

Jorge Serrano Pérez

Jorge Serrano

Manual de Introducción a

Microsoft®

Visual Basic® 2005

Express Edition

Microsoft®

Manual de Introducción a

Microsoft®

Visual Basic® 2005

Express Edition

por Jorge Serrano Pérez

Agradecimientos

A Javier Izquierdo de Microsoft España, que me tuvo en cuenta y me invitó a escribir esta pequeña densa obra.

A la gente de Microsoft MSDN España Alfonso Rodríguez, David Carmona y cia., por darme la oportunidad también de introducirme y bucear en Visual Basic 2005.

A Andy González de Microsoft Corp. por la amistad y ayuda que me brinda siempre que necesito algo relacionado con los productos de desarrollo de Microsoft.

A Alberto Amescua (Microsoft MVP España) y a los Microsoft MVP por demostrar esa fuerza, compañerismo, entrega y dedicación por mostrar y compartir con los demás los conocimientos y el propio crecimiento intelectual sin esperar nada a cambio en un mundo en el que el consumismo y el egoísmo es el caballo de batalla del día a día.

A otras muchas personas que ahora omito u olvido no intencionadamente y que de alguna manera, forman parte de este apartado de agradecimientos y que sin duda han formado parte directa o indirecta de esta obra.

Dedicatoria

Quiero dedicar esta pequeña obra a todas las personas que sufren algún tipo de discapacidad y que con su superación, esfuerzo y tesón, intentan minimizar esa diferencia para hacerla lo más inapreciable posible. Me gustaría que sirviera de ejemplo para todas aquellas personas con alguna discapacidad o no que por alguna razón, tienen que superar barreras que a veces nos parecen pesados muros infranqueables en el camino de la vida.

El día a día hace olvidar muchas veces esta situación a quienes no tenemos ningún tipo de discapacidad destacable. Aquí dejo este pequeño reconocimiento, y pido tolerancia, comprensión y apoyo a las personas que se encuentran dentro de este grupo y para aquellas que se encuentran con alguna dificultad a lo largo de su vida.

Este libro lo quiero dedicar también a mi familia, a mis amigos y amigas, y a quienes me estiman, ayudan y aprecian. De manera especial, se lo quiero dedicar a mis padres y mis abuelos que me soportan y comprenden.

Índice

CAPÍTULO 1: SOBRE VISUAL BASIC Y ESTE LIBRO

- 1.1.- ¿Visual Basic .NET 2005 ó Visual Basic 2005?
- 1.2.- ¿Visual Studio 2005 ó Visual Basic 2005 Express Edition?
- 1.3.- Microsoft .NET Framework 2.0
- 1.4.- ¿Es necesario utilizar un entorno de desarrollo rápido con Microsoft .NET?
- 1.5.- Cómo y por dónde empezar este manual
- 1.6.- POO, la base de .NET

CAPÍTULO 2: MICROSOFT .NET FRAMEWORK

- 2.1.- SDK
- 2.2.- BCL o Base Class Library
- 2.3.- CLR o Common Language Runtime
- 2.4.- MSIL
- 2.5.- JIT

CAPÍTULO 3: VISUAL BASIC 2005, EL LENGUAJE

- 3.1.- Tipos de datos
 - 3.1.1.- Tipos primitivos
 - 3.1.2.- Declaración de constantes, un tipo de declaración especial
 - 3.1.3.- Declaración de variables
 - 3.1.4.- Palabras clave
 - 3.1.5.- Listas enumeradas
 - 3.1.6.- Matrices
- 3.2.- Comentarios y organización de código
- 3.3.- Control de flujo
 - 3.3.1.- Operadores lógicos y operadores relacionales
 - 3.3.2.- If...Then...Else
 - 3.3.3.- Select...Case
- 3.4.- Bucles
 - 3.4.1.- Bucles de repetición o bucles For
 - 3.4.2.- Bucles Do While...Loop y Do Until...Loop
- 3.5.- Estructuras
- 3.6.- Operadores aritméticos

CAPÍTULO 4: VISUAL BASIC 2005, OTRAS CARACTERÍSTICAS DEL LENGUAJE

- 4.1.- Métodos
- 4.2.- Parámetros como valor y parámetros como referencia
- 4.3.- Funciones
- 4.4.- Propiedades
- 4.5.- Excepciones
- 4.6.- Colecciones
- 4.7.- Ámbito y visibilidad de las variables
- 4.8.- Clases
 - 4.8.1.- Utilizando Namespace
 - 4.8.2.- Utilizando el constructor de la clase
 - 4.8.3.- Utilizando constructores múltiples
 - 4.8.4.- Destruyendo la clase
 - 4.8.5.- Clases parciales
- 4.9.- Estructuras

CAPÍTULO 5: VISUAL BASIC 2005, OTROS ASPECTOS AVANZADOS DEL LENGUAJE

- 5.1.- Funciones recursivas
- 5.2.- Interfaces
- 5.3.- Eventos
- 5.4.- Multihebras o Multithreading
- 5.5.- Delegados
- 5.6.- Herencia

CAPÍTULO 6: VISUAL BASIC 2005, EL ENTORNO

- 6.1.- Visión general del entorno
- 6.2.- Creando una nueva aplicación
- 6.3.- El Cuadro de herramientas
- 6.4.- El Explorador de soluciones
- 6.5.- Los Orígenes de datos
- 6.6.- Ventana de propiedades
- 6.7.- Agregar elementos al proyecto
- 6.8.- Agregando elementos al proyecto
 - 6.8.1.- Windows Forms
 - 6.8.2.- Cuadro de diálogo
 - 6.8.3.- Formulario del explorador
 - 6.8.4.- Formulario primario MDI
 - 6.8.5.- Cuadro Acerca de
 - 6.8.6.- Pantalla de bienvenida
 - 6.8.7.- Otras plantillas

CAPÍTULO 7: VISUAL BASIC 2005, TRABAJANDO CON EL ENTORNO

- 7.1.- Código vs Diseñador
- 7.2.- Ejecutando una aplicación
- 7.3.- Diferencias entre Iniciar depuración e Iniciar sin depurar
- 7.4.- Depurando una aplicación
 - 7.4.1.- Puntos de interrupción
 - 7.4.2.- Deteniendo la depuración
 - 7.4.3.- Visión práctica de la depuración de un ejemplo
 - 7.4.4.- Modificando el código en depuración
- 7.5.- Utilizando los recortes como solución a pequeños problemas
- 7.6.- Diseñando nuestras aplicaciones Windows
 - 7.6.1.- Cuadro de herramientas
 - 7.6.2.- Controles contenedores
 - 7.6.3.- Posicionando los controles en nuestros formularios
 - 7.6.4.- Tabulando los controles en nuestros formularios
- 7.7.- Las propiedades de un proyecto

CAPÍTULO 8: MY, NAMESPACE PARA TODO

- 8.1.- ¿En qué consiste My?
- 8.2.- Funcionamiento de My
- 8.3.- Una primera toma de contacto con My
- 8.4.- El corazón de My

CAPÍTULO 9: XML, LOS DOCUMENTOS EXTENSIBLES

- 9.1.- Agregando la referencia a System.Xml
- 9.2.- Leer XML con XmlTextReader
- 9.3.- Leer XML con XmlDocument
- 9.4.- Leer XML con XPathDocument
- 9.5.- Leer un XML como un DataSet
- 9.6.- Ejemplo práctico para escribir un documento XML

CAPÍTULO 10: BREVE INTRODUCCIÓN AL ACCESO A DATOS

- 10.1.- Una pequeña introducción a ADO.NET
- 10.2.- ¿Acceso conectado o acceso desconectado?
- 10.3.- DataSet, DataView y DataTable
- 10.4.- Ejemplo de conectividad de datos con DataSet
- 10.5.- Recorriendo los datos de un DataSet
- 10.6.- Ejemplo de acceso conectado de datos

CAPÍTULO 1

SOBRE VISUAL BASIC Y ESTE LIBRO

ESTE CAPÍTULO INTRODUCTORIO ACERCA AL LECTOR ALGUNOS ASPECTOS BÁSICOS QUE CONVIENE QUE CONOZCA ANTES DE EMPEZAR A LEER ESTE MANUAL.

Visual Basic tiene el honor de haber sido el lenguaje de programación más extendido y utilizado en la historia de la informática. Pero lejos de haberse quedado anclado en el pasado, este lenguaje ha continuado evolucionando a lo largo de los últimos años.

Con la aparición de la tecnología Microsoft .NET, Visual Basic sufrió la transformación más amplia que jamás haya tenido este lenguaje de programación.

Microsoft elaboró entonces la primera especificación de esta evolución que ha tenido Visual Basic. Hablo de la especificación del lenguaje Visual Basic 7.0, y que sería la que se incorporaría a Visual Basic .NET 2002.

Poco tiempo después, la especificación del lenguaje Visual Basic sufrió pequeños retoques que se incorporaron a la especificación del lenguaje Visual Basic 7.1 y que formaría parte de Visual Basic .NET 2003.

Microsoft sin embargo, no se ha detenido aquí y así ha elaborado la especificación del lenguaje Visual Basic 8.0 que es la especificación que forma parte de Visual Basic 2005 y en la lógicamente, me he basado para escribir este manual.

1.1.- ¿Visual Basic .NET 2005 ó Visual Basic 2005?

Con la última especificación añadida al lenguaje Visual Basic por Microsoft, el lenguaje pasa a denominarse Visual Basic 2005. La palabra .NET ya no acompaña al lenguaje Visual Basic como ha ocurrido con las versiones 2002 y 2003 de Visual Studio. De hecho, ahora Visual Studio en su nueva versión pasa a llamarse Visual Studio 2005.

Por esa razón, en este libro haremos referencia solamente a Visual Basic 2005. Algo a lo que deberemos acostumbrarnos si hemos utilizado anteriormente términos como Visual Basic .NET 2002, Visual Basic .NET 2003, Visual Studio .NET 2002, etc.

1.2.- ¿Visual Studio 2005 ó Visual Basic 2005 Express Edition?

Microsoft está haciendo un importante esfuerzo por acercar y llevar Visual Basic al mayor número de programadores posible. Microsoft pone a disposición de los desarrolladores un entorno integrado de desarrollo de aplicaciones con los lenguajes de programación que Microsoft incorpora a su entorno de desarrollo rápido. Este entorno de desarrollo se llama Visual Studio 2005, y con él podemos desarrollar prácticamente cualquier tipo de aplicación que necesitemos crear.

Por otro lado, y con el objetivo principal de acercar la programación a los programadores, Microsoft ha desarrollado entornos de desarrollo rápido, económicamente más asequibles y sin todas las bondades que ofrece Visual Studio. Son entornos de desarrollo para programadores menos exigentes o con unos requerimientos menores, o un poder adquisitivo más bajo que no requiera de todas las posibilidades que ofrece un paquete como Visual Studio.

De esta manera, aparecen los entornos de desarrollo denominados *Express Edition* y que en el caso de Visual Basic, se denomina **Visual Basic 2005 Express Edition**.

Este manual está orientado precisamente a conocer Visual Basic 8.0 ó lo que es lo mismo, Visual Basic 2005, de la mano del entorno de desarrollo rápido **Visual Basic 2005 Express Edition**.

1.3.- Microsoft .NET Framework 2.0

El conjunto de librerías y el propio corazón de .NET que permite compilar, depurar y ejecutar aplicaciones .NET se denomina Microsoft .NET.

Desde que apareció Microsoft .NET, han aparecido tres versiones de Microsoft .NET Framework. La versión Microsoft .NET Framework 1.0 apareció en primer lugar y fue la que se utiliza dentro de Visual Studio .NET 2002. Poco más tarde apareció Microsoft .NET Framework 1.1 que fue integrada en Visual Studio .NET 2003. Actualmente, Microsoft ha desarrollado la versión Microsoft .NET Framework 2.0 que es la versión que se utiliza en Visual Studio 2005 y en las versiones *Express Edition* de la nueva familia de entornos de desarrollo rápido de Microsoft. Adicionalmente, SQL Server 2005 utiliza también esta versión de .NET.

1.4.- ¿Es necesario utilizar un entorno de desarrollo rápido con Microsoft .NET?

No, no es necesario utilizar un entorno de desarrollo rápido como *Visual Studio 2005* ó *Visual Basic 2005 Express Edition* para poder desarrollar aplicaciones en Microsoft .NET. De hecho, hay otras alternativas algunas de ellas de carácter gratuito disponibles en Internet, para crear aplicaciones .NET de forma rápida y cómoda. Otra manera que tenemos de desarrollar y compilar aplicaciones de .NET es utilizando el SDK o *Software Development Kit* de .NET Framework y la línea de comandos para realizar todas las acciones de compilación, depuración y ejecución.

Aún y así, Microsoft ha desarrollado en el entorno Visual Studio, el software de desarrollo rápido de aplicaciones .NET más robusto y completo del mercado para la tecnología de desarrollo de Microsoft.

Microsoft .NET Framework se integra en el entorno de desarrollo de forma transparente al programador, y lo utiliza para ayudarnos a desarrollar de forma rápida, eficiente y segura, nuestras aplicaciones. Nos permite aumentar el rendimiento y disminuir la curva de tiempo de desarrollo enormemente, por lo que aunque el uso de un entorno de desarrollo rápido para programar aplicaciones .NET es algo que corresponde decidirlo al programador, la conclusión que podemos extraer de esto que comento, es que no es necesario utilizarlo, pero sí muy recomendable.

1.5.- Cómo y por dónde empezar este manual

Este manual está enfocado a aquellos desarrolladores que quieren aprender a utilizar Visual Basic 2005. No se trata de un manual avanzado de Visual Basic 2005, sino de un manual enfocado a la toma de contacto del lenguaje con la base puesta en su introducción, y en el contacto del entorno de desarrollo *Visual Basic 2005 Express Edition*.

Entre otras cosas, no se tendrá en cuenta si el lector viene del mundo Visual Basic o no. El objetivo de este manual es el de servir de base y guía para iniciarse en el desarrollo de aplicaciones con Visual Basic 2005.

Lógicamente, lo primero que necesitaremos conocer son las nociones básicas y generales de Microsoft .NET Framework que nos permitirán familiarizarnos con esta tecnología. Si ya conoce cuales son las partes fundamentales de Microsoft .NET Framework y sabe como funciona, le sugiero pasar directamente al capítulo 3, pero si quiere conocer qué es y cómo funciona Microsoft .NET, entonces le sugiero continuar leyendo los capítulos de este manual de forma secuencial.

1.6.- POO, la base de .NET

Todos los lenguajes de programación de la plataforma .NET, entre ellos Visual Basic 2005, son lenguajes de programación orientados a objetos, por lo que es ampliamente recomendable, tener nociones de programación orientada a objetos para sacar el máximo provecho a este manual y al lenguaje Visual Basic 2005.

CAPÍTULO 2

MICROSOFT .NET FRAMEWORK

ESTE CAPÍTULO INTRODUCTORIO ACERCA AL LECTOR ALGUNOS ASPECTOS BÁSICOS QUE DEBE CONOCER ANTES DE EMPEZAR A LEER ESTE MANUAL.

Microsoft .NET es un entorno integrado de ejecución, compilación, depuración, y desarrollo de aplicaciones. Los diferentes lenguajes de programación de la plataforma, comparten el mismo entorno, normas, reglas, y librerías de Microsoft .NET Framework.

Las reglas sintácticas y algunas diferencias más, son las partes destacables entre un lenguaje de programación y otro dentro de .NET, pero la cantidad de cosas que comparten en común es una de las partes que ha hecho a .NET, un entorno moderno, robusto y eficiente que cubre las expectativas de los desarrolladores modernos y más exigentes.

2.1.- SDK

El SDK es conocido también como *Software Development Kit*, y es el paquete con el cuál, podemos compilar, ejecutar, y depurar nuestras aplicaciones, y utilizar las bibliotecas de clases de .NET que nos facilita enormemente una enorme cantidad de trabajo.

Adicionalmente, el SDK tiene también documentación, ayuda, y ejemplos, y viene preparado con varios compiladores, entre los que está Visual Basic.

Con el SDK, seremos capaces de desarrollar cualquier tipo de aplicación, aunque lo ideal es disponer de un entorno de diseño rápido para poder potenciar la programación y sacar un mayor aprovechamiento y rendimiento a nuestro trabajo y a nuestro tiempo.

2.2.- BCL o Base Class Library

El BCL o bibliotecas de clases de .NET son un enorme conjunto de clases -más de 4000- que poseen una amplia funcionalidad y que nos servirán para desarrollar cualquier tipo de aplicación que nos propongamos. Adicionalmente, podemos desarrollar nuestras propias clases y con eso, podemos contribuir con nuestra experiencia a nuestros desarrollos.

Microsoft .NET Framework, no deja de ser por lo tanto en parte, un enorme repositorio de clases listas para usar desde que instalamos el entorno .NET. En .NET Framework, referenciamos a las BCL mediante lo que se ha denominado Namespace -Espacios de Nombres- y que se engloban dentro del *Namespace System*.

2.3.- CLR o Common Language Runtime

Una de las partes fundamentales de Microsoft .NET Framework, es el CLR o Common Language Runtime, que no es otra cosa que el entorno o motor de ejecución de lenguaje común.

Todo código escrito en .NET es ejecutado bajo el control del CLR como código administrado. Es aquí dónde encontramos una de las diferencias más notables entre las versiones de Visual Basic anteriores a .NET y las versiones de Visual Basic que tienen que ver con la plataforma .NET. Antes de .NET, las aplicaciones desarrolladas con Visual Basic se ejecutaban como código no administrado, mientras que las aplicaciones desarrolladas con Visual Basic bajo el entorno .NET, se ejecutan como código administrado, código administrado siempre por el CLR.

El CLR es el centro neurálgico del .NET Framework encargado de gestionar la ejecución de nuestras aplicaciones, aplicar parámetros de seguridad y ejecutar el denominado recolector de basura entre otras cosas.

Dentro del CLR, encontraremos diferentes partes como ya hemos indicado, cada una responsable de su parcela o responsabilidad. Así podemos encontrar el CTS o Common Type Specification, o Especificación de Tipos de Datos Común. El CTS lo forma los tipos y definiciones de cada tipo de dato utilizable en una aplicación .NET. Cada tipo de dato, hereda su tipo del objeto **System.Object**. El CTS está por otro lado, relacionado con el CLS o Common Language Specification, o lo que es lo mismo, la Especificación Común de Lenguajes, que son las reglas que hay que seguir a la hora de trabajar con los tipos de datos.

Por último, no quiero dejar de hablar brevemente del CLR sin haber mencionado al Garbage Collector o GC, que en su traducción más o menos exacta, lo definiremos como Recolector de Basura, y que tiene la función digna o indigna, de hacer la tarea *más* sucia de .NET, es decir, de hacer las funciones de gestor de limpieza de .NET eliminando de la memoria, todos aquellos objetos que no sean útiles en un momento dado, liberando al sistema de recursos no utilizables. La ejecución del GC es una ejecución desatendida y transparente por el programador y por el usuario, pero si lo deseamos, podemos forzar como programadores, su ejecución bajo demanda.

2.4.- MSIL

MSIL o IL es conocido como Microsoft Intermediate Language o simplemente Intermediate Language, o lo que es lo mismo, lenguaje intermedio.

Todos los lenguajes administrados de la plataforma .NET, deben cumplir un conjunto de reglas y normas, y parte de este ajuste, es que una aplicación escrita en un lenguaje de programación determinado, debe ser compilada en un lenguaje intermedio, de manera tal, que una aplicación escrita por ejemplo en C# y otra igual en Visual Basic, se compilan al prácticamente el mismo lenguaje intermedio.

El IL es un lenguaje muy similar al conocido ensamblador, y contiene instrucciones de bajo nivel. La particularidad del IL es que no está compilado teniendo en cuenta ningún tipo de sistema operativo ni ningún dispositivo hardware en particular, por lo que al final de este proceso, es necesario realizar un último ajuste, el correspondiente a la ejecución de la aplicación del código intermedio en la máquina final donde se ejecuta.

2.5.- JIT

JIT son las siglas de Just In Time, o lo que es lo mismo, el procedimiento de .NET mediante el cuál, una aplicación compilada en código intermedio, es compilada cuando se lanza y ejecutada en última instancia de acuerdo al compilador que transformará el IL en instrucciones de ensamblador específicas para el sistema operativo en el cuál se está ejecutando.

Como particularidad de esto, mencionaré el funcionamiento del CLR de Mono, proyecto de acercar .NET al mundo Linux. Podemos escribir una pequeña aplicación de ejemplo que nos demuestre este funcionamiento, y compilarla en código intermedio. Copiaremos el resultado de esta compilación a código intermedio, y pondremos una copia en Windows bajo Microsoft .NET Framework y otra en Linux bajo Mono. Ejecutando nuestra aplicación en ambos sistemas y con las versiones correspondientes del entorno de ejecución, comprobaremos que ambas aplicaciones son ejecutadas en ambos sistemas operativos.

La explicación es sencilla, el código intermedio es el mismo para la aplicación Windows que para la aplicación Linux. El CLR de .NET Framework para Windows se encargará de compilar en modo JIT la aplicación para este sistema operativo, mientras que el CLR para Linux, se encargará de hacer lo propio para ese sistema operativo.

CAPÍTULO 3

VISUAL BASIC 2005, EL LENGUAJE

ESTE CAPÍTULO NOS ACERCA DE FORMA DIRECTA A LAS ESPECIFICACIONES GENÉRICAS DEL LENGUAJE VISUAL BASIC 2005.

Ya hemos comentado que Visual Basic ha evolucionado en los últimos años de forma paulatina, y así, nos encontramos con la versión más reciente de Visual Basic hasta la fecha. Hablo claramente de Visual Basic 2005.

En este capítulo, veremos las partes fundamentales de Visual Basic 2005 como lenguaje, para que sepamos realizar con él, cualquier trabajo que nos soliciten o que queramos desempeñar. Acompañeme entonces en este capítulo y en los dos siguientes, para que juntos aprendamos las partes fundamentales y más interesantes de este lenguaje.

3.1.- Tipos de datos

Ya hemos comentado algo acerca de los tipos de datos cuando en el capítulo anterior hablábamos de CTS, pero ¿cómo afecta esto en un lenguaje como Visual Basic 2005?.

En realidad, para el programador, escribir una aplicación en .NET es una tarea fácil. A independencia de lo que ocurría antes de .NET, cuando una aplicación interactuaba con otra, compartía funciones, o compartía código, el programador tenía la obligación de conocer cómo estaba diseñada y desarrollada una para poder hacer la segunda y que interactuaran entre sí sin que hubiera problemas.

En .NET, esta problemática ha desaparecido, y así, un programador que escriba una aplicación en C# por ejemplo con el fin de que interactúe con una aplicación desarrollada por otro programador en Visual Basic 2005, es independiente a la forma en la que se ha diseñado o escrito, ya que al compartir el mismo conjunto de tipos de datos de .NET, ambas aplicaciones se entenderán a la perfección.

Este aspecto conviene tenerlo en cuenta y tenerlo muy claro, pues el avance es considerable, y pocos piensan en ello. .NET nos trae como vemos, muchos avances, como los mismos Servicios Web que en este manual no trataremos y que ha resultado ser otro de los avances más interesantes de la informática, aunque inexplicablemente aún hoy y quizás por falta de conocimiento y comprensión de lo que es y de lo que puede hacer por nosotros, aún no ha estallado como era de esperar.

A continuación, veremos los tipos de datos que podemos encontrarnos en Visual Basic 2005, y como utilizarlos.

3.1.1.- Tipos primitivos

Los tipos definidos en .NET Framework, tienen su equivalente al tipo definido en Visual Basic 2005. A continuación, se exponen los diferentes tipos de datos de .NET y su correspondiente equivalencia en Visual Basic 2005.

.NET Framework	Visual Basic 2005
System.Boolean	Bolean
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 (valor sin signo -, sólo +)
System.UInt32	UInteger (valor sin signo -, sólo +)
System.UInt64	ULong (valor sin signo -, sólo +)

Tabla 3.1: tipos primitivos en .NET

Aún y así, podemos escribir nuestras aplicaciones utilizando ambos tipos de datos obteniendo el mismo resultado. La declaración de cualquier tipo de datos en Visual Basic 2005, se realiza por lo general salvo excepciones que veremos más adelante, declarando la variable anteponiendo la palabra reservada **Dim** delante de la variable seguida de la palabra reservada **As** y del tipo de dato a declarar. Podemos obviar también el tipo de dato a declarar, pero lo más recomendable es indicar el tipo de dato para evitar errores o malas interpretaciones de los datos.

Un ejemplo práctico de declaración y uso de variables dentro de un procedimiento, método o evento en Visual Basic 2005, es el que corresponde con el siguiente código de ejemplo:

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
    Dim intVar1 As Integer
    Dim intVar2 As Int32
    intVar1 = 12345
    intVar2 = intVar1
    MessageBox.Show("intVar1: " & intVar1 & " " & _
        intVar1.GetType.ToString() & vbCrLf & _
        "intVar2: " & intVar2 & " " & _
        intVar2.GetType.ToString())
End Sub
```

En este ejemplo podemos observar que hemos declarado dos variables, una de tipo **Integer** y otra de tipo **Int32** que según la tabla 1, son dos tipos de declaraciones equivalentes.

En este mismo ejemplo, mostramos un mensaje por pantalla con los resultados de las dos variables y con el tipo de variable que representa. En la figura 3.1 podemos observar este ejemplo en ejecución:

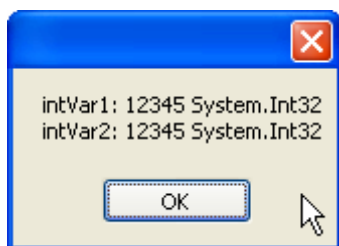


Figura 3.1: ejemplo de declaración de tipos de variables en ejecución.



Nota del código fuente:

Si apreciamos el código fuente anterior, observaremos que hay líneas de código que aparecen con el carácter `_` que indica a Visual Basic la continuación del código en la siguiente línea. La continuación de una instrucción de código en la línea siguiente, está formado por un espacio en blanco, seguido del carácter `_`.

Como vemos, .NET transforma el tipo de datos **Integer** de Visual Basic 2005 en un tipo de datos propio de .NET que es el tipo de datos **Int32**. Evidentemente, podemos declarar una variable dentro de nuestra aplicación de ambas formas, aunque lo más frecuente será hacerlo mediante la palabra reservada **Integer**.

Pero con todo y con esto, la declaración de variables en .NET puede hacerse más extensiva, ya que podemos declarar las variables también, utilizando unos símbolos o caracteres detrás de las variables, que indiquen el tipo de variable utilizada. La siguiente tabla aclarará en algo esto que comento, y el posterior ejemplo, terminará por explicarlo de forma práctica.

Tipo de Dato	Símbolo	Carácter
Short		S
Integer	%	I
Long	&	L
Single	!	F
Double	#	R
Decimal	@	D
UShort		US
UInteger		UI
ULong		UL
String	\$	

Tabla 3.2: declaración explícita de los tipos de datos en .NET

De esta manera, podemos declarar un objeto **-Object-** en un tipo de datos de forma implícita. El tipo de dato **Object**, es un tipo de dato que aún no hemos tratado y del cuál heredan el resto de tipos de datos. De hecho y por decirlo de alguna forma, es el comodín de los tipos de datos de .NET.

A continuación, veremos un sencillo ejemplo que explique y aclare el funcionamiento de los tipos de datos indicados así implícita o explícitamente.

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
    Dim intVar1 As Integer
    Dim intVar2 As Int32
    Dim intVar3, intVar4 As Object
    intVar1 = 12345
    intVar2 = intVar1
    intVar3 = 12345I
    intVar4 = 12345%
    MessageBox.Show("intVar1: " & intVar1 & " " & _
        intVar1.GetType.ToString() & vbCrLf & _
        "intVar2: " & intVar2 & " " & _
        intVar2.GetType.ToString() & vbCrLf & _
        "intVar3: " & intVar3 & " " & _
        intVar3.GetType.ToString() & vbCrLf & _
        "intVar4: " & intVar4 & " " & _
        intVar4.GetType.ToString())
End Sub
```

Este ejemplo en ejecución es el que puede verse en la figura 3.2.

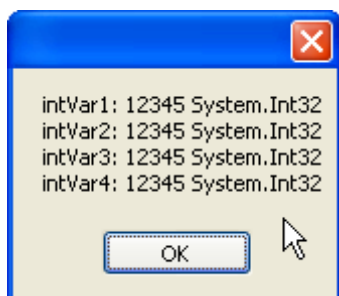


Figura 3.2: ejemplo de declaración implícita y explícita de tipos de datos en ejecución.

Otro aspecto a tener en cuenta en Visual Basic 2005, es la posibilidad de trabajar con dígitos en formato **Octal** y en formato **Hexadecimal**. Para esto, trabajaremos con el carácter **O** y el carácter **H** anteponiendo a estos caracteres el símbolo **&**.

El siguiente ejemplo práctico, nos muestra como trabajar con estos caracteres para indicar una cantidad en formato **Octal** o **Hexadecimal**.

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
    Dim Oct As Integer
    Dim Hex As String
    Oct = &O3
    Hex = &HA0
    MessageBox.Show("Valor Octal: " & Oct & _
vbCrLf & _
"Valor Hexadecimal: " & Hex)
End Sub
```

Otra significación con la asignación directa o indirecta de valores es el apartado referente al trabajo con fechas. Si queremos, podemos incluir una fecha entre los caracteres #, teniendo en cuenta que la fecha debe ir en formato MES/DIA/AÑO. Sirva el siguiente ejemplo como demostración de esto que estamos comentando:

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
    Dim dateVar As Date
    dateVar = #12/17/2005#
    MessageBox.Show(dateVar)
End Sub
```

3.1.2.- Declaración de constantes, un tipo de declaración especial

Antes de continuar adentrándonos en la declaración de variables, vamos a explicar un tipo de declaración muy útil y frecuentemente usado, la declaración y uso de constantes.

Dentro de una aplicación, puede ser adecuado e interesante la declaración y uso de variables constantes cuyo valor asignado, no sea modificable a lo largo de la aplicación y que se utilice para un caso o ejecución determinada.

El típico valor constante de ejemplo en toda demostración del uso y declaración de variables constantes es el valor **PI**. Para declarar una variable de tipo constante, tendremos que declarar el tipo de variable con la palabra reservada **Const** delante de la variable. El siguiente ejemplo, facilitará la comprensión de la declaración y uso de este tipo de constantes.

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
    Const PI As Double = 3.1416
    Dim dValor As Decimal
    dValor = (2 * PI) ^ 2
    MessageBox.Show(dValor)
End Sub
```

Este pequeño ejemplo demostrativo en ejecución del uso de declaración de constantes es el que puede verse en la figura 3.3.

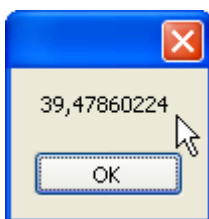


Figura 3.3: ejemplo demostrativo del uso de variables constantes.

3.1.3.- Declaración de variables

Ya hemos adelantado y ya hemos visto, como declarar variables en Visual Basic 2005. Anteriormente, hemos visto que la declaración de una variable dentro de un procedimiento, se realiza anteponiendo la palabra reservada **Dim** al nombre de la variable, seguida de la palabra reservada **As** y el tipo de dato declarado.

Un ejemplo sencillo sería:

```
Dim strMiVar As String
```

Pero hay un aspecto en la declaración de variables que conviene conocer, ya que este aspecto, es el diferenciador más importante entre el Visual Basic de la plataforma .NET y el Visual Basic anterior a .NET.

Si declaramos un conjunto de variables de un mismo tipo y las declaramos de la forma:

```
Dim strMiVar1, strMiVar2, strMiVar3 As String
```

Estaremos declarando las tres variables *strMiVar1*, *strMiVar2* y *strMiVar3* como variables de tipo **String**. En versiones anteriores a .NET de Visual Basic, esta misma declarativa, hacía que las dos primeras variables se declararan de tipo **Variant** y la última de tipo **String**.



Ojo:

El tipo de declaración **Variant** de versiones de Visual Basic anteriores a .NET, ya no existe. El tipo **Object**, es el tipo más adecuado para sustituir a este tipo de declaración.

De todas las maneras, en Visual Basic 2005, podemos declarar una variable y asignarla un valor inicial a la hora de declarar esta variable. El siguiente ejemplo, ayudará a comprender mejor esto:

```
Dim strMiVar As String = "Ejemplo en Visual Basic 2005"
```

Inclusive podemos declarar variables de forma anidada y asignarle valores directamente, como por ejemplo:

```
Dim Val1 As Integer = 2, Val2 As Integer = 3, Val3 As Integer = Val1  
+ Val2  
MessageBox.Show(Val3)
```

Este pequeño ejemplo nos mostrará el valor de la suma de las dos primeras variables declaradas e inicializadas como se indica en la figura 3.4.

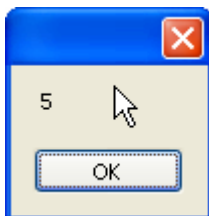


Figura 3.4: ejecución del ejemplo de inicialización de variables declaradas.

Otro aspecto destacable en la declaración de variables en Visual Basic 2005, es el uso y gestión de variables de tipo **String**.

Este tipo de variables se declaran como **Nothing** en su inicialización. Es decir, una variable **String** que se declara por primera vez, no se inicializa a "" -cadena vacía- sino que se inicializa a un valor **Nothing**. Como la teoría puede superar a veces a la práctica y sino *tocamos* no creemos, lo mejor es ver esto con un ejemplo que nos facilite la comprensión de todo lo que estamos explicando:

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
    Dim strValor As String
    If strValor Is Nothing Then
        MessageBox.Show("Nothing")
    Else
        MessageBox.Show("Tiene datos")
    End If
End Sub
```

Este primer ejemplo, mostrará en pantalla que el valor de la variable **strValor** es **Nothing**, y que por lo tanto, se trata de una variable no inicializada.

Dentro del entorno de *Visual Basic 2005 Express Edition*, recibiremos una pequeña ayuda visual al observar que la palabra **strValor** aparece subrayada indicándonos que no se ha asignado un valor a la variable. Pero esto sólo es un aviso, no es un error, y de hecho, la aplicación de ejemplo se puede compilar y ejecutar sin ninguna complicación.

```
Dim strValor As String
If strValor Is Nothing Then
    'La variable 'strValor' se utiliza antes de que se le haya asignado un valor. Por
    'ejecución.'
Else
    MessageBox.Show("Tiene datos")
```

Figura 3.5: mensaje de advertencia de Visual Basic 2005 sobre la inicialización de variable.

El segundo ejemplo sin embargo, es el mismo que el anterior con la salvedad de que la variable **strValor** es inicializada. En este segundo ejemplo, la variable ya no vale **Nothing**, sino que su valor ha cambiado al valor modificado en su asignación:

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
    Dim strValor As String = "Visual Basic 2005"
    If strValor Is Nothing Then
        MessageBox.Show("Nothing")
    Else
        MessageBox.Show("Tiene datos")
    End If
End Sub
```

Por último, mencionar un aspecto destacable sobre el carácter " en las cadenas de texto. Una cadena de texto debe de ser introducida entre caracteres ". Sin embargo, podemos encontrarnos con la posibilidad de que queremos escribir este mismo carácter dentro de la cadena de texto. Esto se hará duplicando siempre el carácter " que queremos escribir. Si por ejemplo queremos escribir la palabra *Hola*, declararemos nuestra variable como:

```
txtVar = "Hola"
```

Si por otro lado, deseamos escribir *Hol"a*, deberemos entonces escribir algo similar a:

```
txtVar = "Hol" "a"
```

De esta manera, aparecerá en pantalla el texto **Hol"a**.

3.1.4.- Palabras clave

Las palabras clave, son palabras con un significado especial dentro de un lenguaje de programación, en nuestro caso, dentro de Visual Basic 2005.

A las palabras clave, también se las conoce como palabras reservadas y no pueden ser utilizadas como identificadores excepto en casos en los cuales se puede forzar su uso. Esto último se consigue poniendo la definición del identificador entre los caracteres [y].

Un ejemplo práctico de este uso es el que se detalla a continuación:

```
Public Class Form1
    Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
        Dim [String] As String
        [String] = "Hola Visual Basic 2005"
        MessageBox.Show([String])
    End Sub
End Class
```

Las palabras clave de Visual Basic 2005 son las siguientes:

AddHandler	AddressOf	Alias	And
AndAlso	As	Boolean	ByRef
Byte	ByVal	Call	Case
Catch	CBool	CByte	CChar
CDate	CDBl	CDec	Char
CInt	Class	CLng	CObj
Const	Continue	CShort	CShort
CSng	CStr	CType	CUInt
CULng	CUShort	Date	Decimal
Declare	Default	Delegate	Dim
DirectCast	Do	Double	Each
Else	ElseIf	End	EndIf
Enum	Erase	Error	Event
Exit	False	Finally	For
Friend	Function	Get	GetType
Global	GoSub	GoTo	Handles
If	Implements	Imports	In
Inherits	Integer	Interface	Is
IsNot	Let	Lib	Like
Long	Loop	Me	Mod
Module	MustInherit	MustOverride	MyBase
MyClass	Namespace	Narrowing	New
Next	Not	Nothing	NotInheritable
NotOverridable	Object	Of	On
Operador	Option	Optional	Or
OrElse	Overloads	Overridable	Overrides
ParamArray	Partial	Private	Property
Protected	Public	RaiseEvent	ReadOnly
ReDim	REM	RemoveHandler	Resume
Return	SByte	Select	Set
Shadows	Shared	Short	Single
Static	Step	Stop	String
Structure	Sub	SyncLock	Then
Throw	To	True	Try
TryCast	TypeOf	UInteger	ULong
UShort	Using	Variant	Wend
When	While	Widening	With
WithEvents	WriteOnly	Xor	

3.1.5.- Listas enumeradas

Otro tipo de declaración es la que se hace a las denominadas listas enumeradas. Las listas enumeradas son listas de variables o datos que hacen referencia a índices y que empieza desde el 0 en adelante, aunque estos valores pueden ser alterados según nuestra conveniencia.

Para declarar una lista enumerada, tendremos que hacerlo utilizando la palabra reservada **Enum**. El siguiente ejemplo, demuestra como declarar una lista enumerada y como usar sus valores:

```
Private Enum Ejemplo
    Valor1
    Valor2
    Valor3
End Enum

Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.Load
    Dim MiEjemplo As Ejemplo
    MiEjemplo = Ejemplo.Valor1
    MessageBox.Show(MiEjemplo)
End Sub
```

Este pequeño ejemplo en ejecución es el que puede verse en la figura 3.6.



Figura 3.6: ejecución del ejemplo de inicialización de variables declaradas.

Aún y así y como ya hemos adelantado brevemente, podemos forzar a que la lista enumerada posea valores forzosos según nuestros intereses. El siguiente ejemplo, demuestra esto que comento:

```
Private Enum Ejemplo
    Valor1 = 3
    Valor2 = 5
    Valor3 = 1
End Enum

Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.Load
    Dim MiEjemplo As Ejemplo
    MiEjemplo = Ejemplo.Valor1
    MessageBox.Show(MiEjemplo)
End Sub
```

Este ejemplo en ejecución es el que puede verse en la figura 3.7.



Figura 3.7: ejecución del ejemplo de inicialización de variables declaradas forzando valores.

Pero también podemos forzar el tipo de datos de una lista enumerada. Eso sí, el tipo de datos debe de pertenecer al grupo de tipos de datos entero. La figura 3.8 muestra dentro del entorno de desarrollo Visual Basic 2005, este grupo de tipos de datos.

```
Private Enum Ejemplo as Integer
    Valor1 = 3
    Valor2 = 5
    Valor3 = 1
End Enum

Private Sub Form1_Load()
    Dim MiEjemplo As Integer
    MiEjemplo = Ejemplo.Valor3
    MessageBox.Show(MiEjemplo)
End Sub
```


Figura 3.8: declaración posible de tipos de datos dentro de una lista enumerada.

A tener en cuenta:

No es nada relevante, pero sí es algo que debemos tener en cuenta por si en algún momento dado, deseamos conocer el tipo o nombre de variable seleccionada de la lista enumerada. Utilizando los ejemplos anteriores, observamos las siguientes líneas de código:

```
Dim MiEjemplo As Ejemplo
MiEjemplo = Ejemplo.Valor2
MessageBox.Show(MiEjemplo.ToString() & " : " & MiEjemplo)
```

Esto dará como resultado el nombre de la lista enumerada seguida del valor, tal y como se indica en la siguiente figura.



En algunos momentos, este uso puede resultar especialmente útil.

3.1.6.- Matrices

La declaración de matrices es otra de las declarativas en Visual Basic 2005 que debemos tener en cuenta y que en más ocasiones podemos utilizar. La declaración de una matriz se realiza anteponiendo al nombre de la variable dos paréntesis abierto y cerrado, e indicando si así lo queremos, la dimensión de la matriz. Por ejemplo, podemos declarar una matriz de estas dos maneras posibles:

```
Dim MiArray() As Byte
```

Y de la forma:

```
Dim MiArray(10) As Byte
```

En el primero de los casos, hemos declarado una matriz de tipo **Byte** sin dimensiones, es decir, no se ha reservado en memoria el tamaño de la matriz declarada. En el segundo de los casos, hemos declarado la misma matriz con el mismo tipo de datos, indicando además el tamaño que deberá ocupar en memoria. Hemos reservado su espacio lo utilizemos o no.

Toda matriz declarada en Visual Basic 2005, empezará con el subíndice 0 y así en adelante. En el caso de la matriz declarada como **MiArray(10)**, esta matriz no tendrá 10 posiciones en memoria reservadas para ser utilizadas, sino 11, es decir, la cantidad de posiciones reservadas en memoria para una matriz, oscila desde 0 hasta el número de posiciones indicada más 1. `MiArray(10) = 10 + 1` posiciones.

Un ejemplo sencillo y que nos permita comprender esto de forma rápida es el que se detalla a continuación:

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
    Dim MiArray(10), I As Byte
    For I = 0 To 10
        MiArray(I) = I * 2
    Next
    Dim strValores As String = ""
    For I = 0 To 10
        strValores += MiArray(I) & " "
    Next
    MessageBox.Show(strValores.Trim)
End Sub
```

Nuestro ejemplo de demostración en ejecución es el que se puede ver en la figura 3.10.

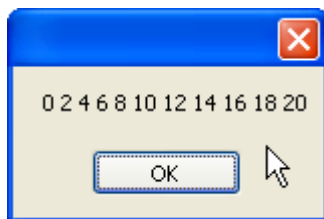


Figura 3.10: declaración posible de tipos de datos dentro de una lista enumerada.



Nota del código fuente:

La línea

```
strValores += MiArray(I) & " "
```

es equivalente a escribir:

```
strValores = strValores + MiArray(I) & " "
```

Se utiliza por lo tanto += en este caso, para concatenar un dato en otro.



Aviso:

Los índices inferiores de una matriz en Visual Basic 2005, siempre deben empezar por 0.

Otra particularidad es utilizar la palabra reservada **To** para indicar las posiciones reservadas para nuestra matriz. En sí, es una equivalencia y por ello, podemos declarar una matriz de estas formas:

```
Dim MiArray(10) As Byte
```

O bien:

```
Dim MiArray(0 To 10) As Byte
```

Una particularidad dentro de las matrices, es la inicialización de estas dentro de nuestras aplicaciones. Para realizar esto, basta con introducir los valores entre llaves. Los siguientes breves ejemplos, nos enseñan y demuestran como llevar a cabo esto.

```
Dim MiArray1() As Byte = {2}
Dim MiArray2() As String = {"Ejemplo"}
MessageBox.Show(MiArray1(0) & " - " & MiArray2(0))
```

El único requisito para inicializar matrices es no indicar el número de posiciones que tendrá esta.

Otra particularidad en el uso de matrices, son las matrices de más de una dimensión. Hasta ahora hemos visto como declarar y trabajar con matrices de una dimensión, pero en el caso de utilizar matrices de más de una dimensión, la forma es similar aunque no exactamente igual, ya que deberemos tener en cuenta que podemos movernos en dos direcciones dentro de la matriz, teniendo en este caso, tantas columnas como dimensiones haya.

Un ejemplo práctico de esto que comentamos es el que se detalla a continuación:

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
    Dim MiArray1(2, 1) As String
    MiArray1(0, 0) = "María"
    MiArray1(0, 1) = 28
    MiArray1(1, 0) = "Juan"
    MiArray1(1, 1) = 33
    MiArray1(2, 0) = "Lara"
    MiArray1(2, 1) = 23
    Dim I As Integer
    Dim strCadena As String = ""
    For I = 0 To 2
        strCadena += MiArray1(I, 0) & " " & MiArray1(I, 1) & vbCrLf
    Next
    MessageBox.Show(strCadena)
End Sub
```

Este ejemplo en ejecución es el que se puede observar en la figura 3.11.



Figura 3.11: ejecución de una matriz de más de una dimensión.

Otro aspecto a la hora de trabajar con matrices en Visual Basic 2005, es la capacidad de cambiar o modificar su tamaño o dimensión en memoria. Esto lo podemos lograr utilizando las palabras reservadas **ReDim** y **ReDim Preserve**.

Un ejemplo práctico que demuestre esto que estamos comentando es el que se detalla a continuación:

```
Dim MiArray() As String
ReDim MiArray(2)
MiArray(0) = "Uno" : MiArray(1) = "Dos" : MiArray(2) = "Tres"
ReDim Preserve MiArray(3)
MiArray(3) = "Cuatro"
MessageBox.Show(MiArray(0))
```



Nota del código fuente:

Podemos concatenar varias líneas o instrucciones de código fuente separándolas con el carácter `:` como se puede observar en el código anterior.

Por último, hablaremos de otro aspecto relacionado con las matrices. Hablamos de sus tamaños o dimensiones. En muchos momentos, nos puede resultar muy útil recorrer los elementos de una matriz para trabajar o manipular los datos que contiene ésta, pero puede ocurrir también, que no conozcamos la dimensión de una matriz en un momento dado, por lo que lo más útil es utilizar algunos métodos de Visual Basic 2005 que nos facilite estas tareas.

Esto se consigue utilizando la propiedad `Length` de la variable matriz declarada. Así nos dará el número de elementos que tiene la matriz. Un ejemplo del uso de este método sería:

```
Dim MiArray(2) As String
MiArray(0) = "Uno" : MiArray(1) = "Dos" : MiArray(2) = "Tres"
MessageBox.Show(MiArray.Length)
```

La única particularidad a tener en cuenta con el uso de este método, es que en nuestro ejemplo, `MiArray.Length` nos indicará 3 como resultado y no 2, ya que la información que nos está devolviendo esta propiedad es el número de elementos o dimensiones dentro de la matriz.

Si por otro lado, lo que deseamos es eliminar todos los elementos de una matriz de forma rápida, lo mejor será utilizar la palabra reservada `Nothing` para quitar las referencias de los elementos de la matriz en memoria. Un ejemplo práctico de esto que comentamos es el que se detalla a continuación:

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.Load
    Dim MiArray() As String
    ReDim MiArray(2)
    MiArray(0) = "Uno" : MiArray(1) = "Dos" : MiArray(2) = "Tres"
    MiArray = Nothing
    If MiArray Is Nothing Then
        MessageBox.Show("Nothing")
    Else
        MessageBox.Show(MiArray.Length)
    End If
End Sub
```

De hecho, tenemos que tener cuidado a la hora de trabajar con una matriz y crear otra dependiendo de la cantidad de elementos de la primera. Esto lo veremos a continuación.

Por eso y para casi concluir el completo apartado de las matrices diremos que en otras muchas ocasiones, consideraremos interesante el hecho de copiar una matriz en otra para manipular sus datos, ordenarlos o realizar cualquier otra acción que consideremos oportuna. El siguiente ejemplo demuestra esto:

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.Load
    Dim strMiArray() As String = {"Urano", "Neptuno", "Plutón", "Sol"}
    Dim strOtroArray() As String
    ReDim strOtroArray(strMiArray.Length - 1)
    strMiArray.CopyTo(strOtroArray, 0)
    MessageBox.Show(strOtroArray(0))
End Sub
```

Debemos notar la declaración de redimensionado de la segunda matriz utilizando para ello la propiedad `Length` para saber la dimensión de la matriz. Si observamos bien esto, veremos que estamos creando una segunda matriz con una dimensión igual a `strMiArray.Length - 1`. Recordemos que los elementos de la matriz empiezan por 0 y que esta propiedad informa del número total de elementos, siendo el índice 0 el primer elemento.

Ya para finalizar este apartado del uso, gestión y manipulación de matrices, hablaremos de otra forma de recorrer una matriz, utilizando para ello, el bucle **For Each**, un bucle especialmente útil para el tratamiento de este tipo de situaciones. El uso de este bucle sería de la forma siguiente:

```
For Each <cadena> In <matriz>
...
Next
```

Quizás un ejemplo, nos facilite muchísimo la comprensión del uso de este tipo de bucles:

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.Load
    Dim MiArray() As String = {"Uno", "Dos", "Tres"}
    Dim strCadena As String
    Dim strMsg As String = ""
    For Each strCadena In MiArray
        strMsg += strCadena & " "
    Next
    MessageBox.Show(strMsg.Trim)
End Sub
```

3.2.- Comentarios y organización de código

Cuando escribimos código en Visual Basic 2005 y también dentro del entorno *Microsoft Visual Basic 2005 Express Edition*, tenemos la posibilidad de añadir comentarios y organizar el código.

Para añadir comentarios a nuestro código, lo podemos hacer utilizando el carácter **'** o utilizando la palabra reservada **REM**. Un ejemplo del uso de comentarios sería por ejemplo:

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.Load
    'Esto es un comentario
    REM esto es otro comentario
End Sub
```

Otra alternativa que tenemos a la hora de escribir el código de nuestras aplicaciones es la posibilidad de añadir pequeñas porciones o regiones de comentarios que nos facilite la organización, clasificación, y ordenación del código de nuestro código. Esto lo lograremos utilizando la instrucción:

```
#Region "<texto>"
...
#End Region
```

Un ejemplo del uso de esta instrucción es la que se detalla a continuación:

```
#Region "Ejemplo"
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.Load
    'Esto es un comentario
    REM esto es otro comentario
End Sub
#End Region
```

Una particularidad de esta instrucción, es como hemos comentado ya, la posibilidad de organizar el código o instrucciones que quedan introducidas dentro de la instrucción **#Region "..."** ... **#End Region**. De hecho, en *Visual Basic 2005 Express Edition*, podemos contraer y expandir el contenido de estas instrucciones tal y como se muestra en las figuras 3.12 y 3.13.

```
Public Class Form1
    #Region "Ejemplo"
        Private Sub Form1_Load(ByVal send
            'Esto es un comentario
            REM esto es otro comentario
        End Sub
    #End Region
End Class
```

Figura 3.12: código expandido en el entorno de Visual Basic 2005 Express Edition.

```
Public Class Form1
    Ejemplo
End Class
```

Figura 3.13: código contraído en el entorno de Visual Basic 2005 Express Edition.

3.3.- Control de flujo

Cuando escribimos código, muchas veces tenemos uno o más caminos de realizar las tareas encomendadas, es decir, de elegir el flujo a través del cuál debe discurrir nuestra aplicación. Eso es justamente lo que veremos a continuación.

3.3.1.- Operadores lógicos y operadores relacionales

Uno de los aspectos a tener en cuenta a la hora de hacer comparaciones, utilizar o seleccionar un camino o flujo u otro, es el uso de operadores lógicos y operadores relacionales.

En la tabla 3.3, podemos observar los operadores relacionales con su correspondiente descripción:

Operador relacional	Descripción
=	Da como resultado verdadero si las expresiones comparadas son iguales
<	Da como resultado verdadero si la expresión de la izquierda es menor que la expresión de la derecha
>	Da como resultado verdadero si la expresión de la izquierda es mayor que la expresión de la derecha
<>	Da como resultado verdadero si la expresión de la izquierda es distinta que la expresión de la derecha
<=	Da como resultado verdadero si la expresión de la izquierda es menor o igual que la expresión de la derecha
>=	Da como resultado verdadero si la expresión de la izquierda es mayor o igual que la expresión de la derecha

Tabla 3.3: operadores relacionales en Visual Basic 2005

Por otro lado, en la tabla 3.4 podemos observar los operadores lógicos que en muchas ocasiones se utilizan junto a los operadores relacionales para combinar las diferentes acciones:

Operador lógico	Descripción
And	Da como resultado verdadero si las dos expresiones comparadas son verdaderas
Or	Da como resultado verdadero si una de las dos expresiones comparadas es verdadera
Not	Invierte el valor de la expresión. Dos negaciones es una afirmación, y negar una afirmación es el valor contrario de la afirmación
Xor	Da como resultado verdadero solamente si una de las dos expresiones comparadas

	es verdadera
AndAlso	Da como resultado verdadero si las dos expresiones comparadas son verdaderas, con la particularidad de que evalúa la segunda parte de la expresión si la primera la cumple
OrElse	Da como resultado verdadero si una de las dos expresiones comparadas es verdadera con la particularidad de que si cumple la primera expresión, no continúa con la siguiente dando por verdadera la comparación lógica

Tabla 3.4: operadores lógicos en Visual Basic 2005

3.3.2.- If...Then...Else

La construcción **If...Then...Else**, nos permite ejecutar una o más condiciones que podemos comparar utilizando operadores relacionales y operadores lógicos.

En este tipo de comparaciones, podemos comparar las condiciones de forma anidada o tan compleja como lo deseemos. Lo mejor es ver esto con un ejemplo práctico:

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.Load
    Dim intValor As Integer = 5
    If intValor > 5 Then
        MessageBox.Show("Valor mayor que 5")
    Else
        MessageBox.Show("Valor menor que 6")
    End If
End Sub
```

Este ejemplo en ejecución es el que se puede observar en la figura 3.14.

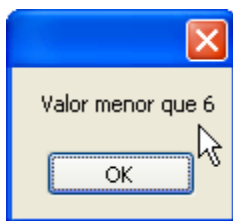


Figura 3.14: evaluación de la condición If...Then...Else.

Una particularidad de este tipo de instrucciones es la que corresponde a la palabra reservada **IIf** que se utiliza para ejecutar la condición de forma anidada en una sola instrucción, siendo su sintaxis la siguiente:

```
IIf(<expresión>, True, False)
```

Un ejemplo práctico del uso de esta cláusula que sólo corresponde a un tipo de uso especial de Visual Basic, es la siguiente:

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.Load
    Dim intValor As Integer = 5
    Dim strMsg As String = ""
    MessageBox.Show(IIf(intValor > 5, "Valor mayor que 5", "Valor menor que 6"))
End Sub
```

El resultado de ejecutar este ejemplo, es el mismo que el ejemplo anterior y que puede verse en la figura 14.

En el caso de querer comparar condiciones de forma anidada, emplearemos tantas cláusulas **Else** o **ElseIf** como deseemos. Un breve ejemplo de demostración de este uso es el siguiente:

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
    Dim intValue As Integer = 5
    If intValue > 5 Then
        MessageBox.Show("Valor mayor que 5")
    ElseIf intValue < 5 Then
        MessageBox.Show("Valor menor que 5")
    Else
        MessageBox.Show("Valor igual a 5")
    End If
End Sub
```

También y en el caso de utilizar una única condición, podemos declarar la instrucción **If** dentro de la misma línea de código sin usar **End If**, como por ejemplo:

```
If intValue > 5 Then MessageBox.Show("Valor mayor que 5")
```

3.3.3.- Select...Case

La sentencia **Select...Case** se utiliza por lo general, para evaluar varios grupos de sentencias e instrucciones dependiendo del valor de la expresión a evaluar. Hay diferentes formas de utilizar esta sentencia, las cuales veremos a continuación:

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
    Dim intValue As Integer = 5
    Select Case intValue
        Case Is > 5
            MessageBox.Show("Valor mayor que 5")
        Case Is = 5
            MessageBox.Show("Valor igual a 5")
        Case Is < 5
            MessageBox.Show("Valor menor que 5")
    End Select
End Sub
```

Observamos en este ejemplo, que el valor de la expresión *intValue* se cumple en alguno de los casos dentro de la sentencia de evaluación. En el caso de que no ocurra esto, podemos utilizar la sentencia **Case Else** como veremos a continuación:

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
    Dim intValue As Integer = 5
    Select Case intValue
        Case Is > 5
            MessageBox.Show("Valor mayor que 5")
        Case Is < 5
            MessageBox.Show("Valor menor que 5")
        Case Else
            MessageBox.Show("Valor igual a 5")
    End Select
End Sub
```

Pero dentro de esta sentencia, podemos también utilizar la palabra reservada **To** en la cláusula **Case** como por ejemplo **Case 1 To 5**. Para comprender mejor esto, lo mejor es verlo con un ejemplo:

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
    Dim intValue As Integer = 5
    Select Case intValue
        Case 1 To 3
```

```

        MessageBox.Show("Valor entre 1 y 3")
    Case 4 To 6
        MessageBox.Show("Valor entre 4 y 5")
    Case Else
        MessageBox.Show("Valor no comprendido entre 1 y 6")
    End Select
End Sub
    
```

También podemos comparar el valor de la expresión con varios valores, como por ejemplo:

```

    Case 1, 2, 3
    
```

3.4.- Bucles

En otras circunstancias a la hora de escribir nuestras aplicaciones, nos podemos ver interesados o incluso obligados a utilizar bucles en nuestro código, con el fin y objetivo de evaluar expresiones y sentencias.

Esto es justamente lo que veremos a continuación.

3.4.1.- Bucles de repetición o bucles For

Es uno de los bucles más extendidos para recorrer una determinada porción de código un número de veces limitado. Si ese número de veces se convierte en ilimitado, se denomina bucle infinito y la ejecución del proceso se volvería eterna, por lo que es un tipo de bucle a controlar en nuestras aplicaciones.

La nomenclatura de uso de este tipo de bucle es de la siguiente manera:

```

For <variable> = <valor inicial> To <valor final> Step <salto>
...
Next
    
```

Un ejemplo nos ayudará a comprender esto de forma sencilla y práctica:

```

Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
    Dim intValor As Integer = 5
    Dim I As Integer, strCadena As String = ""
    For I = 1 To intValor
        strCadena += Chr(I + 64)
    Next
    MessageBox.Show(strCadena)
End Sub
    
```

Este ejemplo en ejecución es el que puede observarse en la figura 3.15.



Figura 3.15: ejecución de un ejemplo de demostración del uso de un bucle For.

Como vemos en el ejemplo anterior, no hemos utilizado la palabra reservada **Step** ya que no es necesario utilizarla. A continuación veremos un ejemplo del uso de esta palabra reservada.

```

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

```

Dim intValue As Integer = 5
Dim I As Integer, strCadena As String = ""
For I = 1 To intValue Step 2
    strCadena += Chr(I + 64)
Next
MessageBox.Show(strCadena)
End Sub
  
```

Este ejemplo en ejecución es el que se puede observar en la figura 3.16.



Figura 3.16: ejecución de un ejemplo de demostración del uso de un bucle For con Step.

3.4.2.- Bucles Do While...Loop y Do Until...Loop

El uso del bucle **Do While...Loop** es realmente útil en el caso de querer realizar una o varias tareas mientras se cumpla una determinada condición. De esta manera, este bucle se utiliza de la forma:

```

Do While <condición>
...
Loop
  
```

Un ejemplo práctico que nos aleccione en el uso de este bucle sería el siguiente:

```

Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
    Dim intValue As Byte = 5
    Dim intResultado As Byte
    Do While intValue > 0
        intResultado += intValue
        intValue -= 1
    Loop
    MessageBox.Show(intResultado)
End Sub
  
```

Nuestro ejemplo en ejecución es el que puede verse en la figura 3.17.

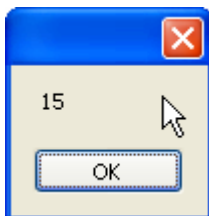


Figura 3.17: ejecución de un ejemplo de uso del bucle Do While.

Otra particularidad de este bucle es el uso del mismo mediante la siguiente forma:

```

Do
...
Loop While <condición>
  
```

De hecho, nuestro ejemplo anterior, quedaría de la forma:

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
    Dim intValor As Byte = 5
    Dim intResultado As Byte
    Do
        intResultado += intValor
        intValor -= 1
    Loop While intValor > 0
    MessageBox.Show(intResultado)
End Sub
```

Pero también podemos utilizar otro bucle, el bucle **Do Until...Loop** que nos permite ejecutar una o más sentencias de código, hasta que se cumpla una determinada condición. En este caso, la estructura del bucle sería de la forma:

```
Do Until <condición>
...
Loop
```

Un ejemplo práctico del uso de este tipo de bucles sería el que se detalla a continuación:

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
    Dim intValor As Byte = 5
    Dim intResultado As Byte
    Do Until intValor < 1
        intResultado += intValor
        intValor -= 1
    Loop
    MessageBox.Show(intResultado)
End Sub
```

De igual manera, podemos hacer uso de este bucle utilizándolo de la forma:

```
Do
...
Loop Until <condición>
```

En este caso, el ejemplo práctico del uso de este bucle sería de la forma:

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
    Dim intValor As Byte = 5
    Dim intResultado As Byte
    Do
        intResultado += intValor
        intValor -= 1
    Loop Until intValor < 1
    MessageBox.Show(intResultado)
End Sub
```

3.5.- Estructuras

Las estructuras están formadas por uno ó más miembros y cada miembro puede ser de un tipo de datos determinado, pudiendo tener una estructura con varios miembros de diferentes tipos de datos.

Por otro lado, las estructuras se definen de la forma siguiente:

```
Structure <nombre>
    Public <nombre> As <tipo>
End Structure
```

Un ejemplo práctico del uso y declaración de estructuras sería el siguiente:

```
Private Structure Persona
    Public Nombre As String
    Public Edad As Byte
End Structure

Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.Load
    Dim MiEst As Persona
    MiEst.Nombre = "Santiago"
    MiEst.Edad = "32"
    MessageBox.Show(MiEst.Nombre & " tiene " & MiEst.Edad & " años")
End Sub
```

Este ejemplo en ejecución es el que se muestra en la figura 3.18.



Figura 3.18: ejecución de un ejemplo de uso estructuras.

Visual Basic por otro lado, nos ofrece la posibilidad de trabajar con la palabra reservada **with** para realizar acciones repetitivas sin necesidad de escribir una porción de código muchas veces. En el caso de trabajar con estructuras, este uso es muy interesante, aunque su uso se puede extender a más situaciones. Un ejemplo práctico de este uso sería el siguiente:

```
Private Structure Persona
    Public Nombre As String
    Public Edad As Byte
End Structure

Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.Load
    Dim MiEst As Persona
    With MiEst
        .Nombre = "Santiago"
        .Edad = 32
    End With
    MessageBox.Show(MiEst.Nombre & " tiene " & MiEst.Edad & " años")
End Sub
```

3.6.- Operadores aritméticos

Los operadores aritméticos se utilizan en los lenguajes de programación para realizar operaciones matemáticas por lo general, aunque hay excepciones en las cuales utilizamos este tipo de operadores, para realizar operaciones con cadenas de texto.

Los operadores aritméticos que podemos usar en Visual Basic 2005 son los que se detallan en la tabla 3.5:

Operador aritmético	Descripción
+	Operador para realizar la suma de dos elementos
-	Operador para realizar la resta de dos elementos
*	Operador para realizar el producto de dos elementos
/	Operador para realizar la división de dos elementos

\	Operador para realizar la división entera (sin decimales) de dos elementos
^	Operador para elevar un número al exponente indicado
Mod	Operador para extraer el resto de la división de dos elementos

Tabla 3.5: operadores aritméticos en Visual Basic 2005

Para ver como el funcionamiento de operadores aún utilizándose para operaciones matemáticas, se pueden utilizar también en operaciones entre caracteres, vamos a ver un ejemplo que nos demuestre este hecho:

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.Load
    Dim I, J As Integer
    Dim strI, strJ As String
    I = 2 : J = 3
    strI = "Uno" : strJ = "Dos"
    MessageBox.Show(I + J & vbCrLf & strI + " " + strJ)
End Sub
```

Este ejemplo en ejecución es el que puede verse en la figura 3.19.

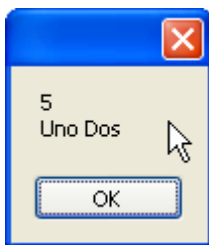


Figura 3.19: ejecución del ejemplo de demostración del uso de operadores aritméticos.

CAPÍTULO 4

VISUAL BASIC 2005, OTRAS CARACTERÍSTICAS DEL LENGUAJE

ESTE CAPÍTULO AVANZA EN EL RECORRIDO DE LAS ESPECIFICACIONES GENÉRICAS DEL LENGUAJE VISUAL BASIC 2005.

Ya hemos visto algunas de las partes más generales y necesarias para la comprensión de la programación con Visual Basic 2005.

A continuación, veremos algunas de las partes de tipo avanzado correspondientes al lenguaje Visual Basic 2005 y que conviene que conozcamos, cosas como la creación de métodos, clases, propiedades, conocer genéricamente el ámbito de las variables de una aplicación, etc.

4.1.- Métodos

Los métodos son las partes de código más cercanas a las funciones que se encargan de ejecutar una o varias instrucciones. El siguiente ejemplo, muestra la declaración y uso de un método sencillo:

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

Private Sub MiMetodo()
    Dim strNombre As String
    strNombre = "Visual Basic 2005"
    MessageBox.Show(strNombre)
End Sub
```

De todos los modos, los métodos pueden ser utilizados de manera tal, que podamos pasarlos argumentos con el fin de trabajar con ellos. Los argumentos pasados, pueden ser de cualquier tipo. Un ejemplo práctico sería el siguiente:

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.Load
    MiMetodo("Visual Basic 2005")
End Sub

Private Sub MiMetodo(ByVal strTxt As String)
    MessageBox.Show(strTxt)
End Sub
```

Otra particularidad a tener en cuenta cuando trabajamos con métodos, es la posibilidad de llamarlos por el nombre. Esto se consigue utilizando la instrucción **CallByName**. Un ejemplo del uso en Visual Basic 2005 de esta instrucción es la siguiente:

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

Private Sub MiMetodo()
    CallByName(Me, "Text", CallType.Set, "Mi Formulario")
End Sub
```

Aún y así, podemos hacer un uso de parámetros mucho más avanzados al que hemos hecho hasta ahora, es decir, podemos utilizar y pasar parámetros como valor o como referencia. ¿Qué significa esto?. Es justamente lo que veremos en el siguiente apartado.

4.2.- Parámetros como valor y parámetros como referencia

Cuando trabajamos con parámetros, podemos pasar un parámetro como valor **ByVal** o como referencia **ByRef**. El comportamiento de este tipo de valores es diferente. Los parámetros se pasan de la misma manera, pero el resultado puede ser completamente distinto.

Digamos que un parámetro pasado como valor, es para entendernos, una copia del objeto en memoria que será utilizada por el método para trabajar con ese valor, manipularlo, etc., y una vez que el método se destruye de la memoria al finalizar su uso, el valor de este parámetro desaparece.

Un parámetro pasado como referencia es ligeramente diferente, de hecho, en este caso se envía no una copia del elemento en memoria para que trabaje el método con él, sino un puntero al objeto original, por lo que todo lo que se realice con ese objeto, se está realizando directamente con él mismo, no con una copia de ese elemento.

La mejor manera de ver esto en funcionamiento es con ejemplos. A continuación veremos un ejemplo de cómo utilizar un método con parámetros pasados como valores:

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.Load
    Dim strValor As String = "Visual Basic"
    MiMetodo(strValor)
    MessageBox.Show(strValor)
End Sub

Private Sub MiMetodo(ByVal strTxt As String)
    strTxt += " 2005"
    MessageBox.Show(strTxt)
End Sub
```

En este caso, en pantalla el método nos devolverá el texto **Visual Basic 2005** primero, y **Visual Basic** después.

Pero este resultado cambia radicalmente cuando sobre el mismo código, modificamos el parámetro pasado como valor por un parámetro pasado como referencia. El código de ejemplo quedaría de la forma siguiente:

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.Load
    Dim strValor As String = "Visual Basic"
    MiMetodo(strValor)
    MessageBox.Show(strValor)
End Sub

Private Sub MiMetodo(ByRef strTxt As String)
    strTxt += " 2005"
    MessageBox.Show(strTxt)
End Sub
```

El resultado que se obtiene en este segundo caso es diferente al anterior, de hecho, el método nos devolverá el texto **Visual Basic 2005** primero, y **Visual Basic 2005** después, lo cual significa que hemos trabajado directamente con el parámetro inicial pasado como referencia dentro del método.

Otra particularidad al usar parámetros, es el uso de los denominados parámetros opcionales que se usan con la palabra reservada **Optional**. Los parámetros opcionales se utilizan por lo general para ejecutar necesidades

específicas, por lo que podemos pasar el parámetro marcado como opcional o no, es decir, no estamos obligados a pasarlos. Un ejemplo de uso de parámetros opcionales es el siguiente:

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.Load
    MiMetodo(2)
    MiMetodo(2, "VB 2005")
End Sub

Private Sub MiMetodo(ByVal intVal As Integer, Optional ByVal strVal As String = "")
    MessageBox.Show(IIf(strVal.Trim <> "", intVal & " y " & strVal, intVal))
End Sub
```

Como podemos observar en el ejemplo anterior, podemos llamar al método utilizando el valor opcional o no. La única obligatoriedad que tenemos en Visual Basic 2005 a la hora de describir el método con el valor opcional, es que tenemos que darle un valor por defecto. A la hora de llamar al método, podemos obviar el uso de ese parámetro o no. Si lo obvias, el método asignará al parámetro opcional el valor de inicialización indicado.

Pero no sólo podemos usar parámetros opcionales como aspecto destacable en el uso de parámetros dentro de nuestros métodos, también podemos utilizar un conjunto de valores pasados como parámetros sin importarnos cuántos valores hay en la matriz de parámetros y cuántos valores lo forman. Esto se logra con el uso de **ParamArray**, el cuál veremos a continuación en un sencillo ejemplo:

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.Load
    Dim Matriz() = {"Uno", "Dos", "Tres"}
    MiMetodo(Matriz)
End Sub

Private Sub MiMetodo(ByVal ParamArray MiMatriz())
    Dim Elemento As Object
    Dim strCadena As String = ""
    For Each Elemento In MiMatriz
        strCadena += Elemento & " "
    Next
    MessageBox.Show(strCadena.Trim)
End Sub
```

En este ejemplo, se pasa como parámetro una matriz con un conjunto **n** de elementos y que podemos manipular dentro del método sin tener en cuenta cuántos elementos lo contiene.

4.3.- Funciones

Las funciones son prácticamente idénticas a los métodos, con la salvedad de que devuelven valores para tratarlo en nuestras aplicaciones. De igual manera, podemos trabajar con funciones pasando parámetros por valor o parámetros por referencia. De hecho, vamos a ver también los dos ejemplos.

En el caso de pasar parámetros por valor, lo haremos de una forma similar a la empleada anteriormente:

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.Load
    MessageBox.Show(IIf(MiMetodo(128), "Valor > 100", "Valor <= 100"))
End Sub

Private Function MiMetodo(ByVal bteVal As Byte) As Boolean
    If bteVal < 101 Then
        Return False
    Else
        Return True
    End If
End Function
```

```
End If
End Function
```

Nuestro ejemplo en ejecución es el que se muestra en la figura 4.1.



Figura 4.1: ejecución de una función con parámetros por valor.

Opuesto como ya hemos comentado, es el comportamiento de las funciones que tienen parámetros pasados como referencia, al igual que pasaba con los métodos. Un ejemplo nos facilitará muchísimo la comprensión:

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
    MessageBox.Show(IIf(MiMetodo(128), "Valor > 100", "Valor <= 100"))
End Sub

Private Function MiMetodo(ByRef bteVal As Byte) As Boolean
    bteVal /= 2
    If bteVal < 101 Then
        Return False
    Else
        Return True
    End If
End Function
```

Nuestro ejemplo en ejecución de parámetros pasados como referencia es el que puede verse en la figura 4.2.

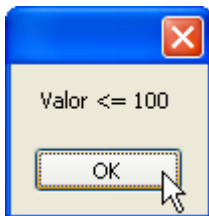


Figura 4.2: ejecución de una función con parámetros por valor.

4.4.- Propiedades

Las propiedades tienen una similitud enorme con los métodos. De hecho, son casi iguales, con la salvedad de que se utilizan como una caja en la cuál puedo meter y sacar un objeto con el valor y el estado en el que se encuentre.

Supongamos la propiedad **Color**. Imaginemos esta cualidad, el color, como una propiedad, es decir, sólo hay una y dependiendo de las modificaciones que podamos hacer en un instante *t*, esta propiedad tendrá un valor u otro. Imaginemos por lo tanto, que podemos *meter la mano en la caja* de esta propiedad **Color** y manipular y conocer su estado. Este es el funcionamiento de las propiedades en Visual Basic 2005 y se declara de la siguiente forma:

```
Private <_variable> As <tipo>

Property <variable> As <tipo>
    Get
        Return <_variable>
    End Get
```

```

Set(ByVal <valor> As <tipo>)
    <_variable> = <valor>
End Set
End Property
  
```

Lo mejor es ver esto que comentamos con un ejemplo práctico que nos ayude a entenderlo mejor.

```

Private _color As String = "Blanco"

Private Property Color() As String
    Get
        Return _color
    End Get
    Set(ByVal value As String)
        _color = value
    End Set
End Property

Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
    MessageBox.Show(Color)
    Color = "Rojo"
    MessageBox.Show(Color)
End Sub
  
```

4.5.- Excepciones

Visual Basic 2005 es un lenguaje orientado a objetos y por lo tanto, la gestión de errores se realiza mediante excepciones. Cuando se produce un error se lanza una excepción. Si utilizamos las instrucciones de código necesarias para gestionar las excepciones, tendremos un control claro en todo momento, sobre la ejecución de nuestras aplicaciones.

Las instrucciones para usar excepciones tienen una estructura como la que se detalla a continuación:

```

Try
    <sentencias>
Catch <tipo_excepción>
    <control_excepción>
Finally
    <sentencias>
End Try
  
```

La cláusula **Finally** no es estrictamente necesaria, aunque dependiendo de la excepción o código a tratar, puede resultar muy interesante. Un ejemplo práctico del uso y gestión de excepciones sería el siguiente:

```

Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
    Dim I As Integer = 1
    Try
        I = I / 0
    Catch ex As Exception
        MessageBox.Show(ex.Message)
    Finally
        I = 1
    End Try
End Sub
  
```

Este ejemplo en ejecución, es el que puede verse en la figura 4.3.

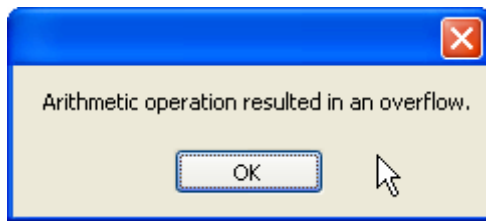


Figura 4.3: ejecución de un ejemplo del uso de excepciones.

Sin embargo, podemos utilizar mensajes preparados ya para gestionar las excepciones o personalizar éstas con nuestras propias instrucciones de gestión y uso de excepciones. Un ejemplo del uso de excepciones con mensajes ya preparados en Visual Basic 2005 sería el siguiente:

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.Load
    Dim I As Integer = 1
    Try
        I = I / 0
    Catch ex As OverflowException
        MessageBox.Show("Se ha producido un error de desbordamiento")
    End Try
End Sub
```

4.6.- Colecciones

Las colecciones funcionan de forma muy parecida a las matrices con la salvedad de que sus elementos pueden ser manipulados y recorridos de forma ligeramente diferente.

La nomenclatura de definición de una colección es mediante las instrucciones:

```
Dim <variable> As New Collection()
```

Existen diferentes métodos que podemos utilizar para manipular los elementos de una colección. El método **Add** lo utilizaremos para añadir un elemento a la colección, mientras que el método **Remove** lo utilizaremos para eliminar un elemento dado de la colección.

La particularidad entre colecciones y matrices, es que cuando se elimina un elemento de una colección, la lista de elementos se corre una posición a la izquierda ocupando el espacio del elemento eliminado. Lo mismo sucede para añadir un elemento a la colección. En el caso de las matrices sin embargo, podemos borrar el elemento que ocupa una posición determinada en una matriz, pero no podemos eliminar el elemento como en una colección. Y al añadir un elemento, lo añadiremos siempre al final de la matriz.

Un ejemplo práctico del uso de colecciones que ponga en práctica esto que comentamos es el siguiente:

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.Load
    Dim MiCol As New Collection()
    MiCol.Add("Uno")
    MiCol.Add("Dos")
    MiCol.Add("Tres")
    Dim I As Integer
    Dim strCadena As String = ""
    MiCol.Remove(2)
    For I = 1 To MiCol.Count
        strCadena += MiCol.Item(I) & " "
    Next
    MessageBox.Show(strCadena.Trim)
End Sub
```

Este ejemplo en ejecución es el que se muestra en la figura 4.4.



Figura 4.4: ejecución de un ejemplo del uso de excepciones.

4.7.- Ámbito y visibilidad de las variables

La declaración tipo de variables en Visual Basic 2005, es utilizando la palabra clave **Dim**. Cuando se declara una variable como **Dim**, se declara con el ámbito por defecto que es el ámbito de variable local.

La declaración de una variable como **Public** es extender la visibilidad de la variable a todo el ámbito. Por otro lado, una variable declarada como **Private** extiende la visibilidad de la variable a un ámbito local. Pero también podemos declarar las variables como **Protected** que permite acceder a la variable dentro de la propia clase en la que está declarada o en una clase derivada. Por último, también podemos declarar las variables como **Friend** mediante la cuál, sólo podrán acceder a estas porciones de código dentro del mismo ensamblado.

Adicionalmente, se pueden conjugar los permisos de ámbito utilizando combinaciones como **Protected Friend**.

Sin embargo, existen otras formas de declarar variables como mediante la palabra reservada **Shared** que especifica que estas variables no están asociadas con una instancia específica de clase o de estructura.

Otra particularidad de declaración de variables es el hecho de declararla mediante la palabra reservada **Shadows** que indica que el método o porción que está definiéndose, oculta al heredado de la clase base.

Por otro lado, podemos utilizar la declarativa **Static** para procurar que la variable declarada como tal, exista desde que se inicia el programa hasta el final.

Finalmente, podemos declarar las variables como **ReadOnly**, y en este caso, lo que estaremos indicando es que esta variable es de sólo lectura y no de escritura.

Aún y así, debemos tener en cuenta que podemos definir también variables dentro de métodos, bucles, etc., y el ámbito de la variable en este caso, cambia radicalmente. Un ejemplo nos ayudará a entender esto que comento de una manera clara y meridiana:

```
Public Class Form1
    Private I As Integer = 7

    Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
        Dim bolValor As Boolean = True
        If bolValor Then
            Dim I As Integer = 1
            MessageBox.Show(I)
        Else
            Dim I As Integer = 0
            MessageBox.Show(I)
        End If
        MessageBox.Show(I)
    End Sub
End Class
```

Si observamos este código de ejemplo detenidamente, veremos que dentro de la clase **Form1**, que es la clase a la que pertenece el formulario Windows, hemos declarado una variable de tipo **Integer** a la cuál hemos dado un valor.

En el evento **Form1_Load**, hemos declarado dos variables de tipo **Integer** que tienen el mismo nombre de declaración que la anterior variable declarada en la clase.

Esto significa que la variable **Integer** definida en la clase, tiene un ámbito general privado dentro de toda la clase. Es por lo tanto, visible dentro de toda la clase, pero el código del evento **Form1_Load** nos indica que estas mismas variables, son declaradas y visibles dentro de la porción de código en la cuál se declara, en nuestro caso un bucle **if**.

Dicho de otro modo, nuestro ejemplo, mostrará en pantalla el valor 1 primero, y el valor 7 después. El primer valor corresponderá al valor tomado dentro del bucle **if** y el segundo valor, al valor correspondiente al ámbito de visibilidad de la variable declarada en la clase, ya que en ese momento ya estamos fuera del bucle **if**.

4.8.- Clases

Las clases en .NET y en Visual Basic 2005 por lo tanto, tienen la singularidad de poderse programar y agrupar en nombres de espacio.

Para declarar una clase tipo, lo haremos de la forma:

```
Public Class <nombre_clase>
    ...
End Class
```

4.8.1.- Utilizando Namespace

El nombre de espacio o **Namespace**, es utilizado para organizar las clases e identificarlas de forma fácil. Esto se logra utilizando la siguiente instrucción:

```
Namespace <nombre>
    Public Class <nombre_clase>
        ...
    End Class
End Namespace
```

De esta manera, para hacer referencia a la clase desde un formulario Windows por ejemplo, lo haremos de la forma:

```
Dim <variable> As New <nombre de espacio>.<clase>
```

De hecho un ejemplo de definición y uso de una clase es el siguiente:

```
Namespace VB2005
    Public Class clsPrueba
        Public Function MiFuncion() As Boolean
            If Now.TimeOfDay.Hours > 12 Then
                Return True
            Else
                Return False
            End If
        End Function
    End Class
End Namespace
```

La siguiente porción de código, nos muestra como consumir la clase:

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
    Dim MiClase As New VB2005.clsPrueba
    MessageBox.Show(If(MiClase.MiFuncion(), "P.M.", "A.M. "))
End Sub
```

Sino hubiéramos codificado la clase utilizando la palabra clave **Namespace**, la declaración de uso de la clase sería de la forma ***Dim MiClase As New clsPrueba***.



Interesante:

Un formulario Windows no deja de ser una clase. Desde Visual Basic 2005 podemos crear nuestras propias clases, compilarlas y distribuirlas para reutilizarlas desde otros lenguajes de .NET como C# por ejemplo.

4.8.2.- Utilizando el constructor de la clase

El constructor de una clase en Visual Basic 2005, permite ser utilizado al instanciar cualquier clase. El constructor en Visual Basic 2005, se identifica por medio del método **New**. Todas las clases se inicializan con este método si lo tiene y en nuestras clases, podemos crearlo con parámetros o sin ellos. Un ejemplo de uso de una clase con este método de inicialización sería el siguiente.

Por un lado la definición de la clase de la forma:

```
Namespace VB2005
    Public Class clsPrueba
        Private _horas As Byte

        Private Property Horas() As Byte
            Get
                Return _horas
            End Get
            Set(ByVal value As Byte)
                _horas = value
            End Set
        End Property

        Public Sub New(ByVal Tiempo As Byte)
            Horas = Tiempo
        End Sub

        Public Function MiFuncion() As Boolean
            If Horas > 12 Then
                Return True
            Else
                Return False
            End If
        End Function
    End Class
End Namespace
```

Por otro lado, el uso de la parte consumidora de la clase, por ejemplo desde un formulario Windows, se realizaría de la forma:

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
    Dim MiClase As New VB2005.clsPrueba(Now.TimeOfDay.Hours)
    MessageBox.Show(If(MiClase.MiFuncion(), "P.M.", "A.M. "))
End Sub
```

4.8.3.- Utilizando constructores múltiples

Hemos visto en el apartado anterior como utilizar el constructor de una clase, pero lo que no hemos comentado aún es que se puede utilizar un constructor múltiple dentro de una clase. A continuación, se expone el ejemplo de una clase que utiliza más de un constructor:

```
Namespace VB2005
    Public Class clsPrueba
        Private _horas As Byte

        Private Property Horas() As Byte
            Get
                Return _horas
            End Get
            Set(ByVal value As Byte)
                _horas = value
            End Set
        End Property

        Public Sub New()
            Horas = Now.TimeOfDay.Hours
        End Sub

        Public Sub New(ByVal Tiempo As Byte)
            Horas = Tiempo
        End Sub

        Public Function MiFuncion() As Boolean
            If Horas > 12 Then
                Return True
            Else
                Return False
            End If
        End Function
    End Class
End Namespace
```

Así, llamar a una clase nos permite utilizar cualquiera de los constructores declarados, por lo que sería igualmente válido declarar la clase como *Dim MiClase As New VB2005.clsPrueba(Now.TimeOfDay.Hours)* o bien como *Dim MiClase As New VB2005.clsPrueba()*. Ambos casos de declaración son válidos.

**Interesante:**

Sino se define un constructor en una clase, Visual Basic 2005 lo genera por nosotros siendo un constructor sin parámetros.

4.8.4.- Destruyendo la clase

Una clase tiene un tiempo de vida determinado y cuando ya se finaliza su uso, se pone en marcha el recolector de basura o GC o *Garbage Collector* que hemos comentado en los capítulos iniciales de este manual. Teóricamente sin embargo, una clase se debería eliminar al asignarle el valor **Nothing** a la clase, aunque esto no significa que el recolector de basura se ponga en marcha y se elimine la clase totalmente de la memoria del ordenador.

4.8.5.- Clases parciales

Una nueva característica añadida a la especificación del lenguaje Visual Basic 2005 es lo que se denominan clases parciales, que son indicaciones de que la declaración de clase parcial forma parte de una clase principal.

De esta manera, podemos *romper* una clase en varios ficheros perteneciendo todos ellos a la misma clase principal.

De hecho y sin que nosotros lo sepamos, Visual Basic 2005 inicia los proyectos con clases parciales. Para comprobar esto, abriremos un nuevo proyecto de aplicación Windows dentro de *Visual Basic 2005 Express Edition* y haremos clic sobre el botón **Mostrar todos los archivos** del **Explorador de soluciones** tal y como se muestra en la figura 4.5.

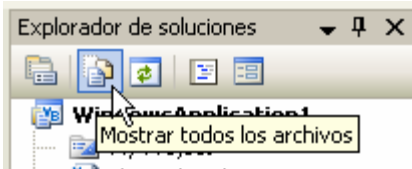


Figura 4.5: opción para mostrar todos los archivos del proyecto.

Al pulsar esta opción, observaremos que se muestran todos los archivos del proyecto, y entre ellos, observaremos que el formulario **Form1** contiene dos archivos adicionales, del cuál nos interesa ahora el denominado **Form1.Designer.vb** como se muestra en la figura 4.6.

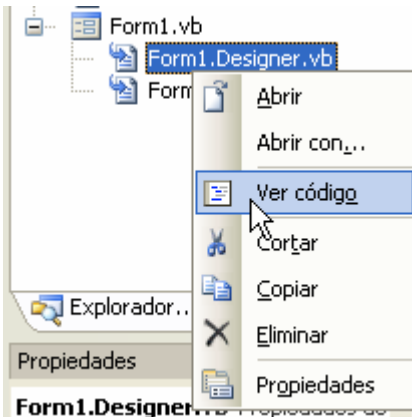


Figura 4.6: ficheros del formulario Form1.

Observaremos que hay dos ficheros pertenecientes a **Form1**, el anteriormente comentado **Form1.Designer.vb** y el **Form1.resx**. El primero de ellos es el que nos interesa destacar y el segundo, es el correspondiente a los recursos propios del formulario **Form1** y que no profundizaremos por no ser especialmente relevante en estos momentos.

Si observamos el código del fichero **Form1.Designer.vb**, veremos que encontramos el siguiente texto o parecido en la cabecera del código:

```
Partial Public Class Form1
```

Aquí lo que se está indicando es que esta clase, es una clase parcial de la clase **Form1**.

Como vemos, la definición de la clase parcial, se hace de la forma:

```
Partial Class <objeto>
```

Sin embargo, las clases parciales son clases, y por lo tanto, pueden ser utilizadas en formularios, clases propias, u otros objetos. A continuación veremos como utilizar una clase parcial en nuestro proyecto.

Iniciaremos un nuevo proyecto e insertaremos dos elementos de tipo **Clase** o **Class** en nuestro proyecto. Para hacer esto, haga clic con el botón derecho del ratón sobre el proyecto y seleccione la opción **Agregar > Nuevo elemento** del menú emergente, tal y como se muestra en la figura 4.7.

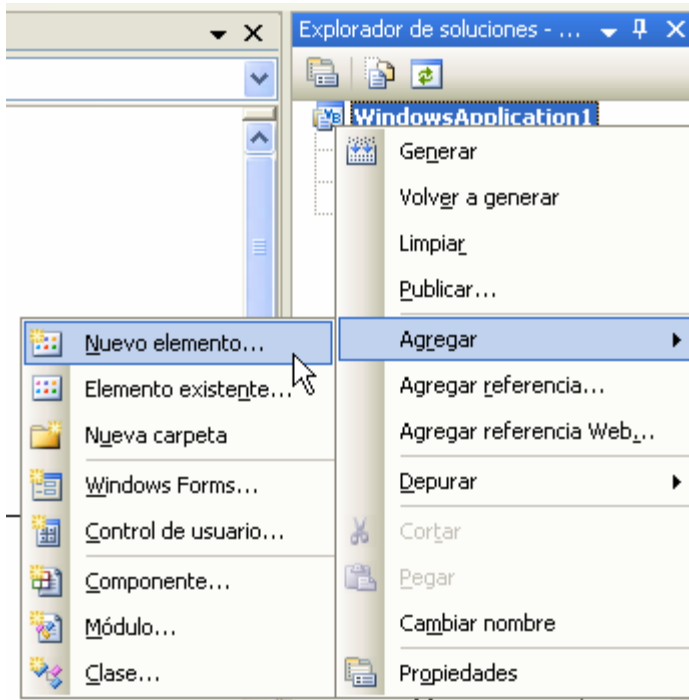


Figura 4.7: opción para agregar un nuevo elemento en Visual Basic 2005 Express Edition.

Dentro de la ventana que aparece correspondiente a la opción de **Agregar un nuevo elemento**, seleccione la plantilla de **Clase** y dele un nombre que considere apropiado como se muestra en la figura 4.8.

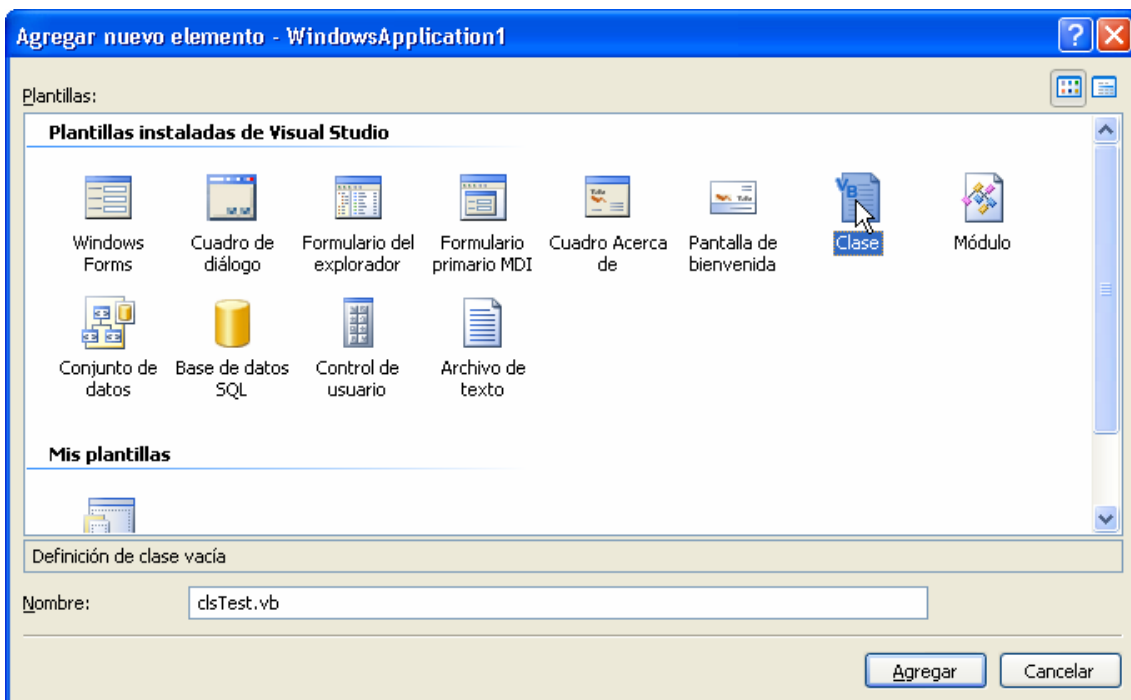


Figura 4.8: ventana para agregar un nuevo elemento de tipo clase al proyecto.

Repita esta operación otra vez más para añadir así, dos elementos de tipo **Clase** al proyecto. En mi caso, he dado los nombres de **clsPrueba.vb** y **clsTest.vb**.

A continuación, tan sólo nos quedará ponernos manos a la obra y escribir la parte de código que nos permita construir una clase de tipo *principal* y otra u otras de tipo *parcial*. Tan sólo conviene puntualizar dos aspectos sobre el tratamiento de clases parciales.

Por un lado, no nos importa para nada, el nombre del elemento de la clase sobre el cuál escribimos la parte principal de la clase y la parte parcial de la misma. Esto nos impide tener un mayor control de la parte principal de la clase, pero al mismo tiempo, permite que dentro de un equipo de desarrollo, haya multitud de desarrolladores trabajando con una clase principal y con métodos o funciones independientes.

Por otro lado, el interfaz *IntelliSense* de *Visual Basic 2005 Express Edition*, es capaz de reconocer directamente los métodos, propiedades, funciones, etc., de las clases principal y parcial. Esto es lo lógico, pero integrarlo en el entorno no es tarea sencilla, algo que .NET hace perfectamente.

Siguiendo con el ejemplo práctico de la creación y uso de clases parciales, vamos a crear la clase principal que podemos escribir dentro de cualquiera de los dos elementos de tipo *Clase* añadidos al proyecto anteriormente y que tendrá el siguiente código:

```
Public Class miClase

    Public Function Suma(ByVal Valor1 As Long, ByVal Valor2 As Long) As Long
        Return Valor1 + Valor2
    End Function

End Class
```

A continuación, añadiremos el código correspondiente al segundo elemento de tipo *Clase* y que corresponderá con la clase parcial. El código en este caso será el siguiente:

```
Partial Public Class miClase

    Public Function Resta(ByVal Valor1 As Long, ByVal Valor2 As Long) As Long
        Return Valor1 - Valor2
    End Function

End Class
```

A continuación consumiremos nuestra clase como si de una clase normal y corriente se tratara, por lo que haremos doble clic sobre el formulario **Form1** y escribiremos el siguiente código:

```
Public Class Form1
    Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
        Dim MiClase As New miClase()
        MessageBox.Show(MiClase.Suma(3, 2) & vbCrLf & MiClase.Resta(3, 2))
    End Sub
End Class
```

Si observamos la ayuda *IntelliSense* del entorno de trabajo, veremos que las funciones **Suma** y **Resta** están integradas en la clase, pese a pertenecer a una clase principal y una clase parcial, como se muestra en la figura 4.9.

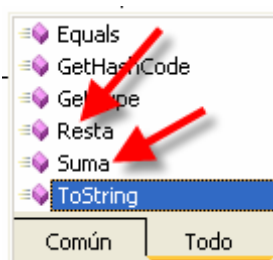


Figura 4.9: funciones de la clase principal y parcial añadidas a la ayuda Intelli-Sense.

Esta es la demostración fehaciente, de que ya trabajemos con clases principales o parciales, el entorno de .NET las trata a todas como partes de la misma clase. Como si fuera un mismo fichero de tipo *Clase*.

4.9.- Estructuras

Las estructuras son tipos de valor definidos por el usuario. Las estructuras se definen mediante la palabra reservada **Structure** y tiene la siguiente forma:

```
Structure <nombre>
    <Public, Private, Friend,...> <nombre> As <tipo>
End Structure
```

Un ejemplo práctico del uso de estructuras es el que se detalla a continuación:

```
Public Class Form1
    Private Structure MiEs
        Public Nombre As String
        Public Num_Hijos As Byte
        Public F_Nac As Date
    End Structure

    Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
        Dim Est As MiEs
        Est.Nombre = "Juan López"
        Est.F_Nac = "02/08/1980"
        Est.Num_Hijos = 0
    End Sub
End Class
```

Como podemos observar, a la hora de definir la estructura, utilizamos una instrucción del tipo:

```
Dim <nombre> As <nombre_de_estructura>
```

Sin embargo, también podríamos utilizar la instrucción de declaración de la estructura del modo:

```
Dim <nombre> As New <nombre_de_estructura>
```

La diferencia es que al poner **New** lo que estamos indicando es que vaya al constructor de la estructura, algo que no existirá a no ser que lo creamos nosotros, por lo que no es necesario definir la estructura con **New** sin tener un constructor.

Aún y así, y como podemos observar, es posible utilizar constructores dentro de nuestras estructuras como hicimos en el caso de las clases. Para hacer esto, deberemos utilizar la palabra reservada **New** en la definición de la estructura y en la declaración del uso de la estructura para de esta forma, ejecutar el constructor indicado. Un ejemplo práctico facilitará la comprensión de esto que comentamos:

```
Public Class Form1
    Private Structure MiEs
        Public Nombre As String
        Public Num_Hijos As Byte
        Public F_Nac As Date

        Sub New(ByVal strNombre As String, ByVal bte_Num_Hijos As Byte, ByVal
date_F_Nac As Date)
            Nombre = strNombre
            Num_Hijos = bte_Num_Hijos
            F_Nac = date_F_Nac
        End Sub
    End Structure

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



```

Dim Est As New MiEs("Juan López", 0, "02/08/1980")
MessageBox.Show(Est.Nombre & vbCrLf & _
                Est.F_Nac & vbCrLf & _
                Est.Num_Hijos)

End Sub
End Class

```

Pero aún hay más. Quizás deseemos utilizar constructores múltiples en nuestras estructuras. Esto es igualmente posible. Para ello, deberemos usar varios constructores **New** dentro de la declaración de la estructura tal y como se muestra en el siguiente ejemplo:

```

Public Class Form1
    Private Structure MiEs
        Public Nombre As String
        Public Shared Num_Hijos As Byte = 0
        Public F_Nac As Date

        Sub New(ByVal strNombre As String, ByVal bte_Num_Hijos As Byte, ByVal
date_F_Nac As Date)
            Nombre = strNombre
            Num_Hijos = bte_Num_Hijos
            F_Nac = date_F_Nac
        End Sub

        Sub New(ByVal strNombre As String, ByVal date_F_Nac As Date)
            Nombre = strNombre
            F_Nac = date_F_Nac
        End Sub
    End Structure

    Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
        Dim Est As New MiEs("Juan López", "02/08/1980")
        MessageBox.Show(Est.Nombre & vbCrLf & _
                        Est.F_Nac & vbCrLf & _
                        MiEs.Num_Hijos)

    End Sub
End Class

```

Otra particularidad de las estructuras, es la posibilidad de utilizarlas como estructuras parciales, como en el caso de las clases parciales anteriormente comentadas.

Para este caso, se utilizará las instrucciones de tipo:

```
Partial Structure <objeto>
```

Un ejemplo práctico del uso de estructuras parciales es el que se detalla a continuación:

```

Public Class Form1
    Private Structure MiEs
        Public Nombre As String
    End Structure

    Partial Structure MiEs
        Public Edad As Byte
    End Structure

    Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
        Dim Est As MiEs
        Est.Edad = 45
        Est.Nombre = "Isabel Gómez"
        MessageBox.Show(Est.Nombre & vbCrLf & _
                        Est.Edad)
    End Sub
End Class

```

```
End Sub  
End Class
```

Evidentemente, podemos conjugar con clases parciales y estructuras parciales combinadas en nuestros desarrollos según nuestras necesidades. Esto de la parcialidad, nos ofrece en .NET una gran flexibilidad, aunque nos fuerza también a perder parte del control sobre el código que escribimos sino tenemos una organización clara de lo que estamos haciendo en un proyecto.

CAPÍTULO 5

VISUAL BASIC 2005, OTROS ASPECTOS AVANZADOS DEL LENGUAJE

ESTE CAPÍTULO NOS MUESTRA ALGUNOS DE LOS ASPECTOS AVANZADOS DEL LENGUAJE VISUAL BASIC 2005.

El capítulo anterior ha sido el más denso de este manual. Quizás podríamos haber separado ese capítulo en varios o haber introducido en este capítulo parte del anterior, ya que el capítulo 4 contenía aspectos avanzados de la programación en Visual Basic 2005, aunque he preferido separar algunos de estos aspectos a este capítulo por ser acciones no de tan uso común como las anteriores.

5.1.- Funciones recursivas

La definición más simple de funciones recursivas es aquella que dice que se llama así a toda función que se llama así misma, de ahí su nombre.

El uso de este tipo de funciones es el habitual en cualquier lenguaje de programación, y por lo tanto, se indica aquí como ejemplo genérico del uso de este tipo de funciones en Visual Basic 2005.

El ejemplo típico de función recursiva es el cálculo del factorial de un número. El siguiente código lanza la función del cálculo del factorial de un número tantas veces como sea necesario:

```
Public Class Form1
    Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
        MessageBox.Show(Factorial(4))
    End Sub

    Public Function Factorial(ByVal N As Long) As Long
        If N = 1 Then
            Return 1
        Else
            Return N * Factorial(N - 1)
        End If
    End Function
End Class
```

Las funciones recursivas deben crearse solamente para facilitar o simplificar enormemente tareas repetitivas, recursivas, que permiten realizar una función general, como el cálculo del factorial de un número como el ejemplo que hemos visto.

5.2.- Interfaces

Las interfaces son una forma especial de una clase que sólo define miembros, a diferencia de una clase que contiene código de ejecución.

Al no existir herencia múltiple en Visual Basic 2005, debemos utilizar el concepto de **Interface** para implementar múltiples interfaces y hacer así, algo similar a la herencia múltiple.

Para definir una interfaz, utilizaremos las instrucciones:

```
Interface <nombre>
```

```
<...>
End Interface
```

Como vemos, la forma de declarar una interfaz es de la misma manera que hacíamos para declarar una clase. De hecho, lo único que debemos hacer es sustituir la palabra reservada **Class** por **Interface**.

La implementación por lo tanto, de una interfaz, es realmente sencilla. Sirva el siguiente ejemplo para mostrar como hacerlo:

```
Public Class Form1
  Implements Ejemplo

  Public Sub Saludo(ByVal strMensaje As String) Implements Ejemplo.Saludo
    MessageBox.Show(strMensaje)
  End Sub

  Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
    Saludo("Hola Mundo")
  End Sub

  Public Interface Ejemplo
    Sub Saludo(ByVal strMsg As String)
  End Interface
End Class
```

5.3.- Eventos

Los eventos nos permiten recoger cualquier interferencia o acción ocurrida en nuestra aplicación con el sistema, ya sea de manera forzada voluntariamente, como por ejemplo un clic de un usuario a un control como un botón, o de manera automáticamente al lanzarse una interrupción determinada como por ejemplo, saber cuando se refresca un control *Grid*.

Lo primero que veremos es como lanzar un evento en Visual Basic 2005. Crearemos un nuevo proyecto e insertaremos en el formulario un control **Button**. Esto lo haremos desplegando la ventana **Cuadro de herramientas** que hay a la izquierda del entorno de desarrollo *Visual Basic 2005 Express Edition* y haciendo doble clic sobre el control **Button** de la solapa **Controles comunes**, tal y como se muestra en la figura 5.1.

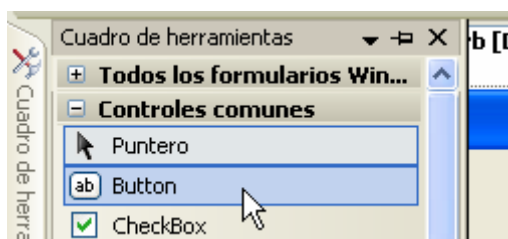


Figura 5.1: control **Button** del Cuadro de herramientas.

Repita la operación e inserte un segundo control **Button** en el formulario. De esta manera, tendremos dos controles **Button** insertados en el formulario como se muestra en la figura 5.2.

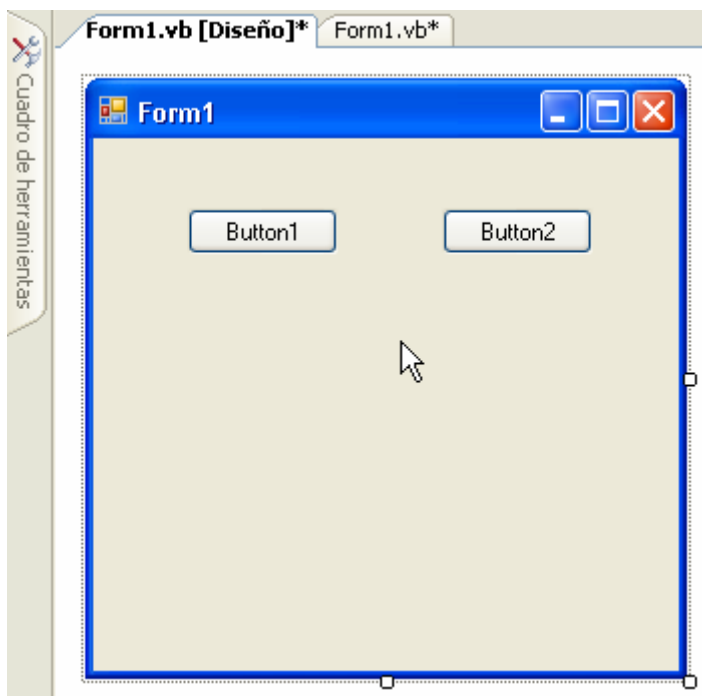


Figura 5.2: controles Button insertados en el formulario.

A continuación escribiremos el siguiente código en nuestra aplicación de ejemplo:

```
Public Class Form1
    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Button1.Click
        MessageBox.Show("Botón 1 pulsado")
    End Sub

    Private Sub Button2_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Button2.Click
        Button1_Click(sender, e)
    End Sub
End Class
```

Como podemos observar en este ejemplo, hemos querido lanzar un evento desde otro.

Ahora bien, no sólo podemos usar o lanzar los eventos generados por Visual Basic 2005, también podemos generar nuestros propios eventos para que los utilicemos dentro de nuestras aplicaciones. Esto es justamente lo que haremos a continuación.

Sin embargo, también podemos usar un evento en varios controles, esto se haría indicando el manejador del evento que queremos utilizar indicándole los controles que lo utilizará. Siguiendo con el ejemplo anterior, modificaremos el código de la siguiente manera:

```
Public Class Form1
    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Button1.Click, Button2.Click
        MessageBox.Show("Botón 1 pulsado")
    End Sub
End Class
```

Como podemos observar, al final de la palabra reservada **Handles** del evento *Button1_Click*, hemos añadido los eventos que queremos que formen parte de este código.

Por otro lado, lo que podemos hacer también, es generar nuestros propios eventos los cuales irán casi siempre asociados a nuestras propias clases. Esto se logra utilizando las palabras reservadas **Event** y **RaiseEvent**.

Adicionalmente, también podemos utilizar los comandos **AddHandler** y **RemoveHandler** para activar y desactivar el evento.

El siguiente ejemplo, nos facilitará la comprensión del uso de eventos personalizados. Para ello, escribiremos el siguiente código como clase de nuestra aplicación:

```
Public Class miClase
    Public Event MiEvento()

    Private _Valor As Integer = 0

    Property Valor() As Integer
        Get
            Return _Valor
        End Get
        Set(ByVal value As Integer)
            _Valor = value
        End Set
    End Property

    Public Sub MiMetodo()
        MessageBox.Show(Valor)
        RaiseEvent MiEvento()
    End Sub

End Class
```

Para ejecutar esta clase, escribiremos en nuestro formulario el siguiente 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 miClase
        MiClase.Valor = 5
        AddHandler MiClase.MiEvento, AddressOf TrataEvento
        MiClase.MiMetodo()
    End Sub

    Public Sub TrataEvento()
        MessageBox.Show("Evento lanzado")
    End Sub

End Class
```

Si queremos en un momento dado forzar el no uso del evento, podemos utilizar la instrucción **RemoveHandler**.

Un ejemplo adicional de la ejecución de la clase anterior, dónde forzamos en primer lugar el no uso del evento para forzar posteriormente su uso, sería el que se detalla a continuación:

```
Public Class Form1
    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button1.Click
        Dim MiClase As New miClase
        MiClase.Valor = 5
        RemoveHandler MiClase.MiEvento, AddressOf TrataEvento
        MiClase.MiMetodo()
        AddHandler MiClase.MiEvento, AddressOf TrataEvento
        MiClase.MiMetodo()
    End Sub
```

```
Public Sub TrataEvento()  
    MessageBox.Show("Evento lanzado")  
End Sub  
End Class
```

5.4.- Multihebras o Multithreading

Un aspecto a tener en cuenta en nuestras aplicaciones es el concepto de hebras múltiples, también conocido como **Multithreading**. Quizás también lo haya escuchado alguna vez como hilos, y es que en sí, este último concepto aclara aún más de que se trata.

Podemos ejecutar una aplicación y querer por ejemplo, lanzar varias ejecuciones paralelas y recoger sus eventos para saber cuando han finalizado todas ellas. Por lo general, nuestra aplicación se va a comportar siempre como una aplicación de tipo lineal, es decir, entrará por un sitio, irá recorriendo el árbol lógico de la aplicación, y finalizará en un punto determinado. Sin embargo, en muchas ocasiones, podemos querer lanzar diferentes procesos que se lancen simultáneamente para recoger los resultados de toda la ejecución de estos procesos y muestre el resultado final en pantalla.

Para utilizar esta técnica, utilizaremos en .NET el nombre de espacio **System.Threading**. Dentro de este nombre de espacio, encontraremos diferentes clases de las cuales, la que más nos interesa es la clase **Thread**.

Un ejemplo práctico del uso de este nombre de espacio y de la ejecución de una aplicación multihilo sería el siguiente:

```
Imports System.Threading  
  
Public Class Form1  
  
    Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As  
System.EventArgs) Handles MyBase.Load  
        Dim MiHilo1 As New Thread(AddressOf Calculo_1)  
        Dim MiHilo2 As New Thread(AddressOf Calculo_2)  
        MiHilo1.Start()  
        MiHilo2.Start()  
    End Sub  
  
    Public Sub Calculo_1()  
        Dim res As Long  
        For I As Integer = 0 To 100000  
            res += (IIf(I < 101, I, 0))  
        Next  
        MessageBox.Show("Calculo_1 " & res)  
    End Sub  
  
    Public Sub Calculo_2()  
        Dim res As Long  
        For I As Integer = 0 To 500  
            res += (IIf(I < 51, I, 0))  
        Next  
        MessageBox.Show("Calculo_2 " & res)  
    End Sub  
  
End Class
```

Este simple ejemplo, nos enseña y demuestra que aunque lancemos varias hebras o hilos de un determinado método en ejecución, estos funcionan y se ejecutan paralelamente haciendo en este caso, que el segundo hilo lanzado termine antes que el primero.

De todos los modos, lo más habitual es utilizar la gestión de hilos en nuestras aplicaciones con los eventos que hemos visto en el apartado anterior. Esto nos permite hacer aplicaciones mucho más robustas y fiables.

5.5.- Delegados

Un delegado es un tipo de referencia asociada a un método compartido, o una instancia de un método de un objeto. Para declarar delegados, debemos utilizar la palabra reservada **Delegate**.

A la hora de trabajar con objetos, la forma más habitual es la de declarar un objeto como una clase y acceder a los métodos de esa clase. Ahora bien, podemos encontrarnos con situaciones muy singulares como por ejemplo la necesidad de que la clase acceda a un método del programa principal. En este caso, el uso de delegados es obligado porque de no ser así, no podremos utilizar ese método.

Esto es lo que veremos a continuación con un ejemplo práctico. En primer lugar, escribiremos el código correspondiente a la clase de la aplicación de ejemplo, del uso de delegados y que es el siguiente:

```
Public Class miClase
    Public Sub MiMetodo(ByVal strMsg As String)
        strMsg += Now.Date
        MessageBox.Show(strMsg)
    End Sub
End Class
```

Para consumir la clase y en este caso el delegado que permita ejecutar una parte de la clase, escribiremos el siguiente código:

```
Public Class Form1
    Delegate Sub MiDelegado(ByVal strMensaje As String)

    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Button1.Click
        Dim Clase As New miClase
        Dim Delegado As MiDelegado
        Delegado = AddressOf Clase.MiMetodo
        Delegado.Invoke("Hoy es día: ")
    End Sub
End Class
```

5.6.- Herencia

La palabra reservada utilizada en Visual Basic 2005 para trabajar con la herencia de clases que pertenece a la programación orientada a objetos, es **Inherits**.

La herencia es a grandes rasgos, la relación en la cuál una clase derivada, deriva de otra clase base. Dicho de una manera más llana aún, es la relación de una clase que se basa en otra ya existente. De este modo, la nueva clase derivada, heredará todas las propiedades y métodos de la clase base de la cuál hereda.

Para entender esto de una forma más clara, utilizaremos el siguiente ejemplo práctico:

```
Public Class Form1
    Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
        Dim MiClase As New Clase_Derivada
        MiClase.strValor = "Ejemplo de herencia"
        MessageBox.Show(MiClase.strValor)
    End Sub
End Class

Public Class Clase_Base
    Public strValor As String
End Class
```



```
Public Class Clase_Derivada  
    Inherits Clase_Base  
End Class
```

Nuestro ejemplo en ejecución, es el que se muestra en la figura 5.3.

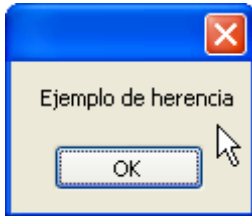


Figura 5.3: ejemplo de ejecución del uso de clases heredadas.

Por último, comentar que una clase, puede heredar de varias al mismo tiempo. Esto es lo que se denomina herencia múltiple.

CAPÍTULO 6

VISUAL BASIC 2005, EL ENTORNO

ESTE CAPÍTULO NOS PERMITIRÁ INTRODUCIRNOS EN EL ENTORNO DE DESARROLLO DE VISUAL BASIC 2005 EXPRESS EDITION.

Ya hemos visto las pinceladas generales del lenguaje Visual Basic 2005. Ahora lo que debemos hacer es familiarizarnos con el entorno de desarrollo, que es justamente lo que haremos en este capítulo.

6.1.- Visión general del entorno

Con *Visual Basic 2005 Express Edition* podemos desarrollar aplicaciones de forma mucho más rápida y eficiente que sin un entorno de desarrollo de estas características.

Con este entorno de desarrollo, podemos crear aplicaciones para Windows, biblioteca de clases, y aplicaciones de consola. El único *pero* es que no podemos desarrollar aplicaciones Web, Servicios Web, Servicios Windows y otro tipo de aplicaciones. Para este tipo de aplicaciones, existen otros productos de la familia Express Edition o de Visual Studio 2005.

En este capítulo nos centraremos por ese motivo, en la creación de aplicaciones Windows principalmente, el conocimiento general del entorno de desarrollo, y posteriormente, el conocimiento avanzado del entorno para sacar el máximo provecho a la programación.

6.2.- Creando una nueva aplicación

¿Cómo crear una nueva aplicación Windows o cualquier otro tipo de proyecto en *Visual Basic 2005 Express Edition*?

Inicie el entorno de desarrollo y seleccione del menú la opción **Archivo > Nuevo > Proyecto** como se muestra en la figura 6.1.

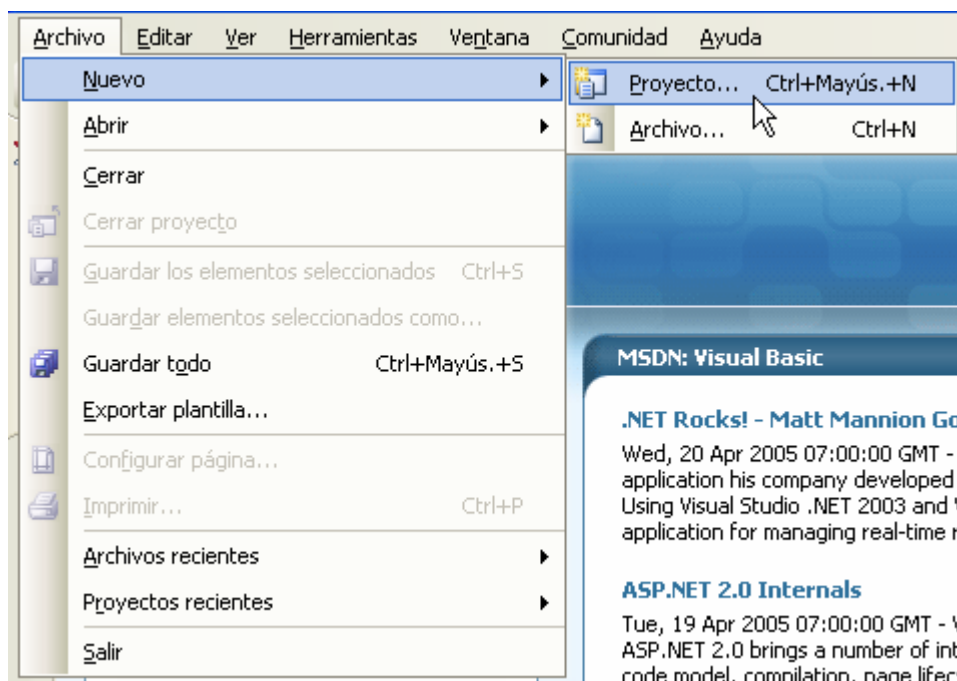


Figura 6.1: opción para crear un nuevo proyecto en Visual Basic 2005 Express Edition.

Aparecerá una nueva ventana como la que se muestra en la figura 6.2., en la cuál podremos seleccionar el tipo de proyecto que queremos crear. En nuestro caso, crearemos una nueva **Aplicación para Windows**.

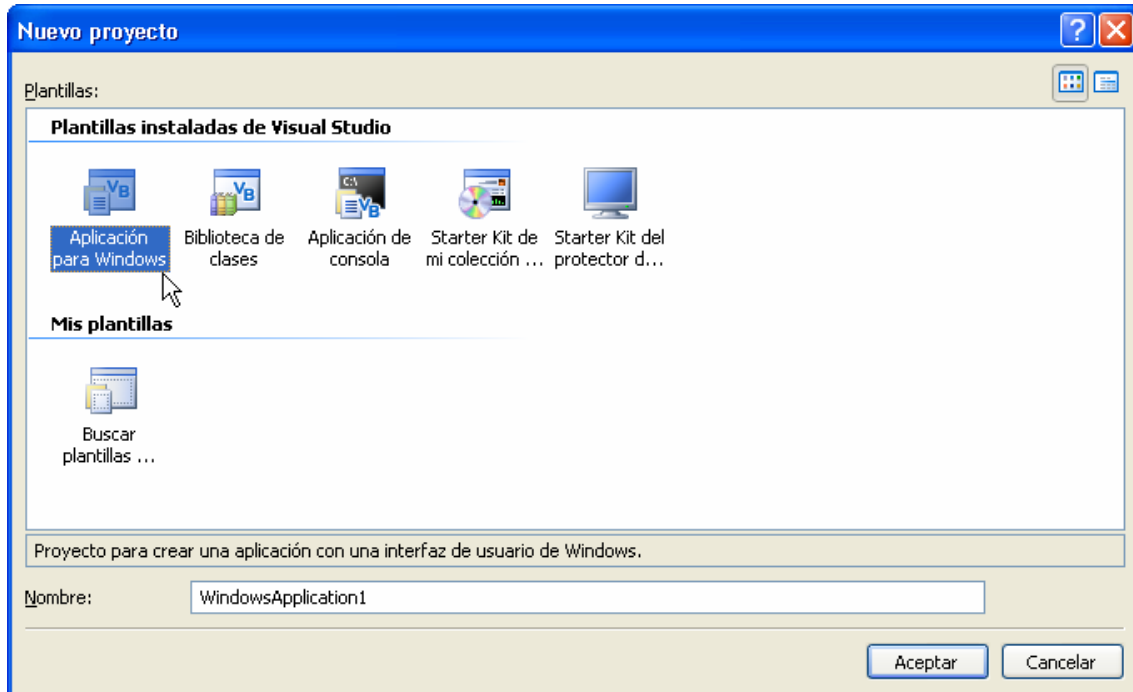


Figura 6.2: plantilla de Aplicación para Windows seleccionada para una nueva aplicación.

Seleccione esta opción y pulse el botón **Aceptar**. Con esto habremos creado la estructura principal de un nuevo proyecto vacío de tipo aplicación para Windows.

6.3.- El Cuadro de herramientas

El **Cuadro de herramientas** está situado en la parte izquierda del entorno de desarrollo rápido, y engloba dentro de sí, todos los controles y componentes que tenemos activados en el entorno para poder utilizarlos, pero también podemos añadir los controles y componentes que creamos nosotros y los que podamos cargar en .NET y que hayan sido desarrollados por terceros.

En la figura 6.3., podemos observar el **Cuadro de herramientas** del entorno.

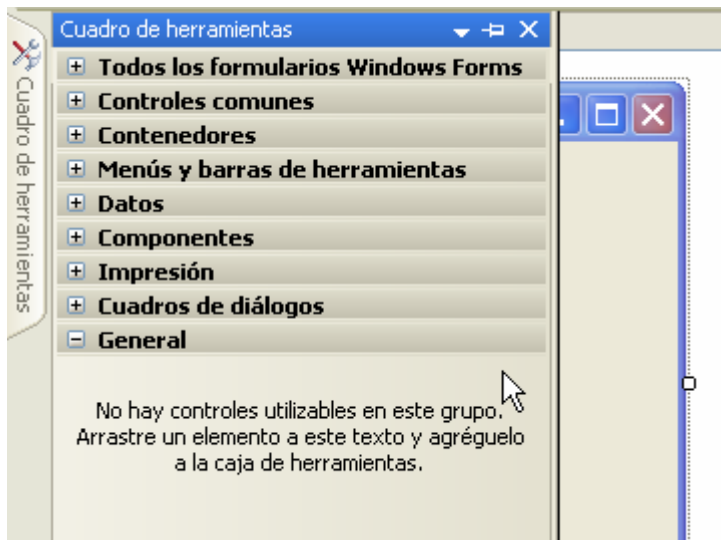


Figura 6.3: Cuadro de herramientas del entorno de desarrollo.

Para insertar un control o un componente dentro de un formulario Windows, deberemos seleccionar el objeto que queremos insertar y hacer doble clic sobre el control, o bien hacer clic en el control y arrastrarlo y soltarlo sobre el formulario Windows, o bien hacer clic en el control y luego hacer clic nuevamente en el formulario y dimensionar un espacio que será el que ocupe el control insertado.

6.4.- El Explorador de soluciones

El **Explorador de soluciones** está situado en la parte derecha del entorno de desarrollo rápido, y en él tenemos todos los proyectos de la solución, y los ficheros y recursos de cada proyecto.

En la figura 6.4., podemos observar la ventana del **Explorador de soluciones**.

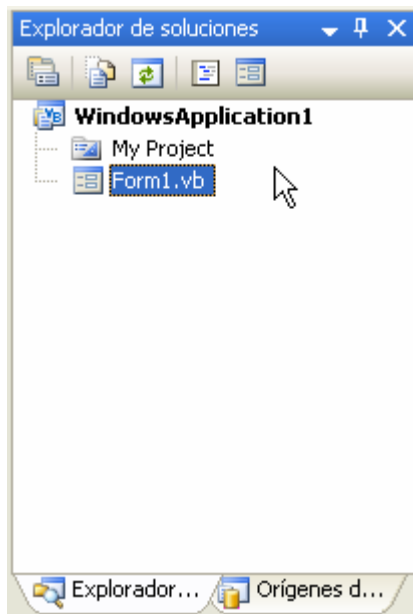


Figura 6.4: Explorador de soluciones del entorno de desarrollo.

Si observamos esta ventana, veremos que en la parte inferior hay dos solapas. La segunda tiene que ver con los **Orígenes de datos**.

6.5.- Los Orígenes de datos

Los **Orígenes de datos** nos permiten añadir conexiones en el entorno con fuentes de datos para agregarlas a un formulario u otro objeto de forma rápida y sencilla con una acción simple de arrastrar y soltar.

Adicionalmente, podemos acceder a las fuentes de datos para modificar o gestionar esas fuentes de datos, ya sean tablas, vistas, procedimientos almacenados, etc.

La ventana de **Orígenes de datos** del entorno, tendrá por lo tanto un aspecto similar al que se presenta en la figura 6.5.

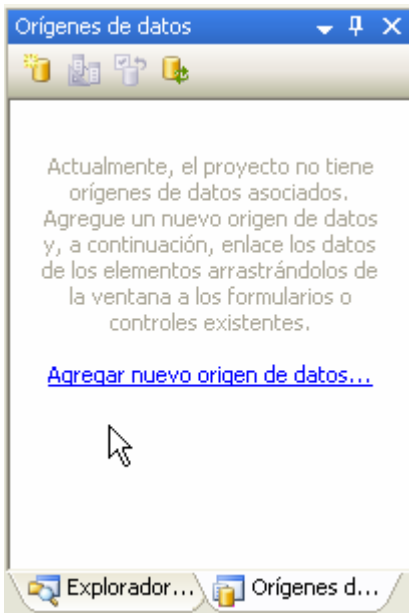


Figura 6.5: Orígenes de datos del entorno de desarrollo.



Recordatorio:

En el caso de .NET y en cuanto a los productos Microsoft, existen dos versiones de Microsoft SQL Server 2005. Hay una versión comercial de pago y una versión Express Edition que es una versión gratuita del producto que se integra perfectamente con el entorno de desarrollo.

6.6.- Ventana de propiedades

La ventana de **Propiedades** nos facilita el acceso a las propiedades del formulario y controles y componentes de la aplicación. En sí, nos posibilita acceder a las propiedades de todos los objetos utilizados dentro del proyecto.

En la figura 6.6., podemos observar la ventana de **Propiedades**:

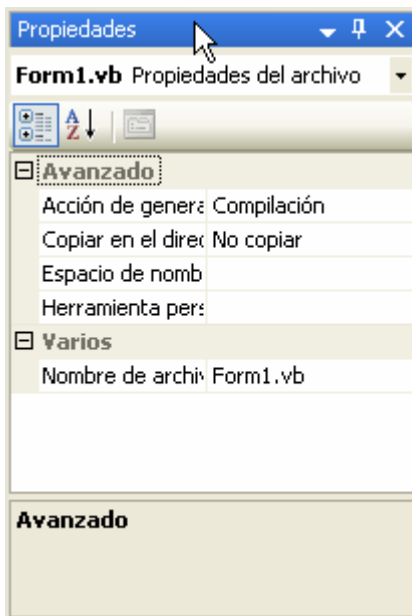


Figura 6.6: Propiedades en el entorno de desarrollo.

6.7.- Agregar elementos al proyecto

Para agregar elementos al proyecto, tenemos tres alternativas. La primera de ellas, es hacer esta acción desde la barra de botones del entorno de desarrollo. La segunda posibilidad que tenemos es hacerlo desde la ventana del **Explorador de soluciones**.

Aquí veremos estas dos alternativas para aprender a movernos en el entorno de desarrollo.

La primera de estas alternativas para agregar un elemento al proyecto, es como hemos comentado, utilizando la barra de botones del entorno de desarrollo. Para ello, haremos clic sobre el botón de **Agregar elemento**, y dentro de esta opción, en el botón **Agregar nuevo elemento...** como se muestra en la figura 6.7.

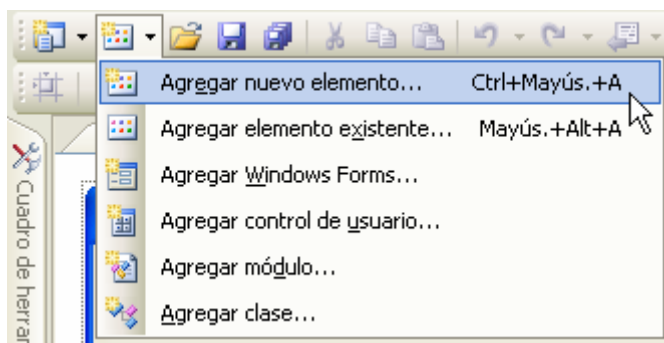


Figura 6.7: opción de Agregar nuevo elemento... dentro del entorno de desarrollo.

Como podemos observar, también podemos pulsar las teclas rápidas **Ctrl+Mayús.+A** para acceder a esta misma acción.

Adicionalmente, observamos en la figura 6.7. que podemos también añadir otros tipos de elementos al proyecto de forma directa, aunque lo habitual es abrir la ventana para agregar nuevos elementos al proyecto.

La segunda de las opciones que tenemos para agregar un elemento al proyecto, es hacerlo desde la ventana del **Explorador de soluciones**. Para lograr esto, deberemos acudir a esta ventana y hacer clic con el botón derecho del ratón sobre el proyecto y seleccionar la opción **Agregar > Nuevo elemento...** del menú emergente, tal y como se muestra en la figura 6.8.

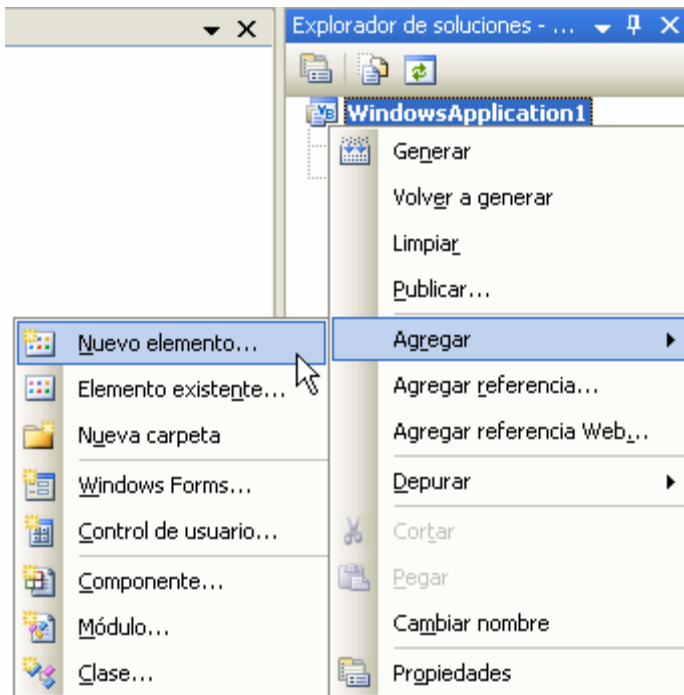


Figura 6.8: opción de agregar Nuevo elemento... dentro del entorno de desarrollo.

Cuando seleccionamos esta opción de agregar un nuevo elemento al entorno, aparecerá una ventana como la que se muestra en la figura 6.9.

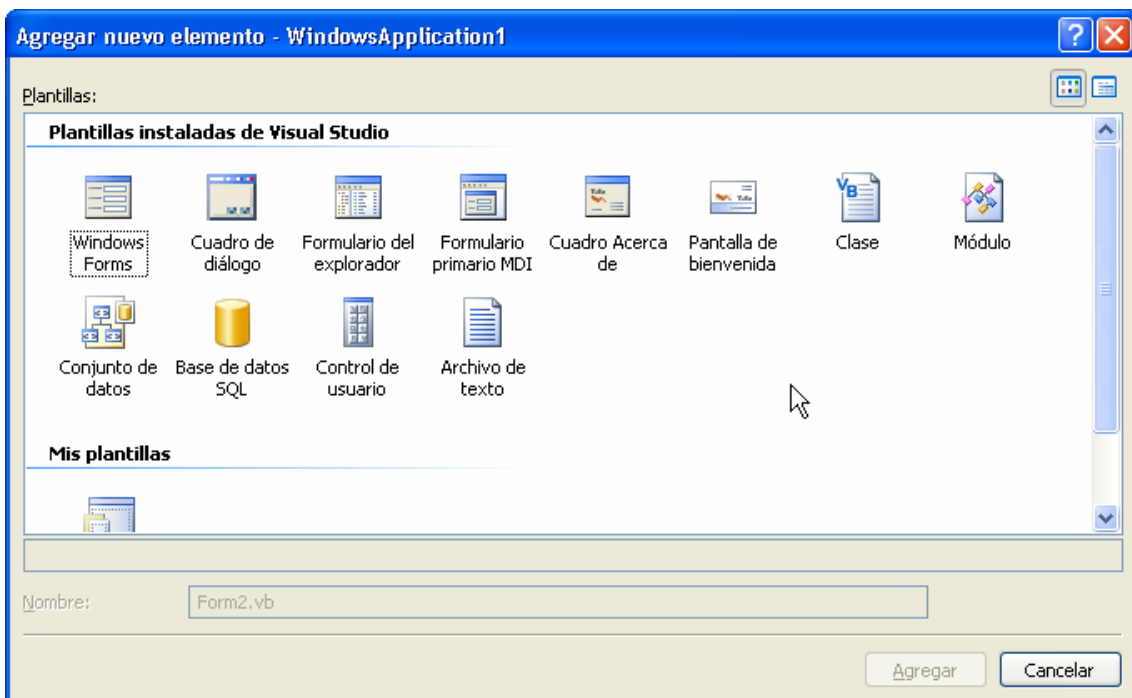


Figura 6.9: ventana para Agregar nuevo elemento en el entorno de desarrollo.

Desde esta ventana, elegiremos la plantilla o elemento que deseamos añadir al proyecto, algo que veremos a continuación de forma más detallada.

6.8.- Agregando elementos al proyecto

Cuando seleccionamos la opción de agregar un nuevo elemento al proyecto, se abre una ventana con todas las plantillas disponibles, de las cuales, aquí veremos las más frecuentes.

Las plantillas que podemos utilizar en el entorno son la de **Windows Forms**, **Cuadro de diálogo**, **Formulario del explorador**, **Formulario primario MDI**, **Cuadro Acerca de**, **Pantalla de bienvenida**, **Clase**, **Módulo**, **Conjunto de datos**, **Base de datos SQL**, **Control de usuario** y **Archivo de texto**.

6.8.1.- Windows Forms

Poco podemos añadir ya sobre esta plantilla, pues es la plantilla utilizada por defecto en cualquier tipo de proyecto que iniciamos o con cualquier elemento que añadimos al proyecto. Se trata por lo tanto de un formulario vacío, sin ningún control, componente, u objeto insertado en él, que nos permita añadir los controles o componentes que deseemos.

En la figura 6.10., podemos observar un formulario Windows cargado por defecto.

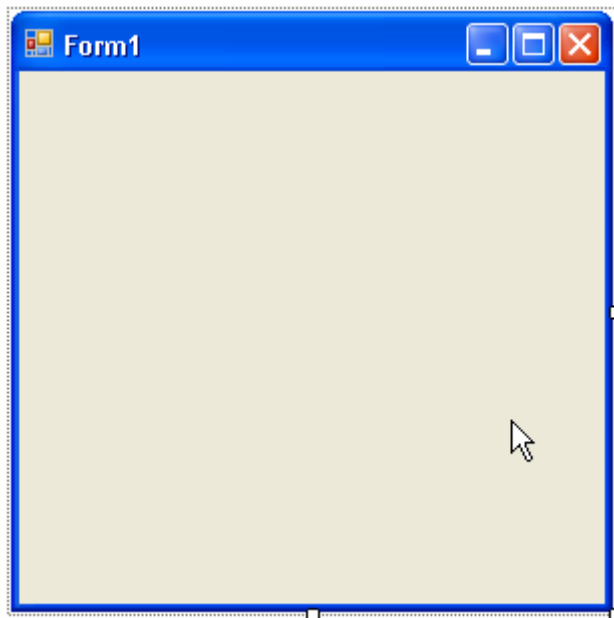


Figura 6.10: formulario Windows por defecto creado en el entorno.

6.8.2.- Cuadro de diálogo

Al seleccionar esta plantilla, el entorno de desarrollo nos inserta un formulario Windows con un control **TabletLayoutPanel**, y dentro de este control, dos controles **Button**. Se trata de la estructura típica de un formulario Windows de este tipo, pero sin programación de ningún tipo.

En la figura 6.11., podemos observar este formulario o plantilla de formulario, insertado en el entorno de desarrollo.

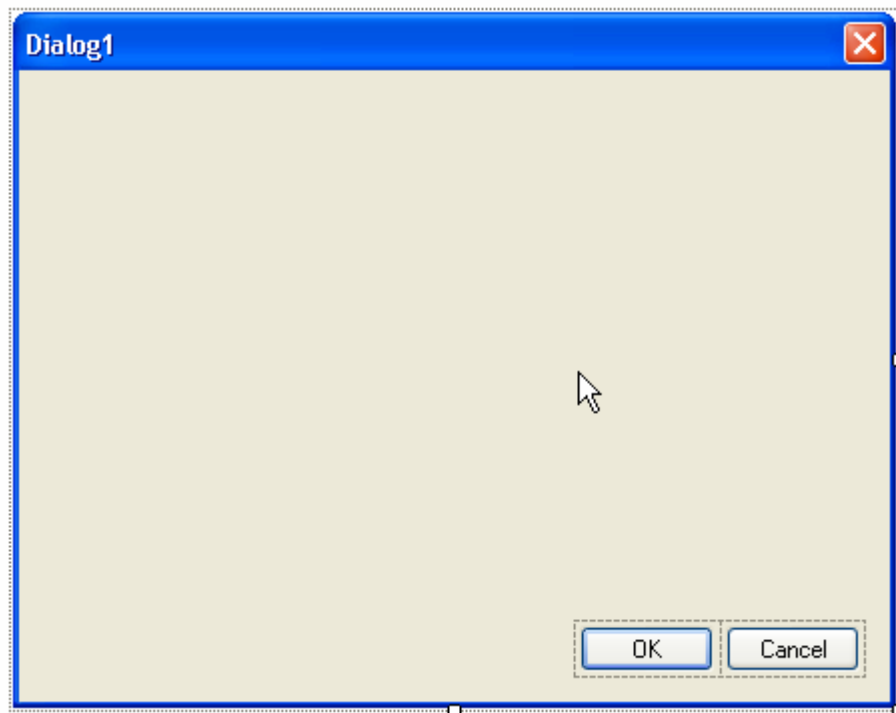


Figura 6.11: Cuadro de diálogo creado en el entorno.

6.8.3.- Formulario del explorador

Al seleccionar esta plantilla, el entorno de desarrollo nos presenta un formulario con todos los objetos necesarios insertados en él, para disponer de un formulario de tipo explorador.

En la figura 6.12., podemos ver la captura de esta plantilla dentro del entorno de desarrollo.

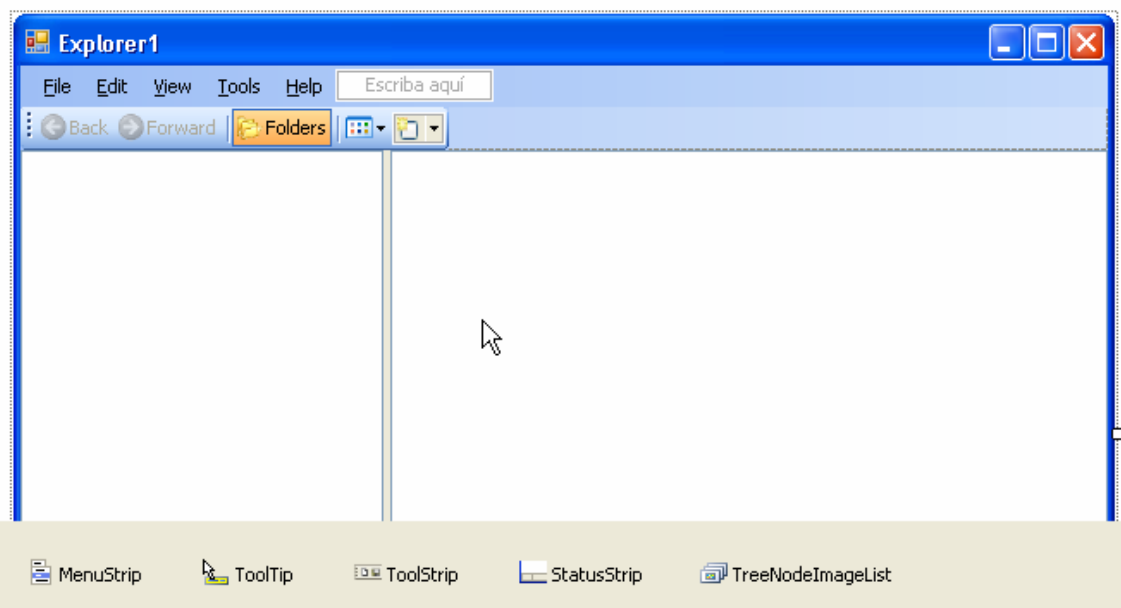


Figura 6.12: Formulario del explorador creado en el entorno.

Aún y así, esta plantilla no presenta el código completo de este tipo de formulario, el cuál es necesario programar para realizar la base principal de nuestra aplicación, pero sirve de gran ayuda para ahorrarnos gran parte del trabajo.

6.8.4.- Formulario primario MDI

Esta plantilla nos inserta en el entorno, un formulario MDI con sus menús, botones, etc. Se trata de la base de un formulario MDI, dónde lógicamente nos queda parte de una aplicación MDI por desarrollar, pero la base nos ahorra una gran cantidad de trabajo en el entorno.

En la figura 6.13., podemos observar este formulario en el entorno de desarrollo de Visual Basic 2005.

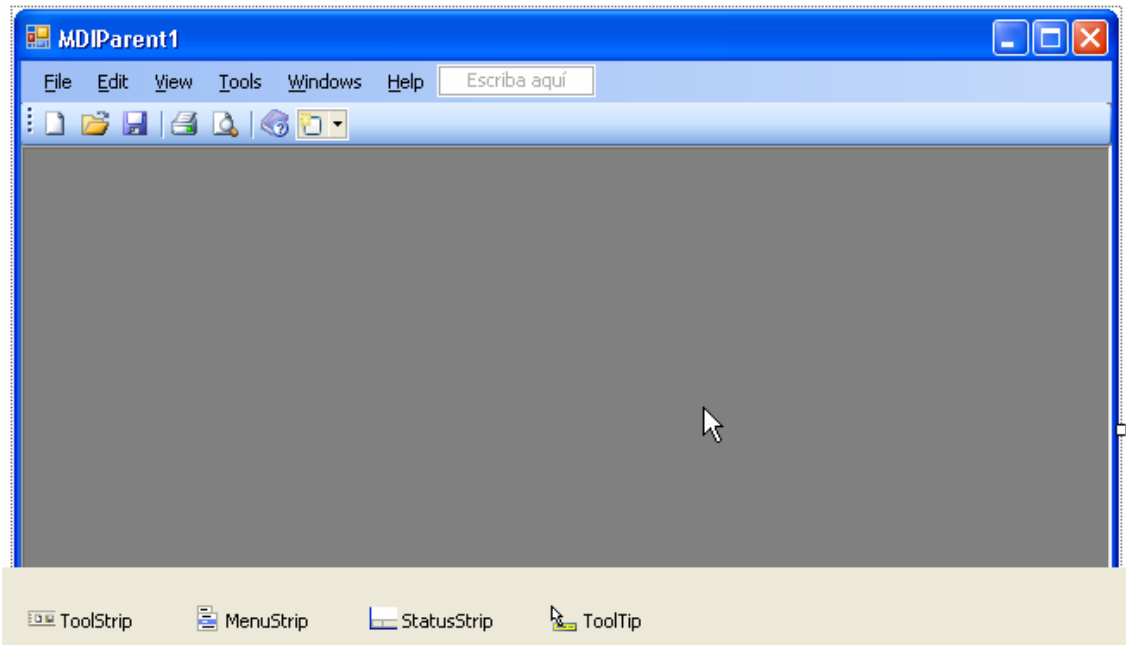


Figura 6.13: Formulario primario MDI creado en el entorno.

6.8.5.- Cuadro Acerca de

Esta plantilla nos inserta dentro del entorno, un formulario de tipo *Acerca de*. En casi todas las aplicaciones Windows, es habitual encontrarnos con formularios que nos cuentan cosas o aspectos acerca del programa. Este formulario es la base de este tipo de ventanas.

En la figura 6.14., podemos observar esta ventana insertada en el entorno de desarrollo.

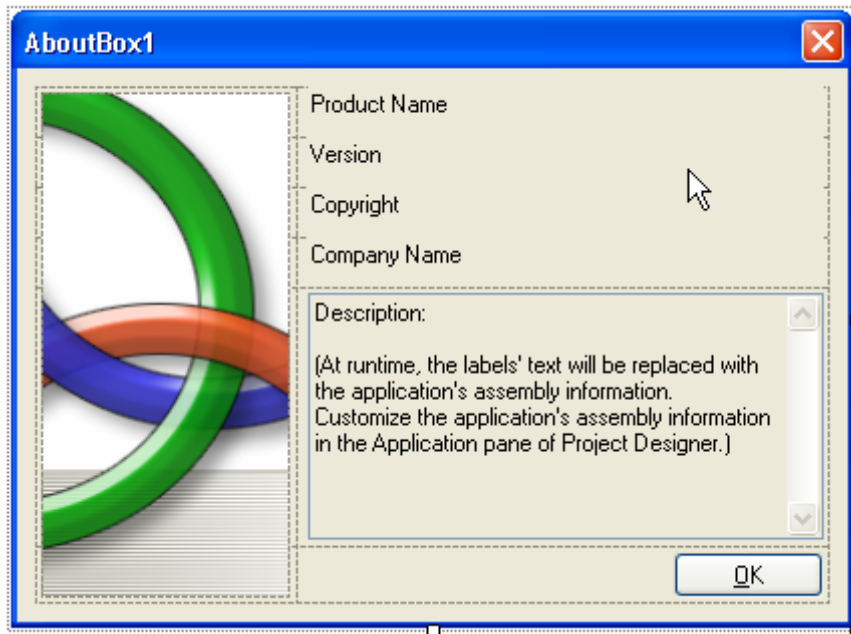


Figura 6.14: Cuadro Acerca de creado en el entorno.

6.8.6.- Pantalla de bienvenida

Al iniciar una aplicación Windows, es también muy habitual, encontrarnos con una pantalla de inicio o presentación de la aplicación. Esta ventana es la que se puede conseguir introducir en nuestro proyecto Windows con la *Plantilla de bienvenida*.

En la figura 6.15., podemos observar esta plantilla dentro del entorno de desarrollo, insertada en nuestro proyecto.



Figura 6.15: Pantalla de bienvenida creada en el entorno.

Sobre esta base, podremos modificar esta plantilla a nuestro antojo y utilizarla como presentación de nuestra aplicación al iniciar ésta.

6.8.7.- Otras plantillas

Hasta ahora, hemos visto las plantillas generales que nos ofrece el entorno de *Visual Basic 2005 Express Edition*, para crear formularios Windows base con las partes más utilizadas y generales para nuestras aplicaciones Windows.

Sin embargo, hay otras plantillas que enumeraremos a continuación y que no son formularios Windows propiamente dichos, pero sí forman o pueden formar parte de cualquier aplicación Windows en este caso.

Por un lado, podemos agregar a nuestro proyecto **Clases** y **Módulos**. En este manual, ya hemos hablado de clases, pero no de módulos.

Los módulos, son clases con características especiales. Por ejemplo, no puede ser derivado de otra clase ni utilizarlo como base para la definición de una nueva clase. La visibilidad se extiende por todo el módulo en el que está definido, y por tanto, sus métodos son compartidos. El punto de entrada de un módulo, es normalmente el representado por la palabra clave **Main**. El uso de los módulos por lo tanto, se ve muchas veces recomendado según los objetivos de nuestra aplicación.

Por otro lado, tenemos dentro de las plantillas, la posibilidad de trabajar con fuentes de datos añadiendo un **Conjunto de datos** o una **Base de datos SQL**.

Finalmente, hay otras dos plantillas muy diferentes entre sí. Una de ellas se trata de una plantilla para añadir un **Archivo de texto** al entorno, que podemos utilizar como información adicional o con algún objetivo determinado. La otra plantilla es la base para crear nuestros propios **Controles de usuario**. De esta manera, podemos crear la base de un control personalizado para utilizar en nuestras aplicaciones o proyectos, ya que podremos reutilizarlo como deseemos.

CAPÍTULO 7

VISUAL BASIC 2005, TRABAJANDO CON EL ENTORNO

ESTE CAPÍTULO NOS MOSTRARÁ COMO TRABAJAR CON NUESTRAS APLICACIONES DENTRO DEL ENTORNO DE DESARROLLO DE VISUAL BASIC 2005 EXPRESS EDITION.

En este capítulo, aprenderemos a utilizar el entorno de trabajo de *Visual Basic 2005 Express Edition* en nuestras aplicaciones Windows.

Hasta ahora, hemos visto como utilizar el entorno de desarrollo de forma general, pero hay que ponerse en faena cuando nos ponemos delante de una aplicación, algo que es justamente lo que vamos a hacer en este capítulo.

7.1.- Código vs Diseñador

Dentro del entorno de desarrollo, podemos trabajar con el código o con el diseñador del objeto si admite objetos. Por eso, podemos cambiar entre diseñador y código, según nuestras necesidades.

Dentro del **Explorador de soluciones**, podemos hacer doble clic sobre un elemento del proyecto y así, podremos abrir por defecto el diseñador del objeto si lo tiene. Si se abre el código directamente, es porque el objeto que abrimos no tiene diseñador asociado. En general y para entender esto de una forma mejor, un formulario Windows tiene una parte de diseño que corresponde al propio formulario, y una parte de código que corresponde con el código asociado al formulario.

Si queremos abrir el código de un objeto, bastará con pulsar el botón derecho del ratón sobre el objeto dentro del **Explorador de soluciones**, y seleccionar la opción **Ver código** como se muestra en la figura 7.1.

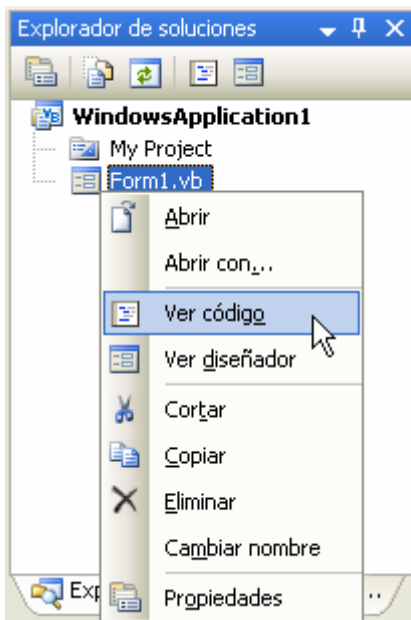


Figura 7.1: opción para ver el código de un objeto desde el Explorador de soluciones.

Otra forma de hacer esto mismo, es utilizando la barra de botones del **Explorador de soluciones**.

El entorno tiene un botón que nos permite ver el código fuente asociado a un objeto, tal y como se muestra en la figura 7.2.

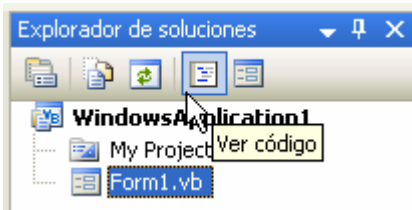


Figura 7.2: opción de la barra de botones del Explorador de soluciones, para ver el código de un objeto.

Asimismo, si queremos ver el objeto en el diseñador del entorno, deberemos hacer clic sobre el correspondiente botón dentro del **Explorador de soluciones**, tal y como se muestra en la figura 7.3.

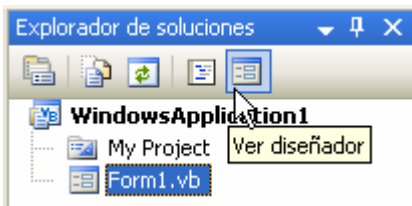


Figura 7.3: opción de la barra de botones del Explorador de soluciones, para ver el diseñador de un objeto.

Aún y con esto, tenemos otra alternativa de acceder al código o diseñador de un determinado objeto del proyecto. Haciendo clic sobre el objeto elegido en el **Explorador de soluciones**, acudiremos al menú **Ver > Código** para acceder al código de este objeto, tal y como se muestra en la figura 7.4.

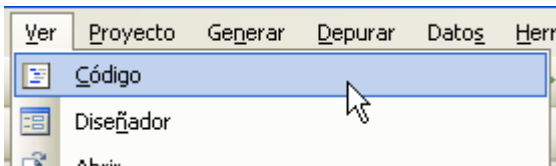


Figura 7.4: opción del menú para ver el código de un objeto.

De igual manera haremos, para ver el diseñador de un objeto en el menú del entorno. En esta ocasión, seleccionaremos la opción del menú **Ver > Diseñador**, tal y como se muestra en la figura 7.5.

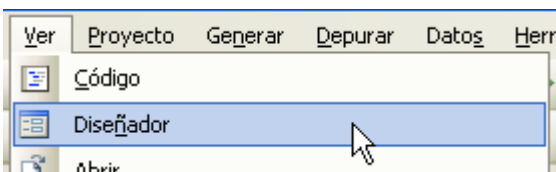


Figura 7.5: opción del menú para ver el diseñador de un objeto.

Para finalizar, también podemos acceder al código o diseñador de un objeto de otras maneras. Si nos encontramos en el código de nuestro objeto, podemos hacer clic con el botón derecho del ratón sobre cualquier superficie del código, y ahí seleccionaremos la opción **Ver diseñador** del menú contextual que nos aparecerá, como nos muestra la figura 7.6.

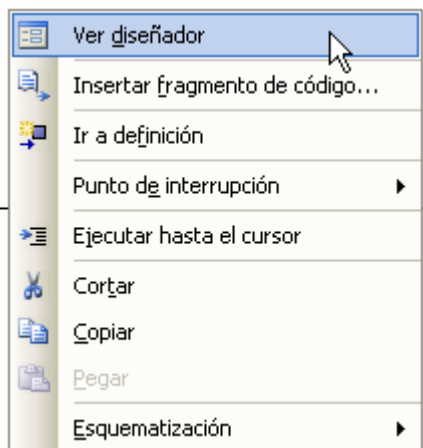


Figura 7.6: opción del menú contextual sobre el código, para ver el diseñador de un objeto.

De igual forma, si nos encontramos en el diseñador del objeto, un formulario Windows por ejemplo, podremos hacer clic con el botón derecho del ratón sobre cualquier superficie del diseñador, y elegir la opción **Ver código** del menú emergente que aparece, tal y como se muestra en la figura 7.7.

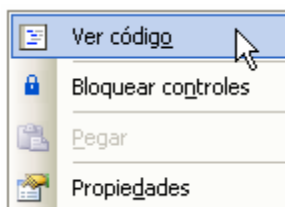


Figura 7.7: opción del menú contextual sobre el diseñador, para ver el código de un objeto.

7.2.- Ejecutando una aplicación

Para ejecutar una aplicación, bastará con pulsar la tecla **F5** en el entorno de desarrollo. La aplicación iniciará todos los procesos internos de compilación y lanzará la aplicación en pantalla como si hubiéramos ejecutado la aplicación de forma normal.

Otra alternativa que nos ofrece el entorno de desarrollo, es la de iniciar la depuración utilizando para ello la barra de botones como nos muestra la figura 7.8.



Figura 7.8: Iniciar depuración en la barra de botones del entorno de desarrollo.

La alternativa a pulsar la tecla **F5** del entorno, es seleccionar esta misma acción desde el menú del entorno. Para ello, acudiremos al menú y seleccionaremos la opción **Depurar > Iniciar depuración** como se muestra en la figura 7.9.

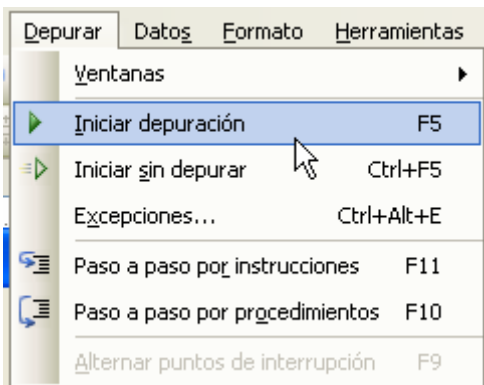


Figura 7.9: Iniciar depuración en el menú de opciones del entorno de desarrollo.

7.3.- Diferencias entre Iniciar depuración e Iniciar sin depurar

En la figura 7.9., podíamos observar como iniciar una aplicación depurándola dentro del entorno de desarrollo, pero si hemos observado bien esa figura, habremos visto otra opción dentro del menú, que se denomina **Iniciar sin depurar** y que es la opción que se muestra en la figura 7.10., y que puede ser accedida pulsando las teclas alternativas **Ctrl+F5**.

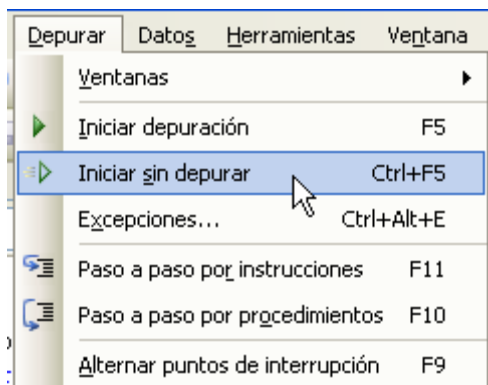


Figura 7.10: Iniciar sin depurar en el menú de opciones del entorno de desarrollo.

La diferencia entre estas dos opciones es muy clara y la entenderemos rápidamente. Por un lado, la iniciación con depuración que es la primera opción tratada, nos permite ejecutar la aplicación parándonos en los puntos de interrupción marcados por nosotros mismos como veremos a continuación, o bien, parándose el entorno directamente sobre aquellas líneas de código que realiza una operación prohibida o que genera una interrupción no tratada.

Por otro lado, el inicio sin depuración nos permite ejecutar la aplicación, ignorando los puntos de interrupción marcados por nosotros, si tenemos marcado alguno, e impidiendo que el entorno se detenga en el lugar del código dónde se genera una interrupción determinada.

Para comprobar esto, ejecutaremos el siguiente código:

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
    Dim I As Integer
    I = I / 0
End Sub
```

Pulsaremos en primer lugar, la tecla **F5**. De esta manera, iniciaremos el entorno de ejecución en modo depuración.

Indudablemente, hemos escrito un código que generará un error de ejecución y que se nos presentará en pantalla como se muestra en la figura 7.11.

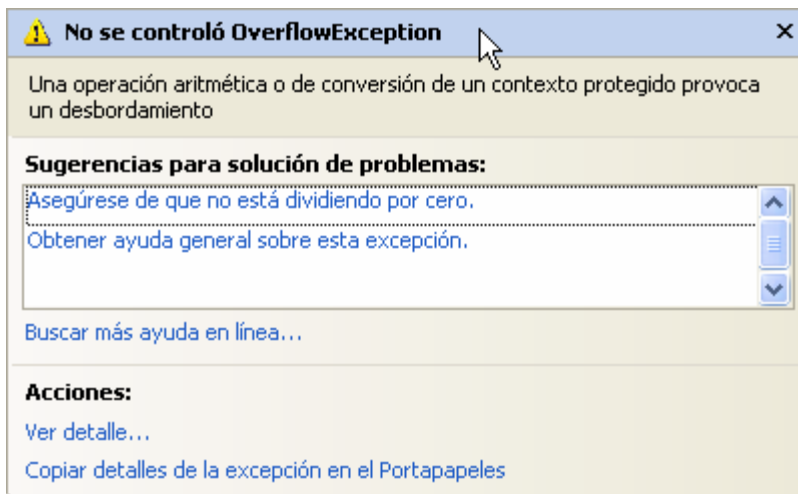


Figura 7.11: gestión de errores y excepciones en el entorno de depuración.

Como vemos, la ejecución del código de la aplicación encuentra un error y el sistema se detiene en la línea de código en la cuál se ha encontrado ese fallo.

Pero si pulsásemos la tecla **Ctrl+F5**, se iniciará la ejecución de la aplicación sin atender a estos modos de depuración, y en nuestro caso, nos encontraremos ante una pantalla como la que se muestra en la figura 7.12.

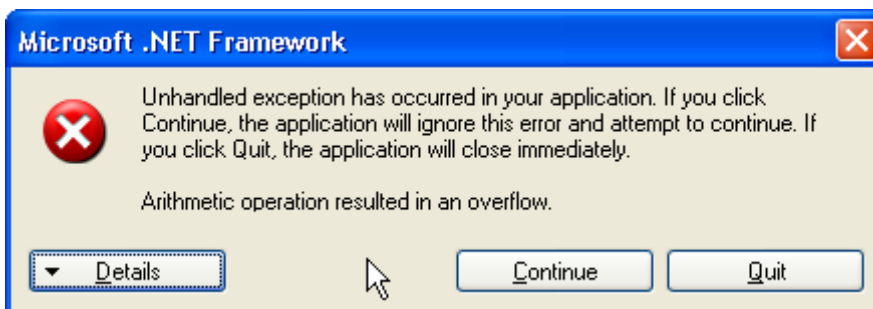


Figura 7.12: gestión de errores y excepciones del entorno de ejecución.

Ahora bien, hemos hablado de puntos de interrupción y aún no sabemos como gestionarlos y usarlos. Eso es justamente lo que haremos a continuación.

7.4.- Depurando una aplicación

La tarea más común en el desarrollo de cualquier aplicación Windows, es la depuración. Todas las aplicaciones deben ser depuradas y probadas antes de finalizarlas, y así podremos detectar errores, probar soluciones, y confiar en la resolución de los problemas y en el funcionamiento global de la aplicación, además de la detección de problemas, errores o fallos.

7.4.1.- Puntos de interrupción

Los puntos de interrupción son pequeñas señales que se ponen en el código, para que cuando el flujo lógico de la aplicación en ejecución pase por estos puntos, se detenga y nos permita depurar el código para seguir el flujo lógico de funcionamiento de nuestras aplicaciones.

Para añadir un punto de interrupción al código del entorno, deberemos pulsar la tecla **F9** sobre al línea o líneas de código sobre las cuales queremos marcar esos puntos de interrupción.

Una vez que hemos marcado estos puntos de interrupción, el entorno nos marca un punto rojo y una línea roja, la fila sobre la cuál hemos marcado esa interrupción.

7.4.2.- Deteniendo la depuración

Cuando iniciamos la aplicación con depuración, podemos detener esta acción desde la barra de botones o desde la barra de menús.

Para detener la depuración iniciada en el entorno desde la barra de botones de *Visual Basic 2005 Express Edition*, haremos clic en el correspondiente botón que aparece en la figura 7.13.

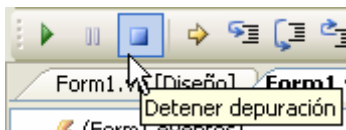


Figura 7.13: Detener depuración desde la barra de botones del entorno de ejecución.

Adicionalmente a esto, también podemos detener la depuración desde la barra de menús. Esto se logra desde el menú seleccionando la opción **Depurar > Detener depuración** como se muestra en la figura 7.14.

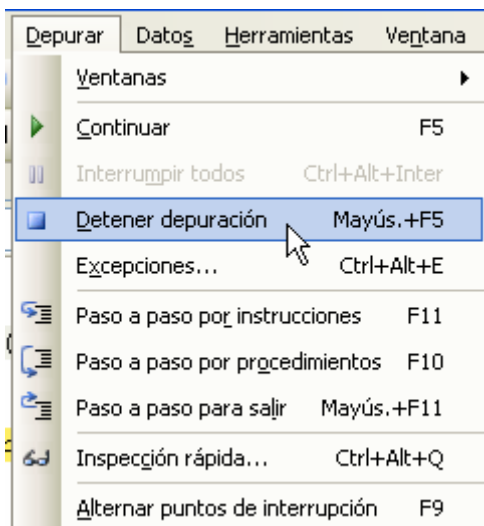


Figura 7.14: Detener depuración desde el menú del entorno de ejecución.

7.4.3.- Visión práctica de la depuración de un ejemplo

La mejor manera de aprender a depurar una aplicación, es trabajando con un ejemplo práctico que nos muestre como hacerlo.

En primer lugar, iniciaremos una aplicación Windows y escribiremos el siguiente código fuente:

```
Public Class Form1
    Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
        Dim I As Integer
        Dim strTexto As String = " "
        For I = 0 To 5
            strTexto &= I
        Next
        MiMetodo(strTexto)
        MessageBox.Show(strTexto)
    End Sub
End Class
```

```
End Sub

Private Sub MiMetodo(ByRef strTxt As String)
    Dim I As Integer
    For I = 0 To 5
        strTxt &= I
    Next
End Sub

End Class
```

A continuación seleccionaremos los puntos de interrupción de la aplicación que consideramos oportunos para aprender las acciones básicas de depuración de aplicaciones.

Para ello pulsaremos la tecla **F9** sobre varias líneas de código como se muestra en la figura 7.15.

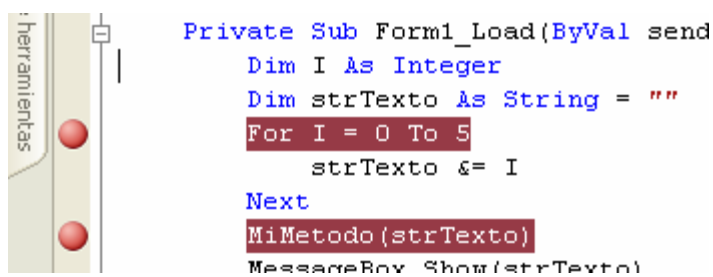


Figura 7.15: puntos de interrupción marcados en la aplicación de ejemplo.

Una vez hecho esto, pulsaremos la tecla **F5** que corresponde a la ejecución con depuración de la aplicación.

La primera interrupción de la aplicación se producirá en la instrucción:

```
For I = 0 To 5
```

Como vemos, hemos creado un punto de interrupción dentro de un bucle, por lo que para recorrerlo depurándolo, pulsaremos la tecla **F10** repetitivamente para pasar de una línea a otra.

También podemos hacer esta misma acción pulsando el botón **Paso a paso por procedimientos** como se muestra en la figura 7.16.

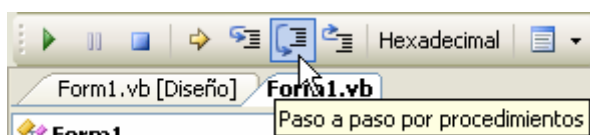


Figura 7.16: Paso a paso por procedimientos desde la barra de botones del entorno.

Como tenemos otro punto de interrupción después del bucle y puesto que estamos ejecutando la depuración dentro del bucle, si consideramos conveniente el buen funcionamiento de esta parte del código, podemos pulsar la tecla **F5** para que se ejecute la aplicación de ejemplo en su modo normal nuevamente hasta que encuentre otro punto de interrupción.

Eso es lo que haremos, encontrando así que el código se para en el siguiente código:

```
MiMetodo(strTexto)
```

Este es un caso especial de depuración, ya que este código está llamando a un método, por lo que si pulsamos la tecla **F10**, lo que estaremos haciendo es pasar por el método recogiendo el valor que nos devuelve, pero no depurándolo como queremos. Para entrar en el método y depurarlo adecuadamente, deberemos pulsar la tecla **F11** dentro de este punto de interrupción.

Una vez dentro del método, podremos pulsar la tecla **F10** nuevamente para depurar el método. También podemos saltar al método desde la barra de botones del entorno haciendo clic sobre el botón **Paso a paso por instrucciones** como se muestra en la figura 7.17.

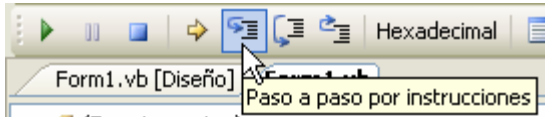


Figura 7.17: Paso a paso por instrucciones desde la barra de botones del entorno.

Cuando depuramos una aplicación con puntos de interrupción, podemos también añadir, quitar y modificar los puntos de interrupción. Otra de las posibilidades que nos ofrece *Visual Basic 2005 Express Edition* y Visual Studio 2005 en general, es la posibilidad de depurar y modificar el código en depuración y volver a ejecutar la aplicación reconociendo los cambios introducidos en la depuración, algo que veremos a continuación.

7.4.4.- Modificando el código en depuración

Otra de las alternativas que podemos llevar a cabo a la hora de depurar el código de nuestras aplicaciones, es la de modificar la línea y líneas de código que hay dentro de la aplicación cuando nos encontramos con un error.

Imaginemos el siguiente ejemplo sencillo para demostrar como realizar este tipo de acciones:

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
    Dim I As Integer = 5
    I = I / 0
    MessageBox.Show(I)
End Sub
```

Ejecutemos la aplicación con depuración pulsando la tecla **F5** de ejecución. La aplicación se detendrá en el punto de error de desbordamiento detectado tal y como se muestra en la figura 7.18.

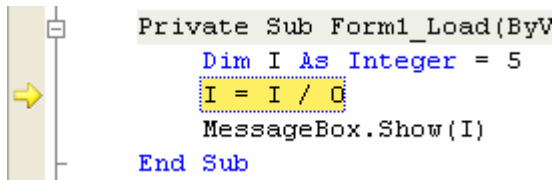


Figura 7.18: punto de interrupción marcado por el sistema ante una excepción.

Al mismo tiempo, *Visual Basic 2005 Express Edition* nos informará del error con una pantalla como la que se muestra en la figura 7.19.

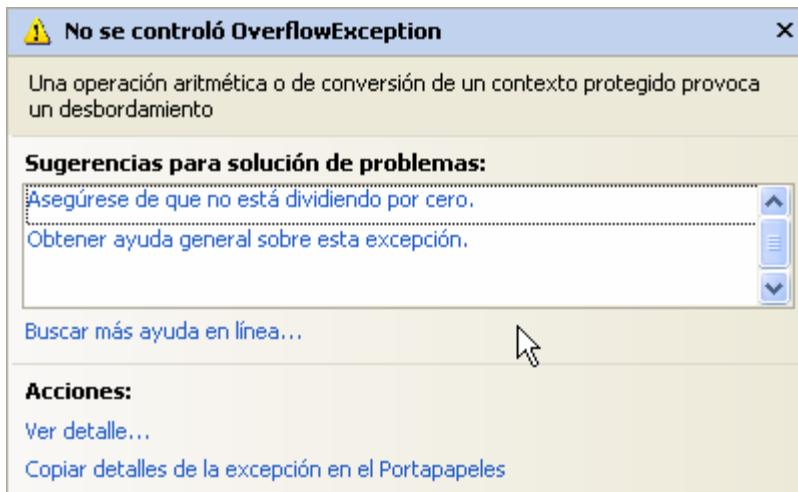


Figura 7.19: ventana de mensaje de depuración y consejo de Visual Basic 2005 Express Edition.

En esta ventana, el entorno nos informa del error detectado e incluso nos puede dar consejos útiles para resolver este error.

Si nos posicionamos en el código, podremos modificar éste para solventar el problema detectado como se muestra en la figura 7.20.

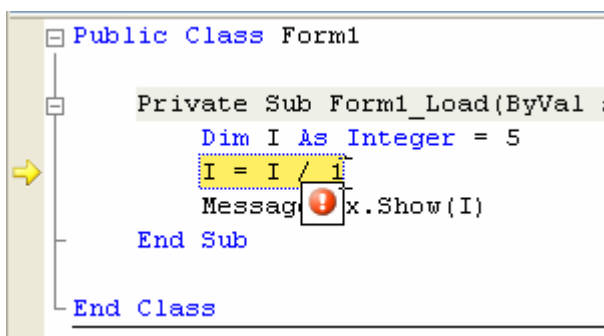


Figura 7.20: en tiempo de depuración, podemos modificar el código para resolver un problema detectado.

Una vez modificado el código, podemos pulsar la tecla **F5** de ejecución nuevamente, para relanzar la aplicación una vez más. De esta manera, comprobaremos que la modificación realizada resuelve el problema que detectó en este caso el entorno de desarrollo.

7.5.- Utilizando los recortes como solución a pequeños problemas

Una novedad introducida en el entorno de trabajo, son los denominados recortes de código o fragmentos de código. Estos fragmentos son pequeñas porciones de código insertadas dentro del entorno, que nos facilita tareas muy repetitivas.

Visual Basic 2005 Express Edition, viene con un conjunto de recortes o fragmentos de código listos para ser utilizados, pero nosotros mismos podemos crear los nuestros propios para incorporarlos en el entorno. De hecho, hay ya herramientas en Internet desarrolladas por terceros para realizar estas tareas.

Para ver los fragmentos de código en acción, haga clic con el botón derecho del ratón sobre el código y seleccione la opción **Insertar fragmento de código...** como se muestra en la figura 7.21.

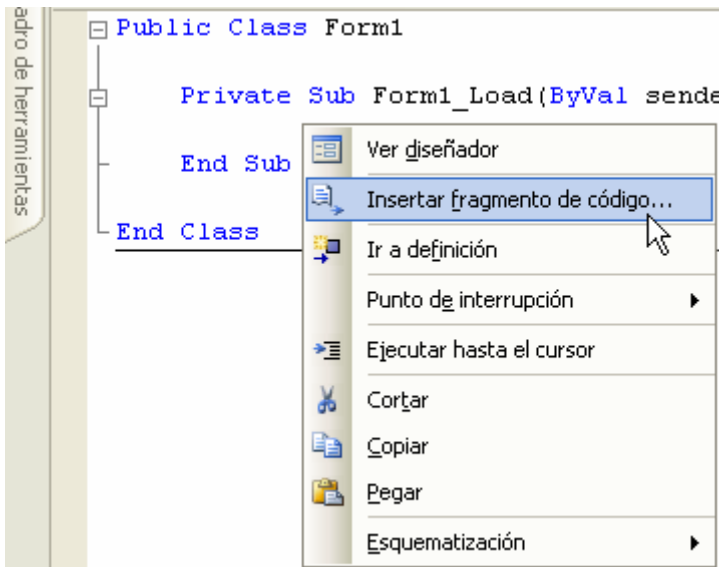


Figura 7.21: opción para Insertar fragmento de código... en el entorno de desarrollo.

De esta manera, aparecerá una ventana como la que se muestra en la figura 7.22 con un repositorio de recursos a los que podremos acceder con pequeños clics de ratón o de teclado por medio de la tecla **Enter** y de los cursores. De esta forma, accederemos a los recortes o fragmentos de código del entorno.

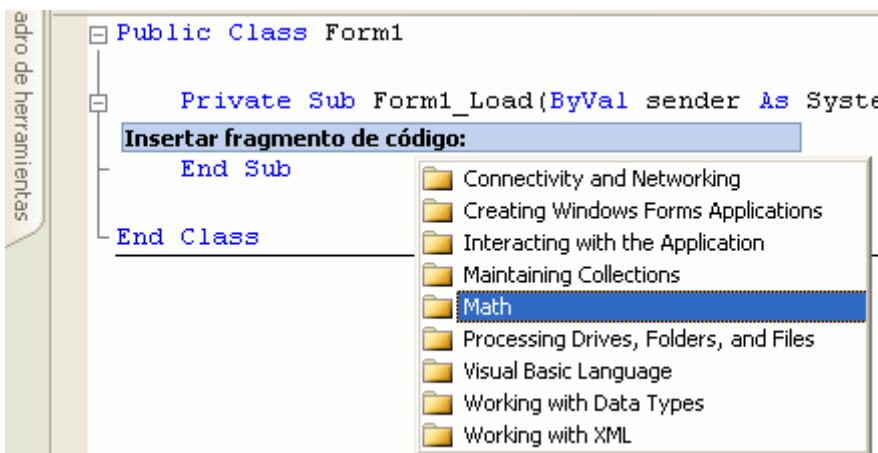


Figura 7.22: ventana de fragmentos de código en el entorno de desarrollo.

Al ir seleccionando los fragmentos de código, iremos adecuando el entorno de la manera en la que se muestra en la figura 7.23.

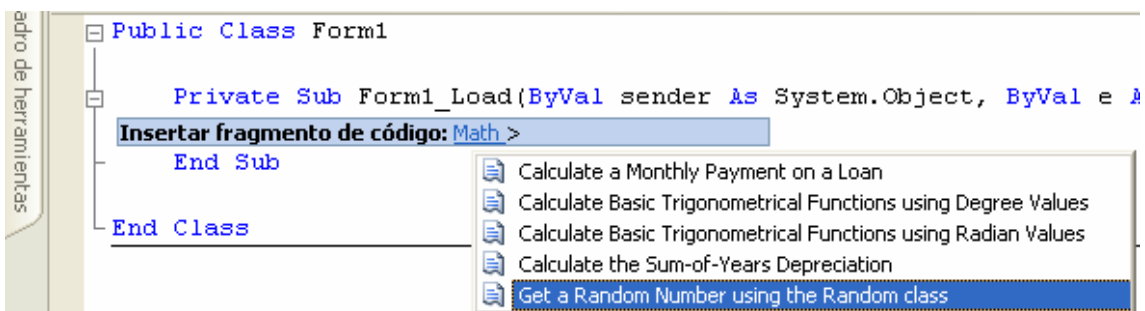


Figura 7.23: fragmentos de código seleccionados en el entorno.

Una vez seleccionado el fragmento, aparecerá en el código de la aplicación, el código base insertado que debemos pulir levemente para adecuarlo a nuestras necesidades. Al tratarse de un recorte o fragmento de código, lo que se inserta es la base de ese código, tal y como se muestra en la figura 7.24.

```

Public Class Form1
    Private Sub Form1_Load(ByVal sender As System.Object, ...
        Dim generator As New Random
        Dim randomValue As Integer
        ' Generates numbers between 1 and 5, inclusive.
        randomValue = generator.Next(1, 6)
    End Sub
End Class
  
```

Figura 7.24: fragmentos de código seleccionados en el entorno.

Indudablemente, siempre que trabajemos con fragmentos de código, debemos realizar pequeños cambios y adecuaciones para prepararlo a nuestras necesidades.

7.6.- Diseñando nuestras aplicaciones Windows

Otra de las partes más destacables del entorno de trabajo *Visual Basic 2005 Express Edition*, es el trabajo que debemos hacer con los controles dentro de un formulario.

En un formulario, podemos insertar muchísimos controles y objetos, pero disponerlos en el formulario de forma que queden bien definidos, es una tarea a veces más que trabajosa.

A continuación veremos algunas de las acciones generales a la hora de trabajar con controles en el entorno de trabajo, y repasaremos también algunos de los controles nuevos destacables incorporados a esta nueva versión de .NET Framework.

7.6.1.- Cuadro de herramientas

Dentro de la ventana **Cuadro de herramientas**, encontraremos todos los controles y componentes listos para ser utilizados en nuestros formularios Windows.

Adicionalmente, podemos agregar controles y componentes adicionales, reorganizar los controles y componentes, cambiar el nombre de estos objetos o de las solapas que los contienen, etc.

7.6.2.- Controles contenedores

Existen un conjunto de controles denominados contenedores que encontraremos en la ventana del **Cuadro de herramientas**, tal y como se muestra en la figura 7.25.

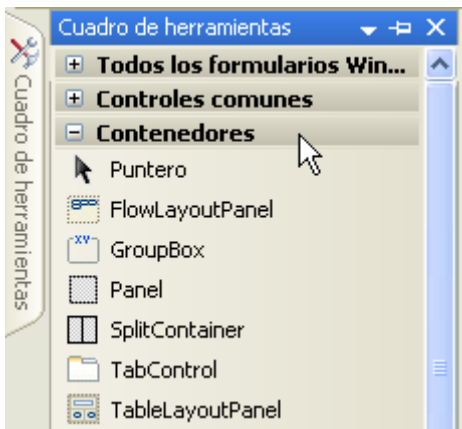


Figura 7.25: Contenedores del Cuadro de herramientas.

Un control contenedor no es otra cosa que un control con la capacidad de contener otros controles dentro de estos. Esto nos ofrece la posibilidad de ajustar, redimensionar, situar y mover controles sin apenas esfuerzo.

Como ejemplo del comportamiento de uno de estos controles en el entorno de desarrollo, añadiremos un control **TableLayoutPanel** para contener otros controles. Este control insertado en el formulario Windows es similar al que se muestra en la figura 7.26.

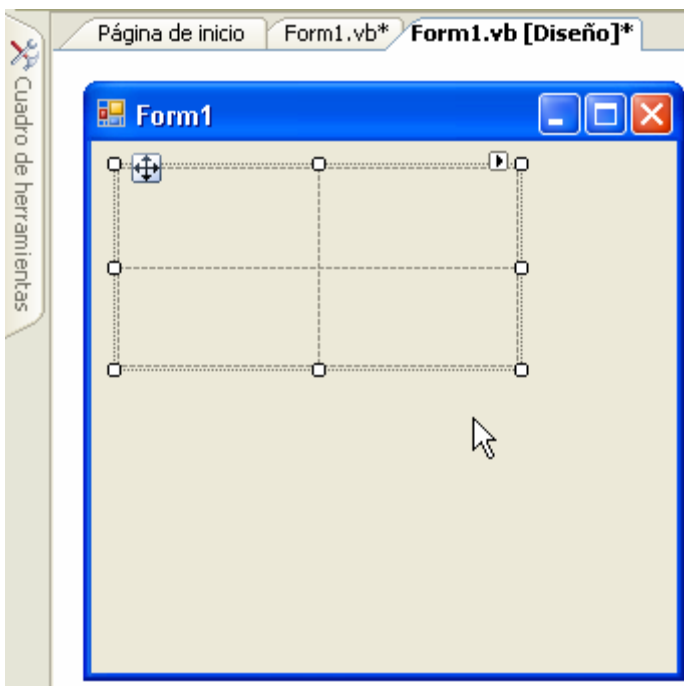


Figura 7.26: control TableLayoutPanel insertado en un formulario Windows.

Como vemos, este control simula una tabla de 2 filas y 2 columnas. En nuestro caso, eliminaremos la fila inferior y nos quedaremos con la fila superior. Para hacer esto, nos posicionaremos sobre la segunda fila y pulsaremos el botón derecho del ratón. Del menú emergente, seleccionaremos la opción **Fila > Eliminar** tal y como se muestra en la figura 7.27.

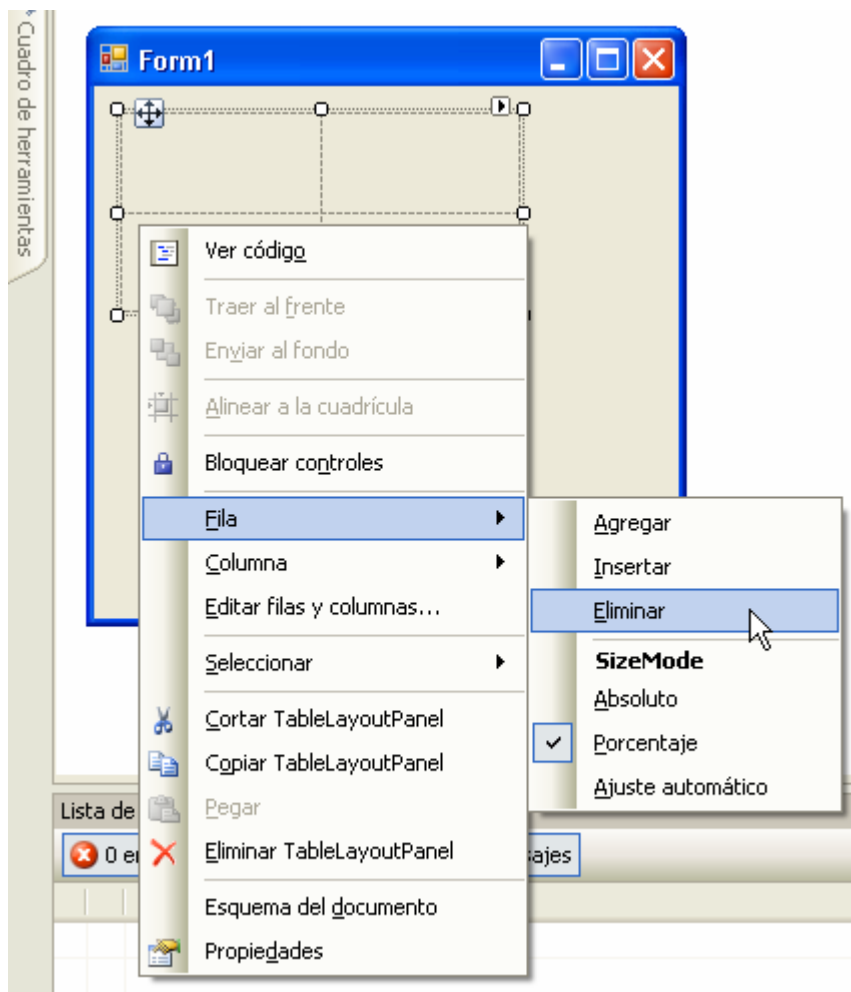


Figura 7.27: eliminando una fila del control `TableLayoutPanel`.

Después de seleccionar esta opción, nuestro control `TableLayoutPanel`, estará formado por una tabla de 1 fila y 2 columnas como se muestra en la figura 7.28.

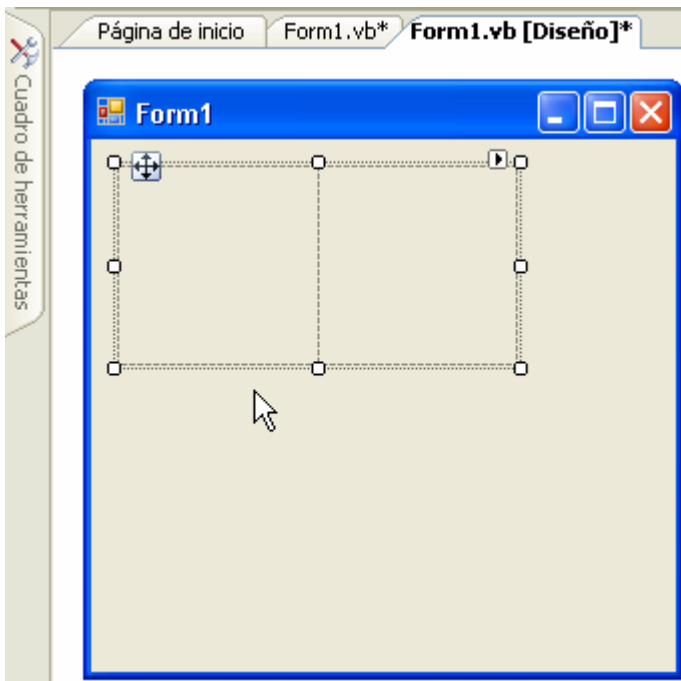


Figura 7.28: control `TableLayoutPanel` con 1 fila y 2 columnas.

Ahora ya tenemos nuestro formulario Windows con nuestro control `TableLayoutPanel` listo para ser utilizado. Como es un control contenedor, lo que tendremos que hacer a continuación es insertar dos controles que se sitúen dentro de cada fila y columna del control contenedor.

En nuestro ejemplo, insertaremos dos controles `Button` que encontraremos en la solapa **Controles comunes** de la ventana **Cuadro de herramientas** del entorno de desarrollo. Estos controles insertados en el formulario quedarán tal y como se muestran en la figura 7.29.

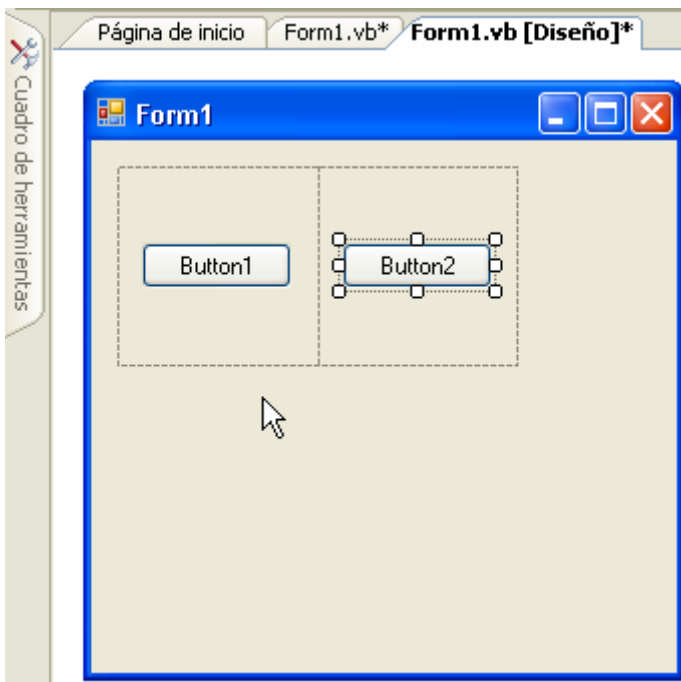


Figura 7.29: controles `Button` insertados en el control `TableLayoutPanel`.

Lo que haremos ahora para comprobar como funciona este control es redimensionar el control alargando y reduciendo el espacio del control como se muestra en la figura 7.30.

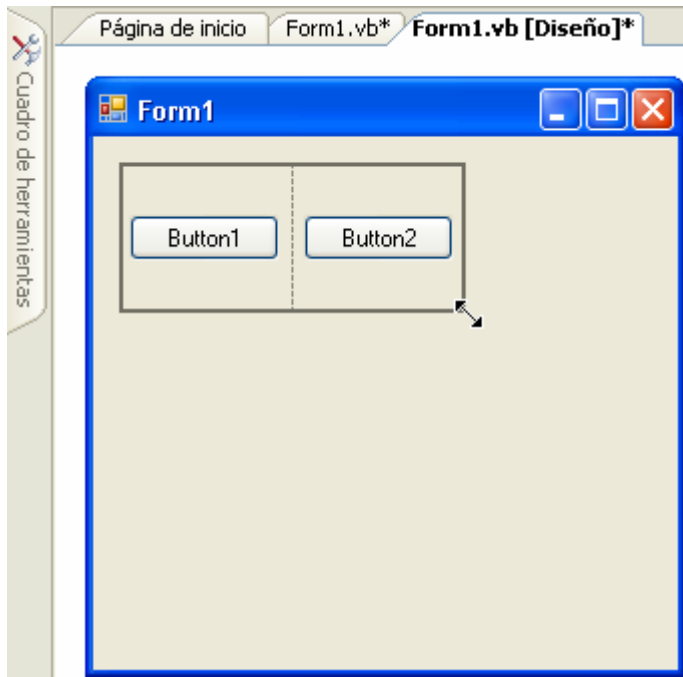


Figura 7.30: redimensión del control TableLayoutPanel en el formulario Windows.

Como podemos observar, los controles contenidos en el control contenedor, se redimensionan de acuerdo a las dimensiones del control contenedor. De todas las maneras, podemos manipular los espacios que hay de alto y ancho dentro de cada fila y columna del control **TableLayoutPanel**.

Para tener acceso a las propiedades de este control, podemos acceder a la ventana de propiedades pulsando la tecla **F4** del teclado o bien, pulsando el botón derecho del ratón sobre el control y seleccionando la **opción Editar filas y columnas...** del menú emergente, tal y como se muestra en la figura 7.31.

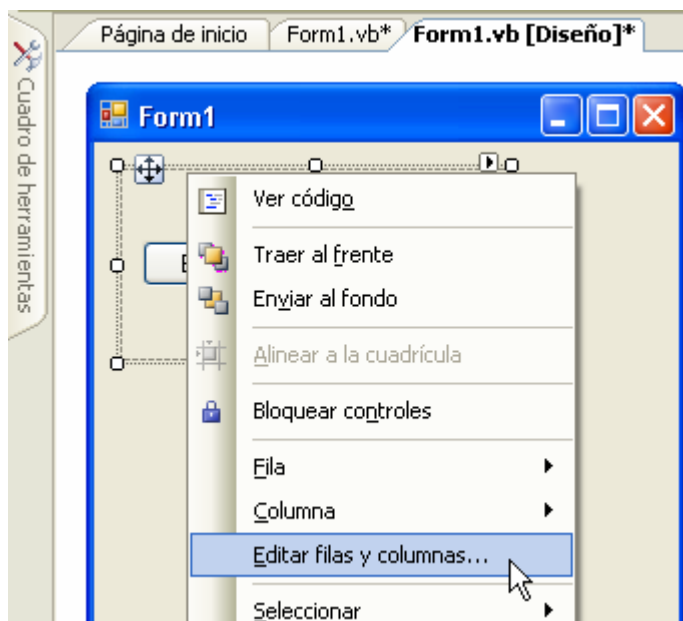


Figura 7.31: redimensión del control TableLayoutPanel en el formulario Windows.

También podemos acceder a estas propiedades de forma directa desde la opción **Fila** y **Columna** del mismo menú emergente.

7.6.3.- Posicionando los controles en nuestros formularios

Cuando diseñamos nuestras aplicaciones con *Visual Basic 2005 Express Edition*, el entorno nos facilita un conjunto de herramientas que nos ayudan enormemente a posicionar nuestros controles en los formularios y a manipular estos para conseguir que se muestren en un formulario tal y como deseamos.

Insertaremos dos controles **Button** y dos controles **TextBox** como se muestra en la figura 7.32.

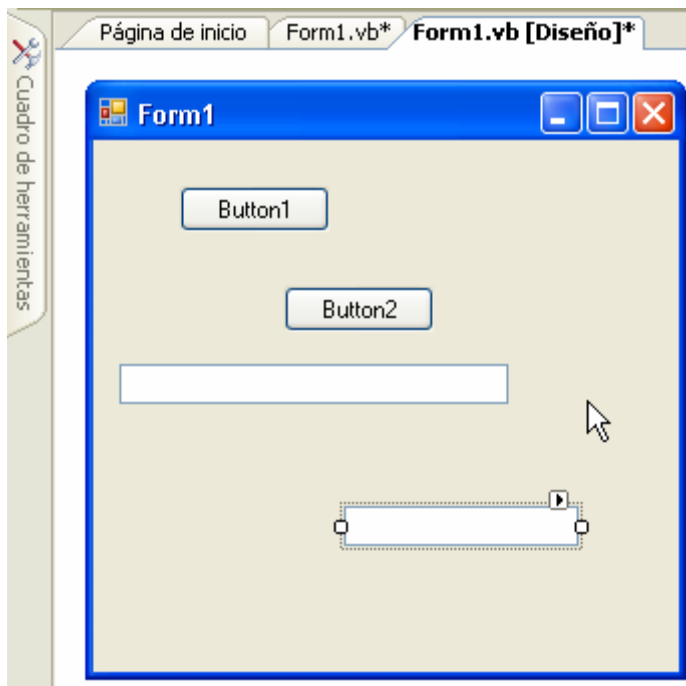


Figura 7.32: controles **Button** y **TextBox** insertados en nuestro formulario **Windows**.

Ahora bien, nuestro objetivo es poner estos controles de forma ordenada, así que situaremos el control **Button** antes que el **TextBox** para posicionarlos de una forma más o menos ordenada.

De esta manera tendremos nuestros controles dispuestos en el formulario más o menos como se muestra en la figura 7.33.

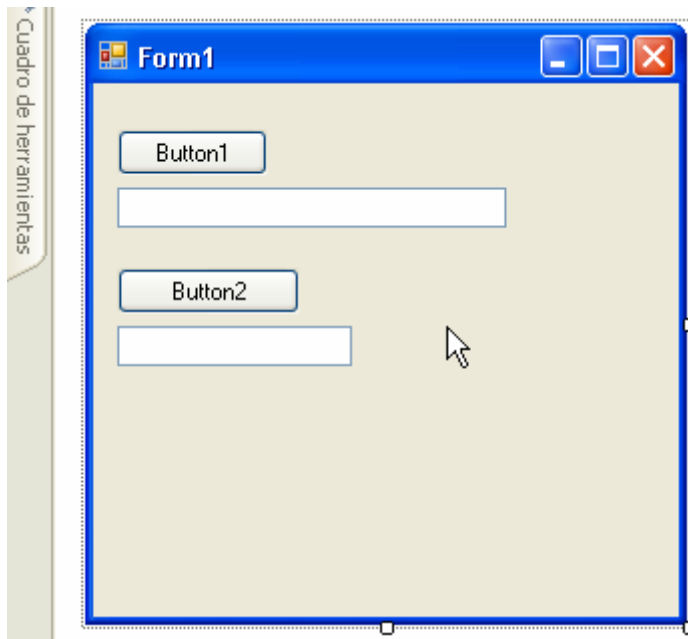


Figura 7.33: controles **Button** y **TextBox** insertados en nuestro formulario **Windows**.

La tarea que tenemos ahora por delante, es la de posicionar los controles dentro del entorno de desarrollo para dejar los controles de nuestro formulario lo más ordenado posible.

Seleccione los dos controles **Button** seleccionando en primer lugar el control que cuyo tamaño queremos que sirva de patrón. En mi caso he seleccionado el control **Button1**. Posteriormente y sin soltar la tecla **Ctrl** he seleccionado el segundo control, el control **Button2** y he elegido del menú de botones la opción **Igualar tamaño** como se muestra en la figura 7.34.

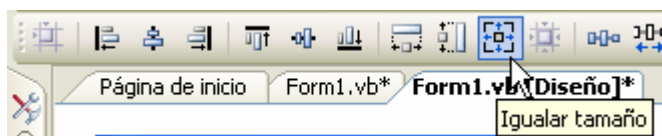


Figura 7.34: igualando los tamaños de los controles **Button** del formulario.

Con esto, nuestros controles **Button** tendrán el mismo tamaño. De igual manera, haremos con los controles **TextBox**, seleccionando por ejemplo el control **TextBox1** en primer lugar.

Una vez realizadas estas simples operaciones, los controles de nuestro formulario, tendrán un aspecto similar al que se muestra en la figura 7.35.

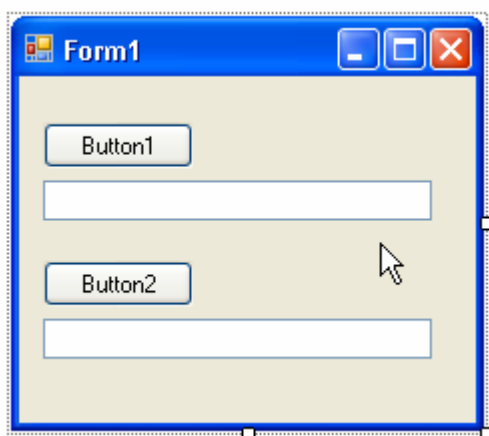


Figura 7.35: igualando los tamaños de todos los controles del formulario.

Otra característica del entorno *Visual Basic 2005 Express Edition* que debemos que tener en cuenta cuando desarrollamos aplicaciones Windows es la posibilidad de este entorno de preparar los controles de manera tal que nos permita alinearlos.

Hay opciones como la anterior, que nos permite esta posibilidad, pero dentro del entorno hay unas guías que nos ayudan eficientemente a saber cuando un determinado control se encuentra alineado con otro o no. En la figura 7.36., podemos observar esta operación para alinear controles en el entorno.

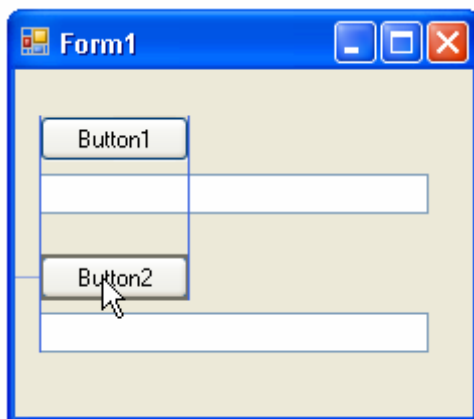


Figura 7.36: alineando los controles de un formulario con ayuda del entorno.

Esas guías o líneas azules, nos permite saber si un control está o no alineado con respecto a los otros objetos o controles cercanos insertados dentro del formulario. En esta figura 7.36., observamos que los controles están correctamente alineados.

Sólo como añadido breve a estas operaciones, de las cuales hay muchas y aquí sólo hemos visto una pequeña introducción, comentar que si seleccionamos dos controles de un formulario Windows, podemos modificar su tamaño al mismo tiempo, aunque lo recomendable primero, es seleccionar la opción de **Alinear lados izquierdos** como se muestra en la opción de la figura 7.37.

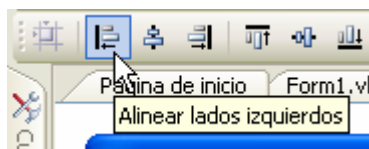


Figura 7.37: alineando lados izquierdos de los controles insertados en el formulario.

Y posteriormente a esta acción, seleccionar los dos controles y modificar los tamaños alargando cualquiera de los controles y seleccionando otras opciones de alineación de controles del entorno de desarrollo.

7.6.4.- Tabulando los controles en nuestros formularios

Otra de las opciones con la que debemos trabajar es la de tabular los controles. Para esto, podemos acceder a la opción **Orden de tabulación** de la barra de botones tal y como se muestra en la ventana 7.38.

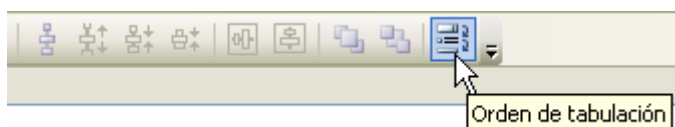


Figura 7.38: orden de tabulación de los controles insertados en un formulario.

O bien también, podemos seleccionar esta opción accediendo al menú **Ver > Orden de tabulación** como se muestra en la figura 7.39.

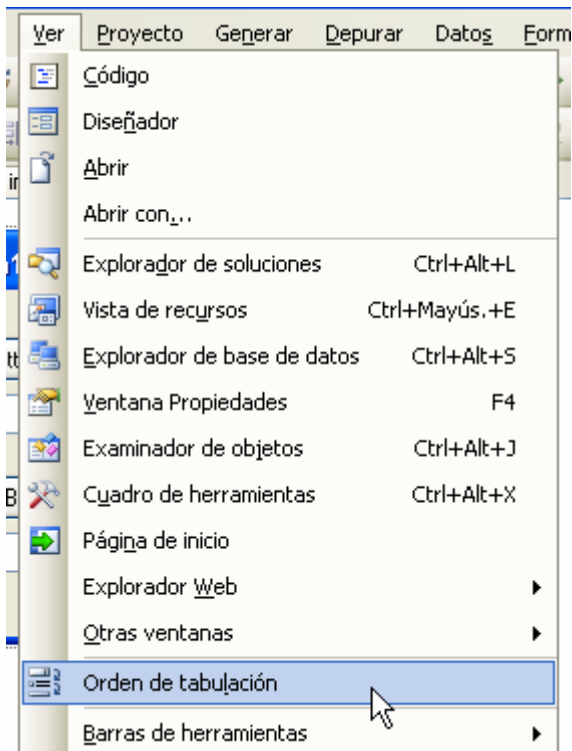


Figura 7.39: orden de tabulación de los controles insertados en un formulario desde el menú del entorno.

Al seleccionar cualquiera de estas dos opciones, los controles de nuestro formulario aparecerán con unos dígitos encima de estos que representan el orden de tabulación, orden que podemos modificar haciendo clic sobre estos dígitos y modificándolos según nuestras necesidades.

En la figura 7.40., podemos observar algunos controles insertados en el formulario y el orden de tabulación que representa cada control.

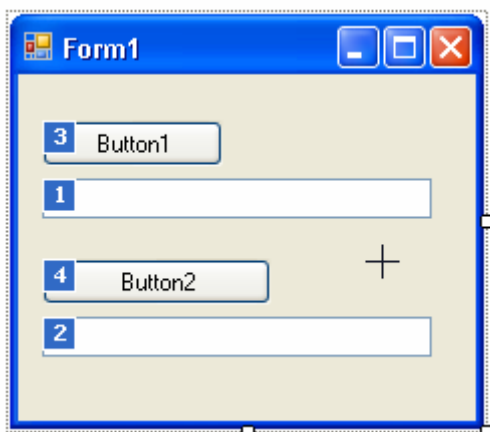


Figura 7.40: controles de un formulario con su correspondiente orden de tabulación.

De esta manera, y haciendo clic sobre punto, número ó dígito, podemos reordenar los órdenes de tabulación correspondientes a cada control insertado en el formulario, tal y como se indica en la figura 7.41.

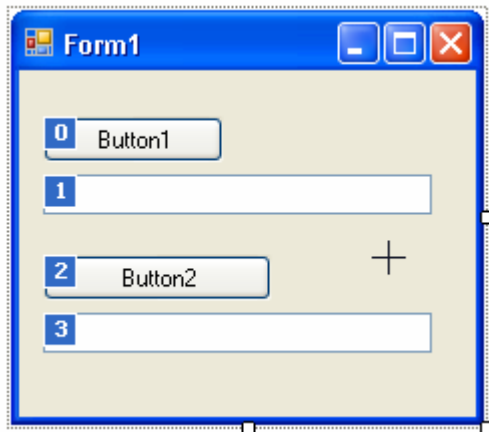


Figura 7.41: controles con su orden de tabulación correctamente indicado.

7.7.- Las propiedades de un proyecto

Si hacemos clic con el botón derecho del ratón sobre el proyecto y seleccionamos la opción Propiedades del menú emergente tal y como se muestra en la figura 7.42., accederemos a la ventana de propiedades del proyecto.

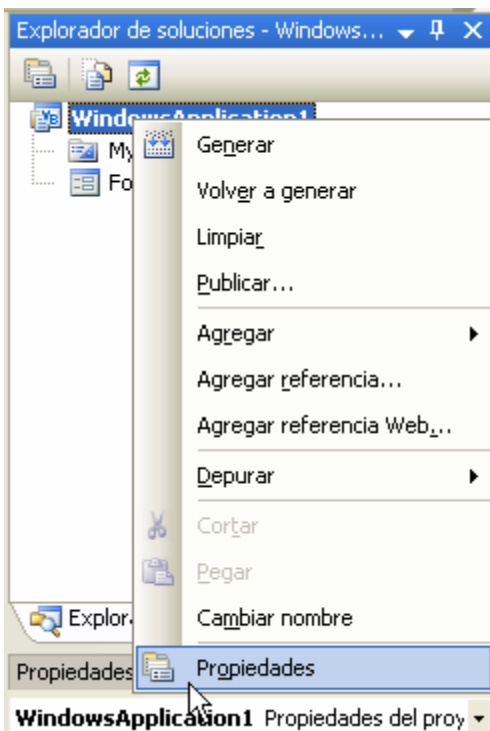


Figura 7.42: opción de Propiedades del proyecto.

Al seleccionar esta opción, accederemos a la ventana de **Propiedades** del proyecto. Ventana dentro de la cuál, podremos acceder a diferentes puntos de la aplicación relacionados con la aplicación, la compilación, las referencias de la aplicación a las librerías de uso, la configuración, los recursos, la firma, la seguridad y la publicación, todo ello de nuestra aplicación.

En la figura 7.43., podemos observar la ventana **Propiedades** del proyecto.

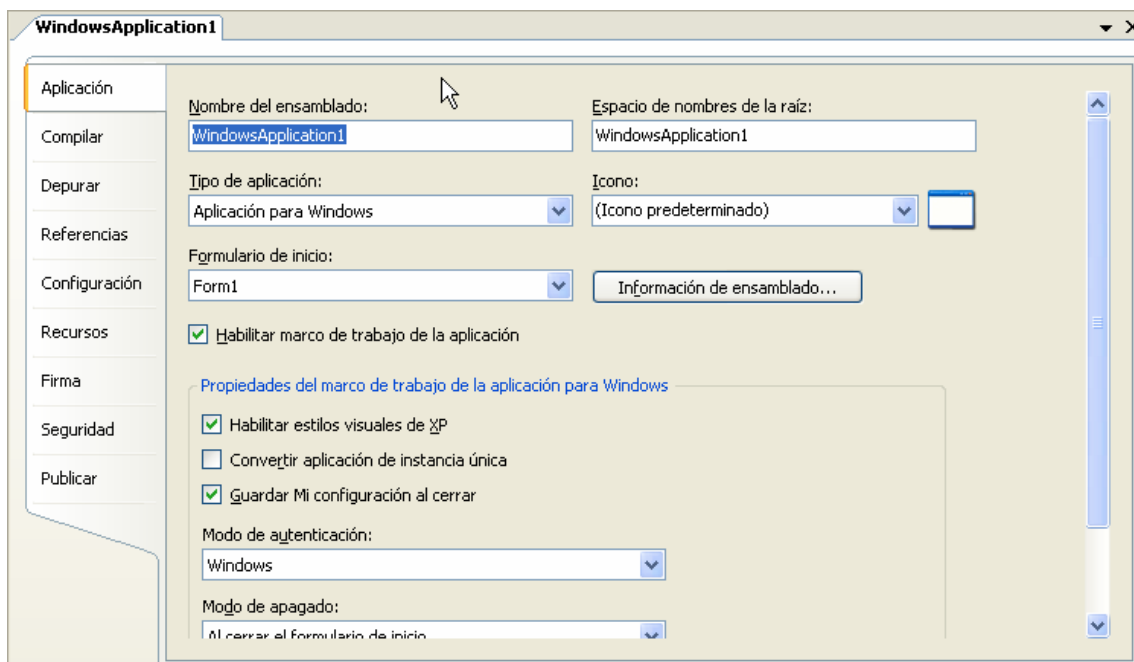


Figura 7.43: ventana de Propiedades abierta en el proyecto.

Dentro de esta ventana, se reúnen las partes fundamentales, directamente relacionadas con nuestro proyecto. De esta manera, tenemos visibilidad directa con toda la configuración de la aplicación y tenemos así, el control completo de la implantación de nuestro proyecto.

CAPÍTULO 8

MY, NAMESPACE PARA TODO

ESTE CAPÍTULO NOS MOSTRARÁ LAS CUALIDADES Y CARACTERÍSTICAS DEL NOMBRE DE ESPACIO MY INCORPORADO A VISUAL BASIC 2005.

En este capítulo aprenderemos a utilizar el nombre de espacio o Namespace **My**, que ha sido incorporado a Visual Basic 2005 para permitirnos acceder a funciones o acciones de uso frecuente y común en aplicaciones Windows.

8.1.- ¿En qué consiste My?

My es un nombre de espacios genérico de Visual Basic 2005 que nos permite acceder a diferentes servicios y objetos de forma directa sin necesidad de instanciarlo previamente.

Esto nos proporciona una rapidez y sencillez bastante alta a la hora de acceder a funciones de carácter general. Así, podemos ahorrar en muchas ocasiones, la necesidad de utilizar las APIs del sistema para realizar estas funciones o acciones generales. Pero, ¿cómo funciona **My**?. Esto es lo que veremos a continuación.

8.2.- Funcionamiento de My

Para utilizar este nombre de espacio, bastará con escribir la palabra reservada **my** en el entorno de desarrollo y así, aparecerá un conjunto de funciones organizadas de manera tal, que nos permita su uso extensivo dentro de nuestras aplicaciones.

Al escribir la palabra reservada **my** dentro del entorno de desarrollo, aparecerá una lista de funciones y clases listas para ser utilizadas tal y como se muestra en la figura 8.1.

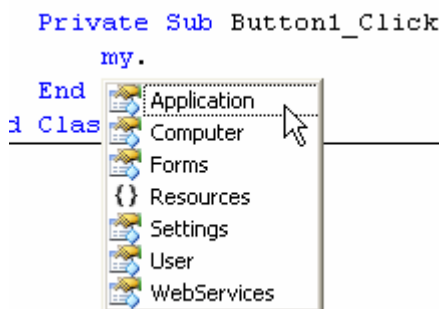


Figura 8.1: nombre de espacio My junto a las clases listas para ser utilizadas en Visual Basic 2005.

Cada función y clase tiene su cometido, de esta manera, **My.Application** nos proporciona información sobre la ejecución de la aplicación como el directorio actual dónde se ejecuta la aplicación, la versión de la aplicación, posibilidades para escribir logs ya sean de sistema o personalizados por nosotros mismos, etc.

My.Computer, nos proporciona acceso sobre la plataforma y el hardware del equipo. Así, podemos obtener información del registro, el teclado y la regionalización de éste, la pantalla, etc.

My.Forms nos proporciona una colección de todos los formularios del proyecto actual, permitiéndonos de manera sencilla, acceder a todos ellos para manipular algunas acciones generales, como mostrarlos, ocultarlos, etc., todo ello sin necesidad de instanciar los formularios.

My.Resources proporciona acceso a los recursos de la aplicación, permitiéndonos manejar la información de recursos de nuestra aplicación de forma dinámica.

My.Settings nos proporciona información sobre la configuración de nuestra aplicación. De esta manera, no sólo podemos salvar o guardar la configuración de usuario que indiquemos, sino que también podemos dar marcha atrás y utilizar la última configuración usada.

My.User proporciona información sobre el usuario que está ejecutándose en el sistema, proporcionando así, información de carácter general como el dominio y nombre de usuario, de que grupos es miembro el usuario, etc.

My.WebServices nos proporciona acceso rápido y sencillo a los Servicios Web que han sido añadidos como referencias a nuestro proyecto.

8.3.- Una primera toma de contacto con My

Para iniciarnos en el uso de **My**, probaremos este interesante nombre de espacio con un sencillo ejemplo que nos muestre y demuestre como utilizarlo. Será nuestro particular *Hola Mundo* de este nombre de espacio.

Sirva como ejemplo la necesidad que podemos tener en un momento dado de hacer sonar un determinado sonido. Si queremos realizar esta operación, tendremos a buen seguro que añadir interesantes porciones de código, pero con este nombre de espacio, todo estará solucionado en apenas una simple línea de código.

Añadiremos a nuestro formulario Windows un control **Button** y añadiremos el siguiente código fuente:

```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Button1.Click
    My.Computer.Audio.Play("c:\windows\media\notify.wav")
End Sub
```

Si ejecutamos este código y pulsamos el botón del formulario, oiremos el correspondiente sonido wav en el sistema operativo Windows.

Como vemos, el objeto **My** nos proporciona entre otras cosas, la posibilidad de simplificar enormemente nuestras aplicaciones. En apenas una línea de código, hemos sido capaces de realizar una simple función que en otras ocasiones o con versiones anteriores de Visual Basic, consumiría gran parte de nuestro tiempo o nos obligaría a escribir nuestra propia clase, eso sin contar las líneas de código a las que estaríamos obligados a escribir para realizar esta operación.

Para que veamos con otro ejemplo la increíble simplicidad de **My**, escribiremos a continuación el siguiente código:

```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Button1.Click
    Dim strTexto As String =
My.Computer.FileSystem.ReadAllText("c:\Prueba.txt")
    MessageBox.Show(strTexto)
End Sub
```

Este código, nos permite abrir un fichero de texto, leer su contenido y en este caso, almacenar su contenido en una variable para mostrarlo por pantalla.

8.4.- El corazón de My

En sí, **My** no es otra cosa que un atajo que nos permite realizar acciones *pesadas* de una manera mucho más ligera, ahorrando gran cantidad de código y tiempo.

El equipo de desarrollo de Visual Basic, se marcó como objetivo el hecho de desarrollar un nombre de espacio que facilitara la vida a los programadores que utilizan un entorno RAD (*Rapid Application Development*) como *Visual Basic 2005 Express Edition* y un lenguaje de programación tan usado y extendido como Visual Basic 2005.

De esta manera, usando el nombre de espacio **My**, los programadores pueden escribir código de aplicaciones de forma extraordinariamente rápida, sin tantas complicaciones y reduciendo así el código enormemente.

CAPÍTULO 9

XML, LOS DOCUMENTOS EXTENSIBLES

ESTE CAPÍTULO NOS PERMITIRÁ CONOCER COMO USAR XML COMO DOCUMENTOS EXTENSIBLE DE MARCAS EN NUESTRAS APLICACIONES VISUAL BASIC 2005.

Los documentos XML nos proporcionan muchas ventajas dentro del desarrollo de aplicaciones actuales. La interacción entre diferentes sistemas o entornos, aplicaciones con diferentes naturalezas y otros recursos, proporcionan a XML una forma de comunicación ideal en la mayoría de los casos.

En Visual Basic 2005 podemos usar XML en nuestras aplicaciones. Recoger datos, tratar y manipular datos, traspasar esos datos a un **DataSet** para su tratamiento como fuente de datos, etc.

En este capítulo veremos algunas de las formas de trabajar con documentos XML y como Visual Basic 2005, nos facilita una vez más, la vida en el tratamiento de este tipo de documentos.

9.1.- Agregando la referencia a System.Xml

El nombre de espacio encargado de tratar y manipular los documentos XML es el denominado **System.Xml**. Este nombre de espacio lo deberemos agregar al proyecto mediante la palabra reservada **Imports**, pero antes, deberemos agregar una referencia a la librería en el entorno, por eso, accederemos a la ventana de **Propiedades** del proyecto que vimos en el capítulo 7.

Una vez dentro de la ventana **Propiedades**, accederemos a la solapa **Referencias**, y dentro de esta ventana, seleccionaremos el botón **Agregar** como se indica en la figura 9.1.

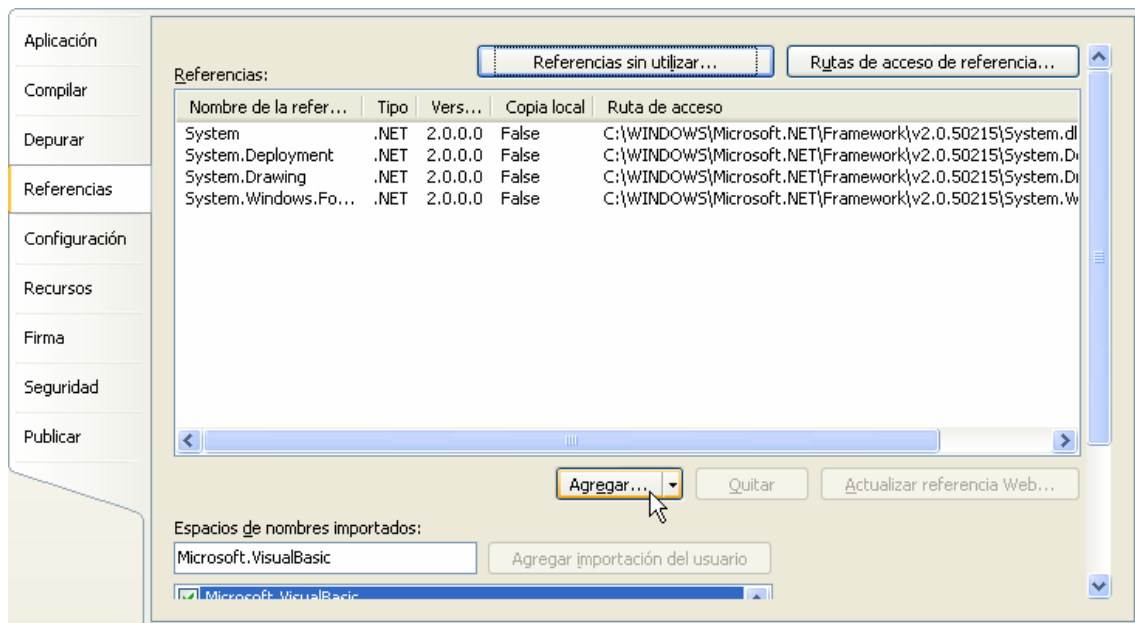


Figura 9.1: botón **Agregar** de la solapa **Referencias** dentro de la ventana **Propiedades** del proyecto.

Al pulsar el botón **Agregar**, el entorno abrirá una ventana con las librerías listas para ser incluidas en el proyecto. En nuestro caso, buscaremos la librería **System.Xml.dll** que se encuentra en Windows como se indica en la figura 9.2., y haremos doble clic sobre ella para agregarla a la ventana de referencias del proyecto.

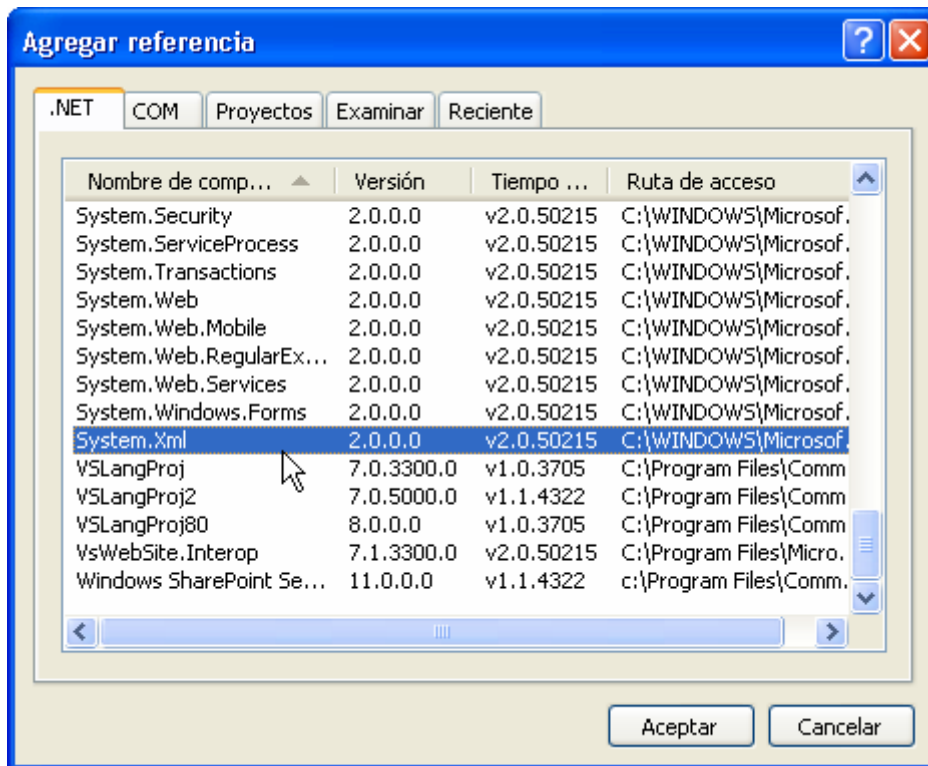


Figura 9.2: ventana para Agregar referencia al proyecto abierto.

Una vez seleccionada la librería, ésta aparecerá en el proyecto referenciada, pero aún podemos usarla directamente declarándola previamente o bien, importarla mediante la palabra reservada **Imports**.

9.2.- Leer XML con XmlTextReader

A continuación veremos la forma que tenemos en Visual Basic 2005, de leer un documento XML a través del objeto **XmlTextReader**.

Una vez que tenemos agregada la referencia al proyecto, bastará con importar la librería (si queremos) mediante la instrucción **Imports** y trabajar con la clase directamente.

El siguiente documento XML será el que empleemos como ejemplo para leerlo con **XmlTextReader**, como veremos a continuación:

```
<?xml version="1.0" encoding="UTF-8"?>
<Coches>
  <Marca Fabricante="Audi">
    <Modelo>A3</Modelo>
  </Marca>
  <Marca Fabricante="Citroen">
    <Modelo>C1</Modelo>
    <Modelo>C5</Modelo>
  </Marca>
  <Marca Fabricante="Seat">
    <Modelo>Arosa</Modelo>
    <Modelo>Leon</Modelo>
  </Marca>
</Coches>
```

Este documento XML deberá ser recorrido para leer su contenido. En nuestro caso, hemos utilizado un código fuente, forzando la estructura del documento, para demostrar simplemente, como trabajar con **XmlTextReader**.

De esta forma, nuestro código será el que se asemeje al siguiente:

```
Imports System.Xml

Public Class Form1

#Region "PROPIEDAD: Fabricante"
  Private m_Fabricante As String

  Private Property Fabricante() As String
  Get
    Return m_Fabricante
  End Get
  Set(ByVal value As String)
    m_Fabricante = value
  End Set
End Property
#End Region

#Region "PROPIEDAD: Modelo"
  Private m_Modelo As String

  Private Property Modelo() As String
  Get
    Return m_Modelo
  End Get
  Set(ByVal value As String)
    m_Modelo = value
  End Set
End Property
#End Region

  Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
    Dim xmltr As New XmlTextReader("c:\prueba.xml")
    xmltr.WhitespaceHandling = WhitespaceHandling.None
    Dim bolDatos As Boolean = False
    xmltr.Read()
    xmltr.Read()
    xmltr.Read()
    'Ahora estamos posicionados donde queremos
    While Not xmltr.EOF
      'Si es Marca => Fabricante
      If xmltr.LocalName = "Marca" Then
        'Leemos Fabricante
        Fabricante = xmltr.GetAttribute("Fabricante")
        'Continuamos leyendo
        xmltr.Read()
      ElseIf xmltr.LocalName = "Modelo" Then
        'Leemos Modelo
        Modelo = xmltr.ReadElementString("Modelo")
        'Indicamos que ya tenemos datos de Fabricante y Modelo
        bolDatos = True
      Else
        'Sino hay ninguna de las anteriores, leemos el siguiente
        elemento
        xmltr.Read()
      End If
      'Si hay datos, Fabricante y Modelo, mostramos los datos en
      pantalla
      If bolDatos Then MessageBox.Show(Fabricante & vbCrLf & Modelo) :
bolDatos = False
    End While
    'Cerramos el documento XML
```

```
xmltr.Close()
End Sub
End Class
```

Este ejemplo, nos mostrará por pantalla los datos relacionados con las marcas o fabricantes de automóviles y su correspondiente modelo. Como podemos ver, tenemos dos funciones que serán las encargadas de leer el atributo y el elemento. Estas son **GetAttribute** y **ReadElementString**.

A continuación, veremos otro ejemplo, de cómo leer documentos XML con el objeto **XmlDocument**.

9.3.- Leer XML con XmlDocument

A continuación y después de ver **XmlTextReader**, veremos como usar **XmlDocument** para leer documentos XML en Visual Basic 2005.

Reutilizando el documento XML anterior y de una forma muy parecida, escribimos el siguiente código:

```
Imports System.Xml

Public Class Form1

#Region "PROPIEDAD: Fabricante"
  Private m_Fabricante As String

  Private Property Fabricante() As String
  Get
    Return m_Fabricante
  End Get
  Set(ByVal value As String)
    m_Fabricante = value
  End Set
End Property
#End Region

#Region "PROPIEDAD: Modelo"
  Private m_Modelo As String

  Private Property Modelo() As String
  Get
    Return m_Modelo
  End Get
  Set(ByVal value As String)
    m_Modelo = value
  End Set
End Property
#End Region

  Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
    'Preparamos las variables
    Dim xmld As New XmlDocument
    Dim nodelist As XmlNodeList
    Dim node As XmlNode
    'Cargamos el documento XML
    xmld.Load("c:\prueba.xml")
    nodelist = xmld.SelectNodes("/Coches/Marca")
    'Recorremos cada nodo
    For Each node In nodelist
      'Recogemos el valor del Fabricante
      Fabricante = node.Attributes.GetNamedItem("Fabricante").Value
      'Recorremos la lista de hijos
      For I As Integer = 1 To node.ChildNodes.Count
        'Recogemos el valor del Modelo
```

```
        Modelo = node.ChildNodes.Item(I - 1).InnerText
        'Mostramos la información en pantalla
        MessageBox.Show(Fabricante & vbCrLf & Modelo)
    Next
Next
End Sub
End Class
```

Como podemos observar, leemos la información del documento XML con ayuda de la función **GetNamedItem** y de la propiedad **ChildNodes**.

Así, recorremos su información y la mostramos por pantalla.

Aún y así, también es posible leer documentos XML con ayuda de **XPathDocument**, que es lo que veremos a continuación.

9.4.- Leer XML con XPathDocument

A continuación veremos ahora, como leer documentos XML en Visual Basic 2005 con ayuda de **XPathDocument**. De esta manera, haremos un repaso general a como leer documentos XML desde Visual Basic 2005.

Escriba el siguiente código fuente:

```
Imports System.Xml.XPath

Public Class Form1

    Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
        'Leemos el documento XML y declaramos las variables a utilizar
        Dim xmlPathDoc As New XPathDocument("c:\prueba.xml")
        Dim xmlNav As XPathNavigator
        Dim xmlNI As XPathNodeIterator
        'Creamos el proceso de navegación
        xmlNav = xmlPathDoc.CreateNavigator
        'Indicamos el árbol del documento XML de dónde
        'obtendremos los datos que nos interesan
        xmlNI = xmlNav.Select("/Coches/Marca/Modelo")
        'Recorremos el documento XML
        While xmlNI.MoveNext()
            'Mostramos la información
            MessageBox.Show(xmlNI.Current.Value)
        End While
    End Sub
End Class
```

Nuestro ejemplo en ejecución nos mostrará el contenido del documento XML con los modelos de los vehículos que hay dentro del documento.

Otra posibilidad sin embargo a la hora de trabajar con documentos XML es la que nos permite acceder a los documentos XML como si fuera una fuente de datos. Esto lo lograremos utilizando para ello la clase **DataSet**.

9.5.- Leer un XML como un DataSet

La clase **DataSet** nos permite trabajar con un documento XML como si de una fuente de datos se tratara.

En el siguiente ejemplo, leeremos el documento XML con el cuál hemos trabajado hasta ahora, para volcar su información en un **DataSet**. En nuestro caso no tenemos un esquema del documento, por lo que el

documento será insertado de manera tal, que se almacene cierta relación entre los datos, por lo que en nuestro caso, tendremos dos tablas dentro del **DataSet** relacionadas entre sí.

Antes de continuar, debemos indicar que para trabajar con la clase **DataSet**, deberemos agregar una referencia al nombre de espacio **System.Data** como ya hemos realizado anteriormente para el nombre de espacio **System.Xml.XPath**.

El siguiente ejemplo por lo tanto, nos muestra una forma de cargar un documento XML en un **DataSet**, y como recorrerlo con el objetivo de mostrar sus datos:

```
Imports System.Xml
Imports System.Data

Public Class Form1

    #Region "PROPIEDAD: Fabricante"
        Private m_Fabricante As String

        Private Property Fabricante() As String
            Get
                Return m_Fabricante
            End Get
            Set(ByVal value As String)
                m_Fabricante = value
            End Set
        End Property
    #End Region

    #Region "PROPIEDAD: Modelo"
        Private m_Modelo As String

        Private Property Modelo() As String
            Get
                Return m_Modelo
            End Get
            Set(ByVal value As String)
                m_Modelo = value
            End Set
        End Property
    #End Region

    Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
        'Declaramos un DataSet
        Dim dtMiDS As New DataSet
        'Leemos los datos del documento XML
        dtMiDS.ReadXml("c:\prueba.xml")
        Dim miDoc As New XmlDocument(dtMiDS)
        'Recorremos las tablas y datos del DataSet
        Dim row
        For Each row In miDoc.DataSet.Tables(0).Rows
            'Las dos tablas están relacionadas
            'mediante un campo de tipo índice
            Dim intRelacion = row(0)
            Fabricante = row(1).ToString()
            Dim row2
            For Each row2 In miDoc.DataSet.Tables(1).Rows
                If row2(1) = intRelacion Then
                    Modelo = row2(0).ToString()
                    'Mostramos la pareja de valores Fabricante - Modelo
                    MessageBox.Show(Fabricante & vbCrLf & Modelo)
                End If
            Next
        Next
    Next
End Sub
```

```
End Sub
End Class
```

De esta manera, recorreremos los campos de las tablas de nuestro **DataSet** y mostramos su contenido.

9.6.- Ejemplo práctico para escribir un documento XML

Sirva el siguiente ejemplo como demostración de cómo podemos escribir un documento XML de manera sencilla:

```
Imports System.Xml
Imports System.Data

Public Class Form1

  #Region "PROPIEDAD: Fabricante"
    Private m_Fabricante As String

    Private Property Fabricante() As String
      Get
        Return m_Fabricante
      End Get
      Set(ByVal value As String)
        m_Fabricante = value
      End Set
    End Property
  #End Region

  #Region "PROPIEDAD: Modelo"
    Private m_Modelo As String

    Private Property Modelo() As String
      Get
        Return m_Modelo
      End Get
      Set(ByVal value As String)
        m_Modelo = value
      End Set
    End Property
  #End Region

  Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
    Dim txtUE As New System.Text.UTF8Encoding
    Dim objXML As New XmlTextWriter("c:\prueba_wr.xml", txtUE)
    With objXML
      .WriteStartDocument()
      .WriteStartElement("Coches")
      '-----
      .WriteStartElement("Marca")
      .WriteAttributeString("Fabricante", "Audi")
      .WriteStartElement("Modelo")
      .WriteName("A3")
      .WriteEndElement()
      .WriteEndElement()
      '-----
      .WriteStartElement("Marca")
      .WriteAttributeString("Fabricante", "Citroen")
      .WriteStartElement("Modelo")
      .WriteName("C1")
      .WriteEndElement()
      .WriteStartElement("Modelo")
      .WriteName("C5")
    End With
  End Sub
End Class
```

```
.WriteEndElement()  
.WriteEndElement()  
'-----  
.WriteStartElement("Marca")  
.WriteAttributeString("Fabricante", "Seat")  
.WriteStartElement("Modelo")  
.WriteName("Arosa")  
.WriteEndElement()  
.WriteStartElement("Modelo")  
.WriteName("Leon")  
.WriteEndElement()  
.WriteEndElement()  
'-----  
.WriteEndElement()  
.WriteEndDocument()  
.Close()  
End With  
End Sub  
End Class
```

Como podemos observar, la forma de escribir un documento XML es realmente sencilla. La clase **XmlTextWriter** es la encargada de realizar la escritura del documento XML, y tan sólo debemos indicar las partes que forman parte del documento y los valores de estas partes.

CAPÍTULO 10

BREVE INTRODUCCIÓN AL ACCESO A DATOS

ESTE CAPÍTULO NOS PERMITIRÁ INTRODUCIRNOS BREVEMENTE EN EL DESARROLLO DE APLICACIONES DE ACCESO A DATOS CON VISUAL BASIC 2005.

Las aplicaciones de acceso a datos, forman casi de forma habitual, parte de una aplicación Windows estándar. Sin lugar a dudas, no siempre necesitaremos acceder a fuentes de datos en nuestras aplicaciones, de hecho, podemos incluso trabajar con documentos XML, pero en muchas ocasiones, nos resultará imprescindible acceder a fuentes de datos.

En el siguiente capítulo, veremos como acceder a fuentes de datos de forma sencilla, desde nuestro entorno de desarrollo.

10.1.- Una pequeña introducción a ADO.NET

Si conoce el desarrollo de aplicaciones con versiones anteriores de Visual Basic a la tecnología .NET, entonces le sonará algunos aspectos entre otros como ADO, DAO.

Visual Basic 2005 por su lado, ha visto de cerca el nacimiento de ADO.NET. Sin embargo, no es recomendable comparar ADO.NET con los anteriores métodos de conexión a fuentes de datos. Nos olvidaremos de todo esto y nos centraremos en la forma de conectar de Visual Basic 2005 con fuentes de datos a través de ADO.NET 2.0.

ADO.NET es por lo tanto, un conjunto de clases que nos permiten leer e interactuar con fuentes de datos almacenadas en bases de datos y otras fuentes de almacenamiento de datos.

Estas clases de ADO.NET, las encontraremos dentro de **System.Data**. Entre todas las clases de este nombre de espacio, destacaremos la clase **DataView**, **DataSet** y **DataTable**.

Si usted sabe, conoce o ha trabajado con **RecordSet** en anteriores versiones de Visual Basic, debe saber que ahora, podremos hacer lo mismo con los objetos **DataSet** y **DataReader**.

Pero sobre todo, algo que destaca a ADO.NET sobre otros métodos de acceso, es que nos permite el desarrollo de aplicaciones en n-capas. Todo esto, unido a la posibilidad de trabajar con estándares como documentos XML, nos proporciona una intercomunicación entre entornos y dispositivos que está asegurada.

Aún y así, existe otro detalle a tener en cuenta a la hora de trabajar con fuentes de datos junto a las clases y objetos de la tecnología .NET, y es que podemos trabajar con fuentes de datos conectadas o fuentes de datos desconectadas. La diferencia entre ellas, es la que veremos de forma breve a continuación.

10.2.- ¿Acceso conectado o acceso desconectado?

Cuando trabajamos con fuentes de datos dentro de la tecnología .NET, podemos trabajar con fuentes de datos conectadas o fuentes de datos desconectadas.

La diferencia entre ambos métodos de conexión es clara.

Por un lado, el trabajo con fuentes de datos conectadas, requiere que exista un canal de comunicación existente entre la aplicación y la fuente de datos en todo momento. De esta manera, la conectividad con la aplicación, la apertura de la conexión, etc., que es la tarea más pesada en la comunicación de una aplicación con una fuente de datos, permanece abierta mientras se trabaja con la fuente de datos, estemos en un instante dado manipulando datos de la fuente de datos o no. Por decirlo de otra forma, la puerta de acceso la hemos dejado abierta y pasaremos de una habitación a otra y viceversa cuando queramos.

Por otro lado, el trabajo con fuentes de datos desconectadas, requiere que cuando se accede a la base de datos, se recoja no sólo una copia de los datos con los que vamos a trabajar, sino que además se almacene una copia de la estructura de la tabla o tablas que hemos decidido descargar y otros datos relacionados con las tablas. En sí, es una copia exacta o una foto exacta de los datos de la tabla o tablas y de todas las características de la tabla o tablas. Por lo tanto, se abre un canal de comunicación, se recogen los datos para trabajar con ellos y se cierra nuevamente el canal de comunicación. Esos datos se almacenan en memoria, y se trabaja por lo tanto, con los datos de memoria, no con la fuente de datos directamente como en el caso anterior dónde trabajábamos con fuentes de datos conectadas.

La única particularidad a tener en cuenta en este último caso, es que como lógicamente haremos una copia de la estructura, propiedades, características y datos de la tabla o tablas con las que queremos trabajar de forma desconectada, no es muy recomendable por no decir totalmente prohibido, hacer una copia de cualquier tabla, ya que podemos sobrecargar la aplicación y afectar considerablemente en el rendimiento de ésta.

10.3.- DataSet, DataView y DataTable

Ya hemos hablado antes de **DataSet**, pero es bueno hacer un repaso breve no sólo a esta clase, sino a la clase **DataView** y **DataTable** también.

La clase **DataSet** contiene un conjunto de datos volcado desde un proveedor de datos determinado. Esta clase, está preparada especialmente, para trabajar con fuentes de datos desconectadas. Dentro de una clase **DataSet**, podemos establecer colecciones **DataTables** y **DataRelations**.

La colección **DataTable** contiene o puede contener, una o varias tablas de datos. La colección **DataRelation**, contiene por su parte, las relaciones entre las **DataTables**.

La clase **DataView** por otro lado, nos permite crear múltiples vistas de nuestros datos para presentar los datos de su correspondiente **DataTable** posteriormente. Adicionalmente a esto, también podemos ordenar y filtrar los datos, buscar y navegar un conjunto de datos dado, etc.

La clase **DataTable**, nos permite representar como ya hemos adelantado, una determinada tabla en memoria para que podamos interactuar con ella.

10.4.- Ejemplo de conectividad de datos con DataSet

Para que veamos como conectarnos a una fuente de datos, en nuestro caso, Microsoft SQL Server 2000, escribiremos el siguiente ejemplo:

```
Imports System.Data

Public Class Form1

    Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
        'Preparamos la cadena de conexión con la base de datos
        Dim Conexion As String = "Data
Source=BURGOS;Database=PRUEBAS;Integrated Security=True"
        Dim MiTabla As DataTable
        Dim MiColumna As DataColumn
        'Declaramos el objeto DataSet
        Dim MiDataSet As New DataSet()
        'Ejecutamos el comando SELECT para la conexión establecida
        Dim Comando As New SqlClient.SqlDataAdapter("SELECT * FROM MiTabla",
Conexion)
        'Volcamos los datos al DataSet
        Comando.Fill(MiDataSet, "PERSONAS")
        'Liberamos el adaptador de datos,
        'pues ya tenemos los datos en el DataSet
        Comando = Nothing
    End Sub
End Class
```

```

Dim strTexto As String = ""
' Recorremos las tablas
For Each MiTabla In MiDataSet.Tables
    strTexto += "Tabla: " & MiTabla.TableName & vbCrLf & vbCrLf
    ' Recorremos las Columnas de cada Tabla
    For Each MiColumna In MiTabla.Columns
        strTexto += MiColumna.ColumnName & vbTab & _
            "(" & MiColumna.DataType.Name & ")" & vbCrLf
    Next
Next
'Mostramos la información por pantalla
MessageBox.Show(strTexto)
End Sub
End Class

```

Nuestra aplicación en ejecución es la que se muestra en la figura 10.1.

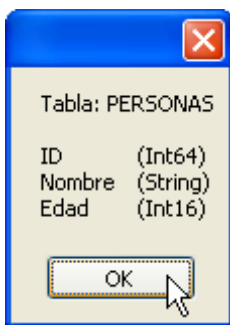


Figura 10.1: ejecución del ejemplo de conectividad DataSet.

10.5.- Recorriendo los datos de un DataSet

El siguiente ejemplo, nos muestra como recorrer los datos de un **DataSet**. Para ello, nos basamos en el ejemplo anterior y en los mismos datos de la base de datos. De esta manera, tenemos el siguiente código:

```

Imports System.Data

Public Class Form1

    Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
        'Preparamos la cadena de conexión con la base de datos
        Dim Conexion As String = "Data
Source=BURGOS;Database=PRUEBAS;Integrated Security=True"
        'Declaramos el objeto DataSet
        Dim MiDataSet As New DataSet()
        'Ejecutamos el comando SELECT para la conexión establecida
        Dim Comando As New SqlCommand("SELECT * FROM MiTabla",
Conexion)
        'Volcamos los datos al DataSet
        Comando.Fill(MiDataSet, "PERSONAS")
        'Liberamos el adaptador de datos,
        'pues ya tenemos los datos en el DataSet
        Comando = Nothing
        Dim strTexto As String = ""
        ' Recorremos las tablas
        Dim Row
        For Each Row In MiDataSet.Tables(0).Rows
            strTexto += Row(1).ToString() & " tiene " & Row(2).ToString() & "
años" & vbCrLf
        Next
        'Mostramos la información por pantalla

```

```

    MessageBox.Show(strTexto)
  End Sub
End Class

```

Este pequeño ejemplo en ejecución es el que se muestra en la figura 10.2.

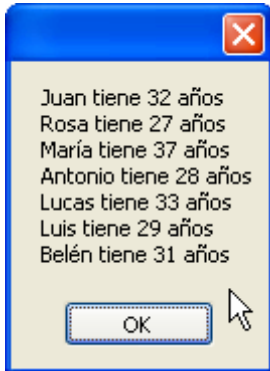


Figura 10.2: ejecución del ejemplo de acceso a datos con un DataSet.

A continuación, veremos un ejemplo de acceso conectado a datos. De esta manera, habremos cubierto la parte general del acceso de datos con Visual Basic 2005.

10.6.- Ejemplo de acceso conectado de datos

Hemos visto como conectar un **DataSet** a una fuente de datos, como volcar los datos de la fuente de datos al **DataSet**, y como trabajar con esos datos estando ya desconectados del servidor de datos. A continuación veremos un ejemplo idéntico al anterior, con la particularidad de que en este caso, estaremos accediendo al servidor en todo momento. Es decir, trataremos los datos en un ambiente conectado.

El siguiente ejemplo, nos muestra como realizar estas operaciones de forma práctica:

```

Imports System.Data

Public Class Form1

    Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
        'Preparamos la cadena de conexión con la base de datos
        Dim Conexion As String = "Data
Source=BURGOS;Database=PRUEBAS;Integrated Security=True"
        'Establecemos la conexión a utilizar
        Dim MiConexion As New SqlClient.SqlConnection(Conexion)
        'Declaramos el objeto DataReader
        Dim MiDataReader As SqlClient.SqlDataReader
        'Ejecutamos el comando SELECT para la conexión establecida
        Dim Comando As New SqlClient.SqlCommand("SELECT * FROM MiTabla",
MiConexion)
        'Abrimos la conexión
        MiConexion.Open()
        'Indicamos al objeto DataReader el comando a ejecutar
        MiDataReader = Comando.ExecuteReader
        'Declaramos la variable vacía para almacenar la información de los
campos de la tabla
        Dim strTexto As String = ""
        'Recorremos el DataReader mientras haya datos
        While MiDataReader.Read()
            strTexto += MiDataReader("Nombre") & " tiene " &
MiDataReader("Edad") & " años" & vbCrLf
        End While
        'Liberamos el Comando

```

```
Comando = Nothing
'Mostramos la información por pantalla
MessageBox.Show(strTexto)
End Sub
End Class
```

Nuestro pequeño ejemplo en ejecución es el que se muestra en la figura 10.3.

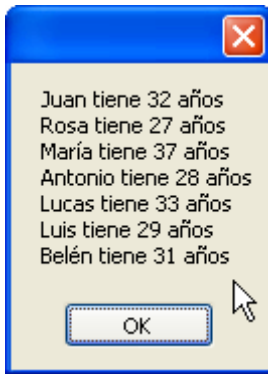


Figura 10.3: ejecución del ejemplo de acceso conectado a datos.