow

    Private tableSocios As SociosDataTable

    Friend Sub New(ByVal rb As System.Data.DataRowBuilder)
        MyBase.New(rb)
        Me.tableSocios = CType(Me.Table, SociosDataTable)
    End Sub

    Public Property NIF() As String
        Get
            Return CType(Me(Me.tableSocios.NIFColumn), String)
        End Get
        Set
            Me(Me.tableSocios.NIFColumn) = value
        End Set
    End Property

    Public Property Nombre() As String
        Get
            Try
                Return CType(Me(Me.tableSocios.NombreColumn), String)
            Catch e As System.InvalidCastException
                Throw New System.Data.StrongTypingException("El valor de la columna 'Nombre' de la tabla 'Socios' no se puede convertir en tipo String")
            End Try
        End Get
        Set
            Me(Me.tableSocios.NombreColumn) = value
        End Set
    End Property

    Public Property Apellido1() As String
        Get
            Try
                Return CType(Me(Me.tableSocios.Apellido1Column), String)
            Catch e As System.InvalidCastException
                Throw New System.Data.StrongTypingException("El valor de la columna 'Apellido1' de la tabla 'Socios' no se puede convertir en tipo String")
            End Try
        End Get
        Set
            Me(Me.tableSocios.Apellido1Column) = value
        End Set
    End Property

    Public Property Apellido2() As String
        Get
            Try
                Return CType(Me(Me.tableSocios.Apellido2Column), String)
            Catch e As System.InvalidCastException
                Throw New System.Data.StrongTypingException("El valor de la columna 'Apellido2' de la tabla 'Socios' no se puede convertir en tipo String")
            End Try
        End Get
        Set
            Me(Me.tableSocios.Apellido2Column) = value
        End Set
    End Property

    Public Property Telefono() As String
        Get
            Try
                Return CType(Me(Me.tableSocios.TelefonoColumn), String)
            Catch e As System.InvalidCastException
                Throw New System.Data.StrongTypingException("El valor de la columna 'Telefono' de la tabla 'Socios' no se puede convertir en tipo String")
            End Try
        End Get
        Set
            Me(Me.tableSocios.TelefonoColumn) = value
        End Set
    End Property

```

```

End Property

Public Property Email() As String
    Get
        Try
            Return CType(Me(Me.tableSocios.EmailColumn),String)
        Catch e As System.InvalidCastException
            Throw New System.Data.StrongTypingException("El valor de la columna 'Email' de la tabla 'Socios' es nulo")
        End Try
    End Get
    Set
        Me(Me.tableSocios.EmailColumn) = value
    End Set
End Property

Public Property Direccion() As String
    Get
        Try
            Return CType(Me(Me.tableSocios.DireccionColumn),String)
        Catch e As System.InvalidCastException
            Throw New System.Data.StrongTypingException("El valor de la columna 'Direccion' de la tabla 'Socios' es nulo")
        End Try
    End Get
    Set
        Me(Me.tableSocios.DireccionColumn) = value
    End Set
End Property

Public Property Ciudad() As String
    Get
        Try
            Return CType(Me(Me.tableSocios.CiudadColumn),String)
        Catch e As System.InvalidCastException
            Throw New System.Data.StrongTypingException("El valor de la columna 'Ciudad' de la tabla 'Socios' es nulo")
        End Try
    End Get
    Set
        Me(Me.tableSocios.CiudadColumn) = value
    End Set
End Property

Public Property Provincia() As String
    Get
        Try
            Return CType(Me(Me.tableSocios.ProvinciaColumn),String)
        Catch e As System.InvalidCastException
            Throw New System.Data.StrongTypingException("El valor de la columna 'Provincia' de la tabla 'Socios' es nulo")
        End Try
    End Get
    Set
        Me(Me.tableSocios.ProvinciaColumn) = value
    End Set
End Property

Public Property CP() As String
    Get
        Try
            Return CType(Me(Me.tableSocios.CPColumn),String)
        Catch e As System.InvalidCastException
            Throw New System.Data.StrongTypingException("El valor de la columna 'CP' de la tabla 'Socios' es nulo")
        End Try
    End Get
    Set
        Me(Me.tableSocios.CPColumn) = value
    End Set
End Property

Public Function IsNombreNull() As Boolean
    Return Me.IsNull(Me.tableSocios.NombreColumn)
End Function

Public Sub SetNombreNull()
    Me(Me.tableSocios.NombreColumn) = System.Convert.DBNull
End Sub

Public Function IsApellido1Null() As Boolean
    Return Me.IsNull(Me.tableSocios.Apellido1Column)
End Function

Public Sub SetApellido1Null()
    Me(Me.tableSocios.Apellido1Column) = System.Convert.DBNull
End Sub

Public Function IsApellido2Null() As Boolean
    Return Me.IsNull(Me.tableSocios.Apellido2Column)
End Function

```

```

Public Sub SetApellido2Null()
    Me(Me.tableSocios.Apellido2Column) = System.Convert.DBNull
End Sub

Public Function IsTelefonoNull() As Boolean
    Return Me.IsNull(Me.tableSocios.TelefonoColumn)
End Function

Public Sub SetTelefonoNull()
    Me(Me.tableSocios.TelefonoColumn) = System.Convert.DBNull
End Sub

Public Function IsEmailNull() As Boolean
    Return Me.IsNull(Me.tableSocios.EmailColumn)
End Function

Public Sub SetEmailNull()
    Me(Me.tableSocios.EmailColumn) = System.Convert.DBNull
End Sub

Public Function IsDireccionNull() As Boolean
    Return Me.IsNull(Me.tableSocios.DireccionColumn)
End Function

Public Sub SetDireccionNull()
    Me(Me.tableSocios.DireccionColumn) = System.Convert.DBNull
End Sub

Public Function IsCiudadNull() As Boolean
    Return Me.IsNull(Me.tableSocios.CiudadColumn)
End Function

Public Sub SetCiudadNull()
    Me(Me.tableSocios.CiudadColumn) = System.Convert.DBNull
End Sub

Public Function IsProvinciaNull() As Boolean
    Return Me.IsNull(Me.tableSocios.ProvinciaColumn)
End Function

Public Sub SetProvinciaNull()
    Me(Me.tableSocios.ProvinciaColumn) = System.Convert.DBNull
End Sub

Public Function IsCPNull() As Boolean
    Return Me.IsNull(Me.tableSocios.CPColumn)
End Function

Public Sub SetCPNull()
    Me(Me.tableSocios.CPColumn) = System.Convert.DBNull
End Sub
End Class

Public Class SociosRowChangeEvent
    Inherits System.EventArgs

    Private eventRow As SociosRow

    Private eventAction As System.Data.DataRowAction

    Public Sub New(ByVal row As SociosRow, ByVal action As System.Data.DataRowAction)
        MyBase.New
        Me.eventRow = row
        Me.eventAction = action
    End Sub

    Public ReadOnly Property Row() As SociosRow
        Get
            Return Me.eventRow
        End Get
    End Property

    Public ReadOnly Property Action() As System.Data.DataRowAction
        Get
            Return Me.eventAction
        End Get
    End Property
End Class
End Class

Namespace DataSet1TableAdapters

<System.ComponentModel.DesignerCategoryAttribute("code"), _
System.ComponentModel.ToolboxItem(true), _
System.ComponentModel.DataObjectAttribute(true), _
System.ComponentModel.DesignerAttribute("Microsoft.VSDesigner.DataSource.Design.TableAdapterDesigner", Micros
    " Version=8.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a"), _

```

```

System.ComponentModel.Design.HelpKeywordAttribute("vs.data.TableAdapter")> _
Partial Public Class SociosTableAdapter
    Inherits System.ComponentModel.Component

    Private WithEvents m_adapter As System.Data.SqlClient.SqlDataAdapter

    Private m_connection As System.Data.SqlClient.SqlConnection

    Private m_commandCollection() As System.Data.SqlClient.SqlCommand

    Private m_clearBeforeFill As Boolean

    Public Sub New()
        MyBase.New
        Me.m_clearBeforeFill = true
    End Sub

    Private ReadOnly Property Adapter() As System.Data.SqlClient.SqlDataAdapter
        Get
            If (Me.m_adapter Is Nothing) Then
                Me.InitAdapter
            End If
            Return Me.m_adapter
        End Get
    End Property

    Friend Property Connection() As System.Data.SqlClient.SqlConnection
        Get
            If (Me.m_connection Is Nothing) Then
                Me.InitConnection
            End If
            Return Me.m_connection
        End Get
        Set
            Me.m_connection = value
            If (Not (Me.Adapter.InsertCommand) Is Nothing) Then
                Me.Adapter.InsertCommand.Connection = value
            End If
            If (Not (Me.Adapter.DeleteCommand) Is Nothing) Then
                Me.Adapter.DeleteCommand.Connection = value
            End If
            If (Not (Me.Adapter.UpdateCommand) Is Nothing) Then
                Me.Adapter.UpdateCommand.Connection = value
            End If
            Dim i As Integer = 0
            Do While (i < Me.CommandCollection.Length)
                If (Not (Me.CommandCollection(i)) Is Nothing) Then
                    CType(Me.CommandCollection(i), System.Data.SqlClient.SqlCommand).Connection = value
                End If
                i = (i + 1)
            Loop
        End Set
    End Property

    Protected ReadOnly Property CommandCollection() As System.Data.SqlClient.SqlCommand()
        Get
            If (Me.m_commandCollection Is Nothing) Then
                Me.InitCommandCollection
            End If
            Return Me.m_commandCollection
        End Get
    End Property

    Public Property ClearBeforeFill() As Boolean
        Get
            Return Me.m_clearBeforeFill
        End Get
        Set
            Me.m_clearBeforeFill = value
        End Set
    End Property

    Private Sub InitAdapter()
        Me.m_adapter = New System.Data.SqlClient.SqlDataAdapter
        Dim tableMapping As System.Data.Common.DataTableMapping = New System.Data.Common.DataTableMapping
        tableMapping.SourceTable = "Table"
        tableMapping.DataSetName = "Socios"
        tableMapping.ColumnMappings.Add("NIF", "NIF")
        tableMapping.ColumnMappings.Add("Nombre", "Nombre")
        tableMapping.ColumnMappings.Add("Apellido1", "Apellido1")
        tableMapping.ColumnMappings.Add("Apellido2", "Apellido2")
        tableMapping.ColumnMappings.Add("Telefono", "Telefono")
        tableMapping.ColumnMappings.Add("Email", "Email")
        tableMapping.ColumnMappings.Add("Direccion", "Direccion")
        tableMapping.ColumnMappings.Add("Ciudad", "Ciudad")
        tableMapping.ColumnMappings.Add("Provincia", "Provincia")
        tableMapping.ColumnMappings.Add("CP", "CP")
    End Sub

```

```

Me.m_adapter.TableMappings.Add(tableMapping)
Me.m_adapter.DeleteCommand = New System.Data.SqlClient.SqlCommand
Me.m_adapter.DeleteCommand.Connection = Me.Connection
Me.m_adapter.DeleteCommand.CommandText = "DELETE FROM [dbo].[Socios] WHERE (([NIF] = @Original_NIF) AND
    "1 AND [Nombre] IS NULL) OR ([Nombre] = @Original_Nombre)) AND ((@IsNull_Apellido" & _
    "1 = 1 AND [Apellido1] IS NULL) OR ([Apellido1] = @Original_Apellido1)) AND ((@Is" & _
    "Null_Apellido2 = 1 AND [Apellido2] IS NULL) OR ([Apellido2] = @Original_Apellido" & _
    "2)) AND ((@IsNull_Teléfono = 1 AND [Teléfono] IS NULL) OR ([Teléfono] = @Origina" & _
    "l_Teléfono)) AND ((@IsNull_Email = 1 AND [Email] IS NULL) OR ([Email] = @Origina" & _
    "l_Email)) AND ((@IsNull_Dirección = 1 AND [Dirección] IS NULL) OR ([Dirección] = "& -
    "@Original_Dirección)) AND ((@IsNull_Ciudad = 1 AND [Ciudad] IS NULL) OR ([Ciuda" & -
    "d] = @Original_Ciudad)) AND ((@IsNull_Provincia = 1 AND [Provincia] IS NULL) OR ("& -
    "[Provincia] = @Original_Provincia)) AND ((@IsNull_CP = 1 AND [CP] IS NULL) OR ("& -
    "[CP] = @Original_CP)))"
Me.m_adapter.DeleteCommand.CommandType = System.Data.CommandType.Text
Me.m_adapter.DeleteCommand.Parameters.Add(New System.Data.SqlClient.SqlParameter("@Original_NIF", Syst
Me.m_adapter.DeleteCommand.Parameters.Add(New System.Data.SqlClient.SqlParameter("@IsNull_Nombre", Syst
Me.m_adapter.DeleteCommand.Parameters.Add(New System.Data.SqlClient.SqlParameter("@Original_Nombre", Sy
Me.m_adapter.DeleteCommand.Parameters.Add(New System.Data.SqlClient.SqlParameter("@IsNull_Apellido1", S
Me.m_adapter.DeleteCommand.Parameters.Add(New System.Data.SqlClient.SqlParameter("@Original_Apellido1",
Me.m_adapter.DeleteCommand.Parameters.Add(New System.Data.SqlClient.SqlParameter("@IsNull_Apellido2", S
Me.m_adapter.DeleteCommand.Parameters.Add(New System.Data.SqlClient.SqlParameter("@Original_Apellido2",
Me.m_adapter.DeleteCommand.Parameters.Add(New System.Data.SqlClient.SqlParameter("@IsNull_Teléfono", Sy
Me.m_adapter.DeleteCommand.Parameters.Add(New System.Data.SqlClient.SqlParameter("@Original_Teléfono",
Me.m_adapter.DeleteCommand.Parameters.Add(New System.Data.SqlClient.SqlParameter("@IsNull_Email", Syst
Me.m_adapter.DeleteCommand.Parameters.Add(New System.Data.SqlClient.SqlParameter("@Original_Email", Sys
Me.m_adapter.DeleteCommand.Parameters.Add(New System.Data.SqlClient.SqlParameter("@IsNull_Dirección", S
Me.m_adapter.DeleteCommand.Parameters.Add(New System.Data.SqlClient.SqlParameter("@Original_Dirección",
Me.m_adapter.DeleteCommand.Parameters.Add(New System.Data.SqlClient.SqlParameter("@IsNull_Ciudad", Syst
Me.m_adapter.DeleteCommand.Parameters.Add(New System.Data.SqlClient.SqlParameter("@Original_Ciudad", Sy
Me.m_adapter.DeleteCommand.Parameters.Add(New System.Data.SqlClient.SqlParameter("@IsNull_Provincia", S
Me.m_adapter.DeleteCommand.Parameters.Add(New System.Data.SqlClient.SqlParameter("@Original_Provincia",
Me.m_adapter.DeleteCommand.Parameters.Add(New System.Data.SqlClient.SqlParameter("@IsNull_CP", System.D
Me.m_adapter.DeleteCommand.Parameters.Add(New System.Data.SqlClient.SqlParameter("@Original_CP", System
Me.m_adapter.InsertCommand = New System.Data.SqlClient.SqlCommand
Me.m_adapter.InsertCommand.Connection = Me.Connection
Me.m_adapter.InsertCommand.CommandText = "INSERT INTO [dbo].[Socios] ([NIF], [Nombre], [Apellido1], [Ap
    ", [Email], [Dirección], [Ciudad], [Provincia], [CP]) VALUES (@NIF, @Nombre, @Ape" & _
    "llido1, @Apellido2, @Telefono, @Email, @Dirección, @Ciudad, @Provincia, @CP);&Global.Microsoft.Vis
    "ELECT NIF, Nombre, Apellido1, Apellido2, Telefono, Email, Dirección, Ciudad, Pro" & -
    "vincia, CP FROM Socios WHERE (NIF = @NIF)"
Me.m_adapter.InsertCommand.CommandType = System.Data.CommandType.Text
Me.m_adapter.InsertCommand.Parameters.Add(New System.Data.SqlClient.SqlParameter("@NIF", System.Data.Sq
Me.m_adapter.InsertCommand.Parameters.Add(New System.Data.SqlClient.SqlParameter("@Nombre", System.Data
Me.m_adapter.InsertCommand.Parameters.Add(New System.Data.SqlClient.SqlParameter("@Apellido1", System.D
Me.m_adapter.InsertCommand.Parameters.Add(New System.Data.SqlClient.SqlParameter("@Apellido2", System.D
Me.m_adapter.InsertCommand.Parameters.Add(New System.Data.SqlClient.SqlParameter("@Telefono", System.D
Me.m_adapter.InsertCommand.Parameters.Add(New System.Data.SqlClient.SqlParameter("@Email", System.Data.
Me.m_adapter.InsertCommand.Parameters.Add(New System.Data.SqlClient.SqlParameter("@Dirección", System.D
Me.m_adapter.InsertCommand.Parameters.Add(New System.Data.SqlClient.SqlParameter("@Ciudad", System.Data
Me.m_adapter.InsertCommand.Parameters.Add(New System.Data.SqlClient.SqlParameter("@Provincia", System.D
Me.m_adapter.InsertCommand.Parameters.Add(New System.Data.SqlClient.SqlParameter("@CP", System.Data.Sql
Me.m_adapter.UpdateCommand = New System.Data.SqlClient.SqlCommand
Me.m_adapter.UpdateCommand.Connection = Me.Connection
Me.m_adapter.UpdateCommand.CommandText = "UPDATE [dbo].[Socios] SET [NIF] = @NIF, [Nombre] = @Nombre, [
    "do1, [Apellido2] = @Apellido2, [Telefono] = @Telefono, [Email] = @Email, [Direcc]" & -
    "ion] = @Dirección, [Ciudad] = @Ciudad, [Provincia] = @Provincia, [CP] = @CP WHER" & -
    "E (([NIF] = @Original_NIF) AND ((@IsNull_Nombre = 1 AND [Nombre] IS NULL) OR ([N" & -
    "ombre] = @Original_Nombre)) AND ((@IsNull_Apellido1 = 1 AND [Apellido1] IS NULL)" & -
    " OR ([Apellido1] = @Original_Apellido1)) AND ((@IsNull_Apellido2 = 1 AND [Apelli" & -
    "do2] IS NULL) OR ([Apellido2] = @Original_Apellido2)) AND ((@IsNull_Teléfono = 1" & -
    " AND [Teléfono] IS NULL) OR ([Teléfono] = @Original_Teléfono)) AND ((@IsNull_Ema" & -
    "il = 1 AND [Email] IS NULL) OR ([Email] = @Original_Email)) AND ((@IsNull_Direcc" & -
    "ion = 1 AND [Dirección] IS NULL) OR ([Dirección] = @Original_Dirección)) AND ((@" & -
    "IsNull_Ciudad = 1 AND [Ciudad] IS NULL) OR ([Ciudad] = @Original_Ciudad)) AND ((& -
    "@IsNull_Provincia = 1 AND [Provincia] IS NULL) OR ([Provincia] = @Original_Provi" & -
    "ncia)) AND ((@IsNull_CP = 1 AND [CP] IS NULL) OR ([CP] = @Original_CP)));&Global.Microsoft.Visua
    "T NIF, Nombre, Apellido1, Apellido2, Telefono, Email, Dirección, Ciudad, Provinci" & -
    "ia, CP FROM Socios WHERE (NIF = @NIF)"
Me.m_adapter.UpdateCommand.CommandType = System.Data.CommandType.Text
Me.m_adapter.UpdateCommand.Parameters.Add(New System.Data.SqlClient.SqlParameter("@NIF", System.Data.Sq
Me.m_adapter.UpdateCommand.Parameters.Add(New System.Data.SqlClient.SqlParameter("@Nombre", System.Data
Me.m_adapter.UpdateCommand.Parameters.Add(New System.Data.SqlClient.SqlParameter("@Apellido1", System.D
Me.m_adapter.UpdateCommand.Parameters.Add(New System.Data.SqlClient.SqlParameter("@Apellido2", System.D
Me.m_adapter.UpdateCommand.Parameters.Add(New System.Data.SqlClient.SqlParameter("@Telefono", System.D
Me.m_adapter.UpdateCommand.Parameters.Add(New System.Data.SqlClient.SqlParameter("@Email", System.Data.
Me.m_adapter.UpdateCommand.Parameters.Add(New System.Data.SqlClient.SqlParameter("@Dirección", System.D
Me.m_adapter.UpdateCommand.Parameters.Add(New System.Data.SqlClient.SqlParameter("@Ciudad", System.Data
Me.m_adapter.UpdateCommand.Parameters.Add(New System.Data.SqlClient.SqlParameter("@Provincia", System.D
Me.m_adapter.UpdateCommand.Parameters.Add(New System.Data.SqlClient.SqlParameter("@CP", System.Data.Sql
Me.m_adapter.UpdateCommand.Parameters.Add(New System.Data.SqlClient.SqlParameter("@Original_NIF", Syst
Me.m_adapter.UpdateCommand.Parameters.Add(New System.Data.SqlClient.SqlParameter("@IsNull_Nombre", Syst
Me.m_adapter.UpdateCommand.Parameters.Add(New System.Data.SqlClient.SqlParameter("@Original_Nombre", Sy
Me.m_adapter.UpdateCommand.Parameters.Add(New System.Data.SqlClient.SqlParameter("@IsNull_Apellido1", S
Me.m_adapter.UpdateCommand.Parameters.Add(New System.Data.SqlClient.SqlParameter("@Original_Apellido1",
Me.m_adapter.UpdateCommand.Parameters.Add(New System.Data.SqlClient.SqlParameter("@IsNull_Apellido2", S
Me.m_adapter.UpdateCommand.Parameters.Add(New System.Data.SqlClient.SqlParameter("@Original_Apellido2",

```

```

Me.m_adapter.UpdateCommand.Parameters.Add(New System.Data.SqlClient.SqlParameter("@IsNull_Teléfono", System.Data.SqlDbType.Bit))
Me.m_adapter.UpdateCommand.Parameters.Add(New System.Data.SqlClient.SqlParameter("@Original_Teléfono", System.Data.SqlDbType.VarChar))
Me.m_adapter.UpdateCommand.Parameters.Add(New System.Data.SqlClient.SqlParameter("@IsNull_Email", System.Data.SqlDbType.VarChar))
Me.m_adapter.UpdateCommand.Parameters.Add(New System.Data.SqlClient.SqlParameter("@Original_Email", System.Data.SqlDbType.VarChar))
Me.m_adapter.UpdateCommand.Parameters.Add(New System.Data.SqlClient.SqlParameter("@IsNull_Dirección", System.Data.SqlDbType.VarChar))
Me.m_adapter.UpdateCommand.Parameters.Add(New System.Data.SqlClient.SqlParameter("@Original_Dirección", System.Data.SqlDbType.VarChar))
Me.m_adapter.UpdateCommand.Parameters.Add(New System.Data.SqlClient.SqlParameter("@IsNull_Ciudad", System.Data.SqlDbType.VarChar))
Me.m_adapter.UpdateCommand.Parameters.Add(New System.Data.SqlClient.SqlParameter("@Original_Ciudad", System.Data.SqlDbType.VarChar))
Me.m_adapter.UpdateCommand.Parameters.Add(New System.Data.SqlClient.SqlParameter("@IsNull_Provincia", System.Data.SqlDbType.VarChar))
Me.m_adapter.UpdateCommand.Parameters.Add(New System.Data.SqlClient.SqlParameter("@Original_Provincia", System.Data.SqlDbType.VarChar))
Me.m_adapter.UpdateCommand.Parameters.Add(New System.Data.SqlClient.SqlParameter("@IsNull_CP", System.Data.SqlDbType.VarChar))
Me.m_adapter.UpdateCommand.Parameters.Add(New System.Data.SqlClient.SqlParameter("@Original_CP", System.Data.SqlDbType.VarChar))

End Sub

Private Sub InitConnection()
    Me.m_connection = New System.Data.SqlClient.SqlConnection
    Me.m_connection.ConnectionString = WindowsApplication3.Settings.Default.MSDNVideoConnectionString1
End Sub

Private Sub InitCommandCollection()
    Me.m_commandCollection = New System.Data.SqlClient.SqlCommandCollection()
    Me.m_commandCollection(0) = New System.Data.SqlClient.SqlCommand
    Me.m_commandCollection(0).Connection = Me.Connection
    Me.m_commandCollection(0).CommandText = "SELECT NIF, Nombre, Apellido1, Apellido2, Telefono, Email, Dirección, CP FROM dbo.Socios"
    Me.m_commandCollection(0).CommandType = System.Data.CommandType.Text
End Sub

<System.ComponentModel.DataObjectMethodAttribute(System.ComponentModel.DataObjectMethodType.Fill, true)>
Public Overloads Overridable Function Fill(ByVal dataTable As DataSet1.SociosDataTable) As Integer
    Me.Adapter.SelectCommand = Me.CommandCollection(0)
    If (Me.m_clearBeforeFill = true) Then
        dataTable.Clear
    End If
    Dim returnValue As Integer = Me.Adapter.Fill(dataTable)
    Return returnValue
End Function

<System.ComponentModel.DataObjectMethodAttribute(System.ComponentModel.DataObjectMethodType.[Select], true)>
Public Overloads Overridable Function GetData() As DataSet1.SociosDataTable
    Me.Adapter.SelectCommand = Me.CommandCollection(0)
    Dim dataTable As DataSet1.SociosDataTable = New DataSet1.SociosDataTable
    Me.Adapter.Fill(dataTable)
    Return dataTable
End Function

Public Overloads Overridable Function Update(ByVal dataTable As DataSet1.SociosDataTable) As Integer
    Return Me.Adapter.Update(dataTable)
End Function

Public Overloads Overridable Function Update(ByVal dataSet As DataSet1) As Integer
    Return Me.Adapter.Update(dataSet, "Socios")
End Function

Public Overloads Overridable Function Update(ByVal dataRow As System.Data.DataRow) As Integer
    Return Me.Adapter.Update(New System.Data.DataRow() {dataRow})
End Function

Public Overloads Overridable Function Update(ByVal dataRows() As System.Data.DataRow) As Integer
    Return Me.Adapter.Update(dataRows)
End Function

<System.ComponentModel.DataObjectMethodAttribute(System.ComponentModel.DataObjectMethodType.Delete, true)>
Public Overloads Overridable Function Delete(ByVal Original_NIF As String, ByVal Original_Nombre As String)
    If (Original_NIF Is Nothing) Then
        Throw New System.ArgumentNullException("Original_NIF")
    Else
        Me.Adapter.DeleteCommand.Parameters(0).Value = CType(Original_NIF, String)
    End If
    If (Original_Nombre Is Nothing) Then
        Me.Adapter.DeleteCommand.Parameters(1).Value = CType(1, Integer)
        Me.Adapter.DeleteCommand.Parameters(2).Value = System.DBNull.Value
    Else
        Me.Adapter.DeleteCommand.Parameters(1).Value = CType(0, Integer)
        Me.Adapter.DeleteCommand.Parameters(2).Value = CType(Original_Nombre, String)
    End If
    If (Original_Apellido1 Is Nothing) Then
        Me.Adapter.DeleteCommand.Parameters(3).Value = CType(1, Integer)
        Me.Adapter.DeleteCommand.Parameters(4).Value = System.DBNull.Value
    Else
        Me.Adapter.DeleteCommand.Parameters(3).Value = CType(0, Integer)
        Me.Adapter.DeleteCommand.Parameters(4).Value = CType(Original_Apellido1, String)
    End If
    If (Original_Apellido2 Is Nothing) Then
        Me.Adapter.DeleteCommand.Parameters(5).Value = CType(1, Integer)
        Me.Adapter.DeleteCommand.Parameters(6).Value = System.DBNull.Value
    Else

```

```

        Me.Adapter.DeleteCommand.Parameters(5).Value = CType(0, Integer)
        Me.Adapter.DeleteCommand.Parameters(6).Value = CType(Original_Apellido2, String)
    End If
    If (Original_Telefono Is Nothing) Then
        Me.Adapter.DeleteCommand.Parameters(7).Value = CType(1, Integer)
        Me.Adapter.DeleteCommand.Parameters(8).Value = System.DBNull.Value
    Else
        Me.Adapter.DeleteCommand.Parameters(7).Value = CType(0, Integer)
        Me.Adapter.DeleteCommand.Parameters(8).Value = CType(Original_Telefono, String)
    End If
    If (Original_Email Is Nothing) Then
        Me.Adapter.DeleteCommand.Parameters(9).Value = CType(1, Integer)
        Me.Adapter.DeleteCommand.Parameters(10).Value = System.DBNull.Value
    Else
        Me.Adapter.DeleteCommand.Parameters(9).Value = CType(0, Integer)
        Me.Adapter.DeleteCommand.Parameters(10).Value = CType(Original_Email, String)
    End If
    If (Original_Direccion Is Nothing) Then
        Me.Adapter.DeleteCommand.Parameters(11).Value = CType(1, Integer)
        Me.Adapter.DeleteCommand.Parameters(12).Value = System.DBNull.Value
    Else
        Me.Adapter.DeleteCommand.Parameters(11).Value = CType(0, Integer)
        Me.Adapter.DeleteCommand.Parameters(12).Value = CType(Original_Direccion, String)
    End If
    If (Original_Ciudad Is Nothing) Then
        Me.Adapter.DeleteCommand.Parameters(13).Value = CType(1, Integer)
        Me.Adapter.DeleteCommand.Parameters(14).Value = System.DBNull.Value
    Else
        Me.Adapter.DeleteCommand.Parameters(13).Value = CType(0, Integer)
        Me.Adapter.DeleteCommand.Parameters(14).Value = CType(Original_Ciudad, String)
    End If
    If (Original_Provincia Is Nothing) Then
        Me.Adapter.DeleteCommand.Parameters(15).Value = CType(1, Integer)
        Me.Adapter.DeleteCommand.Parameters(16).Value = System.DBNull.Value
    Else
        Me.Adapter.DeleteCommand.Parameters(15).Value = CType(0, Integer)
        Me.Adapter.DeleteCommand.Parameters(16).Value = CType(Original_Provincia, String)
    End If
    If (Original_CP Is Nothing) Then
        Me.Adapter.DeleteCommand.Parameters(17).Value = CType(1, Integer)
        Me.Adapter.DeleteCommand.Parameters(18).Value = System.DBNull.Value
    Else
        Me.Adapter.DeleteCommand.Parameters(17).Value = CType(0, Integer)
        Me.Adapter.DeleteCommand.Parameters(18).Value = CType(Original_CP, String)
    End If
    Dim previousConnectionState As System.Data.ConnectionState = Me.Adapter.DeleteCommand.Connection.State
    Me.Adapter.DeleteCommand.Connection.Open
    Try
        Return Me.Adapter.DeleteCommand.ExecuteNonQuery
    Finally
        If (previousConnectionState = System.Data.ConnectionState.Closed) Then
            Me.Adapter.DeleteCommand.Connection.Close
        End If
    End Try
End Function

<System.ComponentModel.DataObjectMethodAttribute(System.ComponentModel.DataObjectType.Insert, true)>
Public Overloads Overrides Function Insert(ByVal NIF As String, ByVal Nombre As String, ByVal Apellido1
If (NIF Is Nothing) Then
    Throw New System.ArgumentNullException("NIF")
Else
    Me.Adapter.InsertCommand.Parameters(0).Value = CType(NIF, String)
End If
If (Nombre Is Nothing) Then
    Me.Adapter.InsertCommand.Parameters(1).Value = System.DBNull.Value
Else
    Me.Adapter.InsertCommand.Parameters(1).Value = CType(Nombre, String)
End If
If (Apellido1 Is Nothing) Then
    Me.Adapter.InsertCommand.Parameters(2).Value = System.DBNull.Value
Else
    Me.Adapter.InsertCommand.Parameters(2).Value = CType(Apellido1, String)
End If
If (Apellido2 Is Nothing) Then
    Me.Adapter.InsertCommand.Parameters(3).Value = System.DBNull.Value
Else
    Me.Adapter.InsertCommand.Parameters(3).Value = CType(Apellido2, String)
End If
If (Telefono Is Nothing) Then
    Me.Adapter.InsertCommand.Parameters(4).Value = System.DBNull.Value
Else
    Me.Adapter.InsertCommand.Parameters(4).Value = CType(Telefono, String)
End If
If (Email Is Nothing) Then
    Me.Adapter.InsertCommand.Parameters(5).Value = System.DBNull.Value
Else
    Me.Adapter.InsertCommand.Parameters(5).Value = CType>Email, String)
```

```

End If
If (Direccion Is Nothing) Then
    Me.Adapter.InsertCommand.Parameters(6).Value = System.DBNull.Value
Else
    Me.Adapter.InsertCommand.Parameters(6).Value = CType(Direccion, String)
End If
If (Ciudad Is Nothing) Then
    Me.Adapter.InsertCommand.Parameters(7).Value = System.DBNull.Value
Else
    Me.Adapter.InsertCommand.Parameters(7).Value = CType(Ciudad, String)
End If
If (Provincia Is Nothing) Then
    Me.Adapter.InsertCommand.Parameters(8).Value = System.DBNull.Value
Else
    Me.Adapter.InsertCommand.Parameters(8).Value = CType(Provincia, String)
End If
If (CP Is Nothing) Then
    Me.Adapter.InsertCommand.Parameters(9).Value = System.DBNull.Value
Else
    Me.Adapter.InsertCommand.Parameters(9).Value = CType(CP, String)
End If
Dim previousConnectionState As System.Data.ConnectionState = Me.Adapter.InsertCommand.Connection.State
Me.Adapter.InsertCommand.Connection.Open
Try
    Return Me.Adapter.InsertCommand.ExecuteNonQuery
Finally
    If (previousConnectionState = System.Data.ConnectionState.Closed) Then
        Me.Adapter.InsertCommand.Connection.Close
    End If
End Try
End Function

<System.ComponentModel.DataObjectMethodAttribute(System.ComponentModel.DataObjectMethodType.Update, true)>
Public Overloads Overrides Function Update( _
    ByVal NIF As String, _
    ByVal Nombre As String, _
    ByVal Apellido1 As String, _
    ByVal Apellido2 As String, _
    ByVal Telefono As String, _
    ByVal Email As String, _
    ByVal Direccion As String, _
    ByVal Ciudad As String, _
    ByVal Provincia As String, _
    ByVal CP As String, _
    ByVal Original_NIF As String, _
    ByVal Original_Nombre As String, _
    ByVal Original_Apellido1 As String, _
    ByVal Original_Apellido2 As String, _
    ByVal Original_Telefono As String, _
    ByVal Original_Email As String, _
    ByVal Original_Direccion As String, _
    ByVal Original_Ciudad As String, _
    ByVal Original_Provincia As String, _
    ByVal Original_CP As String) As Integer
If (NIF Is Nothing) Then
    Throw New System.ArgumentNullException("NIF")
Else
    Me.Adapter.UpdateCommand.Parameters(0).Value = CType(NIF, String)
End If
If (Nombre Is Nothing) Then
    Me.Adapter.UpdateCommand.Parameters(1).Value = System.DBNull.Value
Else
    Me.Adapter.UpdateCommand.Parameters(1).Value = CType(Nombre, String)
End If
If (Apellido1 Is Nothing) Then
    Me.Adapter.UpdateCommand.Parameters(2).Value = System.DBNull.Value
Else
    Me.Adapter.UpdateCommand.Parameters(2).Value = CType(Apellido1, String)
End If
If (Apellido2 Is Nothing) Then
    Me.Adapter.UpdateCommand.Parameters(3).Value = System.DBNull.Value
Else
    Me.Adapter.UpdateCommand.Parameters(3).Value = CType(Apellido2, String)
End If
If (Telefono Is Nothing) Then
    Me.Adapter.UpdateCommand.Parameters(4).Value = System.DBNull.Value
Else
    Me.Adapter.UpdateCommand.Parameters(4).Value = CType(Telefono, String)
End If
If (Email Is Nothing) Then
    Me.Adapter.UpdateCommand.Parameters(5).Value = System.DBNull.Value
Else
    Me.Adapter.UpdateCommand.Parameters(5).Value = CType>Email, String)
End If
If (Direccion Is Nothing) Then
    Me.Adapter.UpdateCommand.Parameters(6).Value = System.DBNull.Value
Else

```

```

        Me.Adapter.UpdateCommand.Parameters(6).Value = CType(Direccion, String)
End If
If (Ciudad Is Nothing) Then
    Me.Adapter.UpdateCommand.Parameters(7).Value = System.DBNull.Value
Else
    Me.Adapter.UpdateCommand.Parameters(7).Value = CType(Ciudad, String)
End If
If (Provincia Is Nothing) Then
    Me.Adapter.UpdateCommand.Parameters(8).Value = System.DBNull.Value
Else
    Me.Adapter.UpdateCommand.Parameters(8).Value = CType(Provincia, String)
End If
If (CP Is Nothing) Then
    Me.Adapter.UpdateCommand.Parameters(9).Value = System.DBNull.Value
Else
    Me.Adapter.UpdateCommand.Parameters(9).Value = CType(CP, String)
End If
If (Original_NIF Is Nothing) Then
    Throw New System.ArgumentNullException("Original_NIF")
Else
    Me.Adapter.UpdateCommand.Parameters(10).Value = CType(Original_NIF, String)
End If
If (Original_Nombre Is Nothing) Then
    Me.Adapter.UpdateCommand.Parameters(11).Value = CType(1, Integer)
    Me.Adapter.UpdateCommand.Parameters(12).Value = System.DBNull.Value
Else
    Me.Adapter.UpdateCommand.Parameters(11).Value = CType(0, Integer)
    Me.Adapter.UpdateCommand.Parameters(12).Value = CType(Original_Nombre, String)
End If
If (Original_Apellidol Is Nothing) Then
    Me.Adapter.UpdateCommand.Parameters(13).Value = CType(1, Integer)
    Me.Adapter.UpdateCommand.Parameters(14).Value = System.DBNull.Value
Else
    Me.Adapter.UpdateCommand.Parameters(13).Value = CType(0, Integer)
    Me.Adapter.UpdateCommand.Parameters(14).Value = CType(Original_Apellidol, String)
End If
If (Original_Apellido2 Is Nothing) Then
    Me.Adapter.UpdateCommand.Parameters(15).Value = CType(1, Integer)
    Me.Adapter.UpdateCommand.Parameters(16).Value = System.DBNull.Value
Else
    Me.Adapter.UpdateCommand.Parameters(15).Value = CType(0, Integer)
    Me.Adapter.UpdateCommand.Parameters(16).Value = CType(Original_Apellido2, String)
End If
If (Original_Telefono Is Nothing) Then
    Me.Adapter.UpdateCommand.Parameters(17).Value = CType(1, Integer)
    Me.Adapter.UpdateCommand.Parameters(18).Value = System.DBNull.Value
Else
    Me.Adapter.UpdateCommand.Parameters(17).Value = CType(0, Integer)
    Me.Adapter.UpdateCommand.Parameters(18).Value = CType(Original_Telefono, String)
End If
If (Original_Email Is Nothing) Then
    Me.Adapter.UpdateCommand.Parameters(19).Value = CType(1, Integer)
    Me.Adapter.UpdateCommand.Parameters(20).Value = System.DBNull.Value
Else
    Me.Adapter.UpdateCommand.Parameters(19).Value = CType(0, Integer)
    Me.Adapter.UpdateCommand.Parameters(20).Value = CType(Original_Email, String)
End If
If (Original_Direccion Is Nothing) Then
    Me.Adapter.UpdateCommand.Parameters(21).Value = CType(1, Integer)
    Me.Adapter.UpdateCommand.Parameters(22).Value = System.DBNull.Value
Else
    Me.Adapter.UpdateCommand.Parameters(21).Value = CType(0, Integer)
    Me.Adapter.UpdateCommand.Parameters(22).Value = CType(Original_Direccion, String)
End If
If (Original_Ciudad Is Nothing) Then
    Me.Adapter.UpdateCommand.Parameters(23).Value = CType(1, Integer)
    Me.Adapter.UpdateCommand.Parameters(24).Value = System.DBNull.Value
Else
    Me.Adapter.UpdateCommand.Parameters(23).Value = CType(0, Integer)
    Me.Adapter.UpdateCommand.Parameters(24).Value = CType(Original_Ciudad, String)
End If
If (Original_Provincia Is Nothing) Then
    Me.Adapter.UpdateCommand.Parameters(25).Value = CType(1, Integer)
    Me.Adapter.UpdateCommand.Parameters(26).Value = System.DBNull.Value
Else
    Me.Adapter.UpdateCommand.Parameters(25).Value = CType(0, Integer)
    Me.Adapter.UpdateCommand.Parameters(26).Value = CType(Original_Provincia, String)
End If
If (Original_CP Is Nothing) Then
    Me.Adapter.UpdateCommand.Parameters(27).Value = CType(1, Integer)
    Me.Adapter.UpdateCommand.Parameters(28).Value = System.DBNull.Value
Else
    Me.Adapter.UpdateCommand.Parameters(27).Value = CType(0, Integer)
    Me.Adapter.UpdateCommand.Parameters(28).Value = CType(Original_CP, String)
End If
Dim previousConnectionState As System.Data.ConnectionState = Me.Adapter.UpdateCommand.Connection.State
Me.Adapter.UpdateCommand.Connection.Open

```

```
Try
    Return Me.Adapter.UpdateCommand.ExecuteNonQuery
Finally
    If (previousConnectionState = System.Data.ConnectionState.Closed) Then
        Me.Adapter.UpdateCommand.Connection.Close
    End If
End Try
End Function
End Class
End Namespace
```

Más adelante veremos como usarlo, pero antes veremos otra forma de crear nuestro *DataSet tipado*.

Esta que hemos visto, es la forma más sencilla y habitual de crearlo, pero existe otra manera que es un procedimiento manual que a modo general, conviene que la conozcamos.

Lección 4: DataSet tipados

- ¿Qué son los DataSets tipados?
- Generando nuestros DataSets tipados
- Generando un DataSet tipado con Visual Studio
- Generando un DataSet tipado con la línea de comandos
- Usando los DataSets tipados

Módulo 5 - Capítulo 4

4. Generando un DataSet tipado con la línea de comandos

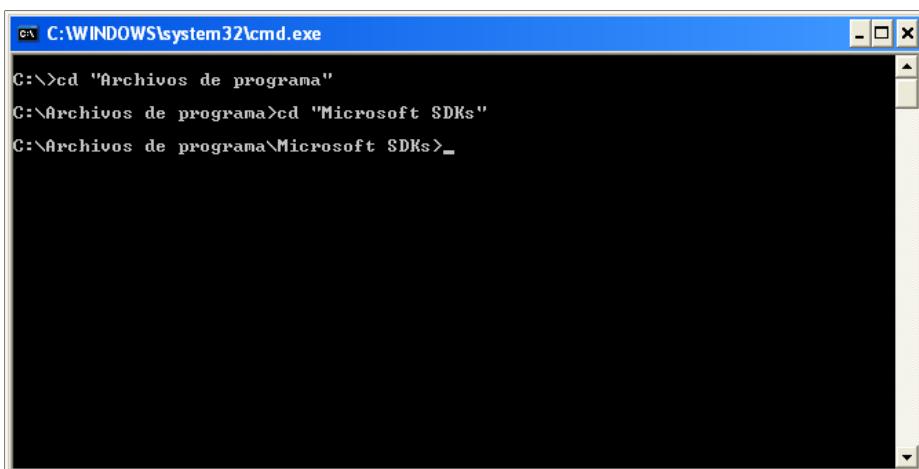
Otra manera de generar nuestros propios *DataSets tipados* es mediante la línea de comandos de MS-DOS con el uso de la herramienta **xsd.exe**.

A continuación veremos como realizar esta tarea con este comando.

Usando la herramienta XSD.exe

XSD no es otra cosa que un fichero ejecutable, que encontraremos en el SDK de .NET y que nos permitirá crear nuestros *DataSet tipados* de manera rápida y sencilla.

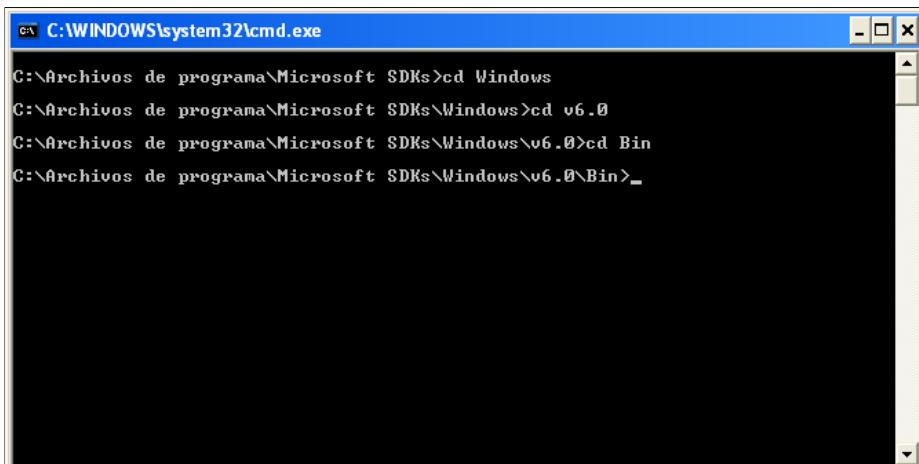
Para llevar a cabo nuestra tarea, abra una ventana MS-DOS y síntese en el directorio en el cuál tenemos instalado nuestro entorno *Visual Studio 2010*, como se indica en la figura 1.



Ventana de MS-DOS situada en el directorio de Visual Studio 2010

Figura 1

Una vez situados sobre este directorio, nos moveremos al subdirectorio **SDK\v2.0\Bin** tal y como se indica en la figura 2.



Ventana de MS-DOS situada en el subdirectorio Bin de Visual Studio 2010

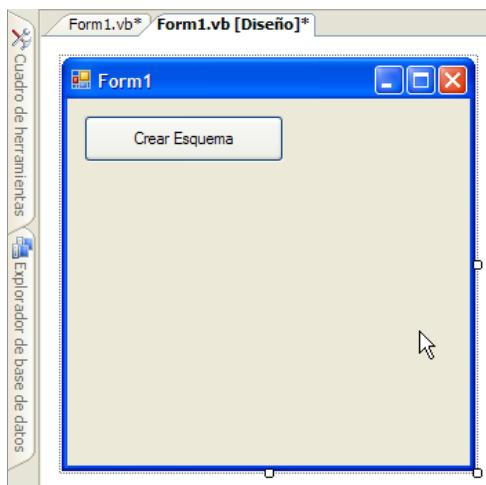
Figura 2

A continuación, lo que deberemos hacer es el fichero *xsd* correspondiente.

Para eso, deberemos hacerlo mediante un editor de textos o bien, desde una pequeña aplicación Windows que generaremos para tal propósito.

En nuestro caso lo haremos con una aplicación Windows, para lo cuál, crearemos un nuevo proyecto de **Aplicación para Windows**.

En el formulario Windows, insertaremos un control **Button** tal y como se muestra en la figura 3.



Control Button insertado en el formulario de generación del esquema

Figura 3

A continuación, escribiremos el siguiente código que lo que nos permitirá, será crear el fichero o documento XML de extensión *xsd* correspondiente, el cuál contiene la información para generar el *DataSet tipado*.

El código de la aplicación de generación del esquema es el que se detalla a continuación:

```
Código

Imports System.Data
Imports System.Data.SqlClient
Imports System.Xml

Public Class Form1

    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button1.Click
        ' Establecemos los parámetros de la conexión
        Dim MiSqlConnection As New SqlConnection("server=.;uid=sa;password=VisualBasic;database=MSDNVideo")
        ' Indicamos la sentencia SELECT para la conexión anterior
        Dim MiSqlDataAdapter As New SqlDataAdapter("SELECT TOP 1 * FROM SOCIOS", MiSqlConnection)
        ' Declaramos un objeto DataSet para almacenar el contenido
        Dim MiDataSet As New DataSet()
        ' Rellenamos el DataSet
        MiSqlDataAdapter.Fill(MiDataSet, "SOCIOS")
        ' Declaramos un objeto DataTable
        Dim MiDataTable As DataTable = MiDataSet.Tables("SOCIOS")
        ' Completamos algunos parámetros de los campos de la tabla
        MiDataTable.Columns("NIF").Unique = True
        MiDataTable.Columns("Nombre").MaxLength = 50
        MiDataTable.Columns("Apellido1").MaxLength = 50
        MiDataTable.Columns("Apellido2").MaxLength = 50
        MiDataTable.Columns("Telefono").MaxLength = 13
        MiDataTable.Columns("Email").MaxLength = 50
        MiDataTable.Columns("Direccion").MaxLength = 100
        MiDataTable.Columns("Ciudad").MaxLength = 50
        MiDataTable.Columns("Provincia").MaxLength = 50
        MiDataTable.Columns("CP").MaxLength = 50
        ' Escribimos el esquema de la tabla al disco duro del PC
        MiDataSet.WriteXmlSchema("c:\MiEsquema.xsd")
        ' Cerramos la conexión
        MiSqlConnection.Close()
        ' Mostramos un mensaje de esquema creado correctamente
        MessageBox.Show("Esquema creado correctamente")
    End Sub
End Class
```

Una vez que hemos escrito el código fuente necesario para generar el esquema, ejecutaremos nuestra aplicación, obteniendo una ejecución afirmativa como la que se indica en la figura 4.



Ejecución del programa de generación del esquema

Figura 4

El código de nuestro esquema, será una vez que se ha creado, como se indica en el siguiente código:

```

Código
<?xml version="1.0" standalone="yes"?>
<xss:schema id="NewDataSet" xmlns="" xmlns:xss="http://www.w3.org/2001/XMLSchema" xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
<xss:element name="NewDataSet" msdata:IsDataSet="true" msdata:UseCurrentLocale="true">
<xss:complexType>
<xss:choice minOccurs="0" maxOccurs="unbounded">
<xss:element name="SOCIOS">
<xss:complexType>
<xss:sequence>
<xss:element name="NIF" type="xs:string" minOccurs="0" />
<xss:element name="Nombre" minOccurs="0">
<xss:simpleType>
<xss:restriction base="xs:string">
<xss:maxLength value="50" />
</xss:restriction>
</xss:simpleType>
</xss:element>
<xss:element name="Apellido1" minOccurs="0">
<xss:simpleType>
<xss:restriction base="xs:string">
<xss:maxLength value="50" />
</xss:restriction>
</xss:simpleType>
</xss:element>
<xss:element name="Apellido2" minOccurs="0">
<xss:simpleType>
<xss:restriction base="xs:string">
<xss:maxLength value="50" />
</xss:restriction>
</xss:simpleType>
</xss:element>
<xss:element name="Telefono" minOccurs="0">
<xss:simpleType>
<xss:restriction base="xs:string">
<xss:maxLength value="13" />
</xss:restriction>
</xss:simpleType>
</xss:element>
<xss:element name="Email" minOccurs="0">
<xss:simpleType>
<xss:restriction base="xs:string">
<xss:maxLength value="50" />
</xss:restriction>
</xss:simpleType>
</xss:element>
<xss:element name="Direccion" minOccurs="0">
<xss:simpleType>
<xss:restriction base="xs:string">
<xss:maxLength value="100" />
</xss:restriction>
</xss:simpleType>
</xss:element>
<xss:element name="Ciudad" minOccurs="0">
<xss:simpleType>
<xss:restriction base="xs:string">
<xss:maxLength value="50" />
</xss:restriction>
</xss:simpleType>
</xss:element>
<xss:element name="Provincia" minOccurs="0">
<xss:simpleType>
<xss:restriction base="xs:string">
<xss:maxLength value="50" />
</xss:restriction>
</xss:simpleType>
</xss:element>
<xss:element name="CP" minOccurs="0">
<xss:simpleType>
<xss:restriction base="xs:string">
<xss:maxLength value="50" />
</xss:restriction>
</xss:simpleType>
</xss:element>
</xss:sequence>

```

```

</xs:complexType>
</xs:element>
</xs:choice>
</xs:complexType>
<xs:unique name="Constraint1">
  <xs:selector xpath=".//SOCIOS" />
  <xs:field xpath="NIF" />
</xs:unique>
</xs:element>
</xs:schema>

```

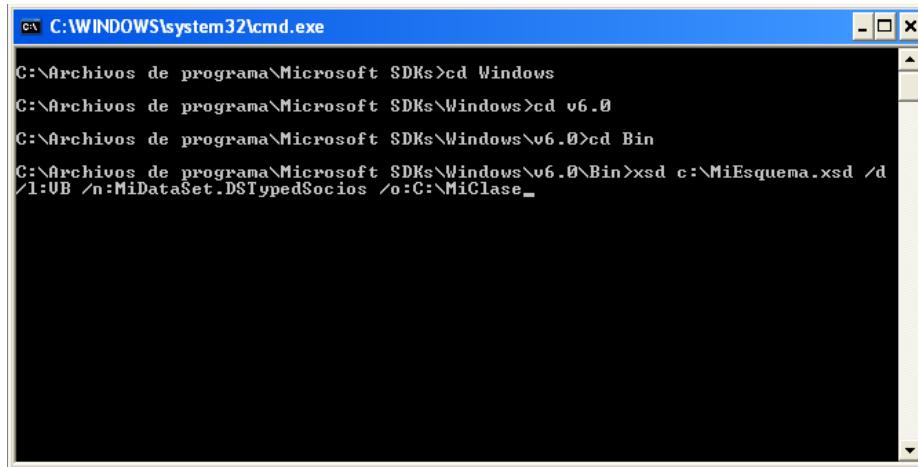
Lo que haremos a continuación, será utilizar la herramienta **xsd.exe** que está dentro del directorio **SDK\v2.0\Bin**.

Con la herramienta **xsd.exe**, generaremos la clase que podremos usar para el *DataSet tipados*.

Una vez por lo tanto, que estamos s tuados en la línea de comandos de MS-DOS, escribiremos la instrucción:

```
xsd c:\MiEsquema.xsd /d /l:VB /n:MiDataSet.DSTypedSocios /o:c:\
```

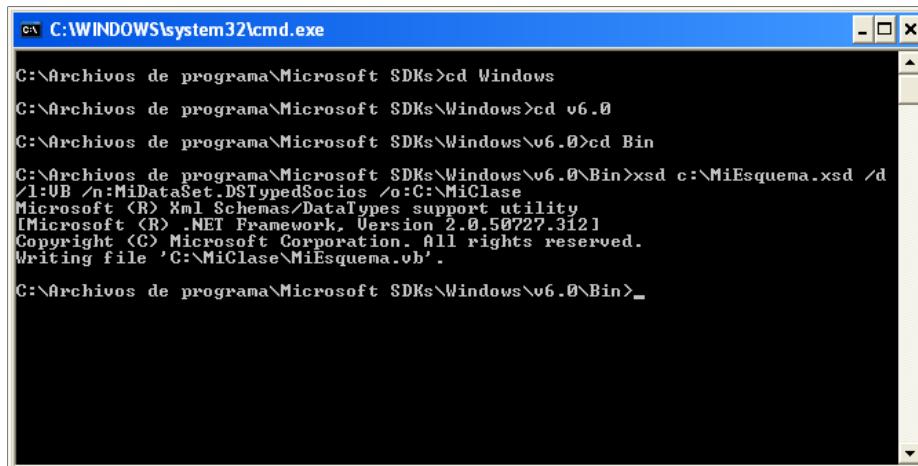
Esto lo haremos tal y como se indica en la figura 5.



Comandos de creación de la clase del esquema creado

Figura 5

Si todo ha ido correctamente, la clase para usar el *DataSet tipado*, se habrá creado como se indica en la figura 6.



Clase de DataSet tipado creada con la aplicación xsd.exe

Figura 6

El código de nuestra clase tendrá un aspecto similar al que se indica en el siguiente código fuente:

Código

```

' <auto-generated>
' Una herramienta generó este código.
' Versión del motor en tiempo de ejecución:2.0.50215.44
'
' Los cambios en este archivo podrían causar un comportamiento incorrecto y se perderán si
' el código se vuelve a generar.
' </auto-generated>

Option Strict Off
Option Explicit On

Imports System

'
'Este código fuente fue generado automáticamente por xsd, Versión=2.0.50215.44.

```

```

Namespace MiDataSet.DSTypedSocios

<Serializable(), _
System.ComponentModel.DesignerCategoryAttribute("code"), _
System.ComponentModel.ToolboxItem(true), _
System.Xml.Serialization.XmlSchemaProviderAttribute("GetTypedDataSetSchema"), _
System.Xml.Serialization.XmlRootAttribute("NewDataSet"), _
System.ComponentModel.Design.HelpKeywordAttribute("vs.data.DataSet"),
System.Diagnostics.CodeAnalysis.SuppressMessageAttribute("Microsoft.Usage", "CA2240:ImplementISerializableCorrectly")> _
Partial Public Class NewDataSet
    Inherits System.Data.DataSet

    Private tableSOCIOS As SOCIOSDataTable

    Private _schemaSerializationMode As System.Data.SchemaSerializationMode = System.Data.SchemaSerializationMode.IncludeSchema

<System.Diagnostics.CodeAnalysis.SuppressMessageAttribute("Microsoft.Usage", "CA2214:DoNotCallOverridableMethodsInConstructors")>
Public Sub New()
    MyBase.New
    Me.BeginInit
    Me.InitClass
    Dim schemaChangedHandler As System.ComponentModel.CollectionChangeEventHandler = AddressOf Me.SchemaChanged
    AddHandler MyBase.Tables.CollectionChanged, schemaChangedHandler
    AddHandler MyBase.Relations.CollectionChanged, schemaChangedHandler
    Me.EndInit
End Sub

<System.Diagnostics.CodeAnalysis.SuppressMessageAttribute("Microsoft.Usage", "CA2214:DoNotCallOverridableMethodsInConstructors")>
Protected Sub New(ByVal info As System.Runtime.Serialization.SerializationInfo, ByVal context As System.Runtime.Serialization.Str
    MyBase.New(info, context)
    If (Me.IsBinarySerialized(info, context) = true) Then
        Me.InitVars(false)
        Dim schemaChangedHandler1 As System.ComponentModel.CollectionChangeEventHandler = AddressOf Me.SchemaChanged
        AddHandler Me.Tables.CollectionChanged, schemaChangedHandler1
        AddHandler Me.Relations.CollectionChanged, schemaChangedHandler1
        Return
    End If
    Dim strSchema As String = CType(info.GetValue("XmlSchema", GetType(String)), String)
    If (Me.DetermineSchemaSerializationMode(info, context) = System.Data.SchemaSerializationMode.IncludeSchema) Then
        Dim ds As System.Data.DataSet = New System.Data.DataSet
        ds.ReadXmlSchema(New System.Xml.XmlTextReader(New System.IO.StringReader(strSchema)))
        If (Not (ds.Tables("SOCIOS")) Is Nothing) Then
            MyBase.Tables.Add(New SOCIOSDataTable(ds.Tables("SOCIOS")))
        End If
        Me.DataSetName = ds.DataSetName
        Me.Prefix = ds.Prefix
        Me.Namespace = ds.Namespace
        Me.Locale = ds.Locale
        Me.CaseSensitive = ds.CaseSensitive
        Me.EnforceConstraints = ds.EnforceConstraints
        Me.Merge(ds, false, System.Data.MissingSchemaAction.Add)
        Me.InitVars
    Else
        Me.ReadXmlSchema(New System.Xml.XmlTextReader(New System.IO.StringReader(strSchema)))
    End If
    Me.GetSerializationData(info, context)
    Dim schemaChangedHandler As System.ComponentModel.CollectionChangeEventHandler = AddressOf Me.SchemaChanged
    AddHandler MyBase.Tables.CollectionChanged, schemaChangedHandler
    AddHandler Me.Relations.CollectionChanged, schemaChangedHandler
End Sub

<System.ComponentModel.Browsable(false), _
System.ComponentModel.DesignerSerializationVisibility(System.ComponentModel.DesignerSerializationVisibility.Content)> _
Public ReadOnly Property SOCIOS() As SOCIOSDataTable
    Get
        Return Me.tableSOCIOS
    End Get
End Property

Public Overrides Property SchemaSerializationMode() As System.Data.SchemaSerializationMode
    Get
        Return Me._schemaSerializationMode
    End Get
    Set
        Me._schemaSerializationMode = value
    End Set
End Property

<System.ComponentModel.DesignerSerializationVisibilityAttribute(System.ComponentModel.DesignerSerializationVisibility.Hidden)> _
Public Shadows ReadOnly Property Tables() As System.Data.DataTableCollection
    Get
        Return MyBase.Tables
    End Get
End Property

<System.ComponentModel.DesignerSerializationVisibilityAttribute(System.ComponentModel.DesignerSerializationVisibility.Hidden)> _
Public Shadows ReadOnly Property Relations() As System.Data.DataRelationCollection
    Get
        Return MyBase.Relations
    End Get
End Property

<System.ComponentModel.DefaultValueAttribute(true)> _
Public Shadows Property EnforceConstraints() As Boolean
    Get
        Return MyBase.EnforceConstraints
    End Get
    Set
        MyBase.EnforceConstraints = value
    End Set
End Property

Protected Overrides Sub InitializeDerivedDataSet()

```

```

Me.BeginInit
Me.InitClass
Me.EndInit
End Sub

Public Overrides Function Clone() As System.Data.DataSet
    Dim cln As NewDataSet = CType(MyBase.Clone, NewDataSet)
    cln.InitVars
    Return cln
End Function

Protected Overrides Function ShouldSerializeTables() As Boolean
    Return false
End Function

Protected Overrides Function ShouldSerializeRelations() As Boolean
    Return false
End Function

Protected Overrides Sub ReadXmlSerializable(ByVal reader As System.Xml.XmlReader)
    If (Me.DetermineSchemaSerializationMode(reader) = System.Data.SchemaSerializationMode.IncludeSchema) Then
        Me.Reset
        Dim ds As System.Data.DataSet = New System.Data.DataSet
        ds.ReadXml(reader)
        If (Not (ds.Tables("SOCIOS")) Is Nothing) Then
            MyBase.Tables.Add(New SOCIOSDataTable(ds.Tables("SOCIOS")))
        End If
        Me.DataSetName = ds.DataSetName
        Me.Prefix = ds.Prefix
        Me.Namespace = ds.Namespace
        Me.Locale = ds.Locale
        Me.CaseSensitive = ds.CaseSensitive
        Me.EnforceConstraints = ds.EnforceConstraints
        Me.Merge(ds, false, System.Data.MissingSchemaAction.Add)
        Me.InitVars
    Else
        Me.ReadXml(reader)
        Me.InitVars
    End If
End Sub

Protected Overrides Function GetSchemaSerializable() As System.Xml.Schema.XmlSchema
    Dim stream As System.IO.MemoryStream = New System.IO.MemoryStream
    Me.WriteXmlSchema(New System.Xml.XmlTextWriter(stream, Nothing))
    stream.Position = 0
    Return System.Xml.Schema.XmlSchema.Read(New System.Xml.XmlTextReader(stream), Nothing)
End Function

Friend Overloads Sub InitVars()
    Me.InitVars(true)
End Sub

Friend Overloads Sub InitVars(ByVal initTable As Boolean)
    Me.tableSOCIOS = CType(MyBase.Tables("SOCIOS"), SOCIOSDataTable)
    If (initTable = true) Then
        If (Not (Me.tableSOCIOS) Is Nothing) Then
            Me.tableSOCIOS.InitVars
        End If
    End If
End Sub

Private Sub InitClass()
    Me.DataSetName = "NewDataSet"
    Me.Prefix = ""
    Me.EnforceConstraints = true
    Me.tableSOCIOS = New SOCIOSDataTable
    MyBase.Tables.Add(Me.tableSOCIOS)
End Sub

Private Function ShouldSerializeSOCIOS() As Boolean
    Return false
End Function

Private Sub SchemaChanged(ByVal sender As Object, ByVal e As System.ComponentModel.CollectionChangeEventArgs)
    If (e.Action = System.ComponentModel.CollectionChangeAction.Remove) Then
        Me.InitVars
    End If
End Sub

Public Shared Function GetTypedDataSetSchema(ByVal xs As System.Xml.Schema.XmlSchemaSet) As System.Xml.Schema.XmlSchemaComplexType
    Dim ds As NewDataSet = New NewDataSet
    Dim type As System.Xml.Schema.XmlSchemaComplexType = New System.Xml.Schema.XmlSchemaComplexType
    Dim sequence As System.Xml.Schema.XmlSchemaSequence = New System.Xml.Schema.XmlSchemaSequence
    xs.Add(ds.GetSchemaSerializable)
    Dim any As System.Xml.Schema.XmlSchemaAny = New System.Xml.Schema.XmlSchemaAny
    any.Namespace = ds.Namespace
    sequence.Items.Add(any)
    type.Particle = sequence
    Return type
End Function

Public Delegate Sub SOCIOSRowChangeEventHandler(ByVal sender As Object, ByVal e As SOCIOSRowChangeEvent)

<System.Serializable(), _
System.Xml.Serialization.XmlSchemaProviderAttribute("GetTypedTableSchema")> -
Partial Public Class SOCIOSDataTable
    Inherits System.Data.DataTable
    Implements System.Collections.IEnumerable

    Private columnNIF As System.Data.DataColumn
    Private columnNombre As System.Data.DataColumn
    Private columnApellido As System.Data.DataColumn

```

```

Private columnApellido2 As System.Data.DataColumn
Private columnTelefono As System.Data.DataColumn
Private columnEmail As System.Data.DataColumn
Private columnDireccion As System.Data.DataColumn
Private columnCiudad As System.Data.DataColumn
Private columnProvincia As System.Data.DataColumn
Private columnCP As System.Data.DataColumn

Public Sub New()
    MyBase.New
    Me.TableName = "SOCIOS"
    Me.BeginInit
    Me.InitClass
    Me.EndInit
End Sub

Friend Sub New(ByVal table As System.Data.DataTable)
    MyBase.New
    Me.TableName = table.TableName
    If (table.CaseSensitive <> table.DataSet.CaseSensitive) Then
        Me.CaseSensitive = table.CaseSensitive
    End If
    If (table.Locale.ToString <> table.DataSet.Locale.ToString) Then
        Me.Locale = table.Locale
    End If
    If (table.Namespace <> table.DataSet.Namespace) Then
        Me.Namespace = table.Namespace
    End If
    Me.Prefix = table.Prefix
    Me.MinimumCapacity = table.MinimumCapacity
End Sub

Protected Sub New(ByVal info As System.Runtime.Serialization.SerializationInfo, ByVal context As System.Runtime.Serialization.IFormatter)
    MyBase.New(info, context)
    Me.InitVars
End Sub

Public ReadOnly Property NIFColumn() As System.Data.DataColumn
    Get
        Return Me.columnNIF
    End Get
End Property

Public ReadOnly Property NombreColumn() As System.Data.DataColumn
    Get
        Return Me.columnNombre
    End Get
End Property

Public ReadOnly Property Apellido1Column() As System.Data.DataColumn
    Get
        Return Me.columnApellido1
    End Get
End Property

Public ReadOnly Property Apellido2Column() As System.Data.DataColumn
    Get
        Return Me.columnApellido2
    End Get
End Property

Public ReadOnly Property TelefonoColumn() As System.Data.DataColumn
    Get
        Return Me.columnTelefono
    End Get
End Property

Public ReadOnly Property EmailColumn() As System.Data.DataColumn
    Get
        Return Me.columnEmail
    End Get
End Property

Public ReadOnly Property DireccionColumn() As System.Data.DataColumn
    Get
        Return Me.columnDireccion
    End Get
End Property

Public ReadOnly Property CiudadColumn() As System.Data.DataColumn
    Get
        Return Me.columnCiudad
    End Get
End Property

Public ReadOnly Property ProvinciaColumn() As System.Data.DataColumn
    Get
        Return Me.columnProvincia
    End Get
End Property

Public ReadOnly Property CPColumn() As System.Data.DataColumn
    Get
        Return Me.columnCP
    End Get
End Property

```

```

<System.ComponentModel.Browsable(false)> _
Public ReadOnly Property Count() As Integer
    Get
        Return Me.Rows.Count
    End Get
End Property

Public Default ReadOnly Property Item(ByVal index As Integer) As SOCIOSRow
    Get
        Return CType(Me.Rows(index),SOCIOSRow)
    End Get
End Property

Public Event SOCIOSRowChanged As SOCIOSRowChangeEventHandler
Public Event SOCIOSRowChanging As SOCIOSRowChangeEventHandler
Public Event SOCIOSRowDeleted As SOCIOSRowChangeEventHandler
Public Event SOCIOSRowDeleting As SOCIOSRowChangeEventHandler

Public Overloads Sub AddSOCIOSRow(ByVal row As SOCIOSRow)
    Me.Rows.Add(row)
End Sub

Public Overloads Function AddSOCIOSRow(ByVal NIF As String, ByVal Nombre As String, ByVal Apellido1 As String, ByVal Apellido2 As String)
    Dim rowSOCIOSRow As SOCIOSRow = CType(Me.NewRow,SOCIOSRow)
    rowSOCIOSRow.ItemArray = New Object() {NIF, Nombre, Apellido1, Apellido2, Telefono, Email, Direccion, Ciudad, Provincia, CP}
    Me.Rows.Add(rowSOCIOSRow)
    Return rowSOCIOSRow
End Function

Public Overrides Function GetEnumerator() As System.Collections.IEnumerator Implements System.Collections.IEnumerable.GetEnumerator
    Return Me.Rows.GetEnumerator
End Function

Public Overrides Function Clone() As System.Data.DataTable
    Dim cln As SOCIOSDataTable = CType(MyBase.Clone,SOCIOSDataTable)
    cln.InitVars
    Return cln
End Function

Protected Overrides Function CreateInstance() As System.Data.DataTable
    Return New SOCIOSDataTable
End Function

Friend Sub InitVars()
    Me.columnNIF = MyBase.Columns("NIF")
    Me.columnNombre = MyBase.Columns("Nombre")
    Me.columnApellido1 = MyBase.Columns("Apellido1")
    Me.columnApellido2 = MyBase.Columns("Apellido2")
    Me.columnTelefono = MyBase.Columns("Telefono")
    Me.columnEmail = MyBase.Columns("Email")
    Me.columnDireccion = MyBase.Columns("Direccion")
    Me.columnCiudad = MyBase.Columns("Ciudad")
    Me.columnProvincia = MyBase.Columns("Provincia")
    Me.columnCP = MyBase.Columns("CP")
End Sub

Private Sub InitClass()
    Me.columnNIF = New System.Data.DataColumn("NIF", GetType(String), Nothing, System.Data.MappingType.Element)
    MyBase.Columns.Add(Me.columnNIF)
    Me.columnNombre = New System.Data.DataColumn("Nombre", GetType(String), Nothing, System.Data.MappingType.Element)
    MyBase.Columns.Add(Me.columnNombre)
    Me.columnApellido1 = New System.Data.DataColumn("Apellido1", GetType(String), Nothing, System.Data.MappingType.Element)
    MyBase.Columns.Add(Me.columnApellido1)
    Me.columnApellido2 = New System.Data.DataColumn("Apellido2", GetType(String), Nothing, System.Data.MappingType.Element)
    MyBase.Columns.Add(Me.columnApellido2)
    Me.columnTelefono = New System.Data.DataColumn("Telefono", GetType(String), Nothing, System.Data.MappingType.Element)
    MyBase.Columns.Add(Me.columnTelefono)
    Me.columnEmail = New System.Data.DataColumn("Email", GetType(String), Nothing, System.Data.MappingType.Element)
    MyBase.Columns.Add(Me.columnEmail)
    Me.columnDireccion = New System.Data.DataColumn("Direccion", GetType(String), Nothing, System.Data.MappingType.Element)
    MyBase.Columns.Add(Me.columnDireccion)
    Me.columnCiudad = New System.Data.DataColumn("Ciudad", GetType(String), Nothing, System.Data.MappingType.Element)
    MyBase.Columns.Add(Me.columnCiudad)
    Me.columnProvincia = New System.Data.DataColumn("Provincia", GetType(String), Nothing, System.Data.MappingType.Element)
    MyBase.Columns.Add(Me.columnProvincia)
    Me.columnCP = New System.Data.DataColumn("CP", GetType(String), Nothing, System.Data.MappingType.Element)
    MyBase.Columns.Add(Me.columnCP)
    Me.Constraints.Add(New System.Data.UniqueConstraint("Constraint1", New System.Data.DataColumn() {Me.columnNIF}, false))
    Me.columnNIF.Unique = true
    Me.columnNombre.MaxLength = 50
    Me.columnApellido1.MaxLength = 50
    Me.columnApellido2.MaxLength = 50
    Me.columnTelefono.MaxLength = 13
    Me.columnEmail.MaxLength = 50
    Me.columnDireccion.MaxLength = 100
    Me.columnCiudad.MaxLength = 50
    Me.columnProvincia.MaxLength = 50
    Me.columnCP.MaxLength = 50
End Sub

Public Function NewSOCIOSRow() As SOCIOSRow
    Return CType(Me.NewRow,SOCIOSRow)
End Function

Protected Overrides Function NewRowFromBuilder(ByVal builder As System.Data.DataRowBuilder) As System.Data.DataRow
    Return New SOCIOSRow(builder)
End Function

Protected Overrides Function GetRowType() As System.Type
    Return GetType(SOCIOSRow)
End Function

```

```

Protected Overrides Sub OnRowChanged(ByVal e As System.Data.DataRowChangeEventArgs)
    MyBase.OnRowChanged(e)
    If (Not (Me.SOCIOSRowChangedEvent) Is Nothing) Then
        RaiseEvent SOCIOSRowChanged(Me, New SOCIOSRowChangeEvent(CType(e.Row,SOCIOSRow), e.Action))
    End If
End Sub

Protected Overrides Sub OnRowChanging(ByVal e As System.Data.DataRowChangeEventArgs)
    MyBase.OnRowChanging(e)
    If (Not (Me.SOCIOSRowChangingEvent) Is Nothing) Then
        RaiseEvent SOCIOSRowChanging(Me, New SOCIOSRowChangeEvent(CType(e.Row,SOCIOSRow), e.Action))
    End If
End Sub

Protected Overrides Sub OnRowDeleted(ByVal e As System.Data.DataRowChangeEventArgs)
    MyBase.OnRowDeleted(e)
    If (Not (Me.SOCIOSRowDeletedEvent) Is Nothing) Then
        RaiseEvent SOCIOSRowDeleted(Me, New SOCIOSRowChangeEvent(CType(e.Row,SOCIOSRow), e.Action))
    End If
End Sub

Protected Overrides Sub OnRowDeleting(ByVal e As System.Data.DataRowChangeEventArgs)
    MyBase.OnRowDeleting(e)
    If (Not (Me.SOCIOSRowDeletingEvent) Is Nothing) Then
        RaiseEvent SOCIOSRowDeleting(Me, New SOCIOSRowChangeEvent(CType(e.Row,SOCIOSRow), e.Action))
    End If
End Sub

Public Sub RemoveSOCIOSRow(ByVal row As SOCIOSRow)
    Me.Rows.Remove(row)
End Sub

Public Shared Function GetTypedTableSchema(ByVal xs As System.Xml.Schema.XmlSchemaSet) As System.Xml.Schema.XmlSchemaComplexType
    Dim type As System.Xml.Schema.XmlSchemaComplexType = New System.Xml.Schema.XmlSchemaComplexType
    Dim sequence As System.Xml.Schema.XmlSchemaSequence = New System.Xml.Schema.XmlSchemaSequence
    Dim ds As NewDataSet = New NewDataSet
    xs.Add(ds.GetSchemaSerializable)
    Dim any1 As System.Xml.Schema.XmlSchemaAny = New System.Xml.Schema.XmlSchemaAny
    any1.Namespace = "http://www.w3.org/2001/XMLSchema"
    any1.MinOccurs = New Decimal(0)
    any1.MaxOccurs = Decimal.MaxValue
    any1.ProcessContents = System.Xml.Schema.XmlSchemaContentProcessing.Lax
    sequence.Items.Add(any1)
    Dim any2 As System.Xml.Schema.XmlSchemaAny = New System.Xml.Schema.XmlSchemaAny
    any2.Namespace = "urn:schemas-microsoft-com:xml-diffgram-v1"
    any2.MinOccurs = New Decimal(1)
    any2.ProcessContents = System.Xml.Schema.XmlSchemaContentProcessing.Lax
    sequence.Items.Add(any2)
    Dim attribute1 As System.Xml.Schema.XmlSchemaAttribute = New System.Xml.Schema.XmlSchemaAttribute
    attribute1.Name = "namespace"
    attribute1.FixedValue = ds.Namespace
    type.Attributes.Add(attribute1)
    Dim attribute2 As System.Xml.Schema.XmlSchemaAttribute = New System.Xml.Schema.XmlSchemaAttribute
    attribute2.Name = "tableName"
    attribute2.FixedValue = "SOCIOSDataTable"
    type.Attributes.Add(attribute2)
    type.Particle = sequence
    Return type
End Function
End Class

Partial Public Class SOCIOSRow
    Inherits System.Data.DataRow

    Private tableSOCIOS As SOCIOSDataTable

    Friend Sub New(ByVal rb As System.Data.DataRowBuilder)
        MyBase.New(rb)
        Me.tableSOCIOS = CType(Me.Table, SOCIOSDataTable)
    End Sub

    Public Property NIF() As String
        Get
            Try
                Return CType(Me(Me.tableSOCIOS.NIFColumn), String)
            Catch e As System.InvalidCastException
                Throw New System.Data.StrongTypingException("El valor de la columna 'NIF' de la tabla 'SOCIOS' es DBNull.", e)
            End Try
        End Get
        Set
            Me(Me.tableSOCIOS.NIFColumn) = value
        End Set
    End Property

    Public Property Nombre() As String
        Get
            Try
                Return CType(Me(Me.tableSOCIOS.NombreColumn), String)
            Catch e As System.InvalidCastException
                Throw New System.Data.StrongTypingException("El valor de la columna 'Nombre' de la tabla 'SOCIOS' es DBNull.", e)
            End Try
        End Get
        Set
            Me(Me.tableSOCIOS.NombreColumn) = value
        End Set
    End Property

    Public Property Apellido1() As String
        Get
            Try
                Return CType(Me(Me.tableSOCIOS.Apellido1Column), String)
            Catch e As System.InvalidCastException
                Throw New System.Data.StrongTypingException("El valor de la columna 'Apellido1' de la tabla 'SOCIOS' es DBNull.", e)
            End Try
        End Get
        Set
            Me(Me.tableSOCIOS.Apellido1Column) = value
        End Set
    End Property

```

```

        End Try
    End Get
    Set
        Me(Me.tableSOCIOS.Apellido1Column) = value
    End Set
End Property

Public Property Apellido2() As String
Get
    Try
        Return CType(Me(Me.tableSOCIOS.Apellido2Column),String)
    Catch e As System.InvalidCastException
        Throw New System.Data.StrongTypingException("El valor de la columna 'Apellido2' de la tabla 'SOCIOS' es DBNull.", e)
    End Try
End Get
Set
    Me(Me.tableSOCIOS.Apellido2Column) = value
End Set
End Property

Public Property Telefono() As String
Get
    Try
        Return CType(Me(Me.tableSOCIOS.TelefonoColumn),String)
    Catch e As System.InvalidCastException
        Throw New System.Data.StrongTypingException("El valor de la columna 'Telefono' de la tabla 'SOCIOS' es DBNull.", e)
    End Try
End Get
Set
    Me(Me.tableSOCIOS.TelefonoColumn) = value
End Set
End Property

Public Property Email() As String
Get
    Try
        Return CType(Me(Me.tableSOCIOS.EmailColumn),String)
    Catch e As System.InvalidCastException
        Throw New System.Data.StrongTypingException("El valor de la columna 'Email' de la tabla 'SOCIOS' es DBNull.", e)
    End Try
End Get
Set
    Me(Me.tableSOCIOS.EmailColumn) = value
End Set
End Property

Public Property Direccion() As String
Get
    Try
        Return CType(Me(Me.tableSOCIOS.DireccionColumn),String)
    Catch e As System.InvalidCastException
        Throw New System.Data.StrongTypingException("El valor de la columna 'Direccion' de la tabla 'SOCIOS' es DBNull.", e)
    End Try
End Get
Set
    Me(Me.tableSOCIOS.DireccionColumn) = value
End Set
End Property

Public Property Ciudad() As String
Get
    Try
        Return CType(Me(Me.tableSOCIOS.CiudadColumn),String)
    Catch e As System.InvalidCastException
        Throw New System.Data.StrongTypingException("El valor de la columna 'Ciudad' de la tabla 'SOCIOS' es DBNull.", e)
    End Try
End Get
Set
    Me(Me.tableSOCIOS.CiudadColumn) = value
End Set
End Property

Public Property Provincia() As String
Get
    Try
        Return CType(Me(Me.tableSOCIOS.ProvinciaColumn),String)
    Catch e As System.InvalidCastException
        Throw New System.Data.StrongTypingException("El valor de la columna 'Provincia' de la tabla 'SOCIOS' es DBNull.", e)
    End Try
End Get
Set
    Me(Me.tableSOCIOS.ProvinciaColumn) = value
End Set
End Property

Public Property CP() As String
Get
    Try
        Return CType(Me(Me.tableSOCIOS.CPColumn),String)
    Catch e As System.InvalidCastException
        Throw New System.Data.StrongTypingException("El valor de la columna 'CP' de la tabla 'SOCIOS' es DBNull.", e)
    End Try
End Get
Set
    Me(Me.tableSOCIOS.CPColumn) = value
End Set
End Property

Public Function IsNIFNull() As Boolean
    Return Me.IsNotNull(Me.tableSOCIOS.NIFColumn)
End Function

Public Sub SetNIFNull()
    Me(Me.tableSOCIOS.NIFColumn) = System.Convert.DBNull

```

```

End Sub

Public Function IsNombreNull() As Boolean
    Return Me.IsNull(Me.tableSOCIOS.NombreColumn)
End Function

Public Sub SetNombreNull()
    Me(Me.tableSOCIOS.NombreColumn) = System.Convert.DBNull
End Sub

Public Function IsApellido1Null() As Boolean
    Return Me.IsNull(Me.tableSOCIOS.Apellido1Column)
End Function

Public Sub SetApellido1Null()
    Me(Me.tableSOCIOS.Apellido1Column) = System.Convert.DBNull
End Sub

Public Function IsApellido2Null() As Boolean
    Return Me.IsNull(Me.tableSOCIOS.Apellido2Column)
End Function

Public Sub SetApellido2Null()
    Me(Me.tableSOCIOS.Apellido2Column) = System.Convert.DBNull
End Sub

Public Function IsTelefonoNull() As Boolean
    Return Me.IsNull(Me.tableSOCIOS.TelefonoColumn)
End Function

Public Sub SetTelefonoNull()
    Me(Me.tableSOCIOS.TelefonoColumn) = System.Convert.DBNull
End Sub

Public Function IsEmailNull() As Boolean
    Return Me.IsNull(Me.tableSOCIOS.EmailColumn)
End Function

Public Sub SetEmailNull()
    Me(Me.tableSOCIOS.EmailColumn) = System.Convert.DBNull
End Sub

Public Function IsDireccionNull() As Boolean
    Return Me.IsNull(Me.tableSOCIOS.DireccionColumn)
End Function

Public Sub SetDireccionNull()
    Me(Me.tableSOCIOS.DireccionColumn) = System.Convert.DBNull
End Sub

Public Function IsCiudadNull() As Boolean
    Return Me.IsNull(Me.tableSOCIOS.CiudadColumn)
End Function

Public Sub SetCiudadNull()
    Me(Me.tableSOCIOS.CiudadColumn) = System.Convert.DBNull
End Sub

Public Function IsProvinciaNull() As Boolean
    Return Me.IsNull(Me.tableSOCIOS.ProvinciaColumn)
End Function

Public Sub SetProvinciaNull()
    Me(Me.tableSOCIOS.ProvinciaColumn) = System.Convert.DBNull
End Sub

Public Function IsCPNull() As Boolean
    Return Me.IsNull(Me.tableSOCIOS.CPColumn)
End Function

Public Sub SetCPNull()
    Me(Me.tableSOCIOS.CPColumn) = System.Convert.DBNull
End Sub
End Class

Public Class SOCIOSRowChangeEvent
    Inherits System.EventArgs

    Private eventRow As SOCIOSRow

    Private eventAction As System.Data.DataRowAction

    Public Sub New(ByVal row As SOCIOSRow, ByVal action As System.Data.DataRowAction)
        MyBase.New
        Me.eventRow = row
        Me.eventAction = action
    End Sub

    Public ReadOnly Property Row() As SOCIOSRow
        Get
            Return Me.eventRow
        End Get
    End Property

    Public ReadOnly Property Action() As System.Data.DataRowAction
        Get
            Return Me.eventAction
        End Get
    End Property
End Class
End Class
End Namespace

```

En este punto, podemos incluir el archivo *MiEsquema.vb* a nuestro proyecto, o bien compilarlo desde la línea de comandos o desde el entorno de desarrollo. Sin embargo, no realizaremos esa acción. Tan solo comentaré que si quisiéramos compilar desde la línea de comandos el fichero con el compilador de Visual Basic, utilizaríamos el fichero ejecutable **vbc.exe**, que corresponde con las iniciales de *Visual Basic Compiler*, y que por defecto estará en la carpeta C:\WINDOWS\Microsoft.NETFramework\v4.0.

Ahora que ya sabemos como generar nuestros *DataSet tipados*, aprenderemos a usarlos en nuestras aplicaciones.

Lección 4: DataSet tipados

- ¿Qué son los DataSets tipados?
 - Generando nuestros DataSets tipados
 - Generando un DataSet tipado con Visual Studio
 - Generando un DataSet tipado con la línea de comandos
- Usando los DataSets tipados

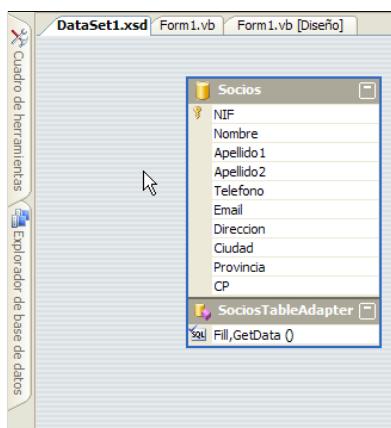
Módulo 5 - Capítulo 4

5. Usando los DataSets tipados

A continuación veremos un ejemplo del uso de *DataSets tipados* utilizando Visual Studio 2010.

Para esto nos servirá lo que ya hemos visto en la generación de un *DataSet tipado* con Visual Studio 2010.

Recordemos que ya habíamos añadido un elemento *DataSet1.xsd* a nuestro proyecto, tal y como se muestra en la figura 1.



DataSet1.xsd de ejemplo, añadido al formulario, para utilizarlo como DataSet tipado
Figura 1

Uso rápido de nuestro DataSet tipado

El uso de un *DataSet tipado*, no tiene muchas diferencias respecto a un *DataSet no tipado*. Este último ya lo hemos visto, y el *DataSet tipado* lo veremos a continuación de forma práctica.

Este primer ejemplo, muestra de forma sencilla, como trabajar con el esquema que hemos creado y como hacerlo rápidamente a través de nuestra aplicación de prueba.

El código de ejemplo que nos sirve de toma de contacto, es el que se indica a continuación:

```
Código

Imports system.data
Imports System.Data.SqlClient
Imports System.Xml

Public Class Form1

    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button1.Click
        ' Establecemos la conexión
        Dim MiSqlConnection As New SqlConnection("server=.;uid=sa;password=VisualBasic;database=MSDNVideo")
        ' Declaramos un objeto DataAdapter y le indicamos la conexión
        Dim MiSqlDataAdapter As New SqlDataAdapter("SELECT * FROM SOCIOS", MiSqlConnection)
        ' Declaramos un objeto DataSet con el esquema del DataSet tipado
        Dim MiDtTyped As New DataSet
        ' Rellenamos el DataSet tipado con la información de la tabla del SELECT
        MiSqlDataAdapter.Fill(MiDtTyped, "SOCIOS")
        ' Declaramos un objeto para recorrer los datos del DataSet
        Dim MisDatos As DataSet.SociosRow
        ' Recorremos los datos del DataSet tipado y los mostramos
        For Each MisDatos In MiDtTyped.Socios
            MessageBox.Show(MisDatos.Nombre & " " & MisDatos.Apellido1 & " " & MisDatos.Apellido2)
        Next
    End Sub
End Class
```

Nuestro ejemplo en ejecución, es el que se puede ver en la figura 2.



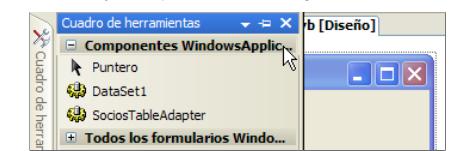
Ejemplo en ejecución del uso sencillo de DataSets tipados
Figura 2

A continuación veremos otros tipos de ejecución de *DataSets tipados* mucho más complejos.

Atención especial al Cuadro de herramientas

Cuando trabajamos con *DataSets tipados* como lo hemos hecho hasta ahora, habremos notado que entre otras cosas, tenemos las capacidades o posibilidades de trabajar con el *DataSet* como objetos.

Obviamente, estos objetos están incluidos en el *Cuadro de herramientas*, tal y como puede verse en la figura 3.

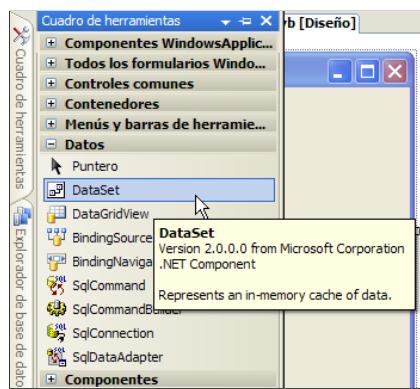


Componentes creados por el entorno para trabajar con DataSets tipados
Figura 3

Para trabajar con ellos, podemos arrastrarlos sobre el formulario como hacemos con cualquier control o componente.

Usando las herramientas automáticas para trabajar con DataSets tipados

Aún así, vamos a arrastrar sobre el formulario, un componente *DataSet* como se indica en la figura 4.

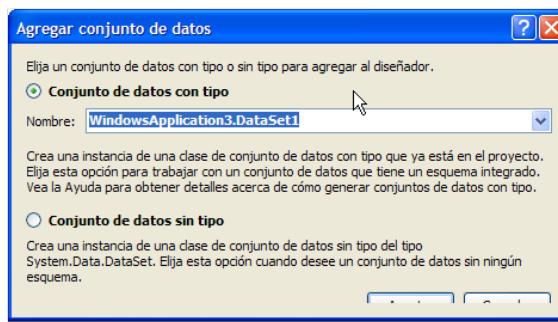


Selección de un control *DataSet* del Cuadro de herramientas

Figura 4

En este punto, recordemos que tenemos nuestro *DataSet tipado* o esquema ya creado y que para usar este esquema desde nuestro objeto *DataSet*, podemos utilizar las herramientas del entorno .NET.

Cuando arrastramos el componente *DataSet* sobre el formulario, aparecerá una ventana como la que se muestra en la figura 5, y que nos permitirá indicar si se trata de un *DataSet tipado* o un *DataSet no tipado*.



Ventana para agregar un conjunto de datos

Figura 5

Por defecto, aparecerá seleccionada la opción de *Conjunto de datos con tipo* y el *DataSet* o esquema que hemos creado.

Presionaremos el botón **Aceptar** y de esta manera, nuestro objeto *DataSet* habrá quedado insertado y preparado en el formulario, para utilizar el esquema del *DataSet* indicado.

Para no complicarlo, he decidido renombrar el control *DataSet* como *dtSet*.

El DataSet quedará insertado en nuestro entorno como se indica en la figura 6.

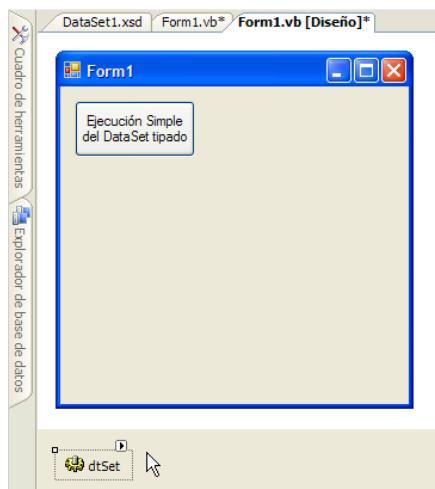


Figura 6

Para usar el objeto *DataSet* insertado, deberemos acceder a él a través de código, de forma muy parecida a lo que lo hacíamos anteriormente.

```
Código

Imports system.data
Imports System.Data.SqlClient
Imports System.Xml

Public Class Form1

    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button1.Click
        ' Establecemos la conexión
        Dim MiSqlConnection As New SqlConnection("server=.;uid=sa;password=VisualBasic;database=MSDNVideo")
        ' Declaramos un objeto DataAdapter y le indicamos la conexión
        Dim MiSqlDataAdapter As New SqlDataAdapter("SELECT * FROM SOCIOS", MiSqlConnection)
        ' Rellenamos el DataSet tipado con la información de la tabla del SELECT
        MiSqlDataAdapter.Fill(dtSet, "SOCIOS")
        ' Declaramos un objeto para recorrer los datos del DataSet
        Dim MisDatos As DataSet1.SociosRow
        ' Recorremos los datos del DataSet tipado y los mostramos
        For Each MisDatos In dtSet.Socios
            MessageBox.Show(MisDatos.Nombre & " " & MisDatos.Apellido1 & " " & MisDatos.Apellido2)
        Next
    End Sub
End Class
```

El funcionamiento de nuestra aplicación de ejemplo, es igual al que hemos visto en la figura 2.

Usando DataAdapter con DataSets tipados

Escribiremos a continuación una pequeña aplicación Windows como la hemos hecho anteriormente.

Crearemos nuestro esquema como lo hemos hecho hasta ahora y añadiremos dos botones como se indica en la figura 7. *DataSet* como se indica en la figura 4.

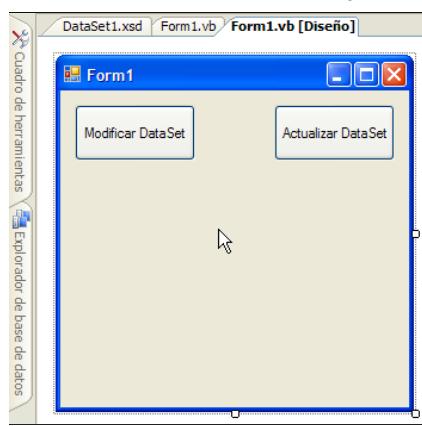
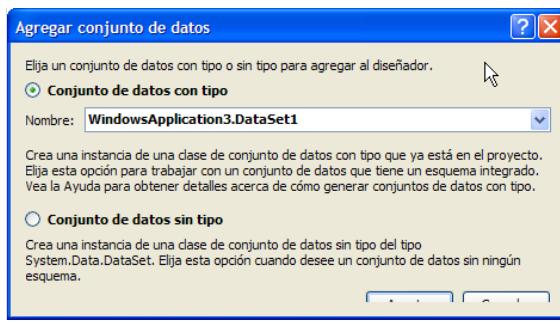


Figura 7

A continuación añadiremos un *DataSet* al cuál le asignaremos el nombre del *DataSet tipado* correspondiente al esquema creado, tal y como se indica en la figura 8.



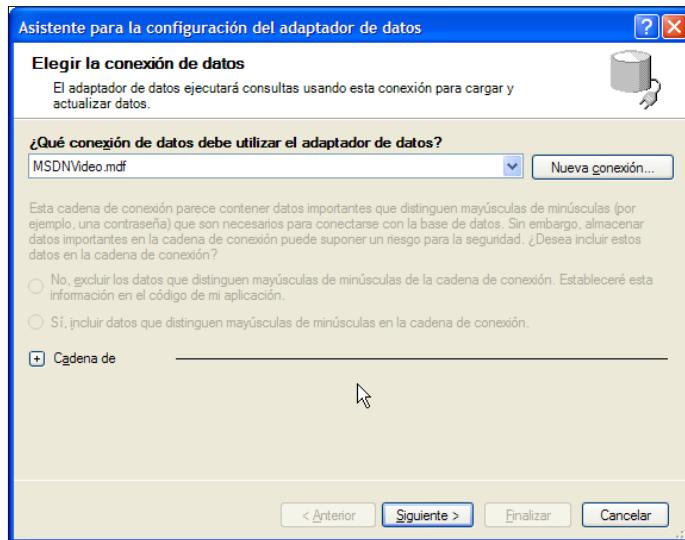
DataSet tipado del esquema asignado al objeto DataSet

Figura 8

A este objeto *DataSet*, le he llamado *dtSet*.

A continuación, añadiremos un componente *SqlDataAdapter* al formulario Windows.

En este punto, aparecerá una ventana como la que se muestra en la figura 9.

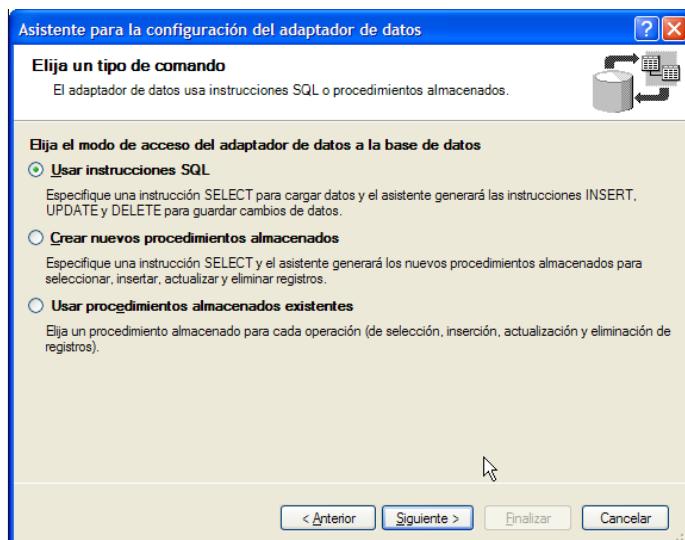


Asistente del SqlDataAdapter

Figura 9

Elegiremos la conexión adecuada y presionaremos el botón **Siguiente**.

A continuación, el asistente nos mostrará una información como la que se indica en la figura 10.



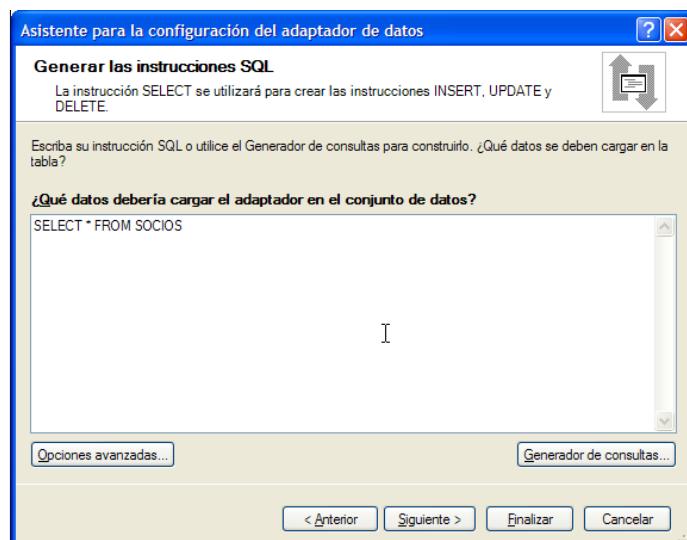
Opción para elegir el tipo de comando a utilizar

Figura 10

Elegiremos la primera de las opciones que es la que se indica en la figura 10, y presionaremos nuevamente el botón **Siguiente**.

El asistente continuará presentándonos ventanas de información para configurar nuestro componente *SqlDataAdapter*.

Dentro de esta ventana y dentro de los datos que deseamos cargar, escribiremos la instrucción SQL *SELECT * FROM SOCIOS* como se indica en la figura 11.



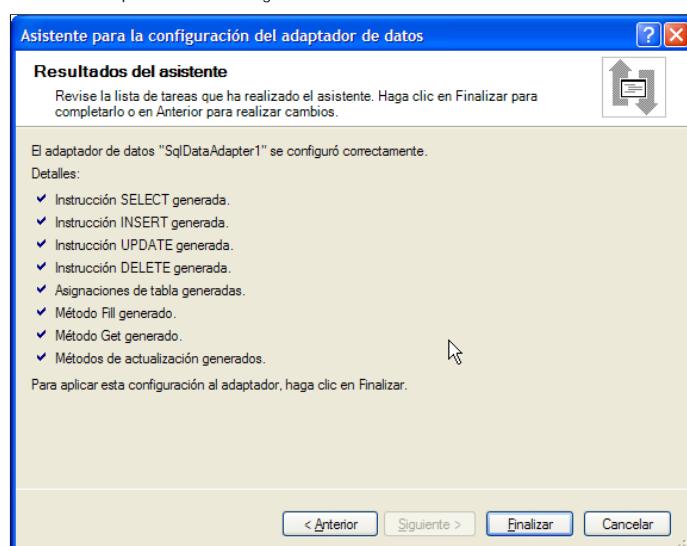
Opción de generación de las instrucciones SQL

Figura 11

A continuación, presionaremos el botón **Siguiente**.

De esta manera, el asistente terminará de crear las instrucciones necesarias para trabajar con el componente *SqlDataAdapter*.

Si todo ha ido correctamente, aparecerá una ventana como la que se indica en la figura 12.

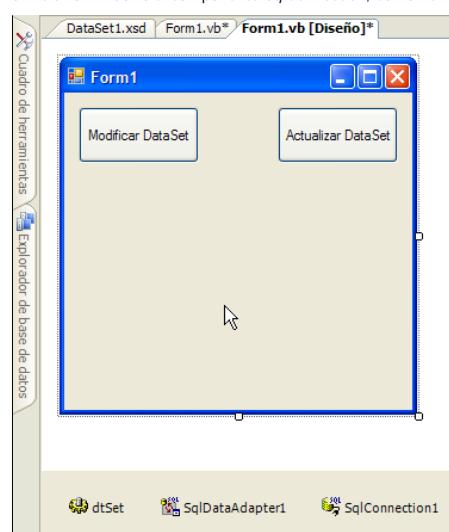


Resultados de la configuración del componente *SqlDataAdapter*

Figura 12

Para concluir, haremos clic sobre el botón **Finalizar**.

Automáticamente, en *Visual Studio 2010*, aparecerá añadido al formulario Windows el componente *SqlConnection*, como vemos en la figura 13.



Componentes añadidos al formulario Windows

Figura 13

A continuación, escribiremos el código fuente de la aplicación, que se encarga de recoger los datos de la base de datos, insertarlos en un *DataSet*, para modificarlos y actualizar las modificaciones en la base de datos.

El código fuente de la aplicación, quedará como se detalla a continuación:

```
Código

Imports System.Data
Imports System.Data.SqlClient
Imports System.Xml

Public Class Form1
    Dim MiSqlConnection As SqlConnection
    Dim MiSqlDataAdapter As SqlDataAdapter

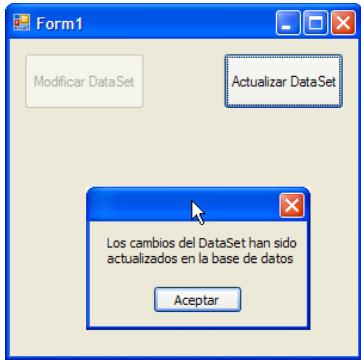
    Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.Load
        ' Rellenamos el DataSet tipado con la información de la tabla del SELECT
        SqlDataAdapter1.Fill(dtSet, "SOCIOS")
    End Sub

    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button1.Click
        ' Declararemos un objeto para trabajar con los datos del DataSet
        Dim MisDatos As DataSet.SociosRow
        ' Almacenamos en él, la información del DataSet para
        ' el NIF = "111111"
        MisDatos = dtSet.Socios.FindByNIF("111111")
        ' Modificaremos el campo CP
        MisDatos.CP = "28022"
        ' Deshabilitamos como medida de seguridad el botón
        Button1.Enabled = False
    End Sub

    Private Sub Button2_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button2.Click
        If dtSet.HasChanges Then
            ' El DataSet tiene cambios
            ' Declaramos un objeto DataSet
            Dim dtSetModificado As DataSet
            ' Le pasamos los datos modificados en el DataSet original
            dtSetModificado = dtSet.GetChanges
            ' Actualizamos el DataSet que ha cambiado a través del DataAdapter
            SqlDataAdapter1.Update(dtSetModificado)
            ' Aceptamos los cambios en el DataSet para
            ' seguir trabajando con él por ejemplo
            dtSet.AcceptChanges()
            ' Mostramos un mensaje en pantalla indicando que
            ' se han modificado los datos
            MessageBox.Show("Los cambios del DataSet han sido" & vbCrLf & "actualizados en la base de datos")
            ' Deshabilitamos como medida de seguridad el botón
            Button2.Enabled = False
        Else
            ' El DataSet no tiene cambios
            MessageBox.Show("No hay cambios en el DataSet")
        End If
    End Sub

End Class
```

Nuestro ejemplo en ejecución, es el que puede verse en la figura 14.

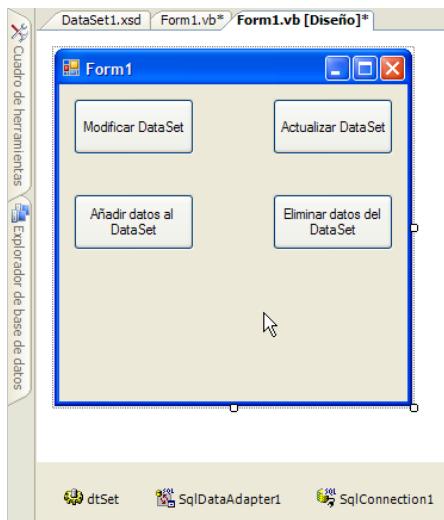


Ejemplo del DataAdapter y DataSet tipado en ejecución
Figura 14

Nota:

Cuando trabajamos con un componente *DataAdapter*, aparece un asistente. En el caso de elegir la opción de generación de sentencias SELECT, UPDATE, DELETE e INSERT, el asistente creará a partir de la SELECT, esas instrucciones por nosotros.

Dentro de este mismo ejemplo, añada otros dos botones como se muestra en la figura 15.



Ejemplo del DataAdapter y DataSet tipado en ejecución

Figura 15

Estos botones nos servirán para añadir una fila nueva a nuestra base de datos y para eliminar una fila existente (en nuestro caso la nueva fila añadida).

El código fuente de nuestra aplicación de ejemplo, es el que se detalla a continuación:

```
Código

Imports system.data
Imports System.Data.SqlClient
Imports System.Xml

Public Class Form1
    Dim MiSqlConnection As SqlConnection
    Dim MiSqlDataAdapter As SqlDataAdapter

    Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.Load
        ' Deshabilitamos el botón de Borrar registro,
        ' porque hasta que no se cree no lo borraremos
        Button4.Enabled = False
        ' Rellenamos el DataSet tipado con la información de la tabla del SELECT
        SqlDataAdapter1.Fill(dtSet, "SOCIOS")
    End Sub

    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button1.Click
        ' Declaramos un objeto para trabajar con los datos del DataSet
        Dim MisDatos As DataSet.SociosRow
        ' Almacenamos en él, la información del DataSet para
        ' el NIF = "111111"
        MisDatos = dtSet.Socios.FindByNIF("111111")
        ' Modificaremos el campo CP
        MisDatos.CP = "28022"
        ' Deshabilitamos como medida de seguridad el botón
        Button1.Enabled = False
    End Sub

    Private Sub Button2_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button2.Click
        If dtSet.HasChanges Then
            ' El DataSet tiene cambios
            ' Declaramos un objeto DataSet
            Dim dtSetModificado As DataSet
            ' Le pasamos los datos modificados en el DataSet original
            dtSetModificado = dtSet.GetChanges
            ' Actualizamos el DataSet que ha cambiado a través del DataAdapter
            SqlDataAdapter1.Update(dtSetModificado)
            ' Aceptamos los cambios en el DataSet para
            ' seguir trabajando con él por ejemplo
            dtSet.AcceptChanges()
            ' Mostramos un mensaje en pantalla indicando que
            ' se han modificado los datos
            MessageBox.Show("Los cambios del DataSet han sido" & vbCrLf & "actualizados en la base de datos")
            ' Deshabilitamos como medida de seguridad el botón
            Button2.Enabled = False
        Else
            ' El DataSet no tiene cambios
            MessageBox.Show("No hay cambios en el DataSet")
        End If
    End Sub

    Private Sub Button3_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button3.Click
        ' Añadimos una nueva fila con datos al DataSet tipado
        dtSet.Socios.AddSociosRow("111112", "María", "Sánchez", "Rodríguez", "1231234", "maria@cuantadecorreo.com", "C\ San Fernando, 13", "Barcelona", "Barcelon
        ' Actualizamos los datos a través del DataAdapter
        SqlDataAdapter1.Update(dtSet)
        ' Deshabilitamos el botón de añadir datos
        Button3.Enabled = False
        ' Habilitamos el botón de eliminar datos
        Button4.Enabled = True
        ' Mostramos un mensaje indicando que la
        ' acción se ha realizado correctamente
        MessageBox.Show("Registro añadido correctamente")
    End Sub

    Private Sub Button4_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button4.Click
        ' Borramos la fila añadida
        dtSet.Socios.FindByNIF("111112").Delete()
        ' Actualizamos los datos a través del DataAdapter
        SqlDataAdapter1.Update(dtSet)
        ' Mostramos un mensaje indicando que la
        ' acción se ha realizado correctamente
        MessageBox.Show("Registro eliminado correctamente")
        ' Deshabilitamos el botón de eliminar
        Button4.Enabled = False
    End Sub
```

Nuestro ejemplo en ejecución es el que se muestra en la figura 16.



Ejemplo del uso de DataAdapter y DataSet tipado con todas las acciones de inserción y eliminación de datos incluidas

Figura 16

Como vemos, el uso del *DataSet tipado*, posee grandes ventajas sobre los *DataSets no tipados*, permitiéndonos trabajar con datos de forma muy rápida y sencilla. La parte más compleja quizás, es la preparación del esquema de datos, pero una vez realizada esta tarea, el trabajo con los datos es asombrosamente rápida y segura.

Lección 5: Enlace a formularios

- **¿Qué son los datos maestro detalle?**
- **Configurando la fuente de datos**
- **Preparando el origen de datos**
- **Incrustando los datos maestro detalle**
- **Manipulando los datos maestro detalle**

Introducción

Visual Studio 2010 proporciona a los desarrolladores un conjunto de herramientas y asistentes que facilitan enormemente el trabajo y ahorran grandes cantidades de tiempo en el desarrollo de diferentes aplicaciones.

De los asistentes o herramientas que nos proporcionan mayor rendimiento en el entorno de desarrollo rápido de Microsoft, está la que nos permite trabajar con fuentes de datos como los datos de una aplicación típica maestro detalle.

Entre otras cosas, nos centraremos en esta acción, que será la que veremos en detalle a continuación.

Módulo 5 - Capítulo 5

- 1. [¿Que son los datos maestro detalle?](#)
- 2. [Configurando la fuente de datos](#)
- 3. [Preparando el origen de datos](#)
- 4. [Incrustando los datos maestro detalle](#)
- 5. [Manipulando los datos maestro detalle](#)

Lección 5: Enlace a formularios

¿Qué son los datos maestro detalle?

- Configurando la fuente de datos
- Preparando el origen de datos
- Incrustando los datos maestro detalle
- Manipulando los datos maestro detalle

Módulo 5 - Capítulo 5

1. ¿Qué son los datos Maestro Detalle?

El desarrollador de aplicaciones que debe trabajar con datos y fuentes de datos relacionadas entre sí, encuentra con frecuencia problemas de desarrollo en aplicaciones con datos interrelacionados.

Además, este tipo de aplicaciones, consumen gran parte del tiempo de desarrollo y son por lo general, acciones repetitivas.

Supongamos como ejemplo general, la tabla *Socios* de un videoclub.

Además, relacionemos los socios del videoclub, con una tabla *Alquileres*, para saber si un socio determinado tiene películas alquiladas, y en ese caso, saber cuáles.

Este sencillo ejemplo, es un claro exponente de una aplicación que relaciona datos maestro detalle.

Ambas tablas deben estar relacionadas para recopilar la información que se necesite en un momento dado.

Los datos maestros serían expuestos por los socios del videoclub, mientras que los datos detalle estarían relacionados con los datos de los alquileres de los socios.

En nuestro caso, vamos a cambiar las palabras maestro y detalle por padre e hijo, y a partir de ahora, nos referiremos a padre como la tabla *Socios*, e hijo como la tabla *Alquileres*. De esta forma, ubicaremos sin problemas ambos conceptos dentro del entorno de *Visual Studio 2010*, ya que éste tiene alguna ligera connotación que podría infundirnos a error como observará más adelante.

Por suerte, *Visual Studio 2010* nos proporciona un conjunto de herramientas, que hace que realizar una aplicación Windows con todas las características de una aplicación maestro detalle, sea un auténtico juego de niños, que nos llevará aproximadamente un minuto de nuestro tiempo como mucho.

¿No me cree?. Continúe el capítulo, y se sorprenderá de lo que *Visual Studio 2010* puede hacer por usted.

Lección 5: Enlace a formularios

- ¿Qué son los datos maestro detalle?
- Configurando la fuente de datos
 - Preparando el origen de datos
 - Incrustando los datos maestro detalle
 - Manipulando los datos maestro detalle

Módulo 5 - Capítulo 5

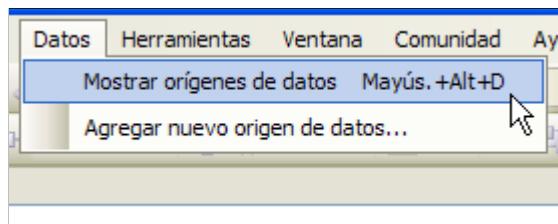
2. Configurando la fuente de datos

Trabajar con fuentes de datos requiere como tarea inicial, que tengamos listo y preparado un origen de fuentes de datos válido.

Para esta tarea, deberemos configurar la fuente de datos que vamos a utilizar, algo que vamos a aprender a hacer a continuación.

Configurando el origen de la fuente de datos

Iniciaremos una nueva aplicación Windows con *Visual Studio 2010* y seleccionaremos el menú **Datos > Mostrar orígenes de datos** como se indica en la figura 1.



Menú para mostrar los orígenes de datos

Figura 1

En este punto, aparecerá una ventana como la que se muestra en la figura 2.

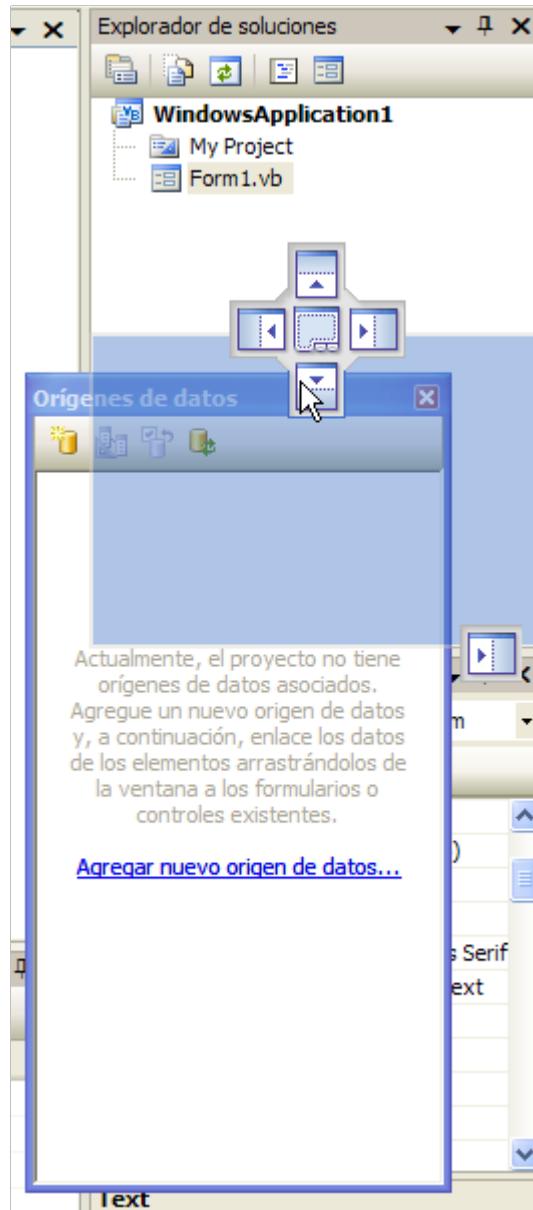


Ventana de orígenes de datos

Figura 2

Como podemos apreciar, la ventana no tiene por defecto ningún origen de datos asignado, además, esta ventana inicialmente, no está anclada a ningún sitio del entorno.

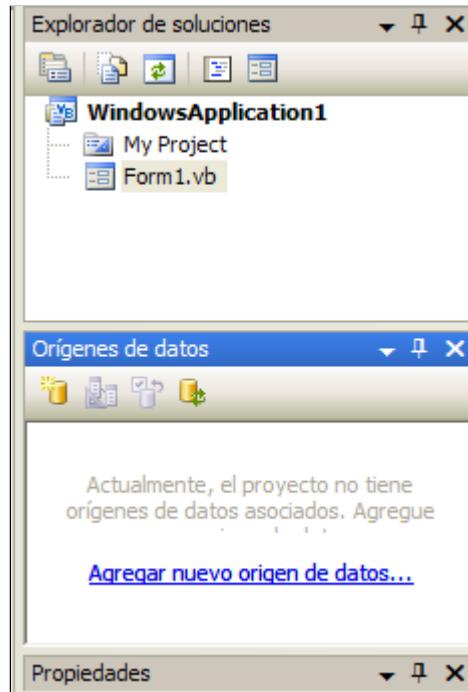
Para situarla en un lugar específico del entorno, haga clic sobre la ventana y arrástrela sobre la parte en la que desee situarla, como se indica por ejemplo, en la figura 3.



La ventana orígenes de datos podemos situarla dónde deseemos dentro de Visual Studio 2010

Figura 3

En este punto, la ventana de orígenes de datos, quedará anclada en el entorno de desarrollo, como se muestra en la figura 4.



La ventana orígenes de datos anclada en Visual Studio 2010

Figura 4

Cada uno, puede situar esta ventana dónde considere oportuno. En mi caso la he situado entre las ventanas del *Explorador de soluciones* y de *Propiedades*, pero podemos situarla dónde consideremos oportuno.

Sobre la configuración del origen de datos, haga clic sobre la opción *Agregar nuevo origen de datos...* como se indica en la figura 5.



Opción para agregar un nuevo origen de datos

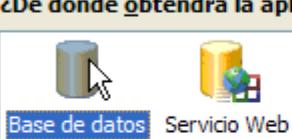
Figura 5

Aparecerá una ventana como la que se muestra en la figura 6 en la cuál indicaremos el lugar de dónde la aplicación obtendrá los datos, y que en nuestro caso será de una *Base de datos*.

Asistente para configuración de orígenes de datos



Elegir un tipo de origen de datos



Base de datos



Servicio Web



Objeto

Le permite conectarse a una base de datos y seleccionar los objetos de la base de datos para su aplicación. Esta opción crea un conjunto de datos.

< Anterior

Siguiente >

Finalizar

Cancelar

Como tipo de origen de datos elegiremos una Base de datos

Figura 6

Una vez seleccionado la opción de *Base de datos* como tipo de origen de datos, presionaremos el botón **Siguiente**.

En la siguiente ventana, elegiremos la conexión de la base de datos que vamos a utilizar, o presionaremos sobre el botón **Nueva conexión...** sino tenemos seleccionada la conexión que queremos utilizar.

En la figura 7, podemos ver como hemos establecido la conexión con nuestra fuente de datos, que utilizaremos en nuestro ejemplo de creación de la aplicación de acceso a datos maestro detalle.

Asistente para configuración de orígenes de datos



Elija la conexión de datos

¿Qué conexión de datos debería utilizar la aplicación para conectarse a la base de datos?

MSDNVideo.mdf

Nueva conexión...



Esta cadena de conexión parece contener datos importantes que distinguen mayúsculas de minúsculas (por ejemplo, una contraseña) que son necesarios para conectarse con la base de datos. Sin embargo, almacenar datos importantes en la cadena de conexión puede suponer un riesgo para la seguridad. ¿Desea incluir estos datos en la cadena de conexión?

- No, excluir los datos que distinguen mayúsculas de minúsculas de la cadena de conexión. Estableceré esta información en el código de mi aplicación.
- Sí, incluir datos que distinguen mayúsculas de minúsculas en la cadena de conexión.

Cadena de conexión: _____

< Anterior

Siguiente >

Finalizar

Cancelar

Ventana dónde elegimos la conexión de datos

Figura 7

A continuación, haremos clic en el botón **Siguiente**.

En la nueva ventana que aparece ahora dentro del asistente para la creación del origen de datos, indicaremos el nombre de la cadena de conexión que crearemos. En nuestro caso, no modificaremos el nombre de la cadena de conexión, dejándola por defecto como se muestra en la figura 8.

Asistente para configuración de orígenes de datos



Guardar cadena de conexión en el archivo de config. de la aplicación

El almacenamiento de las cadenas de conexión del archivo de configuración de aplicación facilita el mantenimiento y la implementación. Para guardar la cadena de conexión en el archivo de configuración de la aplicación, escriba un nombre en el cuadro y, a continuación, haga clic en Siguiente.

¿Desea guardar la cadena de conexión en el archivo de configuración de la aplicación?

Sí, guardar la conexión como:

MSDNVideoConnectionString



< Anterior

Siguiente >

Finalizar

Cancelar

Asignamos un nombre para el nombre de la cadena de conexión

Figura 8

A continuación, haremos clic sobre el botón **Siguiente**.

En este punto, el asistente se conecta a la base de datos para recuperar la información de la base de datos y mostrarla en la ventana del asistente como se muestra en la figura 9.

Asistente para configuración de orígenes de datos



Elija los objetos de base de datos

¿Qué objetos de la base de datos desea tener en el conjunto de datos?

- + Tablas
- + Vistas
- + Procedimientos almacenados
- + Funciones



Nombre de DataSet:

MSDNVideoDataSet

< Anterior

Siguiente >

Finalizar

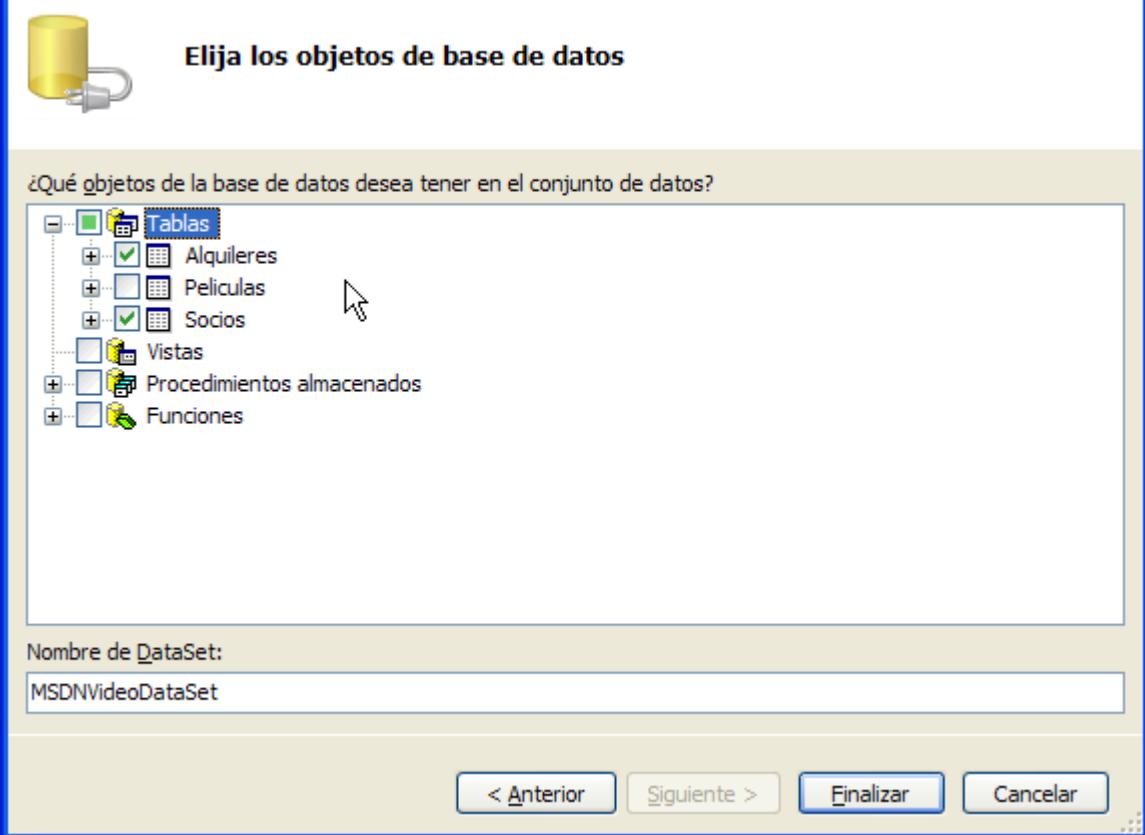
Cancelar

Ventana con los objetos de la base de datos seleccionada

Figura 9

A continuación, despliegue el objeto **Tablas** y seleccione las tablas *Alquileres* y *Socios* como se indica en la figura 10.

Asistente para configuración de orígenes de datos

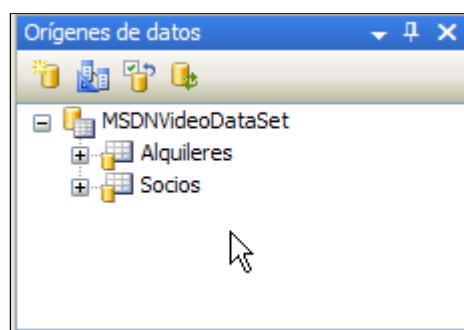


Selección de los objetos de la base de datos seleccionada

Figura 10

Una vez que hemos seleccionado los objetos de la base de datos, haremos clic sobre el botón **Finalizar** para que el asistente concluya las opciones añadidas al asistente.

La ventana del origen de datos, quedará ahora configurado de una forma similar a la que se presenta en la figura 11.



Ventana de origen de datos con la configuración añadida

Figura 11

A partir de aquí, deberemos preparar las tablas del origen de datos para que se comporten como auténticos datos e informaciones maestro detalle.

Esto es lo que veremos en el siguiente módulo.

Lección 5: Enlace a formularios

- ¿Qué son los datos maestro detalle?
 - Configurando la fuente de datos
- Preparando el origen de datos**
- Incrustando los datos maestro detalle
 - Manipulando los datos maestro detalle

Módulo 5 - Capítulo 5

3. Preparando el origen de datos

Ya hemos aprendido a añadir nuestro origen de datos, y ahora aprenderemos a prepararlo para poder utilizarlo posteriormente en la aplicación Windows.

La preparación del origen de datos, nos permitirá seleccionar que tabla o datos queremos que actúen como maestro y cuales como detalle, o dicho de otra forma, que datos queremos que sean padre y cuales hijo.

Nuestro objetivo principal es mostrar la tabla *Alquileres* como tabla hijo y la tabla *Socios* como padre de la tabla anterior.

Preparamos e incrustaremos primero la tabla *Socios* dentro del formulario Windows como *Detalle* de la información, y posteriormente insertaremos la tabla *Alquileres* dentro del formulario.

Por último, asignaremos alguna relación que permita trabajar con las dos tablas en nuestro formulario Windows sin perder la conexión entre ambas tablas y permitiéndonos acceder a la información que nos proporciona dicha relación.

Preparando la tabla padre

Lo primero que haremos será preparar la tabla padre para poderla añadir al formulario Windows.

Por esa razón, haremos clic sobre la tabla *Socios* de la ventana de *Orígenes de datos* como se indica en la figura 1.

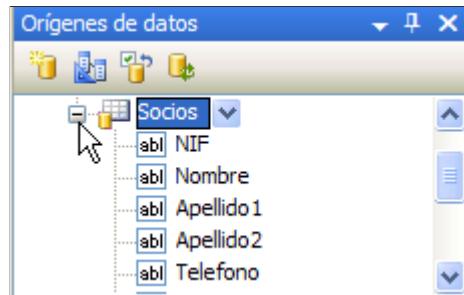
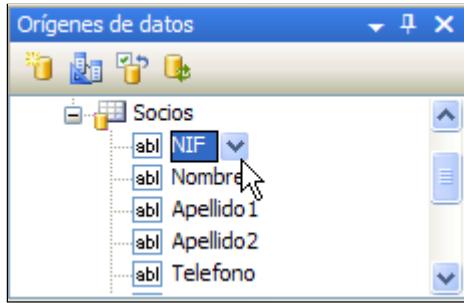


Tabla Socios de la ventana de orígenes de datos

Figura 1

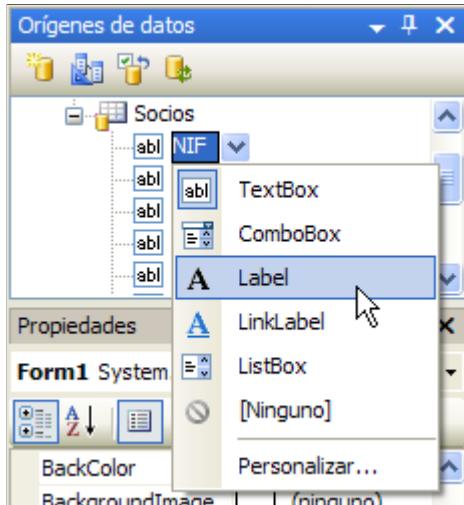
En la ventana de *Orígenes de datos* y en concreto con la tabla *Socios* desplegada, centraremos nuestra atención en el campo **NIF** como se indica en la figura 2.



Campo **NIF** de la tabla **Socios**

Figura 2

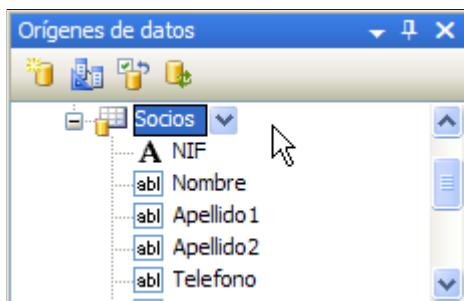
Pulse sobre la lista desplegable que aparece a la derecha del campo **NIF** y seleccione la opción *Label* como se indica en la figura 3.



Lista desplegable con la opción **Label** seleccionada como campo de la tabla desplegada

Figura 3

En este caso, la ventana de *Orígenes de datos* quedará informada tal y como se indica en la figura 4.



Campo **NIF** modificado en la ventana de **Orígenes de datos**

Figura 4

A continuación, haremos clic sobre la tabla *Socios* como se indica en la figura 5, y posteriormente presionaremos sobre la lista desplegable que aparece a la derecha de la tabla.

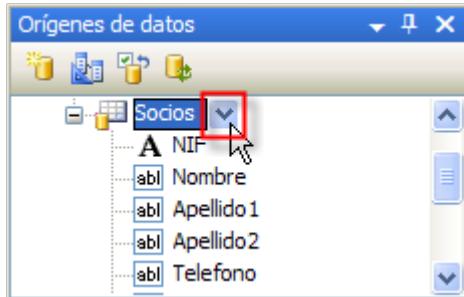
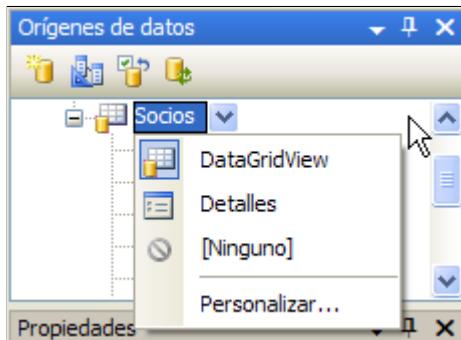


Tabla Socios seleccionada en la ventana de Orígenes de datos

Figura 5

Si hacemos clic sobre la lista desplegable, aparecerá una lista de opciones o posibilidades, para indicar cómo queremos que sean los campos de la tabla seleccionada con respecto a qué tipo de controles queremos que sean.

Esto es lo que se indica en la figura 6.



Lista desplegable de la tabla Socios de la ventana de Orígenes de datos

Figura 6

Por defecto, una tabla queda determinada con un icono que representa el control *DataGridView*, aunque se puede modificar la representación que deseamos tengan los datos dentro de un formulario seleccionando cualquiera de las opciones que tenemos de la lista desplegable.

Estas opciones pueden ser cualquiera de las siguientes:

-  Representa los datos volcados dentro de un control *DataGridView*
-  Representa los datos volcados dentro de controles estándar como *TextBox* u otros controles para reflejarla como *Detalle* de la información
-  No representa ningún control como tipo de control de volcado de datos

En el caso de la tabla *Socios* que usaremos como tabla padre, cambiaremos la representación por defecto de *DataGridView* para indicarle que nos represente la información en controles, tal y como se indica en la figura 7.

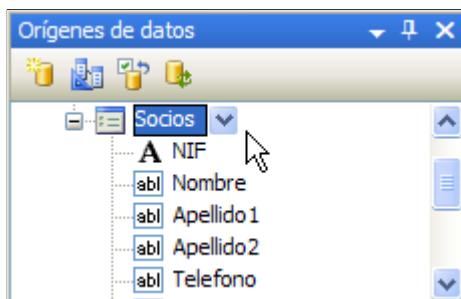


Tabla Socios seleccionada en la ventana de Orígenes de datos como Detalle

Figura 7

Ahora que tenemos la tabla maestra ya preparada, pasaremos a preparar la tabla hija.

Preparando la tabla hija

Ahora que ya tenemos preparada la tabla tabla madre, prepararemos la tabla hija de los alquileres de las películas de video, para poder usar su información dentro del formulario Windows.

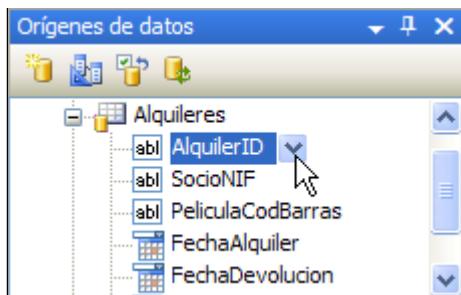
Por esa razón, haga clic sobre la tabla *Alquileres* de la ventana de *Orígenes de datos* como se indica en la figura 8.



Tabla Alquileres de la ventana de orígenes de datos

Figura 8

Dentro de la ventana de *Orígenes de datos* y en concreto de la tabla *Alquileres* desplegada, centraremos nuestra atención en el campo **AlquilerID** y **SocioNIF** como se indica en la figura 9.

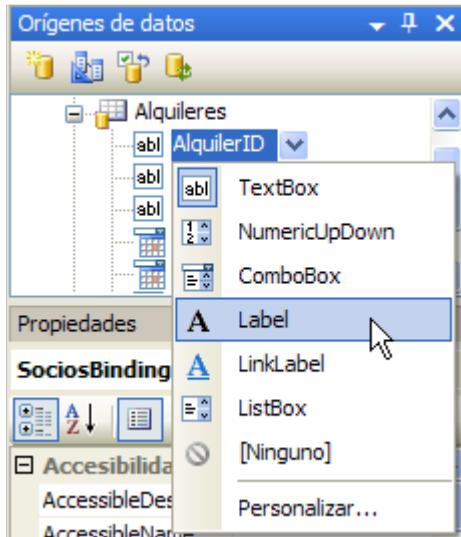


Campos AlquilerID y SocioNIF de la tabla Alquileres

Figura 9

Sobre el campo **AlquilerID** y **SocioNIF**, haremos una pequeña modificación.

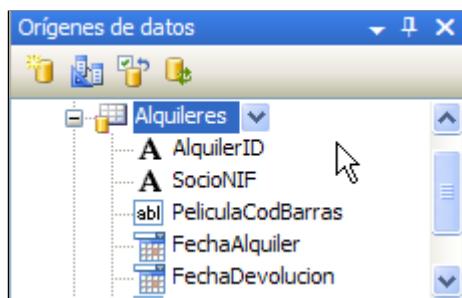
Pulse sobre la lista desplegable que aparece a la derecha del campo **AlquilerID** y **SocioNIF** y seleccione la opción *Label* como se indica en la figura 10.



Lista desplegable de opciones de un campo de la tabla desplegada

Figura 10

De esta manera, el campo **AlquilerID** y **SocioNIF** quedarán modificados en la ventana *Orígenes de datos* como se indica en la figura 11.



Campo AlquilerID y SocioNIF modificados en la ventana de Orígenes de datos

Figura 11

A continuación, haremos clic sobre la tabla *Alquileres* como se indica en la figura 12, y posteriormente presionaremos sobre la lista desplegable que aparece a la derecha de la tabla.

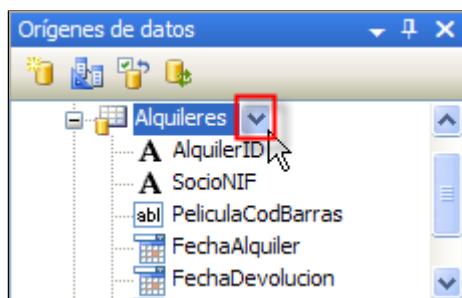
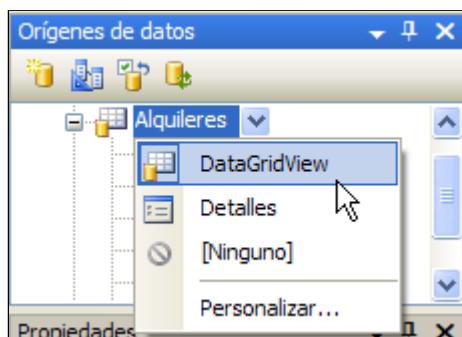


Tabla Alquileres seleccionada en la ventana de Orígenes de datos

Figura 12

Nos aseguraremos que está seleccionada la opción *DataGridView* que es la que aparece por defecto.

Esto es lo que se indica en la figura 13.



En la tabla Alquileres, nos aseguraremos de seleccionar la opción DataGridView

Figura 13

Una vez que tenemos las tabla maestra y detalle preparadas para utilizarlas, las añadiremos al formulario Windows para que tengan el funcionamiento esperado.

Lección 5: Enlace a formularios

- ¿Qué son los datos maestro detalle?
 - Configurando la fuente de datos
 - Preparando el origen de datos
- Incrustando los datos maestro detalle**
- Manipulando los datos maestro detalle

Módulo 5 - Capítulo 5

4. Incrustando los datos maestro detalle

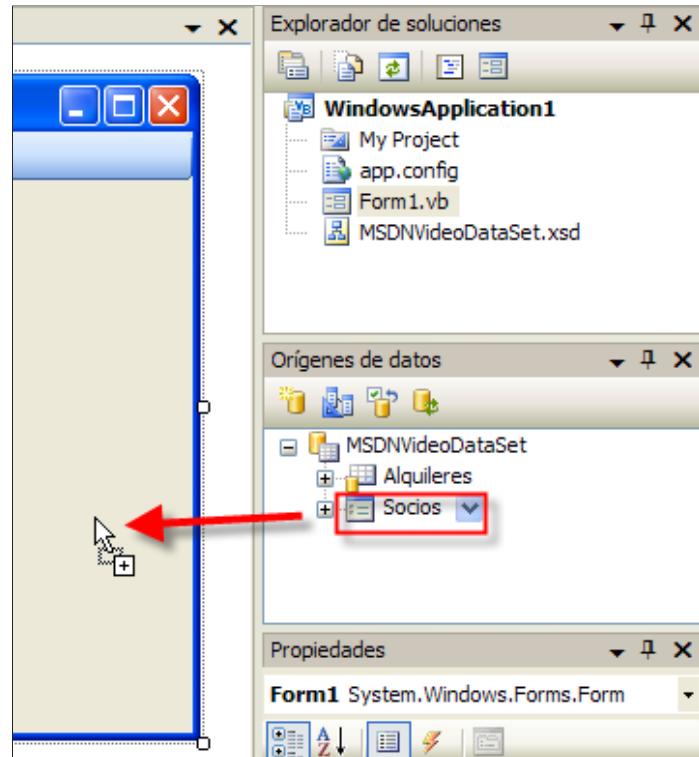
Ya sabemos como crear un origen de datos, también sabemos como preparar los datos maestro y detalle, y por último, lo que nos queda es insertar esos datos en el formulario, y relacionar todos sus datos para que funcionen de la forma esperada.

A continuación, veremos como llevar a cabo esta tarea y aprenderemos a hacerlo posible de forma muy rápida y sencilla.

Incrustando la tabla padre en el formulario

Nuestra primera acción, será incrustar en el formulario los datos o tabla padre, que en nuestro caso es la formada por la tabla *Socios*.

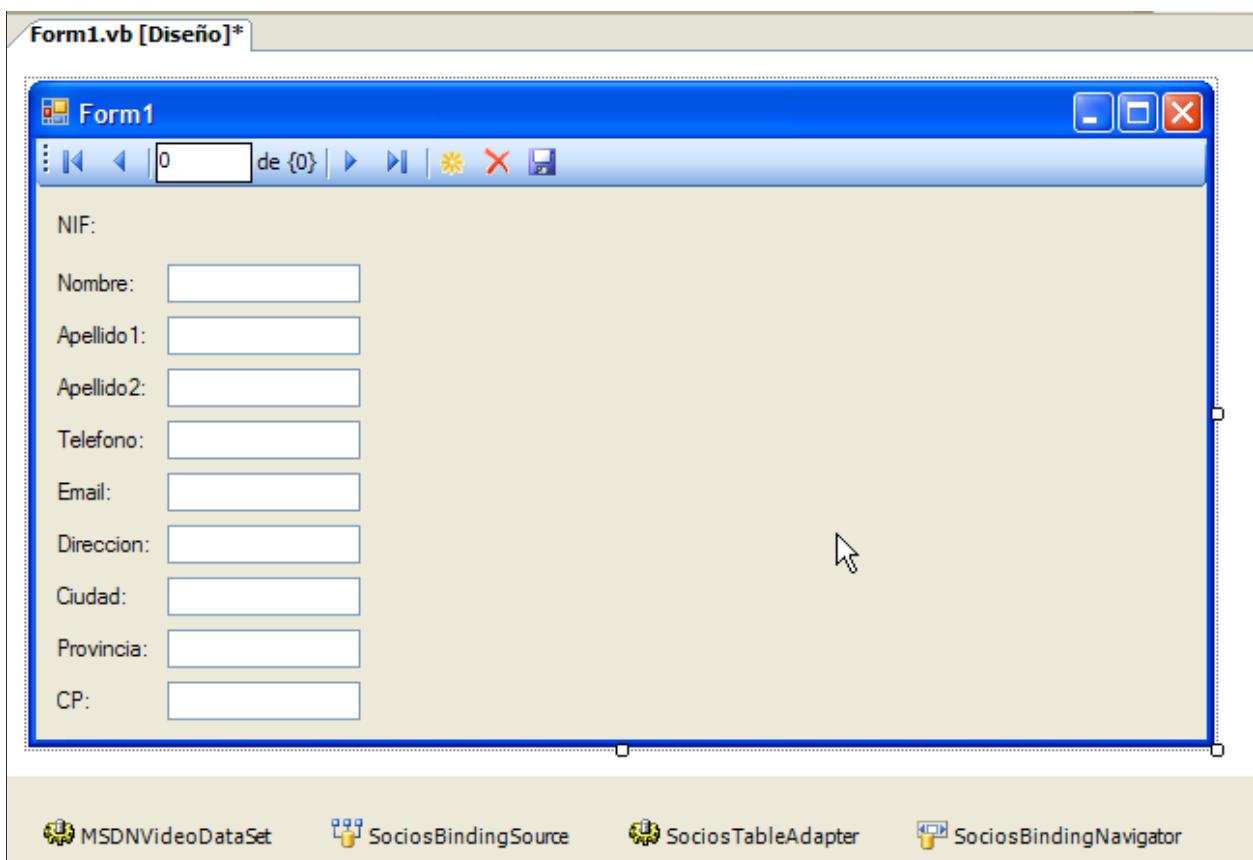
Para situar los datos de la tabla *Socios* dentro de un formulario y en concreto como una información de *detalle*, bastará con arrastrar y soltar la tabla *Socios* sobre el formulario como se indica en la figura 1.



Para presentar la información como detalle, arrastraremos la tabla Socios de la ventana Orígenes de datos sobre el formulario Windows

Figura 1

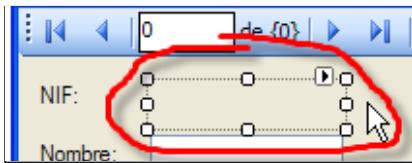
Observaremos que *Visual Studio 2010*, generará por nosotros un conjunto de componentes y controles, que por defecto tendrá una apariencia similar a la que se presenta en la figura 2.



Controles y Componentes de la tabla maestra añadidos al formulario Windows

Figura 2

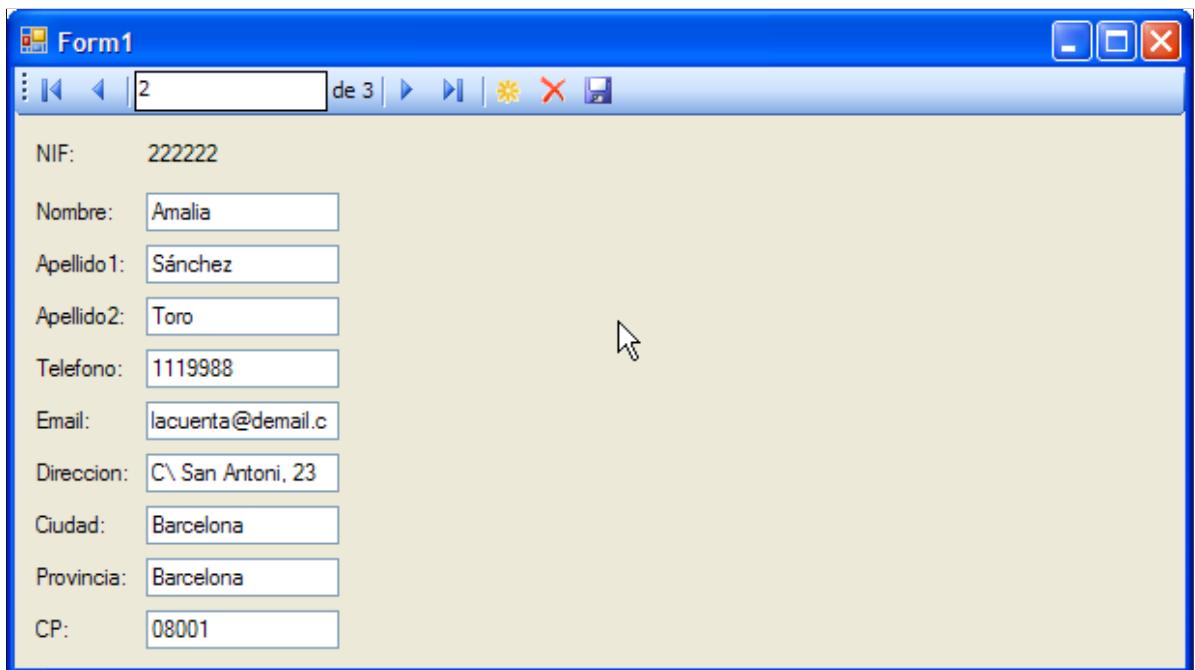
Observe por un momento, que el campo **NIF** que declaramos como *Label* en la ventana de *Orígenes de datos*, aparece como tal en el formulario, tal y como se indica en la figura 3.



Campo NIF de la tabla, representado como un control Label en el formulario Windows

Figura 3

Si ejecutamos nuestra aplicación, observaremos que esta actúa como una típica aplicación de acceso a datos que nos permite navegar a través de los campos de la tabla *Socios*, tal y como se indica en la figura 4.



Aplicación en ejecución de la tabla de detalle incrustada en el formulario Windows

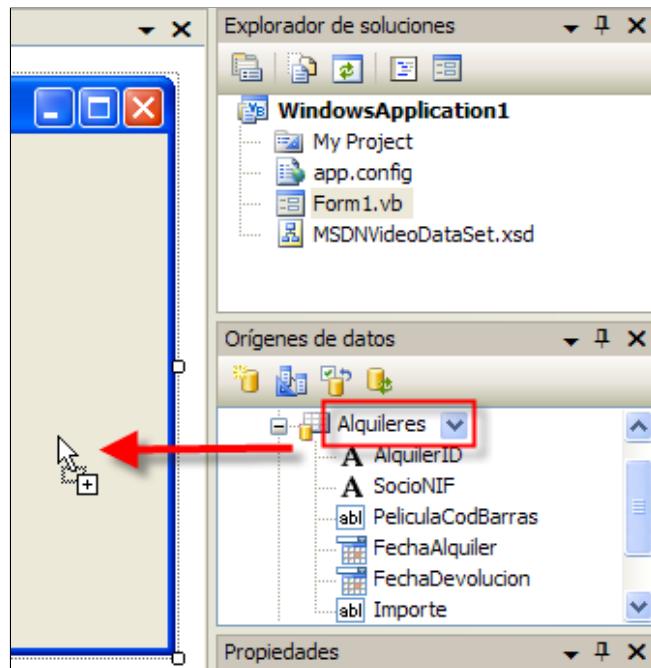
Figura 4

A continuación, insertaremos en el formulario la tabla *Alquileres* y relacionaremos ambas tablas para que se muestren los datos relacionados, dentro del formulario Windows.

Incrustando la tabla hija en el formulario

Ya tenemos la tabla padre insertada en nuestro formulario Windows. Nuestra segunda acción, será la de incrustar en el formulario los datos o tabla hoja, que en nuestro caso es la formada por la tabla *Alquileres*, la cuál además, posee una relación entre campos con la tabla *Socios* insertada anteriormente.

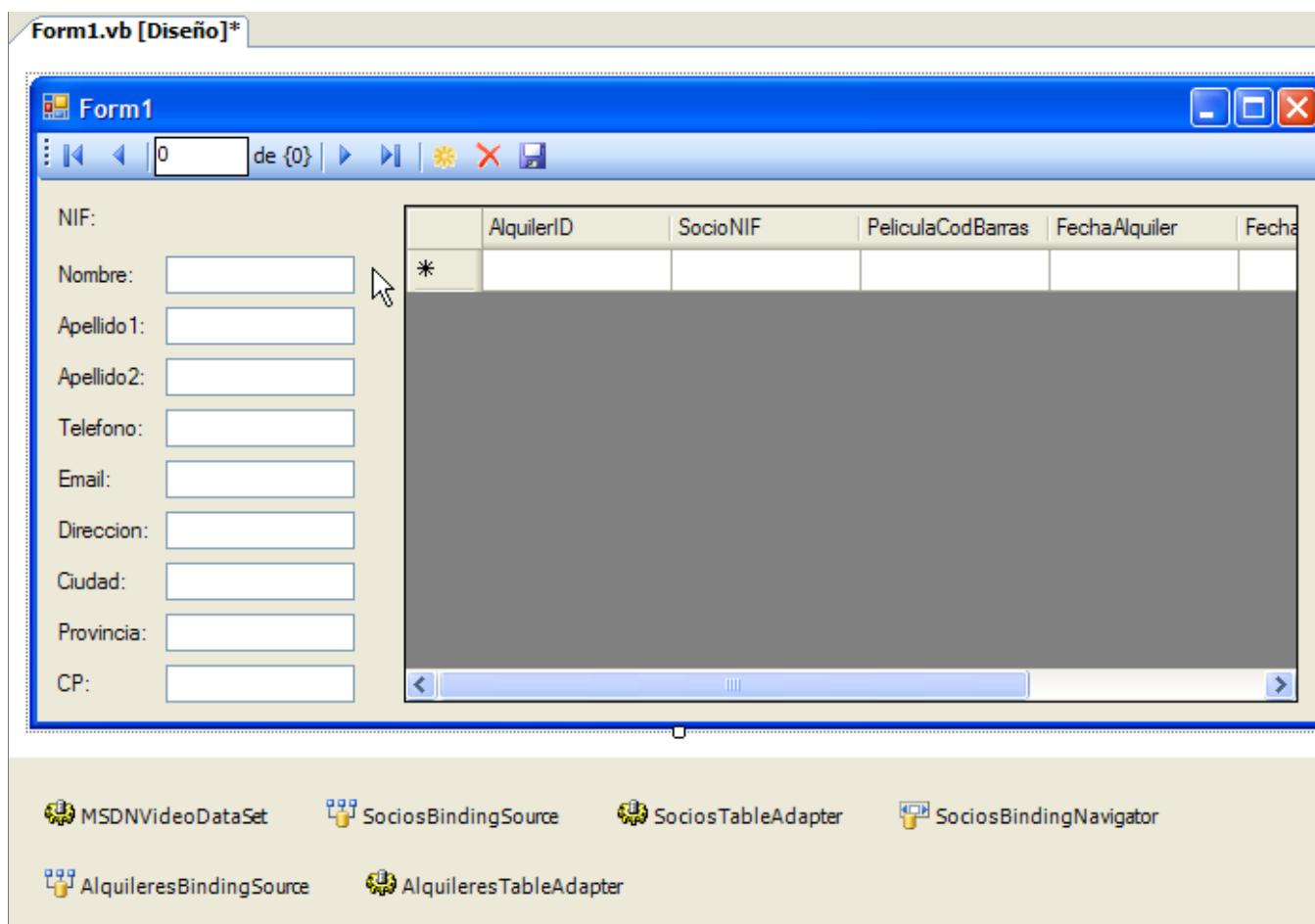
Para llevar a cabo esta acción arrastraremos y soltaremos la tabla *Alquileres* sobre el formulario como se indica en la figura 5.



Para presentar la información de la tabla Alquileres, arrastraremos la tabla de la ventana Orígenes de datos sobre el formulario Windows

Figura 5

Observe que *Visual Studio 2010*, genera por nosotros más componentes y controles, que por defecto tendrá una apariencia similar a la que se presenta en la figura 6.



Controles y Componentes de la tabla maestra y detalle añadidos al formulario Windows

Figura 6

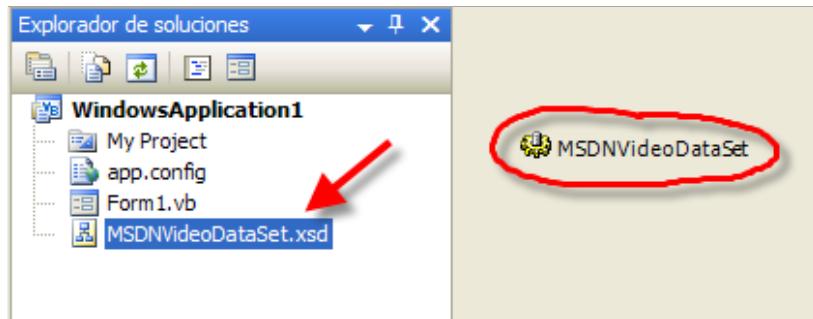
Como podemos observar, el entorno de trabajo ha hecho por nosotros el trabajo más complejo para representar los datos de forma rápida y sencilla.

El esquema de *datos tipados*, aparecía ya en nuestro proyecto cuando asignamos el correspondiente origen de datos.

Ahora lo que ha ocurrido, es que al arrastrar y soltar la tabla padre *Socios* de la ventana de *Orígenes de datos*, en el entorno se ha añadido un componente de nombre *MSDNVideoDataSet* que es el que permitirá relacionar el *DataSet tipado* con nuestros datos.

Este componente será usado por la relación maestro detalle de las dos tablas añadidas al formulario.

En la figura 7, podemos ver el esquema añadido a nuestro proyecto, y el componente del que estamos hablando.



Esquema del DataSet tipado añadido al proyecto y su componente de relación

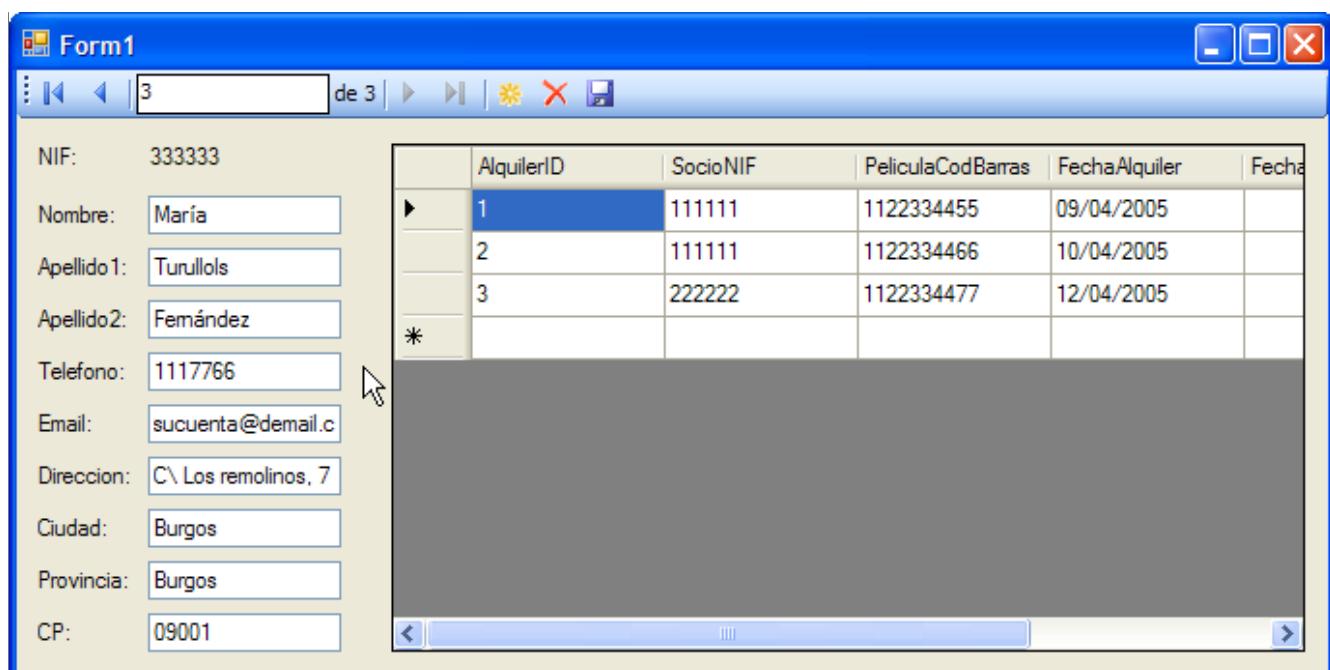
Figura 7

Ejecute la aplicación y observe el comportamiento de la misma.

Observará por lo tanto, que los datos entre detalle y maestra, no están relacionados.

Si navegamos a través de los datos de detalle a través del objeto *SociosBindingNavigator*, el control *DataGridView* no representa la relación de los datos seleccionados.

Esto es lo que se muestra en la figura 8.



Ejecución de la aplicación confirmando que los datos mostrados no están relacionados

Figura 8

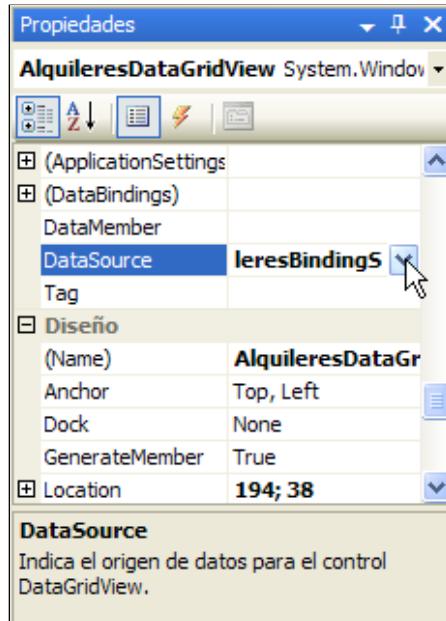
A continuación, la tarea que nos queda para completar el correcto funcionamiento de nuestra aplicación, es la de relacionar la tabla detalle y la tabla maestra entre sí, para que los datos que se muestran en la aplicación, estén relacionados entre sí.

Relacionando la tabla padre con la tabla hija

La tarea más sencilla es la de relacionar la tabla detalle con la tabla maestra. Es una tarea sencilla, porque *Visual Studio 2010* nos proporciona las herramientas necesarias para simplificar al máximo esta tarea.

Para llevar a cabo esta tarea, haga clic sobre el control *DataGridView* que corresponde a los datos de la tabla maestra, y acceda a la ventana de *Propiedades*.

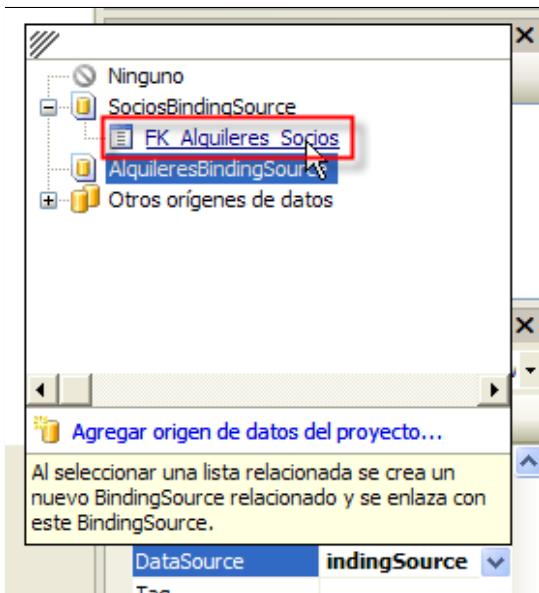
Dentro de la ventana de *Propiedades*, acceda a la propiedad *DataSource* como se indica en la figura 9.



Propiedad **DataSource** del control **DataGridView** de la información maestra

Figura 9

Despliegue esta propiedad, y de la lista desplegable que aparece, seleccione la opción **FK_Alquileres_Socios** como se indica en la figura 10.



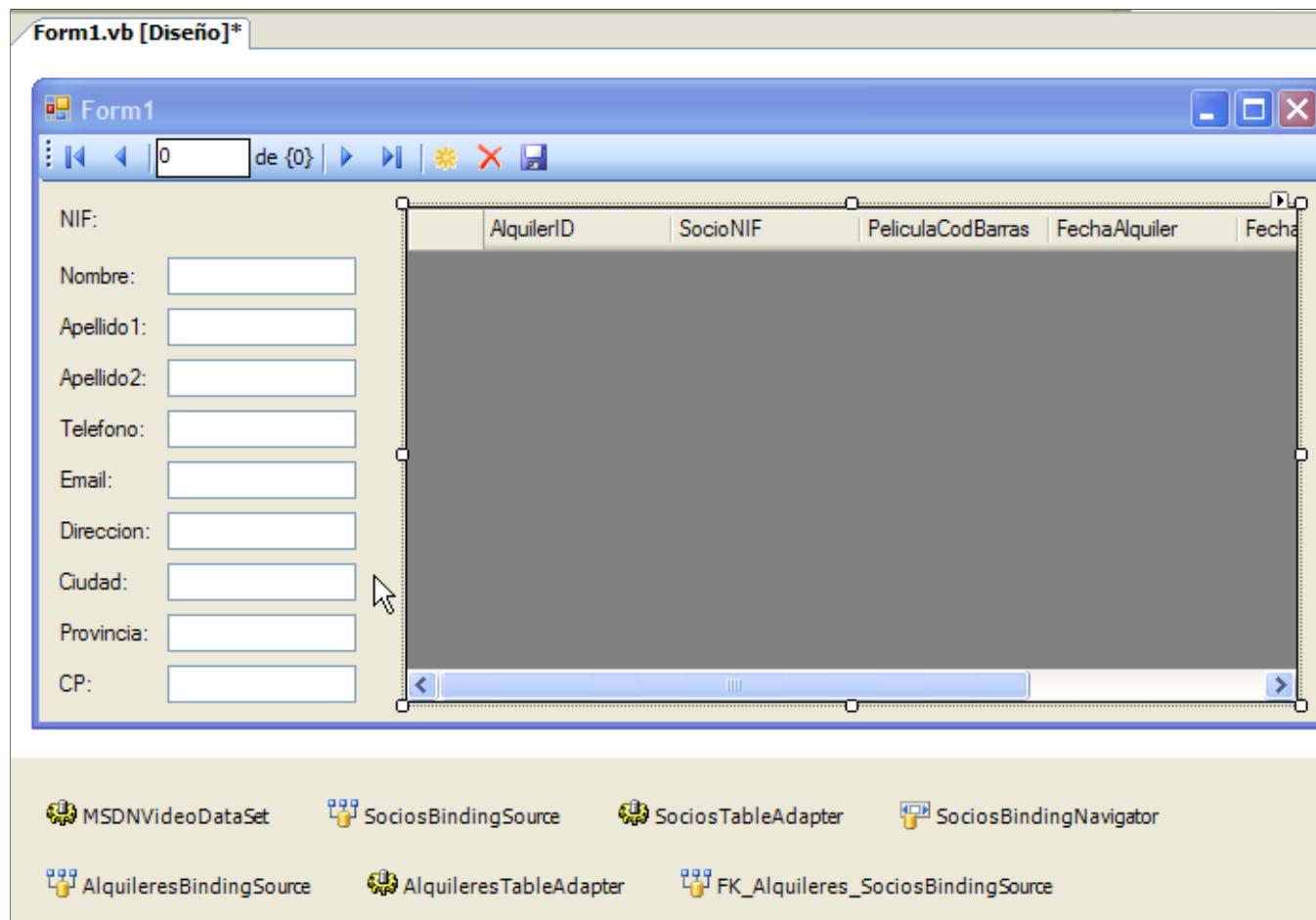
Asignación de la clave de relación entre las tablas

Figura 10

Cuando se asigna el campo de relación de las tablas, dentro de la aplicación se añade esta relación para que cuando naveguemos entre los datos de la tabla *Socios* aparezca toda la información de la tabla *Alquileres* relacionada con la tabla *Socios*.

Esto de lo que hablamos, está supeditado por el componente *FK_Alquileres_SociosBindingSource* que es lo que se

indica en la figura 11.

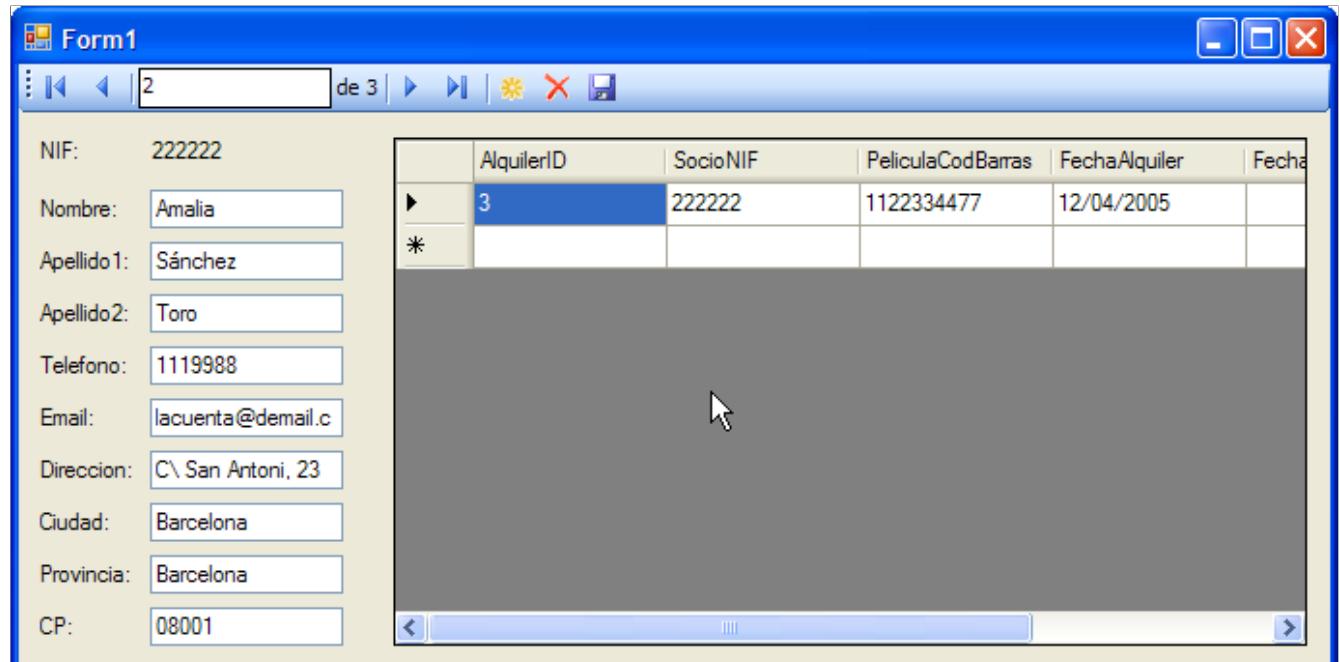


Controles y componentes incluido el de relación entre tablas, añadidos al formulario Windows

Figura 11

Para finalizar, ejecutaremos nuestra aplicación y comprobaremos que el funcionamiento de esta, incluida la relación entre tablas, funciona como esperábamos.

En la figura 12, podemos observar el comportamiento de nuestra aplicación en ejecución.



Aplicación en ejecución, mostrando la correcta relación entre las tablas

Figura 12

Lección 5: Enlace a formularios

- [¿Qué son los datos maestro detalle?](#)
 - [Configurando la fuente de datos](#)
 - [Preparando el origen de datos](#)
 - [Incrustando los datos maestro detalle](#)
- [Manipulando los datos maestro detalle](#)

Módulo 5 - Capítulo 5

5. Manipulando los datos maestro detalle

Obviamente, los datos maestro detalle no nos sirve únicamente para insertar las tablas de datos en un formulario, mostrarlos y navegar por ellos.

Además de esto, podemos también manipular los datos maestro detalle, modificarlos, actualizarlos, borrarlos, sin hacer ninguna acción adicional.

El control *BindingNavigator* ya proporciona todas las características necesarias para realizar estas acciones. Podemos personalizar el control para permitir o denegar estas acciones.

Además, dentro de la ventana de *Orígenes de datos*, podemos seleccionar diferentes campos de las tablas y cambiar el tipo de control en el que queremos representar sus datos.

A continuación veremos un breve ejemplo de como manipular datos para que nos sirva de aprendizaje de cómo hacer esto posible.

Modificando datos

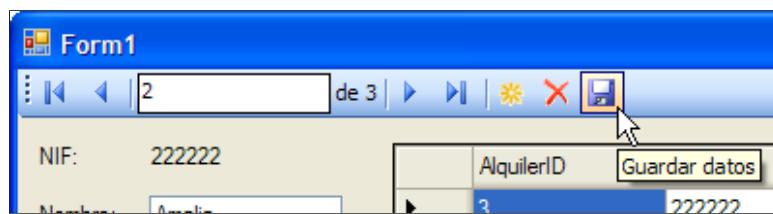
Ejecute la aplicación de ejemplo que hemos diseñado hasta ahora y sitúese en alguno de sus campos.

Centrándonos en la información de la tabla *Socios*, cambiaremos un campo determinado, como el que se muestra en la figura 1.

Modificaremos el valor de un campo para que nos sirva de ejemplo

Figura 1

Acto seguido, cuando hayamos realizado la modificación, haremos clic sobre la opción de *Guardar datos*, tal y como se muestra en la figura 1.



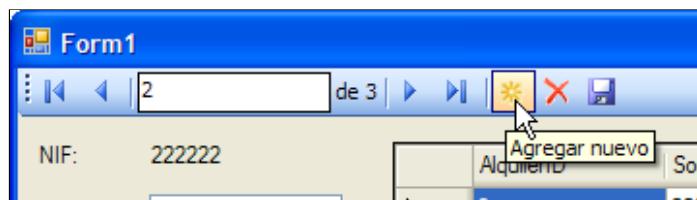
Opción del control *BindingNavigator* para guardar los datos modificados

Figura 2

Como vemos, la manipulación de datos es realmente sencilla y en la relación de datos mostrada, no tiene porqué presentarnos ninguna dificultad.

Insertando y eliminando datos

Si queremos agregar datos, deberemos hacer clic sobre la opción *Agregar nuevo* del control *BindingNavigator* como se muestra en la figura 3.



Añadir un registro nuevo es realmente sencillo

Figura 3

De la misma manera funciona el mecanismo para eliminar un registro, tal y como se muestra en la figura 4.



Eliminar un registro de forma rápida

Figura 4

Recuerde presionar el icono si quiere que los cambios y modificaciones realizadas se mantengan. Pulsando sobre ese icono, la acción de manipulación de datos se lanza contra la base de datos.

Módulo 6: Servicios Web

- **Introducción a los Servicios Web**
- **Creación de Servicios Web**
- **Consumo de Servicios Web**

Contenido

Este módulo presenta al alumno los fundamentos de los Servicios Web y las Arquitecturas Orientadas a Servicios (SOA).

Tras una introducción a los servicios Web y sus conceptos asociados se ve la forma de crear y consumir servicios Web. Para la creación de un servicio utilizaremos Visual Studio 2010. Si desea más información sobre esta herramienta puede dirigirse al curso de Desarrollo de Aplicaciones Web con ASP.NET.

- **Lección 1: Introducción a los servicios Web**

- ¿Qué son los servicios Web?
- Comunicación entre componentes
- SOAP
 - Breve historia de SOAP
 - Las bases tecnológicas de SOAP
 - Descubrimiento de servicios: WSDL y UDDI

- **Lección 2: Creación de servicios Web**

- Nuestro primer servicio Web
 - Crear un proyecto de tipo servicio Web con Visual Studio 2010
 - Crear un servicio Web usando un solo fichero
 - Eliminar ficheros de un proyecto
 - Analizando el contenido de un servicio Web
 - Atributos aplicables a los servicios Web
 - Definición de la clase a usar en el servicio Web
 - Añadir métodos para usarlos en el servicio Web
 - Probar nuestro servicio Web

- **Lección 3: Consumo de servicios Web**

- Desde una aplicación de Windows
 - Alojar el servicio Web en un servidor local
 - Activar el servidor Web para usar con un directorio local
- Crear un proyecto Windows para usar el servicio Web
 - Añadir una referencia para acceder al servicio Web

-
- Acceder al servicio Web desde el código
 - ¿Que es lo que puede fallar?

Lección 1: Introducción a los Servicios Web

- **¿Qué son los Servicios Web?**
- **Comunicaciones entre componentes**
- **SOAP**

Introducción a los servicios Web

Veamos lo que nos dice la documentación de Visual Studio sobre los servicios Web (o servicios Web XML que es como los denomina Microsoft):

"Un servicio Web XML es una entidad programable que proporciona un elemento de funcionalidad determinado, como lógica de aplicación, al que se puede tener acceso desde diversos sistemas potencialmente distintos mediante estándares de Internet muy extendidos, como XML y HTTP."

Un poco más claro queda lo que continúa en el siguiente párrafo de la mencionada ayuda:

"Un servicio Web XML puede ser utilizado internamente por una aplicación o bien ser expuesto de forma externa en Internet por varias aplicaciones. Dado que a través de una interfaz estándar es posible el acceso a un servicio Web XML, éste permite el funcionamiento de una serie de sistemas heterogéneos como un conjunto integrado."

Mejor?

Simplificando, y siendo algo más prácticos, podemos definir un servicio Web XML como una clase a la que podemos acceder utilizando estándares de Internet.

Como es de suponer, el tener que utilizar esos estándares de comunicación de Internet es porque esa "clase" está alojada en un servidor de Internet, es decir, un servicio Web es una clase que está alojada en la Web y que podemos acceder a ella mediante ciertos estándares como XML, que a su vez utiliza otro estándar: SOAP, (*Simple Object Access Protocol*), que es el lenguaje que define cómo nos comunicaremos con el servicio Web.

Antes de pasar a ver ejemplos prácticos sobre cómo crear y utilizar servicios Web utilizando Visual Studio 2010, veamos un poco la historia de porqué finalmente los servicios Web son lo que son y cómo nos facilitan toda la comunicación a través de la red, ya sea local o global.

- **¿Qué son los servicios Web?**
 - Un poco de historia: modelos de desarrollo
- **Comunicación entre componentes**

- **SOAP**

- Breve historia de SOAP
 - Las bases tecnológicas de SOAP
 - Descubrimiento de servicios: WSDL y UDDI
-

Lección 1: Introducción a los Servicios Web

¿Qué son los Servicios Web?

- Comunicaciones entre componentes
- SOAP

¿Qué son los servicios Web?

La expresión "Servicio Web" se oye con fuerza desde hace unos años en el ámbito del desarrollo de aplicaciones e incluso en ambientes poco técnicos y de dirección. Lo cierto es que no se trata de un concepto tan novedoso como cabría esperar y las innovaciones que conlleva no son tanto tecnológicas, como conceptuales.

En esta lección explicaremos desde un punto de vista no-técnico los conceptos relacionados con los Servicios Web, cómo funcionan, de dónde vienen y a dónde van.

Un poco de historia: modelos de desarrollo

El hecho de poder comunicar componentes de software entre sí tiene una enorme importancia. Hasta no hace tantos años era muy típico hacer **aplicaciones de una sola pieza**, "monolíticas":

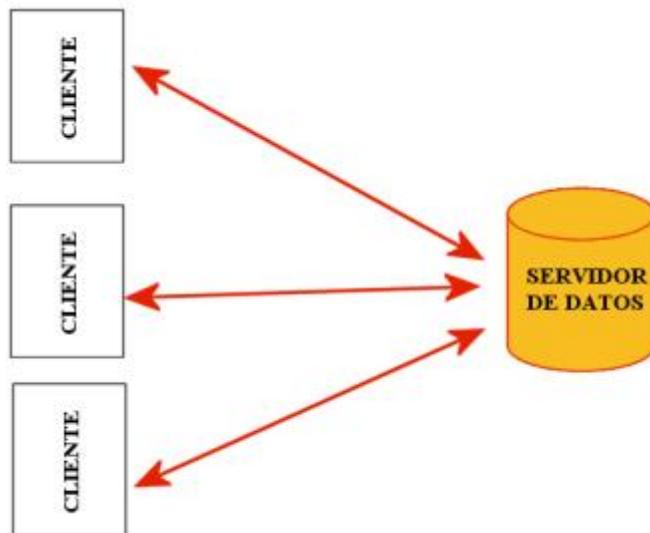


Figura 6.1. Aplicación "monolítica" aunque distribuida

Estos programas podían acceder a un sistema gestor de datos a través de la red, pero toda la lógica del flujo de datos, la seguridad y las interacciones con las personas se encontraban en el ordenador del usuario en forma de un gran ejecutable. Se suelen conocer también como "*fat clients*". La situación descrita no es la ideal ya que implica problemas de toda índole a la hora de instalar las aplicaciones y sobre todo cuando se modifican o actualizan (mantenimiento complejo y engorroso).

Una metodología de desarrollo mucho mejor aunque más laboriosa a la hora de programar es el modelo **Cliente-Servidor en tres capas**:

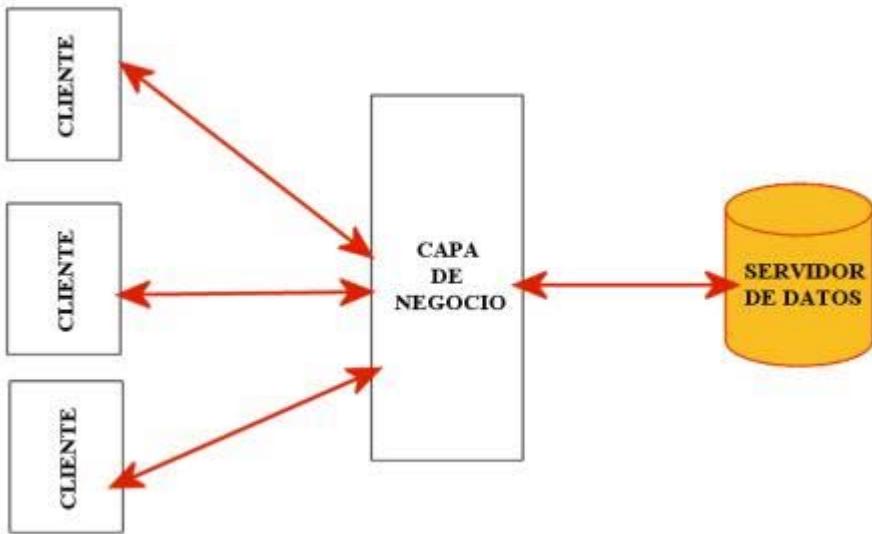


Figura 6.2. Aplicación Cliente Servidor en tres capas

En este modelo toda la lógica de los datos, su validación, los permisos, etc., residen en un servidor intermedio y son utilizados por todos los clientes a través de una red. En este caso en el ordenador del usuario lo único que hay es una capa de presentación que se ocupa básicamente de recoger y recibir datos, es decir, actúa de intermediario entre el usuario y las reglas de negocio residentes en la capa intermedia. Este modelo es más eficiente y está muy evolucionado respecto al anterior pero aún se puede ir más allá.

La arquitectura de **desarrollo en n-capas** (*n-tier* que dicen los anglosajones) lleva el concepto cliente-servidor un paso hacia adelante, dividiendo la capa intermedia en muchas otras capas especializadas cada una de las cuales puede residir en un servidor diferente:

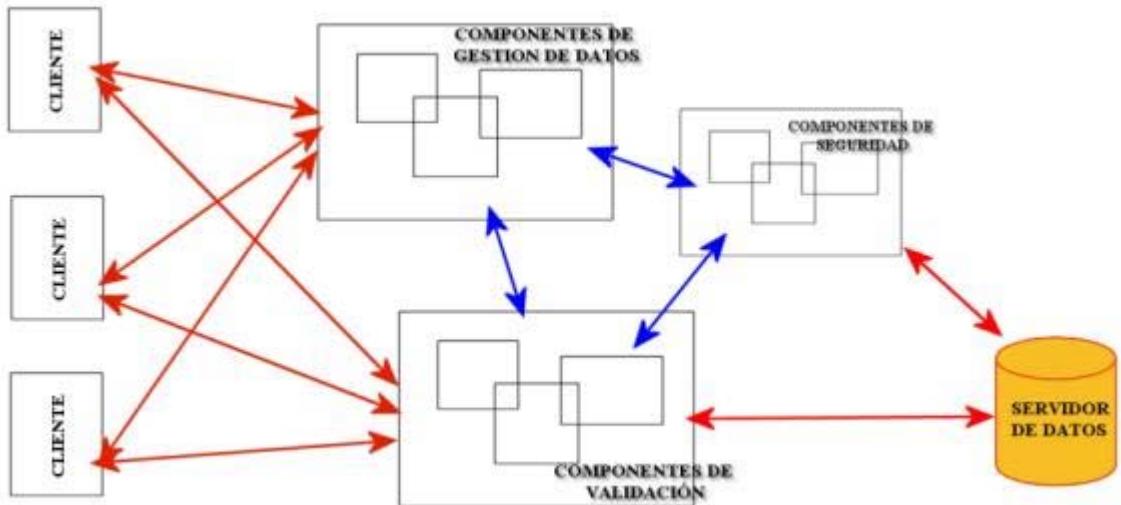


Figura 6.3. Arquitectura de desarrollo basada en componentes

En este modelo existe una gran variedad de componentes especializados en tareas específicas como la validación de datos, la autenticación y seguridad o el acceso a datos. Dichos componentes deben trabajar unos con otros como piezas de un mecanismo, gestionando la información que circula entre el usuario y el servidor de datos.

La belleza de este modelo radica en que cada uno de ellos (o cada grupo de ellos) puede residir en un servidor diferente, siendo transparente su ubicación para los clientes que los utilizan. Ello aumenta mucho la escalabilidad de las aplicaciones, pues basta con añadir nuevos servidores e instalar los componentes en ellos para poder atender más peticiones.

Por otra parte, y esto es muy interesante también, mientras sus interfaces de programación sean las mismas, es posible sustituir cualquier componente por otro actualizado o que actúe de manera distinta para corregir errores o cambiar el modo de trabajo de la aplicación global, y todo ello sin que los clientes sean conscientes de ello. Obviamente esto ofrece más ventajas aún ya que, por ejemplo, no es necesario reinstalar la aplicación en cada cliente, sino que basta con sustituir un componente en un único lugar y automáticamente todos los usuarios tendrán su aplicación actualizada.

El concepto de **Arquitectura Orientada a Servicios** o **SOA** se basa en el uso de este tipo de componentes que suplen las necesidades de una o varias aplicaciones, son independientes entre sí y trabajan independientemente del sistema operativo o la plataforma.

Aunque muchos programadores piensan que SOA está relacionado únicamente con los Servicios Web lo cierto es que se pueden conseguir arquitecturas SOA con otras tecnologías como veremos.

Lección 1: Introducción a los Servicios Web

- ¿Qué son los Servicios Web?
- Comunicaciones entre componentes
- SOAP

Comunicación entre componentes

Existen diversas dificultades técnicas a la hora de llevar a la práctica las arquitecturas orientadas a servicios. Entre éstas están la comunicación entre las distintas capas y componentes que constituyen la aplicación, la gestión de peticiones y el balanceado de carga entre servidores cuando un mismo componente reside en varios de ellos (para aplicaciones muy grandes con muchos clientes), la gestión de transacciones entre componentes y algunas otras cosas más.

Existe la posibilidad de escribir un protocolo de comunicaciones propio que se ocupe de todas estas cuestiones, pero por supuesto se trata de una opción muy poco realista dada la complejidad que conllevaría. Para responder a estas necesidades de los programadores, diversos fabricantes y asociaciones de la industria crearon servicios y protocolos específicos orientados a la interacción distribuida de componentes. Aunque existe una gran variedad, de todos ellos los más importantes sin duda son:

- **DCOM** (*Distributed Common Object Model*), la propuesta de Microsoft, ligada a sus sistemas Windows. Se trata de algo más que un protocolo de invocación remota de procedimientos (RPC) ya que su última encarnación, COM+, incluye servicios avanzados para balanceado de carga, gestión de transacciones o llamadas asíncronas. Los parámetros son transmitidos a través de la red mediante un formato binario propio llamado NDR (*Network Data Representation*).
- **RMI** (*Remote Method Invocation*), es la metodología de llamada remota a procedimientos de Java. No se centra en la definición de interfaces para compatibilidad binaria de componentes, ni en otros conceptos avanzados, y se basa en la existencia de un cliente y un servidor que actúan de intermediarios entre los componentes que se quieren comunicar. Es una tecnología bastante simple que es fácil de utilizar para aplicaciones básicas.
- **CORBA** (*Common Object Request Broker Architecture*). Se trata de una serie de convenciones que describen cómo deben comunicarse los distintos componentes, cómo deben transferir los datos de las llamadas y sus resultados o cómo se describen las interfaces de programación de los componentes para que los demás sepan cómo utilizarlos. Fue desarrollado por el OMG (*Object Management Group*) en la segunda mitad de la década de los '90 y es el modelo que más éxito ha tenido en el mundo UNIX. Su método de empaquetado y transmisión de datos a través de la red se llama CDR (*Common Data Representation*).

representation). Existen diversas implementaciones de distintos fabricantes.

Estos modelos son buenos y muy eficientes, cumpliendo bien su trabajo pero tienen algunas limitaciones importantes siendo las principales las siguientes:

- Es difícil la comunicación entre los distintos modelos
 - Están ligados a plataformas de desarrollo específicas, lo que dificulta la comunicación entre ellas
 - Su utilización a través de Internet se complica debido a cuestiones de seguridad de las que enseguida hablaremos.
 - Existen en el mercado puentes CORBA/DCOM que permiten la comunicación entre componentes COM y componentes CORBA, pero su utilización es difícil y añaden una nueva capa de complejidad a las aplicaciones además de disminuir su rendimiento.
-

Lección 1: Introducción a los Servicios Web

- ¿Qué son los Servicios Web?
 - Comunicaciones entre componentes
- SOAP**

SOAP

Las expectativas actuales respecto a los componentes han aumentado. Al igual que podemos usar un navegador web para acceder a cualquier página independientemente del sistema operativo del servidor en que resida, ¿por qué no podríamos invocar métodos de componentes a través de la red independientemente de dónde se encuentren, del lenguaje en el que estén escritos y de la plataforma de computación en la que se ejecuten?.

Esto es precisamente lo que ofrecen los Servicios Web. Gracias a ellos se derriban las antiguas divisiones resultantes de los modelos de componentes descritos, y la integración de las aplicaciones, la ubicuidad de sus componentes y su reutilización a través de la red se convierten en una realidad.

La tecnología que está detrás de todo ello se llama **SOAP** (jabón en inglés). Este acrónimo (*Simple Object Access Protocol*) describe un concepto tecnológico basado en la **sencillez** y la **flexibilidad** que hace uso de **tecnologías y estándares comunes** para conseguir las promesas de la ubicuidad de los servicios, la transparencia de los datos y la independencia de la plataforma que según hemos visto, se hacen necesarios en las aplicaciones actuales.

Breve historia de SOAP

SOAP empezó como un protocolo de invocación remota basado en XML diseñado por Dave Winer de UserLand, llamado XML-RPC. A partir de éste se obtuvo en Septiembre de 1999 la versión 1.0 de SOAP, en la que participó activamente Microsoft y el archiconocido experto en programación Don Box.

Esta primera versión fue más o menos despreciada por los principales fabricantes de software que en esa época tenían en marcha un proyecto más ambicioso llamado ebXML. Esto puso en peligro en su nacimiento la existencia de SOAP ya que si los grandes no lo apoyaban poco se podía hacer. Por fortuna uno de estos grandes fabricantes, IBM, decidió apoyarlo y en la actualidad no sólo acepta SOAP sino que es uno de los motores detrás de su desarrollo (dos importantes personas de IBM y su filial Lotus, David Ehnebuske y Noah Mendelsohn, son autores de la especificación 1.1 de SOAP). Sun Microsystems también anunció oficialmente en Junio de 2000 que soportaba el estándar. El hecho de que los tres gigantes del software (Microsoft, IBM y Sun) apoyen SOAP ha hecho que muchos fabricantes de Middleware y puentes CORBA-DCOM (como Roguewave o IONA) ofrezcan productos para

SOAP, así como otras muchas pequeñas empresas de software.

El paso definitivo para asegurar el éxito de SOAP y los servicios web es su envío al W3C (World Wide Web Consortium) para proponerlo como estándar. La última versión de la especificación se puede encontrar en www.w3.org/TR/SOAP/.

Este soporte mayoritario hace que su éxito y pervivencia estén asegurados y hoy todas las herramientas de desarrollo del mercado ofrecen en menor o mayor medida soporte para SOAP. Por supuesto .NET y Visual Studio son los entornos más avanzados en la adopción de SOAP.

La base tecnológica de SOAP

Lo interesante de SOAP es que utiliza para su implementación tecnologías y estándares muy conocidos y accesibles como son XML o el protocolo HTTP.

- Dado que los mensajes entre componentes y los datos de los parámetros para llamadas a métodos remotos se envían en formato XML basado en texto plano, SOAP se puede utilizar para comunicarse con cualquier plataforma de computación, consiguiendo la ansiada ubicuidad de los componentes.
- El uso de HTTP como protocolo principal de comunicación hace que cualquier servidor web del mercado pueda actuar como servidor SOAP, reduciendo la cantidad de software a desarrollar y haciendo la tecnología disponible inmediatamente. Además en la mayoría de los casos se puede hacer uso de SOAP a través de los cortafuegos que defienden las redes, ya que no suelen tener bloqueadas las peticiones a través del puerto 80, el puerto por defecto de HTTP (de ahí la ubicuidad, aunque se pueden usar otros puertos distintos al 80, por supuesto).

La seguridad se puede conseguir a través de los métodos habituales en los servidores web y por tanto se dispone de autenticación de usuarios y cifrado de información de forma transparente al programador, usando protocolos y técnicas como IPSec o SSL, ampliamente conocidos y usados en el mundo web.

Por otra parte la escalabilidad se obtiene a través del propio servidor web o incluso del sistema operativo, ya que la mayoría de ellos (por ejemplo IIS) poseen capacidades de ampliación mediante clusters de servidores, enrutadores que discriminan las peticiones y otras técnicas para crear Web Farms, o conjuntos de servidores web que trabajan como si fueran uno solo para así poder atender a más clientes simultáneamente.

Nota:

Existen ampliaciones al protocolo SOAP base que definen protocolos y convenciones para tareas específicas como las mencionadas de seguridad, enrutado de mensajes, los eventos y muchas otras cuestiones avanzadas. En .NET se implementan mediante los conocidos *Web Services Enhancements* (WSE) actualmente por su versión 3.0, y en un futuro inmediato con Windows Communication Foundation, la nueva plataforma de servicios de comunicaciones de Windows. El estudio de éstos se sale del ámbito de este curso.

Como vemos, las tecnologías utilizadas son conocidas y la especificación SOAP se refiere más bien a la manera de usarlas. De este modo las áreas cubiertas por la especificación se refieren a cómo se codifican los mensajes XML que contienen las llamadas a procedimientos y sus respuestas, y a la manera en que HTTP debe intercambiar estos mensajes. Si nos referimos a la esencia del estándar, SOAP trata de sustituir a los diferentes formatos propietarios de empaquetamiento de datos que utilizan otras tecnologías (como DCOM o CORBA con NDR y CDR respectivamente), así como los protocolos propietarios empleados para transmitir estos datos empaquetados.

HTTP es el único protocolo definido en el estándar para SOAP pero éste es lo suficientemente abierto como para permitir que se empleen otros protocolos distintos para transmitir mensajes SOAP. Por citar unos pocos, se podría utilizar SMTP (correo electrónico), MSMQ (*Microsoft Messaging Queue*) para enviar de manera asíncrona las llamadas a procedimientos con SOAP, etc...

Descubrimiento de servicios: WSDL y UDDI

Otro de los estándares que se definen en SOAP es **WSDL** (*Web Service Definition Language*). Se trata de un formato estándar para describir las interfaces de los servicios web. WSDL describe qué métodos están disponibles a través de un servicio Web y cuáles son los parámetros y valores devueltos por éstos. Antes de usar un componente que actúa como servicio web se debe leer su archivo WSDL para averiguar cómo utilizarlo.

Nota:

Para aquellos programadores que conocen otras arquitecturas podemos decir que WSDL es el equivalente en XML a los lenguajes IDL (*Interface Description Language*) de DCOM y CORBA.

Se ha definido también un formato estándar para publicación de información de servicios web llamado UDDI (*Universal Description Discovery and Integration*). Esta especificación permite la creación de directorios de servicios web, donde se definen métodos que permiten consultarlos para encontrar fácilmente aquel servicio que se necesite. Windows Server 2003 incluye gratuitamente un servidor para implementar directorios UDDI en organizaciones.

Lección 2: Creación de Servicios Web

- Nuestro primer Servicio Web

Creación de servicios Web

ASP.NET 2.0 nos facilita grandemente la creación de servicios Web XML, y si nos apoyamos en una herramienta como lo es Visual Studio 2010, incluso en la versión Express, nos daremos cuenta de que no necesitamos ser expertos en los protocolos utilizados por los servicios Web para poder crearlos y utilizarlos.

Tal como comentamos en la introducción de la lección anterior, los servicios Web realmente son clases, clases que exponen unos métodos y son esos métodos los que podremos utilizar para acceder a lo que el servicio Web nos ofrece.

En esta lección veremos de forma simple y práctica, cómo crear un servicio Web utilizando ASP.NET 2.0 y Visual Studio 2010.

Nota:

Si está utilizando Visual Basic 2010 Express debe saber que no permite la creación de servicios web. Si desea crear un servicio puede utilizar la herramienta Visual Web Developer Express o cualquier edición superior de Visual Studio 2010. Si desea profundizar en esta herramienta puede consultar el curso "Desarrollo de Aplicaciones Web con ASP.NET".

• Nuestro primer servicio Web

- Crear un proyecto de tipo servicio Web con Visual Studio 2010
 - Crear un servicio Web usando un solo fichero
 - Eliminar ficheros de un proyecto
 - Analizando el contenido de un servicio Web
 - Atributos aplicables a los servicios Web
 - Definición de la clase a usar en el servicio Web
 - Añadir métodos para usarlos en el servicio Web
- Probar nuestro servicio Web

Lección 2: Creación de Servicios Web

Nuestro primer Servicio Web

Nuestro primer servicio Web

Es ya un clásico en el mundillo de la programación mostrar el mensaje "Hola, Mundo" al crear la primera aplicación y eso es lo que vamos a hacer en nuestro primer servicio Web, crear uno que tenga un método que devuelva una cadena con la frase: "Hola, Mundo".

Crear un proyecto de tipo servicio Web con Visual Studio 2010

Para crear un nuevo proyecto con Visual Studio 2010 podemos hacerlo de varias formas, pero si nos gusta decidir dónde se almacenará nuestro código, la mejor forma de hacerlo es mediante el menú de archivos, seleccionando la opción **Nuevo sitio Web...** Al hacerlo tendremos un cuadro de diálogo como el mostrado en la figura 6.4:

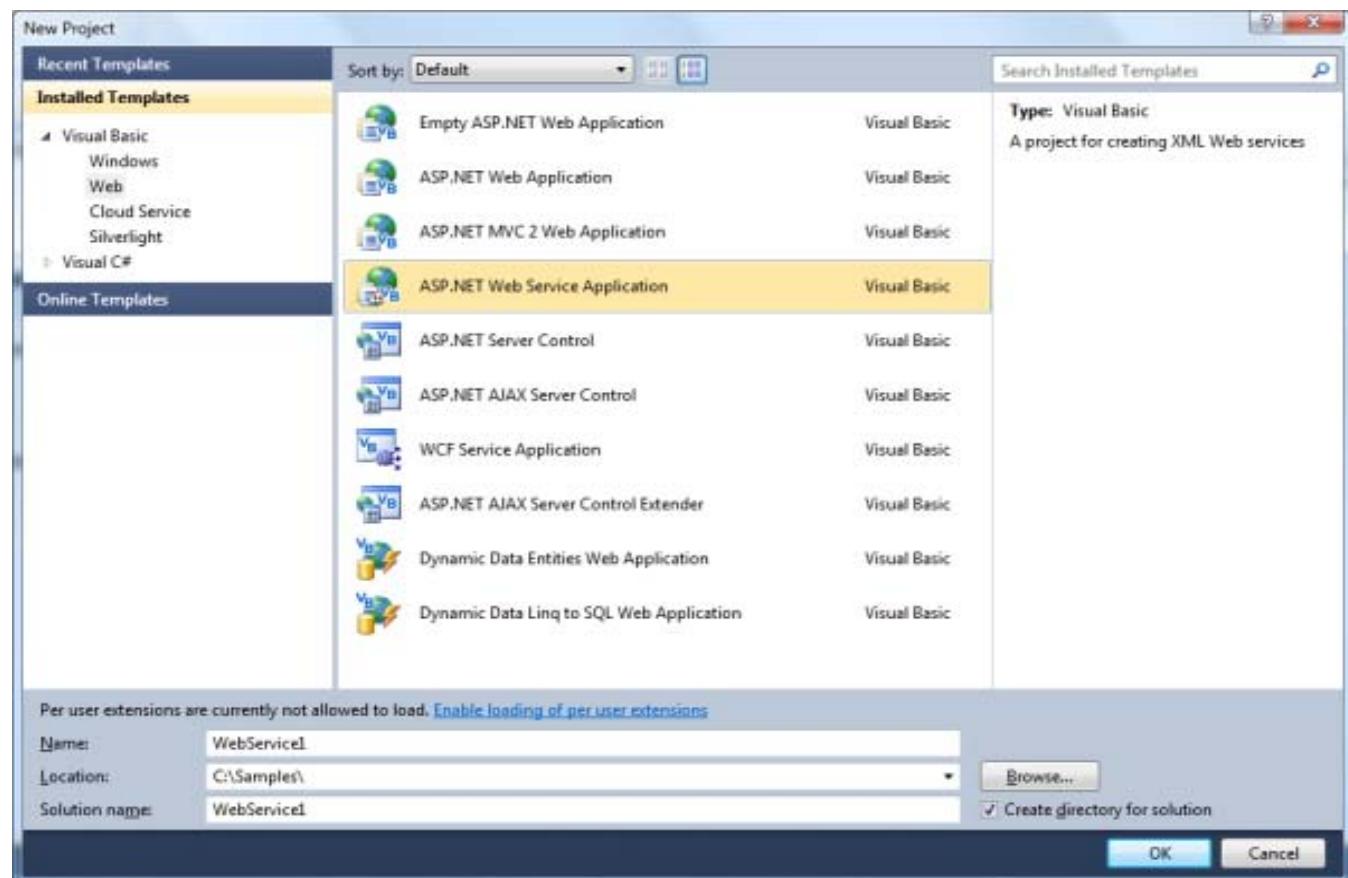


Figura 6.4. Cuadro de diálogo para crear un servicio Web

Visual Studio 2010, nos permite crear las aplicaciones y servicios Web, en varias ubicaciones, entre ellas el sistema de archivos local o de red local, aunque también podemos indicar una dirección HTTP o FTP, esas opciones las tenemos en la lista desplegable que hay junto a la etiqueta Ubicación, indicaremos el directorio en el que queremos alojar nuestro código y el lenguaje que utilizaremos, en este caso concreto lo haremos con Visual Basic.

Nota:

Si utiliza una versión Express de Visual Studio, debe saber que sólo puede desarrollador Servicios Web con Visual Web Developer. En los ejemplos utilizaremos esta versión, aunque todas las versiones superiores de Visual Studio 2010 tienen un funcionamiento similar.

El proyecto creado añade una nueva clase y un fichero con extensión .asmx que es la utilizada para los servicios Web. En la clase incluiremos todo el código necesario para crear la "clase" que usaremos para comunicarnos con las aplicaciones cliente.

El fichero con extensión .asmx simplemente contiene el código de ASP.NET 2.0 que servirá al runtime de .NET para saber que estamos tratando con un servicio Web, además de indicarle dónde está alojado el código que contiene la clase y cómo se llama esta, y cual es el lenguaje que utilizaremos, tal como podemos ver en el siguiente código:

```
<%@ WebService Language="vb" CodeBehind="~/App_Code/Service.vb" Class="Service" %>
```

La parte importante de esa línea de código ASP.NET es **<%@ WebService**, con ella estamos indicando que nuestra intención es definir un servicio Web.

Por otra parte, el atributo *CodeBehind* le informa a ASP.NET 2.0 que el código no está incluido en el fichero .asmx, sino en uno independiente. Este es el tratamiento predeterminado al crear un nuevo servicio Web.

Crear un servicio Web usando un solo fichero

Pero si queremos añadir otro servicio Web a nuestro proyecto, seleccionando la opción **Agregar nuevo elemento** del menú **Sitio Web**, el cuadro de diálogo que nos muestra los distintos tipos de ficheros que podemos agregar, tal como podemos ver en la figura 6.5:

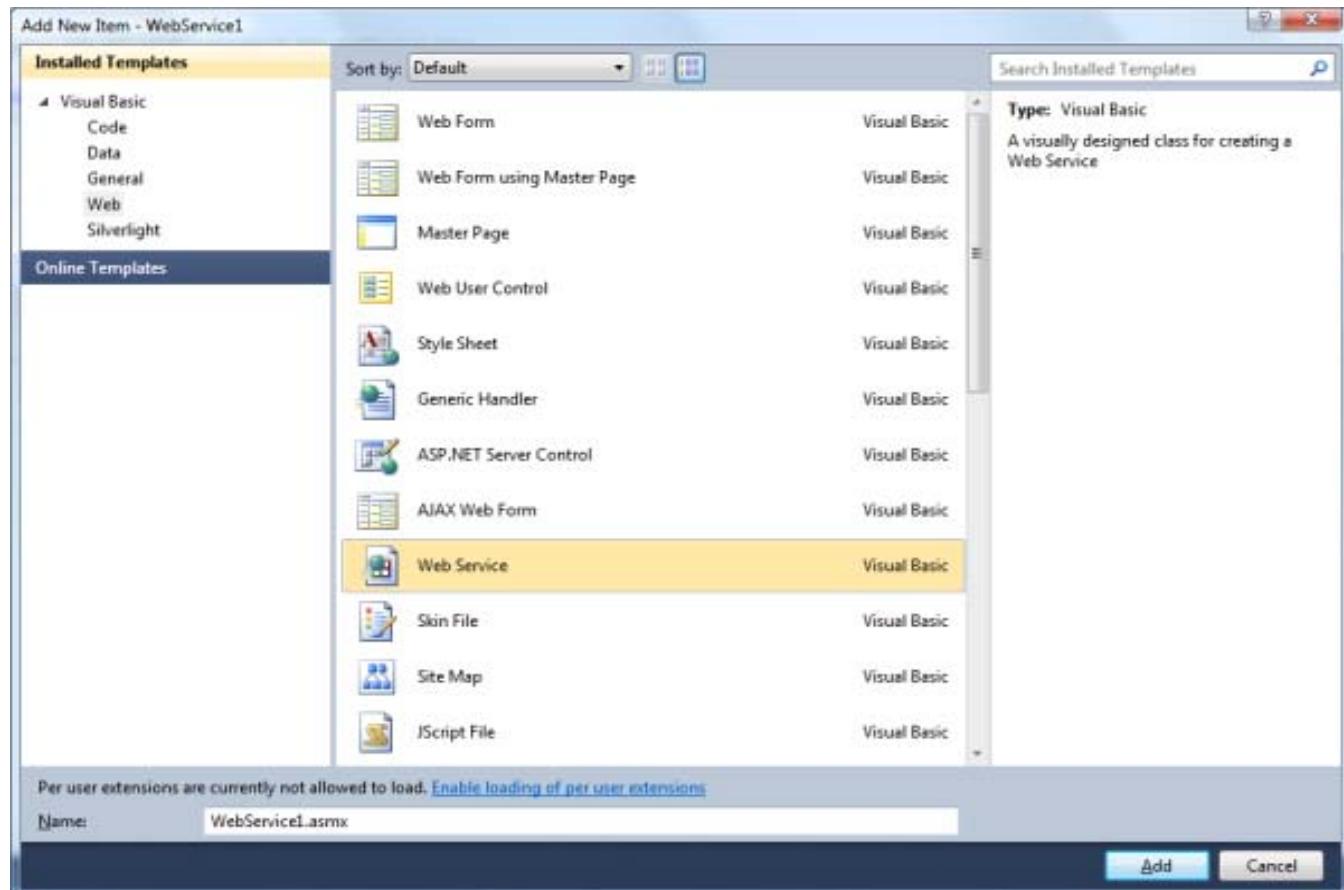


Figura 6.5. Al agregar un nuevo elemento, podemos indicar dónde alojar el código

Si seleccionamos esta forma de hacerlo, tanto la directiva de ASP.NET como el código del lenguaje seleccionado estarán en el mismo fichero, en nuestro ejemplo **HolaMundoSW.asmx**.

Como podemos apreciar en la figura 6.6, al crear el servicio Web de esta forma, tenemos todo el código necesario en un solo fichero, esta será la forma recomendada de hacerlo, al menos si el servicio Web es simple, como es el caso actual.

```
1 <%@ WebService Language="VB" Class="HolaMundoSW" %>
2
3 Imports System.Web
4 Imports System.Web.Services
5 Imports System.Web.Services.Protocols
6
7 <WebService(Namespace := "http://tempuri.org/")> _
8 <WebServiceBinding(ConformsTo:=WsiProfiles.BasicProfile1_1)> _
9 Public Class HolaMundoSW
10     Inherits System.Web.Services.WebService
11
12     <WebMethod()> _
13     Public Function HelloWorld() As String
14         Return "Hello World"
15     End Function
16
17 End Class
18
```

Figura 6.6. Un solo fichero para contener el servicio Web

Tal como podemos apreciar el código de ASP.NET 2.0 se simplifica un poco, porque ya no tenemos que indicarle al runtime de .NET dónde se encuentra nuestro código, lo que si tenemos que seguir indicando es que este fichero realmente es un servicio Web, cómo se llama la clase y que lenguaje vamos a utilizar para escribir el código.

Eliminar ficheros de un proyecto

Para este ejemplo, vamos a quedarnos con este fichero en nuestro proyecto, por tanto el resto de los ficheros que tenemos creados podemos eliminarlos, esto lo haremos desde el propio explorador de soluciones, del cual podemos eliminar todos los ficheros salvo el que vamos a usar en el ejemplo, por tanto dejaremos el fichero **HolaMundoSW.asmx**.

En la figura 6.7 podemos ver el contenido actual de nuestro proyecto y en la figura 6.8 el que debemos dejar después de eliminar el resto de elementos.

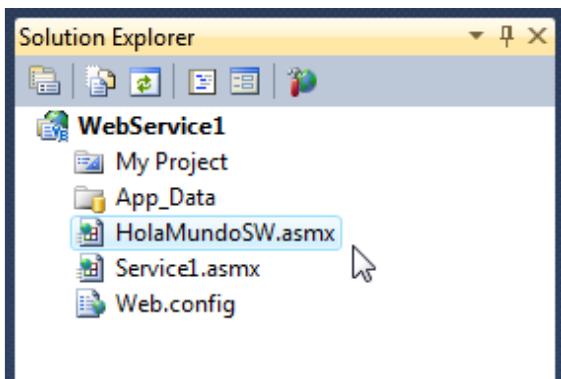


Figura 6.7. El explorador de soluciones antes de eliminar los ficheros

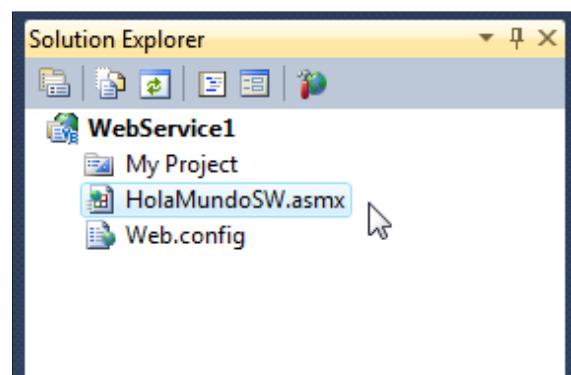


Figura 6.8. El explorador de soluciones con nuestro servicio Web

Nota:

Cuando eliminamos ficheros o carpetas del explorador de soluciones, estos se eliminan físicamente.

Analizando el contenido de un servicio Web

Como podemos comprobar en la figura 6.6, una vez que hemos cerrado la directiva de ASP.NET 2.0 que indica que el contenido de ese fichero es un servicio Web, lo que sigue es el código en el lenguaje de nuestra preferencia, en este caso Visual Basic (VB).

Después de las típicas importaciones de espacios de nombres viene el código que hará que esta clase realmente se convierta en un servicio Web.

Atributos aplicables a los servicios Web

Con lo primero que nos encontramos es con un atributo llamado *WebService*, este atributo no tiene nada que ver con la directiva de ASP.NET y realmente es opcional, es decir, no es necesario para convertir nuestra clase en un servicio Web.

El atributo *WebService* es totalmente opcional, pero siempre deberíamos incluirlo en nuestros servicios Web, por dos razones:

1. Nos servirá para indicar el espacio de nombres en el que estará nuestro servicio Web. Este espacio de nombres no tiene relación directa con los espacios de nombres de nuestras clases, ya que solo se utiliza para identificar los servicios Web. Aunque al igual que los espacios de nombres declarados con la instrucción *Namespace* de Visual Basic, servirá para diferenciarlos de otros servicios Web alojados en nuestro mismo servidor Web o en otros diferentes, de esta forma evitaremos conflictos si en una misma aplicación cliente queremos tener dos servicios Web diferentes pero que utilizan el mismo nombre.
2. La segunda razón es porque en ese mismo atributo podemos indicar una descripción de nuestro servicio Web. Esta descripción será la que se muestre cuando "descubramos" un servicio Web utilizando UDDI.

La recomendación para el espacio de nombres a usar en los servicios Web, al menos para evitar posibles conflictos, es darle el mismo nombre de la ubicación de nuestro servidor, ya que así nos aseguramos que no existirá otro sitio Web que se llame igual que el nuestro.

Nota:

Si no indicamos un espacio de nombres para nuestro servicio Web, ASP.NET utilizará por defecto <http://tempuri.org/> que es el que VWD utiliza de forma predeterminada.

Por tanto vamos a cambiar el valor asignado a la propiedad **Namespace**, para que apunte a nuestro "hipotético" servidor Web: <http://miServidorWeb.com/ServiciosWeb/>, quedando el atributo de la siguiente forma:

```
<WebService(Namespace:="http://miServidorWeb.com/ServiciosWeb/")>
```

Para añadir el valor que la propiedad **Description** tendrá, lo haremos como con el resto de atributos: separándola con una coma del contenido del espacio de nombres, tal como podemos apreciar en el siguiente código:

```
<WebService(Namespace:="http://miServidorWeb.com/ServiciosWeb/", _  
Description:="¡Mi primer servicio Web para saludar al mundo!")>
```

Definición de la clase a usar en el servicio Web

La definición de la clase que será el corazón del servicio Web, (recordemos que un servicio Web en el fondo es una clase que utilizamos desde un sitio de Internet), la haremos como es habitual en Visual Basic, aunque si queremos utilizar ciertas características propias de las aplicaciones ASP.NET como es acceder a objetos *Session* o *Application*, podemos derivarla a partir de la clase *WebService*, aunque esto es totalmente opcional y no influye en la creación de nuestro servicio Web.

```
Public Class HolaMundoSW  
    Inherits System.Web.Services.WebService
```

Lo que si es importante es que el nombre de la clase coincida con la indicada en el atributo de la directiva de ASP.NET.

Añadir métodos para usarlos en el servicio Web

Ahora viene la parte interesante, aunque todos los preparativos preliminares también lo son, pero si no definimos ningún método en nuestra clase, de poco nos servirá el servicio Web, ya que esos métodos serán los que utilicemos para comunicarnos con él.

Los métodos los declararemos de la forma habitual, pero si queremos que sean expuestos como parte del servicio Web, debemos utilizar el atributo *WebMethod*.

Con este atributo le estamos indicando a la "infraestructura" que hay detrás de los servicios Web, que tenga en cuenta ese método para utilizarlo desde nuestra clase. No vamos a entrar en detalles de que es lo que ocurre tras el telón, pero debemos saber que sin ese atributo, nuestro método será totalmente invisible al mundo exterior, aunque aún así lo podríamos seguir usando desde nuestra clase o desde cualquier otra clase que tengamos en nuestro proyecto, ya que al fin y al cabo es "sólo" un método más.

Este atributo lo utilizaremos como de costumbre: indicándolo antes de la declaración del método.

En el siguiente código vemos la declaración de un método que se incluye como plantilla de los servicios Web que creemos con Visual Studio 2010, y que nos puede servir para nuestros propósitos, aunque lo "castellanizaremos" para que sea más fácilmente comprensible para los que hablamos el idioma de Cervantes:

```
<WebMethod()>_  
Public Function HolaMundo() As String  
    Return "¡Hola, Mundo!"  
End Function
```

El atributo *WebMethod* también contiene propiedades, entre ellas una que nos servirá para aclarar que es lo que hace: *Description*.

Al igual que ocurre con el atributo *WebService*, la propiedad **Description** nos servirá para explicarle a los que "descubran" nuestro servicio Web que es lo que hace ese método. Esto es particularmente útil si nuestro servicio Web tiene varios métodos Web y los nombres utilizados no son muy "aclaratorios".

Por tanto, modificaremos el código para que tenga el siguiente aspecto:

```
<WebMethodgetDescription:="Método para saludar al mundo">_  
Public Function HolaMundo() As String
```

Nota:

El atributo *WebMethod* también incluye una propiedad llamada *MessageName*, la cual podemos utilizar para evitar conflictos de sobrecarga de métodos, ya que los protocolos usados actualmente con los servicios Web requieren un identificador único para cada método Web.

Probar nuestro servicio Web

Ya tenemos creado nuestro primer servicio Web, ahora solo nos queda comprobar que todo funciona a la perfección.

La forma más fácil de hacerlo es mostrándolo en el navegador. De esta forma sabremos si funciona e incluso podemos verlo en "acción".

Para mostrar el servicio Web en el navegador, lo seleccionamos en el **Explorador de soluciones**, presionamos con el botón secundario del mouse y del menú desplegable que nos muestra, seleccionamos **Ver en el explorador**, tal como nos muestra la figura 6.9:

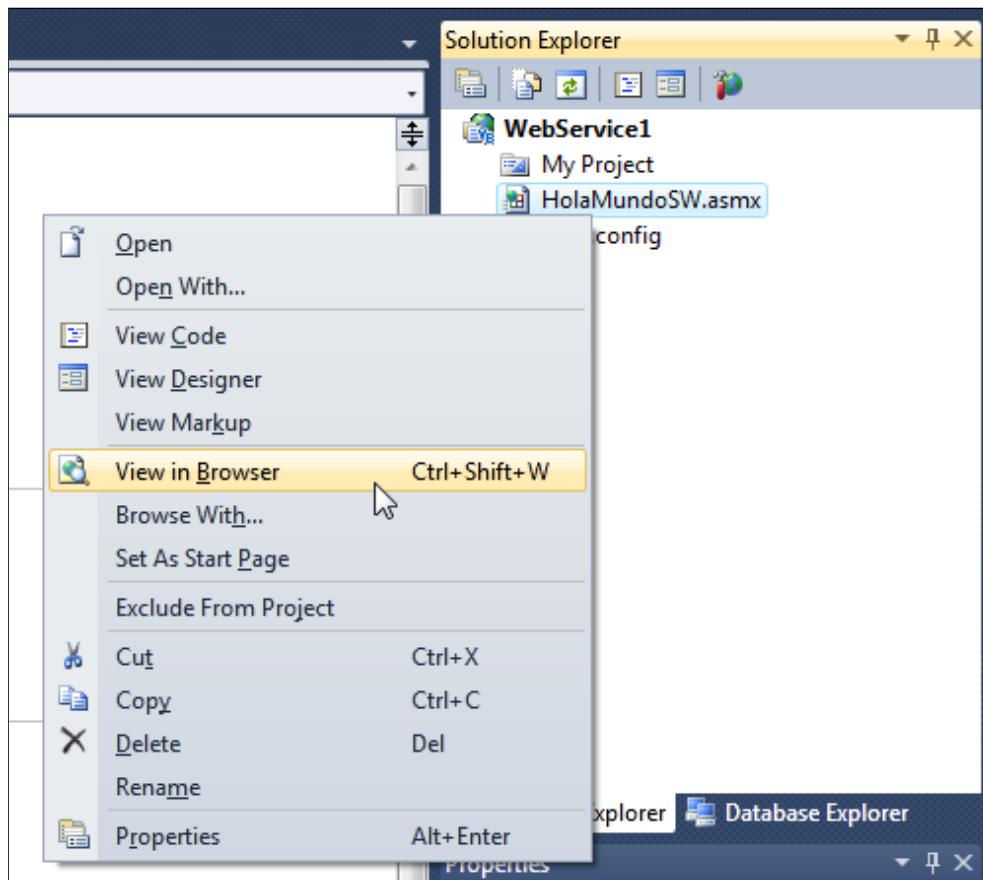


Figura 6.9. Mostar el servicio Web en el navegador

Cuando lo mostramos en el Internet Explorer, podemos ver cierta información del servicio Web, principalmente las descripciones que hemos usado, además de un par de links, que nos mostrarán información extra de nuestro servicio Web y del método.

En las siguientes imágenes (6.10 y 6.11) vemos lo que nos muestra el navegador:

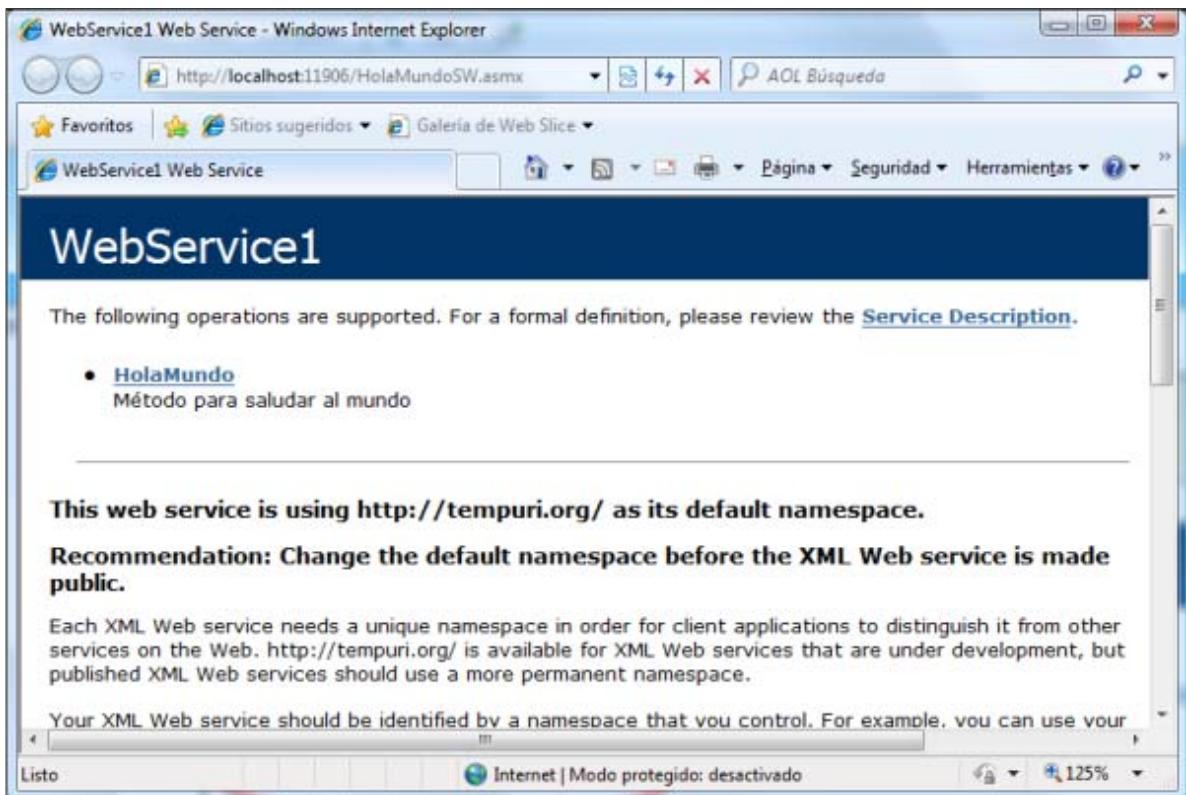


Figura 6.10. El servicio Web al utilizarlo desde Internet Explorer

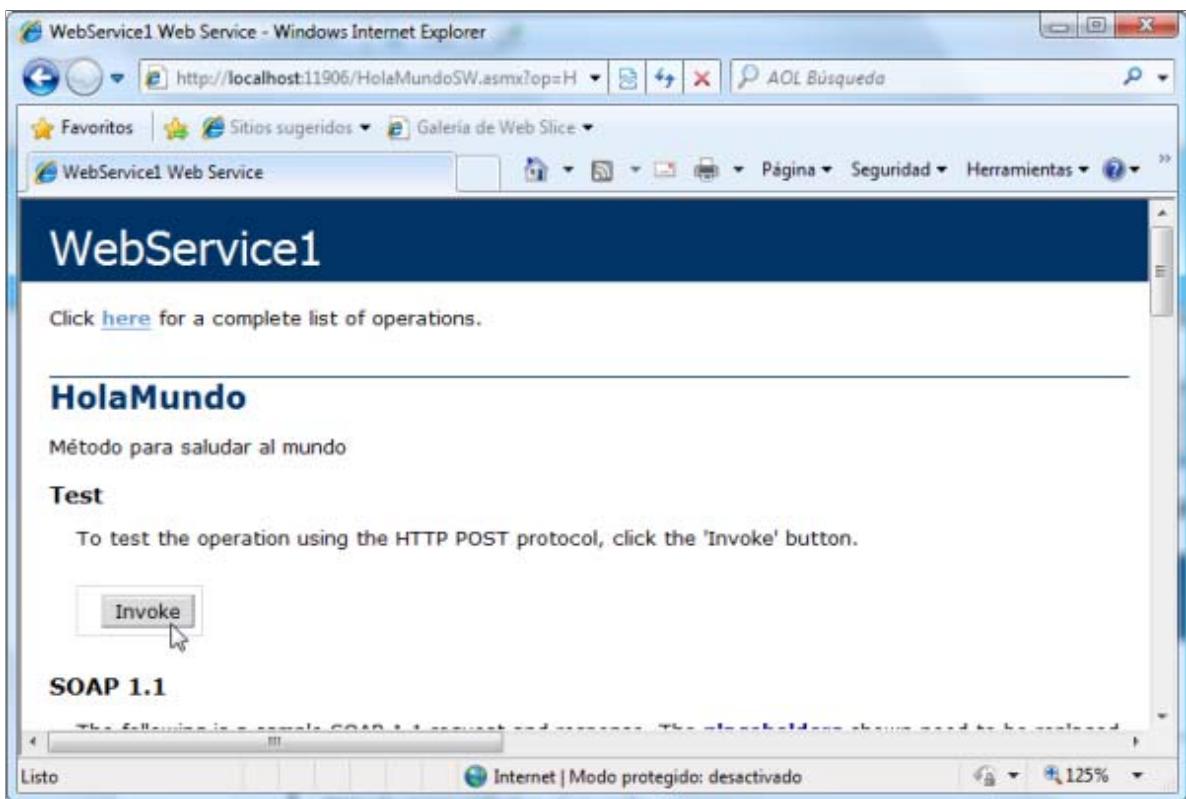


Figura 6.11. Información sobre el método Web HolaMundo

Además de darnos la oportunidad de probar el método, presionando el botón **Invocar**, también podemos ver el código que se utiliza para comunicar nuestro servicio Web con el resto de las aplicaciones.
Pero como comprobaremos en la siguiente lección es totalmente transparente para nosotros,

ya que es el propio Visual Studio 2010, por medio de ASP.NET, el que se encarga de todo el "trabajo sucio", dejándonos a nosotros solo con el código que tenemos que escribir.

Lección 3: Consumo de Servicios Web

- Utilizar los Servicios Web desde una aplicación Windows

Consumo de servicios Web

En esta lección veremos cómo "consumir" el servicio Web creado anteriormente desde una aplicación de Windows Forms utilizando el entorno de desarrollo (IDE) de Visual Studio 2010.

Como comprobaremos, el propio compilador de Visual Basic se encargará de todos los pormenores para que podamos utilizar la clase incluida en el servicio Web como si de una clase normal y corriente se tratara, ya que, esa es una de las ventajas de apoyarse en un entorno integrado de la calidad de Visual Studio 2010, que sólo debemos preocuparnos de escribir el código y del resto... ¡nos olvidamos! (o casi).

• Utilizar los servicios Web desde una aplicación de Windows

- Utilizar los servicios Web desde una aplicación de Windows
 - Alojar el servicio Web en un servidor local
 - Activar el servidor Web para usar con un directorio local
- Crear un proyecto Windows para usar el servicio Web
 - Añadir una referencia para acceder al servicio Web
 - ¿Dónde están las referencias en Visual Basic 2010?
 - Acceder al servicio Web desde el código
 - ¿Qué es lo que puede fallar?

Lección 3: Consumo de Servicios Web

Utilizar los Servicios Web desde una aplicación Windows

Utilizar los servicios Web desde una aplicación de Windows

Ahora vamos a ver cómo podemos "consumir" un servicio Web desde una aplicación de Windows.

Alojar el servicio Web en un servidor local

Para que nuestro ejemplo funcione, tenemos que copiar el servicio Web que creamos en la lección anterior en el directorio al que hace referencia el Web local de nuestro equipo (localhost). Si no hemos cambiado la ubicación que tiene de forma predeterminada, estará en la carpeta **C:\Inetput\wwwroot**, esto si es que tenemos el IIS (Internet Information Server) instalado en nuestro equipo.

Si no hemos instalado el IIS, (con las versiones anteriores de Visual Studio .NET era un requisito para poder crear cualquier tipo de proyecto Web con ASP.NET), no debemos preocuparnos, ya que la versión 2.0 de .NET Framework incluye su propio servidor Web, y con estos sencillos pasos vamos a ponerlo en marcha para probar nuestros servicios Web.

Nota:

Si quiere mayor diversión y no copiar ningún servicio web en su máquina, acceda a cualquier servicio web público de Internet. Existen de todo tipo, desde servidores de hora hasta complejos sistemas de mapas. Le proponemos algunos ejemplos:

- Lista de servicios web gratuitos: <http://www.webservicex.net/WS/wscatlist.aspx>
- Fotos por satélite: <http://terraservice.net/webservices.aspx>
- MapPoint (evaluación): <https://mappoint-css.partners.extranet.microsoft.com/MwsSignup/Eval.aspx>
- Servicios web de Amazon: <http://www.amazon.co.uk/exec/obidos/tg/browse/-/3242161/026-5630606-4874040>

Activar el servidor Web para usar con un directorio local

Si no dispone de IIS y quiere alojar un servicio web en su máquina, éstos son los pasos que debemos dar activar el servidor Web incluido con .NET Framework 2.0:

1. Lo primero que debemos hacer es crear un directorio en cualquier unidad de nuestro equipo, en estos ejemplos vamos a utilizar el directorio **E:\VS2008B2\sitioWeb**, pero

- puede ser cualquier otro, lo importante es que este será el que utilizaremos en estos pasos.
2. En ese directorio copiaremos nuestro servicio Web, en concreto el fichero .asmx creado en la lección anterior.
 3. Abrimos una ventana de comandos y nos cambiamos al directorio **C:\WINDOWS\Microsoft.NET\Framework\v2.0.50215**, que es en el que están los ejecutables de .NET Framework 2.0.
 4. Escribimos lo siguiente: **WebDev.WebServer.EXE /port:8080 /path:"E:\VS2008B2\ sitioWeb" /vpath:"/"**
Esto hará que se inicie el servidor Web de .NET permitiéndonos acceder a nuestro servicio Web usando la siguiente dirección: **http://localhost:8080/HolaMundoSW.asmx**
 5. De esta forma, hasta que no desconectemos el servidor Web, este estará en ese puerto, tal como podemos ver en la figura 6.12. Para no perder la conexión, debemos dejar abierta la ventana de comandos.

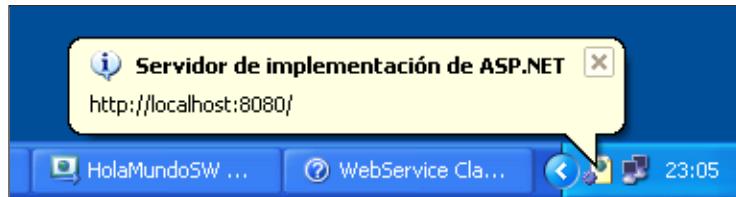


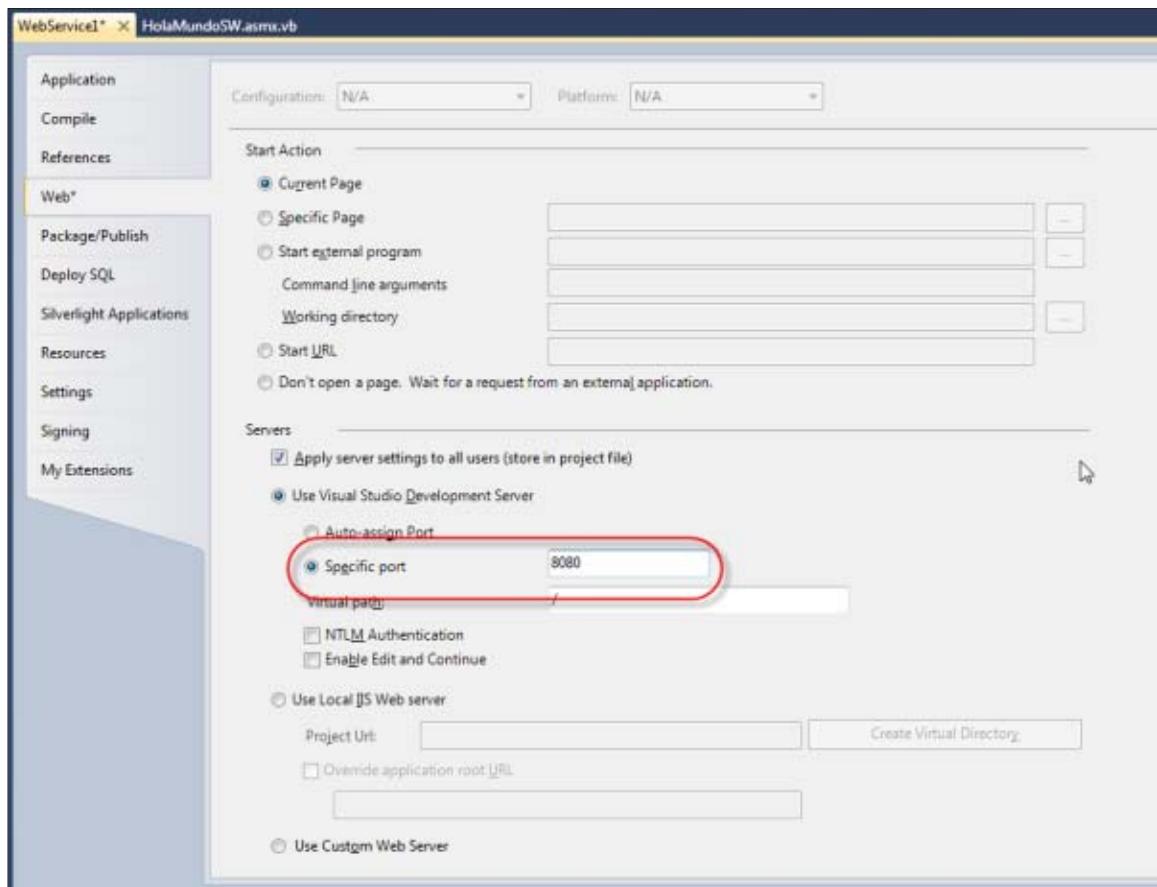
Figura 6.12. Servidor Web local activado en el puerto 8080

Nota:

Estos pasos solo son necesarios si no tenemos instalado y en ejecución el ISS, en cuyo caso tendríamos que copia el servicio Web en el directorio C:\Inetput\wwwroot y usar <http://localhost/HolaMundoSW.asmx> para acceder al servicio Web.

En otro orden de cosas, Visual Studio 2010 lanza un Servidor Web virtual, conocido como Cassini, que permite ejecutar aplicaciones Web en entornos de desarrollo sin IIS.

No obstante, si accedemos a las propiedades del proyecto, podremos indicar igualmente el puerto específico que queremos usar, tal y como se muestra en la siguiente imagen:



Crear un proyecto Windows para usar el servicio Web

Lo siguiente que tenemos que hacer es abrir el Visual Studio 2010 y crear un nuevo proyecto de Windows, al que le daremos el nombre **ClienteWindows**. De forma automática se creará un formulario, al que le añadimos una etiqueta a la que asignaremos a la propiedad *Font* una fuente de 16 puntos en negrita, para que se vea bien el texto, y le cambiamos el nombre a: **labelSaludo**. También añadiremos un botón al que le daremos el nombre **btnUsarServicioWeb**, en el texto pondremos: **Saludo desde servicio Web**, como es natural, tendremos que cambiar el tamaño para que veamos el texto completo.

Añadir una referencia para acceder al servicio Web

Al igual que ocurre con el resto de ensamblados de .NET, si queremos acceder a la clase que tiene el servicio Web, debemos crear una referencia en nuestro proyecto, con idea de que tengamos a nuestra disposición las clases que contiene, en nuestro caso la clase **HolaMundoSW**. Pero como es un servicio Web, que teóricamente está alojado en algún servidor de Internet, en lugar de agregar una referencia normal, añadiremos una referencia Web. Para hacerlo, nos iremos al **Explorador de soluciones / References**, presionaremos con el botón secundario del mouse para que se muestre el menú contextual, del que elegiremos la opción **Agregar referencia Web**, tal como vemos en la figura 6.13:

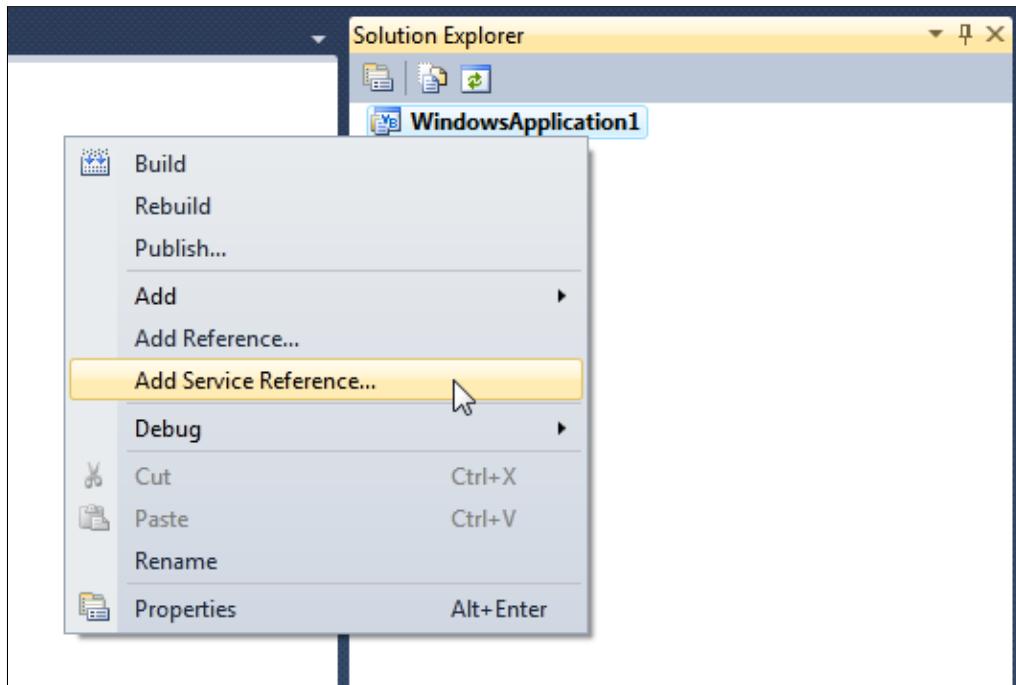


Figura 6.13. Agregar una referencia Web

Nota: ¿Dónde están las referencias en Visual Basic 2010?

En Visual Basic 2010, hay ciertos elementos que se ocultan al usuario, entre ellos las referencias que tenemos en nuestro proyecto. Entre ellos algunos de los que se incluyen en el Explorador de soluciones.

Para mostrar todas las opciones de nuestro proyecto, debemos "obligar" al IDE de Visual Basic 2010 a que nos muestre esos ficheros y opciones ocultas, para hacerlo presionaremos el segundo botón de la barra de herramientas del Explorador de soluciones, tal como comentamos en la lección 5 del módulo 1.

Acto seguido aparecerá un cuadro de diálogo que nos servirá para indicar dónde está alojado el servicio Web, si hemos seguido los pasos indicados anteriormente, el servicio Web estará alojado en <http://localhost:8080/HolaMundoSW.asmx>, por tanto será eso lo que escribiremos en el combo que hay junto a URL, una vez "descubierto" el servicio Web, se mostrará la información, tal como lo hacía el Internet Explorer, y ya estaremos listos para agregar la referencia Web, para ello tendremos que presionar en el botón **Agregar referencia**, tal como vemos en la figura 6.14.

Nota:

Si está utilizando un servicio web público, introduzca la URL del servicio en lugar de <http://localhost:8080/HolaMundoSW.asmx>. La dirección de los servicios suele venir indicada en la documentación de ayuda del servicio y corresponde al archivo WSDL que define el interfaz del mismo.

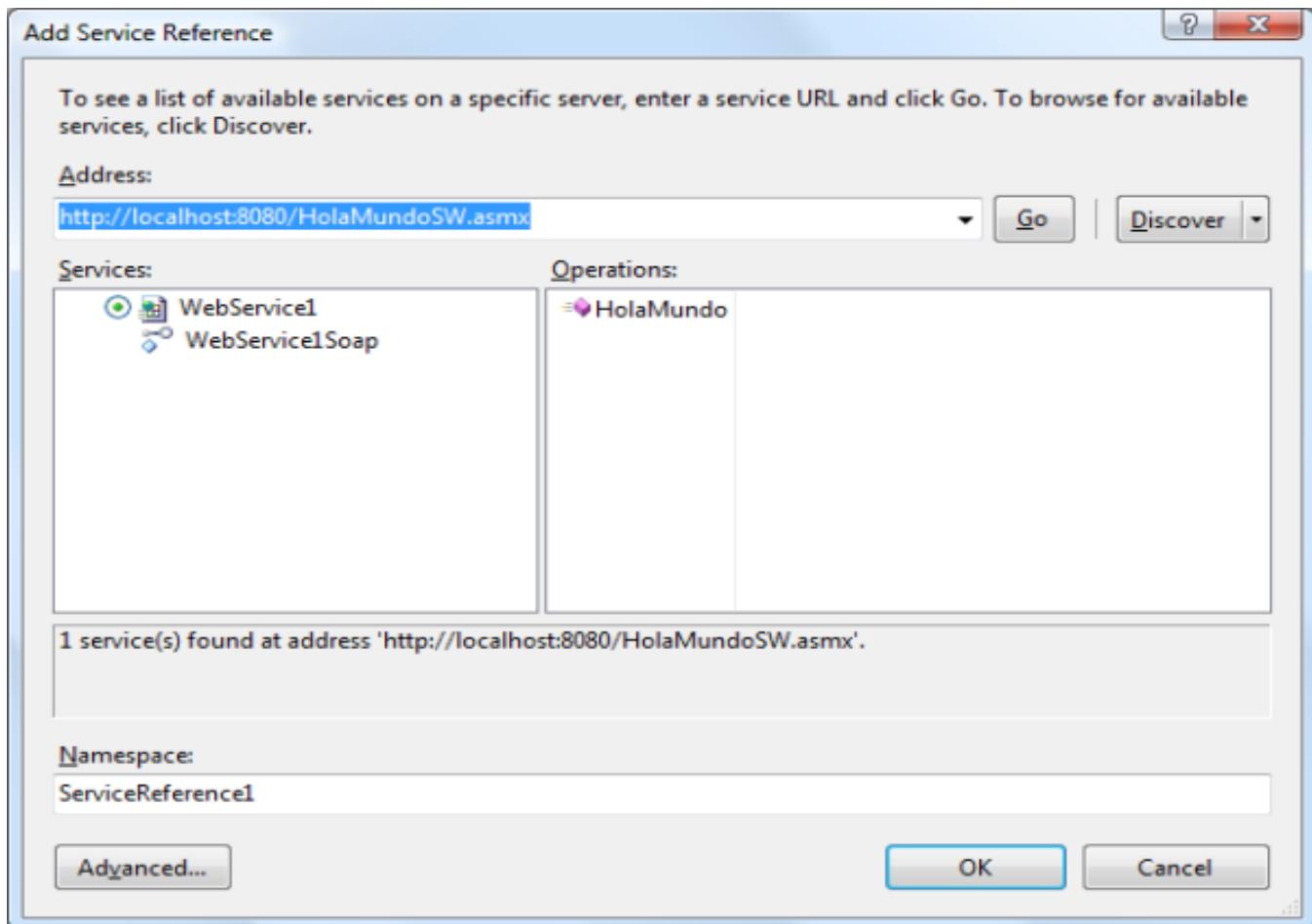


Figura 6.14. Agregar una referencia Web a nuestro proyecto

Una vez hecho esto, tendremos nuestra clase en un espacio de nombres que se llamará tal como se indica en la caja de textos que hay bajo **Nombre de referencia Web**.

Nota:

Los servicios Web que podemos utilizar en nuestros proyectos, no tienen por qué estar hechos con el mismo lenguaje de programación que el utilizado en el cliente. Ya que a lo que estamos accediendo, aunque no nos percatemos, y por simplificar, es a un ensamblado de .NET, por tanto, los servicios Web pueden estar escritos por cualquier lenguaje de .NET que permita la creación de los mismos.

Acceder al servicio Web desde el código

Ya solo nos queda escribir el código para acceder a la clase, para ello declararemos una variable cuyo tipo de datos será la clase **HolaMundoSW**, la instanciaremos y accederemos al método **HolaMundo**, el cual, como sabemos, devuelve una cadena, que asignaremos a la etiqueta **labelSaludo**.

Lo primero que tenemos que hacer es crear el método que detectará la pulsación en el botón, a estas alturas no creo que haya que decir cómo hacerlo..

```
Private Sub btnUsarServicioWeb_Click(ByVal sender As Object, ByVal e As EventArgs) _
    Handles btnUsarServicioWeb.Click
    Dim hsw As New localhost.HolaMundoSW
    labelSaludo.Text = hsw.HolaMundo()
End Sub
```

Y ya podemos probar a ver si funciona, para ello, presionamos **F5** y si todo va bien, se ejecutará

la aplicación y podremos presionar el botón para obtener el resultado mostrado en la figura 6.15:



Figura 6.15. El mensaje devuelto por el servicio Web

¿Qué es lo que puede fallar?

Si ha seguido correctamente estos pasos el servicio web debe funcionar sin problemas. En caso de error asegúrese de que el servicio web está funcionando correctamente. Para ello puede usar su navegador de Internet y acceder directamente a la dirección del servicio. Si no le funciona y es un servidor remoto de Internet, el error puede venir de su conectividad con Internet o la dirección que consiguió del servicio. Si es un servicio local y no puede acceder a él, asegúrese de que el servidor web está arrancado, bien sea IIS o el propio servidor incluido con el Framework .NET 2.0.

Módulo 7: LINQ

- Para qué LINQ
- LINQ to Objects
- LINQ to XML
- LINQ y ADO.NET
- LINQ to DataSet
- LINQ to SQL
- LINQ to Entities

Contenido

En este módulo veremos los aspectos generales y particulares de LINQ

A continuación se exponen los apartados que se verán en este módulo.

- [Para qué LINQ](#)
 - [LINQ to Objects](#)
 - [LINQ to XML](#)
 - [LINQ y ADO.NET](#)
 - [LINQ to DataSet](#)
 - [LINQ to SQL](#)
 - [LINQ to Entities](#)
-

Módulo 7: LINQ

Para qué LINQ

- LINQ to Objects
- LINQ to XML
- LINQ y ADO.NET
- LINQ to DataSet
- LINQ to SQL
- LINQ to Entities

Para qué LINQ

LINQ o Language INtegrated Query es una de las novedades más importantes de Microsoft .NET Framework 3.5 y que afecta por lo tanto a Visual Studio 2010 y con ello a Visual Basic 2010.

Nota: LINQ no está disponible en Visual Studio 2005 ni versiones anteriores.

LINQ constituye ante todo una revolución a la hora de desarrollar aplicaciones Software. Mucha gente no lo ve así, pero es demasiado pronto como para que el fenómeno LINQ termine de calar entre los desarrolladores, si bien, quienes lo utilizan, quedan prendados de él.

La misión fundamental de LINQ es la de enlazar los datos con la programación y los lenguajes de programación tratando de eliminar las barreras que separaban a los datos de la programación, y dotando por otro lado a la aplicación, de más seguridad a los programadores a la hora de trabajar con datos.

LINQ entre otras cosas, nos permite comprobar los tipos de datos con los que estamos operando en tiempo de compilación, por lo que reducimos en gran medida los riesgos que solemos arrastrar los programadores en muchos proyectos Software.

Pero todo esto no tendría lugar sino fuera por los iteradores y las colecciones, aspectos todos ellos, que toman una especial importancia a la hora de trabajar con LINQ.

Para trabajar con LINQ, utilizaremos en muchas situaciones la interfaz *IQueryable* y *IEnumerable* que se encuentra en el nombre de espacio **System.Linq** del ensamblado **System.Core.dll**. Por defecto, al iniciar un proyecto con Visual Studio 2010, el entorno generará la estructura base de la aplicación y agregará las referencias necesarias para trabajar con LINQ.

¿Y qué es lo que nos permite LINQ?. LINQ nos permite por lo tanto, trabajar con datos, colecciones y elementos de una forma muy precisa, utilizando para ello sentencias integradas muy similares al lenguaje SQL.

Orígenes de datos LINQ

Para trabajar con colecciones, datos y elementos con LINQ, Microsoft nos ha proporcionado una serie de objetos o extensiones que nos permitirán interactuar con los datos adecuados y LINQ.

De acuerdo a la Arquitectura de LINQ, ésta se puede asemejar a la que se indica en la siguiente imagen:

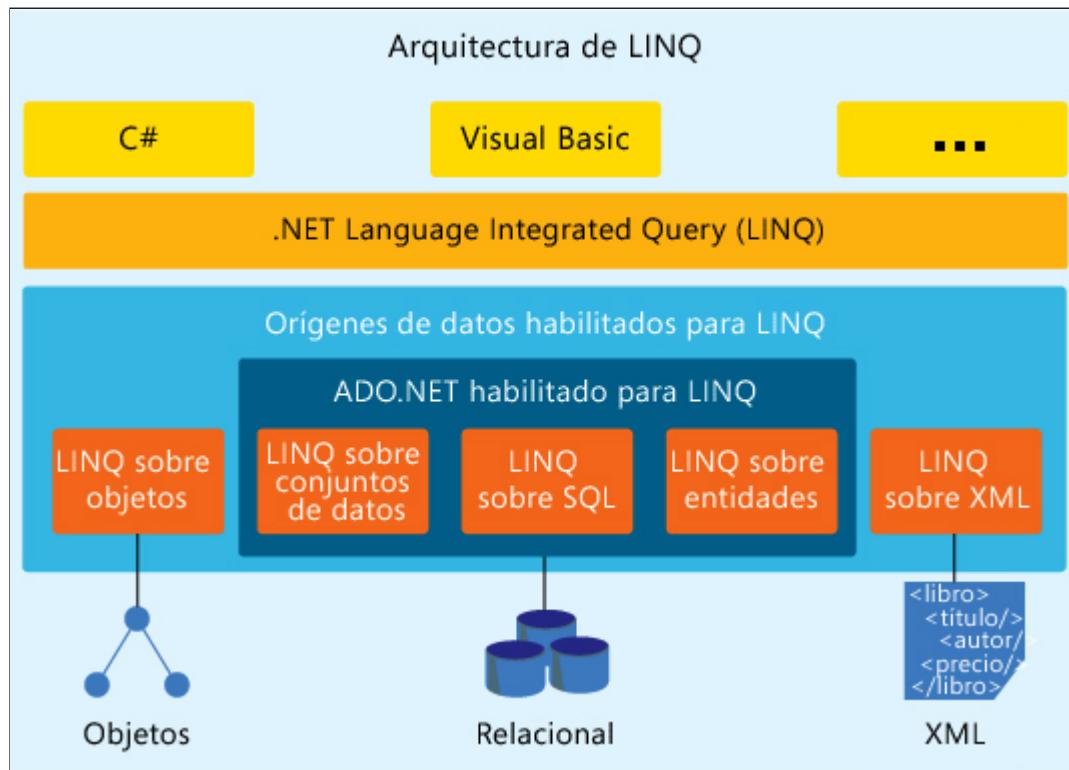


Figura 7.1.- Arquitectura de LINQ

En esta arquitectura, podemos ver a los lenguajes de programación en la capa más alta, y a continuación LINQ como lenguaje integrado de consultas, y debajo de éste, los orígenes de datos o extensiones que podremos utilizar con LINQ.

Respecto a las extensiones, debemos destacar las que pertenecen a los objetos de datos relaciones, las que tienen relación con los objetos propiamente dichos, y las que permiten trabajar con XML. Todos estos extensores, nos permiten trabajar con LINQ de una forma muy parecida.

Podría haber más extensores, incluso podríamos crear el nuestro propio. De hecho, en Internet hay diferentes proyectos, muchos de ellos de carácter código abierto, que tienen por misión crear extensores de LINQ para utilizar con .NET Framework 3.5. Muchos de esos proyectos, pueden encontrarse en [CodePlex](#).

Así por lo tanto, tenemos dentro de .NET Framework 3.5 los siguientes extensores:

- Si queremos trabajar con objetos, lo razonable es que trabajemos con LINQ to Objects.
- Si lo que queremos es trabajar con documentos XML, utilizaremos LINQ to XML.
- Pero si queremos acceder y manipular datos, entonces podremos trabajar con tres diferentes extensores dependiendo de nuestras necesidades. Estos extensores son LINQ to DataSet, también conocido como LINQ sobre conjunto de datos (recordemos el capítulo de datos donde se hablaba de DataSets), LINQ to SQL para trabajar con bases de datos SQL Server, y LINQ to Entities para trabajar con entidades, o lo que es lo

mismo, con cualquier fuente de datos, no solo con SQL Server.

A continuación veremos como usar cada uno de los extensores de LINQ en Visual Basic 2010.

Módulo 7: LINQ

- Para qué LINQ
- LINQ to Objects**
- LINQ to XML
- LINQ y ADO.NET
- LINQ to DataSet
- LINQ to SQL
- LINQ to Entities

LINQ to Objects

Con LINQ to Objects trabajaremos con objetos, matrices y colecciones de datos que están en memoria, desde nuestras aplicaciones escritas en Visual Basic 2010.

Con LINQ to Objects, podemos trabajar con LINQ siempre y cuando los objetos con los que queremos trabajar, implementen las interfaces **IEnumerable** e **IEnumerable<T>**.

Nota aclaratoria: recuerde que si una clase empieza por **I** seguida del nombre de la clase, suele indicar según la nomenclatura recomendada por Microsoft, que se trata de una interfaz. También es posible que encuentre en diferentes documentos, **IEnumerable<T>** como **IEnumerable(Of T)**. Ambas significan lo mismo, y tienen relación directa con los datos tipados de la interfaz, o dicho de otra forma, tiene relación con los datos genéricos. En la documentación general de MSDN, lo encontrará siempre como **IEnumerable<T>**.

Para recorrer los elementos de una colección, utilizaremos normalmente el bucle **For Each**.

El ensamblado **System.Linq** que se incluye por defecto en cualquier aplicación de Visual Studio 2010 y que reside en **System.Core.dll**, nos permite utilizar directamente LINQ dentro de nuestras aplicaciones. Si eliminamos la referencia a este ensamblado, no podremos utilizar LINQ en nuestras aplicaciones.

Gracias a LINQ, podemos utilizar otra nomenclatura diferente para acceder a los elementos de un objeto o colección que están en memoria, algo que veremos en el siguiente apartado.

Practicando con LINQ to Objects

En este primer ejemplo, mostraremos como trabajar con una matriz de datos. Es un sencillo ejemplo que nos permitirá comprender mejor como funciona LINQ to Objects.

Lo primero que haremos es iniciar con Visual Studio 2010 un nuevo proyecto de Visual Basic 2010 y de tipo aplicación Windows.

Dentro del formulario Windows insertaremos un control **Button**, y a continuación, escribiremos el siguiente código fuente para nuestra aplicación.

Código

```
Public Class Form1

    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button1.Click
        ' Declararemos la matriz y la inicializaremos con valores
        Dim material() As String = {"mesa", "reloj", "libro", "pluma", "borrador", "pelota", "botella"}
        ' Ejecutamos LINQ para obtener los datos buscados
        ' Concretamente, buscamos los materiales que empiezan por "b"
        Dim materialBuscado = From busqueda In material _
                               Where busqueda.StartsWith("b") _
                               Select busqueda
        ' En materialBuscado tendremos toda la selección realizada, por lo
        ' que deberemos recorrerla para extraer los elementos encontrados
        Dim resultado As String = ""
        For Each elementos In materialBuscado
            resultado &= elementos & vbCrLf
        Next
        ' Mostramos la información
        MessageBox.Show(resultado)
    End Sub
```

```
End Class
```

En el ejemplo que acabamos de preparar, podemos ver como con LINQ, como lenguaje integrado de consultas, hemos accedido a los elementos de la matriz que están en la memoria.

Como habrá podido comprobar, LINQ es en su estructura, muy similar a SQL. Conviene destacar, que la forma de aprender a utilizar y manejar LINQ adecuadamente, es a base de práctica. También es interesante recalcar, que LINQ representa una revolución para el programador, y que Microsoft está haciendo grandes esfuerzos por dar a conocer y extender LINQ como un nuevo paradigma en el desarrollo de aplicaciones Software.

Si analizamos un poco el ejemplo que hemos realizado, veremos que dentro del código de Visual Basic podemos utilizar cláusulas como From, Where o Select, más propias de SQL que de un lenguaje de programación como Visual Basic. LINQ por lo tanto, nos proporciona o nos acerca el lenguaje natural de SQL para trabajar directamente con él en nuestras aplicaciones, seleccionando los elementos que queremos, proporcionando cláusulas de condición, u ordenando los elementos para obtenerlos en su salida directamente tal y como los queremos.

Dentro del ejemplo de LINQ, podemos comprobar que para seleccionar un subconjunto de elementos de un conjunto principal, utilizamos una serie de operandos diferentes. En primer lugar la cláusula u operando **From** en la que indicamos un alias dentro del conjunto de elementos *From <alias> In <conjunto_de_elementos>*. Posteriormente y dentro de la misma cláusula, utilizamos la cláusula u operando **Where** que nos permite indicar la condición o condiciones que queremos utilizar. Finalmente y en el ejemplo realizado, utilizamos la cláusula u operando **Select** que tiene por misión seleccionar el elemento o conjunto de elementos que queremos extraer.

A continuación, expondremos un ejemplo un poco más completo del uso de LINQ to Object. Aprovechando la base del ejemplo anterior, escribiremos una clase *Empleado* y luego, consumiremos esa clase para trabajar con ella desde LINQ to Objects. Veremos lo sumamente fácil, rápido e intuitivo que resulta trabajar con LINQ to Objects en Visual Basic 2010.

Código

```
Class Empleado
    ' Nombre del empleado
    Private m_Nombre As String
    Public Property Nombre() As String
        Get
            Return m_Nombre
        End Get
        Set(ByVal value As String)
            m_Nombre = value
        End Set
    End Property

    ' NIF del empleado
    Private m_NIF As String
    Public Property NIF() As String
        Get
            Return m_NIF
        End Get
        Set(ByVal value As String)
            m_NIF = value
        End Set
    End Property

    ' Edad del empleado
    Private m_Edad As Byte
    Public Property Edad() As Byte
        Get
            Return m_Edad
        End Get
        Set(ByVal value As Byte)
            m_Edad = value
        End Set
    End Property
End Class

Public Class Form1

    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button1.Click
        ' Declaramos la matriz y la inicializamos con valores
        Dim personal() As Empleado = {New Empleado() With {.Nombre = "Juan", .NIF = "111A", .Edad = 32}, _
                                       New Empleado() With {.Nombre = "María", .NIF = "222C", .Edad = 43}, _
                                       New Empleado() With {.Nombre = "José", .NIF = "333R", .Edad = 45}, _
                                       New Empleado() With {.Nombre = "Rosa", .NIF = "333F", .Edad = 33}, _
                                       New Empleado() With {.Nombre = "Javier", .NIF = "444G", .Edad = 41}}
        ' Ejecutamos LINQ para obtener los datos buscados
        ' Concretamente, buscamos los empleados cuya edad es mayor de 40 años
        Dim empleadosBuscados = From busqueda In personal -
                               Where busqueda.Edad > 40 -
                               Select busqueda.Nombre, busqueda.Edad -
                               Order By Edad Descending
        ' En empleadosBuscados tendremos toda la selección realizada, por lo
        ' que deberemos recorrerla para extraer los elementos encontrados
        Dim resultado As String = ""
    End Sub
End Class
```

```
For Each elementos In empleadosBuscados
    resultado &= String.Format("{0} tiene {1} años", elementos.Nombre, elementos.Edad) & vbCrLf
Next
' Mostramos la información
MessageBox.Show(resultado)
End Sub

End Class
```

Nuevo en Visual Basic 2010: Una nueva característica de Visual Basic 2010, es la que permite inicializar la clase y sus propiedades directamente. Para ello, utilizaremos la cláusula **With** tal y como podemos ver en el ejemplo anterior.

Analizando el ejemplo que acabamos de ver, vemos que en este caso estamos trabajando con objetos *Empleado* que están en memoria, y con sus datos inicializados en la aplicación.

Dentro de la sentencia LINQ que estamos utilizando, hemos seleccionado dos elementos de los datos con los que estamos trabajando, y hemos realizado la ordenación de sus datos de acuerdo al operando o cláusula **Order By**. Incluso le hemos indicado el orden de ordenación mediante el uso de la palabra reservada **Descending**, indicando así que queremos ordenar los elementos por el campo **Edad** y en orden descendiente. Por defecto, la ordenación es en sentido ascendente y no hace falta indicarlo.

A la hora de trabajar con el entorno de desarrollo, *Intellisense* constituye para nosotros una herramienta de productividad enorme, ya que es capaz de indicarnos directamente en LINQ los campos disponibles, permitiéndonos así, seleccionar el campo o campos que deseamos utilizar.

Módulo 7: LINQ

- Para qué LINQ
- LINQ to Objects
- LINQ to XML**
- LINQ y ADO.NET
- LINQ to DataSet
- LINQ to SQL
- LINQ to Entities

LINQ to XML

Con LINQ to XML trabajaremos con documentos XML desde nuestras aplicaciones escritas en Visual Basic 2010.

LINQ to XML en Visual Studio 2010, posee diferencias notables y ventajosas con respecto a C# 3.0. Muchos son los programadores de C# que desean incluir algunas de las características que están incluidas en LINQ to XML para Visual Basic 2010.

Visual Basic 2010 proporciona por lo tanto, una integración total con XML, que permite manipular los documentos XML directamente.

Pruebe esto: copie al portapapeles de Windows el contenido de un documento XML y escriba en Visual Studio 2010, y dentro de su código de Visual Basic 2010 la siguiente instrucción: **Dim variable =** y a continuación pegue ahí el contenido del documento XML que tendrá en el portapapeles de Windows. Observará que Visual Basic 2010 es capaz de tratar el documento XML de forma directa.

Para trabajar con LINQ to XML, Microsoft nos ha proporcionado un nuevo ensamblado de nombre **System.Linq.Xml.dll**. A través de Visual Basic 2010, podemos utilizar los literales XML de forma directa, permitiéndonos explotar los documentos XML de una forma mucho más ágil y transparente, aumentando enormemente así la productividad de nuestros desarrollos.

Practicando con LINQ to XML

A continuación, realizaremos un ejemplo de como trabajar con LINQ to XML.

Para realizar el siguiente ejemplo, nos apoyaremos en el siguiente documento XML:

Código

```
<?xml version='1.0'?>
<!-- Este documento XML representa el inventario de coches -->
<coches>
    <coche marca="BMW">
        <modelo>520</modelo>
        <potencia>125CV</potencia>
    </coche>
    <coche marca="BMW">
        <modelo>525</modelo>
        <potencia>135CV</potencia>
    </coche>
    <coche marca="Citroen">
        <modelo>C3</modelo>
        <potencia>75CV</potencia>
    </coche>
    <coche marca="Citroen">
        <modelo>C4</modelo>
        <potencia>115CV</potencia>
    </coche>
    <coche marca="Citroen">
        <modelo>C5</modelo>
        <potencia>135CV</potencia>
    </coche>
</coches>
```

Insertaremos el documento XML anterior dentro del código de nuestra aplicación para utilizarlo de forma directa.

Basándonos en la base del ejemplo anterior, escribiremos un ejemplo que utilice el documento XML anterior en bruto. El código de nuestro ejemplo, quedará por lo tanto de la siguiente forma:

Código

```
Public Class Form1

    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button1.Click
        Dim datos = SelectXML(120)
        Dim resultado As String = ""
        For Each dato In datos
            resultado &= String.Format("{0} tiene un coche {1}", dato.marcaCoche, dato.modeloCoche) & vbCrLf
        Next
        MessageBox.Show(resultado)
```

```

End Sub

Private Function SelectXML(ByVal potencia As Byte) As Object
    Dim documentoXML = <?xml version='1.0'?>
        <!-- Este documento XML representa el inventario de coches -->
        <coches>
            <coche marca="BMW">
                <modelo>520</modelo>
                <potencia>125</potencia>
            </coche>
            <coche marca="BMW">
                <modelo>525</modelo>
                <potencia>135</potencia>
            </coche>
            <coche marca="Citroen">
                <modelo>C3</modelo>
                <potencia>75</potencia>
            </coche>
            <coche marca="Citroen">
                <modelo>C4</modelo>
                <potencia>115</potencia>
            </coche>
            <coche marca="Citroen">
                <modelo>C5</modelo>
                <potencia>135</potencia>
            </coche>
        </coches>
    Return From datos In documentoXML...<coche> _
        Where datos.<potencia>.Value > potencia _
        Select marcaCoche = datos.@marca, modeloCoche = datos.<modelo>.Value
End Function

End Class

```

En este ejemplo, podemos ver cosas muy curiosas.

Por un lado, la forma en la que accedemos a los bloques del documento XML lo hacemos mediante la instrucción **documentoXML**.... El uso de ... nos permite situarnos en cada bloque de datos del documento XML, o lo que es lo mismo, en cada bloque de coche.

Por otro lado, para extraer el valor de un campo hijo en el bloque determinado, lo haremos utilizando su etiqueta entre los caracteres < y >, y usando el término **Value**, como por ejemplo datos..Value.

Finalmente, para utilizar el valor de una etiqueta padre, utilizaremos el carácter @, como en el caso de **datos.@marca**.

Ahora bien, ¿que pasaría si el documento XML en lugar de estar incrustado dentro del código estuviera en una carpeta del disco duro?. En ese caso, podremos cargar el documento XML en memoria para utilizarlo adecuadamente.

A continuación, veremos un ejemplo de como usar esta característica:

Código
<pre> Public Class Form1 Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button1.Click Dim documentoXML As XDocument = XDocument.Load("C:\DesarrollaConMSDN.xml") Dim datosSeleccionados = From datos In documentoXML...<coche> _ Where datos.@marca.ToUpper() = "BMW" _ Select modeloCoche = datos.<modelo>.Value, potenciaCoche = datos.<potencia>.Value Dim resultado As String = "" For Each dato In datosSeleccionados resultado &= String.Format("{0} tiene un coche {1} de {2}CV", "BMW", dato.modeloCoche, dato.potenciaCoche) & vbCrLf Next MessageBox.Show(resultado) End Sub End Class </pre>

Al ejecutar este ejemplo, veremos que el comportamiento no cambia demasiado, de hecho, el funcionamiento es exactamente el mismo, con la salvedad de que en un caso el documento XML se ha cargado en bruto, y en este último caso, se ha cargado desde una unidad de disco duro.

Trabajando con literales y LINQ to XML

Sin embargo, dentro del uso de documento XML y Visual Basic 2010, hay algo que resulta realmente atractivo para el programador. Estoy hablando del uso de literales.

A continuación, veremos un ejemplo del uso de literales, que es a mi juicio, la mejor forma de entender en qué consiste esta característica agregada a Visual Basic 2010.

Basándonos una vez más en la base de los ejemplos anteriores, es decir, un formulario Windows con un control **Button**, vamos a desarrollar un ejemplo que nos muestre el uso de literales.

Para llevar a cabo nuestra tarea, vamos a agregar nuestro formulario un control **WebBrowser**. En mi caso, y para darle una estética más visual, he decidido agregar también dos controles **Panel** y he jugado con la propiedad **Dock** de los controles insertados en el formulario.

Una vez hecho esto, vamos a escribir el código de nuestro ejemplo que quedará de la siguiente forma:

Código
<pre> Public Structure Vehiculo Public Enum TipoVehiculo camion coche moto End Enum </pre>

```

Public Tipo As TipoVehiculo
Public Matricula As String
Public Velocidad As Integer
End Structure

Public Class Form1

    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button1.Click
        Dim listaVehiculos As New List(Of Vehiculo)
        Dim vehiculoUno As New Vehiculo With {.Matricula = "0000AAA", .Tipo = Vehiculo.TipoVehiculo.cochе, .Velocidad = 235}
        listaVehiculos.Add(vehiculoUno)
        Dim vehiculoDos As New Vehiculo With {.Matricula = "1111AAA", .Tipo = Vehiculo.TipoVehiculo.moto, .Velocidad = 265}
        listaVehiculos.Add(vehiculoDos)
        Dim vehiculoTres As New Vehiculo With {.Matricula = "2222AAA", .Tipo = Vehiculo.TipoVehiculo.cochе, .Velocidad = 190}
        listaVehiculos.Add(vehiculoTres)
        Dim vehiculoCuatro As New Vehiculo With {.Matricula = "3333AAA", .Tipo = Vehiculo.TipoVehiculo.camion, .Velocidad = 160}
        listaVehiculos.Add(vehiculoCuatro)
        Dim vehiculos = <?xml version='1.0'?>
            <vehiculos>
                <%= From datos In listaVehiculos %
                    Select <vehiculo>
                        <tipo><%= datos.Tipo.ToString() %></tipo>
                        <matricula><%= datos.Matricula %></matricula>
                        <velocidad><%= datos.Velocidad %></velocidad>
                    </vehiculo> %
                </vehiculos>
        vehiculos.Save("C:\vehiculos.xml")
        Me.WebBrowser1.Navigate("C:\vehiculos.xml")
        MessageBox.Show("Datos guardados")
    End Sub

End Class

```

Observando con detenimiento este ejemplo, podemos ver que por un lado hemos creado una estructura de datos, que utilizaremos más adelante para crear los objetos que utilizaremos para crear al aire nuestro documento XML.

Dentro del código del evento **Click** del control **Button**, crearemos objetos de tipo **Vehiculo**, que corresponde con la estructura anteriormente creada, y construiremos un documento XML con los datos de los objetos creados. Finalmente, guardaremos el documento XML creado en memoria al disco duro, y mostraremos su contenido en el control **WebBrowser**.

El resultado final obtenido, es el que se puede ver en la siguiente imagen:

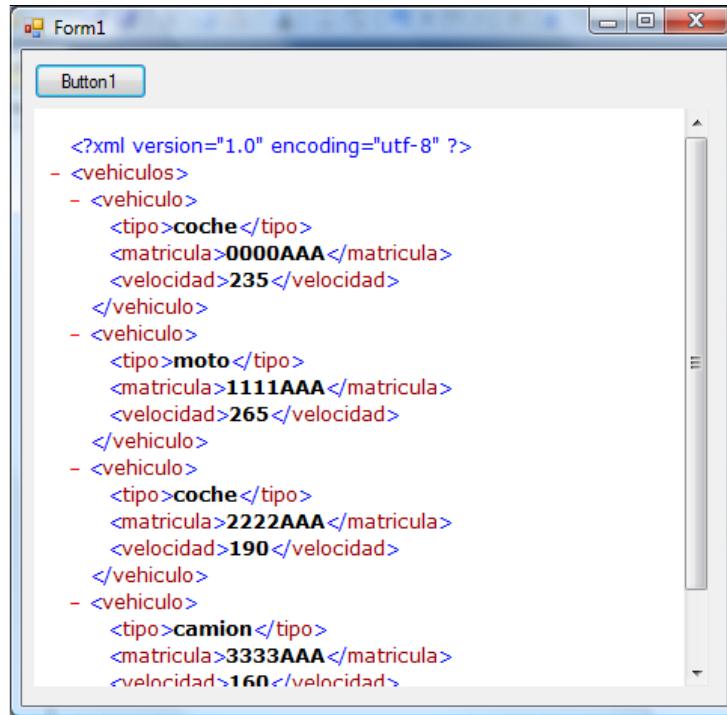


Figura 7.2.- Ejemplo en ejecución del uso de literales en LINQ to XML

Como podemos apreciar, el trabajo con documentos XML en Visual Basic 2010 es realmente sencillo y productivo gracias a LINQ.

Módulo 7: LINQ

- Para qué LINQ
 - LINQ to Objects
 - LINQ to XML
- LINQ y ADO.NET**
- LINQ to DataSet
 - LINQ to SQL
 - LINQ to Entities

LINQ y ADO.NET

Hasta el momento, hemos visto como trabajar con LINQ to Objects y LINQ to XML, sin embargo, aún no hemos visto como trabajar con fuentes de datos y LINQ.

Como recordará, al principio del capítulo dedicado a LINQ, vimos un resumen de los proveedores existentes de LINQ.

Vimos en una imagen, ese resumen. La imagen de recordatorio, es la siguiente:

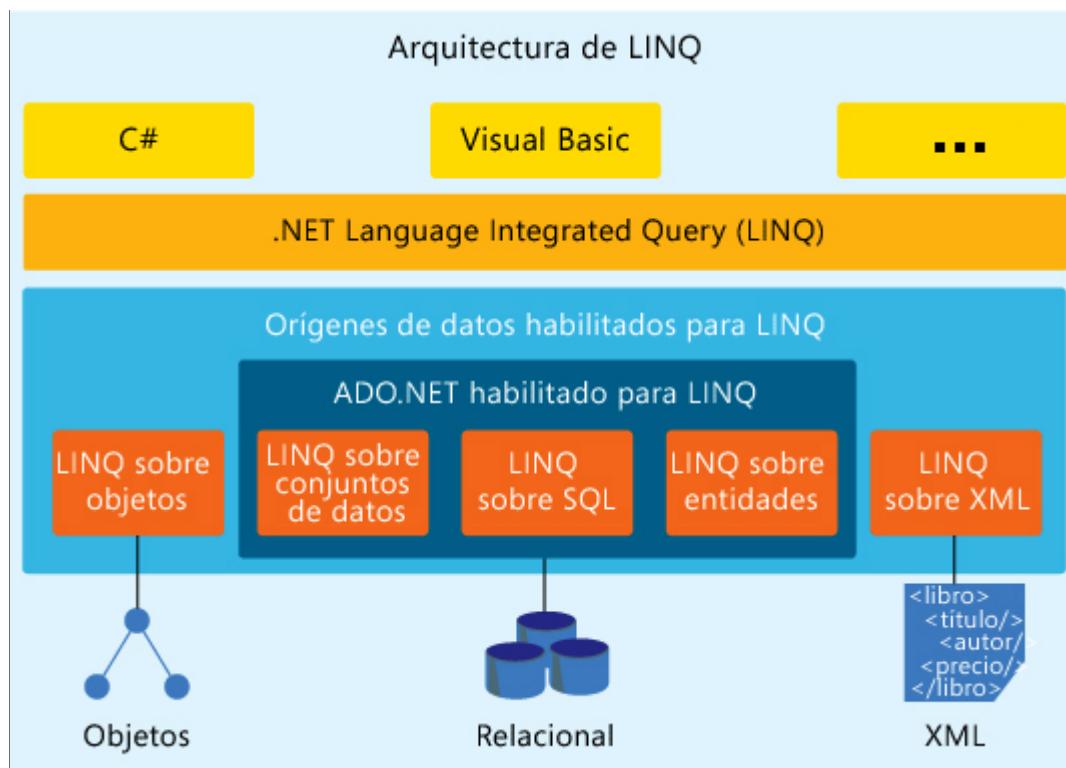


Figura 7.3.- Arquitectura de LINQ

Como puede observar en la imagen, la parte que corresponde al trabajo con LINQ y ADO.NET, está constituida por tres proveedores que nos permiten trabajar con datos. Esto son LINQ to DataSet, LINQ to SQL y LINQ to Entities.

A continuación, veremos cada uno de ellos y como se utiliza en Visual Basic 2010.

Módulo 7: LINQ

- Para qué LINQ
- LINQ to Objects
- LINQ to XML
- LINQ y ADO.NET
- **LINQ to DataSet**
- LINQ to SQL
- LINQ to Entities

LINQ to DataSet

Como todos sabemos a estas alturas, los DataSet constuyen la base para trabajar con datos en memoria, desconectados de la fuente de datos.

LINQ to DataSet nos ofrece la posibilidad de trabajar con LINQ y fuentes de datos desconectadas, o lo que es lo mismo, usar LINQ para manipular y trabajar con los datos de un DataSet.

Para trabajar con LINQ y DataSet, no debemos pensar en nuevo paradigma de trabajo y manipulación de los datos, sino de pensar en LINQ como un servicio más que se ofrece para trabajar con DataSet.

Es decir, podremos trabajar con los DataSet tal y como lo hacemos hasta ahora, y aplicar LINQ para tratar de sacar todo el provecho posible.

La única regla de oro no obstante, es que los DataSet con los que trabajaremos, serán DataSet tipados, es decir, DataSet que poseen unos tipos de datos establecidos.

Practicando con LINQ to DataSet

En el siguiente ejemplo, veremos como utilizar LINQ en DataSets.

Para ello, crearemos una estructura a la que iremos dando una serie de valores y crearemos así, una lista de estructuras para crear posteriormente un DataSet y su correspondiente DataTable, e ir agregando los elementos de la estructura al DataTable de ese DataSet. Posteriormente, utilizaremos LINQ para acceder a los datos del DataSet.

El ejemplo, quedará de la siguiente forma:

```
Código

Public Structure Persona
    Public Enum SexoPersona
        hombre
        mujer
    End Enum

    Public Nombre As String
    Public Sexo As SexoPersona
    Public Edad As Byte
End Structure

Public Class Form1

    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button1.Click
        ' Creamos una lista de tipo Persona
        Dim listaPersonas As New List(Of Persona)
        Dim PersonaUno As New Persona With {.Nombre = "Daniel", .Edad = 27, .Sexo = Persona.SexoPersona.hombre}
        listaPersonas.Add(PersonaUno)
        Dim PersonaDos As New Persona With {.Nombre = "Jimena", .Edad = 29, .Sexo = Persona.SexoPersona.mujer}
        listaPersonas.Add(PersonaDos)
        Dim PersonaTres As New Persona With {.Nombre = "Alberto", .Edad = 24, .Sexo = Persona.SexoPersona.hombre}
        listaPersonas.Add(PersonaTres)
        ' Creamos el DataSet
        Dim personasDataSet As New DataSet()
        ' Agregamos una tabla al DataSet
        personasDataSet.Tables.Add("TablaPersonas")
        ' Agregamos tres columnas a la tabla
        personasDataSet.Tables("TablaPersonas").Columns.Add("Nombre")
        personasDataSet.Tables("TablaPersonas").Columns.Add("Edad")
        personasDataSet.Tables("TablaPersonas").Columns.Add("Sexo")
        ' Recorremos los elementos de la lista Persona
        ' y vamos agregando esos elementos al DataSet
        For Each empleado In listaPersonas
            ' Creamos una nueva fila
            Dim fila As DataRow = personasDataSet.Tables("TablaPersonas").NewRow()
            fila("Nombre") = empleado.Nombre
            fila("Edad") = empleado.Edad
            fila("Sexo") = empleado.Sexo
            ' Agregamos la fila
            personasDataSet.Tables("TablaPersonas").Rows.Add(fila)
        Next
        ' Buscamos datos dentro del DataSet
        Dim datosBuscados = From datos In personasDataSet.Tables("TablaPersonas") _
                            Where datos.Item("Edad") > 25 _
                            Select nombre = datos.Item("Nombre"), flagSexo = If(datos.Item("Sexo") = "hombre", True, False), sexo = datos.Item("Sexo")
        ' Mostramos los datos
        Dim resultado As String = ""
        For Each dato In datosBuscados
            resultado &= String.Format("{0} es {1} {2}", dato.nombre, If(dato.flagSexo, "un", "una"), dato.sexo) & vbCrLf
        Next
        MessageBox.Show(resultado)
    End Sub
End Class
```

Novedad en Visual Basic 2010: una de las novedades de Visual Basic 2010, es que ahora, **If** puede funcionar de la misma forma a como lo hacía **IIf**. En nuestros desarrollos podremos seguir utilizando **IIf**, pero **If**, también funciona desde Visual Basic 2010 de la misma forma.

En el ejemplo anterior, podemos ver que buena parte del código la hemos dedicado a crear el objeto DataSet y su objeto DataTable, alimentar ese DataTable con elementos, y posteriormente, crear la instrucción LINQ correspondiente para extraer los elementos resultantes.

Como podemos apreciar, la forma de trabajar es siempre la misma.

Algo que aún no hemos mencionado cuando trabajamos con LINQ es todo lo relativo a la eficiencia, productividad e inmediatez a la hora de trabajar con LINQ y los datos.

El uso de LINQ es realmente rápido, mucho más de lo que mucha gente puede creer en un primer momento.

A continuación, veremos como trabajar con LINQ y otros modelos o fuentes de datos.

Módulo 7: LINQ

- Para qué LINQ
- LINQ to Objects
- LINQ to XML
- LINQ y ADO.NET
- LINQ to DataSet
- LINQ to SQL**
- LINQ to Entities

LINQ to SQL

Anteriormente, hemos visto como trabajar con LINQ to DataSet para trabajar con fuentes de datos desconectadas, pero, ¿Y si queremos trabajar con fuentes de datos conectadas?

En ese caso, podremos utilizar LINQ to SQL o LINQ to Entities.

¿Cuál de ellos debemos utilizar?

LINQ to Entities está preparado para trabajar con cualquier fuente de datos conectada, mientras que LINQ to SQL está preparada para trabajar únicamente con fuentes de datos Microsoft SQL Server, por lo que parece claro, que si tenemos un motor de base de datos como Microsoft SQL Server, podemos trabajar tanto con LINQ to SQL como con LINQ to Entities, aunque si el motor de base de datos con el que estamos trabajando es Oracle, LINQ to SQL no nos servirá.

Aún y así, LINQ to Entities es ligeramente diferente a LINQ to SQL y ofrece unas características más avanzadas.

¿Sentencias LINQ o sentencias SQL?

LINQ to SQL destaca por permitirnos trabajar con objetos relacionales (O/R) a través del diseñador que ha sido incluido en Visual Studio 2010.

Adicionalmente, LINQ to SQL tiene la habilidad de construir la instrucción SQL correspondiente.

El programador debe olvidarse de como construir esa sentencia SQL, ya que es el propio compilador y el motor de ejecución de .NET Framework, el que se encarga de crear y lanzar por nosotros la sentencia SQL correspondiente.

Como comentábamos antes, LINQ to SQL trabaja con Microsoft SQL Server, por lo que las sentencias SQL creadas por LINQ to SQL serán para ese motor de base de datos.

De forma diferente a la que mucha gente piensa, esas sentencias generadas por LINQ son sentencias optimizadas en rendimiento, así que debemos estar tranquilos a la hora de utilizar LINQ.

Practicando con LINQ to SQL

Para practicar con LINQ to SQL deberemos iniciar Visual Studio 2010. En concreto, iniciaremos un proyecto de formulario Windows e insertaremos un control **Button** y un control **DataGridView** dentro del formulario.

Una vez hecho esto, haremos clic sobre el proyecto con el botón derecho del ratón, y aparecerá una ventana dentro de la cual, deberemos seleccionar la plantilla **Clases de LINQ to SQL** perteneciente al grupo **Datos**. Esto es lo que se puede ver en la siguiente imagen:

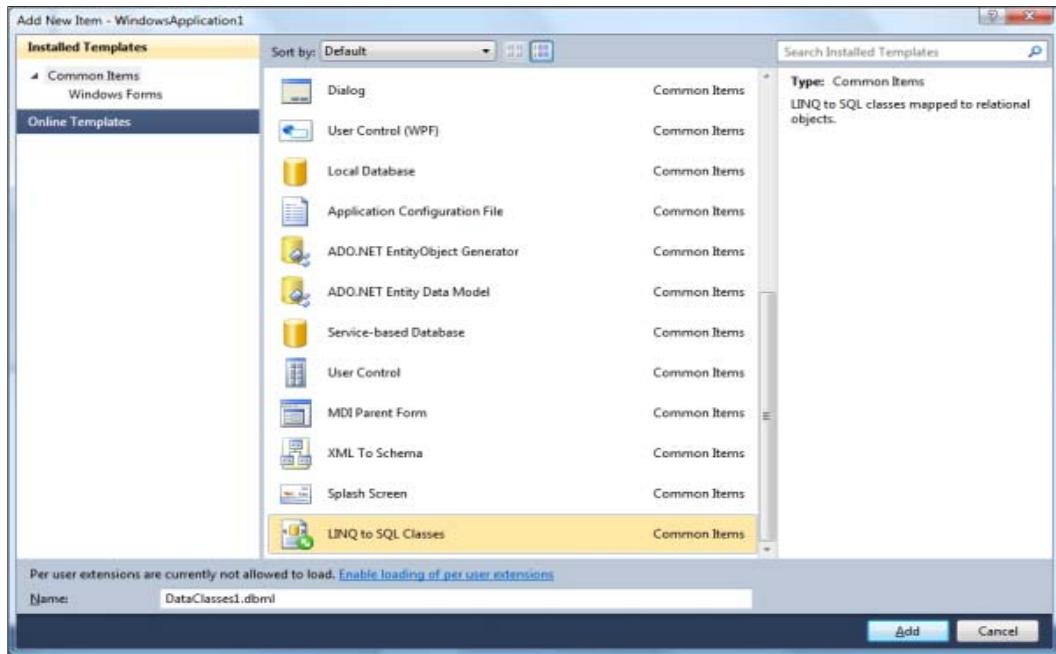


Figura 7.4.- Selección de la plantilla Clases de LINQ to SQL

Una vez seleccionada esta plantilla, el proyecto mostrará una nueva ventana que corresponde con el diseñador del modelo relacional, donde deberemos indicar que partes del modelo de datos de SQL Server queremos incluir en la plantilla.

A continuación, deberemos realizar la siguiente tarea, que consiste en crear una conexión con la base de datos de SQL Server. Para ello, podemos utilizar el **Explorador de servidores** y agregar en él la conexión que queremos utilizar en LINQ to SQL.

Cuando hayamos agregado la conexión con la base de datos, y dentro de la ventana del **Explorador de servidores**, podremos arrastrar y soltar sobre el diseñador, los objetos del modelo de datos que queramos utilizar en LINQ to SQL.

En el ejemplo con el que vamos a trabajar, los objetos que hemos arrastrado y soltado sobre el diseñador, son los que se muestran en la siguiente figura:

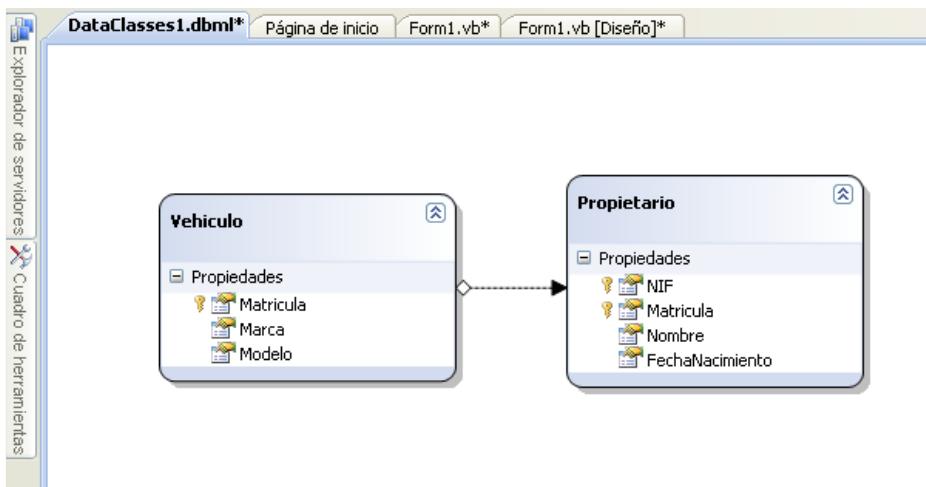


Figura 7.5.- Objetos de la base de datos en el diseñador de LINQ to SQL

A observar: la extensión del diseñador del modelo relacional de LINQ to SQL es **dbml**, que corresponde con la plantilla de LINQ to SQL que hemos seleccionado en la figura 7.4.

Como podemos ver en la figura anterior, el diseñador del modelo relacional de LINQ to SQL, muestra en este caso, dos tablas y su posible relación. Las tablas contienen además, datos con los que podremos operar o trabajar. Aquí es donde entra LINQ y donde usándolo, obtendremos una gran productividad en nuestros desarrollos.

Para finalizar nuestro ejemplo, escribiremos el siguiente código fuente:

Código

```
Public Class Form1
```

```

Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button1.Click
    ' Declaramos el contexto
    Dim contexto As New DataClasses1DataContext()
    ' Ejecutamos la sentencia LINQ correspondiente
    Dim datos = From tablaPropietario In contexto.Propietarios,
                tablaVehiculos In contexto.Vehiculos
                Where tablaPropietario.Matricula = tablaVehiculos.Matricula
                Group tablaVehiculos By tablaPropietario.NIF Into Group
                Select NIF, Cantidad = NIF.Count()
    ' Mostramos el resultado final
    Me.DataGridView1.DataSource = datos
End Sub

End Class

```

Nuestro ejemplo en ejecución, es el que se muestra en la figura siguiente:

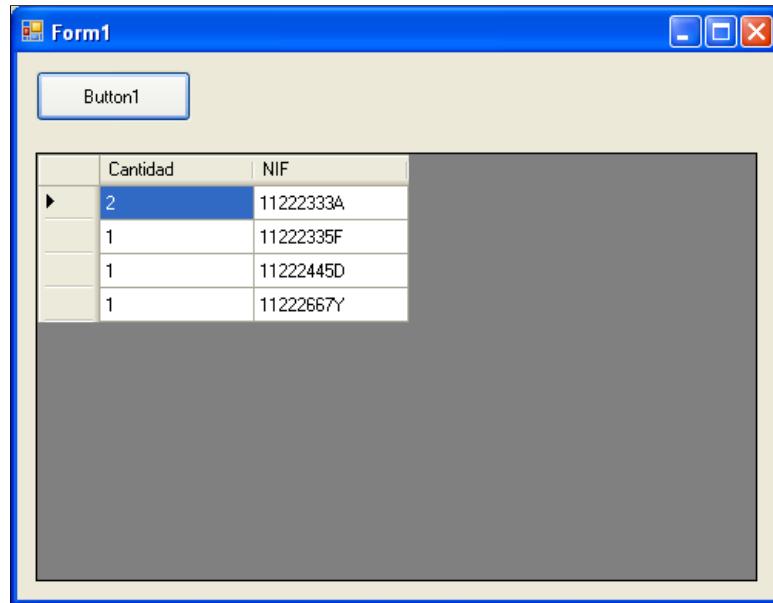


Figura 7.6.- Ejemplo del uso de LINQ to SQL en ejecución

Como podemos apreciar en el código fuente de la aplicación que hemos escrito, la cantidad de código fuente que hemos agregado es realmente pequeña. Analizando nuestro código fuente, vemos que por un lado, hemos declarado el contexto, o lo que es lo mismo, la clase del modelo relacional que hemos construido (el fichero de extensión **dbml**).

Si abrimos este fichero, observaremos que tiene un código fuente asociado que corresponde con el modelo relacional pasado a una clase, conservando en él cada uno de los campos de la tabla o de las tablas, con sus tipos de datos (fuertemente tipada) y sus relaciones si las hay.

De esta manera, estamos trabajando sobre una clase y sus propiedades, que corresponden con los campos de las tablas, por lo que resulta muy difícil equivocarse.

Otra particularidad del modelo relacional creado en Visual Studio 2010, es que el modelo relacional se conecta con SQL Server cuando es utilizado, y esto es prácticamente transparente para el programador, ya que todo este proceso de conectarse, ejecutar la instrucción correspondiente, obtener los datos y desconectarse nuevamente de la fuente de datos, la realiza el sistema dentro de la clase del modelo relacional (fichero **dbml**), pero la pregunta que podríamos hacernos ahora es... ¿dónde se encuentra por lo tanto la necesaria cadena de conexión para conectarse a la fuente de datos?.

La cadena de conexión la encontraremos en la configuración de la aplicación.

Para acceder a ella, deberemos hacer clic con el botón derecho del ratón sobre el proyecto, y seleccionar la opción

Propiedades del menú emergente.

Finalmente, accederemos a la solapa **Configuración** para acceder al valor de la cadena de conexión que utiliza LINQ to SQL para conectarse.

Esto es lo que se puede ver en la siguiente imagen:

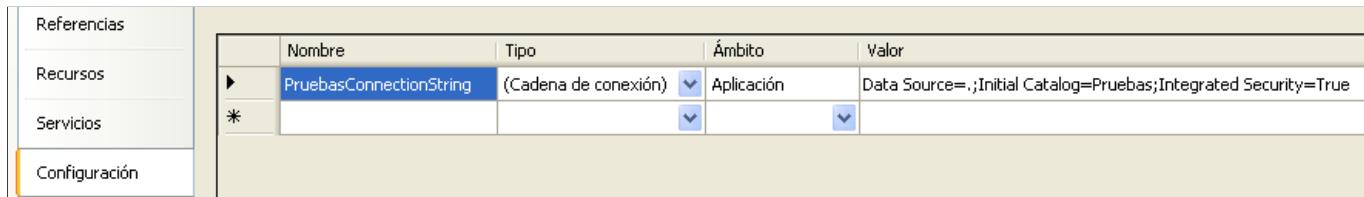


Figura 7.7.- Solapa de Configuración del proyecto, con la cadena de conexión del ejemplo de LINQ to SQL

Módulo 7: LINQ

- [Para qué LINQ](#)
- [LINQ to Objects](#)
- [LINQ to XML](#)
- [LINQ y ADO.NET](#)
- [LINQ to DataSet](#)
- [LINQ to SQL](#)
- [**LINQ to Entities**](#)

LINQ to Entities

A estas alturas, tendrá claro algunas de las ventajas de trabajar con LINQ, sin embargo, LINQ dentro de Microsoft .NET Framework 3.5 no proporciona un acceso directo a cualquier tipo de objeto, solamente a aquellos que estén preparados para trabajar con LINQ.

Dentro de los proveedores de LINQ, Microsoft proporciona uno muy especial, muy parecido a LINQ to SQL, pero con notables diferencias.

Este proveedor se llama LINQ to Entities, y es un proveedor preparado para trabajar con cualquier gestor de base de datos, proporcionando algunas ventajas que no proporciona LINQ to SQL.

Aún y así, también LINQ to Entities posee algunas diferencias con respecto a LINQ to SQL que le perjudican, como por ejemplo que LINQ to SQL sí está preparado para trabajar con procedimientos almacenados, mientras que la primera versión de LINQ to Entities no lo está. De hecho, ésta es una de las características más demandadas por la comunidad de desarrolladores y que el equipo de Microsoft de LINQ to Entities, tiene encima de la mesa para ponerse con ella cuanto antes.

Recordemos además, que mientras LINQ to SQL solo está preparado para trabajar con Microsoft SQL Server, LINQ to Entities está preparado para trabajar con cualquier motor de base de datos. Solo será necesario tener instalados los componentes adecuados para llevar a buen término nuestras necesidades. De hecho, en Internet puede encontrar ya algunos de estos proveedores listos para ser usados en motores de bases de datos como Oracle, DB2 o MySQL.

Con LINQ to Entities, podemos trabajar por lo tanto, con motores de bases de datos y con EDM (Entity Data Model), o lo que es lo mismo, una representación lógica de un modelo de datos.

LINQ to SQL nos ofrecía un mapeo de datos muy sencillo, y para muchos desarrollos Software, será el proveedor adecuado, pero en otras circunstancias, necesitamos algo más. Ese "algo más" nos lo proporciona LINQ to Entities, que se encarga de mapear los datos con mecanismos más complejos y potentes a lo que lo hace LINQ to SQL.

Finalmente, y para que no se lleve a engaños, debemos indicar que tanto LINQ to SQL como LINQ to Entities, no están preparados para soportar relaciones de muchos a muchos con carga útil (*Payload relationship*). El objetivo del equipo de trabajo de LINQ to Entities, es tener esta característica lista para la próxima versión de LINQ to Entities.

Pero para utilizar LINQ to Entities, necesitaremos no obstante, instalar un par de agregados a nuestra instalación de Visual Studio 2010.

Preparando Visual Studio 2010 para trabajar con LINQ to Entities

Para trabajar con LINQ to Entities, deberemos preparar adecuadamente nuestro entorno de trabajo, Visual Studio 2010. Deberemos descargar e instalar Microsoft .NET Framework 3.5 SP1 (Service Pack 1) en primer lugar, reiniciar el sistema (recomendado), e instalar Visual Studio 2010 SP1 posteriormente.

Es posible que entre la instalación de Microsoft .NET Framework 3.5 SP1 y de Visual Studio 2010 SP1, el sistema le pida instalar algún paquete de Software necesario para llevar a buen término la instalación. Esos posibles avisos le aparecerán en la ventana de instalación, y lo único que tendrá que hacer es seguir las instrucciones.

Tenga a mano también, los CD o DVD originales de la instalación, porque es muy posible que el proceso de actualización le pida esa información.

Practicando con LINQ to Entities

En este punto, deberíamos estar listos para trabajar con LINQ to Entities, y es eso justamente, lo que vamos a hacer ahora.

Iniciaremos un nuevo proyecto de tipo aplicación Windows con Visual Studio 2010, e insertaremos en el formulario un control **Button** y un control **DataGridView**.

La siguiente acción que realizaremos, será la de agregar un elemento de tipo LINQ to Entities, que nos permitirá trabajar con la fuente de datos que elijamos.

Para agregar un nuevo elemento al proyecto, haremos clic dentro de la ventana del *Explorador de soluciones* sobre el proyecto con el botón derecho del ratón, y seleccionaremos la opción **Agregar > Nuevo elemento** del menú emergente.

Aparecerá una ventana dentro de la cual, deberemos encontrar la plantilla **ADO.NET Entity Data Model**, la cual localizaremos rápidamente en el grupo de plantillas de tipo **Datos**.
Esta es la ventana que se muestra a continuación:

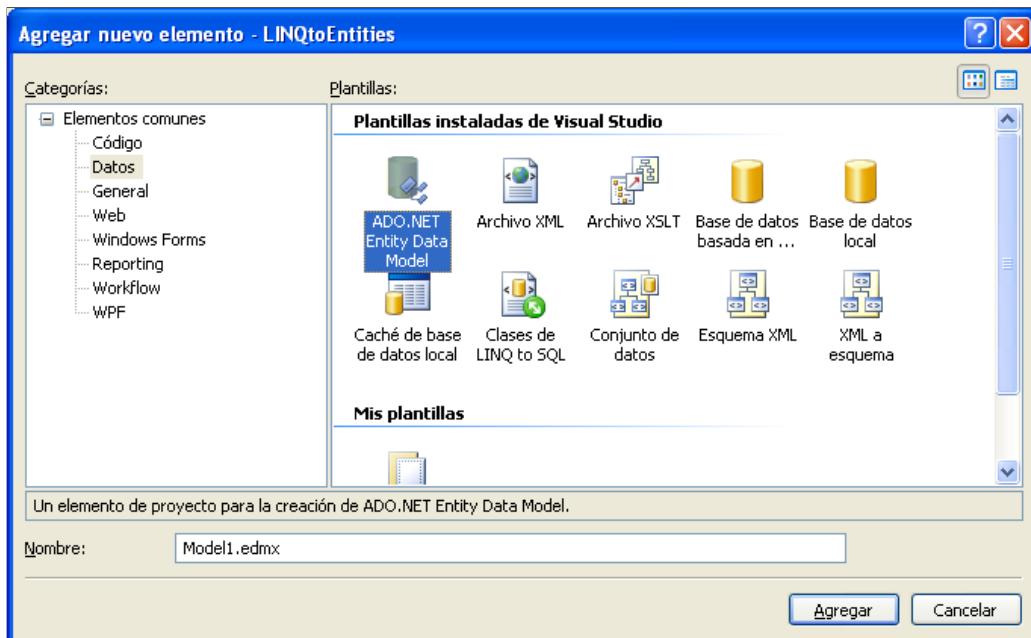


Figura 7.8.- Selección de la plantilla ADO.NET Entity Data Model

Haremos doble clic sobre la plantilla y ésta se nos incorporará al proyecto.

Al incorporarse esta plantilla al proyecto, Visual Studio 2010 lanzará un asistente, cuya primera ventana, consiste en seleccionar el contenido del modelo. Dentro de la selección, podemos elegir entre generar el modelo desde la base de datos, o generar el modelo partiendo de un modelo vacío.

Nuestra intención es seleccionar todas las tablas en el modelo, por lo que seleccionaremos la primera opción.
Esto es lo que se puede ver en la siguiente imagen:

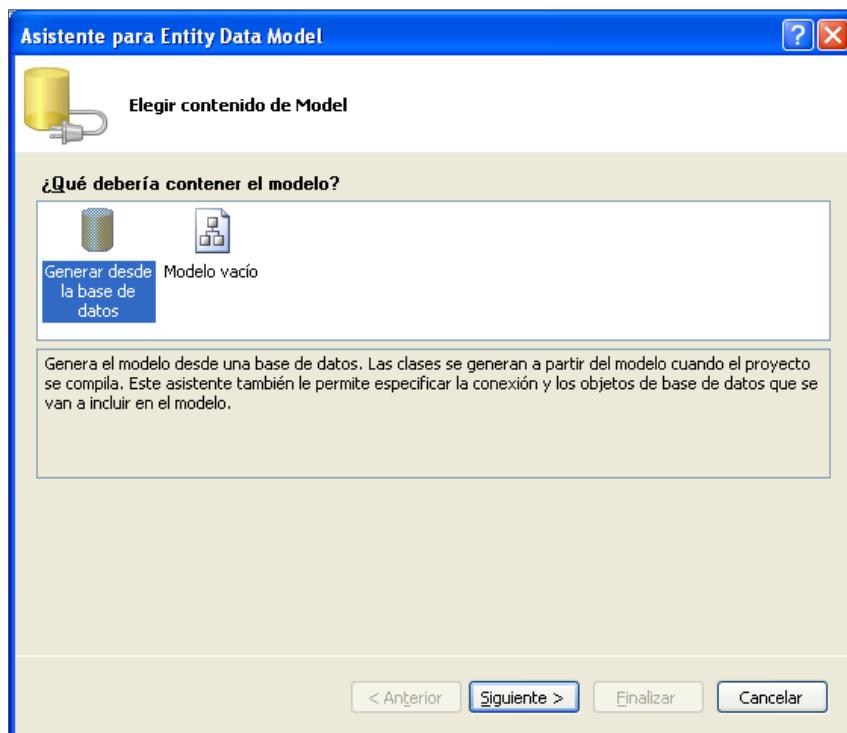


Figura 7.9.- Ventana principal del asistente para el EDM - Entity Data Model

A continuación, haremos clic sobre el botón **Siguiente**.

De esta forma, aparecerá una ventana dentro de la cual, podremos elegir la conexión de datos que queremos utilizar.

Nos aseguraremos de que tenemos la conexión de datos adecuada, y en caso contrario, crearemos una.

En esta misma ventana, podremos crear un nombre para el nombre de la conexión que será utilizado en el archivo de configuración de la aplicación **App.config**. Podríamos de todas las maneras, no seleccionar esta opción que aparece seleccionada por defecto. En nuestro, hemos dejado las opciones tal y como aparecen en la imagen.

Para este ejemplo, utilizaré una conexión de datos de Microsoft SQL Server. En mi caso, la ventana con la configuración creada es la que se indica en la siguiente imagen:

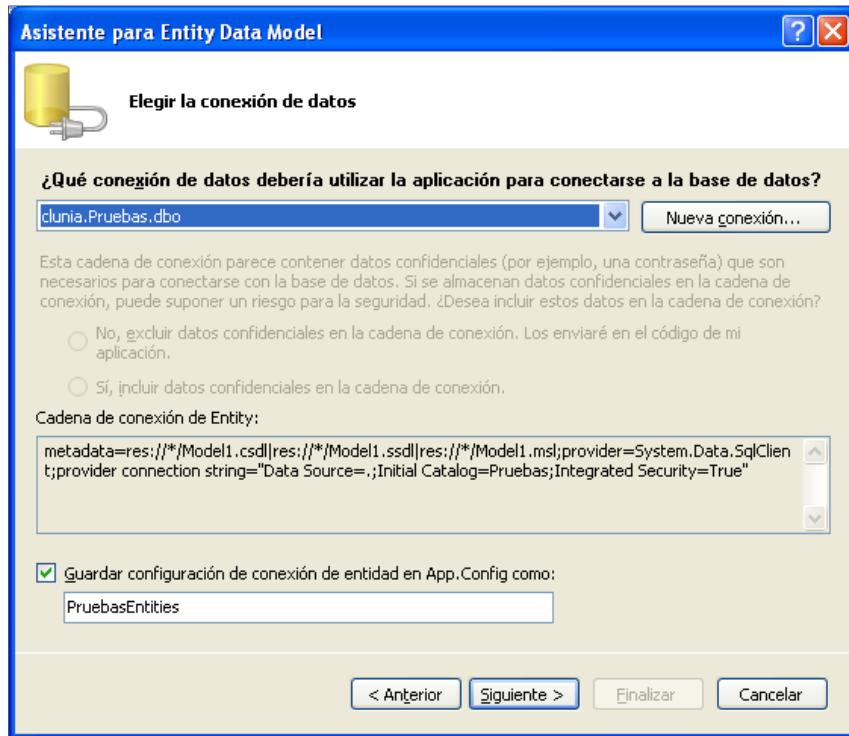


Figura 7.10.- Ventana de selección de la conexión de datos

Una vez realizado estos pasos, haremos clic en el asistente sobre el botón **Siguiente**.

Aparecerá entonces una ventana en el asistente, dentro de la cual podremos seleccionar los objetos de la base de datos que queremos utilizar.

Como podremos ver en esa ventana, podremos indicar no solo las tablas que queremos agregar al modelo, sino también las vistas y los procedimientos almacenados.

Para este ejemplo, solo seleccionaremos las tablas de nuestro modelo, que es exactamente el mismo que utilizamos en el capítulo de LINQ to SQL.

Esta selección es la que se muestra en la siguiente imagen:

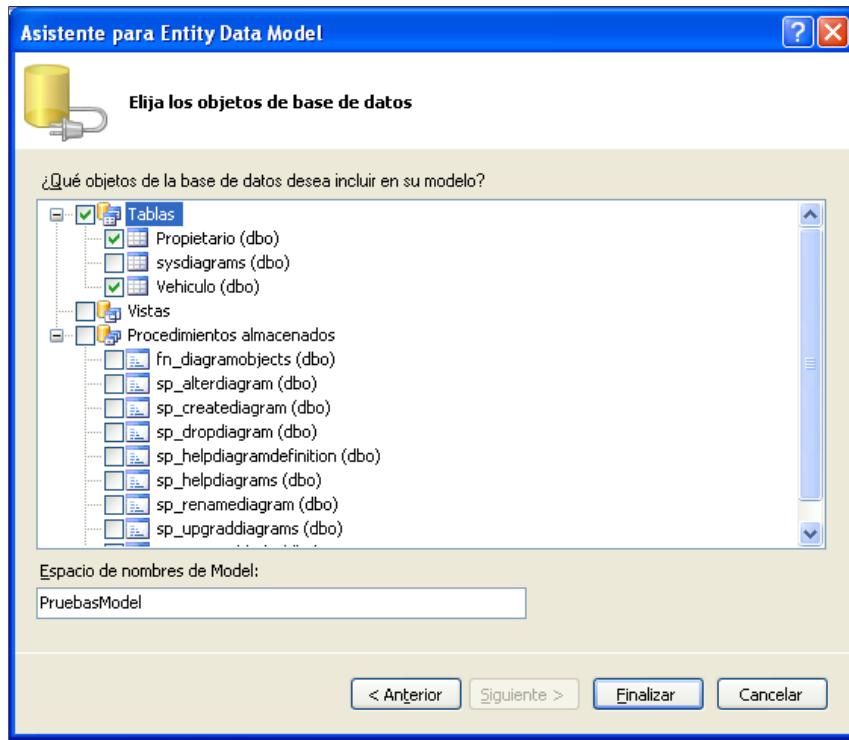


Figura 7.11.- Objetos de la base de datos seleccionados en LINQ to Entities

Para concluir con el asistente, haremos clic sobre el botón **Finalizar**.

De esta manera, se generará la configuración del modelo tal y como lo hemos indicado en el asistente, y se generarán las siguientes partes:

La primera cosa que nos llamará la atención es el modelo de datos que se habrá dibujado en Visual Studio 2010, y que en nuestro ejemplo corresponderá con el siguiente:

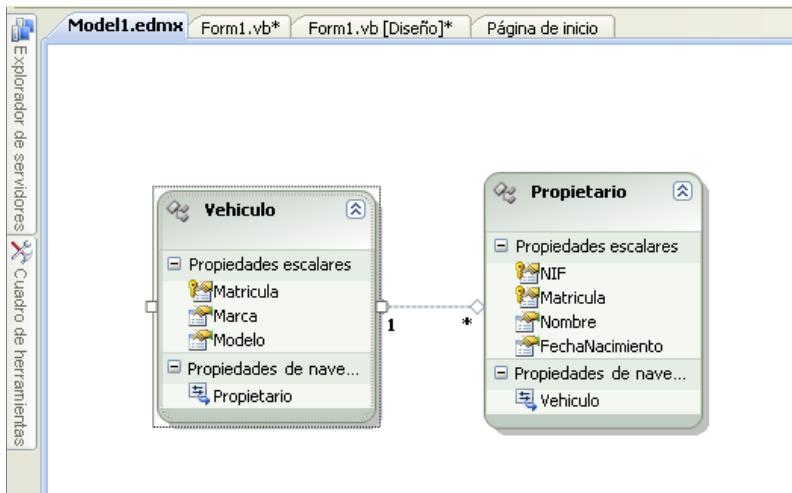


Figura 7.12.- EDM representado en Visual Studio 2010 después de ejecutar el asistente de LINQ to Entities

En nuestro diagrama, podemos localizar la entidad *Vehiculo* y la entidad *Propietario*.

Como podemos observar en el diagrama del EDM, el sistema ha determinado por nosotros una relación de tipo 1 a muchos entre la entidad *Vehiculo* y la entidad *Propietario*, por lo que podríamos decir rápidamente viendo el diagrama, que una persona puede ser propietario de más de un vehículo.

En la parte inferior de Visual Studio 2010, podremos encontrar también, una ventana que contiene los detalles de asignación de cada una de las entidades.

Un ejemplo de los detalles de asignación de la tabla *Vehiculo* es la que se indica en la siguiente imagen:

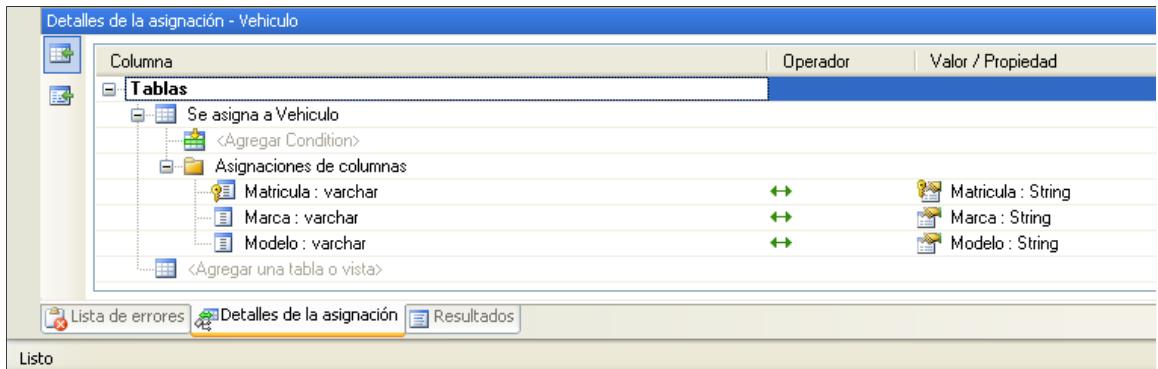


Figura 7.13.- Detalles de asignación de la entidad Vehículo de nuestro EDM en LINQ to Entities

Si observamos detenidamente la ventana con los detalles de asignación, veremos que los campos de la tabla se han convertido en propiedades, y que los campos clave de la tabla, están así indicados en el EDM.

En sí, el EDM representa fielmente el modelo de datos relacional, permitiéndonos trabajar con él de una forma rápida y directa, evitando errores de asignación y asegurándonos una fiabilidad adecuada.

Pero hasta aquí, la preparación del EDM y algunos comentarios que hemos hecho para que se familiarice con él.

¿Cuál es nuestra siguiente tarea?. Desarrollar un ejemplo práctico que nos muestre como utilizar LINQ to Entities en nuestros desarrollos.

Eso es justamente lo que haremos a continuación.

Ya tenemos preparado nuestro formulario Windows con los controles que queremos utilizar. Eso ya lo indicamos al principio de este capítulo.

A continuación, deberemos escribir el código de nuestra aplicación, que quedará de la siguiente manera:

Código

```

Public Class Form1

    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button1.Click
        ' Declaramos el contexto
        Dim contexto As New PruebasEntities()
        ' La conexión abre y cierra el contexto por nosotros
        ' Indicamos la instrucción LINQ que vamos a ejecutar
        Dim datos = From tablaPropietario In contexto.Propietario,
                    tablaVehiculo In contexto.Vehiculo
                    Where tablaPropietario.Matricula = tablaVehiculo.Matricula
                    Order By tablaPropietario.Nombre
                    Select tablaPropietario.Nombre, tablaPropietario.NIF, tablaVehiculo.Marca, tablaVehiculo.Modelo
        ' Mostramos los datos en el control DataGridView
        Me.DataGridView1.DataSource = datos
        ' Ajustamos el tamaño del control DataGridView
        ' al tamaño de los datos de las filas y columnas
        Me.DataGridView1.AutoResizeColumns(DataGridViewAutoSizeColumnsMode.AllCells)
        Me.DataGridView1.AutoResizeRows(DataGridViewAutoSizeRowsMode.AllCells)
    End Sub

End Class

```

Nuestro ejemplo en ejecución, es el que se muestra en la figura siguiente:

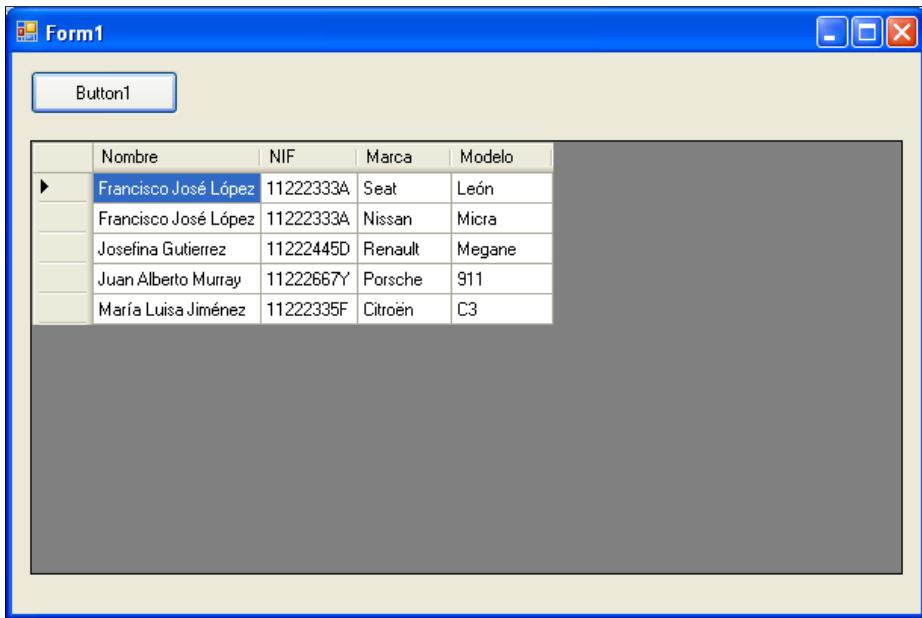


Figura 7.14.- Ejemplo del uso de LINQ to Entities en ejecución

Como podemos apreciar en el código fuente de la aplicación que hemos escrito, hemos declarado el contexto para utilizarlo en el proceso de obtención de los datos.

Posteriormente hemos declarado la instrucción LINQ a ejecutar. Como podemos ver, no hemos abierto ni cerrado ninguna conexión con la base de datos. Esto lo realiza LINQ to Entities por nosotros de forma transparente.

Finalmente, hemos mostrado los datos obtenidos en la ejecución de la instrucción LINQ, dentro del control **DataGridView**.

Ahora bien, ¿dónde se encuentra la cadena de conexión?.

Si accedemos al fichero **App.Config**, veremos un apartado que en nuestro caso es de la siguiente forma:

Código

```
<connectionStrings>
  <add name="PruebasEntities" connectionString="metadata=res://*/Model1.csdl|res://*/Model1.ssdl|res://*/Model1.msl;
provider=System.Data.SqlClient;provider connection string="Data Source=.;Initial Catalog=Pruebas;
Integrated Security=True;MultipleActiveResultSets=True"" providerName="System.Data.EntityClient" />
</connectionStrings>
```

Si nos fijamos en lo que aquí se indica, podemos observar que LINQ to Entities está compuesto por el modelo de entidades (EDM), que consiste en un fichero de extensión (**edmx**), y tres ficheros que por defecto estarán incrustados en el ensamblado como archivo de recursos, y que contienen la extensión (**csdl**), (**ssdl**) y (**msl**).

En esta parte del documento XML, podemos ver también el proveedor de acceso a datos que vamos a utilizar, la cadena de conexión con la fuente de datos, y el nombre del proveedor de LINQ (**System.Data.EntityClient**).

Esta parte de la aplicación es la que contiene información relativa a la propia conectividad con la fuente de datos y a los parámetros de configuración necesarios para acceder y manipular eficientemente los datos.

Módulo 8: DLR – Dynamic Programming

- VB e IronPython
- VB e IronRuby

Contenido

En este módulo veremos como trabajar con IronPython desde Visual Basic para ejecutar dinámicamente diferentes instrucciones de código.

- [VB e IronPython](#)
 - [VB e IronPython](#)
-

Módulo 8: DLR – Dynamic Programming

VB e IronPython

- VB e IronRuby

VB e IronPython

IronPython es una implementación del lenguaje de programación Python para la plataforma .NET y Silverlight.

Nota: IronPython debe ser descargado independientemente. En CodePlex hay un proyecto dedicado en exclusiva a IronPython. Este proyecto puede ser localizado en esta [dirección web](#).

El objetivo de IronPython es el de proporcionar un mecanismo para que podamos ejecutar código Python desde nuestras aplicaciones .NET. De hecho, IronPython es un intérprete de Python, es decir, que podemos coger cualquier código de Python y ejecutarlo con IronPython sin problemas.

Las ventajas por lo tanto de usar IronPython es que podemos además integrarlo con nuestras aplicaciones .NET, de hecho, la idea principal es la de ejecutar al vuelo o dinámicamente código Python.

A continuación veremos como llevar a cabo esta tarea. No hace falta comentar que lo primero que debemos hacer es descargar e instalar IronPython.

Ejemplo práctico del uso de IronPython

A continuación iniciaremos Visual Studio 2010 y con él una aplicación de Windows con Visual Basic 2010.

Agregaremos una referencias al proyecto a los ensamblados:

- IronPython
- IronPython.Modules
- Microsoft.Scripting
- Microsoft.Scripting.Core

Dentro del proyecto, agregaremos referencias a los siguientes nombres de espacio:

- Imports IronPython.Hosting
- Imports Microsoft.Scripting.Hosting

Una vez hecho esto, agregaremos la carpeta Lib de IronPython y que encontraremos en la carpeta de instalación de IronPython a la carpeta Debug de nuestro proyecto.

Finalmente, haremos doble clic sobre el formulario y escribiremos el siguiente código:

Código

```
Dim engine As ScriptEngine = Python.CreateEngine()  
Dim code As String = "(2 * 2) + (8 / 3)"
```

```
Dim source As ScriptSource = engine.CreateScriptSourceFromString(code, Microsoft.Scripting.SourceCodeKind.Expression)
Dim result As Integer = source.Execute(Of Int32)()
MessageBox.Show(result.ToString())
```

En esta ejemplo, hemos visto como ejecutar una aplicación que lanza en tiempo de ejecución una instrucción Python.

Ahora bien, quizás tengamos la idea de lanzar una instrucción Python con parámetro incluido, en cuyo caso deberemos indicar previamente el parámetro y su valor.

En el siguiente ejemplo, veremos como llevar a cabo esta tarea. La forma de realizar el ejemplo es la misma que en el caso anterior, variando únicamente las líneas de código que queremos lanzar.

En primer lugar escribiremos una clase que utilizaremos para lanzarla contra Python:

Código

```
Public Class Employee

    Public Property Id As Integer
    Public Property Name As String
    Public Property Salary As Integer

    Public Sub Employee(ByVal id As Integer,
                        ByVal name As String,
                        ByVal salary As Integer)
        Me.Id = id
        Me.Name = name
        Me.Salary = salary
    End Sub

End Class
```

Una vez definida nuestra clase, ejecutaremos el siguiente código para consumir los valores de la clase y obtener el resultado correspondiente:

Código

```
Dim engine As ScriptEngine = Python.CreateEngine()
Dim runtime As ScriptRuntime = engine.Runtime
Dim scope As ScriptScope = runtime.CreateScope()
Dim code As String = "emp.Salary * 0.3"
Dim source As ScriptSource = engine.CreateScriptSourceFromString(code, Microsoft.Scripting.SourceCodeKind.Expression)
Dim empleado = New Employee With {.Id = 1, .Name = "Francisco", .Salary = 1000}
scope.SetVariable("emp", empleado)
Dim result = Convert.ToDouble(source.Execute(scope))
MessageBox.Show(result.ToString())
```

Como podemos apreciar, el uso de IronPython nos permite crear aplicaciones mucho más ricas que nos permitirá ejecutar código en tiempo de ejecución aportando muchas más posibilidades a nuestras aplicaciones.

Módulo 8: DLR – Dynamic Programming

• VB e IronPython

VB e IronRuby

VB e IronRuby

IronRuby es una implementación del lenguaje de programación Ruby para la plataforma .NET y Silverlight.

Nota: IronRuby debe ser descargado independientemente. En CodePlex hay un proyecto dedicado en exclusiva a IronRuby. Este proyecto puede ser localizado en esta ➤ [dirección web](#).

El objetivo de IronRuby es el de proporcionar un mecanismo para que podamos ejecutar código Ruby desde nuestras aplicaciones .NET. De hecho, IronRuby es un intérprete de Ruby, es decir, que podemos coger cualquier código de Ruby y ejecutarlo con IronRuby sin problemas.

Al igual que con IronPython, con IronRuby podemos integrarlo con nuestras aplicaciones .NET, de hecho, la idea principal es la de ejecutar al vuelo o dinámicamente código Ruby, esté o no en un fichero externo o directamente invocarlo desde nuestra aplicación.

A continuación veremos como llevar a cabo esta tarea. No hace falta comentar que lo primero que debemos hacer es descargar e instalar IronRuby.

Ejemplo práctico del uso de IronRuby

A continuación iniciaremos Visual Studio 2010 y con él una aplicación de Windows con Visual Basic 2010.

Agregaremos una referencias al proyecto a los ensamblados:

- IronRuby
- IronRuby.Libraries
- Microsoft.Scripting
- Microsoft.Scripting.Core

Dentro del proyecto, agregaremos referencias a los siguientes nombres de espacio:

- Imports IronRuby.Hosting
- Imports Microsoft.Scripting.Hosting

Finalmente, haremos doble clic sobre el formulario y escribiremos el siguiente código:

Código

```
Dim engine As ScriptEngine = IronRuby.Ruby.CreateEngine()
Dim code As String = "(2 * 2) + (8 / 3)"
Dim source As ScriptSource = engine.CreateScriptSourceFromString(code, Microsoft.Scripting.SourceCodeKind.Expression)
Dim result As Integer = source.Execute(Of Int32)()
MessageBox.Show(result.ToString())
```

En esta ejemplo, hemos visto como ejecutar una aplicación que lanza en tiempo de ejecución una instrucción Ruby.

Ahora bien, quizás tengamos la idea de lanzar una instrucción Ruby con parámetro incluido, en cuyo caso deberemos indicar previamente el parámetro y su valor.

En el siguiente ejemplo, veremos como llevar a cabo esta tarea. La forma de realizar el ejemplo es la misma que en el caso anterior, variando únicamente las líneas de código que queremos lanzar.

En primer lugar escribiremos una clase que utilizaremos para lanzarla contra Ruby:

Código

```
Public Class Employee  
  
    Private Property Id As Integer  
    Private Property Name As String  
    Private Property Salary As Integer  
  
    Public Sub Employee(ByVal id As Integer,  
                        ByVal name As String,  
                        ByVal salary As Integer)  
        Me.Id = id  
        Me.Name = name  
        Me.Salary = salary  
    End Sub  
  
End Class
```

Una vez definida nuestra clase, ejecutaremos el siguiente código para consumir los valores de la clase y obtener el resultado correspondiente:

Código

```
Dim engine As ScriptEngine = IronRuby.Ruby.CreateEngine()  
Dim runtime As ScriptRuntime = engine.Runtime  
Dim scope As ScriptScope = runtime.CreateScope()  
Dim code As String = "emp.Salary * 0.3"  
Dim source As ScriptSource = engine.CreateScriptSourceFromString(code, Microsoft.Scripting.SourceCodeKind.Expression)  
Dim empleado = New Employee With {.Id = 1, .Name = "Francisco", .Salary = 1000}  
scope.SetVariable("emp", empleado)  
Dim result = Convert.ToDouble(source.Execute(scope))  
MessageBox.Show(result.ToString())
```

Como podemos apreciar, el uso de IronRuby nos permite crear aplicaciones mucho más ricas que nos permitirá ejecutar código en tiempo de ejecución aportando muchas más posibilidades a nuestras aplicaciones.

Otra particularidad es la relación de instrucciones de ejecución que existe entre IronPython e IronRuby. Si comparamos dos ejemplos similares, veremos que son prácticamente idénticos, lo cual nos facilita mucho las cosas.

Evidentemente, Python posee una nomenclatura de código diferente a la de Ruby, algo que se recomienda conocer en el caso de utilizar ambos lenguajes en nuestras aplicaciones.



Aplicación de ejemplo MSDN Video

A continuación vamos a desarrollar una aplicación que nos permita explorar las principales características de Visual Studio. Puede descargar el código fuente en su máquina y explorarlo o modificarlo para conocer de forma práctica cómo usar Visual Basic para desarrollar aplicaciones reales. También puede explorar los videos donde construimos esta aplicación paso a paso.

La aplicación

MSDN Video es una aplicación de ejemplo empresarial desarrollada en España por la comunidad de desarrollo y Microsoft. Puede utilizarse para comprobar las mejores prácticas en la construcción aplicaciones distribuidas, escalables y con distintos tipos de clientes (Windows, Web y móviles).

La aplicación que desarrollaremos en este tutorial es una versión reducida de MSDN Video donde podrá ver Visual Studio en acción en un escenario cliente / servidor sencillo.



Código fuente de MSDN Video

Puede descargar el código fuente de MSDN Video y visionarlo o editarlo en su máquina.

Nota:

MSDN Video utiliza una base de datos para almacenar su estado. Si desea ejecutar la aplicación en su máquina necesitará SQL Server 2005 Express o superior instalado.

Videos explicativos

Si desea ver cómo se hizo esta aplicación puede visionar estos videos donde desarrollamos la aplicación paso a paso:

- [Video 1. Creación de la base de datos y el acceso a datos](#)
- [Video 2. Esqueleto de formularios](#)
- [Video 3. Formulario Nuevo Socio](#)
- [Video 4. Formulario Alquilar](#)
- [Video 5. Formulario Devolver](#)
- [Video 6. Conclusiones](#)

MSDN Video empresarial

Puede descargar la versión empresarial de MSDN Video y ver cómo podemos convertir la aplicación que hemos desarrollado aquí en un sistema distribuido completo, con acceso a través de servicios web desde clientes Windows, Web y dispositivos móviles.

Web de MSDN Video

Código fuente, tutoriales y videos explicativos de la versión completa de MSDN Video.

Desarrolla con MSDN

Puedes colaborar con MSDN Video, te proporcionamos materiales de formación, videos y eventos para que aprendas las tecnologías utilizadas en la aplicación.