

Texto introductorio al lenguaje C#, centrado en la sintaxis y las construcciones que con el pueden hacerse. Al final de la obra, se incluyen dos capítulos sobre construcción de aplicaciones en Windows y en Internet, utilizando ASP.NET.

Comienza por un análisis de las características del lenguaje, e incluye las tablas de valores y definiciones reservadas para que puedan servir de referencia rápida al desarrollador, y continua hasta abordar las construcciones más complejas, como la inserción de atributos, manejo de hebras de ejecución, etc..

Todas las sentencias del lenguaje se ilustran con ejemplos explicativos de su funcionamiento, que el lector puede cortar y pegar directamente para su testeo.

Como requisitos previos, debieran de conocerse los fundamentos de la programación orientada a objetos y -al menos- un lenguaje de programación. También se requieren conocimientos del sistema operativo Windows a nivel de usuario.



INTRODUCCIÓN AL LENGUAJE C#

MARINO POSADAS



ADVERTENCIA LEGAL

Todos los derechos de esta obra están reservados a Grupo EIDOS Consultoría y Documentación Informática, S.L.

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

El almacenamiento o archivo de esta obra en un ordenador diferente al inicial está expresamente prohibido, así como cualquier otra forma de descarga (downloading), transmisión o puesta a disposición (aún en sistema streaming).

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

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

Esta obra está destinada exclusivamente para el uso particular del usuario, quedando expresamente prohibido su uso profesional en empresas, centros docentes o cualquier otro, incluyendo a sus empleados de cualquier tipo, colaboradores y/o alumnos.

Si Vd. desea autorización para el uso profesional, puede obtenerla enviando un e-mail fmarin@eidos.es o al fax (34)-91-5017824.

Si piensa o tiene alguna duda sobre la legalidad de la autorización de la obra, o que la misma ha llegado hasta Vd. vulnerando lo anterior, le agradeceremos que nos lo comunique al e-mail fmarin@eidos.es o al fax (34)-91-5017824. Esta comunicación será absolutamente confidencial.

Colabore contra el fraude. Si usted piensa que esta obra le ha sido de utilidad, pero no se han abonado los derechos correspondientes, no podremos hacer más obras como ésta.

© Marino Posadas Martín, 2002

© Grupo EIDOS Consultoría y Documentación Informática, S.L., 2002

ISBN 84-88457-55-3

Introducción al lenguaje C#

Marino Posadas Martín

Responsable editorial

Paco Marín (fmarin@eidos.es)

Coordinación de la edición

Antonio Quirós (aquiros@eidos.es)

Autoedición

Magdalena Marín (mmarin@eidos.es)

Marino Posadas (mposadas@eidos.es)

Grupo EIDOS

C/ Téllez 30 Oficina 2

28007-Madrid (España)

Tel: 91 5013234 Fax: 91 (34) 5017824

www.grupoeidos.com/www.eidos.es

www.LaLibreriaDigital.com

Índice

ÍNDICE	5
FUNDAMENTOS DEL ENTORNO COMÚN DE EJECUCIÓN: .NET FRAMEWORK.....	11
EL ENTORNO DE TRABAJO: .NET FRAMEWORK SDK	11
<i>Visual Studio .NET como entorno de desarrollo.....</i>	<i>12</i>
<i>Requisitos hardware.....</i>	<i>13</i>
<i>Sistemas operativos.....</i>	<i>13</i>
Software utilizado.....	13
<i>El papel de Internet en la concepción general de .NET.....</i>	<i>13</i>
<i>Common Language Runtime (CLR)</i>	<i>14</i>
Un nuevo perfil para el desarrollo.....	16
El lenguaje intermedio y los compiladores JIT	16
Compiladores Just-in-Time (JIT)	17
LOS ELEMENTOS COMUNES DEL SISTEMA	17
Sistema Común de Tipos.....	18
Metadatos	18
Unos tipos de datos especiales: Eventos y Delegados (<i>Delegates</i>)	20
<i>La ejecución de aplicaciones.....</i>	<i>20</i>
<i>Librerías de clases y acceso a recursos del sistema</i>	<i>21</i>
Namespaces (espacios calificados o espacios con nombre)	21
<i>Estructura de Aplicaciones: Ensamblados (Assemblies) y distribución de aplicaciones</i>	<i>22</i>
Ensamblados (Assemblies).....	22
INTRODUCCIÓN AL LENGUAJE C#.....	25

ORÍGENES DEL LENGUAJE.....	25
El Entorno de Desarrollo (IDE) de Visual Studio .NET	26
<i>Características generales</i>	29
<i>El ejemplo clásico inicial: “Hola Mundo”</i>	30
Análisis del código	31
Métodos constructores.....	31
<i>Compilación y ejecución de programas en C#</i>	32
Opciones de depuración en el IDE	35
Variantes de la función Main().....	36
Sobre <i>public</i> y <i>static</i>	36
Main() con argumentos	37
LÉXICO USADO EN C#.....	38
<i>Palabras reservadas</i>	38
<i>Operadores</i>	39
<i>Comentarios</i>	41
<i>Identificadores</i>	42
<i>Literales</i>	42
Atributos.....	43
Directivas	44
De preproceso.....	44
Directivas using.....	44
EL SISTEMA COMÚN DE TIPOS (SCT)	47
EL CTS Y LOS TIPOS DE DATOS	47
<i>Diferenciación entre la pila (ó stack) y el montón (heap)</i>	48
<i>Tipos por valor y Tipos por referencia</i>	50
<i>Boxing y unboxing (Empaquetado y desempaquetado de variables)</i>	51
Valores predeterminados	53
Formatos de salida.....	54
Cadenas de formato numérico personalizado	54
system.object: un gran tipo.....	58
Conversión de tipos	59
Conversiones implícitas	60
Conversiones explícitas.....	61
Variables colectivas (Matrices ó Arrays)	63
La clase System.Array.....	66
Las cadenas como matrices de datos <i>char</i>	67
Constantes y Variables de sólo lectura.....	68
Variables de sólo lectura (readonly).....	68
CLASES Y LA PROGRAMACIÓN ORIENTADA A OBJETOS	69
EL TRABAJO CON CLASES EN C#.....	69
<i>Campos</i>	69
Tabla de niveles de accesibilidad	70
Private.....	70
Public.....	70
Protected.....	70
Internal	71
Protected internal.....	71
Static.....	71
Readonly.....	73
Const.....	74
<i>Propiedades</i>	74
<i>Indizadores (Indexers)</i>	76

<i>Métodos</i>	77
Métodos constructores	77
Métodos destructores y el Garbage Collector	82
Métodos estándar	82
Modificadores de argumento	82
La palabra reservada <i>params</i>	84
Modificadores de métodos	85
El modificador <i>extern</i>	88
Modificadores de clase <i>abstract</i> y <i>sealed</i>	88
Sobrecarga de Operadores (Operator Overloading) y conversiones definidas por el usuario ..	90
Conversiones definidas por el usuario	92
Eventos y Delegados (Delegates)	94
Restricciones	96
Eventos	97
INSTRUCCIONES	103
INSTRUCCIONES EN C#	103
INSTRUCCIONES BÁSICAS	104
Asignaciones	105
Llamadas a métodos	106
Instrucción nula	106
Bifurcaciones condicionales e incondicionales	106
Instrucciones <i>if-else</i>	106
Instrucciones <i>switch-case-default</i> , <i>break</i> y <i>goto</i> <etiqueta>	107
Instrucciones Iterativas (Bucles)	108
Bucles <i>while</i>	109
Bucles <i>do-while</i>	109
Instrucciones <i>for-next</i>	110
Bucles <i>foreach-next</i>	110
La instrucción <i>continue</i>	113
Tratamiento de errores	113
La estructura <i>try-catch-finally</i>	114
La instrucción <i>throw</i>	116
Otras Instrucciones	117
Instrucción <i>lock</i>	117
Instrucción <i>using</i>	118
TÉCNICAS ADICIONALES EN LA CONSTRUCCIÓN DE CLASES	121
OTROS ELEMENTOS CONSTRUCTORES	121
Tipos Enumerados	122
Estructuras (<i>structs</i>)	123
Interfaces	124
Indizadores (<i>Indexers</i>)	127
Atributos	129
Atributos predeterminados y atributos de usuario	129
La clase <i>Type</i>	130
Volviendo a los atributos	136
TÉCNICAS AVANZADAS	139
SUBPROCESOS Y DOMINIOS DE APLICACIÓN	139
Dominios de Aplicación	140
Sincronización de subprocesos	143
Código no administrado (<i>unsafe</i>)	145
Manejo de elementos COM desde C#	147

COM Interop: la librería de compatibilidad con ActiveX de .NET	147
Migración Horizontal	147
<i>Metadatos y Documentación XML</i>	153
GENERACIÓN Y MANIPULACIÓN DE MÓDULOS DE ENSAMBLADO (<i>ASSEMBLIES</i>)	157
Calificación operativa de los ensamblados	159
Ensamblados públicos y privados	159
Estructura de un ensamblado	160
Contenido del manifiesto del ensamblado	161
Creación de nombres seguros para un ensamblado	166
APLICACIONES WINDOWS.....	171
EL ESPACIO DE NOMBRES WINDOWS.FORMS	171
<i>Formularios Windows</i>	174
<i>Ejemplo: Subclasificación de ventanas</i>	177
<i>Comportamiento básico de formularios estándar y MDI</i>	178
Algunas propiedades destacadas de los formularios	179
Ubicación y Tamaño	179
Botones de la Ventana	179
Aspecto Visual	179
Comportamiento	180
Otros elementos de la interfaz de usuario	180
Barras de botones y Barras de estado	182
Otras novedades en los controles de interfaz de usuario	183
Componentes	184
Un par de casos prácticos	186
Construcción de un sencillo control con herencia visual	187
Algunas consideraciones sobre el aspecto visual de los controles	189
Algunas clases específicas para el tratamiento gráfico	191
Uso de Gradientes	191
Dibujando Cadenas	192
Un ejemplo completo mediante un visor de gráficos	192
PROGRAMACIÓN CON ASP .NET	197
INTRODUCCIÓN	197
<i>El mecanismo de funcionamiento de ASP.NET</i>	198
El fundamento de ASP.NET	198
DIFERENCIAS FUNDAMENTALES ENTRE ASP .NET Y ASP 3.0	198
<i>Revisión básica de las directivas ASP.NET</i>	199
<i>Orientación a objetos</i>	200
ASPECTOS SINTÁCTICOS	200
WEB FORMS	203
CONTROLES ASP .NET	206
<i>Controles intrínsecos</i>	206
Controles HTML	206
Controles Web	207
<i>Controles de lista</i>	209
<i>Controles ricos</i>	209
<i>Controles de validación</i>	210
DATA BINDING	210
EVENTOS DE LA PÁGINA	211
SEPARACIÓN DE CÓDIGO	212
HOLA MUNDO CON ASP .NET	213
<i>Creación de una primera aplicación ASP.NET</i>	215
INTRODUCCIÓN A LOS WEB FORMS	217

INTRODUCCIÓN A LOS CONTROLES ASP .NET	219
HOLA MUNDO CON WEB FORMS	220
<i>La clase Page</i>	222
EVENTOS DE LA PÁGINA	223
PROPIEDADES DE LA CLASE PAGE	225
MÉTODOS DE LA CLASE PAGE	227
<i>El atributo Code-Behind y los controles de usuario</i>	228
CODE-BEHIND	228
<i>Utilizando los controles Web de la página</i>	230
<i>Tratamiento de eventos</i>	232
CONTROLES DE USUARIO.....	234
<i>La directiva @Control</i>	238
<i>Creación de componentes y controles de servidor</i>	239
INTRODUCCIÓN A LOS COMPONENTES .NET	239
CREACIÓN DE CONTROLES ASP .NET DE SERVIDOR	242
<i>Propiedades de los controles de servidor</i>	245
<i>Propiedades de clase</i>	247
<i>Trazas y depuración en ASP .NET</i>	250
El mecanismo de trazas	250
Trazas a nivel de página	251
Escribiendo en las trazas	253
Trazas a nivel de aplicación	256
Depuración en ASP .NET	258
Depurando con el SDK Debugger.....	259
Conclusión.....	263
BIBLIOGRAFÍA Y RECURSOS	265
BIBLIOGRAFÍA	265
RECURSOS EN INTERNET	266
<i>Genéricos sobre .NET</i>	266
<i>Sobre el lenguaje C#</i>	266
<i>Revistas gratuitas</i>	266
<i>Componentes en .NET</i>	267

Fundamentos del entorno común de ejecución: .NET Framework

El entorno de trabajo: .NET Framework SDK

Antes de comenzar el estudio del nuevo lenguaje propuesto por el danés **Anders Hejlsberg**¹ --y unos pocos colaboradores--, resulta imprescindible situarlo en el entorno adecuado, ya que no es una propuesta individual ni una iniciativa lingüística, sino un elemento más de una plataforma de desarrollo y ejecución de aplicaciones.

Dicha plataforma, que constituye el corazón de lo que comercialmente se conoce como **.NET Framework**, consta de un conjunto de ejecutables y librerías de enlace dinámico que sirven como soporte de desarrollo y de ejecución de programas escritos y compilados para ella.

Esto no significa, por supuesto, que los formatos de los ejecutables o las DLL que se utilizan en .NET no sean compatibles con Windows, sino que —en principio— se trata de que sean compatibles con cualquier sistema operativo que tenga instalado ese conjunto de librerías: hay que destacar que, al día de la fecha, .NET está disponible para las versiones de Windows posteriores a la 98/SR2, y para Linux, según anunció recientemente el mentor de esta iniciativa, el mejicano **Miguel de Icaza**, del *Grupo Ximian*, quién trabaja en un proyecto llamado *Mono*, nacido con el propósito de crear una

¹ Hejlsberg ingresó como arquitecto de software en Microsoft a finales de 1997, proveniente de Borland Co., donde había trabajado ya en el diseño e implantación de los lenguajes Turbo Pascal y Delphi. Remito al lector interesado a la entrevista que se incluye en el Apéndice B de esta obra, para algunas opiniones personales del autor del lenguaje.

versión Linux de .NET. Icaza, también anunciaba la disponibilidad del compilador para el lenguaje que es objeto de esta obra².

Por otro lado, escribir aplicaciones para .NET Framework, tampoco implica que nuestro ordenador deba disponer de nada en especial³, salvo el conjunto de ficheros (librerías de ejecución, compiladores, utilidades, etc.) que conocemos como SDK de la plataforma .NET. Dicho SDK se distribuye y puede obtenerse de forma gratuita en los sitios de descarga de **Microsoft** (**MSDN** y otros) y estrictamente hablando son lo único necesario para producir y ejecutar programas para la plataforma. No obstante, un entorno de desarrollo como Visual Studio resultará lo más adecuado, dada su integración y manejo nativo de recursos, disponibilidad de ayudas, editores de código, gestores de recursos remotos para servicios Web, etc., pero no es imprescindible.

Este paquete de desarrollo de software para .NET Framework (**Software Development Kit** o SDK), contiene la propia plataforma .NET y un conjunto de herramientas independientes, algunas de las cuales funcionan en modo comando (en una ventana MS-DOS) y otras en modo gráfico. Los elementos imprescindibles para poder desarrollar aplicaciones para .NET están contenidos en este conjunto de herramientas.

Visual Studio .NET como entorno de desarrollo

Visual Studio .NET es la nueva versión de la familia de herramientas de desarrollo de software de Microsoft, orientada hacia su nuevo entorno de programación. Visual Studio .NET, al tratarse de un entorno de desarrollo integrado (**IDE**, a partir de ahora), aúna todas las herramientas del SDK: compiladores, editores, ayuda, etc., facilitando en gran medida la creación de programas.

No obstante, esta obra se centra principalmente en las características del nuevo lenguaje C#, y aunque –en principio– no nos haría falta más que un editor de texto y los compiladores de línea de comandos para generar ejecutables, usaremos con frecuencia el IDE para elaborar ejemplos del lenguaje, por las ayudas (como la tecnología **IntelliSense**), que presenta. Una vez concluida la iniciación al lenguaje (capítulos 2 al 7, ambos inclusive), abordaremos en 3 capítulos adicionales la construcción de aplicaciones Windows, la creación de páginas y soluciones para Internet mediante ASP.NET y Servicios Web XML, y el tratamiento de datos con ADO.NET (El análisis de los mil y un recovecos que el conjunto de librerías del SDK tiene, será –esperamos– motivo de otra obra específica).

Baste comentar que en la versión más avanzada de Visual Studio (versión *Architect*), se dispone de herramientas y características que permiten gestionar prácticamente todas las necesidades que pueda encontrar un desarrollador. A medida que profundicemos en las características propias de C#, iremos revisamos algunas de las más interesantes.

Pero, antes de continuar, un recordatorio: los requisitos necesarios para la implantación

² Esta iniciativa, y otras posibles, en plataformas como McIntosh, se deben a que la Especificación Común de Lenguaje (CLS) en la que se basa .NET y el mismo lenguaje C#, han sido aprobados como estándares por la entidad de normalización ECMA, hacia finales de 2001.

³ Los requisitos de hardware se comentan un poco más adelante en este capítulo.

Requisitos hardware

En una rápida revisión de requisitos mínimos de hardware y software, la Tabla 1 muestra una lista con las características mínimas y recomendadas que debe tener el equipo en el que instalemos VS.NET:

	Mínimo	Recomendado
Procesador	Pentium II – 450 MHz	Pentium III – 733 MHz
Memoria	128 MB	256 MB
Espacio en disco duro	2 Gb.	3 Gb.

Tabla 1. Requerimientos hardware para instalar Visual Studio .NET.

Sistemas operativos

VS.NET puede ser instalado en un equipo con uno de los siguientes sistemas operativos:

- Windows XP
- Windows 2000 (se requiere tener instalado el Service Pack 2 ó superior).
- Windows NT 4.0. (se requiere tener instalado el Service Pack 5 ó superior).
- Windows Me.
- Windows 98 (SR2)

Para aprovechar todo el potencial de desarrollo de la plataforma, es recomendable usar como sistema operativo Windows XP/2000, ya que ciertos aspectos del entorno (las características avanzadas de gestión gráfica por ejemplo) no están disponibles si instalamos .NET en otro sistema con menos prestaciones.

Finalmente, de acuerdo con Microsoft, la siguiente versión del sistema —conocido en su fase beta como Windows .NET Server—, cuya aparición está prevista aproximadamente para mediados de 2002, será el primero en llevar .NET Framework instalado de forma predeterminada.

Software utilizado

Respecto a las aplicaciones utilizadas para la creación de los ejemplos de este libro, aparte de VS.NET, hemos usado Visual Studio 6.0, ya que puede convivir perfectamente en el mismo equipo en el que esté instalado .NET Framework. Como base de datos se ha utilizado SQL Server 2000 y como conjunto de herramientas adicionales Office XP.

El papel de Internet en la concepción general de .NET

La idea subyacente tras la creación de .NET Framework es el diseño de un sistema de programación y ejecución de aplicaciones en un mundo en el que dispositivos, servicios y ordenadores trabajen de

forma conjunta para suministrar a los usuarios de esos sistemas soluciones personalizadas y, al tiempo, de una gran riqueza y potencia en su capacidad de expresión, siempre con Internet y sus estándares (HTML, XML, SOAP, etc) como elemento conector y modelo de seguimiento.

De acuerdo con esto, .NET se sustenta fundamentalmente en 4 pilares básicos

- **Bloques de construcción de servicios .NET:** o sea, acceso por programa a servicios tales como almacenamiento de ficheros, servicios generales del sistema operativo, servicios extendidos (como los de datos), calendarios, Servicios Web XML, como Passport.NET (un servicio de verificación de identidad), etc.
- **Software .NET para dispositivos,** que se ejecute en cualquier máquina con acceso a Internet.
- **La experiencia de usuario,** con características como la interfaz natural, los agentes de información, y las etiquetas inteligentes, una tecnología que automatiza hipervínculos a información relacionada con palabras y frases en un documento de usuario.
- **La infraestructura .NET,** que comprende el .NET Framework, Microsoft Visual Studio.NET, los servidores .NET Enterprise, y el próximo Microsoft Windows.NET Server.

Esta última es la que típicamente se asocia con .NET, y consta del entorno común de ejecución (o CLR) y un conjunto de librerías de clases base (**Base Class Libraries**), similares en concepto a las MFC utilizadas por los programadores de C++, o las WFC que utiliza la versión Java de Microsoft.

Common Language Runtime (CLR)

El **Entorno Común de Ejecución** (*Common Language Runtime*, o *CLR* a partir de ahora), es el corazón y la pieza fundamental de toda la idea .NET. El arriba mencionado **Anders Heljsberg**, antiguo arquitecto de desarrollo de **Borland** y diseñador de lenguajes y entornos de desarrollo tan conocidos como *Turbo Pascal* o *Delphi* nos comentaba en una entrevista (accesible en el apéndice B de esta obra y en el sitio Web de la revista **Algoritmo** (www.algoritmodigital.com), que la idea central de .NET era extender la concepción que **Sun** había hecho para **Java**, consiguiendo *no un mismo lenguaje que pueda ser ejecutado en más de una plataforma, sino un lenguaje intermedio de bajo nivel (llamado MSIL, o Microsoft Intermediate Language), y alto rendimiento, que pueda ejecutarse en cualquier plataforma o dispositivo, de forma que permita que la compilación de un lenguaje de alto nivel produzca código MSIL, y posibilite una gran independencia de la semántica del código fuente, respecto al código ejecutable final.*

Ahora bien, para que un número de lenguajes distintos pueda producir el mismo código intermedio, es muy importante que todos ellos se adhieran a una normativa estándar, que proporcione un *conjunto de tipos de datos homogéneo*. Tal normativa, recibe el nombre de **Common Language Specification (CLS)** y ha sido sometida a **ECMA** para su estandarización, en un proceso que concluyó a finales de 2001. La misma entidad de normalización se ha hecho cargo de la estandarización del lenguaje que nos ocupa en este libro –proceso también concluido–, por lo que no sería extraña la aparición de nuevos compiladores de C# para plataformas como *Unix*, *McIntosh* o *Beos*, en los próximos meses.

Al conjunto de librerías que interpretan las instrucciones escritas en MSIL, es a lo que se denomina **Entorno de Ejecución** o **CLR**, y el número de lenguajes que anuncian su disponibilidad en el momento de escribir estas líneas, asciende ya a 26, incluyendo desde lenguajes tradicionales, como *Cobol*, *Pascal Orientado a Objetos*, *Fortran*, *Perl*, *Python*, *SmallTalk*, *Eiffel* o *Java*, hasta los propios de la *suite* de Visual Studio, como *Visual Basic*, *Visual C++*, *C#*, *Javascript* o la última incorporación, *J#* (un lenguaje que admite sintaxis Java y compila a código intermedio MSIL). En principio, no sería demasiado raro asistir a la aparición de nuevos lenguajes para el entorno; por otro lado, puede

aprovecharse mucho del código fuente existente en lenguajes tradicionales, como COBOL, para ser usado conjuntamente a nuevas interfaces de usuario que expongan esa funcionalidad en contextos Cliente/Servidor o Internet.

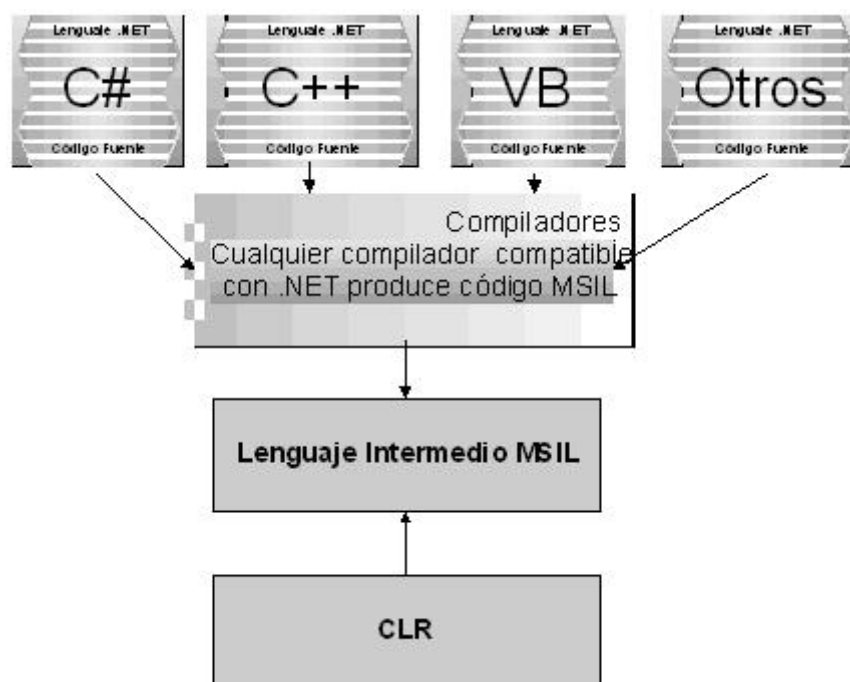


Figura 1. El CLR se encarga de gestionar los recursos solicitados por el código MSIL

¿Se trata entonces de una especificación común y una máquina virtual? No solamente. Una de las características más interesantes que aporta el CLR es la de código administrado (*managed code*). Mientras, en la actualidad el código generado por los distintos compiladores disponibles debe ejecutarse en las plataformas Windows bajo la recomendación de asumir un *comportamiento correcto*⁴, las aplicaciones compiladas para .NET se ejecutan bajo los auspicios del CLR, lo que garantiza la eliminación de un sinnúmero de errores debidos a gestión de memoria, tratamiento de punteros, optimización de recursos, etc, y todo ello, sin importar el lenguaje original en el que fueran escritas.

Otra característica fundamental del CLR es que se encarga totalmente de la gestión de memoria y los recursos asociados con ella. Esta labor la lleva a cabo el **Garbage Collector (GC)**, y elimina la necesidad que los programadores de algunos lenguajes (como C/C++), tenían de realizar esta labor mediante código. Aunque las peculiaridades del funcionamiento del **GC** nos darían material para un capítulo entero, y no son objeto de esta obra, sí que debe comentarse que su implantación tiene un cierto impacto en la forma de programar, ya que elimina la necesidad de invocar a los métodos destructores de objetos (métodos **Finalize()** de una clase), recomendándose, en su lugar, la llamada un método especial, llamado **Dispose()**, que simplemente indica al GC que los recursos correspondientes a ese objeto quedan –literalmente- a disposición del GC para cuando sea el momento de ocuparse de ellos.

⁴ Recogidas bajo el nombre de **WinLogo Program**, las características de ejecución correcta de una aplicación, son muy raramente cumplidas en su integridad por las aplicaciones, favoreciendo la inestabilidad del sistema. Cada versión de Windows, dispone de un conjunto de esas recomendaciones que Microsoft publica en la Red.

Un nuevo perfil para el desarrollo

.NET nos presenta, un perfil de desarrollo en el que la interacción de un lenguaje con el sistema operativo (a la hora de sacar algo por pantalla, conectarse a un origen de datos o realizar otra acción que no tenga que ver estrictamente con la propia clase), se suministra a través de un conjunto de librerías, provistas por el fabricante⁵, que pueden ser referenciadas y utilizadas por cualquier lenguaje .NET. Pero ninguno de estos lenguajes posee de forma nativa la capacidad de interactuar con el entorno de ejecución, al estilo, por ejemplo, de C++ o *Clipper*⁶. Esto elimina un buen número de errores, e independiza las estructuras programáticas creadas en los lenguajes .NET del sistema en el que se ejecutan.⁷

Así, si creamos un componente, éste cumplirá igualmente con todas las especificaciones del estándar, por lo que podrá comunicarse de forma directa con otra aplicación, sea cual sea la sintaxis original de ésta. ***Un lenguaje en .NET se convierte, por tanto, en una mera interfaz sintáctica para comunicarse con las librerías del entorno de ejecución.***

Esto hace que la independencia de la plataforma sea la óptima: las órdenes MSIL, serán interpretadas en función de las API's de cada sistema operativo. Incluso podrían pensarse diferentes soluciones (tipos de ventanas, cajas de diálogo, etc.) que –cumpliendo con el CLS- mostrasen distintos aspectos visuales.

El lenguaje intermedio y los compiladores JIT

Un compilador de lenguaje en .NET, admite código fuente en la entrada y produce –invariablemente- código MSIL en la salida. Nótese que MSIL es en si mismo un lenguaje, en el que podríamos escribir directamente las aplicaciones, de la misma forma que podemos hacerlo en *Ensamblador*. Es labor del creador del compilador hacer que la compatibilidad con CLS sea la máxima posible si queremos que los componentes realizados en ese lenguaje puedan interactuar con otros.

Cuando el programa es invocado para su ejecución, las instrucciones MSIL son –a su vez- convertidas en código máquina de la plataforma de ejecución que corresponda. En realidad, sólo se compilan aquellas funciones que son llamadas por el programa, por una cuestión de rendimiento y economía de recursos. No obstante, una vez llamada una función y generado su código nativo, este permanece en memoria caché para que todas las posteriores llamadas se beneficien de la rapidez aportada por esta.

*Tom Archer*⁸, define el proceso interno que tiene lugar de la siguiente forma:

- Escribimos código fuente en C#.

⁵ Y por tanto, dependientes totalmente de la plataforma. Implementar .NET en un S. Operativo, constaría fundamentalmente de 3 fases: a) Permitir la ejecución de aplicaciones .NET, mediante la creación de las librerías del entorno de ejecución, para que el código MSIL llame a las API's del sistema a que correspondan dichas instrucciones, b) Crear, al menos un compilador que produzca código MSIL (C#, al ser estándar –y por otras razones- ha sido la elección lógica para Linux) y c) Construir las herramientas adicionales del SDK (Editores, Ensambladores, Gestores de Ficheros, etc.), cosa que podría hacerse apoyándose en los dos pasos anteriores.

⁶ Las instrucciones *cin/cout* de C++, o la sentencia *USE <Database>* en *Clipper* son ejemplos clásicos de esta interacción.

⁷ Aparte de estos beneficios, se fomenta de forma notable la coherencia teórica de la implantación de la OOP en lenguajes como C#, ya que estrictamente, los métodos de una clase sólo deberían tener como objeto cambiar el estado de la misma (conjunto de datos almacenados en propiedades de la clase en un instante t, cualquiera).

⁸ Ver Bibliografía.

- Lo compilamos mediante el compilador de C# (csc.exe), produciendo un EXE.
- El compilador de C# produce en la salida un fichero con el formato PE (Portable Executable), que recibe el nombre de assembly (módulo de ensamblado) en .NET, compuesto de dos partes: la parte compilada a MSIL del código fuente, y un fragmento de sólo lectura con la cabecera estándar PE, que incluye un manifiesto (manifest), conteniendo los metadatos del módulo de ensamblado. No obstante, aquí sucede algo importante, que a veces no se comenta en detalle: cuando el compilador crea la salida, importa una función del entorno de ejecución, llamada `_CorExeMain`.
- Cuando se ejecuta la aplicación, el sistema operativo carga el ejecutable, así como cualquier vínculo dinámico (DLL) dependiente, tal como la librería `mscorlib.dll`, que exporta la función `_CorExeMain`, antes citada.
- El mecanismo de carga del sistema, bifurca al punto de entrada del ejecutable, que es incluido por el compilador. Nuevamente, este es el comportamiento estándar de cualquier PE ejecutado en Windows. Sin embargo, como el sistema operativo no puede ejecutar código MSIL, el punto de entrada no es sino un pequeño programa stub que dirige la ejecución a la función `_CorExeMain`, en la librería `mscorlib.dll`.
- La función `_CorExeMain()`, inicia la ejecución del código MSIL situado en el ejecutable.
- Como el código MSIL no se ejecuta directamente, CLR compila directamente el MSIL en instrucciones máquina, utilizando un compilador JIT (just-in-time), a medida que se van produciendo las llamadas a los métodos correspondientes. El código así producido se almacena en cachés, y sólo se recompila en caso de modificación del código fuente.

Compiladores Just-in-Time (JIT)

Respecto a los compiladores JIT, el SDK presenta 3 diferentes:

1. Generador de código fuente en tiempo de instalación (preJitter): Produce código nativo dependiente de la máquina en la que se efectúe la instalación durante el proceso de copia de ficheros en la máquina de destino. Permite despreocuparnos de cuestiones de rendimiento, ya que el módulo de ensamblado es compilado en su integridad. Es la opción preferible si no tenemos que atender a otro tipo de necesidades específicas que requieran de comunicación mediante código intermedio, y se distribuye en una herramienta de línea de comandos llamada `ngen.exe`.
2. JIT estándar de tiempo de ejecución: Es el que se ha descrito en el apartado anterior.
3. EconoJIT: Diseñado para dispositivos con pocos recursos de memoria, tales como portátiles, móviles, etc, descartan el código del caché, a medida que los recursos van escaseando.

Los elementos comunes del sistema

Las siguientes áreas centrales en las que se basa .NET, son comunes a todo el sistema:

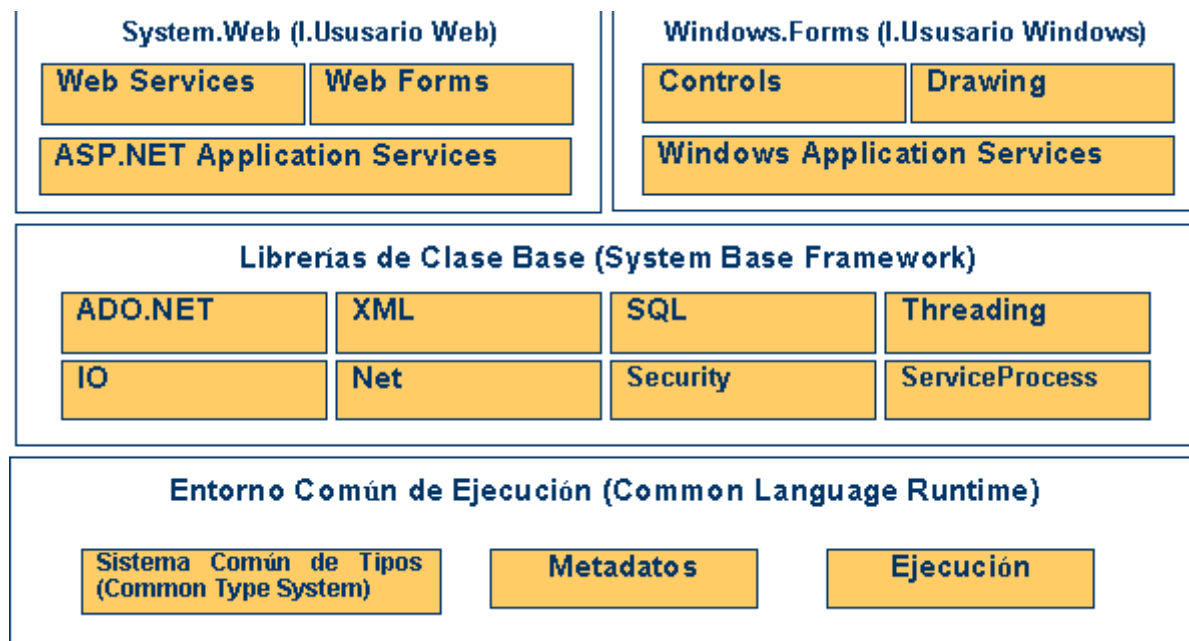


Figura 2. Estructura de Clases del .NET Framework

Sistema Común de Tipos

El Sistema de Tipos Común (*Common Type System*, ó *CTS*) es el medio que permite a .NET disponer de un único pero abundante conjunto de tipos de datos que es compartido por todos los lenguajes de la plataforma, ya que está definido en la especificación CLS.

Permite operar con todos los tipos estándar predeterminados y crear nuestros propios tipos (instancias de clases, en suma), que aporten el comportamiento que necesitemos en nuestras aplicaciones. Funcionan, en todos los sentidos, como los tipos predefinidos, con las ventajas que eso conlleva, y permiten, de hecho, extender en lo necesario el sistema.

Ya que la lista de los tipos predeterminados es larga y –después de todo- no son más que clases, hablaremos de ellos en detalle en el capítulo sobre las clases en C#. De cualquier forma, hay un hecho reseñable respecto al sistema de tipos: dado que todo .NET ha sido concebido para cumplir con un conjunto de estándares, aquellos tipos denominados **tipos base, o tipos de la plataforma** (por contraposición a los llamados tipos de usuario), han sido diseñados de acuerdo con el estándar **XML-Schemas**, que define cuáles son los tipos de datos disponibles para validación en un documento XML. Esto permite al CLR leer y convertir los tipos nativos de formato XML en tipos propios con suma facilidad, y como se verá más adelante, afecta directamente a la forma en que se gestiona el tratamiento de datos con las librerías de ADO.NET

Metadatos

Antes hemos hablado de los pasos a seguir para producir un ejecutable en C#. Uno de los puntos importantes del proceso es la producción de código autodescriptivo (*metadatos*), dentro del ejecutable (o DLL). Resulta, en su función, bastante similar a la labor que realizan en COM+ las interfaces *IUnknown* e *IDispatch*, ya que éstas permiten que otro programa pueda interrogar al componente acerca de su contenido (propiedades y métodos del tipo). La ventaja es que el conjunto de datos utilizado en la descripción de metadatos, es, por un lado, más rico en capacidad de expresión que las librerías de tipos de COM, y por otro, no puede perderse o borrarse al ir embebido en el propio

módulo, lo que paralelamente, elimina la necesidad de utilizar el Registro de Windows u otro método similar.

Al conjunto de información sobre Metadatos de un módulo de ensamblado, se le denomina **Manifiesto de Ensamblado (Manifest)** y contiene todos los metadatos necesarios para especificar los requisitos de versión y la identidad de seguridad del ensamblado, así como las definiciones del ámbito del ensamblado y las referencias a los recursos y las clases que puedan necesitar.

Esto permite al CLR conocer en tiempo de ejecución qué tipos hay que ubicar, y que métodos deben llamarse, y de esa forma preparar el entorno de ejecución adecuadamente. La información de Metadatos puede ser consultada en .NET mediante una librería llamada *Reflection*. El propio entorno de edición de Visual Studio .NET lo usa para la implantación de características como *IntelliSense*, que se han venido usando en las últimas versiones del IDE.

Lo mismo sucede con el Desensamblador de .NET (*IL Dissassembler*, o *ILDASM*), que permite abrir un fichero ejecutable o DLL, y mostrar una jerarquía de contenidos, tal y como se muestra en la Figura 3.

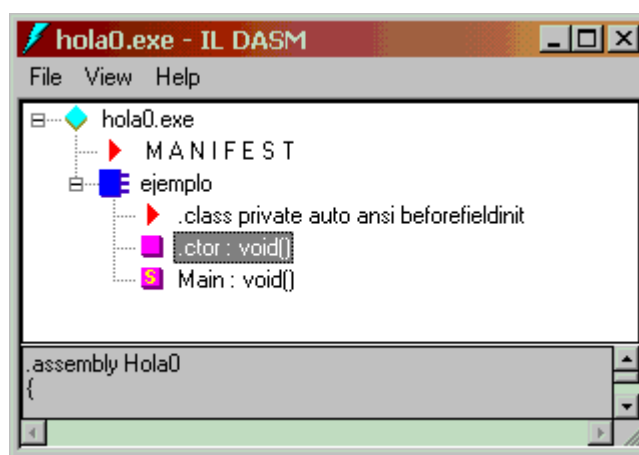


Figura 3. Ventana del explorador de ILDASM mostrando la jerarquía de un ejecutable

Incluso podemos (haciendo doble-click, en los elementos integrantes), obtener ventanas informativas de cada elemento, como se ve en la siguiente figura obtenida mediante este método aplicado al elemento **Manifest** del ejecutable:

```
.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
  .ver 1:0:2411:0
}
.assembly Hola0
{
  // --- The following custom attribute is added automatically, do
  // .custom instance void [mscorlib]System.Diagnostics.Debuggable
  //
  .hash algorithm 0x00000004
  .ver 0:0:0:0
}
.module Hola0.exe
// MVID: {19D8827D-909C-4D99-B1C7-556CF7B07F61}
.imagebase 0x00400000
.subsystem 0x00000003
.file alignment 512
.corflags 0x00000001
// Image base: 0x030F0000
```

Figura 4. Información sobre metadatos proporcionada por la herramienta ILDASM.

Como parte de los metadatos, todos los componentes administrados contienen información sobre los componentes y los recursos utilizados en su creación. El motor de tiempo de ejecución utiliza esta información para garantizar que el componente o la aplicación contienen las versiones especificadas de todo lo necesario, esto hace que haya menos posibilidades de que la ejecución del código se interrumpa debido a una dependencia inadecuada.

Unos tipos de datos especiales: Eventos y Delegados (*Delegates*)

Un evento es algo que causa la ejecución de un fragmento de código. Bajo Windows, el sistema controla las acciones del usuario (mover ratón, pulsar una tecla, etc) y genera los eventos de software correspondientes, estos toman la forma de métodos que se invocan al producirse la acción correspondiente. Extendiendo este concepto, los componentes de software se comunican con los programas que los manejan mediante eventos, permitiendo así la transmisión de información relativa a situaciones en las que puede ser necesario tomar alguna acción concreta (por ejemplo, si un componente se conecta con un origen de datos, y este provoca un error, el componente puede generar un evento que transmita la situación a la interfaz de usuario, quien será la encargada de tratarlo o de ignorarlo).

Visto de otra forma, hay quien lo define igualmente como una forma de comunicar objetos, asociada a una situación. Un objeto puede hacer una llamada a otro que este a la escucha. Esto, sin embargo tiene ciertos peligros: es posible que el objeto de destino ya no esté disponible cuando el emisor hace la llamada, o que esté ocupado realizando otra tarea, etc.

Para evitar estos posibles problemas, .NET Framework ha modificado radicalmente la forma en que esto tiene lugar. En vez de permitir que un programa llame directamente a un método de otro programa, se ha definido un objeto especial, llamado *delegate* (Delegado), encargado de dicha tarea y que hace las veces de intermediario (o delegado), entre el objeto que emite la llamada y el que la recibe. El principio es el mismo, en cierta forma, al que se usaba en programas tipo DOS, cuando utilizábamos extensores del DOS para solicitar memoria extendida. El delegado se encarga de llevar una lista de las asociaciones entre llamada y receptor, de forma que un mismo delegado puede llamar simultáneamente a un conjunto de métodos (delegados tipo *multicast*).

Lo veremos con más detalle en el apartado dedicado a *Delegados y Eventos*.

La ejecución de aplicaciones

En .NET, decimos que la ejecución de aplicaciones es asistida o administrada (*managed*), ya que se es gestionada por el CLR en todo momento. El significado profundo de esto va más allá del mero concepto de seguridad en las transacciones y traducciones de tipos de datos.

Para un programador que utilice recursos natos de .NET significa que no vamos a tener necesidad de utilizar punteros, que no es preciso realizar operaciones de código tendientes a gestionar la memoria, y que tampoco tenemos que preocuparnos por la liberación de recursos, ya que CLR suministra internamente todos los mecanismos necesarios para que eso se lleve a cabo.

Uno de los componentes fundamentales del CLR, es, por esa razón, el Gestor de Recursos (*Garbage Collector*), que mantiene las referencias de los objetos en uso, y se encarga de destruir aquellas cuyo ciclo de vida ha concluido. Corresponde con una clase más de la jerarquía y puede programarse, si queremos conocer como está operando internamente, o bien para destruir de forma explícita un recurso.

La administración automática de la memoria elimina pérdidas, así como otros errores habituales en programación. Con un código administrado se pueden utilizar datos administrados, datos no administrados o ambos en una aplicación .NET Framework. Como los compiladores de lenguajes proporcionan sus propios tipos, como tipos primitivos, no siempre se sabe (o no es necesario saber) si los datos se están administrando.

Librerías de clases y acceso a recursos del sistema

Ya hemos mencionado el carácter de interfaz de los lenguajes .NET. Para que un programa pueda hacer algo más que interactuar con sus elementos internos, necesita utilizar objetos predeterminados almacenados en la –inmensa, dicho sea de paso- librería de clases .NET.

Dichas librerías permiten el acceso a, prácticamente, cualquier cosa en el sistema operativo, desde las antiguas aplicaciones de consola o las más actuales para Windows o Internet, hasta servicios del sistema, o Servicios Web⁹. La consistencia de su creación proporciona, naturalmente, la posibilidad de crear objetos nuevos a partir de ellos mediante la herencia, y de esa forma, anular, modificar o extender la funcionalidad aportada por estos.

Namespaces (espacios calificados o espacios con nombre)

No disponemos de cifras fiables acerca de la magnitud de métodos y propiedades albergados en las librerías de clases de .NET. Vale la pena mencionar, no obstante, que son más aún que los de las propias APIs de Windows, que en su gran mayoría están contenidas en ellos. Este volumen ha hecho que sus creadores utilizaran un concepto que se usa en algunos estándares de Internet (XML, XSL, XSD), para organizar la información por categorías. Se trata del concepto de *namespace* (espacio calificado o espacio con nombre).

La idea es sellar, *marcar digitalmente*, un conjunto de datos (código, típicamente), de forma que pueda considerarse como una unidad, al tiempo que se dota a los lenguajes de la plataforma de la capacidad de hacer referencia a dichas librerías mediante dicha marca digital o *namespace*. Además, estos espacios calificados pueden organizarse jerárquicamente, de forma similar a como se organizan las jerarquías de clases contenidas en ellos, lo que ha permitido realizar una organización funcional de las clases, de forma que todas aquellas relacionadas con un cierto tipo de tareas, se agrupen juntas.

El usuario, también puede crear sus propias definiciones de espacios calificados en sus programas y, de esa forma, indicar al CLR, qué módulos de ensamblado pertenecen a la misma aplicación o cuál es la prioridad en su gestión.

Su programación será objeto de más detallado estudio en capítulos posteriores, pero, respecto a los **namespaces** predefinidos en punto .NET, la Figura 5 es una primera referencia programática para dar una idea de su organización. En extensa nota final de este capítulo, incluimos una breve descripción de la funcionalidad de cada uno de esos espacios calificados, al objeto de que sirvan de referencia rápida al programador.

⁹ Un servicio Web extiende el concepto de componente asociado a un dominio, para llevar su ámbito de trabajo a Internet.

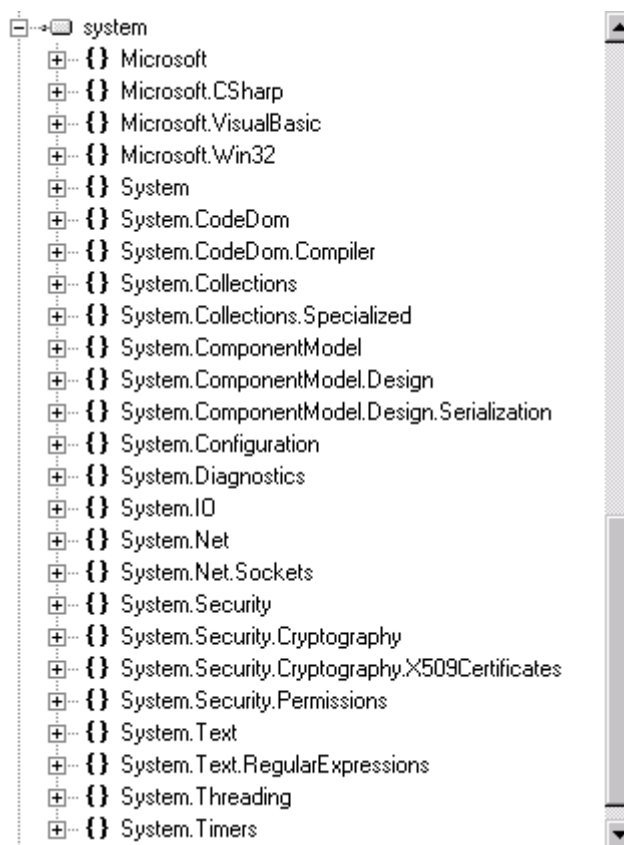


Figura 5. Jearquía de espacios de nombres de la librería System

Como puede verse en el gráfico, muchas de ellas, son –a su vez- cadenas jerárquicas, organizadas según una taxonomía funcional, que ayuda notablemente en la búsqueda de elementos programables.

Estructura de Aplicaciones: Ensamblados (Assemblies) y distribución de aplicaciones

La estructura de una aplicación .NET ha sufrido considerables modificaciones respecto a una aplicación Windows de comportamiento correcto. En primer lugar, una aplicación consta de un número de módulos de ensamblado, que no tienen en absoluto que registrarse en el sistema o depender de un mecanismo adicional para su acceso. En su versión más sencilla, la instalación de una aplicación se reduce a copiar los archivos de que consta en un directorio, y su desinstalación a borrar dicho directorio.

Se elimina con ello algunos de los problemas que habían molestado largamente a los programadores, y que recibían el nombre de “infierno de las DLL” (*DLL Hell*), debido a colisiones entre ficheros de versiones distintas, problemas debidos a la modificación del registro de Windows, etc.

Ensamblados (Assemblies)

El ensamblado se constituye así en la unidad natural de distribución de código en .NET. Es un fichero de extensión EXE o DLL, que consta de un manifiesto (*manifest*) –información de los metadatos del ensamblado-, código fuente, y uno o varios recursos de sistema (ficheros gráficos, etc), embebidos en

un único elemento. Por otro lado, una aplicación puede constar de tantos módulos de ensamblado como se desee (ya sea en forma de EXE's o DLL's).

Independientemente de esas posibilidades, podemos crear ensamblados que no pertenezcan a ninguna aplicación en concreto (usando el parámetro de compilación */t:module*), y que no contengan ningún manifiesto. Este tipo de ensamblados puede añadirse posteriormente a cualquier aplicación mediante el parámetro de compilación */addmodule*, o bien, mediante la herramienta de generación de ensamblados (*assembly generation tool*).

En el caso de que nos interese que una librería quede disponible para otras aplicaciones, la solución que aporta .NET es el llamado *Global Assembly Cache (GAC)*. Se trata de una ubicación en disco que permite situar módulos de ensamblado que tienen la característica de haber sido firmados digitalmente. Para ello se genera una clave de encriptación pública, (basado en el estándar **RSA**) llamada **strong name**, que va incrustada en el ensamblado, y que puede ser accedida por el CLR fácilmente, posibilitando al autor del módulo la asociación de elementos de seguridad, permisos de ejecución y otros.

Ya que se pretende que todo el funcionamiento del entorno sea coherente, las propias librerías de .NET se sitúan en tal ubicación, y una herramienta añadida se inserta en el explorador de Windows, para que cualquiera pueda visualizar sus contenidos, tal y como se muestra en la Figura 6.

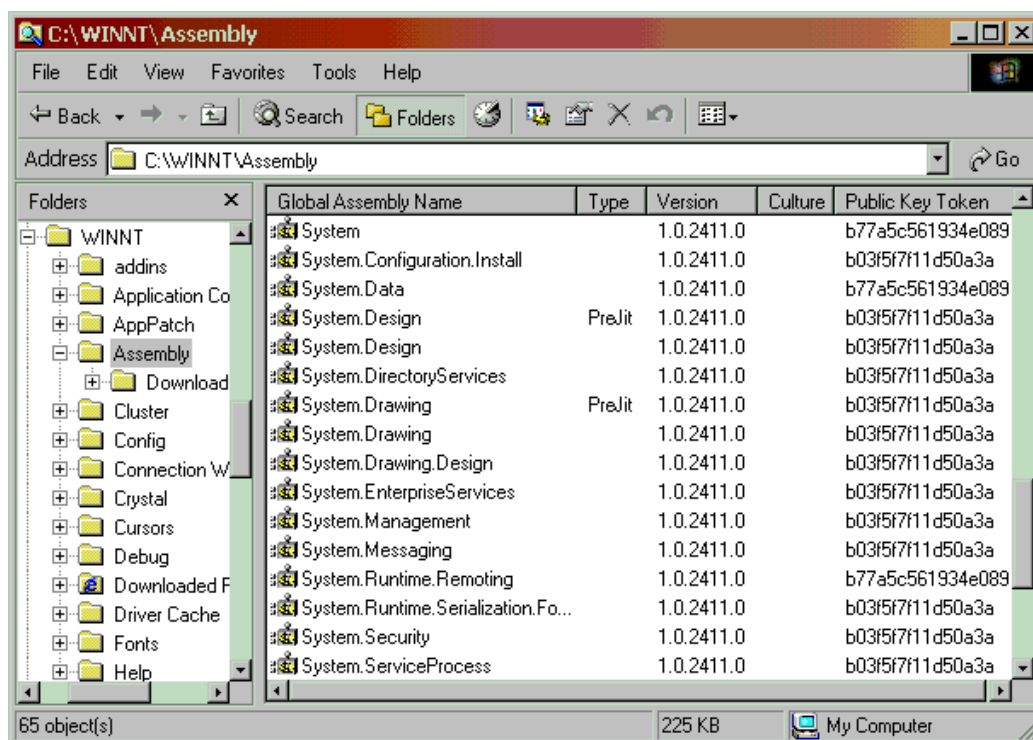


Figura 6. Explorador de Windows mostrando algunos módulos de ensamblado estándares .NET

Para todos los efectos, es como si una referencia a un módulo situado en el GAC sólo estuviera completa mediante el nombre del módulo más la clave pública. Esto permite situar en la misma ubicación módulos con el mismo nombre y distinta clave pública, por ejemplo por razones de versionado, o como puede verse en el gráfico, módulos precompilados (*pre-jitted*) para el uso desde el diseñador de Visual Studio, y otra versión en código MSIL, que pueda interactuar con otro código MSIL.¹

Introducción al lenguaje C#

Orígenes del lenguaje

Los orígenes de C# son recientes, aunque la idea se remonte, en realidad, a 1998, en palabras de su arquitecto principal, el danés *Anders Hejlsberg*. Cuando *Hejlsberg* se cambió a Microsoft, a finales del 97, su primer trabajo fue el de colaborar en la adaptación de Visual J++ a la suite de Visual Studio 6.0. Una vez concluida esa labor (en lo posible, y con las limitaciones de incorporarse en la versión 1.1, cuando gran parte del trabajo de arquitectura ya estaba diseñado), pensó más a fondo sobre la idea de .NET y el nuevo lenguaje. Trajo consigo algunos colaboradores brillantes de **Borland**, como *Peter Goldie* y *Peter Solla*, y en unión de otros arquitectos de software como *Scott Wiltamuth* y *Peter Kukol*, propuso la idea de .NET y la posibilidad de crear un nuevo lenguaje.

Ante la pregunta sobre la necesidad de un nuevo lenguaje, *Hejlsberg* aduce que en un mundo tan cambiante como este, una oportunidad como .NET planteaba la posibilidad de poder crear algo realmente nuevo (lo de menos es la sintaxis), sin tener que cargar con elementos de *compatibilidad hacia atrás*, que tanto han perjudicado el desarrollo de otros lenguajes de programación. Afirma que quería una versión *limpia* de C++/Java, que tuviese todas las ventajas posibles de estos lenguajes, e incluso aprendiese lo bueno de otros (como *Haskell*, o *Visual Basic*) y evitase algunas incoherencias, complejidades y penalizaciones en rendimiento que se observan en los lenguajes citados.

El resultado de todo ese proceso ha sido C#. Aunque el nombre iba a ser inicialmente J++, por su similitud sintáctica, debido a que en muchos de los elementos conceptuales los había heredado más de C++ que de Java, se decidió por denominarlo así, indicando la # (almohadilla), que era una extensión de la idea de C++, a 4 +, que los americanos pronuncian **C-Sharp**, por similitud con la notación musical moderna.

El Entorno de Desarrollo (IDE) de Visual Studio .NET

Antes de abordar el lenguaje en sí, comentaremos algunas de las características más notables del nuevo IDE, que posee un alto grado de modularidad y personalización, muy superior al que se incluía con la versión 6 de Visual Studio. Se han añadido capacidades totalmente nuevas, como la creación de Macros (al estilo de Microsoft Office), una tecnología Intellisense mejorada, editores de todo tipo (de texto, de ficheros HTML, CSS, XML XML-Schemas y XSL, de iconos, de *bitmaps*, cursores, recursos, informes, código de *script*, etc.), así como soporte completo para los servicios Web, y nuevos tipos de aplicaciones como los servicios Windows, aplicaciones ASP.NET, y un largo etcétera.

El nivel de convivencia con la versión anterior es total, ya que están basados en tecnologías completamente distintas y no existe interferencia entre ambas (a diferencia de lo que sucedía entre las versiones 5 y 6 de Visual Studio). Por lo demás, en este libro se ha utilizado la versión **Arquitect** de **Visual Studio .NET** en castellano, que incluye una potente y bastante completa ayuda en línea, totalmente traducida, como puede comprobar el lector en la Figura 7. Podemos encontrar tutoriales, ejemplos básicos en todos los lenguajes soportados directamente, y otro abundante conjunto de ejemplos-solución (algunos completos, como *Fitchmaker*, o *Duwamish Books*, que disponen de todas las características de una aplicación Cliente/Servidor, con acceso a Internet, estrategias de diseño de la aplicación, etc.).



Figura 7. Pantallas de ayuda del IDE de Visual Studio .NET

Además de lo anterior, varios asistentes completan las utilidades del IDE, entre los que caben destacar los de conexiones con orígenes de datos y - un conversor de aplicaciones-para Visual Basic-, que, si bien nunca llega a resultar de la utilidad anunciada (que, oficialmente, es del 95% del código fuente), realiza una extraordinaria labor de correspondencia entre los elementos propios de las aplicaciones VB6 y las de VB.NET, y proporciona un alto grado de compatibilidad a través de una librería específica, llamada **VB6.Compatibility**.

Desde el IDE de Visual Studio .NET, como es natural, también podemos controlar todas las opciones de configuración y ejecución (incluida la lista de argumentos en línea de comandos). A través de la ventana de Opciones de una aplicación, se nos ofrecen (como muestra la Figura 8), todas las

características configurables, incluyendo el objeto inicial de la aplicación, el tipo de aplicación, los argumentos de línea de comandos, etc.

Si las características de configuración implican referencias activas, como en el caso de necesitar componentes adicionales de la librería, u otros de creación propia, el Explorador de Soluciones dispone de la opción Agregar Referencia, donde podemos seleccionar entre las ofrecidas en la ventana de selección, o navegar a un directorio en busca de una referencia no listada por defecto.

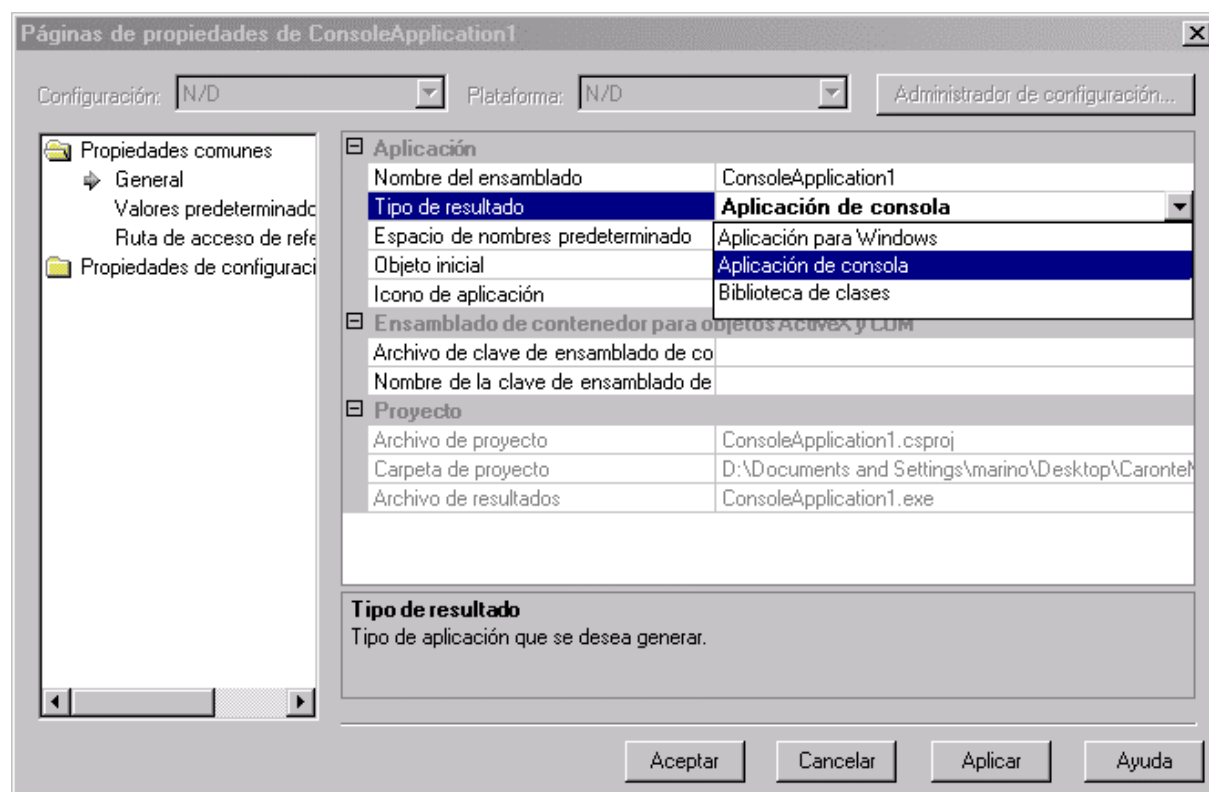


Figura 8. Ventana de configuración de aplicaciones en Visual Studio .NET

Podemos decidir, igualmente, multitud de nuevas opciones pertenecientes a todos los elementos de que consta la aplicación y/o solución que estamos construyendo: el icono asociado de la aplicación, el nombre del ensamblado principal (y sus ubicaciones en tiempo de desarrollo), los valores predeterminados respecto a directorios adicionales, llamado rutas de acceso de referencias, que muestra los directorios en los que se va a buscar cuando se cargue el proyecto para ensamblados a los que el proyecto hace referencia. La configuración es específica del proyecto, equipo y usuario para los que está establecida. Una especificación de ruta de acceso relativa se convertirá en ruta de acceso completa y se almacenará como tal. Se da por supuesto que las rutas de acceso al directorio del proyecto son rutas de acceso relativas.

Nota: antes hemos comentado que .NET permite la utilización de componentes COM en las aplicaciones .NET. Aunque tocaremos el tema más adelante, conviene indicar aquí, que la forma en la que esto se lleva a cabo, es casi transparente si utilizamos el IDE, puesto que al hacer referencia a un objeto COM, se nos pregunta si deseamos que se genere un **proxy** de comunicación con el componente, y en caso afirmativo, se crea uno para permitir la comunicación entre ambos y se incluyen en el código los enlaces necesarios. Esto incrementa al grado de compatibilidad con aplicaciones bien construidas siguiendo la metodología multicapa, y se lleva a cabo mediante una librería denominada Interop.

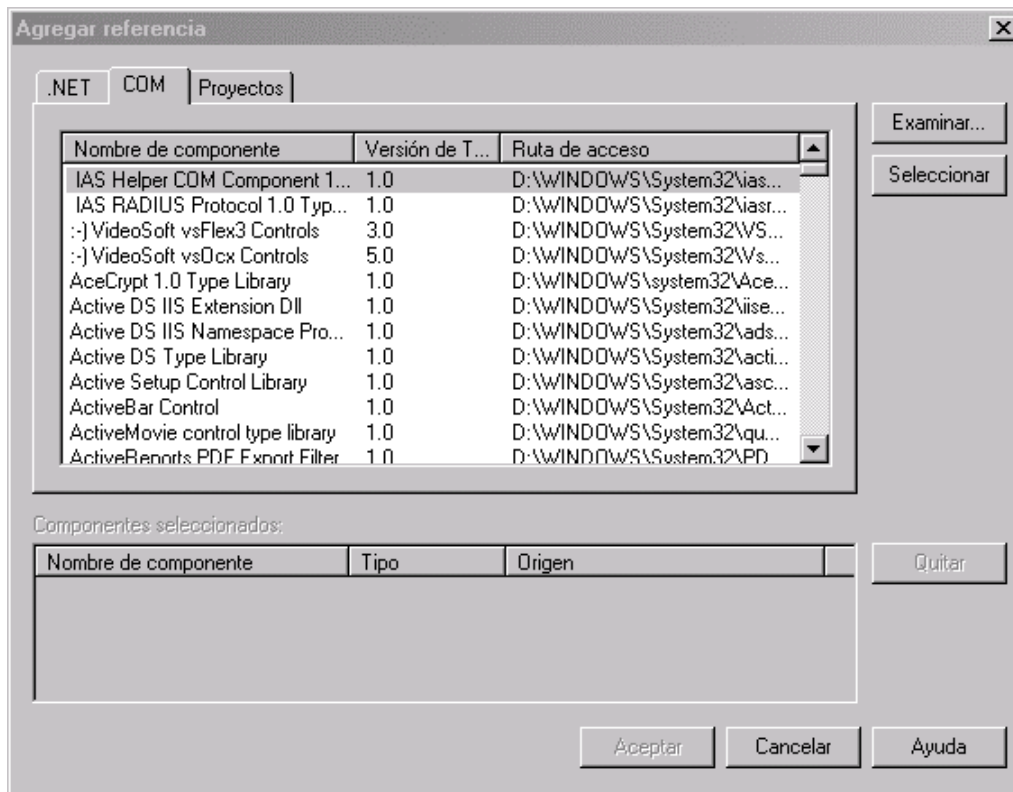


Figura 9. Ventana de referencias mostrando los objetos COM disponibles en el sistema

Conviene recordar, así mismo, que en la versión Architect que hemos usado, se incluye una versión de **Microsoft Visio** (que, aunque no sea parte del IDE, sí que está integrada con él), que es capaz de generar bases de datos a partir de un modelo entidad-relación y también de realizar la operación inversa, o sea, dada una base de datos, es capaz de inferir el modelo entidad-relación correspondiente. También dispone de la capacidad para generar clases a partir de modelos definidos siguiendo el estándar UML, para los tres lenguajes principales de Visual Studio .NET: Visual Basic.NET, C# y C++.

Esta versión de Visio, recoge las actividades principales definidas en el estándar UML, a saber: *Diagramas de Actividad*, *Colaboración*, *Componentes*, *Implementación*, *Secuencia*, *Gráficos de Estado*, *Estructuras Estáticas* y *Casos de Uso*. Se recogen, por tanto, todas las actividades de programación, y se pueden generar informes automáticos y semiautomáticos, -en varios formatos- para el soporte escrito del diseño. En la Figura 10 vemos la ventana de definición de eventos, con los tipos disponibles.

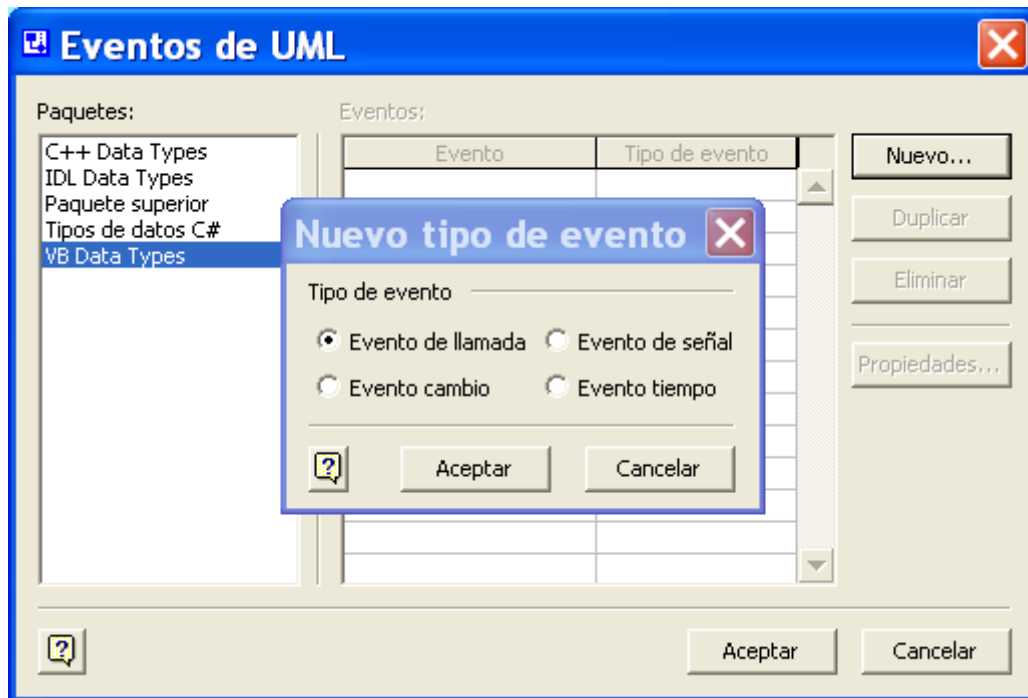


Figura 10. Ventana de definición de eventos para esquemas UML en Microsoft Visio

Características generales

Como hemos visto, existen características comunes a todos los lenguajes del entorno .NET, que lógicamente, ha heredado C#, y otras fruto de la propia concepción del lenguaje, sus herencias de otros lenguajes y sus mejoras. Respecto a las características heredadas cabe destacar:

1. Unidad del sistema de tipos, con presencia de tipos *string* y *bool* para la representación de cadenas y valores booleanos respectivamente (C++ no disponía de tal posibilidad). El tipo *currency* desaparece en detrimento del tipo *decimal*, que permite operaciones con reales de 128 bits, supliendo con creces esa necesidad. También deja de existir el tipo *variant*. Todos los tipos tienen un tamaño fijo, independiente del compilador o plataforma y establecido en el *Common Type System (CTS)*.
2. Exquisita implantación de la OOP, sin excepciones. En C#, todo son objetos, incluyendo los tipos predefinidos. Se admite la herencia simple de clases, y la múltiple de interfaces. Todos los objetos derivan de un objeto base: *System.Object*. La palabra reservada *internal* permite establecer ámbitos de acceso para los módulos de ensamblado.
3. Gestión de eventos segura y multipropósito, gracias a los *delegates*.
4. Tratamiento unificado de excepciones, mediante estructuras *try-catch-finally*.
5. Un conjunto completo de directivas para el compilador, posibilidad de establecer atributos que modifiquen comportamientos, describan tipos, controlen versiones, etc.
6. Gestión automatizada de memoria mediante el *Garbage Collector*.

Como características propias del lenguaje podríamos citar (aunque algunas se encuentren también en otros lenguajes):

1. Sintaxis muy similar a Java y C++, pero simplificada respecto a C++, con un juego de operadores abreviado. Nuevas estructuras de control, como el bucle ***foreach*** para recorrer colecciones. Incorpora el concepto de **Property** que existía en Visual Basic para normalizar el acceso a las propiedades de una clase.
2. Sobrecarga: C# (a diferencia de otros lenguajes del entorno, como Visual Basic, o de fuera del entorno, como Java), permite la sobrecarga de operadores. Dicho proceso es vigilado por el compilador para que los operadores con opuesto deban redefinirse por parejas (< y >, ó == y !=). También pueden sobrecargarse otros operadores, como el [] de acceso a elementos de un array, permitiendo el trabajo con tipos que se comporten como colecciones de objetos, a través del concepto de ***indexer*** (indizador).
3. Comprobación de rango de acceso en Arrays, de conversión de tipos, de inicialización de variables, y de control de desbordamiento (***overflow***)
4. Código autocontenido, sin necesidad de ficheros adicionales de cabecera o extensiones complementarias, como el lenguaje IDL de definición de interfaces para COM.
5. Posibilidad de incluir en un bloque ***unsafe { }*** código fuente estándar (no administrado) de C++, donde pueden manejarse punteros, gestionar directamente memoria, etc.
6. Posibilidad de llamadas a funciones de API's nativas mediante ***PInvoke (Platform Invocation)***. Puede usarse esta característica para la comunicación entre objetos COM y .NET.

Aparte de las citadas, existen un buen número de características adicionales interesantes, que analizaremos con más detalles en capítulos siguientes.

El ejemplo clásico inicial: “Hola Mundo”

Vamos, por fin, a comenzar por el programa que ya se ha hecho tradicional como punto de partida, desde la publicación de la obra “Programación de Windows”, de Charles Petzold, una salida por la consola con el texto “Hola Mundo”:

El código es el que muestra el Código fuente 1.

```
using System;

class HolaMundoCS
{
    public static void Main()
    {
        Console.WriteLine("Hola desde C#");
    }
}
```

Código fuente 1

Y, como es de esperar, produce una salida en una ventana del DOS (Consola), con el texto indicado entre paréntesis por el método *WriteLine* (“Hola desde C#”).

Análisis del código

Vamos a analizar el código fuente para irnos familiarizando con los mecanismos típicos de un programa en C#. La primera declaración (**using**) indica que vamos a utilizar el espacio calificado **System**, que es el básico para todas las aplicaciones que requieran algún servicio del sistema. Note el lector que la declaración podría haberse eliminado si en el código hubiéramos hecho referencia a la jerarquía completa, esto es, si hubiéramos escrito lo que muestra el Código fuente 2.

```
System.Console.WriteLine("Hola Mundo");
```

Código fuente 2

Esto es así porque, como se ha mencionado antes, el **CLR** pone a disposición de cualquier aplicación todos los servicios de sistema operativo a través de un conjunto jerárquico de clases con sólo hacer referencia a ellos. Utilizamos el concepto de **Namespace** exclusivamente para facilitar la escritura. Esto es, usamos la referencia a **System** como *alias* para evitar tener que repetir la palabra **System** cada vez que nos queramos referir a una clase contenida en dicho espacio de nombres.

También es interesante recordar que –por defecto– el programa no precisa de ninguna declaración *namespace* propia (de usuario), ya que nuestra aplicación consta un único módulo, que es totalmente autocontenido. La sentencia **using**, por otra parte, sólo tiene sentido para referencias a espacios con nombre, y no a clases (como el objeto **Console**, por ejemplo). Como nuestra aplicación utiliza una única salida de texto por el dispositivo por defecto (Consola), no es preciso hacer ninguna alusión a Windows ni a sus servicios. La consola está representada en dicho espacio de nombres por el objeto **Console** que dispone de métodos para escribir texto en la consola por defecto, y también para recoger entradas de datos del teclado.

Respecto a los programas sin interfaz de usuario, hemos de recordar que hay casos en los que la labor de un programa es totalmente desatendida (esto es, no dispone de ninguna interfaz de usuario), ya que sus salidas no van dirigidas a un usuario en concreto, sino a un fichero, tal y como ocurre muchas veces con los componentes.

La segunda línea contiene otra declaración importante. *Todo el código de un programa en C# debe estar incluido en una clase*. En C# utilizamos la palabra reservada **class** y las llaves **{}** para delimitar el contenido de una clase. En nuestro caso, esta clase está compuesta por una única función, que hace las veces de función principal: *Main()*. Dicha función esta precedida de 3 calificadores: de ámbito (**public**), de acceso (**static**) y de valor de retorno (**void**).

Métodos constructores

Resulta curioso comprobar que la clase que hemos creado no dispone –en apariencia– de un método constructor. Esto, realmente no es cierto, ya que toda clase debe de contener un método constructor, según uno de los principios fundamentales de la OOP. Lo que sucede, es que si una clase en C# no dispone de un método constructor explícito, el compilador genera uno por defecto, de forma automática, llamando al constructor del objeto básico del que heredan todos los demás: **System.Object**. Podemos comprobarlo viendo el código MSIL generado después de la compilación tal y como aparece en la figura adjunta.

Esto, tiene implicaciones importantes: cualquier clase que creamos hereda implícitamente del objeto **System.Object**, lo que significa que también hereda los métodos definidos en ella. Volveremos más adelante sobre este objeto, especialmente, al abordar el tema de la **Sobrecarga (Overloading)** y

veremos cómo se pueden sobrescribir dichos métodos para obtener funcionalidad adicional (**Overriding**).

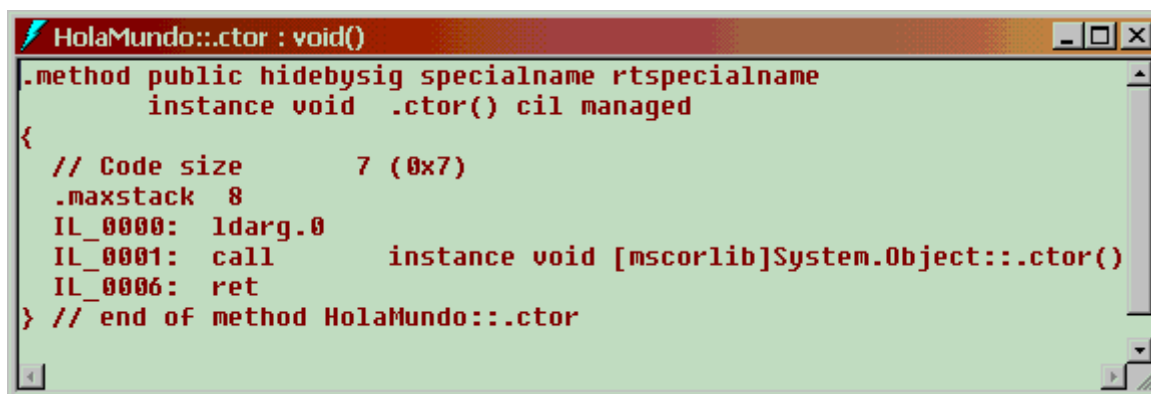


Figura 11. Código MSIL de la llamada al método constructor.

Como vemos, el compilador ha añadido el método constructor (**.ctor**), que, a su vez, llama al método constructor de **System.Object** mediante la instrucción **call** seguida de la referencia al objeto y método llamados. De esa forma, -como ya hemos dicho- se garantiza que cualquier objeto disponga de los métodos correspondientes al objeto base, tales como el método **ToString()**, que devuelve una cadena informativa con el nombre del objeto (de forma predeterminada).

Compilación y ejecución de programas en C#.

El compilador de C# admite como argumentos el nombre del programa fuente y las referencias a las librerías necesarias, más un amplio número de opciones de compilación (ver tabla adjunta), pero el modo de hacer la llamada dependerá del entorno de desarrollo que estemos usando, tal y como veremos más adelante. Quizá una de las novedades más sorprendentes de todo el entorno .NET sea esta especie de “vuelta a las opciones de compilación”, que habíamos abandonado prácticamente con las versiones anteriores de Visual Studio.

Tabla de Opciones del Compilador de C#	
Ficheros de Salida	
/out:<file>	Nombre del fichero de salida (tomado del primer fichero fuente, si no se especifica lo contrario)
/target:exe	Construye un ejecutable de consola (por defecto) (forma corta: /t:exe)
/target:winexe	Construye un ejecutable Windows (forma corta: /t:winexe)
/target:library	Construye una librería (forma corta: /t:library)
/target:module	Construye un módulo que puede añadirse a otro assembly (forma corta: /t:module)

/define:<lista de símbolos>	Define símbolos de compilación condicional (forma corta: /d)
/doc:<file>	Fichero XML de Documentación a generar
Ficheros de entrada	
/recurse:<comodines>	Incluye todos los ficheros especificados en el directorio actual y subdirectorios, dependiendo de los caracteres comodín indicados
/reference:< lista de ficheros>	Referencia a metadatos de los ensamblados especificados (forma corta: /r)
/addmodule:<lista de ficheros>	Enlaza en el ensamblado los módulos indicados
Recursos	
/win32res:<fichero>	Especifica un fichero de recursos tipo Win32 (.res)
/win32icon:<file>	Utiliza ese icono en la salida
/resource:<resinfo>	Incrusta el recurso especificado (forma corta: /res)
/linkresource:<resinfo>	Enlaza el recurso especificado al ensamblado actual (forma corta: /linkres)
Generación de Código	
/debug[+ -]	Emite información de depuración
/debug: {full pdbonly}	Especifica tipo de depuración (por defecto, completa –full- que permite enlazar un depurador a un programa en ejecución.
/optimize[+ -]	Activa las optimizaciones (forma corta: /o)
/incremental[+ -]	Activa la compilación incremental (forma corta: /incr)
Errores y Avisos	
/warnaserror[+ -]	Trata los avisos como si fueran errores
/warn:<n>	Establece el nivel de aviso (0-4) (forma corta: /w)
/nowarn:<lista de avisos>	Desactiva mensajes de aviso concretos
Lenguaje	
/checked[+ -]	Genera comprobaciones de desbordamiento
/unsafe[+ -]	Permite código <i>unsafe</i> (no administrado)

Misceláneo	
@<fichero>	Lee un fichero respuestas para opciones adicionales
/help	Muestra la ayuda del compilador (forma corta: /?)
/nologo	Suprime el mensaje de Copyright
/noconfig	No incluye automáticamente el fichero CSC.RSP (de configuración del compilador)
Avanzado	
/baseaddress:<dirección>	Dirección de memoria base para la librería a generar
/bugreport:<fichero>	Crea un fichero 'Informe de Errores'
/codepage:<n>	Especifica la página de códigos a usar al abrir ficheros de código fuente
/utf8output	Muestra los mensajes del compilador usando UTF-8
/main:<type>	Especifica la clase que contiene el punto de entrada (ignora cualquier otro posible punto de entrada) (forma corta: /m)
/fullpaths	El Compilador genera paths totalmente calificados
/filealign:<n>	Especifica la alineación de secciones en el fichero de salida
/nostdlib[+ -]	No incluye la librería estándar (mscorlib.dll)
/lib:<lista de ficheros>	Especifica directorios adicionales para la búsqueda de referencias

Tabla 2. Lista completa de opciones del compilador de C#

Si estamos trabajando de forma independiente de cualquier IDE, (por ejemplo, si empleamos un editor externo, o incluso el Bloc de Notas), grabaremos el código fuente con la extensión **.cs** (no es obligatorio, pero se recomienda por cuestiones de normalización), y, desde la línea de comandos llamaremos al compilador.

```
Csc holamundo.cs
```

Código fuente 3

Lo que generará un programa llamado **holamundo.exe**, que puede ser invocado para su ejecución como cualquier otro programa Windows. Téngase en cuenta que el *path* predeterminado deberá incluir una referencia al compilador, que se ubica en el directorio de Windows (el primer directorio dependerá de la versión): **C:\WINDOWS\Microsoft.NET\Framework\v1.0.3705**. El fichero generado se situará de forma predeterminada en mismo del fichero fuente. También podemos usar una utilidad llamada **Símbolo del sistema de Visual Studio .NET**, que lanza una ventana DOS, con la configuración correcta de directorios asociada. Dicho programa, no es más que un enlace al fichero **.bat vsvars32.bat**,

que se encuentra en el directorio *C:\Archivos de programa\Microsoft Visual Studio.NET\Common7\Tools*, y que podemos configurar según nuestras necesidades.

Si en lugar de un ejecutable, quisiéramos crear un componente (una librería), para ser usado por otros programas, incluiríamos el modificador */t:library*, que le asignaría automáticamente la extensión DLL. Posteriormente, para vincularla con un ejecutable que la utilizase, incluiríamos la referencia a la librería mediante */r:librería.dll* igualmente a como haremos cuando necesitemos otras librerías de la jerarquía .NET distintas de la principal, que se incluye por defecto (*mscorlib.dll*).

Si por el contrario, preferimos usar el IDE de *Visual Studio*, que tiene las ventajas añadidas de la ayuda sintáctica, examinador de objetos, etc, cualquier llamada a las opciones **Depurar** o **Iniciar sin depurar**, hará que, previamente, se compile el programa antes de la llamada. También podemos compilar solamente, mediante las opciones Construir (*Build*) o Construir todo (*Build All*), que dispone de capacidades de compilación incremental.

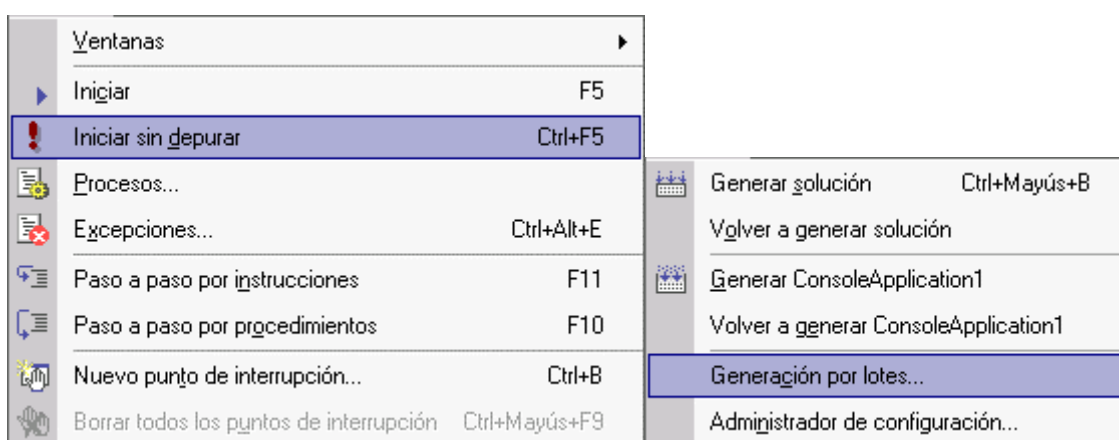


Figura 12. Opciones de compilación y ejecución en el IDE de Visual Studio .NET

Finalmente, si ejecutamos la aplicación de consola mediante la opción *Iniciar*, la ventana DOS que se abre para la ejecución, se cerrará inmediatamente después de terminada ésta, por lo que es más recomendable (en estos casos), lanzar la ejecución mediante **Iniciar sin depurar**, que añade una pausa antes de cerrar la ventana, permitiendo la visualización.

Opciones de depuración en el IDE

Comentaremos mucho más sobre el IDE al hablar de la construcción de aplicaciones en los 3 últimos capítulos, pero en caso de que lector quiera utilizarlo para reproducir los ejemplos que incluimos o para construir sus propias versiones, conviene saber que se dispone de una variadísima gama de opciones de depuración configurables, y esto, para todo tipo de aplicaciones, incluyendo las más complejas, como la depuración código SQL en servidores de datos, o de código ASP en servidores IIS.

Existen varias ventanas de depuración, para la observación del estado de las variables: las ventanas **Local** –que muestra muestran las variables locales respecto al contexto actual-, **Automático**, que muestra Nombre, Valor y Tipo de todas las variables de la instrucción actual y de la anterior, y la ventana de **Inspección de Variables**, que muestra el estado de todas aquellas variables y constantes seleccionadas para visualización. Además existe la posibilidad de ejecución paso a paso por instrucciones, por procedimientos, o hasta el final, más la depuración de procesos (incluso externos) y de excepciones. La Figura 13 muestra las ventanas de depuración principales en la ejecución de un programa, dentro de un punto de ruptura.

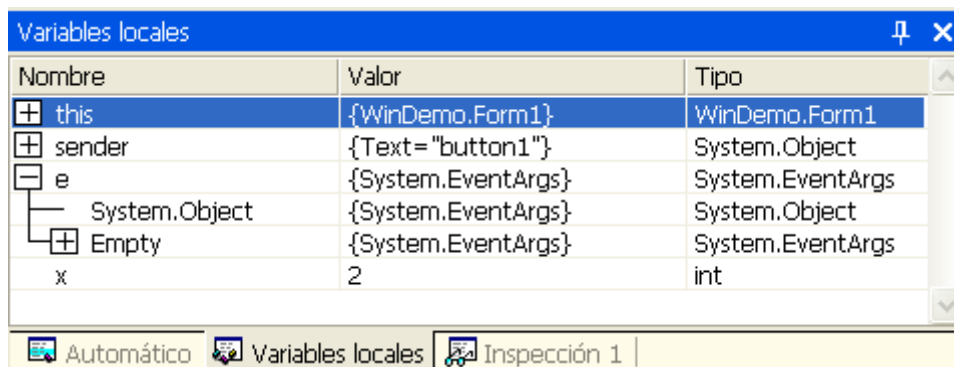


Figura 13. Ventanas de Depuración del IDE de Visual Studio

Variantes de la función Main()

Además de la sintaxis comentada, un programa en C# admite varias sintaxis (con diferente funcionalidad), para el punto de entrada de un programa. De hecho, incluso es posible disponer en una misma aplicación de más de una función Main(), siempre y cuando se indique al compilador mediante una opción cuál es el módulo que contiene el punto de entrada de la misma.

De esa forma, la función Main(), admite las siguientes *signaturas* (o declaraciones formales):

- static void Main()
- static int Main()
- static void Main(string[] args)
- static int Main(string[] args)

Como vemos, se omite en todas ellas la declaración **public**, ya que no es necesaria (los métodos **public**, simplemente se limitan a modificar el ámbito de visibilidad de un método o una propiedad, permitiendo su acceso a cualquier otra clase). Sí que lo es la declaración de accesibilidad **static** ya que el propósito de ésta es permitir el acceso a métodos sin necesidad de instanciar un objeto de la clase que los contiene. De esa forma, la función `_CoreExeMain()`, que comentamos anteriormente, puede acceder a la función `Main()` para que haga las veces de punto de entrada sin necesidad de instanciar ningún objeto de la clase contenedora.

En las definiciones que no implican valor de retorno se usa la palabra reservada **void** para indicar este hecho. Si declaramos Main() como de tipo **int** otra aplicación podría recoger ese valor como código de terminación de la aplicación.

Sobre public y static

El concepto de método estático (**static**) es fundamental en cualquier lenguaje orientado a objetos. Un método **static** puede ser llamado **de forma directa**, sin necesidad de instanciar objetos de la clase que los contiene. Basta con anteponer el nombre de la clase contenedora, seguido del método **static** objeto de la llamada. Eso es exactamente lo que hace la jerarquía de clases de .NET cuando expone al programador un enorme conjunto de métodos jerarquizados, como el método `WriteLine()` del objeto `Console`, que hemos usado antes para la salida por pantalla. Si observamos la definición del método `WriteLine()` a través del *Examinador de Objetos* de Visual Studio .NET, podemos comprobar que **todos** los métodos del objeto `Console` han sido declarados como **public static** (ver Figura 14).

Eso explica que podamos hacer referencia directa al método de salida sin instanciar un objeto de la clase *Console*. La jerarquía .NET expone su funcionalidad de esta forma en cientos de clases, algunas de las cuales reciben, además el calificativo *sealed* indicando que pueden instanciarse objetos de ellas, pero no se permite la herencia, sino sólo la utilización *cerrada* de sus propiedades y métodos (tal cual han sido definidos, sin sobrescribir sus métodos).

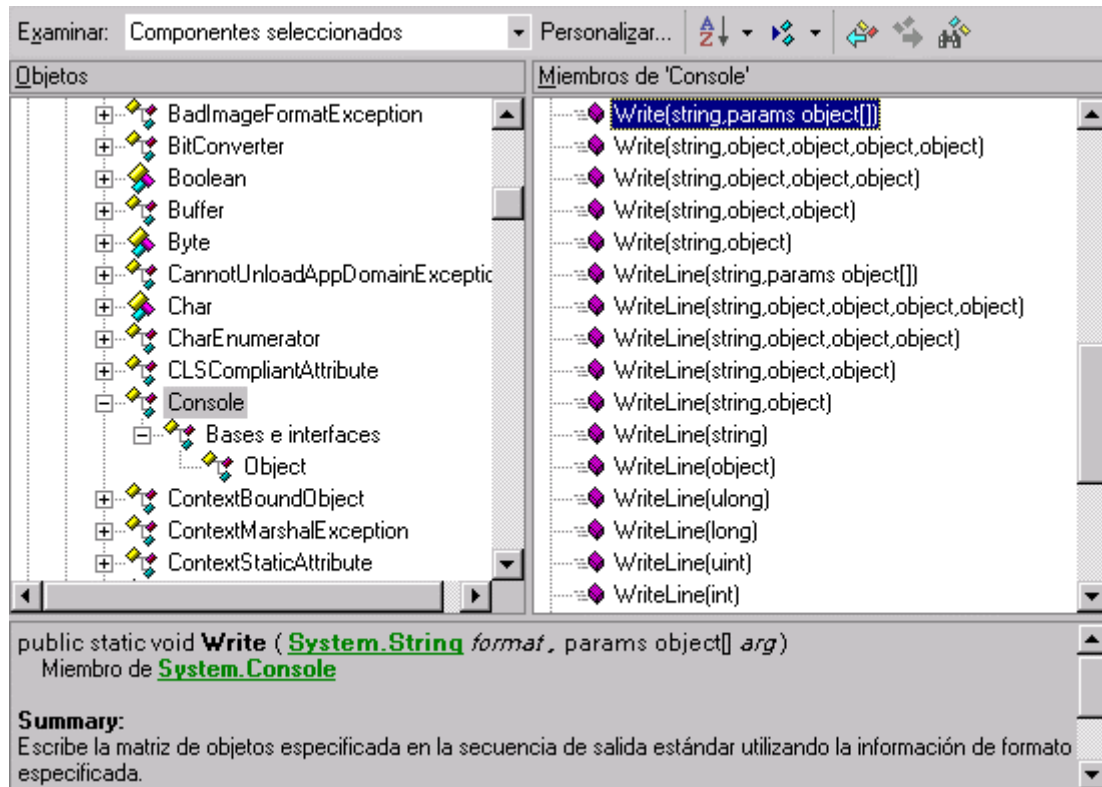


Figura 14. Examinador de Objetos de Visual Studio .NET mostrando la definición del método WriteLine()

```
public sealed class Console : System.Object
    Miembro de System
```

Summary:
 Representa las secuencias de entrada, salida y error estándar para las aplicaciones de consola. No se puede heredar esta clase.

Figura 15. Definición de la clase Console

Main() con argumentos

Al igual que sucede en C/C++, también es posible pasar argumentos a Main() desde la línea de comandos, y además en un número indeterminado. Si ese es el caso, lo indicamos en el código fuente mediante la sintaxis que se ve a continuación, y un bucle foreach puede servir para recorrer los valores y mostrarlos posteriormente. El Código fuente 4 es el código fuente correspondiente.

```
using System;

class HolaMundo1
{
    static void Main(string[] args)
```

```

{
    foreach (string argumento in args)
    {
        Console.WriteLine(argumento);
    }
}

```

Código fuente 4

La variable en línea ***argumento*** es declarada como de tipo ***string*** (los argumentos de línea de comandos siempre se aceptan como ***string***), y recorrida mediante el bucle ***foreach***, imprimiendo los argumentos en la consola predeterminada (veremos más sobre tipos de datos y estructuras de control en siguientes capítulos).

Nuestro programa (***HolaMundo1.exe***), una vez modificado y compilado, generaría la siguiente salida, pasándole como argumentos los valores “Uno”, “Dos”, y “Tres”:

```

Uno
Dos
Tres

```

Código fuente 5

Léxico usado en C#

Antes de comenzar el estudio del lenguaje en más en profundidad, conviene comentar –al menos en un enunciado general- cuáles son las partes que lo constituyen: palabras reservadas, operadores, comentarios, identificadores, literales, atributos y directivas de preproceso.

Palabras reservadas

Se denominan palabras reservadas de un lenguaje aquellos vocablos fijos que constituyen su sintaxis y que, por tanto, tienen invariablemente, un significado definido y específico en ese lenguaje. En C#, las palabras reservadas son las que se indican en la Tabla 3, según la especificación oficial del lenguaje.

Palabras reservadas en C#			
<u>abstract</u>	<u>event</u>	<u>new</u>	<u>struct</u>
<u>as</u>	<u>explicit</u>	<u>null</u>	<u>switch</u>
<u>base</u>	<u>extern</u>	<u>object</u>	<u>this</u>
<u>bool</u>	<u>false</u>	<u>operator</u>	<u>throw</u>
<u>break</u>	<u>finally</u>	<u>out</u>	<u>true</u>

<u>byte</u>	<u>fixed</u>	<u>override</u>	<u>try</u>
<u>case</u>	<u>float</u>	<u>params</u>	<u>typeof</u>
<u>catch</u>	<u>for</u>	<u>private</u>	<u>uint</u>
<u>char</u>	<u>foreach</u>	<u>protected</u>	<u>ulong</u>
<u>checked</u>	<u>goto</u>	<u>public</u>	<u>unchecked</u>
<u>class</u>	<u>if</u>	<u>readonly</u>	<u>unsafe</u>
<u>const</u>	<u>implicit</u>	<u>ref</u>	<u>ushort</u>
<u>continue</u>	<u>in</u>	<u>return</u>	<u>using</u>
<u>decimal</u>	<u>int</u>	<u>sbyte</u>	<u>virtual</u>
<u>default</u>	<u>interface</u>	<u>sealed</u>	<u>volatile</u>
<u>delegate</u>	<u>internal</u>	<u>short</u>	<u>void</u>
<u>do</u>	<u>is</u>	<u>sizeof</u>	<u>while</u>
<u>double</u>	<u>lock</u>	<u>stackalloc</u>	
<u>else</u>	<u>long</u>	<u>static</u>	
<u>enum</u>	<u>namespace</u>	<u>string</u>	

Tabla 3. Lista palabras reservadas del lenguaje C#.

Operadores

Entendemos por operador un símbolo que indica que debe realizarse una operación sobre uno o más argumentos, que reciben el nombre de operandos. A los operadores que actúan sobre un único argumento, se les llama operadores *unarios*

Categoría del Operador	Operadores
Primarios	(x), x.y, f(x), a[x], x++, x--, new, typeof, sizeof, checked, unchecked
Unarios	+, -, !, ~, ++x, --x, (T)x
Multiplicativos	*, /, %
Aditivos	+, -
De desplazamiento	<<, >>

Relacionales	<, >, <=, >=, is
Igualdad	= =
AND Lógico	&
XOR Lógico	^
OR Lógico	
AND Condicional	&&
OR Condicional	
Condicional	?:
Asignación	=, *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, =

Tabla 4. Lista de operadores en C# por orden de precedencia.

Como podemos ver, un completo juego de operadores basado principalmente en los existentes para los lenguajes C++/Java. Aunque la mayor parte de ellos, siguen las reglas establecidas para otros lenguajes, conviene establecer algunas de las reglas generales más importantes que se cumplen en su tratamiento e implantación, y que son las siguientes:

1. La precedencia de los operadores es la que se indica en la tabla adjunta.
2. Todos los operadores, excepto el de asignación (=), se evalúan de izquierda a derecha. Esto significa que se da propiedad transitiva en los operadores. O sea, que la expresión
3. $A+B+C$ es equivalente a $(A+B)+C$. Como siempre, se utilizan los paréntesis para establecer precedencia específica en cualquier expresión. Nótese, además, que en C#, se utilizan distintos símbolos para el operador de asignación (=) y el de comparación (= =), evitando errores de interpretación, como sucede en otros lenguajes, al estilo de Visual Basic.
4. El operador (.) se utiliza para especificar un miembro de una clase o estructura. El operando de la izquierda simboliza la clase y el de la derecha el miembro de esa clase. Se eliminan otros operadores presentes en el lenguaje C++, tales como ($\rightarrow$, ::), para simplificar la sintaxis.

El operador ([]) se utiliza para hacer referencia a los elementos de un array. Todos los arrays en .NET comienzan en 0.

5. Los operadores (++) y --) se utilizan al estilo de C++, como operadores de autoincremento y autodecremento. Pueden ser utilizados dentro de expresiones.
6. **new** es el operador usado para invocar métodos constructores a partir de definiciones de clases.
7. **typeof** es el operador utilizado mediante la librería **Reflection** para obtener información acerca de un tipo.
8. **sizeof** se utiliza para determinar el tamaño de un tipo estándar expresado en bytes. Solo puede ser utilizado con los llamados **value types** (tipos por valor), o miembros de clases, pero no con

las propias clases. Además, sólo pueden usarse dentro de bloques **unsafe** (código C++ embebido en un bloque **unsafe** de C#).

9. **checked** y **unchecked** se utilizan para la gestión de errores, dentro de estructuras **try-catch-finally**. Lo veremos con más detalle, en el apartado de tratamiento de errores.
10. Los operadores de asignación compuesta se utilizan para simplificar expresiones en las que un operando aparece en los dos miembros de la asignación. Por ejemplo, podemos simplificar un contador expresándolo como `x += 1`, en lugar de `x = x + 1`.

Comentarios

Los comentarios en C# son uno de los aspectos extendidos del lenguaje, respecto a sus predecesores. Aunque los comentarios en línea se hacen mediante la doble barra vertical (`//`), de manera similar a C++:

```
//Esto es un comentario
System.Console.ReadLine() //y esto, otro
```

Código fuente 6

Y los bloques de comentario, mediante la sintaxis consistente en encerrar todo el comentario entre los símbolos (`/*` y `*/`).

```
/* Esto es un comentario
según la segunda sintaxis */
```

Código fuente 7

No obstante, se han añadido posibilidades adicionales de cara a la documentación del código. Según esto, el propio IDE de Visual Studio, utiliza tres barras verticales (`///`), para incluir comentarios que pueden más tarde usarse en la confección de un fichero de texto en sintaxis XML.

```
/// <summary>
/// Resumen de la descripción de la clase Test
/// </summary>
```

Código fuente 8

Observe el lector que la información añadida se incluye dentro de dos etiquetas XML de nombre `<summary>` (sumario). Posteriormente, una opción de compilación (`/doc:<Fichero XML>`), permite que se genere un fichero adicional con dicha extensión que puede ser visualizado mediante un navegador, u otro programa adecuado. En el caso de que deseemos modificar las plantillas incluidas en el IDE para la construcción de módulos de código, se pueden modificar dichas etiquetas para que tengan el sentido y la estructura que nos interesen, y posteriormente, mediante ficheros de **Hojas de Estilo Extendidas (XSL/T)**, generar directamente formatos HTML, que puedan convertirse en

Ayudas, mediante el editor de Ayudas que también se incluye en la *suite* de Visual Studio. Volveremos sobre este tema en los capítulos finales.

Observamos aquí algo que se convierte en una constante de todo el entorno de desarrollo: XML es el estándar actual para la escritura de ficheros de datos, y es el formato que se usa siempre que es posible (veremos esto mismo asociado a los ficheros de configuración externa de una aplicación, a los datos devueltos por un servidor mediante ADO.NET, y en muchas otras situaciones).

Identificadores

Un identificador es una palabra definida por el programador, y distinta de cualquier palabra reservada del lenguaje, que se usa para especificar un elemento único dentro de un programa (clase, variable, método, constante, atributo, etc.).

Las reglas de los identificadores explicadas en esta sección se corresponden exactamente con las recomendadas por la norma **Unicode 3.0**¹⁰, con excepción del carácter de subrayado, que está permitido como carácter inicial (como era tradicional en el lenguaje de programación C), los caracteres de escape Unicode están permitidos en los identificadores y el carácter "@" está permitido como *prefijo* para habilitar el uso de palabras clave como identificadores, siempre y cuando no exista otra declaración idéntica sin dicho prefijo (sería interpretado como una declaración duplicada). Además, el carácter @, tiene una utilización especial como prefijo de cadenas de caracteres, cuando lo que se quiere indicar es que la cadena posee delimitadores que deben ser interpretados como literales, como es el caso de la barra separadora de directorios: \).

Literales

Entendemos por literal a un conjunto de caracteres alfanuméricos que representan los valores que puede adoptar un tipo básico del lenguaje. Según esto, existen tantas clases de literales como de tipos en el **Common Type System**: lógicos, enteros, reales, de caracteres, de cadena, y el literal nulo.

Algunos ejemplos serían:

Literales lógicos: Representan a los valores lógicos cierto y falso:

`true`, `false` (con minúsculas)

Literales enteros: Representan números enteros en notación decimal o hexadecimal:

`0`, `-3276`, `0x1A`

Literales reales: Representan números en formato decimal.

`0.0`, `-1000.45`, `1.23E-9` (notación científica)

Literales de caracteres: Representan un carácter individual, típicamente encerrado entre comillas simples: 'a'. También se utilizan para la representación de "secuencias de escape" (representaciones alternativas de un carácter basadas en su posición en una tabla de definiciones estándar: ASCII, Unicode, etc.)

¹⁰ Informe técnico 15, Anexo 7

En C# y en las Extensiones administradas de C++, el carácter de barra invertida hace que el siguiente carácter de la cadena de formato se interprete como una secuencia de escape. Se utiliza con las secuencias tradicionales para de aplicación de formato, como "\n" (nueva línea).

Una secuencia de escape de caracteres Unicode representa un carácter Unicode. Es importante tener en cuenta que Las secuencias de escape de caracteres Unicode se procesan en identificadores, literales de caracteres y literales de cadenas regulares. Un carácter de escape Unicode no se procesa en ninguna otra ubicación (por ejemplo, para formar un operador, un elemento de puntuación o una palabra clave).

La Tabla 5 lista las secuencias de escape Unicode.

Secuencia de escape	Nombre del carácter	Codificación Unicode
\'	Comilla simple	0x0027
\"	Comilla doble	0x0022
\\	Barra invertida	0x005C
\0	Null	0x0000
\a	Alerta	0x0007
\b	Retroceso	0x0008
\f	Avance de página	0x000C
\n	Nueva línea	0x000A
\r	Retorno de carro	0x000D
\t	Tabulación horizontal	0x0009
\v	Tabulación vertical	0x000B

Tabla 5. Secuencias de escape Unicode.

Literales de cadena: (Siempre entre comillas dobles")

"Cadena", "C:\Windows"

El Literal nulo: En C# se representa mediante la palabra reservada **null**. Se usa como valor de las variables tipo **Object** no inicializadas (que contienen una referencia nula).

Atributos

A pesar de que C# es un lenguaje imperativo, sigue las normas de los lenguajes de esta categoría, y contiene elementos declarativos. Ya hemos visto que la accesibilidad de un método de una clase se especifica mediante su declaración como **public**, **protected**, **internal**, **protected internal** o **private**. Al objeto de que los programadores pueden definir sus propias clases de información declarativa, C# generaliza esta capacidad, permitiendo anexar dicha información a entidades del programa y recuperarla en tiempo de ejecución. Tal información declarativa adicional se realiza mediante la definición y el uso de atributos.

Más adelante, el programa puede recuperar dicha información mediante la característica **Reflection** que consiste en la capacidad de un programa de utilizar una librería de clase base para averiguar lo que necesite acerca de sus propios Metadatos.

También se utilizan los atributos para definir acciones que se deben de llevar a cabo en tiempo de ejecución, como sucede, por ejemplo, en la técnica denominada **Platform Invoke**, para declarar una

función perteneciente a una librería del API de Windows, como si fuera propia y poder utilizarla. El Código fuente 9 es una muestra de ésta característica.

```
[DllImport("version.dll")]
public static extern bool GetFileVersionInfo (string sFileName,
    int handle, int size, byte[] infoBuffer);
```

Código fuente 9

En el capítulo 6 volveremos a tratar este aspecto con más detalle.

Directivas

Según la norma clásica, una directiva es una indicación al compilador del lenguaje. Estrictamente hablando, no es una instrucción ejecutable, sino algo que debe realizarse *previamente*. En C# podemos distinguir varias clases de directivas:

De preproceso

Según la documentación oficial del MSDN, “las directivas de preproceso ofrecen la capacidad de omitir condicionalmente secciones de los archivos de código fuente, con el fin de notificar errores y advertencias, así como de delimitar regiones características del código fuente. El término "directivas de preproceso" sólo se utiliza por coherencia con los lenguajes de programación C y C++. En C#, no existe un paso de preproceso individual; las directivas de preproceso se procesan como parte de la fase de análisis léxico.”

Por lo tanto no tienen el mismo carácter que en otros lenguajes clásicos en donde es costumbre su utilización. La lista siguiente recoge las más habituales (todas ellas precedidas del símbolo #):

- `#define` y `#undef`, que permiten definir y anular la definición de símbolos de compilación condicional.
- `#if`, `#elif`, `#else` y `#endif`, para omitir de forma condicional secciones de archivos de código fuente.
- `#line`, que permite controlar los números de línea de errores y advertencias.
- `#error` y `#warning`, que permiten emitir errores y advertencias.
- `#region` y `#endregion`, para marcar de forma explícita secciones del código fuente (éste último se utiliza especialmente en el editor de Visual Studio .NET)

Directivas using

Facilitan la utilización de espacios de nombres y de tipos que se encuentren en otros espacios de nombres, así como la simplificación de la escritura del código fuente.

- La directiva *using-alias* (*using-alias-directive*), permite definir un alias para un espacio de nombres o un tipo, para simplificar la escritura de código fuente.

- La directiva ***using-espacio-de-nombres*** (***using-namespace-directive***) importa los miembros de tipo de un espacio de nombres

Su ámbito se extiende a través de las declaraciones del espacio de nombres dentro de la unidad de compilación o espacio de nombres que la contiene. Las directivas **using** del mismo nivel no se afectan unas a otras y es indiferente el orden en que estén declaradas.

Adicionalmente, el mecanismo ***IntelliSense*** de Visual Studio, utiliza las instrucciones **using** para determinar qué librerías se cargan dinámicamente mientras se mantiene la edición de código, lo que permite al editor sugerir miembros de clases, identificar cierto tipo de errores, etc.

El Sistema Común de Tipos (SCT)

El CTS y los tipos de datos

En .NET Framework, entendemos por **Sistema de Tipos Común** (desde ahora, **CTS**) al conjunto de reglas y especificaciones establecido para definir, utilizar y gestionar los tipos de datos utilizados por cualquier aplicación en .NET.

El objetivo de este capítulo, es, pues, analizar el significado y la forma correcta de manipulación de dichos tipos, entendiendo que en el lenguaje C#, todo son objetos, de forma que cualquier tipo de dato que definamos, hereda algunas de sus características del objeto base del cual desciende toda la jerarquía: **System.Object**, como ya anticipamos en el capítulo anterior.

En muchos lenguajes de programación, existen dos clases de tipos distintas: los que se consideran intrínsecos al lenguaje, que –como en Java- reciben el nombre de *tipos primitivos*, y aquellos que los usuarios del lenguaje pueden definir mediante la escritura de clases, siendo, normalmente, los segundos más elaborados y complejos que aquellos, y sobre todo, teniendo en cuenta que los primeros no son realmente objetos y no derivan de ninguna otra clase, al menos de forma explícita.

Los creadores de C# abordaron el reto de hacer un lenguaje en el que todo fueran objetos (al igual que otros precedentes, como **SmallTalk**), pero que no contara con la penalización en rendimiento asociada a dicha característica. La solución consistió en crear dos categorías de tipos, en función de su ubicación en memoria: los tipos por valor (llamados *value types*), y los tipos por referencia (*reference types*). Aunque ambos puedan ser tratados como objetos si es necesario, su almacenamiento varía, permitiendo un rendimiento muy mejorado en el primer caso.

Otra ventaja que tiene un sistema de tipos común radica en la seguridad. Seguridad en la conversión de tipos y garantía de que cualquier lenguaje que utilizará el CTS podrá interpretar e interactuar con

módulos creados en otros lenguajes sin módulos de traducción especiales. Además, ciertas operaciones no están permitidas de forma implícita, como algunas conversiones, para las que es precisa una indicación explícita.

Piense el lector, lo que sucedía, por ejemplo en Visual Basic: en las versiones anteriores, podíamos llamar a funciones del API de Windows, pero como las definiciones de esas funciones habían sido originalmente hechas en C++, muchas veces se requerían adaptar los tipos de datos VB a los de C++, para que la llamada a la función no terminase en un error de ejecución. Esta situación facilita la construcción de componentes, que -más incluso que los ejecutables- necesitan interactuar con cualquier código.

La Figura 16 esquematiza la lista de tipos, según una división conceptual que se basa en la ubicación de los tipos de datos: la pila (*stack*), o el montón (*heap*)

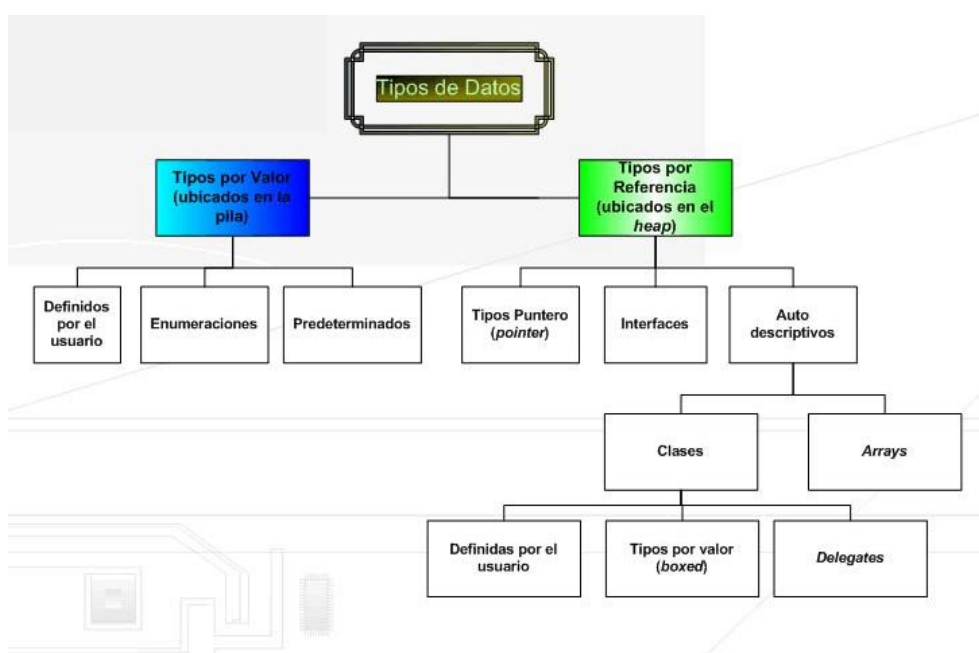


Figura 16. Esquema de los tipos de datos divididos en datos por valor y por referencia

Diferenciación entre la pila (ó *stack*) y el montón (*heap*)

La pila es un área de memoria utilizada para el almacenamiento y recuperación de valores de longitud fija. Hay que recordar que cada programa en ejecución (cada proceso, en términos de Windows), mantiene su propia pila, a la que ningún otro programa puede acceder. Con eso se garantiza la seguridad, y se previenen interacciones no controladas. Cuando se llama a una función, todas las variables locales a esa función son almacenadas (instrucciones **PUSH** de **Ensamblador**) en la pila, para ser posteriormente recuperadas (instrucciones **POP**). El típico error "*stack overflow*" se produce precisamente cuando una rutina (por ejemplo, alguna de tipo recursivo) almacena demasiadas variables en la pila y ésta se desborda (llega un momento en que no dispone de ninguna ubicación libre como respuesta a una petición de almacenamiento).

La otra ubicación posible para los datos de un programa es el montón (o *heap*), que se usa en C/C++ para almacenar valores de longitud variable (por ejemplo, *arrays* de *chars*), o datos cuyo ciclo vital superaba el de una función de manera que se puede hacer referencia a ellos fuera de la función que los define. Aunque la situación en C# es similar, la diferencia con C/C++ es que los datos de este tipo (objetos), se almacenan en una zona denominada *managed heap*, algo así como montón administrado,

de cuya gestión se encarga el Entorno Común de Ejecución (CLR). Internamente, opera de forma más eficiente, y la elección entre el uso de la pila o el montón la toma el compilador, dependiendo del tipo de dato.

```
int x = 1, y = 1; //x e y se declaran por valor
```

Código fuente 10

Esto significa que cada instrucción es separada por el compilador y se asocian los nombres X e Y a sendas direcciones de memoria, almacenando copias independientes de los datos, como vemos en la Figura 17.

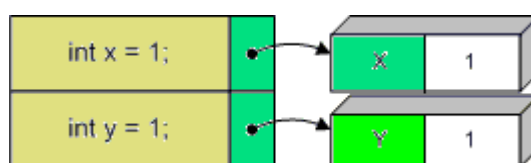


Figura 17. Asignación de tipos predeterminados en la pila

Aparte de tener una incidencia en el rendimiento (el acceso a la pila es más rápido que el acceso al montón, por razones que tienen que ver con los registros de la CPU, pero que no vamos a detallar aquí), en el primer caso, estamos hablando de tipos que no tienen una identidad establecida. Se trata de valores, de secuencias de bits situadas en memoria. El sistema se las arregla para hacer corresponder los nombres de las variables con las direcciones de memoria en las que estas se ubican, pero cada variable guarda una copia distinta del valor, aunque este sea el mismo (como en la figura 2).

Sin embargo, en el caso de los tipos por referencia, se habla de una combinación de identidad, ubicación y valores (secuencias de bits). De acuerdo con esto, dos objetos se consideran iguales *no si almacenan el mismo valor, sino cuando apuntan a la misma dirección de memoria* (que, como consecuencia, va a implicar que los valores sean los mismos, pero que supone que el cambio de estado en una de las variables de objeto significa también el cambio en todas las otras variables que hagan referencia al mismo objeto (o sea, que apunten a la misma dirección)).

La diferencia queda esquematizada en la Figura 18.

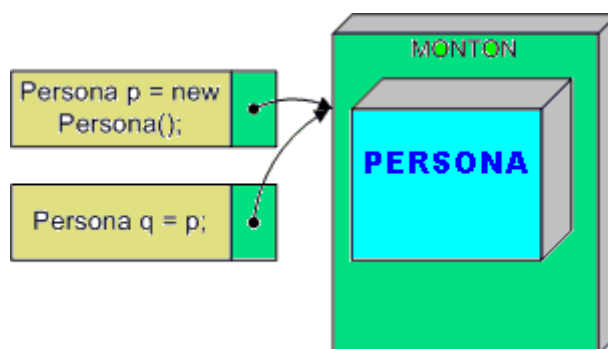


Figura 18. Asignación de variables de objeto que apuntan a la misma dirección de memoria

Nota: Esto significa que aunque se declaren variables miembro de una clase (campos) utilizando las declaraciones abreviadas (`int`, `double`, etc), como son pertenecientes a un objeto situado en el montón, ellas, a su vez, estarán también situadas en el montón. La ubicación depende de la declaración inicial y del ámbito.

Tipos por valor y Tipos por referencia

De acuerdo con esto, decimos que una variable es un tipo por valor, cuando almacena datos reales, y por tanto su valor nunca puede ser nulo. Un tipo por valor se ubica en la pila (*stack*) y puede ser declarado mediante palabras reservadas que sirven de *alias*, tales como *int*.

```
int x;
```

Código fuente 11

Esta acción selecciona un espacio en la pila de 32 bits para poder alojar cualquier valor del rango disponible para un entero. Cuando, más adelante, se asigne un dato a dicha variable, se moverá el dato a tal ubicación. Como consecuencia, cuando se pase una variable de este tipo como argumento, se está pasando en realidad una copia del valor correspondiente, y no una referencia a un objeto.

En los tipos por referencia, la situación es similar a lo que sucede en C++, con los punteros, pero la diferencia es que aquí se trata de *punteros seguros* (*type safe pointers*). El CLR, garantiza que todo tipo por referencia cuyo valor sea distinto de *Null*, apunta a un objeto de ese tipo, ubicado en el montón (*heap*). La declaración que vemos en el Código fuente 12 implica que se ha ubicado en el montón un tipo por referencia (un tipo *string*), y se ha devuelto una referencia a dicho tipo. Por lo tanto, cuando uno de dichos tipos se pase como argumento a una función, es la propia ubicación la que se pasa (y no una copia) y por tanto las modificaciones producidas afectarán al tipo original.

```
string Cadena = "Lenguaje C#"
```

Código fuente 12

Podemos comprender esta situación a través de un ejemplo simple que contenga métodos que reciban dos argumentos, uno de cada tipo, los modifiquen, y luego compruebe su valor. Es el caso del Código fuente 13.

```
using System;
class Class1
{
    static string CambiaCadena(string c)
    {
        c = "Cadena nueva";
        return c;
    }

    static int CambiaEntero(int x)
    {
        x ++;
        return x;
    }

    static void Main()
```

```

    {
        string Cadenal = "Cadena original";
        int x = 0;
        Cadenal = CambiaCadena(Cadenal);
        x = CambiaEntero(x);
        Console.WriteLine("Resultados: {0}, {1}", Cadenal, x);
        Console.ReadLine();
    }
}

```

Salida:

Resultados: Cadena nueva, 1

Código fuente 13

Como puede comprobar el lector ambas funciones realizan, básicamente la misma tarea, pero mientras la primera recibe una referencia al tipo real, la segunda recibe una copia del mismo. Por tanto, la primera resulta afectada por el proceso, pero no así la segunda.

Por lo tanto, surge inmediatamente la pregunta de cómo es posible que los tipos por valor puedan seguir considerándose (y utilizándose como) objetos. La respuesta es que tal magia es posible a través de una técnica conocida como “*Boxing y Unboxing*”.

Boxing y unboxing (Empaquetado y desempaquetado de variables)

Se trata de un mismo mecanismo en realidad, pero funcionando en sentidos distintos. Mediante la técnica de **Boxing** se consigue convertir un tipo por valor en un objeto llevándolo al montón, y devolviendo una referencia a su posición. A partir de ese momento, podremos hacer con él cualquier cosa que haríamos con cualquier otro objeto, y aplicar los mismos principios de la programación orientada a objetos. Mediante su contraria (**Unboxing**) lo que se hace es llevar nuevamente a la pila una copia del dato almacenado en el montón.

En el primero de los procesos, no se necesita ningún tipo de casting mientras que en el contrario sí lo es (en principio, podríamos convertirlo en cualquier tipo de dato).

```

//Proceso de Boxing y Unboxing
int y = 1; //Tipo por valor
object x = y; //Proceso de boxing: y se convierte a referencia para permitir la
asignación
int z = (int) x; //Proceso de Unboxing: x se convierte a entero para copiar su
valor en z

```

Código fuente 14

Como hemos podido comprobar en el ejemplo anterior, el tipos de datos *string* supone una excepción ya que se almacena en el montón, pero de cara al usuario y para la facilidad de manejo de pueden manejar como si se ubicaran en la pila.

La lista completa de tipos de datos en C# y el CTS aparece en la Tabla 6.

Nombre CTS	Alias en C#	Descripción
<i>System.Object</i>	<i>object</i>	La clase base para todos los tipos definidos en CTS
<i>System.String</i>	<i>string</i>	El tipo cadena
<i>System.SByte</i>	<i>sbyte</i>	Byte con signo
<i>System.Byte</i>	<i>byte</i>	Byte sin signo
<i>System.Int16</i>	<i>short</i>	Valor con signo de 16-bits
<i>System.UInt16</i>	<i>ushort</i>	Valor sin signo de 16-bits
<i>System.Int32</i>	<i>int</i>	Valor con signo de 32-bits
<i>System.UInt32</i>	<i>uint</i>	Valor sin signo de 32-bits
<i>System.Int64</i>	<i>long</i>	Valor con signo de 64-bits
<i>System.UInt64</i>	<i>ulong</i>	Valor sin signo de 64-bits
<i>System.Char</i>	<i>char</i>	Carácter Unicode de 16-bits
<i>System.Single</i>	<i>float</i>	Valor flotante IEEE de 32-bits
<i>System.Double</i>	<i>double</i>	Valor flotante IEEE de 64-bits
<i>System.Boolean</i>	<i>bool</i>	Valor lógico (<i>true/false</i>)
<i>System.Decimal</i>	<i>decimal</i>	Tipo de dato de 128-bits exacto a 28 o 29 dígitos—usado principalmente en aplicaciones financieras en las que se requiere un alto grado de precisión

Tabla 6. Tipos predeterminados en .NET

Nótese que el tipo de datos **string**, ubicado en la parte superior (el apartado de objetos, o **class objects**), y sin embargo en su utilización directa no hay diferencia respecto a los tipos estándar predeterminados que aparecen en el resto de la tabla. De tal forma, podemos asignar valores a variables de tipo cadena utilizando una sintaxis muy similar a la utilizada con los enteros. Esto es radicalmente distinto a la utilización que se hacía de las cadenas en C/C++ mediante arrays de *chars*.

Así, operaciones tediosas en esos lenguajes, como la concatenación de cadenas, o el análisis de caracteres dentro de las mismas es mucho más simple. Por ejemplo, la concatenación se consigue simplemente mediante el Código fuente 15.

```
string cad = "Esto es una cadena"; //Tipo string
string cad2 = "Y esto es otra";
```

```
string cad3 = cad + cad2;
```

Código fuente 15

No obstante, seguimos manejando referencias de forma interna (internamente una instancia de System.Object, se ubica en el montón). Eso significa que la primera vez que asignamos una cadena a otra, los valores apuntan a la misma ubicación (o secuencia de caracteres), pero cuando la segunda variable es reasignada a una constante de caracteres, o una expresión que pueda evaluarse como tal, se crea una copia independiente en la memoria, de forma que cada variable mantenga su conjunto distinto de datos, como podemos ver en el Código fuente 16.

```
// Asignación y uso de tipos string en una concatenación
using System;
class EjemploCadena
{
    public static void Main()
    {
        string s1 = "Una cadena";
        string s2 = s1;
        Console.WriteLine("Cadena 1: {0}", s1); //Imprime "Una cadena"
        Console.WriteLine("Cadena 2: {0}", s2); //Imprime "Una cadena"
        s1 = "Otra cadena";
        Console.WriteLine("Cadena 1: {0}", s1); //Imprime "Otra cadena"
        Console.WriteLine("Cadena 2: {0}", s2); //Imprime "Una cadena"
    }
}
```

Código fuente 16

Como podemos ver, la primera vez, los valores son idénticos, pero una vez que se vuelve a asignar el valor, cada referencia mantiene datos propios. Los literales de cadena en C# se deben encerrar siempre entre comillas dobles (lo contrario tendería a ser interpretado como un dato *char*, produciendo un error de compilación).

Valores predeterminados

Según esto, y atendiendo a su naturaleza, cada tipo de dato tiene asociado un valor por defecto, que en realidad, consiste en rellenar de ceros el espacio de memoria asignado para el. Posteriormente, al interpretar esa situación el CLR asignará los siguientes valores, dependiendo de la declaración usada para el tipo:

Tipo de valor	Valor predeterminado
<u>bool</u>	false
<u>byte</u>	0
<u>char</u>	"\0"
<u>decimal</u>	0,0M
<u>double</u>	0,0D

<u>enum</u>	El valor producido por la expresión (E)0, donde E es el identificador de la enumeración.
<u>float</u>	0,0F
<u>int</u>	0
<u>long</u>	0L
<u>sbyte</u>	0
<u>short</u>	0
<u>struct</u>	El valor obtenido al asignar los valores predeterminados a los campos de tipo de valor y el valor null a los campos de tipo de referencia.
<u>uint</u>	0
<u>ulong</u>	0
<u>ushort</u>	0

Tabla 7. Valores predeterminados para cada tipo de dato en el CTS

Formatos de salida

En C#, se incluye una variada gama de formatos posibles para la representación de valores numéricos y alfanuméricos. Al tratarse este de un problema muy típico para los programadores, hemos querido hacer especial hincapié en este punto, y detallar en un conjunto de tablas todas estas posibilidades.

Podemos dividir esta característica en dos grandes grupos, el de formatos numéricos y el de formatos de cadenas.

Cadenas de formato numérico personalizado

Los caracteres que se pueden utilizar para crear **cadenas de formato numérico personalizado** y sus definiciones, se muestran en la tabla siguiente. Téngase en cuenta que la Configuración Regional del Panel de control influye en algunos de los modelos generados por estos caracteres. Los equipos que emplean referencias culturales diferentes mostrarán modelos diferentes. La Tabla 8, tomada de la documentación oficial del producto, ilustra todas estas posibilidades.

Tabla para crear cadenas de formato numérico personalizado		
Carácter de formato	Nombre	Descripción

0	Marcador de posición cero	Si el valor al que se está dando formato tiene un dígito en la posición donde aparece el '0' en la cadena de formato, entonces ese dígito se copia en la cadena de salida. La posición del '0' que aparece más a la izquierda antes del separador decimal y la del '0' que está más a la derecha después del separador decimal determinan el intervalo de dígitos que están siempre presentes en la cadena de salida.
#	Marcador de posición de dígito.	Si el valor al que se está dando formato tiene un dígito en la posición donde aparece '#' en la cadena de formato, entonces ese dígito se copia en la cadena de salida. En caso contrario, no se almacena nada en esa posición de la cadena de salida. Hay que advertir que este indicador nunca muestra el carácter '0' si éste no es un dígito significativo, incluso si '0' es el único dígito de la cadena. Sólo mostrará el carácter '0' si es un dígito significativo del número que se muestra.
.	Separador decimal	El primer carácter '.' de la cadena de formato determina la ubicación del separador decimal en el valor con formato y se omite cualquier carácter '.' adicional. El carácter real que se utiliza como separador decimal viene determinado por la propiedad <i>NumberDecimalSeparator</i> del objeto <i>NumberFormatInfo</i> que controla la aplicación de formato.
,	Separador de miles y escala numérica	El carácter ',' tiene una doble utilidad. En primer lugar, si la cadena de formato contiene un carácter ',' entre dos marcadores de posición de dígitos (0 ó #) y a la izquierda del separador decimal, si hay, entonces el resultado tendrá separadores de miles insertados entre cada grupo de tres dígitos a la izquierda del separador decimal. El carácter real que se utiliza como separador decimal en la cadena de salida viene determinado por la propiedad <i>NumberGroupSeparator</i> del objeto <i>NumberFormatInfo</i> actual que controla la aplicación de formato. En segundo lugar, si la cadena de formato contiene uno o más caracteres ',' situados inmediatamente a la izquierda del separador decimal, entonces el número se dividirá entre el número de caracteres ',' multiplicado por mil antes de que se le dé formato. Por ejemplo, la cadena de formato '0,,' presentará una cantidad de 100 millones como 100. El uso del carácter ',' para indicar la escala no incluye los separadores de miles en el número al que se ha dado formato. Así pues, para indicar la escala de un número para un millón e insertar separadores de miles, hay que utilizar la cadena de formato '#,##0,,'.
%	Marcador de posición de porcentaje.	La presencia de un carácter '%' en una cadena de formato hace que se multiplique un número por 100 antes de que se le dé formato. El símbolo adecuado se inserta en el número en la posición en que aparece '%' en la cadena de formato. El carácter de porcentaje que se utiliza depende de la clase <i>NumberFormatInfo</i> actual.

E0 E+0 E-0 e0 e+0 e-0	Notación científica	Si alguna de las cadenas 'E', 'E+', 'E-', 'e', 'e+' o 'e-' aparece en la cadena de formato y lleva a continuación por lo menos un carácter '0', entonces se le da formato al número mediante notación científica con una 'E' o una 'e' insertadas entre el número y el exponente. El número de caracteres '0' que sigue al indicador de notación científica determina el número mínimo de dígitos para el exponente. Los formatos 'E+' y 'e+' indican que un carácter de signo (más o menos) debe preceder siempre al exponente. Los formatos 'E', 'E-', 'e' o 'e-' indican que un carácter de signo sólo debe preceder a exponentes negativos.
\	Carácter de escape	En C# y en las Extensiones administradas de C++, el carácter de barra invertida hace que el siguiente carácter de la cadena de formato se interprete como una secuencia de escape. Se utiliza con las secuencias tradicionales para de aplicación de formato, como "\n" (nueva línea). En algunos lenguajes, el carácter de escape debe ir precedido por otro carácter de escape cuando se utiliza como un literal. En caso contrario, el compilador interpreta el carácter como una secuencia de escape. Hay que emplear la cadena "\\" para mostrar "\". Hay que tener en cuenta que Visual Basic no admite el carácter de escape; sin embargo ControlChars proporciona la misma funcionalidad.
'ABC' "ABC"	Cadena literal	Los caracteres situados entre comillas sencillas o dobles se copian literalmente en la cadena de salida y no afectan al proceso de dar formato.
;	Separador de secciones	El carácter ';' se utiliza para separar secciones para los números positivos, negativos y cero de la cadena de formato.
Otros	Todos los demás caracteres	Todos los demás caracteres se copian en la cadena de salida como literales en la posición en la que aparecen.

Tabla 8

La especificación oficial de **Microsoft**, concluye este apartado comentando que:

“Hay que tener en cuenta que, para las cadenas de formato de punto fijo (las cadenas que no contienen 'E0', 'E+0', 'E-0', 'e0', 'e+0' o 'e-0'), los números se redondean en tantas cifras decimales como marcadores de posición de dígito haya a la derecha del separador decimal. Si la cadena de formato no contiene ningún separador decimal, el número se redondea al entero más próximo. Si el número tiene más dígitos que marcadores de posición de dígitos a la izquierda del separador decimal, los dígitos adicionales se copian en la cadena de salida justo antes del primer marcador de posición de dígitos”.

De acuerdo con esto, el siguiente programa formatea la salida de valores numéricos dependiendo de la cadena de formato, tal y como vemos en el Código fuente 17.

```

using System;

class Class1
{
    static void Main()
    {
        Console.WriteLine("{0:C} \n", 2,5);
        Console.WriteLine("{0:D5} \n", 25);
        Console.WriteLine("{0:E} \n", 250000);
        Console.WriteLine("{0:F2} \n", 25);
        Console.WriteLine("{0:F0} \n", 25);
        Console.WriteLine("{0:G} \n", 2,5);
        Console.WriteLine("{0:N} \n", 2500000);
        Console.WriteLine("{0:X} \n", 250);
        Console.WriteLine("{0:X} \n", 0xffff);
        Console.ReadLine();
    }
}

```

Código fuente 17

La salida por pantalla es:

```

2,00 ?
00025
2,500000E+005
25,00
25
2
2.500.000,00
FA
FFFF

```

El resultado generado por las cadenas de formato numérico personalizado cuando éstas se aplican a los tipos y valores de datos específicos, se ilustra en la siguiente tabla. Aunque no se muestra explícitamente, todos los ejemplos se aplican a un método **ToString()** (predeterminado en todos los objetos del CTS), donde la columna Formato indica la cadena de formato que se ha pasado. La columna Tipo de datos indica el tipo de datos utilizado, mientras que la columna Valor indica el valor del número al que se está dando formato.

Finalmente, la columna Resultado indica el resultado predeterminado de la aplicación de formato con el inglés de Estados Unidos como referencia cultural. Se pretende que esta tabla sea una guía rápida para comprender cómo afectan a la aplicación de formato los especificadores de formato numérico personalizado.

Formato	Tipo de datos	Valor	Resultado
#####	Double	123	123
00000	Double	123	00123
(###) ### - #####	Double	1234567890	(123) 456 – 7890
#.##	Double	1.2	1.2

0.00	Double	1.2	1.20
00.00	Double	1.2	01.20
#, #	Double	1234567890	1,234,567,890
#, ,	Double	1234567890	1235
#, , ,	Double	1234567890	1
#, ##0, ,	Double	1234567890	1,235
#0.##%	Double	0.086	8.6%
0.###E+0	Double	86000	8.6E+4
0.###E+000	Double	86000	8.6E+004
0.###E-000	Double	86000	8.6E004
[##-##-##]	Double	123456	[12-34-56]
##;(##)	Double	1234	1234
##;(##)	Double	-1234	(1234)

Tabla 9. Cadenas de formato personalizadas

system.object: un gran tipo

La raíz de todos los tipos de datos en C#, es, como ya hemos apuntado, System.Object. Esto, garantiza que cualquier tipo del CTS comparte un conjunto de características común. En concreto, existen 4 métodos públicos asociados con el tipo base, que son heredados por todos los objetos, tal y como se muestran en la Tabla 10.

Métodos públicos de System.Object	
Método	Descripción
<i>bool Equals()</i>	Compara dos referencias a objetos en tiempo de ejecución para comprobar si apuntan al mismo objeto. Si es así el método devuelve <i>true</i>. En los tipos por valor, se devuelve <i>true</i> si los datos almacenados son idénticos.
<i>int GetHashCode()</i>	Obtiene el código <i>hash</i> de un objeto. Las funciones Hash se usan cuando se desea situar dicho código en una tabla <i>hash</i> por razones de rendimiento.

<i>Type GetType()</i>	Se utiliza con métodos del <i>namespace reflection</i> para obtener la información del tipo de un objeto dado.
<i>string ToString</i>	Por defecto, se utiliza para obtener el nombre del objeto. Admite reemplazo (overload) por clases derivadas para devolver cadenas de representación más adecuadas a las necesidades del usuario.

Tabla 10

Además de estos métodos públicos, *System.Object* posee dos métodos protegidos (*protected*): *void Finalize()*, llamado por el CLR en tiempo de ejecución para permitir la limpieza previa a la recolección de basura en memoria (*Garbage Collection*). Esta característica tiene que ver con algo llamado *finalización determinista*, que comentaremos más adelante.

El otro método protegido es *Object MemberwiseClone()* y para comprender su funcionamiento, tenemos que pensar en la estructura de un objeto. Como es sabido, un objeto puede hacer referencia a muchos otros objetos, que –a su vez– pueden hacer lo mismo, etc. El método *MemberwiseClone()* devuelve una copia del objeto, pero ignorando las referencias que este pudiese hacer a otros objetos, si tal es el caso. A este tipo de copia se la llama *Copia superficial* del objeto. El Código fuente 18 (basado en uno similar del MSDN) ilustra esta situación (esto es sólo a efectos ilustrativos, detallaremos más el funcionamiento de la herencia en el capítulo siguiente).

```
using System;
class ClaseBase {
    public static string Empresa = "Empresa1";
    public int Edad;
    public string Nombre;
}
class ClaseDerivada: ClaseBase {
    static void Main() {

        // Crea una instancia de ClaseDerivada y asigna valores a sus campos.
        ClaseDerivada m1 = new ClaseDerivada();
        m1.Edad = 45;
        m1.Nombre = "Juancho";
        // Ejecuta una copia superficial de m1 y la asigna a m2.
        ClaseDerivada m2 = (ClaseDerivada) m1.MemberwiseClone();
    }
}
```

Código fuente 18

Conversión de tipos

Una de las características fundamentales del tratamiento de tipos de datos es la conversión entre ellos. El C++ esta operación recibe el nombre de *casting*. Una forma conveniente de abordar el problema es distinguir entre la **conversión implícita** (llevada a cabo por el compilador) y la **conversión explícita**, que debe indicarse de forma expresa en el código fuente. En C# se sobreentiende que el conjunto de conversiones explícitas incluye al de conversiones implícitas, más las siguientes:

- **Conversiones explícitas numéricas:** Son aquellas en las que se indica en el código fuente el tipo de dato a convertir mediante la técnica del *casting*, hacia otro tipo numérico, como veremos a continuación.
- **Conversiones explícitas de enumeración:** De *sbyte*, *byte*, *short*, *ushort*, *int*, *uint*, *long*, *ulong*, *char*, *float*, *double* o *decimal* a cualquier tipo *enum* así como su opuestos (de *enum* a los tipos citados), más todos aquellos de cualquier tipo *enum* a cualquier otro tipo *enum*.
- **Conversiones explícitas de referencia:** en general, aquellas en las que están implicados tipos por referencia (ver Tabla 1).
- **Conversiones explícitas de interfaz:** similares a las anteriores, pero en las que uno de los operandos se refiere a una interfaz, en lugar de a una clase.
- **Conversiones Unboxing.** (de las que ya hemos visto un ejemplo)
- **Conversiones explícitas definidas por el usuario:** Si el lector ha revisado la lista de palabras reservadas del lenguaje, probablemente, haya reparado en la presencia de dos que tienen que ver directamente con el tema que nos ocupa: *implicit* y *explicit*. Estas definiciones pueden usarse en el código para establecer la forma exacta en que deseamos que se lleve a cabo una conversión de un tipo definido por el programador.

Conversiones implícitas

La Tabla 11 muestra las conversiones numéricas implícitas predefinidas. Las conversiones implícitas se pueden dar en muchas ocasiones, incluidas la invocación a métodos y las instrucciones de asignación.

De	A
<u>sbyte</u>	short, int, long, float, double o decimal
<u>byte</u>	short, ushort, int, uint, long, ulong, float, double o decimal
<u>short</u>	int, long, float, double o decimal
<u>ushort</u>	int, uint, long, ulong, float, double o decimal
<u>int</u>	long, float, double o decimal
<u>uint</u>	long, ulong, float, double o decimal
<u>long</u>	float, double o decimal
<u>char</u>	ushort, int, uint, long, ulong, float, double o decimal
<u>float</u>	double
<u>ulong</u>	float, double o decimal

Tabla 11

Conviene hacer algunas precisiones, no obstante, respecto a dichas conversiones, tal y como nos indica la documentación de Visual Studio:

- Las conversiones de **int**, **uint** o **long** a **float** y de **long** a **double** pueden producir una pérdida de precisión, pero no una pérdida de magnitud.
- No existen conversiones implícitas al tipo **char**.
- No existen conversiones implícitas entre los tipos de punto flotante y el tipo **decimal**.
- Una expresión constante de tipo **int** se puede convertir a **sbyte**, **byte**, **short**, **ushort**, **uint** o **ulong**, siempre que el valor de la expresión constante quede dentro del intervalo del tipo de destino.

En estos casos, el compilador se encarga de realizar la labor de conversión, lo que simplifica el código eliminando una complejidad innecesaria. El Código fuente 19 es un caso de conversión numérica implícita.

```
int x = 0;
log y = x; //conversión implícita. Sólo es posible si el tipo convertido es mayor
que el original
```

Código fuente 19

Conversiones explícitas

Siempre siguiendo la especificación oficial de la documentación de Microsoft, “las conversiones numéricas explícitas se utilizan para convertir cualquier tipo numérico a cualquier otro tipo numérico, para el que no existe conversión implícita, mediante una expresión que utilice el operador de conversión explícita”. La Tabla 12 muestra dichas conversiones.

De	A
<u>sbyte</u>	byte, ushort, uint, ulong o char
<u>byte</u>	sbyte o char
<u>short</u>	sbyte, byte, ushort, uint, ulong o char
<u>ushort</u>	sbyte, byte, short o char
<u>int</u>	sbyte, byte, short, ushort, uint, ulong o char
<u>uint</u>	sbyte, byte, short, ushort, int o char
<u>long</u>	sbyte, byte, short, ushort, int, uint, ulong o char
<u>ulong</u>	sbyte, byte, short, ushort, int, uint, ulong o char
<u>char</u>	sbyte, byte o short

<u>float</u>	sbyte, byte, short, ushort, int, uint, long, ulong, char o decimal
<u>double</u>	sbyte, byte, short, ushort, int, uint, long, ulong, char o decimal
<u>decimal</u>	sbyte, byte, short, ushort, int, uint, long, ulong, char, float o double

Tabla 12

Una vez más, la referencia oficial nos recuerda los siguientes puntos importantes:

- Las conversiones numéricas explícitas pueden producir pérdida de precisión o provocar excepciones.
- Cuando se convierte un valor de tipo **float**, **double** o **decimal** a un tipo integral, este valor se redondea hacia cero al valor entero más próximo. Si el valor entero resultante queda fuera del intervalo del tipo de destino, se produce una excepción **InvalidCastException**.
- Cuando se convierte **double** a **float**, el valor de tipo **double** se redondea al valor de tipo **float** más cercano. Si el valor de tipo **double** es demasiado pequeño o demasiado grande para ajustarse al tipo de destino, el resultado será cero o infinito.
- Cuando se convierte **float** o **double** a **decimal**, el valor de origen se convierte a la representación **decimal** y se redondea hasta el número más cercano después de la posición decimal 28 si es necesario. Según el valor de origen, se puede producir uno de los siguientes resultados:
 - Si el valor de origen es demasiado pequeño para representarlo como **decimal**, el resultado se convierte en cero.
 - Si el valor de origen es NN (no es un número), infinito o demasiado grande para ser representado como **decimal**, se producirá una excepción **InvalidCastException**.
- Cuando se convierte **decimal** a **double** o **float**, el valor de tipo **decimal** se redondea al valor de tipo **double** o **float** más cercano.

Y también que: "una expresión de moldeado (cast) de la forma (T)E, donde T es un tipo y E es una expresión unaria, realiza una conversión explícita del valor de E al tipo T. Esto significa que al igual que en otros lenguajes, pueden producirse situaciones en las que sea necesario llegar a una conversión múltiple (multicasting)". Algunos ejemplos, nos ayudarán a comprender estas características.

```
long x = 0;
int y = (int) x; //Si no lo indicáramos expresamente, el compilador daría un error
                //del tipo: "no se puede convertir implícitamente el //tipo long a int"
short z = (short) x;
object z = (object) x; //Aquí, el moldeado permite llamar al método
                      // ToString() sobre el objeto z...

System.Console.WriteLine("{0}", z.ToString() + "-convertido");
```

Salida:
1-convertido

Código fuente 20

Otro problema distinto plantea el moldeado entre tipos complejos. En C#, la palabras reservadas *checked* (conversión comprobada) y *unchecked* (conversión no comprobada) pueden usarse para garantizar que la conversión de un tipo ha sido correcta, lo que requiere una expresión extra entre

paréntesis (moldeado múltiple), o bien para ignorar explícitamente la excepción que se produciría, por ejemplo, en un caso de desbordamiento. Veamos el Código fuente 21.

```
double x = 1234567.89;
int y = checked ((int)x); // El valor de y es de 1234567. (Se pierden los
decimales)
int z = unchecked (int) (x * x )
```

Código fuente 21

En este ejemplo, como el resultado de la multiplicación $x*x$ desborda lo que un tipo **int** puede almacenar, si quitásemos la palabra *unchecked* se produciría una excepción. Al no ser así, se devuelven como resultado los 32 bits menores del resultado fuera de intervalo (en el caso del ejemplo, el valor -555515061, en lugar del resultado correcto 1524157875019, que es la parte entera de dicha multiplicación).

Otros casos de moldeado múltiple son comunes al trabajar con la jerarquía de clases de .NET cuando se quieren convertir tipos definidos por el usuario en otros o se utiliza conversión de interfaces. La regla, en este caso, es que siempre existe una conversión implícita entre un tipo complejo y su tipo base. Basándonos en un ejemplo de **Tom Archer** [Archer, 2001] analizamos esa situación.

```
class Empleado { }

class EmpleadoActivo : Empleado { }

class EjemploCast
{
    public static void Main ()
    {
        Empleado e = (Empleado) new EmpleadoActivo (); //Correcto.
        //El moldeado contrario no funciona
        EmpleadoActivo ce = (EmpleadoActivo) new Empleado(); //Error
        //También puede usarse la palabra reservada as para el moldeado, lo que no
        impide el error:
        EmpleadoActivo ce = e as Empleado; //También da error, sólo es un cambio
        sintáctico
        System.Console.WriteLine(e.ToString());
    }
}
```

Código fuente 22

Además de ésta posibilidad, la clase **System.Convert** suministra un conjunto de métodos adecuados para la conversión explícita de cualquier tipo de datos base en otro, siempre dentro de las restricciones impuestas por el CTS.

Variables colectivas (Matrices ó Arrays)

Hemos visto como declarar variables estándar con opciones como la declaración múltiple, y cómo se declaran variables de tipos propios o de la jerarquía de .NET mediante la llamada a su método constructor. Nos queda por revisar la declaración de *arrays*. La definición nos dice que “Una matriz es una estructura de datos que contiene una serie de variables, a las que se obtiene acceso a través de

índices calculados. Las variables contenidas en una matriz, también llamadas elementos de la matriz, son todas del mismo tipo, denominado tipo de elemento de la matriz”¹¹.

En C# los *arrays* pueden declararse vacíos:

```
int[] x;
```

Código fuente 23

ó asignarles una dimensión en el momento de su declaración:

```
int[] x = new int[10];
```

Código fuente 24

lo que declararía e inicializaría a 0 un *array* de 10 objetos de tipo *int* (Los *arrays* en el CTS siempre comienzan en 0). También pueden declararse *arrays* multidimensionales mediante la inclusión de una coma, como muestra el Código fuente 25.

```
short[,] y; //declaramos un array de dos dimensiones
```

Código fuente 25

e incluso podemos declarar el *array* y asignarle una lista de valores iniciales, encerrados entre corchetes:

```
byte[,] z = new byte[2,3] {{1,2,3},{4,5,6}};
```

Código fuente 26

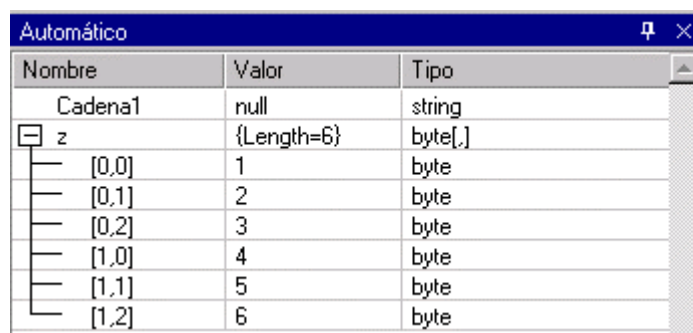
por otra parte, el compilador es capaz de deducir el número de elementos del *array* a partir del número de valores enumerados, por lo que la sintaxis anterior puede expresarse como parece en el Código fuente 27.

```
byte[,] z = {{1,2,3},{4,5,6}};
```

Código fuente 27

¹¹ Según la documentación del MSDN.

Si ejecutamos el programa e incluimos un punto de interrupción al llegar a la declaración, veremos como la ventana **Automático**, nos muestra los contenidos del *array*.



Nombre	Valor	Tipo
Cadena1	null	string
z	{Length=6}	byte[]
[0,0]	1	byte
[0,1]	2	byte
[0,2]	3	byte
[1,0]	4	byte
[1,1]	5	byte
[1,2]	6	byte

Figura 19. Ventana Automático mostrando los valores del array z.

También es posible la declaración dinámica de las dimensiones del *array*, basándonos, por ejemplo, en el valor de otra variable, como en el Código fuente 28.

```
int x = 10;
int[] y = new int[x]; //declara e inicializa un array de 10 elementos.
```

Código fuente 28

Finalmente, podemos utilizar arrays cuyos elementos son –a su vez- otros arrays. Se las denomina *matrices escalonadas* en la documentación, y la sintaxis para definir las, es como muestra el Código fuente 29.

```
byte[] [] s = new byte[5] [];
```

Código fuente 29

también podemos inicializarlas al mismo tiempo.

```
int[] [] numeros = new int[2] [] { new int[] {2,3,4}, new int[] {5,6,7,8,9} };
```

Código fuente 30

Y para el acceso posterior a un elementos de la matriz, utilizaríamos la sintaxis:

```
numeros[2][3] //hace referencia al valor 8.
```

Nótese que la matriz escalonada tiene un carácter jerárquico, al estilo de los documentos XML, y a diferencia de las matrices rectangulares de cualquier dimensión. Podemos apreciar esta particularidad, observando la ventana Automático, tal y como hemos hecho antes, aplicada al ejemplo actual.

Automático		
Nombre	Valor	Tipo
[-] numeros	{Length=2}	int[][]
[+] [0]	{Length=3}	int[]
[+] [1]	{Length=5}	int[]
[-] numeros[1]	{Length=5}	int[]
[0]	5	int
[1]	6	int
[2]	7	int
[3]	8	int
[4]	9	int
numeros[1][3]	8	int

Figura 20. Ventana Automático mostrando los valores del array *numeros*.

Incluso, se pueden combinar matrices rectangulares y escalonadas. Por ejemplo, el Código fuente 31 declara una matriz unidimensional que contiene matrices tridimensionales de matrices bidimensionales de tipo **int**:

```
int[,] [, ,] [,] numeros;
```

Código fuente 31

La clase System.Array

En realidad, todos los arrays manejados en .NET derivan de una clase base denominada **System.Array**. Resulta por tanto, conveniente, recordar algunos de los métodos y propiedades de la clase base, para tener presentes todas las posibilidades del lenguaje en el tratamiento de estos objetos.

Algunas propiedades y métodos de System.Array	
IsFixedSize	Obtiene un valor que indica si Array tiene un tamaño fijo.
IsReadOnly	Obtiene un valor que indica si Array es de sólo lectura.
Length	Obtiene el número total de elementos de todas las dimensiones de Array.
Rank	Obtiene el rango (número de dimensiones) de Array.
Métodos públicos	
Clear	Establece un intervalo de elementos de Array en cero, false o una referencia nula (null), en función del tipo de elemento.
Copy	Sobrecargado. Copia una sección de una matriz Array en otra matriz Array y realiza la conversión de tipos y la conversión boxing de tipos necesarias.

CopyTo	Copia todos los elementos del objeto Array unidimensional actual a un objeto Array unidimensional especificado, empezando en el índice indicado del objeto Array de destino.
GetLength	Obtiene el número de elementos de la dimensión especificada de Array.
GetValue	Sobrecargado. Obtiene el valor del elemento especificado de la matriz Array actual.
Reverse	Sobrecargado. Invierte el orden de los elementos de una matriz Array unidimensional o de una parte de Array.
SetValue	Sobrecargado. Establece el elemento especificado de la matriz Array actual en el valor especificado.
Sort	Sobrecargado. Ordena los elementos en objetos Array unidimensionales.

Tabla 13. Algunos métodos y propiedades de System.Array

Para una visión completa de todos los métodos y propiedades de System.Array, remitimos al lector a la documentación del producto.

Las cadenas como matrices de datos *char*

Una peculiaridad de las cadenas, que nos recuerda el tratamiento que hacen de ellas en los lenguajes antecesores de C# (concretamente en C/C++), es que las cadenas pueden siempre considerarse como matrices de elementos *char*.

Esto significa que podremos recorrer un tipo string como si hubiera sido explícitamente declarado como un array de *chars*, como en el Código fuente 32.

```
string cadena = "Esto es una cadena";
for (int i = 0; i <= cadena.Length; i++)
{
    Console.WriteLine(cadena[i] + "\n");
}
```

Código fuente 32

No obstante, la situación anterior, no debe engañarnos, pues las cadenas no son modificables en sí. Esto es, la operación que muestra el Código fuente 33 no es válida.

```
string cadena = "Esto es una cadena";
cadena[0] = 'H' //Fallo
```

Código fuente 33

Esto es debido a que el compilador se las arregla para compartir en memoria las referencias a cadenas que resulten ser equivalentes léxicamente, para ahorrarse memoria (ya vimos algo al respecto en un ejemplo anterior sobre cadenas). Eso quiere decir que cuando modificamos una cadena, en realidad no se modifica la cadena almacenada originalmente, sino que se hace que la variable haga referencia a otra cadena distinta.

Lo recomendado para el caso de que queramos trabajar de forma intensiva con cadenas modificables, es generarla a partir de la clase ***System.Text.StringBuilder***, que mantiene copias independientes de las cadenas en memoria, y por tanto pueden ser modificadas sin problemas.

Constantes y Variables de sólo lectura

Bien, hemos dejado para el final de éste apartado el estudio de las constantes y las variables de sólo lectura, ya que tienen un número de puntos en común.

Una constante en C# es una expresión que se puede evaluar totalmente en tiempo de compilación. El tipo de una expresión constante puede ser uno de los siguientes: *sbyte*, *byte*, *short*, *ushort*, *int*, *uint*, *long*, *ulong*, *char*, *float*, *double*, *decimal*, *bool*, *string*, cualquier tipo de enumeración o el tipo ***null***.

La sintaxis de declaración de constantes es la siguiente:

```
const <tipo> <nombre> = <valor>;
```

como por ejemplo en los siguientes casos:

```
const int x = 5;  
const int y = x + 5; //x debe estar declarada previamente y ser otra  
constante...
```

Además, todas las constantes son implícitamente estáticas, por lo que se considera un error incluir dicho modificador en la declaración. De hecho, para acceder a su valor desde miembros externos a la clase que los define, se utilizará la sintaxis ***clase.constante***, como con cualquier otro miembro ***static***.

Variables de sólo lectura (readonly)

Un tipo de variable especial es el de sólo lectura. Se declara precedido de la palabra reservada ***readonly*** siguiendo por lo demás la misma sintaxis de las constantes. La gran diferencia es que una variable de este tipo no necesita ser inicializada en la declaración, ni tienen que almacenar valores constantes (pueden ser el resultado de un cálculo), ni tienen que definirse como estáticas. No obstante, la diferencia clave es que –una vez definido su valor–, éste no puede cambiarse. Otra limitación es que sólo pueden definirse como campos (no como variables locales).

```
readonly int x = 5;  
x = 6; //Error
```

Código fuente 34

Clases y la Programación Orientada a Objetos

El trabajo con clases en C#

El lector conoce, sin duda los fundamentos de la programación orientada a objetos, por lo que no creo necesario abundar aquí sobre el tema. Sí parece adecuado, no obstante, abordar la construcción y estudio de las clases siguiendo tales principios fundamentales, que podríamos dividir conceptualmente en dos puntos de vista: el estructural y el funcional.

Entendemos, en el primer caso, aquellos elementos que dan a la clase una arquitectura, y que tradicionalmente, se han subdividido en: campos, métodos, operadores y eventos. En el caso de C#, será necesario además, incluir algunos elementos nuevos que constituyen precisamente las aportaciones más importantes del lenguaje, como son las propiedades (*properties*), los indizadores (*indexers*) y los delegados (*delegates*). Se trata de una división meramente conceptual, ya que estos elementos se traducen a su vez en características del funcionamiento de una clase, y por tanto aportan siempre un componente funcional, a la vez que estructural.

Campos

Un campo es una variable miembro de una clase, que sirve para almacenar un valor. Admiten varios modificadores, que definen el comportamiento del campo en tiempo de ejecución: **private**, **public**, **protected**, **internal**, **static**, **readonly** y **const**. A los cuatro primeros, se les denomina modificadores de acceso, teniendo el resto significados que veremos a continuación.

Tabla de niveles de accesibilidad

La Tabla 14 recoge los niveles de accesibilidad establecidos en C#, junto con su significado.

Accesibilidad declarada	Significado
public	Acceso no restringido.
protected	Acceso limitado a la clase contenedora o a los tipos derivados de esta clase.
internal	Acceso limitado al proyecto actual.
protected internal	Acceso limitado al proyecto actual o a los tipos derivados de la clase contenedora.
private	Acceso limitado al tipo contenedor.

Tabla 14

Private

La declaración **private** es la declaración predeterminada de un miembro en C#. Significa que dicho miembro sólo es visible desde la propia clase y sus clases anidadas (que no derivadas), pero no desde variables de instancia (desde otros objetos que instancien una variable de esa clase). Constituye el pilar fundamental del principio de la **Encapsulación**, y protege del acceso inadvertido a los miembros de una clase, permitiendo preservar la lógica interna definida en su creación.

Public

Aunque existe en todos los lenguajes orientados a objetos, su utilización se justifica raramente, puesto que atenta contra el primer principio de la OOP: la Encapsulación. Un miembro **public** es accesible sin restricciones, por lo que no hay ningún tipo de control sobre los valores que son asignados y/o devueltos.

Entre estos dos límites C# proporciona métodos mixtos de acceso, permitiendo distintos niveles:

Protected

Es idéntico a **private**, a excepción de que se permite que los miembros de las clases derivadas tengan acceso al mismo (al igual que las clases anidadas). Se presta bien para ofrecer funcionalidad a clases derivadas, sin permitir asignaciones inapropiadas, como en el caso de los miembros **public**. El Código fuente 35 ilustra perfectamente esta situación.

```
class A
{
    protected int x = 123;
}

class B : A
{
    void F()
    {
        A a = new A();
        B b = new B();
    }
}
```

```
    a.x = 10;    // Error
    b.x = 10;    // OK
  }
}
```

Código fuente 35

El fallo se produce debido a que la clase A no deriva de la clase B (sino al contrario), y el miembro `x`, está declarado en A, como *protected*.

Internal

Es un modificador de acceso para tipos y miembros de tipos. Eso significa que el modificador es aplicable la declaración de clases, cuando queremos limitar su uso al proyecto en el que son declaradas. Los miembros internos sólo son accesibles dentro de archivos del mismo ensamblado.

El MSDN nos recuerda que “un uso habitual del acceso de tipo interno se da en el desarrollo basado en componentes, ya que permite a un grupo de componentes cooperar de manera privada sin estar expuesto al resto del código de la aplicación. Por ejemplo, una estructura para crear interfaces gráficas de usuario podría proporcionar clases **Control** y **Form** que cooperan mediante miembros con acceso de tipo **internal**. Como estos miembros son internos, no están expuestos al código que utiliza la estructura”.

Protected internal

Es el único modificador que admite más de una palabra reservada. El acceso se limita al proyecto actual o a los tipos derivados de la clase contenedora.

Static

Aunque ya nos hemos referido a él anteriormente, recordemos que sirve para declarar un miembro que pertenece al propio tipo en vez de a un objeto específico. El modificador **static** se puede utilizar con campos, métodos, propiedades, operadores y constructores, pero no con indizadores, destructores o tipos.

Con ello, queremos decir que las referencias al tipo (y, por tanto, a sus miembros y métodos) nunca se harán a través de una instancia de la clase *sino referenciando a la propia clase directamente, como si una instancia de ella con su propio nombre fuera de tipo publico*.

Veamos el Código fuente 36 típico del MSDN para ilustrar la situación.

```
public class A
{
    public static int X;
    internal static int Y;
    private static int Z;
}
internal class B
{
    public static int X;
    internal static int Y;
```

```

private static int Z;
public class C
{
    public static int X;
    internal static int Y;
    private static int Z;
}
private class D
{
    public static int X;
    internal static int Y;
    private static int Z;
}
}

```

Código fuente 36

En esta situación, se dan las siguientes **restricciones de accesibilidad**:

El código anterior introduce dos clases de alto nivel (A y B), dándose la circunstancia de que B posee, a su vez, dos *clases anidadas* (C y D). Según sus modificadores de ámbito la situación es la siguiente:

- El dominio de accesibilidad de A y de A.X es ilimitado.
- El dominio de accesibilidad de A.Y, B, B.X, B.Y, B.C, B.C.X y B.C.Y es el texto del programa contenedor.
- El dominio de accesibilidad de A.Z es el texto del programa de A.
- El dominio de accesibilidad de B.Z y de B.D es el texto del programa de B, incluido el texto del programa de B.C y de B.D.
- El dominio de accesibilidad de B.C.Z es el texto del programa de B.C.
- El dominio de accesibilidad de B.D.X, B.D.Y y de B.D.Z es el texto del programa de B.D.

Observamos aquí algunas conclusiones de interés, como el hecho de que el dominio de accesibilidad de un miembro, nunca es mayor que el de un tipo contenedor, y que toda clase derivada hereda de su antecesora todos sus miembros, a excepción de sus constructores y destructores, lo que no significa que pueda acceder a los mismos. Por tanto, en dicha situación, una clase derivada, aunque herede los miembros privados, no puede acceder a ellos. El Código fuente 37 aclara este concepto.

```

class A
{
    int x;
    static void F(B b) {
        b.x = 1;          // Correcto, aunque el parámetro sea un objeto de la clase B,
                          // el acceso se realiza desde el propio tipo A
    }
}
class B: A
{
    static void F(B b) {
        b.x = 1;          // Error, x no es accesible desde el tipo B.
    }
}

```

Código fuente 37

La encapsulación preserva el acceso al miembro *x* de la clase *A*, que es inaccesible para sus tipos derivados y para cualquier otro no derivado.

Readonly

El modificador ***readonly***, indica, obviamente, que un miembro es de sólo lectura. Se diferencian de las declaraciones ***const*** que veremos a continuación en que no pueden ser calculadas en tiempo de compilación, generalmente, debido a que se precisa una instancia de un objeto para hacerlo, y que deben de ser asignadas, o bien en su declaración, o en el método constructor de la clase.

El Código fuente 38 muestra la declaración y asignación de campos *readonly*, efectuando la inicialización, en el primer caso, dentro de la propia declaración, y en el segundo, en el método constructor predeterminado.

```
using System;
public class ClaseConCamposRO
{
    class ClaseAnidada
    {
        public int x;
        public readonly int y = 2; // Inicializa el campo readonly y
        public readonly int z;

        public ClaseAnidada()
        {
            z = 3; // Inicializa el campo readonly z
        }
        public ClaseAnidada(int p1, int p2, int p3)
        {
            x = p1;
            y = p2;
            z = p3;
        }
    }
    public static void Main()
    {
        ClaseAnidada p1= new ClaseAnidada(4, 5, 6); // Correcto
        Console.WriteLine("p1: x={0}, y={1}, z={2}" , p1.x, p1.y, p1.z);
        ClaseAnidada p2 = new ClaseAnidada();
        p2.x = 7; // Correcto
        Console.WriteLine("p2: x={0}, y={1}, z={2}" , p2.x, p2.y, p2.z);
    }
}
```

Código fuente 38

Por tanto obtendríamos resultados que muestra el Código fuente 39.

```
p1: x=11, y=21, z=32
p2: x=55, y=25, z=24
```

Código fuente 39

En el ejemplo anterior, si se utiliza una instrucción como la que aparece en el Código fuente 40

```
p2.y = 8;           // Error
```

Código fuente 40

se obtendrá el siguiente mensaje de error del compilador:

```
The left-hand side of an assignment must be an l-value
```

que es el mismo error que se obtiene al intentar asignar un valor a una constante.

Const

Se definen como campos cuyo valor puede ser determinado por el compilador en tiempo de compilación y que no se pueden modificar en tiempo de ejecución (son, de forma implícita, *readonly*). Son, pues, más restrictivas que las variables *readonly*, pero pueden definirse también a partir de otras constantes (pero no variables aunque éstas hayan sido declaradas previamente).

```
const int x = 1;
const int y = x + 1;
int z = 2;
const w = z + y + 1; //Error z es una variable
```

Código fuente 41

Propiedades

Sin duda, una de las novedades del lenguaje C# en comparación con sus similares (C++, Java) es la presencia de las propiedades, también llamadas *campos inteligentes*, que no son sino **métodos de acceso** a los campos de una clase, que se ven como campos desde el cliente de esa clase. Se trata de un mecanismo más elegante de acceso al estado¹² de una clase, y una de las aportaciones que el lenguaje C# realiza a la mejora de las técnicas de encapsulación en los lenguajes OOP.

Es curioso notar que la sintaxis utilizada para el acceso a los campos mediante propiedades, parece haber heredado algunos conceptos de Visual Basic, como veremos ahora. Admiten los modificadores *new*, *public*, *protected*, *internal*, *private*, *static*, *virtual*, *sealed*, *override*, *abstract* y *extern*. Además se deben de cumplir algunas normas lógicas, como que el tipo que declara la propiedad tenga, al menos, el mismo nivel de acceso que la propiedad misma.

Las propiedades siguen la notación típica de C# para las estructuras de control, disponiendo de 2 modificadores de acceso: **Get** para la lectura y **Set** para la asignación de valores al campo (escritura). El Código fuente 42 declara una propiedad **Nombre** de tipo lectura/escritura y otra propiedad **FechaDeAlta** de sólo lectura, que son accedidas posteriormente. Un intento de escribir en la propiedad *fechaAlta*, de sólo lectura, genera un error en tiempo de compilación:

¹² Recuérdese que entendemos por *estado* de un tipo (instancia de un objeto) al *conjunto de valores que ese tipo adopta en un instante dado*, y que –de acuerdo con algunos de los teóricos de la OOP- el objetivo de los métodos de una clase no debería ser otro (al menos primariamente) que proveer al tipo de un mecanismo adecuado para la modificación de dicho estado.

```

using System;
class ClaseConPropiedades {

    string nombre;
    System.DateTime fechaAlta;

    public string Nombre
    {
        get { return nombre; }
        set { nombre = value; }
    }

    public DateTime FechaAlta
    {
        get { return fechaAlta; }
    }

    public ClaseConPropiedades(string nombre) //El constructor inicializa
                                                // fechaAlta
    {
        fechaAlta = System.DateTime.Now;
        this.nombre = nombre;
    }
    static void Main()
    {
        ClaseConPropiedades ccp = new ClaseConPropiedades("Posadas");
        Console.WriteLine("El cliente {0} se dio de alta en {1}",
                           ccp.Nombre, ccp.FechaAlta);
        ccp.FechaAlta = new DateTime(2000, 1, 1, 0, 0, 0); //Error
    }
}

```

Código fuente 42

Si compilamos el programa anterior, la línea que muestra el Código fuente 43 genera un error del tipo “No se puede asignar la propiedad o el indizador ‘ClaseConPropiedades.FechaAlta’ -- es de sólo lectura”.

```

ccp.FechaAlta = new DateTime(2000, 1, 1, 0, 0, 0);

```

Código fuente 43

Una consecuencia importante del uso de propiedades es que cuando una clase derivada declara una propiedad con el mismo nombre que una propiedad heredada, la propiedad derivada oculta la propiedad heredada tanto para lectura como para escritura. En el ejemplo que muestra el Código fuente 44.

```

class A
{
    public int P {
        set {...}
    }
}

class B: A

```

```
{
    new public int P {
        get {...}
    }
}
```

Código fuente 44

la propiedad P en B oculta la propiedad P en A para lectura y para escritura.

```
B b = new B();
b.P = 1;           // Error, B.P es de sólo lectura
((A)b).P = 1;      // Correcto, la referencia es a la propiedad P de A
```

Código fuente 45

En este segundo caso, es preciso hacer un casting de tipos para que la referencia sea la correcta.

Indizadores (Indexers)

Hemos visto como las propiedades ofrecen una sintaxis elegante y diferenciadora para acceso a los campos de una clase. Una de las novedades estructurales del lenguaje C# es la existencia de miembros de una clase llamados *indexers*, que son el equivalente a los campos, pero aplicado a matrices (*arrays*).

Estos miembros resultan útiles en aquellas clases que —por su naturaleza— manejen elementos colectivos: conjuntos de números, palabras, caracteres, u otros objetos. Para estos casos, el indizador permitirá que el usuario de la clase utilice una sintaxis similar a la que usamos con los arrays para acceder a uno de los elementos de la clase, tanto en lectura como en escritura.

Imaginemos una clase **CalTareas** que representara las tareas pendientes para los días del año en curso. Sería cómodo poder añadir (o cambiar) tareas con la sintaxis que muestra el Código fuente 46.

```
CalTareas ct = new CalTareas();
ct[5] = "Comprar Reyes"; //Asignamos tarea
Console.WriteLine("Tarea del 5º día {0} ", ct[5]); //Leemos tarea
```

Código fuente 46

Para obtener un comportamiento de este tipo, la clase necesita un *indizador*. Debido a sus características (similares en cuanto al acceso a las propiedades), y por cuestiones de coherencia, se ha utilizado una sintaxis muy similar a éstas, consistente en declarar un miembro de nombre único —*this*—calificado mediante un índice y que permita el acceso a cada uno de los elementos colectivos de la clase. (Adviértase que el comportamiento del indizador es cosa del programador, el indizador solamente suministra el *mecanismo de acceso indexado* a los elementos colectivos de la clase.

Según esto, podríamos construir nuestra clase como muestra el Código fuente 47.

```
using System;
```

```

class CalTareas
{
    string[] Tareas = new string[365];

    public string this [int indice]
    { //Declaramos el indizador
        get
        {
            // Comprobamos los límites
            if (indice < 0 || indice >= 365)
                return ("");
            else
                return Tareas[indice];
        }
        set
        {
            if (!(indice < 0 || indice >= 365))
                Tareas[indice] = value; //Similares en sintaxis a las propiedades
        }
    }
}

class Prueba
{
    public static void Main()
    {
        CalTareas ct = new CalTareas();
        ct[5] = "Comprar Reyes"; //Asignamos tarea
        Console.WriteLine("Tarea del 5° día {0} ", ct[5]); //Leemos tarea
    }
}

```

Código fuente 47

Por tanto, podemos concluir que los **indizadores** son prácticamente propiedades de nombre preestablecido (**this**), que permiten el acceso a campos colectivos (**arrays**) de una clase, usando directamente la sintaxis de matrices. La única limitación existente, estriba en que si se declara más de un indizador en una misma clase, cada indizador deberá tener una signatura diferente.

Métodos

Los métodos de una clase constituyen su componente funcional, siendo aquello que las dota de un comportamiento. Siguiendo las definiciones formales que aporta el MSDN diremos que un método es *“un miembro que implementa un cálculo o una acción que pueden ser realizados por un objeto o una clase. Los métodos tienen una lista de parámetros formales (que puede estar vacía), un valor devuelto (salvo que el tipo de valor devuelto del método sea void), y son estáticos o no estáticos. El acceso a los métodos estáticos se obtiene a través de la clase. El acceso a los métodos no estáticos, también denominados métodos de instancias, se obtiene a través de instancias de la clase”*.

Métodos constructores

Un tipo especial de métodos, los **métodos constructores**, fue introducido en el primer capítulo a propósito de los comentarios sobre el código intermedio (MSIL) y la utilidad de desensamblado (ILDASM). Vamos a completar esos comentarios indicando que una clase puede tener tantos métodos

constructores como se desee, gracias a la característica del polimorfismo, que permite la **sobrecarga** tanto de métodos, como de operadores.

Nota: Puede causar confusión en el lector no habituado a la programación con clases la diferencia entre sobrecarga (**overloading**), que es la capacidad de una clase de declarar el mismo método con diferentes firmas (distintos conjuntos de argumentos), y reemplazo (**override**), mediante el cual una clase derivada sobrescribe (reemplaza) literalmente la implementación de un método en una clase derivada. Mientras en el primer caso solo nos referimos a la declaración formal y la característica implicada es el **polimorfismo**, en el segundo hablamos de sustituir el cuerpo de la función, con lo que manejamos conceptos de **herencia**.

Recordemos que un método constructor es aquel que se llama siempre que se crea una instancia de una clase. Debe de tener el mismo nombre de la clase, pero admite diferentes **firmas**, dependiendo de los distintos conjuntos de parámetros que queramos pasar al constructor. En C# existen 3 tipos de constructores: *de instancia*, *private* y *static*.

Un **constructor de instancia** se utiliza para crear e inicializar instancias, y es –generalmente- el constructor predeterminado. La sintaxis completa de un constructor de este tipo es:

```
[atributos] [modificadores] identificador ([lista-de-parámetros-formales]) [inicializador]
{ cuerpo-del-constructor }
```

donde los tres parámetros más significativos son:

- Los **modificadores** permitidos, que pueden ser **extern** ó cualquiera de los 4 modificadores de acceso
- **identificador**, que es el nombre de la clase, e
- **inicializador** que es una llamada opcional a la clase base (mediante **:base(<lista>)**) o una referencia a la propia clase, expresada mediante **this(<lista>)**, en caso de que deseemos llamar a otro constructor de la misma clase, antes de ejecutar el cuerpo del constructor que hace la llamada.

Dando por sentado que la sintaxis de instanciación de objetos en C# es:

```
<clase> <objeto> = new <clase> (argumentos del constructor),
```

debemos tener en cuenta que en un método constructor se cumplen siempre los siguientes supuestos:

- Un método constructor siempre tiene el mismo nombre de la clase a la que pertenece.
- Un método constructor no devuelve ningún tipo de dato
- Un método constructor admite los modificadores: **public**, **protected**, **internal**, **private**, **extern**
- Un método constructor admite atributos.
- Se dice que la lista de parámetros formales opcional de un constructor de instancia está sujeta a las mismas reglas que la lista de parámetros formales de un método. Dicha lista de parámetros constituye la **firma** del método y se encarga de discriminar la llamada correspondiente, cuando el constructor está sobrecargado.

- Todo método constructor (excepto *System.Object*) efectúa una **llamada implícita previa** a la clase base del constructor, inmediatamente antes de la ejecución de la primera línea del constructor. Se conoce estas llamadas como *inicializadores de constructor* (**Constructor Initializers**).
 - o Estos inicializadores de constructor, si son explícitos (si se indican como tales en el código fuente), pueden presentarse en dos formas, dependiendo de la palabra reservada que usemos en la llamada: **base** o **this**.
 - Si utilizamos **base** estaremos llamando al constructor de la clase base de la que hereda nuestra propia clase.
 - Si usamos **this**, podemos efectuar una llamada interna desde un constructor a otro constructor dentro de la clase.

En el Código fuente 48 vemos un ejemplo de esta situación.

```
using System;

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

    public A(int x)
    {
        Console.WriteLine("A = {0}", x);
    }
}

class B : A
{
    public B(int x)
    {
        Console.WriteLine("B = {0}", x);
    }
}

class Pruebas
{
    public static void Main()
    {
        B b = new B(3);
    }
}
```

Código fuente 48

Podemos mejorar el ejemplo anterior para garantizar que la clase B llama al constructor deseado, (indicado con la línea más oscura), señalando explícitamente al compilador cuál es el constructor a llamar, mediante una modificación del constructor de B, que incluya una llamada **explícita** a la clase base

```
public B(int x): base(x)
```

Código fuente 49

lo que nos garantiza que en la inicialización de B, el argumento recibido –x– es utilizado previamente en una llamada al constructor de su clase base, antes de ejecutarse.

Otra situación posible es aquella en la que deseamos que un determinado método constructor llame, no al constructor de la clase base, sino a otro constructor de la misma clase. Por ejemplo:

```
using System;

class A
{
    public A()
    {
        Console.WriteLine("Constructor predeterminado de A");
    }

    public A(int x):this() //segundo constructor de A, que llama al predeterminado
    {
        Console.WriteLine("A = {0}", x);
    }
}

class Pruebas
{
    public static void Main()
    {
        {
            A a = new A(3);
        }
    }
}
```

Código fuente 50

La salida en ejecución de este programa será la que muestra el Código fuente 51.

```
Constructor predeterminado de A
A = 3
```

Código fuente 51

Los constructores **private** son un tipo especial de constructor de instancias que no es accesible desde fuera de la clase. Suelen tener el cuerpo vacío, y no es posible crear una instancia de una clase con un constructor **private**. Los constructores **private** son útiles para evitar la creación de una clase cuando no hay campos o métodos de instancia (son todos **static** como, por ejemplo, la clase **Math**), o cuando se llama a un método para obtener una instancia de una clase.

```
public class ConstructorEstatico
{
    private ConstructorEstatico() {} // Cuerpo vacío y miembros estáticos
    public static int contador;
    public static int IncrementarContador ()
    {
        return ++contador;
    }
}
```

```

class Pruebas
{
    static void Main()
    {
        // La llamada siguiente a ConstructorEstatico (comentada) generaría un error
        // por que no es accesible
        // ConstructorEstatico cs = new ConstructorEstatico ();    // Error
        ConstructorEstatico.contador = 100;
        ConstructorEstatico.IncrementarContador ();
        Console.WriteLine("Valor: {0}", ConstructorEstatico.contador);
    }
}

```

Salida en ejecución:

Valor: 101

Código fuente 52

Los constructores **static** son un tipo especial de constructor que se utiliza para inicializar (no para crear una instancia) una clase. El MSDN nos recuerda que “se llama automáticamente para inicializar la clase antes de crear la primera instancia o de hacer referencia a cualquier miembro estático. Se declara de la siguiente forma:

[atributos] static identificador () { cuerpo-del-constructor }

Tienen las siguientes características especiales:

Un constructor static no permite modificadores de acceso ni tiene parámetros.

- Es llamado automáticamente para inicializar la clase antes de crear la primera instancia o de hacer referencia a cualquier miembro estático.
- El constructor static no puede ser llamado directamente.
- El usuario no puede controlar cuando se ejecuta el constructor *static* en el programa.
- Los constructores *static* se utilizan normalmente cuando la clase hace uso de un archivo de registro y el constructor escribe entradas en dicho archivo”

Un ejemplo de su uso sería el que muestra el Código fuente 53.

```

using System;
class A
{
    // Constructor estático
    static A ()
    {
        Console.WriteLine("Llamado el constructor estático");
    }

    public static void F()
    {
        Console.WriteLine("Llamado F().");
    }
}

class MainClass

```

```
{
    static void Main()
    {
        A.F();
    }
}
```

Código fuente 53

Con ello se consigue forzar la ejecución del constructor, sea cual sea el miembro de la clase que se llame.

Métodos destructores y el Garbage Collector

En C#, y, a diferencia de la mayoría de los lenguajes que soportan algún tipo de orientación a objetos, los métodos destructores **no debieran ser codificados explícitamente, salvo cuando el tipo encapsule recursos no administrados como ventanas, archivos y conexiones de red**. Esto es debido a que la “*recolección de basura*” (la destrucción de un objeto y posterior reorganización de la memoria ocupada por él) es realizada automáticamente por el **Garbage Collector**, que es parte vital e importantísima del entorno de ejecución de .NET.

Eso significa, que, aunque exista un método para forzar la llamada al destructor de un tipo (que recibe el nombre de ***Finalize()***) su llamada expresa sólo tiene sentido cuando los recursos externos utilizados por el tipo son de peso, o tienen alguna característica especial que les hace susceptibles de una destrucción explícita. En su lugar, los lenguajes del .NET Framework prefieren invocar a un método predeterminado ***Dispose()***, que indica al GC que el objeto en cuestión puede ser destruido y el espacio en memoria ocupado por él, reutilizado, en cuanto su programación predeterminada lo aconseje. Otra razón en contra de la utilización explícita de un método ***Finalize()*** es su penalización en el rendimiento.

La sintaxis de un destructor es similar a la de su constructor opuesto:

```
[atributos] ~ identificador ( ) { cuerpo-del-destructor }
```

El editor de Visual Studio .NET se encarga de situar los métodos adecuados de finalización, por lo que no vamos a hacer más hincapié en este punto.

Métodos estándar

Fuera de las categorías estudiadas, podemos ubicar los llamados métodos estándar, que tienen por objeto dotar al tipo de una funcionalidad específica para ese tipo. Admiten un conjunto de modificadores, que son: **new, public, protected, internal, private, static, virtual, sealed, override, abstract y extern**.

Modificadores de argumento

Además, existen dos palabras reservadas del lenguaje para alterar el modo en que los argumentos de un métodos son enviados y devueltos: **out** y **ref**. Vamos a comenzar con el estudio de estos dos modificadores de argumentos:

Básicamente, tienen el mismo propósito, pero una ligera diferencia de funcionamiento. Ambos permiten pasar argumentos por referencia, en vez de por valor. La diferencia es que los primeros necesitan ser inicializados previamente, y los segundos, no. En cierta forma, esto se relaciona con lo ya comentado acerca de los tipos por valor (*value types*) y los tipos por referencia (*reference types*). Sólo que, en este caso, podrían darse hasta cuatro situaciones, en estas posibles combinaciones:

- **Paso por valor de un *value type*.** (Sintaxis estándar, sin modificadores ni en la llamada, ni en la recepción).
- **Paso por referencia de un *value type*.** (Se añade la palabra clave *ref*, tanto en la llamada, como en la lista de argumentos del método:

```
using System;
class PasaValorPorReferencia
{
    static void Doblar(ref int x)
    // x se pasa por referencia y los cambios afectan al original
    {
        x *= 2;
        Console.WriteLine("Valor dentro del método: {0}", x);
    }
    public static void Main()
    {
        int h = 5;
        Console.WriteLine("Valor antes de llamar al método: {0}", h);
        Doblar(ref h); // pasa h por referencia.
        Console.WriteLine("Valor después de llamar al método: {0}", h);
    }
}
```

Salida:

```
Valor antes de llamar al método: 5
Valor dentro del método: 10
Valor después de llamar al método: 10
```

Código fuente 54

- **Paso por valor de un *reference type*** (Una situación a veces delicada, pues, dependiendo de la manipulación que el método haga de la referencia, ésta puede ser permanente o no). Veamos el caso en otro ejemplo (basado en uno similar del MSDN, y traducido por mí).

```
using System;

class PasoDeReferenciaPorValor
{
    static void Cambiar(int[] arr)
    {
        arr[0]=888; // El cambio a un elemento del array, afecta al original,
        // que no se ha creado un nuevo espacio de almacenamiento
        // es la primera instrucción. Si invertimos el orden de
        // instrucciones, las asignaciones también varían.
        arr = new int[5] {-3, -1, -2, -3, -4}; // Pero este es local.
        Console.WriteLine("Dentro del método el primer elemento es: {0}",
        arr[0]);
    }
}
```

```

public static void Main()
{
    int[] myArray = {1,4,5};
    Console.WriteLine("Dentro de Main, antes de llamar al método, el 1° es:
{0}", myArray [0]);
    Cambiar(myArray);
    Console.WriteLine("Dentro de Main, despues de llamar al método, el 1°
es: {0}", myArray [0]);
    Console.Read();
}
}

```

Salida

```

Dentro de Main, antes de llamar al método, el 1° es: 1
Dentro del método el primer elemento es: -3
Dentro de Main, despues de llamar al método, el 1° es: 888

```

Código fuente 55

El Código fuente 55 puede no parecer (y no lo parece) evidente en un primer vistazo. **myArray** se pasa por valor al método **Cambiar()**, luego, aparentemente, tenemos una copia local y los cambios en la copia local no afectan al original. Pero, aquí **sí** que se ve afectado el original (tras la ejecución de la primera línea). La razón es que el tipo pasado es – intrínsecamente– un objeto. En la primera instrucción, se cambia **el dato en su ubicación original**. Pero en la segunda, se crea un nuevo objeto y, con ello, asignamos un nuevo espacio de almacenamiento, por lo que los cambios realizados aquí no afectan al objeto original. Por eso, si se cambia el orden de las instrucciones variamos el resultado, ya que, al crear primero el nuevo objeto, asignamos un nuevo espacio de almacenamiento, y cualquier cambio a partir de ahí, es un cambio local.

Cuando se desea que el método pueda modificar el propio espacio de almacenamiento, se debe de recurrir al cuarto y último caso:

- **Paso por referencia de un *reference type*** (Es una situación similar a la del “*paso por referencia de un value type*”). Aquí utilizaremos las palabras *ref* tanto en la llamada como en la recepción, por lo que el comportamiento es el esperado inicialmente: podemos crear un nuevo array en el método *Change()* sabiendo que tales modificaciones **sí** se verán en el original.

La palabra reservada *params*

En ocasiones, necesitamos construir métodos que permitan recibir un número no determinado de argumentos (de 0 a n). En esos casos podemos declarar como argumento un array, y anteponerle el modificador **params**. Con ello, lo que hacemos es declarar el argumento como un array dinámico, que puede recibir cualquier número de argumentos – acorde con su tipo – en tiempo de ejecución. El funcionamiento es similar al de las declaraciones del método **Main()** cuando se recogen argumentos de la línea de comandos. La lista de argumentos es almacenada en la variable local, y puede ser recorrida posteriormente con un bucle, como en el siguiente ejemplo:

```

using System;

class EjemploParams
{
    static void LectorVariable(params int[] arr)
    {

```

```

        foreach (int x in arr) {
            Console.WriteLine( x );
        }
    }

    public static void Main()
    {
        //Esta llamada funciona
        LectorVariable( 3, 4, 5);
        // Y esta también
        LectorVariable( 3, 4, 5, 6, 7, 8, 9);
        // E incluso esta (aunque no imprima nada)
        LectorVariable();
        Console.Read();
    }
}

```

Código fuente 56

Modificadores de métodos

Son palabras reservadas que califican el comportamiento de un método tanto desde un punto de vista funcional como de arquitectura. Los principales modificadores de métodos aparecen explicados brevemente en la Tabla 15.

Modificador	Descripción
Static	El método pertenece al propio tipo (clase) en lugar de a un objeto específico. No pueden llamarse desde una variable de instancia.
Abstract	El método es un miembro de una clase abstracta. (Si la clase no se declarara como abstracta, esta opción no está disponible). No está permitido heredar de un tipo abstracto. Su función se limita a la definición de jerarquías de clases.
Virtual	La implementación del método puede cambiarse mediante un miembro de reemplazo (override) de una clase derivada. Pueden llamarse igual que los métodos estándar.
Extern	Sólo se establece su declaración formal. La implementación es externa. Se usa principalmente para declarar funciones de una librería externa a .NET Framework.
Override	Proporciona una nueva implementación de un miembro virtual heredado de una clase base.
New	Oculto explícitamente un miembro heredado de una clase base. Se diferencia de override en que la ocultación impide la llamada al mismo método en las clases antecesoras.
Modificadores Especiales	
Abstract	Aplicable a una clase y también a un método cuya clase haya sido declarada de igual forma, permite definir jerarquías de clases, sin proporcionar implementaciones de los métodos.

Sealed

Aplicable exclusivamente a las clases, establece que la clase no se puede heredar, solo se permite la instanciación de objetos de la clase.

Tabla 15

Hay que tener en cuenta que algunos modificadores se pueden combinar, pero otros son mutuamente excluyentes. Por ejemplo, se pueden establecer combinaciones como **extern static**, **new virtual**, y **override abstract**.

Revisaremos los significados principales de estos modificadores, comenzando con las diferencias entre **virtual** y **override**. En el primer caso se indica de forma explícita la posibilidad de que el método sea sustituido en una clase derivada por un miembro de reemplazo.

Téngase en cuenta que lo que se analiza aquí es la manera en que las jerarquías de clase pueden llegar o no a hacer uso de métodos en clases antecesoras, dependiendo de los modificadores utilizados en su declaración.

```
using System;

class A
{
    public void F() { Console.WriteLine("A.F"); }
    public virtual void G() { Console.WriteLine("A.G"); }
}
class B: A
{
    new public void F() { Console.WriteLine("B.F"); }
    public override void G() { Console.WriteLine("B.G"); }
}
class Test
{
    static void Main() {
        B b = new B();
        A a = b;
        a.F();
        b.F();
        a.G();
        b.G();
    }
}
```

Salida:

```
A.F
B.F
B.G
B.G
```

Código fuente 57

En el ejemplo, A.F() es sustituido en la jerarquía por B.F() que tiene la misma signatura, y se declara como **new** ocultando la implementación de A.F(). Hay que notar aquí que si en B declarásemos F() como **override** se produciría un error en tiempo de compilación, ya que para que un método pueda ser reemplazado (mediante **override**) debe de ser declarado como **virtual**, **abstract** u **override**, lo que no es el caso aquí. Hay una sutil diferencia entre reemplazo y sustitución.

El análisis de otro ejemplo del MSDN puede servirnos para ilustrar esta diferencia, cuando 4 clases definen sucesivamente un mismo método y luego éste es llamado en cadena. Como es posible que un

método de una clase oculte otro método heredado, también lo es disponer de varios métodos virtuales con la misma firma:

```
class A
{
    public virtual void F() { Console.WriteLine("A.F"); }
}
class B: A
{
    public override void F() { Console.WriteLine("B.F"); }
}
class C: B
{
    new public virtual void F() { Console.WriteLine("C.F"); }
    // Si sustituyéramos esta declaración por
    // public override void F() { Console.WriteLine("C.F"); }
    // la salida final serían cuatro llamadas a D.F()
}
class D: C
{
    public override void F() { Console.WriteLine("D.F"); }
}
class Test
{
    static void Main() {
        D d = new D();
        A a = d;
        B b = d;
        C c = d;
        a.F();
        b.F();
        c.F();
        d.F();
    }
}
```

Salida

B.F
B.F
D.F
D.F

Código fuente 58

Aunque estamos llamando a cada uno de los métodos, sólo dos de ellos parecen ejecutarse. Esto es debido al fenómeno de la ocultación que comentamos.

Todo sucede de la siguiente forma: En tiempo de compilación se han declarado cuatro objetos de clases distintas, pero en tiempo de ejecución es el CLR el que decide cómo se resuelven las llamadas: eso significa que cuando A.F va a ser accedido, tal acción es ignorada debido a que su superior jerárquico ha reemplazado el método mediante **override** por B.F, que es al que se llama. En la siguiente instrucción, B.F accede sin problemas a su propio método, pero en la tercera (C.F), el método se declara como **new** y **virtual**, siendo reemplazado en D, lo que provoca que las dos siguientes llamadas ejecuten D.F. Pruebe el lector, a modo de ejercicio, lo que sucede si en C, se sustituye la declaración actual por otra declaración idéntica a D, como se sugiere en el código comentado en el ejemplo.

El modificador **extern**

El modificador **extern** en una declaración de método se usa para indicar que el método se implementa externamente, y suele utilizarse para declarar funciones de API's externas como las propias de Windows, si bien puede tratarse de cualquier otra implementación externa válida.

El Código fuente 59 es ya un clásico en .NET y muestra como una aplicación de consola puede llamar a una función de Windows como *MessageBox*, mediante la declaración de la función externa:

```
using System;
using System.Runtime.InteropServices;
class LlamadaExterna
{
    [DllImport("User32.dll")]
    public static extern int MessageBox(int h, string m, string c, int type);

    public static int Main()
    {
        string cadena;
        Console.Write("Introduzca su nombre: ");
        cadena = Console.ReadLine();
        return MessageBox(0, "Hola, " + cadena, "Mensaje desde Windows", 0);
    }
}
```

Código fuente 59

Modificadores de clase **abstract** y **sealed**

El modificador **abstract** puede aplicarse tanto a clases, como a métodos y propiedades. No puede utilizarse en un método si la clase no ha sido declarada como tal, y aunque las clases abstractas pueden realizar declaraciones formales de métodos, éstos no pueden contener ninguna instrucción ejecutable, por lo que puede decirse que son —implícitamente—métodos virtuales.

Las clases de tipo **abstract** presentan las siguientes características:

- No se pueden crear instancias de una clase abstracta.
- Una clase abstracta puede contener descriptores de acceso y métodos y propiedades abstractas.
- No es posible modificar una clase abstracta con el modificador **sealed**, que significa que la clase no se puede heredar.
- Una clase no abstracta derivada de una clase abstracta debe incluir implementaciones reales de todos los descriptores de acceso y métodos abstractos heredados.

El Código fuente 60 devuelve un error de compilación del tipo “*no se puede instanciar un objeto de la clase abstracta Abstracta*”

```
using System;

abstract class Abstracta
{
    string nombre = "PP";
    public void Dime() { Console.WriteLine(nombre); }
}

class Prueba {
```

```

public static void Main()
{
    Abstracta c = new Abstracta();
    c.Dime();
}

```

Código fuente 60

Podemos modificar el código para que sea una clase heredada la que haga la llamada:

```

using System;

abstract class Abstracta
{
    string nombre = "PP";
    public void Dime() { Console.WriteLine(nombre); }
}

class A : Abstracta {}

class Prueba {
    public static void Main()
    {
        A c = new A();
        c.Dime();
    }
}

```

Código fuente 61

No obstante, véase que la clase base declara una propiedad y un método NO abstractos. Si ese fuera el caso, el método Dime() no podría tener implementación, y la clase heredada debería de ejercer dicha función.

```

using System;

abstract class Abstracta
{
    abstract public string Nombre {
        get ; // Adviértase que no se declara el cuerpo de get
    }
    abstract public void Dime();
}

class A : Abstracta {
    public override string Nombre { //miembro de reemplazo de la propiedad
        get { return "Juancho"; }
    }
    public override void Dime() { //miembro de reemplazo del método
        Console.WriteLine(Nombre);
    }
}

class Prueba {

```

```

public static void Main()
{
    A c = new A();
    c.Dime();
}
}

```

Código fuente 62

Como vemos en el ejemplo final, la clase abstracta fuerza a la clase derivada a implementar, tanto la propiedad, como el método, usando el modificador **override**, y el método **get** de la propiedad abstracta, no incluye siquiera las llaves (puesto que definiría un cuerpo, aunque vacío) .

Para concluir con esta revisión operativa de los modificadores aplicables a los métodos, digamos que el modificador **sealed** sólo es aplicable a clases, e indica que la clase no puede heredarse, sino solamente instanciarse. Aparte de ser usada en jerarquías de clases de usuario, esta situación nos la encontraremos muy a menudo en clases de la jerarquía de .NET Framework, debido a su naturaleza. Por ejemplo, la propia clase `Console`, que hemos estado utilizando en nuestros ejemplos, viene definida en el Examinador de Objetos de Visual Studio como muestra la Figura 21.

```

public sealed class Console : System.Object
    Miembro de System

Summary:
Representa las secuencias de entrada, salida y error estándar para las aplicaciones de consola. No se puede heredar esta clase.

```

Figura 21. Definición de la clase `System.Console` en .NET

Sobrecarga de Operadores (Operator Overloading) y conversiones definidas por el usuario

Hemos visto el funcionamiento de la sobrecarga aplicada a los métodos, pero ésta no es la única forma de sobrecarga que permite el lenguaje C#: también podemos establecer cómo deseamos que sea el comportamiento de nuestros propios tipos cuando les aplicamos los operadores aritméticos clásicos, como `+`, `-`, `*`, etc, y —en genera— cualquier operador.

Como se ha hablado mucho en pro y en contra del uso de la sobrecarga de operadores en los lenguajes de programación, me gustaría citar una opinión a la que me adhiero, y que puede leerse en [Gunnerson 2000]: "Sólo se debería sobrecargar un operador si así se obtiene un código más claro. Y observe que el texto dice más claro, no más corto". O como recomienda la propia documentación del MSDN: "La norma para saber cuándo sobrecargar es muy simple. Si el usuario esperaría poder realizar esa operación, se debería sobrecargar."

Operadores sobrecargables en C#

Operadores Unarios	<code>+</code> , <code>-</code> , <code>!</code> , <code>~</code> , <code>++</code> , <code>--</code> , <code>true</code> , <code>false</code>
Operadores Binarios	<code>+</code> , <code>-</code> , <code>*</code> , <code>/</code> , <code>%</code> , <code>&</code> , <code> </code> , <code>^</code> , <code><<</code> , <code>>></code> , <code>==</code> , <code>!=</code> , <code>></code> , <code><</code> , <code>>=</code> , <code><=</code>

Tabla 16

Hay que observar que ciertos operadores no son sobrecargables, como el operador de asignación (=), si bien se produce una sobrecarga implícita al sobrecargar los operadores binarios aritméticos básicos (+, -, *, /) y utilizarlos posteriormente en expresiones compuestas del tipo $x += 1$.

Por ejemplo, imaginemos que se dispone de una clase que maneja números complejos de la forma $x+yi$, donde $i = \sqrt{-1}$. Como el lector sabrá, tales números admiten una representación en el plano real que especifica, en coordenadas cartesianas los valores de las componentes real e imaginaria del número complejo (respectivamente, x e y). Dado que la operación de suma de complejos está definida en el plano $\mathbb{R} \times \mathbb{R}$ ¹³, sería interesante poder manejar dicha suma mediante la sobrecarga del operador + en nuestra clase.

Podríamos escribir el Código fuente 63.

```
// Ejemplo sobrecarga con números complejos
using System;

public struct Complejo
{
    public int real;
    public int imaginaria;

    public Complejo(int real, int imaginaria)
    {
        this.real = real;
        this.imaginary = imaginaria;
    }

    // Declarar el operador de sobrecarga (+), los tipos
    // que se pueden agregar (dos objetos Complejo), y el
    // tipo de valor devuelto (Complejo):
    public static Complejo operator +(Complejo c1, Complejo c2)
    {
        return new Complejo(c1.real + c2.real, c1.imaginaria + c2.imaginaria);
    }

    // Reemplazar el método ToString para mostrar adecuadamente número complejo
    public override string ToString()
    {
        return(String.Format("{0} + {1}i", real, imaginaria));
    }

    public static void Main()
    {
        Complejo num1 = new Complejo(2,3);
        Complejo num2 = new Complejo(3,4);

        // Agregar dos objetos Complejo (num1 y num2) mediante el operador más sobrecargado
        Complejo sum = num1 + num2;

        // Imprimir los números y la suma usando el método reemplazado ToString:
        Console.WriteLine("Primer número complejo: {0}", num1);
```

¹³ La expresión matemática que indica esta definición tendría la forma:

$(x_1 + yi_1) + (x_2 + yi_2) = (x_1 + x_2) + (y_1 + y_2)i$

```

Console.WriteLine("Segundo número complejo: {0}", num2);
Console.WriteLine("Suma de los dos números: {0}", sum);

}
}

```

Código fuente 63

No obstante, conviene recordar lo dicho al principio, la sobrecarga debe de sopesarse adecuadamente, para evitar lo que algunos llaman *"una solución en busca de un problema"*, o sea, utilizar una característica a toda costa, tenga o no sentido en la solución que intentamos construir. A continuación, concluimos este apartado con la tabla que aparece en el MSDN sobre la posibilidad de sobrecarga de operadores, considerados por familias.

Operadores	Posibilidad de sobrecarga
<code>+, -, !, ~, ++, --, true, false</code>	Estos operadores unarios sí se pueden sobrecargar.
<code>+, -, *, /, %, &, , ^, <<, >></code>	Estos operadores binarios sí se pueden sobrecargar.
<code>==, !=, <, >, <=, >=</code>	Los operadores de comparación sí se pueden sobrecargar (ver la nota inferior).
<code>&&, </code>	Los operadores lógicos condicionales no se pueden sobrecargar, pero se evalúan con <code>&</code> y <code> </code> , que sí pueden sobrecargarse.
<code>[]</code>	El operador de indización de matrices no se puede sobrecargar, pero se pueden definir indizadores.
<code>()</code>	El operador de conversión explícita de tipos no se puede sobrecargar, pero se pueden definir nuevos operadores de conversión.
<code>+=, -=, *=, /=, %=, &=, =, ^=, <<=, >>=</code>	Los operadores de asignación no se pueden sobrecargar, pero <code>+=</code> , por ejemplo, se evalúa con <code>+</code> , el cual sí se puede sobrecargar.
<code>=, ., ?:, ->, new, is, sizeof, typeof</code>	Estos operadores no se pueden sobrecargar.

Tabla 17

Nota: Los operadores de comparación, si se sobrecargan, deben ser por parejas; es decir, si se sobrecarga `==`, también se debe sobrecargar `!=`. También es cierta la opción contraria, análogamente para el caso de `<` y `>`, y para `<=` y `>=`.

Conversiones definidas por el usuario

Ya vimos en el capítulo anterior como se definían y manejaban las conversiones entre tipos, y cómo no eran posibles ciertas conversiones mediante la técnica del *casting*, debido a la naturaleza de los tipos implicados y las restricciones del CTS. No obstante, el usuario puede definir sus propias

conversiones utilizando una técnica similar a la que hemos usado para la sobrecarga de operadores, definiendo las acciones a realizar e indicando si la conversión es implícita o explícita.

Como hemos hecho en ejemplos anteriores, pensamos que puede ser más clara la comprensión de este concepto si primero presentamos el objetivo que se persigue mediante un ejemplo. Imaginemos, pues, que disponemos de una clase que ha definido sus propias conversiones, llamada *MiClase*. Nos gustaría poder utilizar una sintaxis de conversión implícita, como en el Código fuente 64.

```
MiClase x;  
int i = x; // llama implícitamente al operador de conversión de MiClase en int
```

Código fuente 64

Podemos implementar ese comportamiento utilizando una sintaxis similar a la de la sobrecarga de operadores y teniendo en cuenta que la palabra clave ***implicit*** se utiliza para declarar un operador de conversión de tipo implícita.

```
class MiClase  
{  
    public static implicit operator int(MiClase m)  
    {  
        // código para convertir MiClase en int  
    }  
}
```

Código fuente 65

Los operadores de conversión implícita nunca deberían producir excepciones ni pérdida de información, (como bien nos recuerda el MSDN), de modo que puedan utilizarse de forma segura sin intervención del programador. Si un operador de conversión no cumple esos criterios, debería definirse como ***explicit***.

En el caso de la conversión explícita, lo que buscamos es un poder escribir código del tipo:

```
int i;  
MiClase x = (MiClase)i; // Se requiere un moldeado para pasar de int a MiClase
```

Código fuente 66

En este caso, la función de conversión explícita tendría que tener la forma:

```
class MiClase  
{  
  
    public static explicit operator int(MiClase m)  
    {  
  
        // código para convertir MiClase en int  
  
    }  
}
```

```
}  
}
```

Código fuente 67

Al igual que comentamos anteriormente, sólo cabe recordar que estas prácticas deben de tener siempre como objetivo la claridad del código, y no ser utilizadas de forma gratuita, si no se obtiene un claro beneficio de su uso.

Eventos y Delegados (Delegates)

Se decía que los tres componentes funcionales más importantes de una clase eran, sus campos (que definen su **estado**), sus métodos (que definen su **funcionalidad**) y sus eventos (que establecen la forma de **comunicación** con el exterior de la clase). Estos elementos, junto a las palabras reservadas del lenguaje que sirven de modificadores, permiten implementar en una clase las 3 características de la OOP (polimorfismo, encapsulación y herencia), que se ven enriquecidas al evolucionar los compiladores, como sucede en C#. Precisamente, una de las características más novedosas – y una de las auténticas aportaciones del lenguaje C# – la constituye un tipo de objeto especial llamado *delegate*.

Los eventos se han convertido en el mecanismo estándar de comunicación entre clases. Su definición oficial dice que: "*Un evento es un mensaje que envía un objeto cuando ocurre una acción*". Pero, una de las diferencias con las llamadas convencionales entre funciones estriba en que el evento *salta* (digamos en términos más técnicos, que **bifurca incondicionalmente**) hacia otro método de otra clase cuando se produce una circunstancia previamente programada: un click de ratón, una pulsación de teclado, una entrada de datos por un puerto de comunicaciones, etc. La otra diferencia reside en que un evento **no garantiza en ningún momento la ejecución de un método en la clase receptora**. De hecho la clase generadora ni siquiera conoce quiénes pueden ser los receptores del mensaje, y por su parte, la clase receptora puede preferir ignorar el evento, no asociando ningún código ejecutable cuando este se produzca.

En los sistemas operativos gráficos, como Windows, System X de Mac, o X-Windows de Unix, la interfaz de usuario se basa en ventanas, que una vez lanzadas, entran en un bucle de eventos (o mejor dicho, en un bucle *a la escucha de eventos*), que no son sino acciones programadas en clase receptora que se corresponden con saltos incondicionales en la clase emisora). Tales saltos, se traducen internamente en mensajes del sistema operativo, que son procesados por la clase receptora, para comprobar si existe alguna acción asociada (algún tipo de código fuente de respuesta). Si tal código fuente existe, se procesa la llamada y se devuelve el control a la función que produjo el evento.

Estamos, pues ante un mecanismo de comunicación **en un solo sentido, y que puede ser ignorado**, por lo que, algunos autores, han preferido denominarlo como **notificación** debido precisamente a esta circunstancia. Claro está que, igual que pasa con cualquier otro tipo de mensaje sin confirmación, la clase emisora nunca tiene la certeza de que tal evento es recibido realmente. Esto, en algunos lenguajes como C++ se traducía en punteros nulos, u otra serie de características que podían llevar al evento a fallar.

Como en C# no hay punteros, la idea de los creadores del lenguaje fue adoptar una solución similar a la que sirvió para manejar –en su tiempo– los extensores del DOS para memoria extendida: situar un intermediario en la llamada, que pudiera garantizar que esta tiene lugar o bien generar un **error interceptable**, en caso de la clase receptora no esté presente. De ahí su nombre de **delegate**, ya que la clase emisora *delega* la llamada en un tercero especializado, garantizando así, la coherencia en tiempo de ejecución.

Técnicamente, el MSDN nos recuerda que *"un delegado es una clase que puede guardar una referencia a un método (o a más de uno). A diferencia de otras clases, una clase de delegado tiene un prototipo (una **signatura**) y puede guardar referencias únicamente a los métodos que coinciden con su prototipo. [...] Por convención, los delegados de evento de .NET Framework tienen dos parámetros, el origen que provocó el evento y los datos del evento"*. Una declaración de evento tiene el formato que muestra el Código fuente 68.

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

Código fuente 68

Nota: Una declaración de delegado es suficiente para definir una clase de delegado. La declaración proporciona el prototipo del delegado y Common Language Runtime proporciona la implementación.

De forma que, funcionalmente, un delegado actúa como remitente de mensajes (eventos) de la clase que provoca el evento y mantiene una lista de los controladores registrados para el evento (o sea, de métodos a los que el delegado debe invocar cuando se le llame). La forma en que una clase dispara un evento, es por tanto, invocando al delegado que guarda las referencias de métodos a llamar.

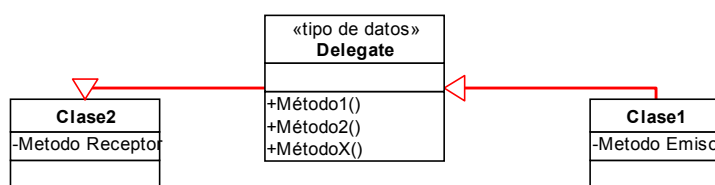


Figura 22. Esquema del funcionamiento de un delegado

Podemos poner en marcha un primer ejemplo con delegados mediante la declaración del delegado y posterior asignación de la lista de métodos a los que debe invocar. El Código fuente 69 ilustra esta situación.

```
using System;

namespace Eventos
{
    delegate int Delegado (int valor);

    class Class2
    {
        public static int Cuadrado(int x)
        {
            Console.WriteLine("Yo calculo el cuadrado de {0}: {1}", x, x*x);
            return (x*x);
        }
        public static int Cubo(int x)
        {
            Console.WriteLine("Yo calculo el cubo de {0}: {1}", x, x*x*x);
            return (x*x*x);
        }
        static void Main()
        {
            Delegado od = new Delegado(Cuadrado); //Llamar al constructor
            od += new Delegado(Cubo); // Añadir más referencias al delegate
        }
    }
}
```

```

        int x = od(3);           //Invocación del delegado
        Console.WriteLine("Último valor recogido: {0}: ", x);
        Console.Read();
    }
}
}

```

Salida:

```

Yo calculo el cuadrado de 3 : 9
Yo calculo el cubo de 3 : 27
Último valor recogido: 27

```

Código fuente 69

Como se llama a dos funciones distintas, se obtienen dos ejecuciones en cadena (siempre en el orden que se han almacenado). Sin embargo, si los métodos llamados devolviesen algún valor, sólo se podría recoger el último (en este caso, el devuelto por la función Cubo()). También se debe de tener en cuenta, que la signatura del delegado (o su prototipo, como se quiera), **define** el tipo de referencias que puede albergar.

Restricciones

Algunas restricciones aplicables al uso de delegados son:

- Sólo se permite el uso del operador de asignación en la llamada al método constructor del delegado
- Sucesivas llamadas utilizarán el operador += para añadir y -= para eliminar referencias a métodos.
- El intento de almacenar una referencia de signatura distinta de la del delegado produce un error en tiempo de compilación.

Como el delegado es, intrínsecamente, una clase, puede servir para poner en comunicación dos clases separadas, aunque pertenecientes al mismo **namespace**. El Código fuente 70 se ha modificado para ilustrar esta diferencia.

```

using System;

namespace Eventos
{
    delegate int DelegadoI (int valor);

    class Class3
    {
        public static int Cuadrado(int x)
        {
            Console.WriteLine("Yo calculo el cuadrado de {0}: {1}", x, x*x);
            return (x*x);
        }
        public static int Cubo(int x)
        {
            Console.WriteLine("Yo calculo el cubo de {0}: {1}", x, x*x*x);
            return (x*x*x);
        }
    }
    class Prueba
    {

```

```
static void Main()
{
    DelegadoI od = new DelegadoI(Class3.Cuadrado); //Llamar al constructor de od
    od += new DelegadoI(Class3.Cubo); // Añadir más referencias al delegate
    int x = od(3); //Invocación del delegado
    Console.WriteLine("Último valor recogido: {0}: ", x);
    Console.Read();
}
}
```

Código fuente 70

Tenemos ahora dos clases separadas (pero dentro del mismo espacio de nombres), más otra que hace de conectora, que es el delegado. Como los métodos de la clase *Class3* son de tipo *static* no se precisa instanciación para añadir sus referencias al delegado. Por lo demás, la ejecución del código anterior, produce un resultado idéntico al del ejemplo anterior.

Eventos

En C#, decimos que un evento es un miembro de una clase pero cuyo tipo es un delegado. El uso habitual en las interfaces gráficas, permite que los elementos de la interfaz comuniquen al programa cuándo se ha producido un cambio de estado (como consecuencia de una acción del usuario). El programador puede responder o no a esa acción, asociando una funcionalidad al método que debe hacer el tratamiento del evento: el **manejador del evento**.

Cuando programamos en Windows bajo Visual Studio, el IDE se encarga de realizar la labor de bajo nivel de **enlazar** el disparador del evento con el receptor, a través de un delegado apropiado para ese tipo de evento. En las aplicaciones de consola, la situación no es tan sencilla (aunque el escenario es básicamente el mismo en términos de los objetos que intervienen), ya que toda la labor de creación, configuración y puesta en marcha tiene que realizarla manualmente el programador.

Como la mayoría de los ejemplos al uso son bastante complejos para una primera aproximación, (y el que voy a proponer también tiene un cierto grado de complejidad), vamos a comentar el funcionamiento con un ejemplo, que –por lo menos– utilice algunas clases fáciles de identificar, para ayudarnos en la comprensión del código. Aprovecho para hacer un pequeño homenaje a los personajes de la Warner Bros. suponiendo la situación siguiente.

En un bosque (*la clase Bosque*), viven el **Lagarto Juancho** (clase LagartoJuancho) y **Pepe Pótamos** (clase PepePotamos). Como Juancho tiene un sueño muy pesado, sólo un **Hipogrito** lanzado por PepePótamos puede despertarlo. Pero, ¿cómo hacemos para que el hipogrito lanzado llegue a oídos de Juancho? Mediante un delegado (**Deleg_EncauzaGritos**) que se encarga de conectar el Hipogrito lanzado mediante el método LanzarHipogrito() con el método OirPepePotamos() de Juancho.

El gráfico adjunto ilustra esa situación. Además, Juancho sólo se despertará cuando la intensidad del Hipogrito supere un número de decibelios (120). Para que el delegado pueda no sólo conectar los métodos, sino **transportar la información del hipogrito**, se debe de crear otra clase (clase **DatosDelHipogrito**), que almacene esa información, y la transmita al conectar los métodos. Finalmente, el método OirPepePotamos() estará programado para que al oír un Hipogrito de intensidad mayor o igual que 120 Db. responda y haga que el HipoGrito se pare.

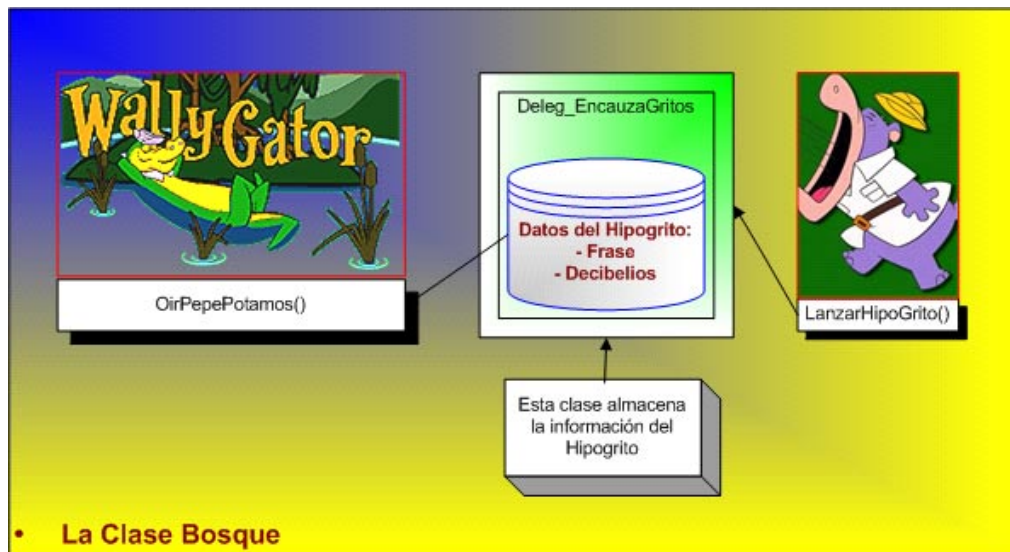


Figura 23. Esquema de funcionamiento del delegado

Por lo tanto, vamos a ir analizando el código necesario para implementar este escenario "bucólico": Lo primero es crear la clase **DatosDelHipoGrito**, que almacenara la información que se transmite (la frase y los decibelios). Posteriormente, tendremos que declarar el delegado correspondiente del tipo **DatosDelHipoGrito** para que pueda transportarlos al producirse el evento.

```
// Esta clase contiene los datos del evento que
// se transmite (Hipogrito) entre el objeto PepePotamos
// y el objeto LagartoJuancho (lagarto objeto)

public class DatosDelHipoGrito : EventArgs //Es obligada la herencia de EventArgs
{
    int decibelios;           // Intensidad del Hipogrito

    public DatosDelHipoGrito(int decibelios) {      // constructor
        this.decibelios = decibelios;
    }
    public int Decibelios {
        get { return(decibelios); }
    }
    //Frase que repite PepePotamos
    public string FrasePepePotamos {
        get {return (";;;Despierta JUANCHO!!! (a "+ decibelios.ToString()+ " Db.)");}
    }
}
```

Código fuente 71

La frase es siempre la misma, así que el constructor sólo recibe la intensidad, en decibelios. Las propiedades de sólo lectura *Decibelios* y *FrasePepePotamos* devuelven los valores correspondientes (Observe el lector que —estrictamente— no es necesario crear un campo *Frase*, basta con la propiedad de acceso).

A continuación, tenemos que definir el delegado que conectara los objetos. Para poder transportar la información del tipo **DatosDelHipoGrito**, su declaración deberá de ser coherente con el tipo transportado:

```
// Declaracion de la clase Delegate que hace las veces
// de enlazador entre el emisor y el receptor

public delegate void Deleg_EncauzaGritos (object sender, DatosDelHipoGrito e);
```

Código fuente 72

Ahora, podemos crear la clase que originará el evento (al Hipogritar). La clase PepePortamos deberá de disponer de un miembro de la clase **Deleg_EncauzaGritos** (miembro que sólo puede ser un evento), además de disponer de algún mecanismo para lanzar dicho evento (el método LanzarHipoGrito()).

```
// La clase PepePotamos que lanza el evento
public class PepePotamos {
    //Definición de la propiedad Parar para detener el ciclo de eventos
    bool parar = false;
    string nombre = "Pepe Pótamos";

    public string Nombre {
        get { return(this.nombre); }
    }

    public bool Parar {
        set { this.parar = value; }
        get { return(this.parar); }
    }

    //Definición del evento HipoGrito (es importante
    //notar que -en la declaración- el evento se califica
    //como del tipo Deleg_EncauzaGritos
    public event Deleg_EncauzaGritos HipoGrito;

    public void LanzarHipoGrito()
    {
        int db = 10; // Intensidad de base del Hipogrito
        //Bucle infinito para repetir el HipoGrito
        //hasta que se reciba la orden de parar-
        for(;;) {
            //Control para detener el ciclo de eventos(repetición del HipoGrito)
            if (parar == true) break;
            db += 25;
            // A cada paso del bucle se crea un nuevo objeto del tipo
            // DatosDelHipogrito con la nueva información
            DatosDelHipoGrito e = new DatosDelHipoGrito(db);
            // Se invoca el evento HipoGrito, pasándole la información
            // almacenada en e
            if (HipoGrito != null) {
                HipoGrito(this, e); //Invoca los delegates. (sender es siempre this)
            }
        }
    }
}
```

Código fuente 73

La clase LagartoJuancho dispone de un único método (OirPepePotamos) que –alcanzado un cierto nivel de decibelios– reaccionará pidiendo el cese del Hipogrito:

```
//La clase LagartoJuancho dispone de un único método OirPepePotamos
//para escuchar los HipoGritos (manejar los eventos)
public class LagartoJuancho
{
    public void OirPepePotamos(object sender, DatosDelHipoGrito e)
    {
        Console.WriteLine(e.FrasePepePotamos + "\n");
        if(e.Decibelios >= 150)
        {
            //Para que Juancho sepa quién le despierta podemos
            //averiguar quién es el que envía el mensaje
            string Respuesta = @"Juancho- Vale, ";
            Respuesta += ((PepePotamos) sender).Nombre + "... ¡Te oigo!";
            Console.WriteLine(Respuesta);
            //Esta asignación es en realidad la comunicación
            //de Juancho con PepePotamos para decirle que pare-
            ((PepePotamos) sender).Parar = true;
            Console.ReadLine();
            return;
        }
    }
}
```

Código fuente 74

Finalmente, necesitamos una clase que ponga en marcha todo este escenario (la clase Bosque), instanciando los objetos correspondientes y asignando el delegado y estableciendo el punto de entrada de la aplicación.

```
//La clase Bosque (donde viven LagartoJuancho y PepePotamos,
//es, quien instancia dichos objetos para darles "vida"),
//y usa el delegate Deleg_EncauzaGritos para vincular el evento
//con el manejador(El Hipogrito con el método OirPepePotamos).
//(En una aplicación Windows, esa función es llevada a cabo por la ventana).
public class Bosque{

    public static void Main ()
    {
        //Instancia el receptor de eventos=> Lagarto Juancho
        LagartoJuancho J = new LagartoJuancho();

        //Instancia el origen de eventos => Pepe Potamos
        PepePotamos P = new PepePotamos();

        //Registra el manejador de eventos(OirPepePotamos), con el
        //origen de eventos (PepePotamos.HipoGrito)
        //(la sintaxis += añade el Delegate a la lista de delegates
        //registrados con el evento HipoGrito
        P.HipoGrito += new Deleg_EncauzaGritos(J.OirPepePotamos);
        P.LanzarHipoGrito();
    }
}
```

Salida

```
---Despierta JUANCHO!!! (a 35 Db.)
---Despierta JUANCHO!!! (a 60 Db.)
---Despierta JUANCHO!!! (a 85 Db.)
---Despierta JUANCHO!!! (a 110 Db.)
---Despierta JUANCHO!!! (a 135 Db.)
---Despierta JUANCHO!!! (a 160 Db.)
Juancho- Vale, Pepe Pótamos... -Te oigo!
```

Código fuente 75

La sintaxis asociada al evento HipoGrito es especial de este tipo de miembros (de los eventos). Los únicos operadores permitidos son += y -= para añadir manejadores de evento a la lista. En la práctica, podríamos (en nuestro ejemplo) hacer que un segundo objeto que dispusiera de un método de escucha reaccionara igualmente al *hipogrito*, y –sobre todo– que lo hiciera de forma adecuada (y posiblemente, distinta) de como lo ha hecho *Juancho*. Esa es la esencia del delegado **multicast**: puede notificar una situación a más de un objeto simultáneamente, y cada uno de los receptores, obrar de acuerdo con sus necesidades.

Por ejemplo, si añadiésemos una clase más (pongamos al GalloClaudio) con un método que respondiera de alguna forma para poder monitorizarlo (como por ejemplo escribiendo algo en pantalla), el método LanzarHipogrito() podría ser recibido por ambos objetos. Tal situación sólo requeriría de una clase similar a Juancho, como la que muestra el Código fuente 76.

```
//La clase GalloClaudio también dispone de un método OirGritos
//pero no reacciona despertándose, sino quejándose de los Hipogritos
public class GalloClaudio
{
    public void OirGritos(object sender, DatosDelHipoGrito e)
    {
        //Respuesta del Gallo Claudio
        string Respuesta = @"GalloClaudio- ";
        Respuesta += ((PepePotamos) sender).Nombre + "¡No grites!";
        Console.WriteLine(Respuesta);
        return;
    }
}
```

Código fuente 76

En el código de la llamada dentro de la clase Bosque, sólo nos faltaría registrar el método OirGritos de GalloClaudio, para que también recibiese la notificación.

```
// Ahora, añadimos también como receptor a GalloClaudio
P.HipoGrito += new Deleg_EncauzaGritos(G.OirGritos);
```

Salida:

```
---Despierta JUANCHO!!! (a 35 Db.)
GalloClaudio- Pepe Pótamos-No grites!
---Despierta JUANCHO!!! (a 60 Db.)
GalloClaudio- Pepe Pótamos -No grites!
---Despierta JUANCHO!!! (a 85 Db.)
GalloClaudio- Pepe Pótamos -No grites!
---Despierta JUANCHO!!! (a 110 Db.)
GalloClaudio- Pepe Pótamos -No grites!
---Despierta JUANCHO!!! (a 135 Db.)
GalloClaudio- Pepe Pótamos -No grites!
---Despierta JUANCHO!!! (a 160 Db.)
Juancho- Vale, Pepe Pótamos... -Te oigo!
GalloClaudio- Pepe Pótamos -No grites!
```

Código fuente 77

El lector podrá extender este ejemplo a situaciones mucho más serias en las que se necesita que un mismo origen de eventos (una situación crítica, por ejemplo), sea recogido simultáneamente por un conjunto de métodos de diversos objetos permitiéndoles actuar en consecuencia (Sistemas de control de dispositivos, situaciones especiales en la bolsa de valores, etc.).

La función principal de los eventos en particular (respecto al uso directo de los delegates), es que permite el control sobre los métodos que se asocia (o eliminan) en un delegado.

Instrucciones

Instrucciones en C#

El flujo de programación, lo que permite la creación de algoritmos, se gestiona en C#, al igual que en la gran mayoría de los lenguajes, mediante instrucciones simples y múltiples, que dan lugar a lo que llamamos estructuras de control. C# incorpora novedades respecto a sus lenguajes antecesores y también soporta las estructuras más conocidas y utilizadas por éstos. Asumiendo que el lector ya tiene un conocimiento general del funcionamiento de casi todas ellas, no nos vamos a detener especialmente en las más conocidas, sino en aquellas que supongan una novedad lingüística, semántica o contextual. De todas formas, revisaremos las instrucciones de programación haciendo un análisis básico por categorías:

- Instrucciones básicas
- Bifurcaciones condicionales e incondicionales
- Bucles
- Tratamiento de errores
- Otras instrucciones

Instrucciones básicas

Como hemos venido haciendo a lo largo de los capítulos anteriores, comenzaremos por establecer la definición de instrucción: entendemos por instrucción cada una de las **unidades de acción dirigidas al CLR** e incluidas en un método de cualquier clase. Consideramos por tanto que las declaraciones de variables en línea con asignación de valor en el segundo miembro, son en realidad, dos instrucciones: la *declarativa*, dirigida al compilador, y la de *asignación*, que resuelve el CLR al inicializar el tipo (o sea, al llamar al método constructor). Esto, podemos confirmarlo con la ayuda de la herramienta de desensamblado (ILDASM.EXE), utilizando un pequeño ejemplo y comprobando que, efectivamente, lo que nosotros declaramos como instrucción de una única línea, se convierte realmente en dos:

Una vez compilado el programa que muestra el Código fuente 78.

```
using System;

namespace EjDeclaraciones
{
    class Class1
    {
        int x = 3;
        static void Main()
        {
            Console.WriteLine("Instrucción ejecutable");
        }
    }
}
```

Código fuente 78

El programa anterior nos permite poner a prueba el Desensamblador y el Depurador de Visual Studio, para comprobar como se construyen las cosas dentro de un programa en C#. El primer punto de atención, es la instrucción **int x = 3;**, que supone una doble acción: la declaración del campo privado **x**, y la asignación de un valor (3) a dicho campo. La primera forma de inspección nos la da el propio Desensamblador, ya que si visualizamos la ventana de desensamblado:

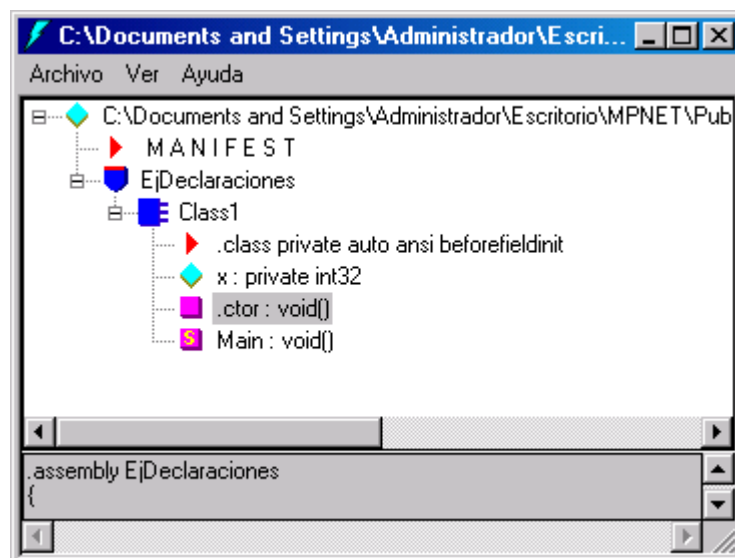


Figura 24. El programa anterior mostrado por el Desensamblador

Apreciamos que existe una entrada separada para la declaración del campo privado **x**, y al abrir el desglose de ese apartado, se nos confirma lo que pensábamos mediante la declaración que muestra la Figura 25.

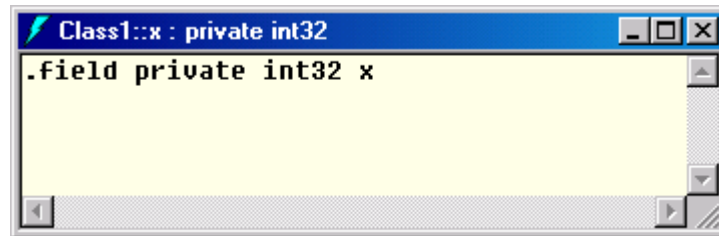


Figura 25. Declaración del campo privado x

Y, además, en el desglose del método constructor, se nos muestra lo que ocurre con el segundo miembro de la instrucción.

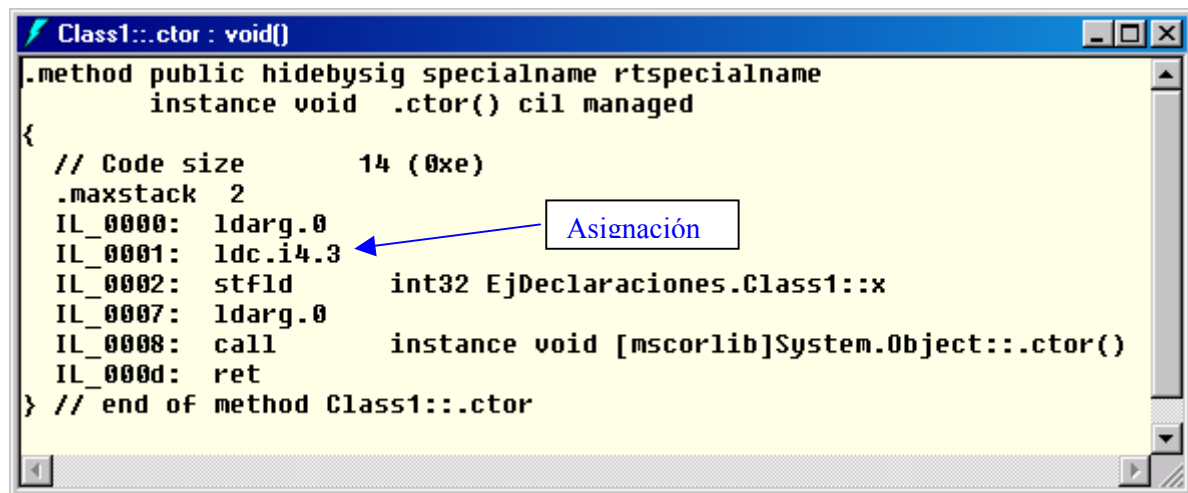


Figura 26. Desensamblador mostrando la asignación en el ejemplo anterior

El gráfico muestra el código MSIL correspondiente a la asignación, dentro de la llamada al método constructor de la clase. Aclarado esto, continuaremos nuestro análisis de las instrucciones básicas: Asignaciones, llamadas a métodos y la instrucción nula.

Asignaciones

Una asignación es, por tanto la instrucción encargada de almacenar un tipo (o dato) dentro de un campo de una clase. Si se trata de un dato, se expresa el valor tal cual, como en el ejemplo anterior, si se trata de un tipo (una instancia de una clase), se espera que el segundo miembro de la igualdad, incluya una llamada al método constructor del tipo. Por ejemplo, en el ejemplo anterior podríamos haber declarado **x** mediante:

```
System.Int32 x = new System.Int32()
```

La diferencia, sería que, en este caso, estaríamos tratando el tipo como un objeto y declarándolo en el montón, de forma que necesitaríamos otra instrucción, dentro de un método, para realizar la asignación (además, como `Class1` no es **static**, necesitamos una instancia de `Class1`):

```
System.Int32 x = new System.Int32()
static void Main()
{
    Class1 c = new Class1();
    c.x = 3
    Console.WriteLine("Instrucción ejecutable");
}
```

Código fuente 80

Nos gusta aclarar este tipo de *sutilezas* del lenguaje, por que más adelante, podría dar origen a muchas posibles malinterpretaciones.

Llamadas a métodos

Las llamadas a métodos siguen siempre la sintaxis:

`<objeto>.<método>(Lista de argumentos);`

donde `<objeto>` puede ser un tipo de la jerarquía de clases de .NET o bien uno propio de usuario, y no es preciso calificar a los métodos con el objeto a que pertenecen, si la llamada se realiza desde el propio tipo.

Instrucción nula

La instrucción nula no realiza ninguna acción. Su sintaxis es simplemente, un punto y coma (;) y es útil su inclusión en casos en los que explícitamente se quiere indicar la ausencia de ejecución en una sentencia o estructura de control.

Bifurcaciones condicionales e incondicionales

Según la teoría general de la información, se trata de las instrucciones que aportan realmente **inteligencia** a un algoritmo, ya que permiten realizar acciones de forma condicional. Comenzamos con la que probablemente, sea (junto a la instrucción de asignación), la instrucción más utilizada en todos los lenguajes: **if**.

Instrucciones *if-else*

Su sintaxis es:

```
if (condición)
    <bloque de instrucciones>
else
    <bloque de instrucciones>
```

donde (condición) es una **expresión evaluable** que devuelve un valor lógico (**true** o **false**).

Nota: No existe la instrucción **Elseif** presente en otros lenguajes, aunque puede sustituirse. Asimismo, es posible extender la instrucción **if**, de modo que puedan controlar varias condiciones, mediante la siguiente **construcción** (que no instrucción) *else-if*:

```
if (condición)
    <bloque de instrucciones>;
else if (Condition_2)
    <bloque de instrucciones>;
else if (Condition_3)
    <bloque de instrucciones>;
...
else
    <bloque de instrucciones>;
```

Código fuente 81

donde el último **else** corresponde al último **if** que no tiene un **else** correspondiente.

Instrucciones switch-case-default, break y goto <etiqueta>.

Para los casos de condiciones múltiples escalonadas, se recomienda la utilización de la sentencia **switch**, que expone el Código fuente 82.

```
switch (expresión)
{
    case expresión-constante:
        instrucción
        instrucción-de-salto
    [default:
        instrucción
        instrucción-de-salto]
}
```

Código fuente 82

donde la *expresión* es cualquier expresión evaluable dentro del contexto de la utilización del tipo, y *expresión-constante*, uno de los posibles valores de *expresión*. La inclusión de la instrucción de salto es obligatoria.

Con esta sintaxis podemos además asignar a varios valores posibles el mismo grupo de acciones, tal y como se muestra en el siguiente ejemplo, traducido del MSDN:

```
// Ejemplo de switch
using System;
class SwitchTest
{
    public static void Main()
    {
        Console.WriteLine("Tamaños: 1=Pequeño 2=Mediano 3=Largo");
```

```
Console.Write("Seleccione tamaño: ");
string s = Console.ReadLine();
int n = int.Parse(s);
int precio = 0;
switch(n)
{
    case 0:
    case 1:
        precio += 25;
        break;
    case 2:
        precio += 25;
        goto case 1;
    case 3:
        precio += 50;
        goto case 1;
    default:
        Console.WriteLine("Selección no válida. Introduzca 1, 2, o 3.");
        break;
}
if (precio != 0)
    Console.WriteLine("Por favor, introduzca {0} céntimos.", precio);
Console.WriteLine("Gracias por su compra.");
}
```

Código fuente 83

En el ejemplo, los valores 0 y 1 reciben la misma asignación. Además, observamos que pueden incluirse instrucciones de bifurcación *goto case <n>* y *break* para controlar el flujo interno **dentro de la estructura**. La instrucción *break* volverá a aparecer en el apartado dedicado a los bucles (sentencias iterativas).

Nótese que la denostada (con razón) sentencia **goto** ha sido severamente limitada en su uso en C#. Solo es posible bifurcar de tal forma si la instrucción **goto** se encuentra dentro del ámbito de la etiqueta. Se permite el uso de una instrucción **goto** para transferir el control fuera de un ámbito anidado, pero no en un ámbito anidado. En cualquier otro caso, se produce un error en tiempo de compilación. El Código fuente 84 compila sin problemas, por que –implícitamente– el goto pertenece a la sentencia **if**:

```
static void Main(string[] args) {
    if (args.Length == 0)
        goto Salir;
    Console.WriteLine(args.Length);

    Salir:
        Console.WriteLine("Terminado");
}
```

Código fuente 84

Instrucciones Iterativas (Bucles)

C# "hereda" de diferentes lenguajes sus instrucciones iterativas: **while**, **do-while**, **for-next** y **foreach-next** (ésta última, heredada de Visual Basic). Dado que son sobradamente conocidas por los programadores, no haremos especial hincapié en ninguna de ellas, salvo aquellos aspectos novedosos

que puedan presentar (como las características de uso de los bucles *foreach*). Nos limitaremos a incluir un ejemplo adaptado por nosotros, a partir de la propia documentación oficial.

Bucles *while*

Tienen la sintaxis ***while (condición) { <bloque de instrucciones> }***, donde el bloque admite sentencias ***break*** para controlar la bifurcación incondicional fuera del bucle.

```
// while
using System;
class PruebaWhile
{
    public static void Main()
    {
        int n = 1;

        while (n < 6)
        {
            Console.WriteLine("Valor de n: {0}", n);
            n++;
        }
    }
}
```

Salida:

```
Valor de n: 1
Valor de n: 2
Valor de n: 3
Valor de n: 4
Valor de n: 5
```

Código fuente 85

Bucles *do-while*

Son muy similares a los anteriores, excepto que la condición se evalúa al final del bucle, con lo que se garantiza que el código interno se ejecute, al menos, una vez.

```
// do-while
using System;
class PruebaDoWhile
{
    public static void Main()
    {
        int x;
        int y = 0;

        do
        {
            x = y++;
            Console.WriteLine(x);
        }

        while(y < 5);
    }
}
```

Salida:

```
0  
1  
2  
3  
4
```

Código fuente 86

Instrucciones *for-next*

Ejecutan bloques de instrucciones mientras la condición especificada sea igual a **true**. Se le puede considerar una herencia directa de la implementación de los lenguajes **C/C++**, tanto en sintaxis como en funcionamiento.

```
// bucles for-next  
using System;  
public class PruebaForNext  
{  
    public static void Main()  
    {  
        for (int i = 1; i <= 5; i++)  
            Console.WriteLine(i);  
    }  
}
```

Salida:

```
1  
2  
3  
4  
5
```

Código fuente 87

Todas las instrucciones de un *for-next* son opcionales. El caso que muestra el Código fuente 88 se utiliza para crear un bucle infinito, del que sólo se puede salir mediante una instrucción **break**.

```
for (;;) {  
    ...  
}
```

Código fuente 88

Bucles *foreach-next*

Se trata de un bucle especial para el recorrido de matrices y colecciones. Aprovecha una característica disponible en toda colección, (el soporte de la interfaz **IEnumerable**) para permitir una sintaxis simplificada en el acceso a cada miembro de la colección.

Una característica algo extraña respecto a la implementación de este bucle en Visual Basic, de donde se puede decir que se inspira, es que su utilización sólo se recomienda en recorridos de lectura, *pero no de asignación*, ya que según la documentación oficial, la asignación podría producir resultados imprevisibles.

La documentación oficial separa claramente la utilización con matrices de la utilización con colecciones, debido al distinto carácter de éstas últimas. El uso con matrices, sencillamente, repite las instrucciones incrustadas para cada elemento de la matriz, como en la adaptación, que aparece en el Código fuente 89, del ejemplo oficial.

```
// Sentencias foreach con matrices
using System;
class Clase1
{
    public static void Main()
    {
        int impar = 0, par = 0;
        int[] arr = new int [] {0,1,2,5,7,8,11};

        foreach (int i in arr)
        {
            if (i%2 == 0)
                par++;
            else
                impar++;
        }

        Console.WriteLine("Encontrados {0} Números Impares, y {1} Números Pares.",
                          impar, par) ;
    }
}
```

Salida:

Encontrados 4 Números Impares, y 3 Números Pares

Código fuente 89

La utilización con colecciones, presenta, sin embargo, más restricciones, según la documentación:

"Para poder iterar en una colección, ésta debe cumplir unos requisitos específicos. Por ejemplo, en la siguiente instrucción foreach:

- foreach (ItemType item in myCollection)

myCollection debe cumplir los siguientes requisitos:

El tipo de la colección:

- Debe ser uno de los tipos interface, class o struct.
- Debe incluir un método de instancia denominado GetEnumerator que devuelva un tipo, por ejemplo, Enumerator (descrito a continuación).

El tipo Enumerator (una clase o estructura) debe contener:

- Una propiedad denominada Current que devuelva ItemType o un tipo que se pueda convertir en él. El descriptor de acceso a la propiedad devuelve el elemento actual de la colección.
- Un método de tipo bool, denominado MoveNext, que incremente el contador de elementos y devuelva true si quedan más elementos en la colección.

Existen tres enfoques en el uso de colecciones:

- Crear una colección mediante las instrucciones anteriores. Esta colección sólo se puede utilizar en un programa escrito en C#.
- Crear una colección genérica mediante las instrucciones anteriores, además de implementar la interfaz *IEnumerable*. Esta colección se puede utilizar en otros lenguajes, como Visual Basic.
- Utilizar una de las colecciones predefinidas en las clases de colecciones."

Como el uso de interfaces será abordado más adelante, al hablar de otros tipos definibles, como los tipos **struct**, vamos a incluir aquí un ejemplo simple de uso de colecciones con el bucle *for-each*, utilizando una de las colecciones de .NET Framework.

```
private void button1_Click(object sender, System.EventArgs e)
{
    int NumControles = 0;
    foreach(Control c in Controls)
    {
        MessageBox.Show(c.GetType().ToString());
        NumControles++;
    }
    MessageBox.Show(NumControles.ToString());
}
```

Código fuente 90

Dentro de una ventana, incluimos 7 controles, (3 de tipo **Label**, 3 de tipo **TextBox** y uno de tipo **Button**), y aprovechamos la característica de los objetos Winform, de que todos los controles que contiene una ventana, se almacenan en la colección **Controls** de dicha ventana. En el ejemplo, se ha asociado al evento Click del botón la ejecución del código fuente que aparece más arriba. En la salida podemos apreciar como el método *GetType()* nos devuelve el tipo de objeto genérico del que hereda cada control:

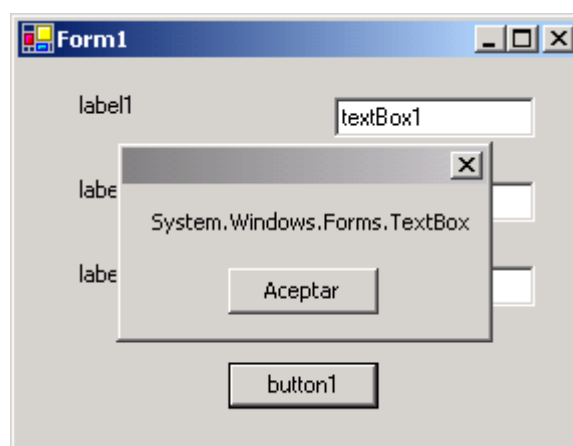


Figura 27. Salida del programa anterior

La variable *NumControles*, sirve de contador de controles. Además, cada vez que se produce una iteración del bucle, se interroga al propio control (representado por la variable *c*) a través de su método **GetType()**, para que nos indique la clase de control de que se trata, y lo muestre en una caja de diálogo

estándar. Finalmente, se muestra el número total de controles. El recorrido de la colección de controles es posible gracias a que todos los controles de un formulario son agregados por el diseñador a la propiedad **ControlCollection**, que implementa la interfaz **IEnumerable**, tal y como comentamos anteriormente.

Cuando queramos averiguar si una clase implementa una determinada interfaz, la manera más sencilla es navegar mediante el Examinador de Objetos hacia las clases base. En el ejemplo, podemos comprobar este extremo, como se ve en la Figura 28.

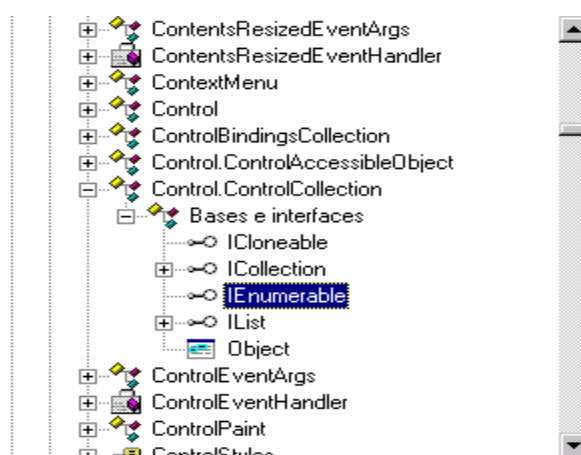


Figura 28. El examinador de objetos mostrando las interfaces soportadas por el tipo ControlCollection

La instrucción *continue*

Al igual que sucede en otros lenguajes donde aparece con frecuencia, el manual oficial nos recuerda que "*se utiliza para indicar que se desea iniciar una nueva iteración de la instrucción envolvente while, do, for o foreach más próxima.*" También se puede utilizar dentro de estructuras *try-catch-finally*.

Sirve para evitar la ejecución de las instrucciones que la siguen dentro de un bucle, bifurcando al final de éste para continuar la iteración. A diferencia de **break**, no produce la salida del bucle. Si se utiliza dentro de una estructura *try-catch-finally*, produce el salto a la instrucción *finally* correspondiente.

Tratamiento de errores

C# dispone de un amplio conjunto de clases especializadas en el tratamiento de errores, de forma que el usuario pueda realizar un ajuste fino y una gestión adecuada en cada caso. En .NET, la palabra error ha sido prácticamente sustituida por **Exception**, que, a su vez, se corresponde con la clase base (**System.Exception**), para el tratamiento de errores y se dispone de una gran cantidad de clases que heredan de esa clase base.

Además, los objetos **Exception**, no sólo garantizan un tratamiento de cualquier tipo de error, sino que permiten la descripción mucho más detallada de los mismos y la escritura de un código más limpio. El CLR y el propio lenguaje nos aseguran poder tratar las excepciones *siempre*, tomando la decisión más adecuada. Como estos objetos pertenecen estrictamente hablando a la jerarquía de clases, no van a ser tratados en éste capítulo, pero si vamos a ver los mecanismos del lenguaje para gestionarlos, tratarlos y, eventualmente, producirlos.

Solamente apuntaremos, de momento, que los objetos **Exception**, admiten tres constructores, que el programador puede utilizar para su creación, tal como nos muestra el Examinador de Objetos.

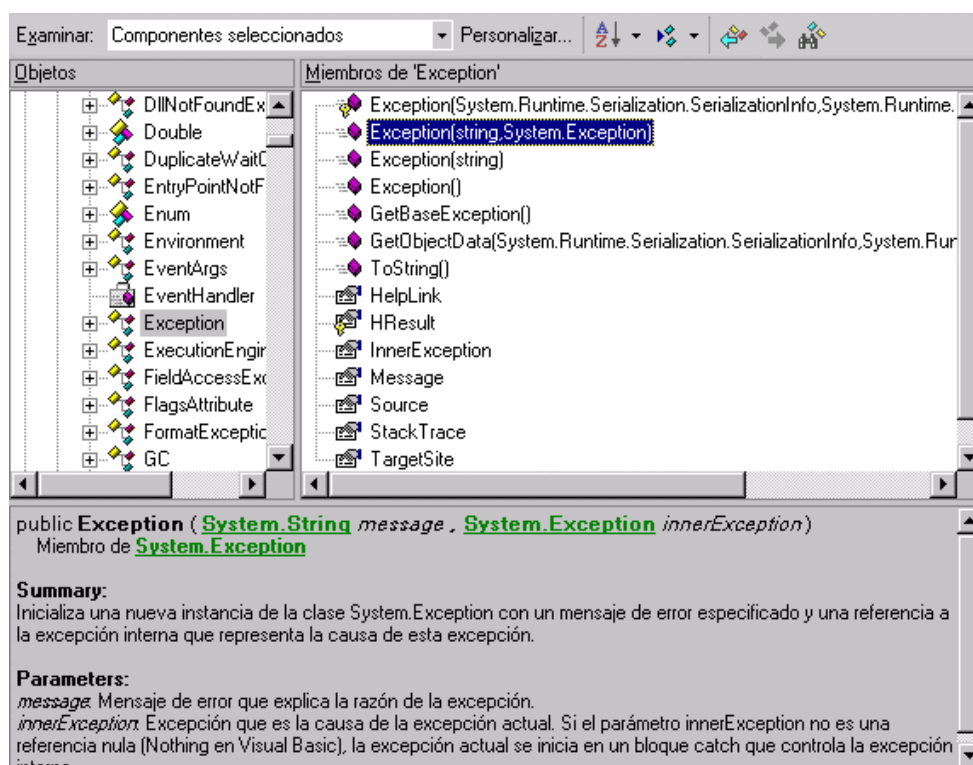


Figura 29. La clase System.Exception, mostrando sus elementos constitutivos.

Esta funcionalidad nos permite, junto a las estructuras adecuadas para ello, implementar una forma adicional de lógica interna de la clase, ya que podemos crear clases que provoquen excepciones que no sean debidas precisamente a un error estricto de tratamiento de información, sino de tipo conceptual, como veremos en el ejemplo siguiente.

La estructura try-catch-finally

En lo referente al lenguaje, la *caza de la excepción*, se hace mediante la estructura **try-catch-finally**. La sintaxis de esta estructura la vemos en el Código fuente 91.

```
try {
    <instrucciones>
}
catch { <exception>
    <tratamiento de la excepción>
}
catch <exception2> {
    <tratamiento de la excepción>
}
...
(opcionalmente...)
[ finally {
    <código que se ejecuta en cualquier caso>
} ]
```

Código fuente 91

Un par de ejemplos, nos permitirán entender el funcionamiento de forma sencilla:

```
using System;

namespace EjExcepciones
{
    class Class1
    {
        [STAThread]
        static void Main(string[] args)
        {
            int x = 0;
            int y = 2/x;
            Console.WriteLine(y);
        }
    }
}
```

Código fuente 92

Como puede comprobarse, la salida de este programa provoca el error que muestra la Figura 30.

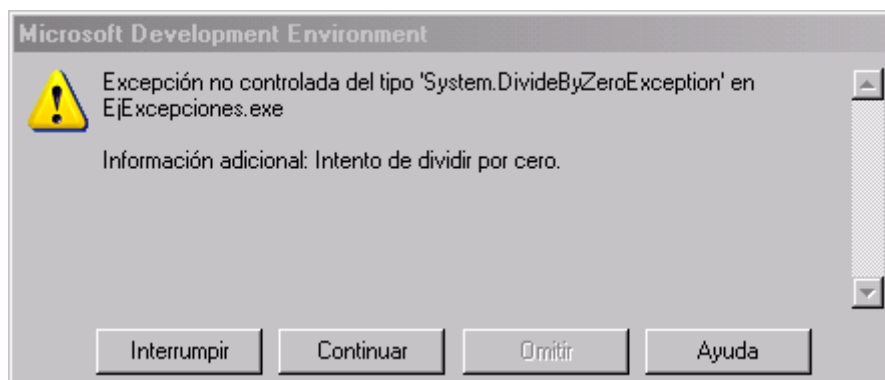


Figura 30. Mensaje de error generado por el CLR al ejecutarse el código anterior

Para realizar el tratamiento del error, debemos de determinar qué líneas del código fuente son susceptibles de producirlo y situar el código dentro de una estructura *try-catch*.

```
static void Main(string[] args)
{
    int x = 0;
    try
    {
        int y = 2/x;
        Console.WriteLine(y);
    }
    catch
    {
        Console.WriteLine("Error de división");
    }
}
```

```
Salida:  
Error de división
```

Código fuente 93

La sintaxis alternativa ***catch (Tipo-Excepción)***, permite ser mucho más específico a la hora de determinar el error producido. Podemos modificar la rutina anterior, para separar el tratamiento de las excepciones de división, del resto de excepciones.

```
try  
{  
    int y = 2/x;  
    Console.WriteLine(y);  
}  
catch (DivideByZeroException) //Tratamiento de la división por 0  
{  
    Console.WriteLine("Error de división");  
}  
catch //Tratamiento genérico (cualquier otro error)  
{  
    Console.WriteLine("Otro error");  
}
```

Código fuente 94

La instrucción throw

De acuerdo con lo indicado antes, también podemos implementar una lógica interna de la clase mediante la generación de excepciones y su tratamiento, gracias a la instrucción ***throw <Exception>***, que permite el lanzamiento de una excepción mediante código.

Por ejemplo, podemos implementar una lógica adicional a una clase ***Trabajador***, para que la edad se reduzca a la edad laboral, comprendida entre 18 y 65 años, lanzando una excepción si se pretende asignar un valor fuera de rango al campo privado de la clase. Para ello, crearemos una clase que herede de *Exception*, y que personalice el mensaje que se produce de acuerdo con el tipo de error.

Otra característica notable es que podemos asignar una variable a la excepción capturada para que sea más fácil su manejo dentro del código fuente.

```
using System;  
  
namespace EjExcepciones  
{  
    class ExRangoEdad : Exception  
    {  
        public string Message  
        {  
            get { return "Error de rango en la edad. Debe de estar entre 18 y 65"; }  
        }  
    }  
  
    class Trabajador  
    {  
        byte edad;  
        public byte Edad  
        {  

```

```
        get { return edad; }
        set {
            if (value < 18 || value > 65 )
            {
                throw new ExRangoEdad();
            }
            else
            {
                edad = value;
            }
        }
    }
}
class EmpleadoTrabajador
{
    [STAThread]
    static void Main()
    {
        Trabajador t = new Trabajador();
        try
        {
            t.Edad = 17;
        }
        catch (ExRangoEdad ex) //Captura Excepción Personalizada
        {
            Console.WriteLine(ex.Message);
        }
        catch //Tratamiento genérico (cualquier otro error)
        {
            Console.WriteLine("Otro error");
        }
        finally
        {
            Console.Read();
        }
    }
}
```

Salida:

Error de rango en la edad. Debe de estar entre 18 y 65

Código fuente 95

Observe el lector que se asigna un valor a la propiedad **Message** de la excepción de usuario, y también una variable a la sentencia de captura, para poder llamar a dicho mensaje de forma más cómoda. El compilador nunca produciría una excepción, ya que el valor 17 es un dato válido de tipo **short**. La excepción la lanza la propiedad de asignación de la propia clase, y es recogida adecuadamente dentro de la primera estructura catch, emitiendo el mensaje informativo.

Otras Instrucciones

Dentro del ámbito que podemos llamar *misceláneo*, caben citar algunas otras instrucciones de propósito diverso:

Instrucción *lock*

Permite bloquear un recurso utilizado en una hebra de ejecución para impedir que otra lo utilice. La zona que se desea proteger es denominada **sección crítica**.

Su sintaxis es:

```
lock (objeto) { bloque de código a proteger }
```

que indica que se desea proteger los recursos de **—objeto—** dentro del bloque de código.

Normalmente, **objeto** será **this**, si se desea proteger una variable de instancia, o bien **typeof(clase)**, si se desea proteger una variable de tipo *static*, o si el bloque se encuentra en un método estático de la clase dada.

Instrucción using

Ya hemos comentado en los primeros capítulos el funcionamiento de la palabra reservada **using** en un contexto de directiva del compilador. Pero **using** puede utilizarse también como instrucción ejecutable, permitiendo obtener uno o varios recursos, ejecutar una instrucción y, a continuación, eliminar el recurso.

La sintaxis declarada para ese recurso es:

```
using ( recurso-llamado )      sentencia-embebida, donde  
recurso-llamado se expresa como  
declaracion-de-variable-local  
expresión
```

Debemos de tener en cuenta la noción de recurso a la hora de utilizar esta sentencia como instrucción. Según el MSDN, “un recurso es una clase o estructura que implementa *System.IDisposable*, que incluye un único método sin parámetros denominado *Dispose*. El código que utiliza un recurso puede llamar a *Dispose* para indicar que ya no lo necesita. Si no llama a *Dispose*, se elimina temporalmente de forma automática como consecuencia de la recolección de elementos no utilizados.”.

Como veremos más adelante, **Dispose()** es el método predeterminado y recomendado para indicar al **Garbage Collector** que puede hacer uso de una referencia a un tipo (genéricamente hablando, de un recurso)

Una instrucción **using** se traduce en tres partes: adquisición, uso y eliminación.

Podemos ver un sencillo ejemplo de uso en el Código fuente 96, en el que un bloque de la forma:

```
using (R r1 = new R()) {  
    r1.F();  
}
```

Código fuente 96

equivale, en realidad, al siguiente conjunto de instrucciones:

```
R r1 = new R();    //Adquisición  
try {  
    r1.F();        // Uso  
}  
finally {
```

```
//Eliminación.  
if (r1 != null) ((IDisposable)r1).Dispose(); }
```

Código fuente 97

también es posible adquirir más de un recurso, mediante una alternativa sintáctica como la que muestra el Código fuente 98.

```
using (R r1 = new R(), r2 = new R() )  
    r1.F();  
    r2.F2();  
}
```

Código fuente 98

En ocasiones, este tipo de bloques se utiliza en la adquisición de recursos gráficos, para asociarlos a controles, y se destruyen una vez utilizados. También puede encontrarse una utilización en las aplicaciones multi-lenguaje, que suelen hacer uso extensivo de ficheros de recursos.

Técnicas adicionales en la construcción de clases

Otros elementos constructores

El lenguaje C# permite, además la construcción de otros tipos de objetos especiales, que complementan sus posibilidades. Algunos son elementos clásicos dentro de la OOP, como la sobrecarga de operadores, otros son novedades de este lenguaje o del entorno .NET, como sucede con los delegados y los atributos.

En éste capítulo, vamos a estudiar los siguientes elementos

- Enumeradores
- Estructuras
- Interfaces
- Indizadores
- Atributos

Tipos Enumerados

Un tipo enumerado define un tipo por valor que sólo puede adoptar uno dentro de una serie de valores posibles, delimitados en la declaración del tipo.

Tienen sentido cuando queremos garantizar que el valor asumido por una variable se mantiene dentro de un conjunto único y restringido de valores posibles, o bien si resulta conveniente otorgar a un conjunto de valores numéricos algún mecanismo descriptivo para acceder a ellos mediante su descripción y no su valor numérico.

Para declarar un tipo enumerado se utiliza la palabra reservada **enum** como vemos en el Código fuente 99.

```
enum <NombreTipo> { Lista de valores,...}
```

Código fuente 99

Donde la lista de valores asume por defecto un conjunto de valores **int** que comienzan por 0. Se puede modificar el tipo de valor numérico asignado a cada elemento indicándolo así en la declaración, así como asignar alias a los valores de alguno de los elementos de enumeración, o forzar a que el elemento de comienzo sea diferente de 0, como en el Código fuente 100.

```
using System;

namespace EjEnum
{
    enum Estaciones : short {Primavera = 1, Verano , Estío = Verano, Otoño,
    Invierno = 100 }

    class Class1
    {
        [STAThread]
        static void Main(string[] args)
        {
            Estaciones e = Estaciones.Primavera;
            Estaciones f = Estaciones.Verano;
            Estaciones g = Estaciones.Estío;
            Estaciones h = Estaciones.Invierno;
            Console.WriteLine(e.GetTypeCode());
            Console.WriteLine(f.ToString());
            Console.WriteLine(g);
            Console.WriteLine((short)g);
            Console.WriteLine((short)h);
            Console.Read();
        }
    }
}
```

Salida

```
Int16
Estío
Estío
2
100
```

Código fuente 100

Note el lector la conversión automática que se realiza llamando al método ToString() del valor enumerado en las líneas 2 y 3, así como el último valor asignado (100), y el hecho de que la última declaración de alias (Estío) es la que prevalece de forma predeterminada.

Estructuras (structs)

Una estructura **struct**—en C# es un tipo por valor que se comporta en todos los demás aspectos como una clase: puede tener campos, métodos y constructores. Resultan adecuados para representar objetos de poca complejidad, en los que el valor es lo más importante, y no su funcionalidad. Además, al almacenarse en la pila permiten un rendimiento óptimo en procesos que lo necesiten, o en operaciones iterativas. Los números complejos, los puntos de un sistema de coordenadas o los pares de valores clave de un diccionario son buenos ejemplos de estructuras.

Las diferencias fundamentales entre un **struct** y una clase son:

- **Declaración de constructores:** Un tipo *struct* no admite constructores por defecto (sin parámetros) definidos por el usuario, ya que el compilador *siempre* crea uno para ellos. Si que se admite, en cambio la creación de constructores personalizados no predeterminados, como se ve en el ejemplo que sigue.
- Para las estructuras no existe herencia. A diferencia de las clases, una estructura no puede heredar de otra estructura o clase, ni puede ser la base de una clase. Sin embargo, las estructuras heredan de la clase **System.ValueType**, y —esta, a su vez— de **System.Object**. Una estructura puede implementar interfaces del mismo modo que las clases.
- Se puede utilizar una estructura sin inicializar (por que el CLR llamará a su constructor predeterminado). Según esto, si en lugar de *struct* se declarase *Estructura* como clase, el Código fuente 101 fallaría, pero no es así.

```
namespace EjEnum
{
    struct Estructura
    {
        public int x;
        public string Lugar;
        public Estructura (string Lugar, int x)
        //Constructor de usuario
        {
            this.Lugar = Lugar; //Sí puede usarse-this-
            this.x = x;
        }
    }
    public class Class2
    {
        static void Main()
        {
            Estructura w = new Estructura(); //Constructor predeterminado
            Console.WriteLine("Lugar: {0} , w: {1} ", w.Lugar, w.x);
            Estructura e = new Estructura("Eidos", 3); //Constructor de
            usuario
            e.x = 1;
            Console.WriteLine("Lugar: {0} , x: {1} ", e.Lugar, e.x);
            Console.Read();
        }
    }
}
```

Salida
Lugar: , x: 0
Lugar: Eidos , x: 1

Código fuente 101

- Cuando se crea un objeto **struct** mediante el operador **new**, también se llama al constructor apropiado. Si no se utiliza **new**, los campos permanecerán sin asignar y el objeto no se podrá utilizar hasta haber inicializado todos los campos.

En el Código fuente 102, vemos un posible efecto colateral de esta situación.

```
using System;

namespace EjEnum
{
    struct Estructura
    {
        public int x;
        public string Lugar;
        public Estructura (string Lugar, int x)
        {
            this.Lugar = Lugar;
            this.x = x;
        }
    }
    public class Class2
    {
        static void Main()
        {
            Estructura w; //No se llama a ningún constructor
            w.Lugar=""; //Compila correctamente
            //La salida falla si no incluimos la siguiente orden:
            w.x = 1; //Inicializados los dos campos -> OK
            Console.WriteLine("Lugar: {0} , w: {1} ", w.Lugar, w.x);
            Console.Read();
        }
    }
}
```

Compilación:
Uso del campo 'x', posiblemente no asignado.
 Tras la modificación, se ejecuta correctamente:
 Lugar: Eidos , w: 1

Código fuente 102

Hay que tener en cuenta que el compilador discrimina adecuadamente cada tipo ubicando espacio en el montón (*heap*), si se trata de una clase, o en la pila (*stack*) si se trata de una estructura.

Interfaces

Una interfaz es un tipo abstracto que solo contiene declaraciones formales. No es algo nuevo del entorno .NET sino que existen casi desde que los primeros lenguajes orientados a objetos hicieron su aparición. En Windows, la tecnología COM les popularizó como mecanismo para reforzar las posibilidades de comunicación entre tipos.

Se define una interfaz como un contrato que establece un conjunto estándar de propiedades, métodos y eventos que se encontrarán en cualquier clase que implemente la interfaz, pero en sí misma, la interfaz no implementa ningún método, solamente declara miembros del tipo.

Muchas veces, el uso de las interfaces se recomienda para hacer que una aplicación sea compatible con algún aspecto del sistema operativo, o del lenguaje de programación utilizado, o ambos. Algunos usos típicos pueden ser la modificación de una clase para que los usuarios de la misma puedan recorrerla como si fuera una colección, o el cumplimiento de normas para que otro objeto pueda mostrar contenidos de la clase.

Imagine el lector la hipotética –o no tanto- clase **Cliente**. Podría ser de interés en algunos procesos que pudiéramos recorrer una colección de clientes utilizando la intuitiva sintaxis del bucle *foreach-next*. Pero, de forma predeterminada, nuestra clase no está preparada para ello: si queremos añadirle esa funcionalidad, tenemos que hacer que implemente la interfaz **IEnumerable**, que permite que el bucle efectúe llamadas a la clase que le devuelvan los datos necesarios para poder recorrerla.

A efectos de nuestro manual, vamos a comentar un ejemplo que aparece en uno de los tutoriales de .NET, que resulta más fácil de comprender en una primera aproximación. En este ejemplo, se juega con una característica de .NET Framework, cual es que la capacidad de comparar objetos se suministra mediante la interfaz llamada **IComparable**, que define un método de comparación generalizado que un tipo debe implementar para crear un método de comparación propio. El *compromiso*, en este caso, es implementar un único método, **CompareTo**, que recibe como argumento un objeto. Su implementación en una clase no predeterminada (un tipo no estándar), nos permite **definir el concepto de comparación, creando un mecanismo operativo en el cuerpo del método CompareTo que permita establecer si otro tipo cualquiera de la misma clase está antes o después en el criterio de ordenación establecido por el programador**.

El valor devuelto por CompareTo tiene los siguientes significados:

Valor	Significado
Menor que cero	El valor de esta instancia es menor que el de <i>obj</i> .
Cero	El valor de esta instancia es igual al de <i>obj</i> .
Mayor que cero	El valor de esta instancia es mayor que el de <i>obj</i> .

Tabla 18

De ahí el concepto de compromiso. Es como si el lenguaje nos dijera: permito que realices comparaciones de tipo no estándar, a cambio de que tú mismo establezcas lo que significa comparar dos objetos de ese tipo. Esto podría ser válido para la clase Clientes, mencionada antes, pero es más útil en clases de naturaleza algo más compleja, en que la comparación no se reduce a un campo de la clase, sino a varios.

Este es el caso de los tipos **Point** que se utilizan en muchos contextos del .NET Framework para establecer ubicaciones visuales de objetos, puntos de trazado gráfico, etc. Si tuviésemos una colección de puntos (o una matriz de puntos en este caso), sería de utilidad poder ordenarlos, por ejemplo, utilizando el método Sort de la clase Array, pero dicho método exige que la clase implemente la interfaz **IComparable**. La solución, en este ejemplo, es calcular la distancia al origen de coordenadas y devolver su valor (con signo), acorde con el significado de la Tabla 18.

```
// Autor: Ejemplos MSDN
// Microsoft
// Traducción y comentarios: Marino Posadas
```

```
using System;

class Point : IComparable {
    public Int32 x, y;

    public Point(Int32 x, Int32 y) {
        this.x = x;
        this.y = y;
    }

    // CompareTo se define en la interfaz IComparable
    public Int32 CompareTo(Object other) { //other es otro objeto de la misma clase
        Point p = (Point) other;
        //Se calcula la hipotenusa del triángulo formado por los dos puntos,
        // y ese es el valor de la distancia al origen
        Double thisDistanceFromOrigin = Math.Sqrt(x * x + y * y);
        Double otherDistanceFromOrigin = Math.Sqrt(p.x * p.x + p.y * p.y);

        return(Math.Sign(thisDistanceFromOrigin - otherDistanceFromOrigin));
    }

    public override String ToString() {
        return(String.Format("{0}, {1}", x, y));
    }
}

class App {
    static void Main() {
        Point[] points = new Point[5];
        points[0] = new Point(2, 2);
        points[1] = new Point(3, 2);
        points[2] = new Point(2, 3);
        points[3] = new Point(7, 8);
        points[4] = new Point(0, 1);

        Array.Sort(points);

        // Mostrar todos los elementos del array
        for (Int32 i = 0; i < points.Length; i++)
            Console.WriteLine("Point {0}: {1}", i, points[i]);

        Console.Write("Pulse Intro para cerrar la ventana...");
        Console.Read();
    }
}
```

Salida

```
Point 0: (0, 1)
Point 1: (2, 2)
Point 2: (2, 3)
Point 3: (3, 2)
Point 4: (7, 8)
Pulse Intro para cerrar la ventana...
```

Código fuente 103

Como puede verse, el método **CompareTo** devuelve un valor negativo, si es menor, 0 si es igual, y positivo si es mayor el punto actual con el comparado. Más adelante el método **Array.Sort()**, realiza la ordenación.

Indizadores (Indexers)

Un indizador es un miembro que permite que el tipo pueda ser tratado como una matriz por otras clases.

El indizador se codifica como un método con el nombre de la palabra reservada *-this-*, un argumento de índice, y unas propiedades de lectura y escritura, para indicar a qué elemento se accede y qué elemento se modifica, cuando un tipo es accedido de esa forma.

El método declarado como indizador puede incluir cualquier conjunto de atributos, al igual que los modificadores de acceso admitidos para otros métodos, y los modificadores especiales **new**, **virtual**, **override**, **sealed**, **abstract** y **extern**. Por lo demás admiten las mismas consideraciones que cualquier otro método, excepción hecha del modificador **static** que no es válido.

Aunque conceptualmente, un indizador se asemeja a una propiedad, existen un número de diferencias que merecen la pena resaltar:

Los indizadores y las propiedades son similares conceptualmente, pero tienen las siguientes diferencias:

- Una propiedad se identifica por su nombre, mientras que un indizador se identifica por su firma.
- Para tener acceso a una propiedad se utiliza un *nombre simple* o un *acceso de miembro*, pero a un elemento de indizador se tiene acceso a través de un *acceso de elemento*.
- Una propiedad puede ser un miembro **static** y un indizador siempre es un miembro de instancia.
- Un descriptor de acceso **get** de una propiedad corresponde a un método sin parámetros, pero un descriptor de acceso **get** de un indizador corresponde a un método con la misma lista de parámetros formales que el indizador.
- Un descriptor de acceso **set** de una propiedad corresponde a un método con un solo parámetro denominado **value**, pero un descriptor de acceso **set** de un indizador corresponde a un método con la misma lista de parámetros formales que el indizador y un parámetro adicional denominado **value**.
- Supone un error en tiempo de compilación que un descriptor de acceso de un indizador declare una variable local con el mismo nombre que un parámetro del indizador.
- En una declaración de propiedad reemplazada, se tiene acceso a la propiedad heredada utilizando la sintaxis **base.P**, donde **P** es el nombre de la propiedad. En una declaración de indizador reemplazado, se tiene acceso al indizador heredado utilizando la sintaxis **base[E]**, donde **E** es una lista de expresiones separadas por comas.

De hecho podemos ver en la Tabla 19 comparativa entre propiedades e indizadores, para aclarar más esta situación.

Propiedad	Indizador
Se identifica por su nombre.	Se identifica por su firma.
Se tiene acceso a su valor mediante un nombre simple o un acceso a miembro.	Se tiene acceso a su valor mediante un acceso a elemento.
Puede ser un miembro estático o de instancia.	Debe ser un miembro de instancia.
Los descriptores de acceso get de una propiedad no tienen parámetros.	Los descriptores de acceso get de un indizador tienen la misma lista de parámetros formales que el indizador.
Los descriptores de acceso set de una	Los descriptores de acceso set de un indizador tienen

propiedad contienen el parámetro implícito value .	la misma lista de parámetros formales que el indizador, además del parámetro value .
---	---

Tabla 19. Tabla comparativa de propiedades e indizadores, según el MSDN

Básicamente, los indizadores nos permiten simplificar la sintaxis con la que operamos cuando manejamos matrices de objetos. (No obstante, no debemos confundir esta característica con la implementación de la interfaz *Innumerable*. Esta implementación es obligada si se quiere recorrer un conjunto utilizando el bucle *foreach-next*).

En el Código fuente 104, vamos a recrear la clase *Peloton* con un indizador, para que se pueda recorrer por el campo *nombre* y un segundo para el campo *clasificacion*.

```
using System;
using System.Collections;

namespace EjIndizadores
{
    class Peloton
    {
        private string [] Corredores = new string[10];

        public string this[int indice]          // Indizador para el nombre
        {
            get {
                // Comprobar los limites del array y que no esté vacío
                if ((indice < 0 || indice >= 10) || Corredores[indice] == null)
                    return "";
                else
                    return Corredores[indice].ToString();
            }

            set {
                if (!(indice < 0 || indice >= 10))
                    Corredores[indice] = value;
            }
        }
    }

    class Clasificar
    {
        [STAThread]
        static void Main(string[] args)
        {
            Peloton p = new Peloton();
            p[1] = "A. Casero";
            p[2] = "L. Armstrong";
            p[3] = "J. Ulrich";

            for (int x = 1; x < 10; x++)
            {
                Console.WriteLine("Ciclista: {0}", p[x]);
            }

            Console.Read();
        }
    }
}
```

Salida:

```
Ciclista: A. Casero
Ciclista: L. Armstrong
Ciclista: J. Ulrich
Ciclista:
```

```
Ciclista:
Ciclista:
Ciclista:
Ciclista:
Ciclista:
Ciclista:
```

Código fuente 104

Recuérdese que una matriz declarada de esa forma, se inicializa a valores **null**, por lo que debemos comprobar la existencia de un valor antes de leer el contenido. En el ejemplo, se ignora el elemento 0 de la matriz, y luego, solo se da valor a 3 de ellos. La utilidad reside en que —aunque la instancia *p* de la clase *Peloton* no ha sido declarada como matriz—, su programación se hace como si lo fuera.

Atributos

Aunque C# es un lenguaje de los llamados imperativos, consigue una gran flexibilidad de comportamiento gracias a algunos elementos de tipo declarativo, como los atributos. Un atributo es un elemento que califica a un tipo o a un miembro de un tipo para establecer un comportamiento determinado en tiempo de ejecución.

La información de atributos puede ser, posteriormente, recogida gracias a la capacidad suministrada por una librería llamada **Reflection** en .NET, que permite a un tipo obtener información sobre sus Metadatos o los de otro tipo.

Atributos predeterminados y atributos de usuario

.NET Framework define un conjunto amplio de atributos predeterminados, a los que ha dotado de significados especiales, pero, quizás, la auténtica potencia de los atributos se consiga mediante la posibilidad de crear atributos personalizados, o de usuario. Para ello, un usuario puede crear atributos escribiendo una clase que herede de *System.Attribute*, e implementando los miembros que se necesiten para su uso posterior.

Además, y como era de esperar a partir de ese planteamiento, el usuario puede declarar sus propias clases de atributo utilizando la herencia y la mayoría de las características ya estudiadas de la orientación a objetos, lo que le permite alterar o añadir funcionalidad a atributos ya funcionales.

Un ejemplo simple de utilización podemos verlo en el Código fuente 105.

```
#define DEBUG
using System.Diagnostics;
class Class1
{
    [Conditional("DEBUG")]
    public static void M() {
        Console.WriteLine("Ejecutado el método M de Class1");
    }
}
class Class2
{
    public static void Test() {
```

```
        Class1.M();  
    }  
}
```

Código fuente 105

El ejemplo, define una declaración de compilación condicional, asociada al identificador `DEBUG`. Como el método `M` de `Class1`, está calificado como `[Conditional("DEBUG")]`, si se omite la definición del identificador la llamada no se efectuará.

Antes de continuar analizando los atributos, es necesario que sepamos algo más del funcionamiento de una clase de la jerarquía que juega un papel fundamental en el manejo de atributos: **Type**.

La clase Type

Se trata de una clase abstracta que suministra un amplísimo conjunto de métodos para el trabajo con técnicas de tipo **Reflection**. De acuerdo con esto, existen numerosas vías por las que podemos obtener una instancia de la clase **Type**. En el Código fuente 106 se ilustran 3 de ellas.

```
using System;  
class Class3  
{  
    public Class3()  
    {  
    }  
}  
class Class2  
{  
    public Class2()  
    {  
    }  
}  
class Prueba  
{  
    static void Main(string[] args)  
    {  
        //Método 1: usar el método GetType() presente en cualquier tipo  
        Class2 c = new Class2();  
        Type t = c.GetType();  
        Console.WriteLine(t.Name);  
        //Método 2: usar el método estático GetType() de la propia clase  
        // Type, pasándole como argumento el nombre textual del tipo  
        Type t2 = null;  
        t2 = Type.GetType("Class2");  
        Console.WriteLine(t2.Name);  
        //Método 3  
        Type t3 = typeof(Class2);  
        Console.WriteLine(t3.Name);  
        Console.Read();  
    }  
}
```

Salida

```
Class2  
Class2  
Class2
```

Código fuente 106

La potencia de la clase `Type` es enorme, como se deduce de sus miembros. Existen métodos para averiguar **cualquier cosa que el tipo almacene en sus Metadatos**, e incluso, para poder hacer una invocación tardía de ellos (**Late Binding**) si lo creemos oportuno.

Por ejemplo, imaginemos que una clase dispone de campos, métodos, soporta una interfaz y hereda de otra clase. Podemos interrogar al tipo correspondiente para que nos dé la lista de todos sus miembros (un poco al estilo de las tablas virtuales de COM, pero pudiendo reunir mucha más información, y sin necesitar leer nada del Registro del Sistema).

En el Código fuente 107 vemos un mecanismo de interrogación basado en **Reflection**, respecto al tipo mencionado.

```
// Autor: Marino Posadas
// Grupo EIDOS
// 2002

using System;
using System.Reflection;
interface INsoportable
{
    void NoPuedo();
}
class Class1
{
    static int x;
    public Class1(int x)
    {
        Class1.x = x;
    }
}

// Class 2 hereda de Class1 y soporta INsoportable
class Class2 : Class1, INsoportable
{
    public int y;
    static int z;
    public Class2(int y): base(y)
    {
        this.y = y;
    }

    public void NoPuedo()
    {
        Console.WriteLine("Que no puedorrr...");
    }
    public int LeerY()
    {
        return y;
    }
}

class Prueba
{
    static void Main(string[] args)
    {
        //Método 1: usar el método GetType() presente en cualquier tipo
        Class2 c = new Class2(5);
        Type t = c.GetType();

        //Ahora podemos averiguar un buen conjunto de información
        //acerca del tipo
        Console.WriteLine("Nombre del tipo:{0}", t.Name);
    }
}
```

```

        Console.WriteLine("¿Es una clase?: {0}", t.IsClass);
        Console.WriteLine("¿Es una interfaz?: {0}", t.IsInterface);
        Console.WriteLine("¿Es un tipo por valor?: {0}", t.IsValueType);
        Console.WriteLine("Tipo del que hereda: {0}", t.BaseType);
        // Información sobre constructores
        ConstructorInfo[] ListaConstructores = t.GetConstructors();
        Console.WriteLine("N° de Constructores: {0}",
ListaConstructores.Length);
        foreach (ConstructorInfo ci in ListaConstructores)
        {
            Console.WriteLine("Tipo: {0}, Nombre: {1} ", ci.MemberType,
ci.Name);
        }

        // Información sobre métodos
        MethodInfo[] ListaMetodos = t.GetMethods();
        Console.WriteLine("N° de Métodos: {0}", ListaMetodos.Length);
        foreach (MethodInfo mi in ListaMetodos)
        {
            Console.WriteLine("Tipo: {0}, Nombre: {1} ", mi.MemberType,
mi.Name);
        }

        // Información general sobre interfaces
        Type[] ListaInterfaces = t.GetInterfaces();
        Console.WriteLine("N° de Interfaces: {0}", ListaInterfaces.Length);
        foreach (MemberInfo mi in ListaInterfaces)
        {
            Console.WriteLine("Tipo: {0}, Nombre: {1} ", mi.MemberType,
mi.Name);
        }

        // Información general sobre miembros de la clase
        MemberInfo[] ListaMiembros = t.GetMembers();
        Console.WriteLine("N° de Miembros: {0}", ListaMiembros.Length);
        foreach (MemberInfo mi in ListaMiembros)
        {
            Console.WriteLine("Tipo: {0}, Nombre: {1} ", mi.MemberType,
mi.Name);
        }

        Console.Read();
    }
}
//-----
//Salida
//-----
Nombre del tipo: Class2
¿Es una clase?: True
¿Es una interfaz?: False
¿Es un tipo por valor?: False
Tipo del que hereda: Class1
N° de Constructores: 1
Tipo: Constructor, Nombre: .ctor
N° de Métodos: 6
Tipo: Method, Nombre: NoPuedo
Tipo: Method, Nombre: GetHashCode
Tipo: Method, Nombre: Equals
Tipo: Method, Nombre: ToString
Tipo: Method, Nombre: LeerY
Tipo: Method, Nombre: GetType
N° de Interfaces: 1
Tipo: TypeInfo, Nombre: INsoportable
N° de Miembros: 8
Tipo: Field, Nombre: y
Tipo: Method, Nombre: NoPuedo

```

```
Tipo: Method, Nombre: GetHashCode
Tipo: Method, Nombre: Equals
Tipo: Method, Nombre: ToString
Tipo: Method, Nombre: LeerY
Tipo: Method, Nombre: GetType
Tipo: Constructor, Nombre: .ctor
```

Código fuente 107

Por tanto, **Reflection** y sus derivados representan una de las características más poderosas de .NET Framework, al extender el concepto de autodescripción, ya introducido en COM+, con el de los metadatos, mucho más independiente y con más información.

Esta capacidad se extiende a los propios módulos de ensamblado, que poseen sus propios metadatos, que pueden ser llamados por otro programa, para interrogarle acerca de sus clases y tipos contenidos. Vamos a ver el funcionamiento de esta característica a través de un ejemplo en el que se lee la información general de tipos del programa anterior, directamente del módulo de ensamblado EjReflection.EXE (en la llamada, podemos utilizar el “nombre amigable”, que no requiere de la extensión del fichero:

```
using System;
using System.Reflection;

namespace EjReflection2
{
    class Class1
    {
        public static void ListaContenidos(Assembly a)
        {
            Console.WriteLine("Lista de todo el contenido del Assembly: {0}", a.FullName);
            Type [] tipos = a.GetTypes();
            foreach ( Type t in tipos )
            {
                Console.WriteLine("Tipo: {0}", t);
                foreach( MethodInfo m in t.GetMethods())
                {
                    Console.WriteLine(m.Name);
                }
            }
            Console.WriteLine("-----");
        }
        Console.Read();
    }

    [STAThread]
    static void Main()
    {
        Assembly a = null;
        try
        {
            a = Assembly.Load("EjReflection");
        }
        catch (Exception e)
        {Console.WriteLine("Fichero no encontrado"); }

        ListaContenidos(a);
    }
}

Salida:
```

```

Lista de todo el contenido del Assembly: EjReflection, Version=1.0.837.23043, Cu
lture=neutral, PublicKeyToken=null
Tipo: INsoportable
NoPuedo
-----
Tipo: Class1
GetHashCode
Equals
ToString
GetType
-----
Tipo: Class2
NoPuedo
GetHashCode
Equals
ToString
LeerY
GetType
-----
Tipo: Prueba
GetHashCode
Equals
ToString
GetType

```

Código fuente 108

Incluso podemos efectuar llamadas a los métodos de una clase “pasiva”, dentro del tipo, mediante un mecanismo que activa una instancia de dicha clase a petición (**Activator**), y otro que permite la invocación de métodos (**Invoke()**), y con ello la implementación de la característica “late binding”.

Por ejemplo, supongamos que queremos extender la capacidad de nuestro ejemplo anterior de forma que podamos llamar realmente al método *NoPuedo()* de **Class2**. Podríamos utilizar la siguiente modificación en nuestro código:

Por un lado, simplificamos el código del objeto a llamar, por razones didácticas, y lo situamos en una librería, que vamos a llamar **LibreriaPower**. El ensamblado final, será, por tanto, una DLL con el siguiente contenido:

```

using System;

namespace LibreriaPower
{
    class Class1
    {
        public void NoPuedo()
        {
            Console.WriteLine("Que no puedorrrrrr....");
        }
    }

    // Class 2 hereda de Class1
    class Class2 : Class1
    {
        public void YoTampocoPuedo()
        {
            Console.WriteLine("Que yo tampocorr....");
        }
    }
}

```

```

    }
}

```

Código fuente 109

A continuación, y sin hacer referencia alguna en el Editor de Proyectos de Visual Studio, podemos utilizar el siguiente código para:

- Ganar acceso dinámico a la librería desde la ubicación donde se encuentre
- Hacer un recorrido de sus contenidos, igual que en los ejemplos anteriores
- Crear una instancia de una de sus clases (*Class2*), y finalmente
- Llamar a uno de sus métodos (*NoPuedo*) y comprobar el resultado de su ejecución:

Por consiguiente, con algunas modificaciones, tendríamos el Código fuente 110.

```

using System;
using System.Reflection;

namespace EjReflection2
{
    class Class1
    {
        public static void ListaContenidos(Assembly a)
        {
            Console.WriteLine("Lista de todo el contenido del Assembly:
{0}", a.FullName);
            Type [] tipos = a.GetTypes();
            foreach ( Type t in tipos )
            {
                Console.WriteLine("Tipo: {0}", t);
                foreach( MethodInfo m in t.GetMethods())
                {
                    Console.WriteLine(m.Name);
                }
                Console.WriteLine("-----");
            }
            Console.Read();
        }

        [STAThread]
        static void Main()
        {
            Assembly a = null;
            try
            {
                // Cargamos la librería desde su ubicación original
                a = Assembly.LoadFrom(@"F:\Documents and
Settings\mposadas\Desktop\Lab\LibroCS\LibreriaPower\obj\Debug\LibreriaPower.dll");
            }
            catch (Exception e)
            { Console.WriteLine("Fichero no encontrado"); }

            ListaContenidos(a);
            Type Ej = a.GetType("LibreriaPower.Class2");

```

```

        object obj = Activator.CreateInstance(Ej);
        MethodInfo m = Ej.GetMethod("NoPuedo");
        m.Invoke(obj, null);    //Llamada al método
    }
}
}

```

Salida:

```

Lista de todo el contenido del Assembly: LibreriaPower, Version=1.0.837.25786, C
ulture=neutral, PublicKeyToken=null
Tipo: LibreriaPower.Class1
GetHashCode
Equals
ToString
NoPuedo
GetType
-----
Tipo: LibreriaPower.Class2
GetHashCode
Equals
ToString
YoTampocoPuedo
NoPuedo
GetType
-----
Que no puedorrrrrrr....

```

Código fuente 110

Las diferencias respecto al código anterior, residen en que —esta vez— convertimos una clase “pasiva”, que permite su acceso a metadatos, en una clase “activa”, que puede ser llamada a petición.

El primer objetivo se consigue mediante la llamada al método **CreateInstance** de la clase **Activator**, que permite crear una instancia de una de las clases referenciadas por un tipo. En el segundo caso, cuando ya tenemos todo listo, el método **Invoke()** permite la llamada real al método.

Hay que tener en cuenta que, aunque en este caso el método no recibía argumentos, podíamos haber realizado la llamada de la misma forma (si ese fuera el caso), con sólo obtener los valores de los parámetros del método a partir de la función **GetParameters()**, de un objeto **MethodInfo**, como los utilizados más arriba, y haber pasado el array con los valores de parámetros como segundo argumento del método **Invoke()**, algo como lo que muestra el Código fuente 111.

```

// Si m es un objeto MethodInfo
ParameterInfo[] p = m.GetParameters();
P[0] = ... un valor ...
P[1] = ... otro valor ...
P[n] = ... enésimo valor ...
m.Invoke(obj, p);

```

Código fuente 111

Volviendo a los atributos

Con todas estas posibilidades en cuenta, podemos ver el Código fuente 112, donde se muestra la utilización algo más elaborada del uso de atributos, pero, esta vez, creando un atributo de usuario, que hereda de la clase **Attribute**:

```

using System;

namespace EjAtributos
{
    [AttributeUsage(AttributeTargets.All)]
    public class AyudaEnMetadatos: Attribute        //Hereda de Attribute
    {
        public AyudaEnMetadatos(string url)
        {
            this.url = url;
        }
        public string Topic = "El tópico";    //Valor por defecto
        private string url;
        public string Url
        {
            get { return url; }
        }
    }

    En class1, se cambia el valor por defecto de Topic
    [AyudaEnMetadatos("http://localhost/Ayudorrrr.htm", Topic = "Item")]
    class Class1
    {
        [AyudaEnMetadatos("http://localhost/Ayudas.htm")]
        public void F() {}
    }
    class PruebaAtributos
    {
        [STAThread]
        static void Main(string[] args)
        {
            Type type = typeof(Class1);
            object[] arr = type.GetCustomAttributes(typeof(AyudaEnMetadatos),true);
            if (arr.Length == 0)
                Console.WriteLine("Class1 has no Help attribute.");
            else
            {
                AyudaEnMetadatos ha = (AyudaEnMetadatos) arr[0];
                Console.WriteLine("Url = {0}, Topic = {1}", ha.Url, ha.Topic);
                Console.Read();
            }
        }
    }
}

```

Código fuente 112

Técnicas avanzadas

Independientemente de las características propias de la construcción y manejo de tipos, C# aporta palabras reservadas que permiten el manejo de un buen número de características avanzadas, que aportan una aproximación más simple a problemas típicos del desarrollo. Esto es especialmente útil en aquellas situaciones que era necesario abordar a través de uso de API's del sistema, o utilizar librerías de terceros.

Uno de esos problemas es la programación conocida como anteriormente como multihebra (o multihilo), y ahora denominada por el MSDN como **subprocesamiento**, esto es, la capacidad de un proceso en ejecución para lanzar más de una secuencia de instrucciones simultáneamente, llevando el control de su ejecución, y permitiendo todas las hebras de un proceso compartan el mismo conjunto de recursos.

No se debiera nunca de abusar del uso de subprocesos, ya que un número excesivo de ellos complica considerablemente la depuración, y también el rendimiento. Debiéramos de tener esto en mente a la hora de utilizarlos, y recordar que un subproceso siempre debe aportar mejoras en el rendimiento o aprovechamiento de los tiempos de espera.

Subprocesos y Dominios de Aplicación

Dado que el trabajo con subprocesos en la vida real, enfoca un problema típico de desarrollo, y no del lenguaje en sí, abordaremos la creación de subprocesos concretos en aplicaciones en otra obra dedicada al ciclo de desarrollo, pero, no obstante, vamos a realizar una primera aproximación aquí, con objeto de que el lector pueda tener una idea suficiente de su funcionamiento, apoyada con ejemplos.

En Win32 (el núcleo común de las API's de Windows para todos los sistemas operativos de 32 bits), el nivel de protección de un subproceso se establece mediante una serie de recursos propios del sistema, tales como los *mutexes* o los semáforos, al objeto de sincronizar el acceso a datos compartidos. Esto hace que, lo que se gana por un lado, se pierda, en cierto sentido, por el otro, ya que el riesgo de corrupción de datos compartidos se vuelve muy alto, y la complejidad que significa la programación correcta de estos mecanismos, notable.

La plataforma .NET, si bien no elimina completamente todas las dificultades de este tipo de programación, sí que las reduce considerablemente, mediante la jerarquía ***System.Threading***, y las características de seguridad implementadas en ella. Para ello, la jerarquía se sirve de un concepto novedoso introducido por el entorno .NET: los **Dominios de Aplicación**.

Dominios de Aplicación

El concepto de **Dominio de Aplicación** (desde ahora, simplemente, **dominio**), no debe ser confundido con el de proceso en Win32. Una aplicación puede albergar un número indeterminado de Dominios de Aplicación, cada uno de los cuales esté completamente aislado de otros dominios o de otros procesos. Las aplicaciones que se ejecutan de diferentes dominios nunca pueden compartir recursos (ni siquiera variables de tipo global o campos estáticos), a no ser que utilicen el protocolo de comunicaciones suministrado por .NET *Remoting*.

El siguiente gráfico ilustra someramente la arquitectura de los dominios de aplicación: Una misma aplicación (Aplicación 1), puede definir dos (o más) dominios en los que pueden lanzarse subprocesos que compartan —eventualmente— ciertos datos. Si queremos aislar la ejecución entre subprocesos, bastará con situarlos en diferentes dominios para que sean imposibles las colisiones entre ambos.

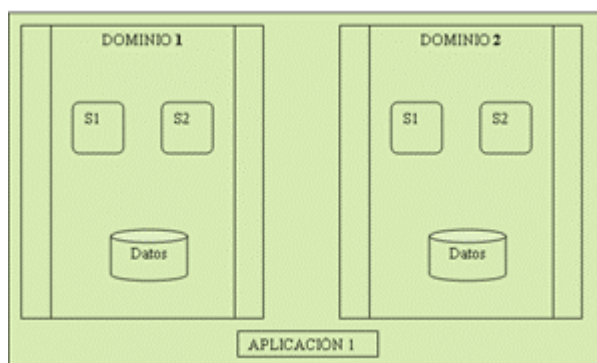


Figura 31. Esquema estructural de dominios y subprocesos

Además del cambio estructural que supone esta arquitectura en relación a la típica de Win32, los dominios de aplicación en .NET son también tipos administrados, con lo que vuelve a intervenir en su manejo el Garbage Collector y el resto de recursos administrados de .NET.

El funcionamiento del sistema de hebras se basa en la asociación de un subproceso a un método de un tipo en ejecución. El manejo se realiza mediante el objeto ***System.Thread***, pero una vez instanciado el objeto es necesario invocar al método *Start()* para lanzar el subproceso. Otra característica especial de funcionamiento, es que —una vez detenido un subproceso— no es posible reiniciarlo: se debe de crear un nuevo objeto ***Thread*** para volver a proceder con la ejecución.

El Código fuente 113 sigue ese mecanismo: asigna una hebra a un proceso, lanza la ejecución, y –tras un tiempo- la detiene-. Cuando se vuelve a intentar reiniciar el mismo proceso, falla. Sin embargo, si creamos un nuevo objeto **Thread**, vuelve a ponerse en funcionamiento.

```
// Autor: MSDN
// Modificado y traducido por el autor

using System;
using System.Threading;

public class Alpha
{
    // Este método se llama al iniciar el subprocesso
    public void Beta()
    {
        while (true)
        {
            Console.WriteLine("Alpha.Beta se está ejecutando en su propio
subproceso.");
        }
    }
}

public class Simple
{
    public static int Main()
    {
        Console.WriteLine("Ejemplo de uso de métodos Start/Stop/Join en un
subproceso");

        Alpha oAlpha = new Alpha();

        // Se crea el objeto subprocesso (oThread), pasando (un puntero a)
        // el método Alpha.Beta al delegado predeterminado (ThreadStart).
        // Este delegado, controla el evento Start() de cualquier objeto
Thread.

        // Con esto, el objeto queda inicializado, pero no lanzado.
        Thread oThread = new Thread(new ThreadStart(oAlpha.Beta));

        // Lanzamos la ejecución
        oThread.Start();

        // bucle de espera en lo que el subprocesso se activa
        while (!oThread.IsAlive);

        // La proceso principal de ejecución, siempre viene representada
        // por el objeto Thread predeterminado. Lo detenemos un momento
        // para permitir que nuestro subprocesso realice su labor
        Thread.Sleep(1);

        // El proceso principal dispone de métodos para averiguar información
        // sobre los dominios de aplicación y sus propiedades
        Console.WriteLine(Thread.GetDomain().FriendlyName);
        // Detenemos el subprocesso oThread
        oThread.Abort();

        // Esperar hasta que el subprocesso finalice.
        oThread.Join();

        Console.WriteLine();
        Console.WriteLine("Alpha.Beta ha finalizado");

        try
        {
```

```

        Console.WriteLine("Intento de reiniciar el subproceso
Alpha.Beta");
        oThread.Start();
    }
    catch (ThreadStateException)
    {
        Console.WriteLine("Excepción del tipo ThreadStateException
intentando reiniciar Alpha.Beta.");
        Console.WriteLine("A continuación, volvemos a crear un objeto
subproceso nuevo, para reiniciarlo");
        Console.Read();
    }

    // Como oThread ya existe, simplemente le volvemos a asignar un
    // nuevo objeto subproceso
    oThread = new Thread(new ThreadStart(oAlpha.Beta));

    // Lanzamos la ejecución
    oThread.Start();
    // bucle de espera en lo que el subproceso se activa
    while (!oThread.IsAlive);

    Thread.Sleep(100);
    // Detenemos el subproceso oThread
    oThread.Abort();

    return 0;
}
}

```

Salida:

Ejemplo de uso de métodos Start/Stop/Join en un subproceso

EjHebras.exe

Alpha.Beta se está ejecutando en su propio subproceso.
Alpha.Beta se está ejecutando en su propio subproceso.
Alpha.Beta se está ejecutando en su propio subproceso.
Alpha.Beta se está ejecutando en su propio subproceso.
Alpha.Beta se está ejecutando en su propio subproceso.
Alpha.Beta se está ejecutando en su propio subproceso.
Alpha.Beta se está ejecutando en su propio subproceso.

.....

Alpha.Beta ha finalizado

Intento de reiniciar el subproceso Alpha.Beta

Excepción del tipo ThreadStateException intentando reiniciar Alpha.Beta.

A continuación, volvemos a crear un objeto subproceso nuevo, para reiniciarlo

Alpha.Beta se está ejecutando en su propio subproceso.
Alpha.Beta se está ejecutando en su propio subproceso.
Alpha.Beta se está ejecutando en su propio subproceso.
Alpha.Beta se está ejecutando en su propio subproceso.
Alpha.Beta se está ejecutando en su propio subproceso.
Alpha.Beta se está ejecutando en su propio subproceso.
Alpha.Beta se está ejecutando en su propio subproceso.
Alpha.Beta se está ejecutando en su propio subproceso.
Finalizado el segundo subproceso

Código fuente 113

Por tanto, un subproceso, una vez concluido, no puede reutilizarse: se necesita una nueva instancia del objeto Thread. Si prestamos atención a la salida de la ventana deresultados en Visual Studio .NET podemos observar, que –efectivamente- se ha creado un dominio de aplicación predeterminado (DefaultDomain), un proceso principal, y dos subprocesos secuenciales, tal y como muestra la Figura 32.

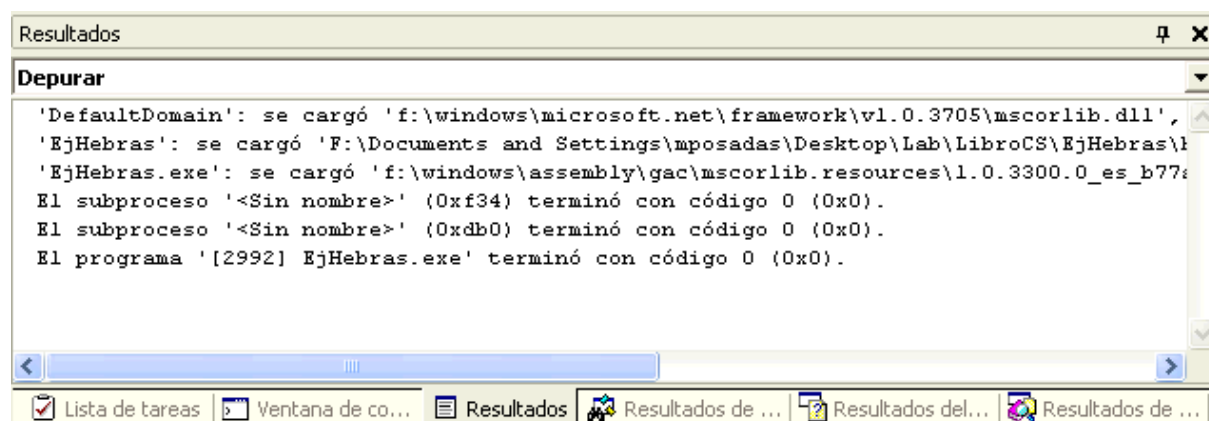


Figura 32. Ventana de Resultados de ejecución del ejemplo anterior

También hay que observar cómo el proceso principal, representado por el objeto *Thread* dispone de información sobre el dominio de aplicación activo (EjHebras.exe). El método *Join()* del proceso principal es un mecanismo que permite garantizar que se espera la terminación de la ejecución de un subproceso.

Sincronización de subprocesos

Uno de los problemas típicos de estos escenarios de trabajo, es la sincronización de subprocesos. En aquellos casos en los que dos subprocesos realizan una tarea que debe ser monitorizada por un tercero, se necesita un mecanismo para poder detener (o *congelar*) la ejecución, en espera de otro.

Esto puede conseguirse en .NET mediante el objeto **Monitor**, perteneciente a la clase *System.Threading*, que proporciona un mecanismo que sincroniza el acceso a los objetos. Dispone de métodos *Wait()* (esperar) y *Pulse()* y *PulseAll()*, que notifica a un proceso en espera (o a todos) que el estado de un objeto ha cambiado, con lo que éste puede determinar si se requiere su reanudación. También contiene otro par de métodos (*Enter()* y *Exit()*) para adquirir y bloquear –respectivamente– bloqueos exclusivos sobre objetos. Normalmente, utilizan la instrucción **lock** de C#, ya comentada en el capítulo anterior, para bloquear el subproceso.

El siguiente ejemplo del MSDN (Código fuente 114) ilustra el uso de **lock**, **Monitor** y **Thread** para la sincronización de subprocesos.

```
using System;
using System.Threading;
using System.Collections;

namespace MonitorCS1
{
    class MonitorSample
    {
        const int MAX_LOOP_TIME = 200;
        Queue m_smplQueue;

        public MonitorSample()
        {
            m_smplQueue = new Queue();
        }
        public void FirstThread()
        {

```

```

        int counter = 0;
        lock(m_smplQueue)
        {
            while(counter < MAX_LOOP_TIME)
            {
                //esperar, si la cola esta ocupada
                Monitor.Wait(m_smplQueue);
                // Añade un elemento a la cola
                m_smplQueue.Enqueue(counter);
                // Libera el subproceso en espera
                Monitor.Pulse(m_smplQueue);
                counter++;
            }
        }
    }
    public void SecondThread()
    {
        lock(m_smplQueue)
        {
            // Libera el subproceso en espera
            Monitor.Pulse(m_smplQueue);
            //Espera en el bucle mientras se desocupa la cola
            //Se sale con un timeout cuando se ha detenido el
            // primer subproceso
            while(Monitor.Wait(m_smplQueue,10))
            {
                // Obtiene el primer elemento.
                int counter = (int)m_smplQueue.Dequeue();
                // Imprime el elemento
                Console.WriteLine(counter.ToString());
                // Libera el subproceso en espera
                Monitor.Pulse(m_smplQueue);
            }
        }
    }
    //Devuelve el numero de elementos de la cola
    public int GetQueueCount()
    {
        return m_smplQueue.Count;
    }

    static void Main(string[] args)
    {
        //Crea el objeto MonitorSample
        MonitorSample test = new MonitorSample();
        //Crea el primer subproceso
        Thread tFirst = new Thread(new ThreadStart(test.FirstThread));
        //Crea el segundo subproceso
        Thread tSecond = new Thread(new ThreadStart(test.SecondThread));
        //Lanza los dos subprocesos
        tFirst.Start();
        tSecond.Start();
        //espera la terminación de ambos
        tFirst.Join();
        tSecond.Join();
        //Imprime el número de elementos de la cola
        Console.WriteLine("Queue Count = " +
test.GetQueueCount().ToString());
        Console.Read();
    }
}

```

Código fuente 114

En general se recomienda que la utilización de subprocesos se realice si se dan una de las siguientes circunstancias:

- Si es necesario que una tarea tenga una prioridad en particular.
- Si tiene una tarea que podría ejecutarse durante bastante tiempo (y, por tanto, bloquear otras tareas).
- Si debe ubicar subprocesos en un apartamento de un único subproceso (todos los subprocesos ThreadPool se encuentran en un apartamento multiproceso).
- Si necesita asociar una identidad estable al subproceso. Por ejemplo, debería utilizar un subproceso dedicado para cancelar dicho subproceso, suspenderlo o descubrirlo por nombre.

El objeto *ThreadPool* (que podríamos traducir por Caché de Subprocesos), proporciona la forma más fácil de controlar varios subprocesos para tareas relativamente cortas que no bloquearán otros subprocesos y cuando no se espera ninguna programación particular de las tareas.

Código no administrado (*unsafe*)

Una de las características más interesantes de C# es que el nivel de compatibilidad con otros lenguajes no se queda simplemente en cuestiones semánticas y sintácticas, sino que permite –incluso– la inserción de código de C/C++ dentro de una estructura de control, llamada *unsafe*. Esta situación podría producirse en diferentes circunstancias, como el que queramos aprovechar rutinas bien probadas codificadas en C++, o que tengamos que pasar argumentos a funciones escritas en cualquier lenguaje no administrado por el CLR, que reciba punteros como argumentos (incluyendo ciertas funciones de las API's del sistema operativo).

Si estamos trabajando con el compilador en línea de comandos, esto se traduce en una directiva de compilación que indica esta situación al compilador:

```
Csc ProgramaConCodigoInseguro.cs /unsafe
```

Código fuente 115

En cambio, si nos encontramos en el entorno de desarrollo de Visual Studio .NET tendremos que indicar esa circunstancia en la página de configuración de nuestra aplicación, dentro del apartado **Permitir código no seguro**, ya que el IDE no lo detecta automáticamente por nosotros, aunque el compilador muestra un mensaje de error en cuanto aparece un bloque marcado como **unsafe**.

```
El código unsafe sólo puede aparecer si se compila con /unsafe
```

Código fuente 116

Sintácticamente, su definición no puede ser más simple.

```
unsafe {  
..... <Bloque de código no administrado>  
}
```

```

o también, podemos preceder la definición de un miembro de la clase
con la palabra unsafe:

class DefiniciónInsegura {
    public unsafe int *x; //Hay que definirlos en líneas distintas
    public unsafe int *y;
}

igualmente, podemos definir la clase o estructura completa como unsafe:

class unsafe DefiniciónInsegura {
    public int *x; //Hay que definirlos en líneas distintas
    public int *y;
}

```

Código fuente 117

En general, y como recomendación global sobre el uso de punteros con definiciones **unsafe**, solo deben definirse punteros cuyos objetos se almacenen completamente en la pila, ya que un puntero a un objeto situado en el montón, no será administrado por el **Garbage Collector**. Para resumir, sólo se consideran válidos aquellos punteros a tipos por valor básicos, enumeraciones y estructuras que no guarden en sus miembros referencias a objetos almacenados en el montón.

El Código fuente 118 es un ejemplo muy simple de la utilización de código inseguro. Naturalmente, se podría incidir muchísimo mas en el tema, (de hecho, se podrían repasar todos los fundamentos de C++ que no incumplen la normativa indicada más arriba, pero –con las directivas indicadas- el lector tendrá una introducción suficiente al tema).

```

using System;

namespace Inseguro
{
    class Class1
    {
        [STAThread]
        static void Main(string[] args)
        {
            unsafe
            {
                int x = 10;
                int *px = &x; //Se asigna al puntero *px la dirección
                             // de memoria de x
                // Ahora se puede leer el valor de x a partir de *px, que
                // apunta a su dirección de memoria
                Console.WriteLine("Valor de x: {0}", *px);
            }
        }
    }
}

```

Salida:

Valor de x: 10

Código fuente 118

Vamos a ver a continuación otro problema relacionado con el manejo del código anterior a .NET, y concretamente el –muy importante- problema de la comunicación con componentes COM (desde COM a .NET y viceversa).

Manejo de elementos COM desde C#

Aunque quizá sería más lógico hablar de cómo Visual Studio es capaz de manejar los componentes COM (por razones que veremos a continuación), al menos en los primeros estadios de la implantación de .NET, el escenario de comunicación en ambos sentidos se va a producir con bastante frecuencia, por lo que es vital comentar detalladamente su funcionamiento aquí.

COM Interop: la librería de compatibilidad con ActiveX de .NET

Uno de los problemas que se plantean ante la aparición de una plataforma de desarrollo como .NET es la necesidad de la migración de aplicaciones. Y dentro de ese panorama la migración puede abordarse –si es que se decide realizar tal migración– adoptando dos estrategias posibles: migración horizontal y migración vertical.

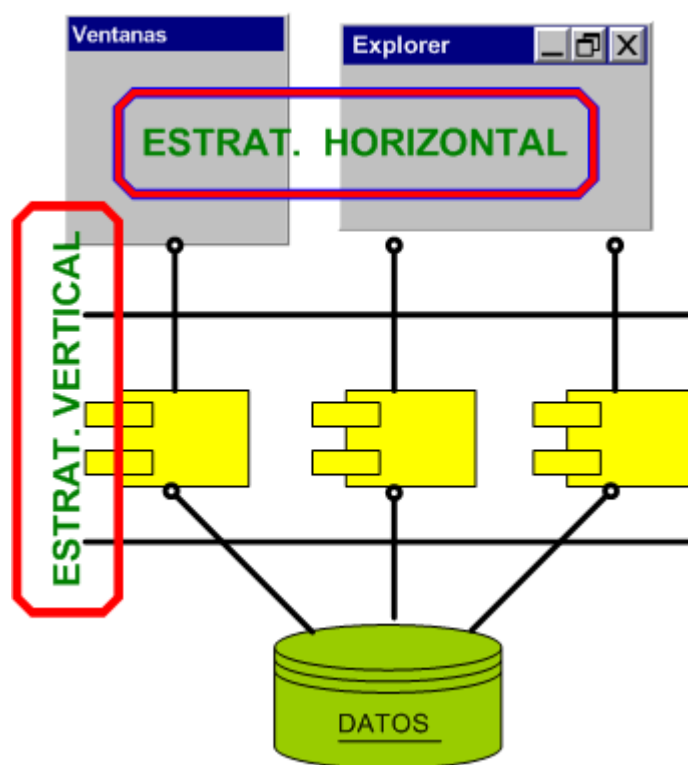


Figura 33. Estrategias de migración de aplicaciones en .NET

Migración Horizontal

Es el supuesto en el que, de acuerdo con un esquema aplicativo de 3 capas, se decide por migrar una de las capas exclusivamente, dejando la otra intacta. Esto aboca inmediatamente a una de estas dos situaciones:

- O tenemos que hacer que la Interfaz de Usuario quede en .NET y se comunique con componentes COM, a través de algún mecanismo de comunicación apropiado,
- o migramos la capa de componentes y tenemos que situar algo que haga de puente entre nuestros componentes y la Interfaz clásica, para que ambas puedan entenderse.

La solución propuesta por .NET es una librería de compatibilidad, llamada **COM Interop**, que permite la generación dinámica de componentes .NET que pueden hacer la labor de "proxy" entre estos dos mundos, que en principio, debido a su naturaleza, son incommunicables.

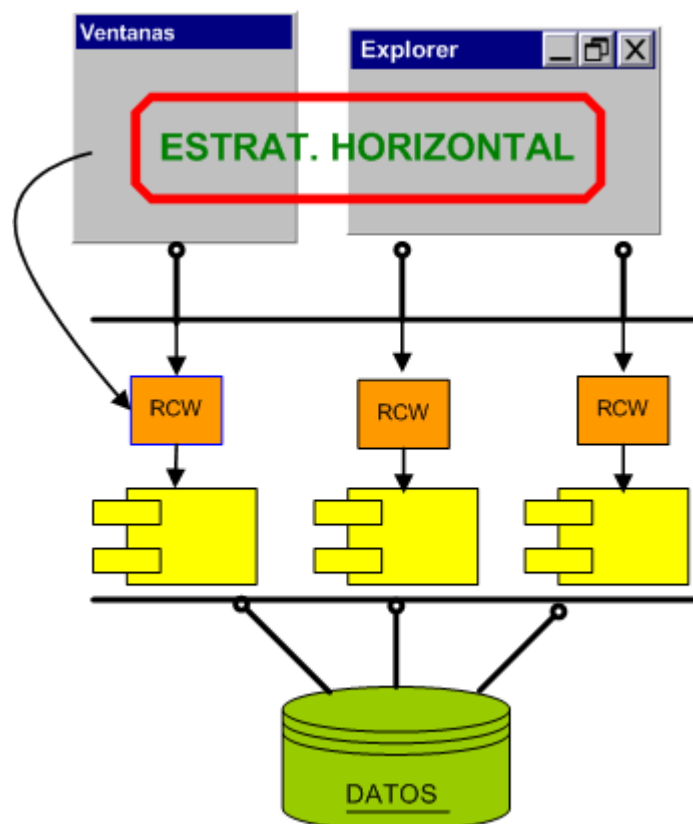


Figura 34. Esquema de la arquitectura de migración horizontal en la capa intermedia

El gráfico adjunto ilustra esta situación en cada caso. A continuación, la migración de la capa intermedia:

Si la situación es esta, lo más conveniente sería la creación de componentes del tipo **CCW** (*COM Callable Wrapper*) o *Proxies de interfaz con COM*, que son generados por Visual Studio de forma automática, o pueden ser generados manualmente, mediante las herramientas de producción adicionales que se incluyen en el SDK de la plataforma.

Dichos componentes, **implementan** exactamente las mismas llamadas que esperaríamos encontrar en los componentes COM correspondientes, pero las resuelven internamente efectuando una serie de conversiones de datos que permiten la **traducción** de un lenguaje a otro.

Si el caso es el de una migración de capa de Interfaz de usuario, la situación es algo distinta, en cuanto al sentido de las llamadas (de .NET a COM)

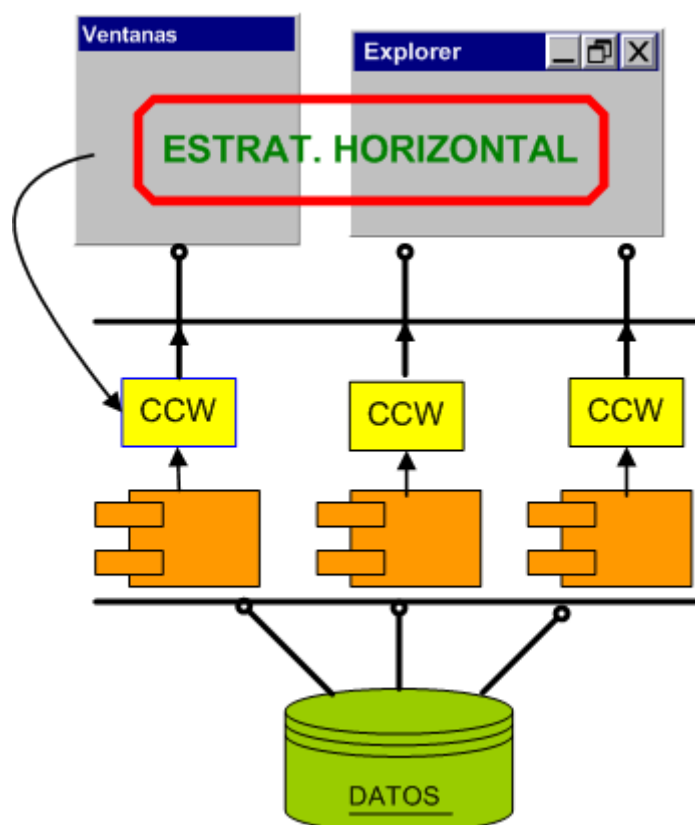


Figura 35. Esquema de la arquitectura de migración horizontal de capa de Interfaz de Usuario

Si lo que preferimos es evitar la utilización de elementos de compatibilidad, lo preferible es seleccionar una capa suficientemente aislada de la aplicación, como si fuera otra aplicación distinta, agregada a la anterior, y realizar la migración (podría ser una página de búsqueda de información en nuestro sitio Web, por ejemplo, aunque esta situación será bastante más difícil de encontrar).

Vamos a poner un pequeño ejemplo de utilización de esta técnica dentro de Visual Studio, y aplicado a la situación más natural: aquella en la que una nueva aplicación hecha en .NET quiere aprovechar las características de un componente ya hecho y registrado, sin crear uno nuevo (por ejemplo, podría ser un componente de capa de negocio que devolviera un conjunto de datos estándar sobre la empresa y que se utilizara en más de una aplicación).

El código fuente del componente COM en este ejemplo, sería algo muy simple. Sencillamente, devolvería una colección de datos relativos a la empresa

```
//Componente ActiveX construido en Visual Basic 6.0
// Autor: Marino Posadas
```

```
Public Function DatosEmpresa() As Collection
```

```
    Dim DE As New Collection
    DE.Add "Grupo EIDOS", "Empresa"
    DE.Add "C/ Téllez 30, Of. 2", "Dirección"
    DE.Add "Madrid", "Población"
    DE.Add "Madrid", "Provincia"
    DE.Add "28007", "CPostal"
    DE.Add "www.eidos.es", "Web"
```

```
Set DatosEmpresa = DE
End Function
```

Código fuente 119

En tal situación, bastaría con buscar el componente en el apartado de referencias del Explorador de Soluciones de Visual Studio, y seleccionarlo directamente desde su ubicación actual:

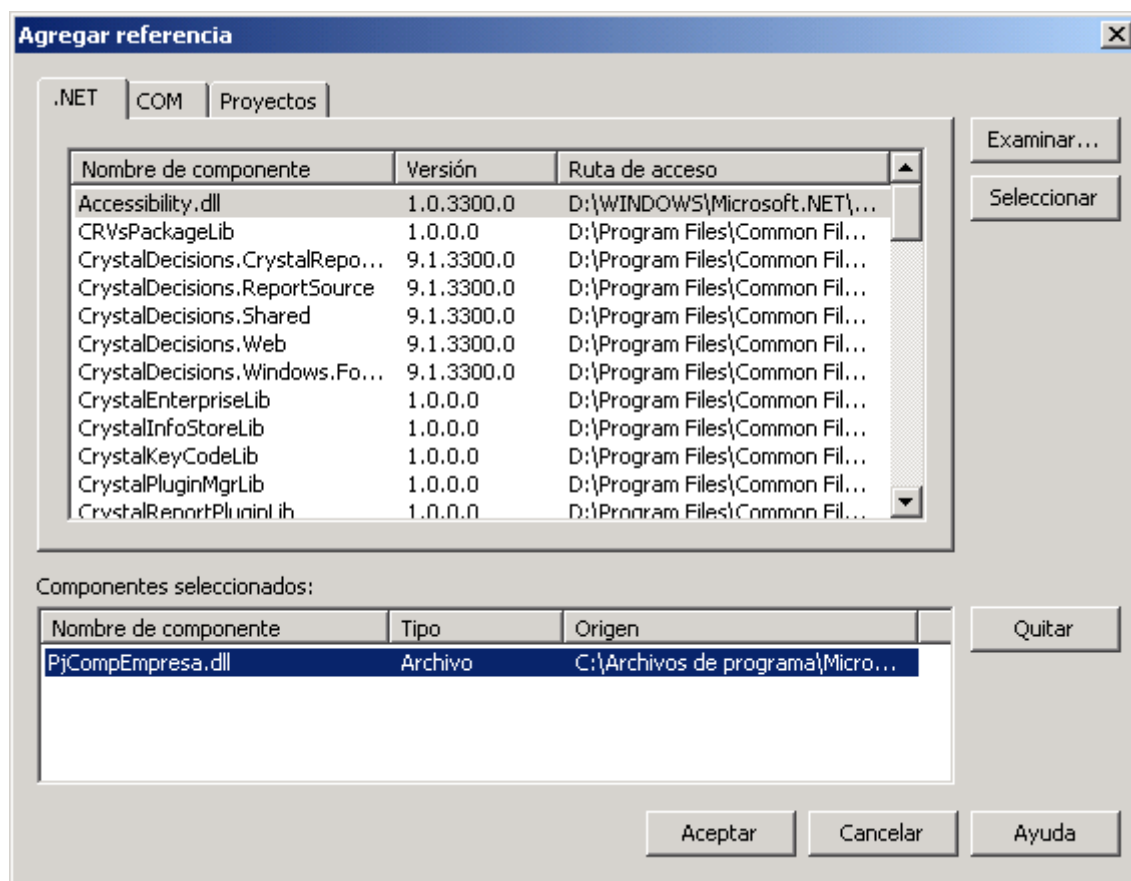


Figura 36. Selección del componente COM desde el entorno de Visual Studio .NET

Al aceptar la selección, Visual Studio, silenciosamente, realiza la creación de un componente de Interoperabilidad, al que da el mismo nombre programático que nuestro componente COM, pero, que tras una observación más atenta, resulta llamarse físicamente: **Interop.ComponenteEmpresa.dll**, y que una vez abierto en el Examinador de Objetos, nos muestra su construcción interna:

Aquí pueden verse también algunas de las librerías adicionales que Visual Studio ha referenciado para posibilitar el funcionamiento del *proxy*: **Interop.VBA** que da el *soporte de traducción* a dicho *proxy*.

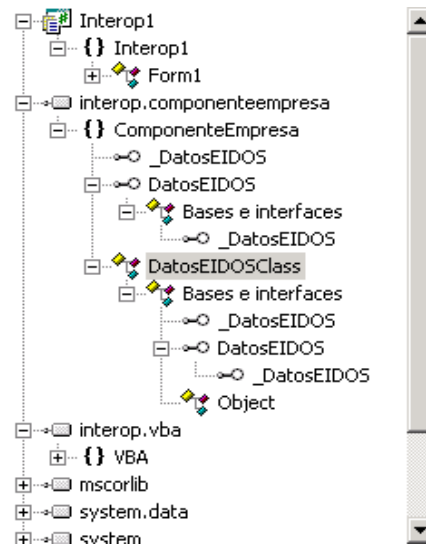


Figura 37. El componente de COM Interop, *DatosEmpresa* desde el Examinador de Objetos

Ahora, no hay más que crear un objeto de instancia de ese componente, y llamarlo, como si fuera cualquier otro componente en .NET (utilizamos una aplicación Windows, para sacar los datos en cajas de mensajes).

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
using ComponenteEmpresa;

namespace Interop1
{
    public class Form1 : System.Windows.Forms.Form
    {
        private System.Windows.Forms.Button button1;
        private System.ComponentModel.Container components = null;
        public Form1()
        {
            InitializeComponent();
        }
        protected override void Dispose( bool disposing )
        {
            if( disposing )
            {
                if (components != null)
                {
                    components.Dispose();
                }
            }
            base.Dispose( disposing );
        }

        #region Windows Form Designer generated code
        /// <summary>
        /// Required method for Designer support - do not modify
        /// the contents of this method with the code editor.
        /// </summary>
        private void InitializeComponent()
        {

```

```

        this.button1 = new System.Windows.Forms.Button();
        this.SuspendLayout();
        //
        // button1
        //
        this.button1.Font = new System.Drawing.Font("Verdana", 12F,
System.Drawing.FontStyle.Bold, System.Drawing.GraphicsUnit.Point,
((System.Byte)(0)));
        this.button1.Location = new System.Drawing.Point(32, 32);
        this.button1.Name = "button1";
        this.button1.Size = new System.Drawing.Size(208, 48);
        this.button1.TabIndex = 0;
        this.button1.Text = "Llamar al Componente Horario";
        this.button1.Click += new
System.EventHandler(this.button1_Click);
        //
        // Form1
        //
        this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
        this.ClientSize = new System.Drawing.Size(280, 117);
        this.Controls.AddRange(new System.Windows.Forms.Control[] {

                                this.button1});

        this.Name = "Form1";
        this.Text = "Form1";
        this.ResumeLayout(false);

    }
#endregion

[STAThread]
static void Main()
{
    Application.Run(new Form1());
}

private void button1_Click(object sender, System.EventArgs e)
{
    ComponenteEmpresa.DatosEIDOSClass D = new DatosEIDOSClass();
    VBA.Collection c = D.DatosEmpresa();
    foreach (object o in c) MessageBox.Show(o.ToString());
}
}

```

Código fuente 120

Lo que nos iría mostrando en cada ciclo uno de los datos de la colección. Es importante reseñar que la variable a la que asignamos el resultado de la llamada a *DatosEmpresa()*, es del tipo **VBA.Collection**, y por lo tanto, otro elemento de compatibilidad. La salida del programa nos irá mostrando los datos de la colección.



Figura 38. Salida de la primera iteración del bucle del programa anterior

Pensamos, por tanto, que la solución de **COM Interop** es interesante como solución transitoria, y como ayuda en la migración de aplicaciones o bien en la adaptación de escenarios al nuevo entorno, así como en aquellos casos en los que necesitamos un control de terceros para realizar una tarea muy compleja de programar, y de la que no disponemos de una versión .NET actualizada. En general, sin embargo, parece preferible utilizar en lo posible todos los recursos nativos de la plataforma.

Metadatos y Documentación XML

Otra característica novedosa del lenguaje, es su capacidad de generar documentación en formato XML a partir de información embebida en el código fuente en forma de comentarios, y también por su acceso a los atributos de usuario, como hemos visto en el capítulo anterior.

Mediante una opción de configuración, podemos indicar al entorno de desarrollo que genere la documentación en fichero cuyo nombre y ubicación es establecido por nosotros, pongamos, por ejemplo, el propio directorio \bin donde genera uno de los ejecutables. Basta con indicarlo, y –durante el proceso de construcción del ejecutable– se creará el archivo indicado. En la Figura 39 puede verse la configuración para un fichero de salida *Ayuda.XML* situado en la ubicación citada antes.

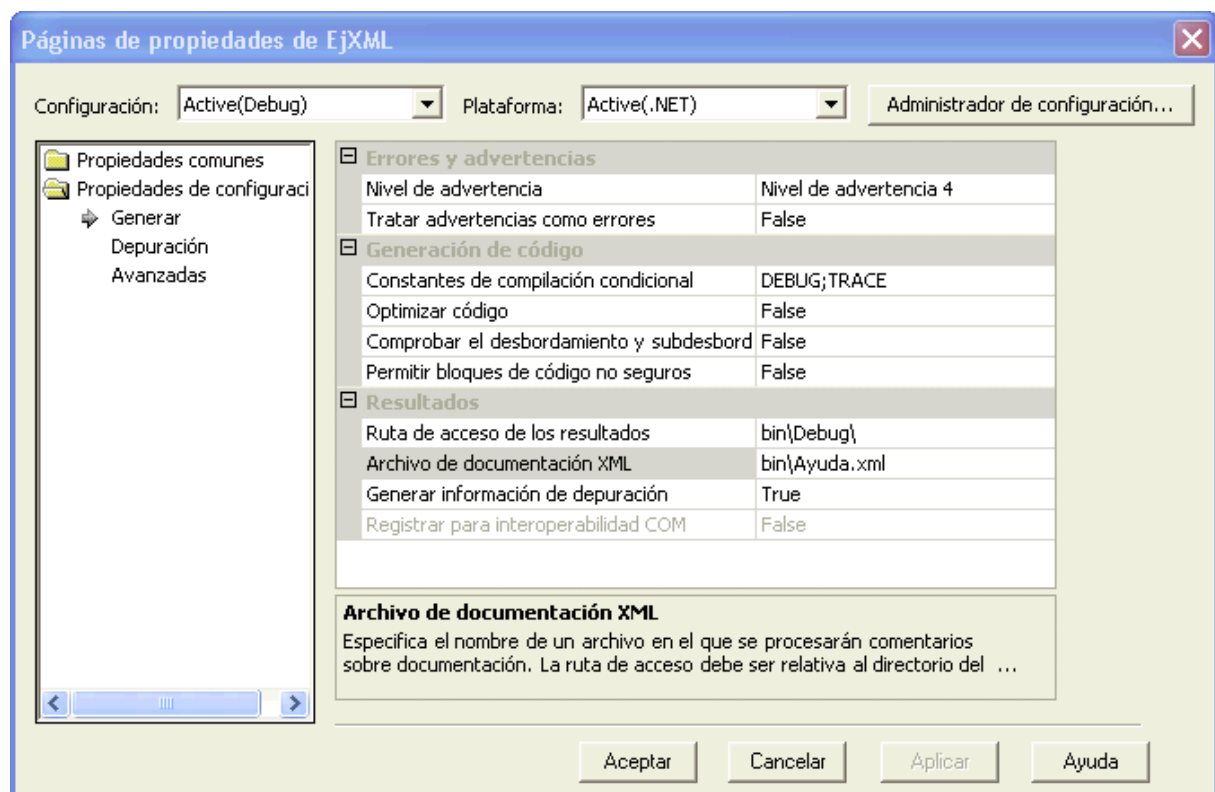


Figura 39. Configuración de la aplicación para la generación de un archivo de ayuda XML

El resultado, depende de los comentarios incluidos en el texto con una triple barra (///) que el compilador asume como elementos a situar en el fichero de salida. Si hacemos la prueba con una aplicación estándar de Windows, y mantenemos los comentarios predeterminados, obtendremos, en la versión española, un fichero de salida con el contenido que aparece en el Código fuente 121.

```
<?xml version="1.0"?>
```

```

<doc>
  <assembly>
    <name>EjXML</name>
  </assembly>
  <members>
    <member name="T:EjXML.Form1">
      <summary>
        Descripción breve de Form1.
      </summary>
    </member>
    <member name="F:EjXML.Form1.components">
      <summary>
        Variable del diseñador requerida.
      </summary>
    </member>
    <member name="M:EjXML.Form1.Dispose(System.Boolean)">
      <summary>
        Limpiar los recursos que se estén utilizando.
      </summary>
    </member>
    <member name="M:EjXML.Form1.InitializeComponent">
      <summary>
        Método necesario para admitir el Diseñador, no se puede modificar
        el contenido del método con el editor de código.
      </summary>
    </member>
    <member name="M:EjXML.Form1.Main">
      <summary>
        Punto de entrada principal de la aplicación.
      </summary>
    </member>
  </members>
</doc>

```

Código fuente 121

El fichero resultante, como puede verse, toma los comentarios XML y los convierte en elementos del fichero de salida XML. En realidad, el compilador de C# admite una serie de palabras reservadas, asociadas a la documentación XML, dotándoles de un significado dinámico, según la Tabla 20.

Etiquetas permitidas en la confección de documentación XML		
<c>	<para>	<see>
<code>	<param>	<seealso>
<example>	<paramref>	<summary>
<exception>	<permission>	<value>
<include>	<remarks>	
<list>	<returns>	

Tabla 20

Donde cada etiqueta adopta un significado, tal como:

`<c>` es para marcar texto como código fuente

`<code>` igual que la anterior, pero permite marcar varias líneas de código

`<para>` se utiliza dentro de otra etiqueta, tal como `<remarks>` o `<returns>`, y permite dar una estructura al texto, etc. (el lector dispone de toda la documentación de cada comando en la propia ayuda de Visual Studio, bajo el epígrafe “Documentación XML”. Hay que citar, no obstante, un elemento que tiene un significado especial, como es **cref**, que se utiliza como atributo de un elemento y tiene la siguiente sintaxis:

`<see cref="miembro"/>` ,

donde `<see>`, permite especificar un vínculo desde dentro del texto, y *miembro*, referencia a un miembro o campo al cual se puede llamar desde el entorno de compilación actual. El compilador comprueba si el elemento de código dado existe y pasa *miembro* al nombre de elemento en el resultado XML; *miembro* debe aparecer entre comillas dobles (" ").

Al objeto de tener una idea más aproximada de cómo funciona el conjunto de etiquetas citado, incluimos a continuación un ejemplo *oficial*, y el resultado de su generación XML:

```
// XMLsample.cs (Autor Microsoft MSDN)
// compile with: /doc:XMLsample.xml
using System;

/// <summary>
/// Documentación resumida del nivel de clase.</summary>
/// <remarks>
/// Los comentarios más extensos se pueden asociar a un tipo o miembro
/// mediante la etiqueta de comentarios</remarks>
public class SomeClass
{
    /// <summary>
    /// Almacén para la propiedad del nombre</summary>
    private string myName = null;

    /// <summary>
    /// Constructor de la clase. </summary>
    public SomeClass()
    {
        // TODO: Agregar aquí la lógica del constructor
    }

    /// <summary>
    /// Propiedad Name </summary>
    /// <value>
    /// Se utiliza una etiqueta de valor para describir el valor de la
    propiedad</value>
    public string Name
    {
        get
        {
            if ( myName == null )
            {
                throw new Exception("Name tiene el valor null");
            }

            return myName;
        }
    }

    /// <summary>
```

```

    /// Descripción de SomeMethod.</summary>
    /// <param name="s"> Descripción del parámetro s</param>
    /// <seealso cref="String">
    /// Se puede utilizar el atributo cref en las etiquetas para referirse a un
tipo o miembro
    /// y el compilador comprobará si la referencia existe. </seealso>
    public void SomeMethod(string s)
    {
    }

    /// <summary>
    /// Otro método. </summary>
    /// <returns>
    /// La etiqueta de resultados describe los resultados obtenidos.</returns>
    /// <seealso cref="SomeMethod(string)">
    /// El atributo cref se utiliza para hacer referencia a un método específico
</seealso>
    public int SomeOtherMethod()
    {
        return 0;
    }

    /// <summary>
    /// Punto de entrada de la aplicación.
    /// </summary>
    /// <param name="args"> Lista de los argumentos de la línea de
comandos</param>
    public static int Main(String[] args)
    {
        // TODO: Agregar aquí el código para iniciar la aplicación

        return 0;
    }
}

```

Código fuente 122

y el fichero XML de salida del fuente anterior, sería el que muestra el Código fuente 123.

```

<?xml version="1.0"?>
<doc>
  <assembly>
    <name>EjXML</name>
  </assembly>
  <members>
    <member name="T:SomeClass">
      <summary>
        Documentación resumida del nivel de clase.</summary>
      <remarks>
        Los comentarios más extensos se pueden asociar a un tipo o miembro
        mediante la etiqueta de comentarios</remarks>
    </member>
    <member name="F:SomeClass.myName">
      <summary>
        Almacén para la propiedad del nombre</summary>
    </member>
    <member name="M:SomeClass.#ctor">
      <summary>
        Constructor de la clase. </summary>
    </member>
    <member name="M:SomeClass.SomeMethod(System.String)">
      <summary>
        Descripción de SomeMethod.</summary>
    </member>
  </members>
</doc>

```

```

        <param name="s"> Descripción del parámetro s</param>
        <seealso cref="T:System.String">
        Se puede utilizar el atributo cref en las etiquetas para referirse a un
        tipo o miembro
        y el compilador comprobará si la referencia existe. </seealso>
    </member>
    <member name="M:SomeClass.SomeOtherMethod">
        <summary>
        Otro método. </summary>
        <returns>
        La etiqueta de resultados describe los resultados obtenidos.</returns>
        <seealso cref="M:SomeClass.SomeMethod(System.String)">
        El atributo cref se utiliza para hacer referencia a un método
        específico </seealso>
    </member>
    <member name="M:SomeClass.Main(System.String[]) ">
        <summary>
        Punto de entrada de la aplicación.
        </summary>
        <param name="args"> Lista de los argumentos de la línea de
        comandos</param>
    </member>
    <member name="P:SomeClass.Name">
        <summary>
        Propiedad Name </summary>
        <value>
        Se utiliza una etiqueta de valor para describir el valor de la
        propiedad</value>
    </member>
</members>
</doc>

```

Código fuente 123

La documentación XML puede enlazarse posteriormente a un fichero de presentación escrito en uno de los dos lenguajes de hojas de estilo: Hojas de Estilo en Cascada (CSS) o Hojas de Estilo Extendidas (XSL/T), para suministrar un modelo visual del fichero XML, convirtiéndolo, eventualmente, en HTML.

Generación y manipulación de módulos de ensamblado (*Assemblies*)

Vamos a terminar esta parte dedicada al lenguaje comentando los aspectos principales de la generación módulos de ensamblado. (Al final de esta obra dedicaremos un apartado al proceso de distribución de las aplicaciones, que incluye la posibilidad de crear aplicaciones específicas de instalación.) Como ya hemos comentado, la unidad principal de creación y distribución implementación, control de versiones, reutilización, ámbitos de activación y permisos de seguridad en los programas hechos para .NET Framework es el Ensamblado (*Assembly*).

Un ensamblado es una colección de tipos y recursos creados para operar en conjunto y formar una **unidad lógica de funcionalidad**. Los ensamblados proporcionan al CLR la información necesaria para conocer las implementaciones de tipos, ya que “para el motor de tiempo de ejecución, un tipo no existe si no es en el contexto de un ensamblado”¹⁴.

¹⁴ Según la documentación oficial.

Acudiendo una vez más a la fuente original, el MSDN, se nos recuerda que un ensamblado realiza las funciones siguientes:

- Contiene el código que ejecuta el **Common Language Runtime**. El código del lenguaje intermedio de Microsoft (**MSIL**) de un archivo ejecutable portable (**PE**)¹⁵ no se ejecuta si no tiene asociado un manifiesto de ensamblado. Cada ensamblado sólo puede tener un punto de entrada (es decir, **DllMain** (librerías), **WinMain** (ejecutable Windows) o **Main** (aplicación de consola).
- Crea un **límite de seguridad**. Un ensamblado es la unidad en la que se solicitan y conceden los permisos.
- Crea un **límite de tipos**. La identidad de cada tipo incluye el nombre del ensamblado en que reside. Por ello, un tipo **T** cargado en el ámbito de un ensamblado no es igual que un tipo denominado **T** cargado en el ámbito de otro ensamblado.
- Crea un **límite de ámbito de referencia**. El manifiesto del ensamblado contiene los metadatos del ensamblado que se utilizan para resolver tipos y satisfacer las solicitudes de recursos. Especifica los tipos y recursos que se exponen fuera del ensamblado. El manifiesto también enumera otros ensamblados de los que depende.
- Forma un **límite de versión**. El ensamblado es la unidad de versión más pequeña del **Common Language Runtime**; todos los tipos y recursos del mismo ensamblado pertenecen a la misma versión. El manifiesto del ensamblado describe las dependencias de versión que se especifiquen para los ensamblados dependientes. (o dicho de otra forma, no pueden convivir dos versiones en el mismo ensamblado, ya que la versión se registra en los Metadatos del mismo, y está indisolublemente unida a él)¹⁶.
- Crea una **unidad de implementación**. Cuando se inicia una aplicación, sólo deben estar presentes los ensamblados a los que llama la aplicación inicialmente. Los demás ensamblados, como los recursos de localización o los ensamblados que contengan clases de utilidad, se pueden recuperar a petición. De este modo, se puede mantener la simplicidad y transparencia de las aplicaciones la primera vez que se descargan.
- Es la unidad que permite la **ejecución simultánea**¹⁷.

¹⁵ El formato PE es el formato estándar de un ejecutable para los sistemas operativos Windows. Los metadatos se almacenan en una sección del archivo ejecutable portable (PE) .NET mientras que el lenguaje intermedio de Microsoft (MSIL), en otra. Para más información sobre los archivos PE y los Metadatos, ver "Metadatos y la estructura del archivo PE", en el MSDN de Microsoft.

¹⁶ Si que se puede, sin embargo, utilizar ensamblados distintos (de versiones distintas) en la misma solución.

¹⁷ Posibilidad de ejecutar varias versiones del mismo ensamblado simultáneamente. Puede estar en el mismo equipo, o en el mismo proceso o dominio de aplicación. Permitir que los ensamblados se ejecuten simultáneamente es esencial para tener un control de versiones sólido en **Common Language Runtime** (fuente: Glosario de términos del MSDN)

Calificación operativa de los ensamblados

Podemos dividir los ensamblados en varias categorías, en función del criterio que se siga para ello: por ejemplo, desde el punto de vista de la ejecución y almacenamiento, podemos dividirlos en **estáticos** o **dinámicos**. Desde el punto de vista del acceso, en **públicos** o **privados**.

Ensamblados públicos y privados

Los ensamblados estáticos pueden incluir tipos de .NET Framework (interfaces y clases), así como recursos para el ensamblado (mapas de bits, archivos JPEG, archivos de recursos, etc.). Los ensamblados estáticos se almacenan en el disco, en archivos PE. Lógicamente, casi todos los recursos que hemos utilizado hasta ahora, se traducen, después de la compilación, en ensamblados estáticos.

También se puede utilizar .NET Framework para crear ensamblados dinámicos, que se ejecutan directamente desde la memoria y no se guardan en el disco antes de su ejecución. Los ensamblados dinámicos se pueden guardar en el disco una vez que se hayan ejecutado. Se crean mediante la librería de clases *Reflection.Emit*.

La idea central de los ensamblados dinámicos es que podemos crear tipos en tiempo de ejecución dependiendo de las necesidades de programación. Esta característica, aparte de que contribuye notablemente a la flexibilidad de la aplicación, hace de los lenguajes .NET candidatos ideales para el tratamiento de información mediante técnicas avanzadas relacionadas con la Inteligencia Artificial, los Algoritmos Genéticos o las Redes Neuronales, en las que, en ocasiones, se crean tipos dinámicamente en función de los datos que se obtienen como resultado de otros procesos de cálculo.

La Tabla 21, muestra el conjunto de abstracciones que pueden crearse con esta librería, junto a sus significados principales:

Abstracción	Descripción
AssemblyBuilder	Clase que define y representa un ensamblado dinámico. Se hereda de la clase Assembly.
ConstructorBuilder	Clase que define y representa un constructor, que describe una operación que crea una instancia de un tipo. Se hereda de la clase ConstructorInfo.
CustomAttributeBuilder	Clase que ayuda a definir atributos personalizados.
EnumBuilder	Clase que ayuda a definir y representar un tipo de enumeración. Se hereda de la clase Type.
EventBuilder	Clase que se utiliza para definir y representar un evento. Se hereda de la clase EventInfo. Los eventos de un tipo de objeto especifican las transiciones de estado con nombre en las que los suscriptores pueden registrar o anular el registro de interés mediante métodos de descriptor de acceso.
FieldBuilder	Clase que se utiliza para definir y representar un campo. Se hereda de la clase FieldInfo. Un campo es una subdivisión con nombre de un valor. Un tipo puede tener miembros de campo. Un módulo o un tipo también puede hacer referencia a los campos definidos en la sección .sdata de un archivo ejecutable portable (PE).
ILGenerator	Clase que se utiliza para definir y representar MSIL. El compilador Just-In-Time (JIT) del motor de tiempo de ejecución convierte las instrucciones de lenguaje intermedio de Microsoft (MSIL) en código nativo.

Label	Clase que constituye una representación opaca de una etiqueta que utiliza la clase ILGenerator. Las etiquetas se corresponden con ubicaciones específicas del código MSIL.
LocalBuilder	Clase que representa una variable local declarada en el cuerpo de un método.
MethodBuilder	Clase que se utiliza para definir y representar un método. Se hereda de la clase MethodInfo. Un método describe una operación que se puede realizar con los valores de un tipo.
ModuleBuilder	Clase que se utiliza para definir y representar un módulo. Se hereda de la clase Module. Un módulo es una unidad de compilación o una unidad de desarrollo. Un módulo creado mediante la API de Reflection Emit se denomina módulo dinámico mientras dure el proceso en el que se creó el módulo. El término módulo dinámico no es aplicable si el módulo se guarda en el disco y después se carga desde éste. Los módulos dinámicos que se pueden guardar en el disco como archivos PE se denominan módulos dinámicos con persistencia. Los módulos dinámicos que sólo existen en la memoria se denominan módulos dinámicos transitorios.
ParameterBuilder	Clase que se utiliza para definir y representar un parámetro. Se hereda de la clase ParameterInfo. Los parámetros son ubicaciones que representan argumentos de métodos o constructores.
PropertyBuilder	Clase que se utiliza para definir y representar una propiedad. Se hereda de la clase PropertyInfo. Las propiedades de un tipo de objeto especifican valores con nombre a los que se obtiene acceso a través de métodos de descriptor de acceso que leen y escriben el valor.
TypeBuilder	Clase que se utiliza para definir y representar un tipo. Se hereda de la clase Type. Los tipos describen valores. El tipo define los valores permitidos y las operaciones permitidas que son compatibles con los valores del tipo. Los tipos pueden tener miembros que sean tipos, métodos, campos, propiedades o eventos.

Tabla 21

Estructura de un ensamblado

Estructuralmente, los ejecutables y DLL's estáticas se dividen principalmente en 3 partes: una definición que indica que el ensamblado es para ser interpretado por el CLR, la información de Metadatos y el código ejecutable propiamente dicho. Su estructura es similar a la de la Figura 40

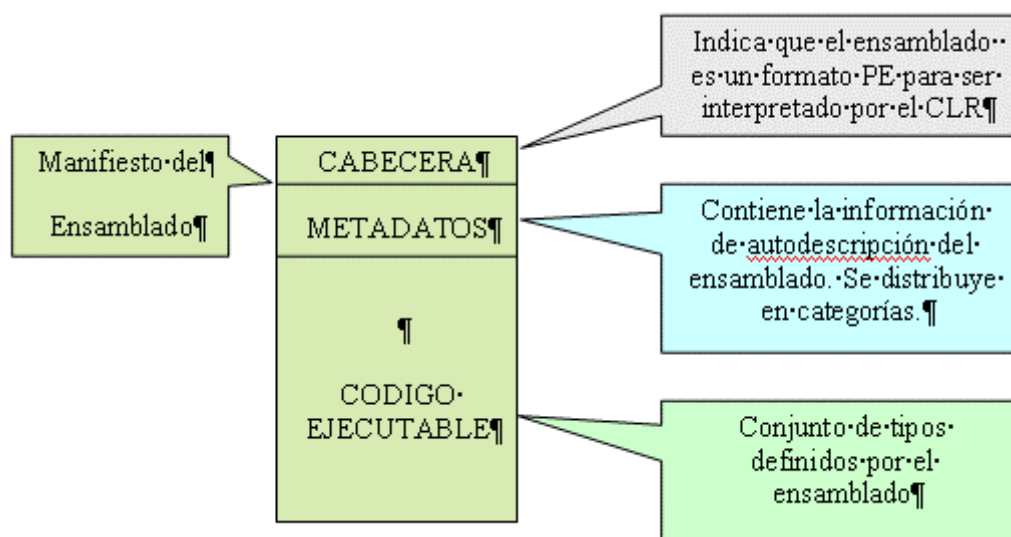


Figura 40. Estructura de un módulo de ensamblado en .NET

Contenido del manifiesto del ensamblado

La documentación de SDK de .NET realiza una lista, que reproducimos a continuación, modificada y adaptada, donde se muestra la información que contiene el **manifiesto** del ensamblado. Los cuatro primeros elementos (nombre del ensamblado, número de versión, referencia cultural e información sobre el nombre seguro) constituyen la **identidad del ensamblado**.

Información	Descripción
Nombre del ensamblado	Cadena de texto donde se especifica el nombre del ensamblado.
Número de versión	Número de versión principal y secundaria, y número de revisión y de versión de compilación. Common Language Runtime utiliza estos números para exigir las directivas de versión.
Referencia cultural	Información sobre la referencia cultural o idioma que admite el ensamblado. Esta información se debe utilizar sólo para designar un ensamblado como ensamblado satélite que contiene información específica sobre la referencia cultural o el idioma. (Se asume que un ensamblado con información de referencia cultural es un ensamblado satélite).
Información sobre el nombre seguro	Clave pública del editor si el ensamblado tiene un nombre seguro

Lista de todos los archivos del ensamblado	Un código hash de cada archivo que contiene el ensamblado y nombre de archivo . Tenga en cuenta que todos los archivos que componen el ensamblado deben encontrarse en el mismo directorio que el archivo que contiene el manifiesto de ensamblado.
Información de referencia de tipos	Información que utiliza el motor de tiempo de ejecución para asignar una referencia de tipos al archivo que contiene su declaración e implementación. Se utiliza para tipos que se exportan desde el ensamblado.
Información sobre ensamblados a los que se hace referencia	Lista de otros ensamblados a los que este ensamblado hace referencia estáticamente. Cada referencia incluye el nombre del ensamblado dependiente, los metadatos del ensamblado (versión, referencia cultural, sistema operativo, etc.) y la clave pública, si el ensamblado tiene un nombre seguro.

Tabla 22

Si queremos agregar o modificar información en el manifiesto del ensamblado, podemos utilizar atributos de ensamblado en su código. Podemos, por ejemplo, cambiar la información de versión y los atributos informativos, como *Trademark*, *Copyright*, *Product*, *Company* e *Informational Version*.

Desde el entorno de desarrollo de Visual Studio .NET, cuando se crea una nueva aplicación, además del fichero propio de código fuente asociado al tipo de aplicación que estemos haciendo, se genera automáticamente un fichero denominado **AssemblyInfo.cs**, que permite la modificación de los valores de atributos del ensamblado, y que es utilizada posteriormente por el compilador en el proceso de construcción del ejecutable. A continuación listamos el código fuente de dicho fichero predeterminado junto a los comentarios que se incluyen en él (Fíjese el lector en que no existe código fuente: sólo atributos que sirven de directivas para el compilador, pero –aun así– se referencian dos librerías (**System.Reflection**, y **System.Runtime.CompilerServices**)).

```
using System.Reflection;
using System.Runtime.CompilerServices;

//
// La información general de un ensamblado se controla mediante el siguiente
// conjunto de atributos. Cambie estos atributos para modificar la información
// asociada con un ensamblado.
//
[assembly: AssemblyTitle("")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("")]
[assembly: AssemblyProduct("")]
[assembly: AssemblyCopyright("")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]

//
```

```
// La información de versión de un ensamblado consta de los siguientes cuatro
valores:
//
//     Versión principal
//     Versión secundaria
//     Versión de compilación
//     Revisión
//
// Puede especificar todos los valores o usar los valores predeterminados (número
de versión de compilación y de revisión)
// usando el símbolo '*' como se muestra a continuación:

[assembly: AssemblyVersion("1.0.*")]

//
// Si desea firmar el ensamblado, debe especificar una clave para su uso. Consulte
la documentación de
// Microsoft .NET Framework para obtener más información sobre la firma de
ensamblados.
//
// Utilice los atributos siguientes para controlar qué clave desea utilizar para
firmar.
//
// Notas:
// (*) Si no se especifica ninguna clave, el ensamblado no se firma.
// (*) KeyName se refiere a una clave instalada en el Proveedor de servicios
//     de cifrado (CSP) en el equipo. KeyFile se refiere a un archivo que
contiene
//     una clave.
// (*) Si se especifican los valores KeyFile y KeyName, tendrá
//     lugar el siguiente proceso:
//     (1) Si KeyName se puede encontrar en el CSP, se utilizará dicha clave.
//     (2) Si KeyName y KeyFile no existen, se instalará
//         y utilizará la clave de KeyFile en el CSP.
// (*) Para crear KeyFile, puede ejecutar la utilidad sn.exe (Strong Name).
//     Cuando se especifica KeyFile, la ubicación de KeyFile debe ser
//     relativa al directorio de resultados del proyecto, que es
//     %Directorio del proyecto%\obj\<configuración>. Por ejemplo, si KeyFile
//     se encuentra en el directorio del proyecto, el atributo AssemblyKeyFile se
especifica
//     como [assembly: AssemblyKeyFile("..\\..\\mykey.snk")]
// (*) Firma retardada es una opción avanzada; consulte la documentación de
//     Microsoft .NET Framework para obtener más información.
//
[assembly: AssemblyDelaySign(false)]
[assembly: AssemblyKeyFile("")]
[assembly: AssemblyKeyName("")]
```

Código fuente 124

De acuerdo con lo anterior, los atributos se dividen en los siguientes conjuntos de información:

- Atributos de **identidad del ensamblado**.
 - o Determinan la identidad de un ensamblado: nombre, versión y referencia cultural. Forman el nombre completo del ensamblado y son necesarios para hacer referencia al ensamblado en el código. Se pueden usar para establecer la versión y la referencia cultural de un ensamblado. El compilador o la utilidad **Assembly Linker (AL.exe)** establece el valor de nombre cuando se crea el ensamblado, basándose en el archivo que contiene el manifiesto del ensamblado. Por ejemplo, para establecer la referencia de versión a la 1.1.100.1 y la referencia cultural a español, el código fuente sería:

```
// establecer la referencia de versión a la 1.1.100.1
[assembly:AssemblyVersionAttribute("1.1.100.1")]
// la referencia cultural a español
[assembly:AssemblyCultureAttribute("es")]
```

Código fuente 125

- Atributos **informativos**.

- o Los atributos informativos se pueden usar para proporcionar información adicional del producto o de la compañía sobre un ensamblado. En la Tabla 23 se describen los atributos informativos que se pueden aplicar a un ensamblado:

Atributo informativo	Descripción
AssemblyCompanyAttribute	Valor de cadena que especifica el nombre de una compañía.
AssemblyCopyrightAttribute	Valor de cadena que especifica la información de copyright.
AssemblyFileVersionAttribute	Valor de cadena que especifica el número de versión del archivo Win32. Normalmente, toma como valor predeterminado la versión del ensamblado.
AssemblyInformationalVersionAttribute	Valor de cadena que especifica información de versión que no usa el motor de tiempo de ejecución, como un número completo de versión del producto.
AssemblyProductAttribute	Valor de cadena que especifica información del producto.
AssemblyTrademarkAttribute	Valor de cadena que especifica información de marca comercial.

Tabla 23

- o Estos atributos pueden aparecer en la página de propiedades de Windows del ensamblado, o se pueden sobrescribir con la opción de compilador /win32res para especificar un archivo de recursos de Win32 propio.

- Atributos de **manifiesto del ensamblado**.

- o se puede usar para proporcionar información del manifiesto del ensamblado, como título, descripción, el alias predeterminado y la configuración. En la Tabla 24 se describen los atributos de manifiesto del ensamblado.

Atributo de manifiesto del ensamblado	Descripción
AssemblyConfigurationAttribute	Valor de cadena que indica la configuración del ensamblado, por ejemplo si se trata de lanzamiento o de depuración. El motor de tiempo de ejecución no utiliza este valor.

AssemblyDefaultAliasAttribute	Valor de cadena que indica el alias predeterminado que utilizarán los ensamblados que hacen referencia. Este valor proporciona un nombre descriptivo si el nombre del ensamblado no lo es (por ejemplo, un valor GUID). Este valor se puede usar también como nombre breve del nombre completo del ensamblado.
AssemblyDescriptionAttribute	Valor de cadena que ofrece una breve descripción que resume la naturaleza y objetivos del ensamblado.
AssemblyTitleAttribute	Valor de cadena que especifica un nombre descriptivo para el ensamblado. Por ejemplo, el ensamblado denominado <i>comdlg</i> tiene el título de <i>Microsoft Common Dialog Control</i> .

Tabla 24

- Los atributos del nombre seguro se pueden usar para establecer un nombre seguro para un ensamblado. En la Tabla 25 se describen los atributos del nombre seguro.

Atributos de nombre seguro	Descripción
AssemblyDelaySignAttribute	Valor booleano que indica que se va a retrasar la firma del ensamblado.
AssemblyKeyFileAttribute	Valor de cadena que indica el nombre del archivo que contiene una clave pública (si se usa retraso de firma) o tanto la clave pública como la privada que se pasan como parámetros al constructor de este atributo. Tenga en cuenta que nombre de archivo es relativo respecto a la ruta de acceso del archivo de salida (.exe o .dll), no a la del archivo de código fuente.
AssemblyKeyNameAttribute	Especifica el contenedor de claves en que se encuentra el par de claves que se pasa como parámetro al constructor de este atributo.

Tabla 25

- o En el siguiente ejemplo de código se muestran los atributos que se han de aplicar cuando se usa la firma retrasada para crear un ensamblado con nombre seguro y con un archivo de clave pública denominado *MiClave.snk*.

```
[assembly:AssemblyKeyFileAttribute(@"..\..\MiClave.snk")]
[assembly:AssemblyDelaySignAttribute(true)]
```

Código fuente 126

Creación de nombres seguros para un ensamblado

Un nombre seguro está formado por la identidad del ensamblado (nombre de texto sencillo, número de versión e información de referencia cultural, si se proporciona), una clave pública y una firma digital. El mecanismo de firmado de un ensamblado permite que se pueda garantizar que la información de un ensamblado no ha sido alterada. Se basa en un sistema de criptografía de clave pública, llamado RSA¹⁸.

Para firmar un ensamblado con un nombre seguro, es necesario tener un par de claves pública y privada. Este par de claves criptográficas se usa en la compilación para crear un ensamblado con nombre seguro. Se puede generar un par de claves utilizando la herramienta Nombre seguro (*Sn.exe*). Normalmente, los archivos de par de claves tienen la extensión **.snk**.

Por ejemplo, el Código fuente 127 genera el fichero FirmaClave.snk:

```
sn -k FirmaClave.snk
```

Código fuente 127

Cuyo contenido es de tipo binario, y guarda un par de claves generadas automáticamente por la utilidad Sn.exe. En caso de que en el proceso de desarrollo, deseemos almacenar aparte la clave pública, el parámetro **-p** nos permite extraerla y guardarla en otro archivo de extensión **.snk**.

```
sn -p firmaclave.snk publica.snk
```

Código fuente 128

Desde Visual Studio .NET, se debe de tener en cuenta que para firmar un ensamblado con un nombre seguro, el IDE busca el archivo de clave en el directorio que contiene la solución de Visual Studio, mientras que el compilador de C# busca el archivo de clave en el directorio que contiene el binario. Para asignar el archivo de clave se ubica en el directorio del proyecto apropiado y se establecen los valores correspondientes en el fichero de configuración.

```
[C#]  
[assembly: AssemblyKeyFileAttribute(@"..\..\firmaclave.snk")]
```

Código fuente 129

Además de las comentadas, Sn.exe, dispone de un conjunto completo de parámetros de configuración cuya lista completa incluimos en la Tabla 26, para que pueda servir de referencia rápida:

¹⁸ RSA utiliza un código de dispersión denominado SHA-1. El lector interesado, puede encontrar información más detallada en la documentación del MSDN.

Lista de parámetros admitidos por la utilidad de generación de Nombres Seguros (Sn.exe)	
-c [<csp>]	Establecer/restablecer el nombre del CSP que se va a utilizar utilizar con operaciones MSCORSN.
-d <container>	Eliminar el contenedor de claves denominado <contenedor>.
-D <assembly1> <assembly2>	Comprobar que el <ensamblado1> y el <ensamblado2> sólo se diferencian por la firma.
-e <assembly> <outfile>	Extraer clave pública de <ensamblado> a <archivo_de_salida>.
-i <infile> <container>	Instalar par de claves desde <archivo_de_entrada> en un contenedor de claves denominado <contenedor>.
-k <outfile>	Generar un nuevo par de claves y escribirlo en <archivo_de_salida>.
-m [y/n]	Habilitar (s), deshabilitar (n) o comprobar (sin parámetros) si los contenedores de claves son específicos para equipos (más que específicos para usuarios).
-o <infile> [<outfile>]	Convertir la clave pública de <archivo_de_entrada> en archivo de texto <archivo_de_salida> separado por comas. Si se omite <archivo_de_salida>, el texto se copia al portapapeles
-p <infile> <outfile>	Extraer la clave pública del par de claves de <archivo_de_entrada> y exportarlo a <archivo_de_salida>.
-pc <container> <outfile>	Extraer la clave pública del par de claves de <contenedor> y exportarlo a <archivo_de_salida>.
-q	Modo no interactivo. Esta opción debe estar al principio de la línea de comandos y suprimir cualquier resultado que no sean mensajes de error
-R <assembly> <infile>	Volver a firmar el ensamblado firmado o parcialmente firmado con el par de claves de <archivo_de_entrada>.
-Rc <assembly> <container>	Volver a firmar el ensamblado firmado o parcialmente firmado con el par de claves del contenedor de claves denominado <contenedor>.
-t[p] <infile>	Mostrar símbolo (token) para clave pública en <archivo_de_entrada> (junto con la propia clave pública si se utiliza -tp).
-T[p] <assembly>	Mostrar símbolo (token) para clave pública de <ensamblado> (junto con la propia clave pública si se utiliza -Tp).
-v[f] <assembly>	Comprobar la consistencia de la firma de nombre seguro en <ensamblado>. Si se especifica -vf, forzar comprobación incluso si no está habilitada en el Registro
-Vl	Mostrar configuración actual para comprobación de nombre seguro en este equipo.

-Vr <assembly> [<userlist>]	Registrar <ensamblado> para pasar por alto la comprobación (con una lista de nombres de usuario opcional separada por comas para la que tendrá efecto esta función). Se puede especificar: <ensamblado> como * para indicar todos los ensamblados o, *,<símbolo_de_clave_pública> para indicar todos los ensamblados con el símbolo de clave pública proporcionado. Debe especificar los símbolos (tokens) de clave pública como una cadena de dígitos hexadecimales.
-Vu <assembly>	Anular el registro de <ensamblado> para pasar por alto la comprobación. Se sigue la misma convención de nomenclatura para <ensamblado> que para -Vr.
-Vx	Quitar todas las entradas de omisión de comprobación.

Tabla 26

Una vez firmado un ensamblado, se puede ubicar en la **Caché Global de Ensamblados (GAC)**, y podrá ser utilizado por cualquier programa que haga referencia a él, de forma similar a como hacemos referencia a las librerías propias del entorno de ejecución. No obstante, esa operación no se efectúa de forma automática, sino que para ello, disponemos de una utilidad, llamada **Gacutil.exe**, a la que basta llamar con el parámetro **/install** ó **/i** y el nombre de la librería, para quede registrada.

Por ejemplo, si disponemos de una librería llamada LibreriaGAC.DLL, y hemos generado el fichero de nombre seguro, utilizando el nombre anterior (*FirmaClave.snk*) que se encuentra en el mismo directorio de la librería, la llamada a Gacutil.exe

```
Gacutil /i LibreriaGAC.DLL
Assembly successfully added to the cache (Ensamblado añadido correctamente al GAC)
```

Código fuente 130

Bastaría para registrarla. Posteriormente, podemos comprobar este extremo desde el propio explorador de Windows, ya que .NET instala una utilidad integrada en él, que permite ver el contenido de las librerías del GAC, como si fuera un directorio más (dentro de Windows\Assembly), tal y como vemos en la Figura 41.

Por lo tanto el nuevo modelo .NET de componentes, también implica cambios en la política de ubicación de los componentes, que ya no dependen del registro, y que no obstante, siguen participando de muchas de las características anteriores, como son la posibilidad de acceso remoto, y la posibilidad de compartir componentes por parte de más de una aplicación.

Veremos más sobre estos aspectos cuando comentemos los temas relacionados con la construcción de componentes y al analizar el funcionamiento de los servicios Web XML, en los capítulos posteriores, dedicados a la construcción de aplicaciones.

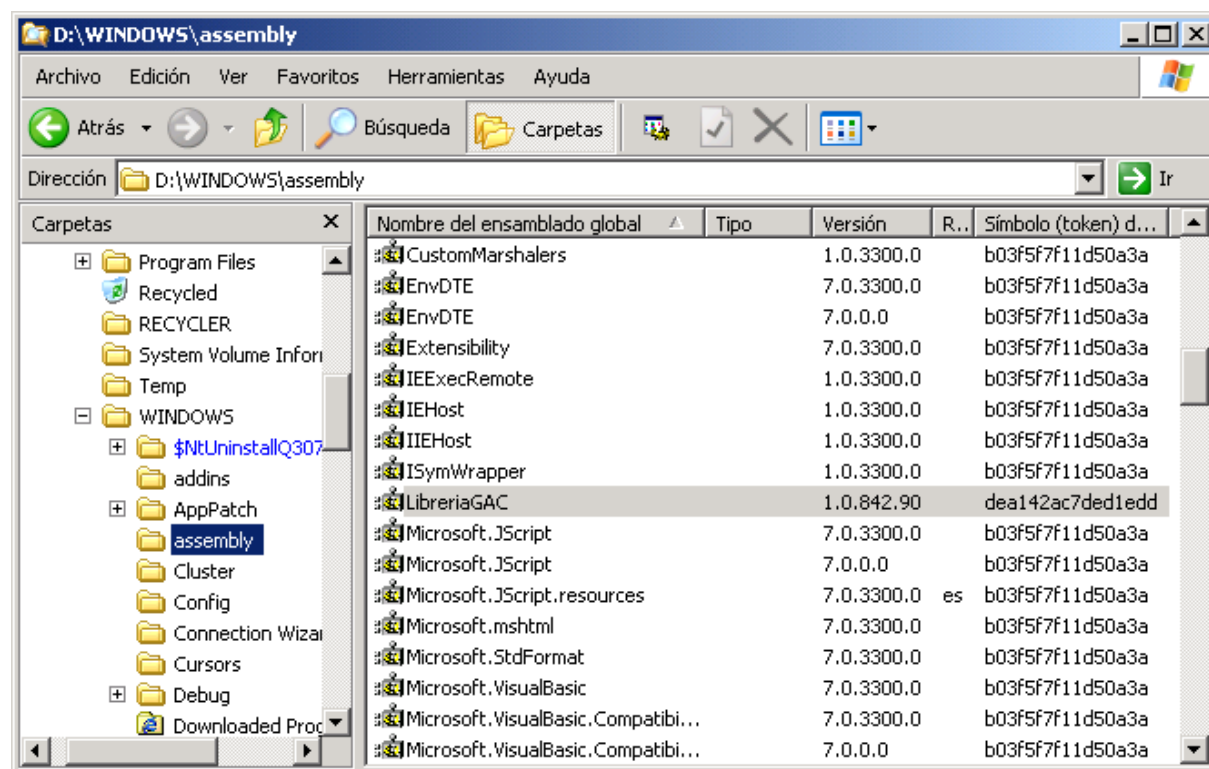


Figura 41. El directorio del Cache Global de Ensamblados (GAC)

Aplicaciones Windows

Hasta ahora, en la mayor parte de los ejemplos, hemos estado utilizando aplicaciones de consola, para centrarnos preferentemente en el lenguaje. A pesar de que esta obra tiene ese objetivo principal (y no el estudio de la construcción de aplicaciones Windows y ASP.NET se tratará en otra obra dedicada a tal efecto), añadimos aquí 3 capítulos dedicados a la construcción de aplicaciones Windows, al tratamiento de datos y a la creación de páginas Web y de Servicios Web, ya que, una vez conocidos los mecanismos de programación, es interesante revisar –siquiera someramente- la construcción de aplicaciones y de ver los recursos que .NET nos ofrece a este respecto.

El espacio de nombres Windows.Forms

Dentro del espacio de nombres denominado *Windows.Forms*, se encuentran la mayoría de los recursos necesarios para la construcción de aplicaciones Windows, que, fundamentalmente, podemos dividir en cuatro categorías:

- Aplicaciones estándar de Windows
- Librerías de clases
- Controles de usuario
- Servicios del Sistema

Observe el lector, que han desaparecido algunos de los tipos de proyecto que se presentaban en Visual Studio 6, y aparecen, no obstante, algunas opciones nuevas como los servicios del sistema.

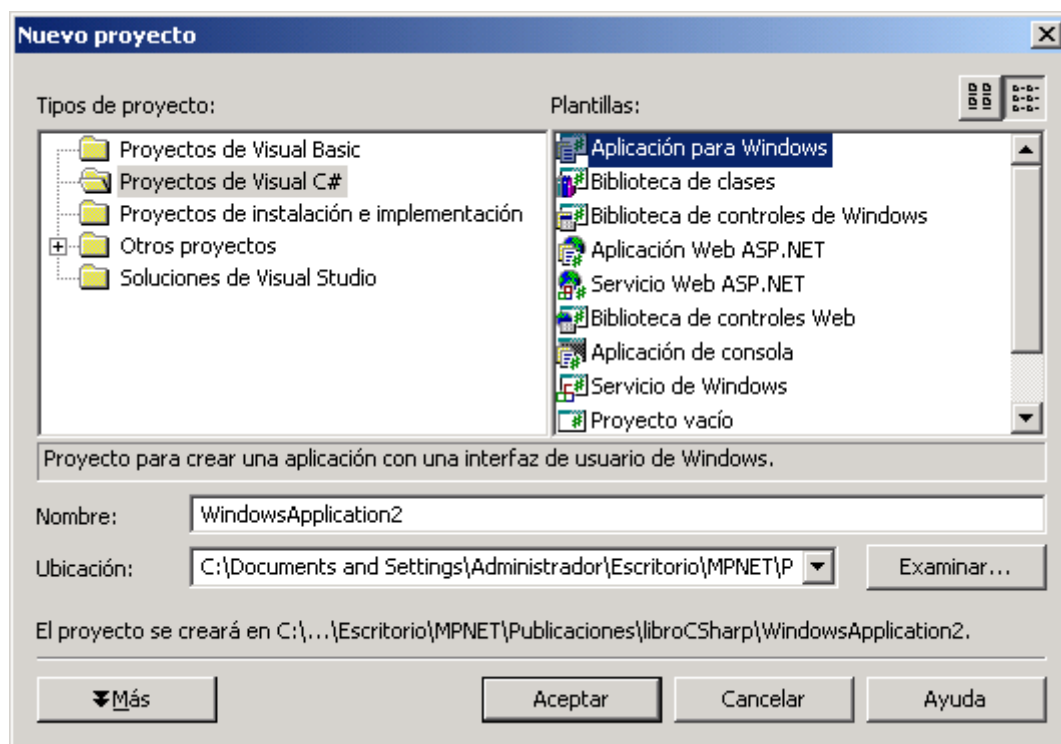


Figura 42. Tipos de Aplicación en Visual Studio .NET

Las aplicaciones que han desaparecido, son todas aquellas relacionadas con ActiveX, ya que el nuevo modelo de ejecutables, basado en Metadatos no soporta la construcción de componentes de ese tipo. Las aplicaciones DHTML, se sustituyen por las nuevas aplicaciones ASP.NET (y servicios Web ASP.NET), y tanto los controles de usuario como los componentes, son ahora simplemente, librerías, ficheros con extensión DLL, aunque basados en plantillas diferentes, dependiendo de la aplicación a construir.

Por su parte, el IDE de Visual Studio, dispone de un conjunto de elementos ya creados para que podamos basarnos en ellos, y nos ofrece vistas de estos elementos en las ventanas adecuadas: de esa forma, podemos ver los servicios del sistema operativo y todo aquello que ahora es posible programar relacionado con el sistema, en la vista del *Explorador de Servidores*.

Esto incluye la posibilidad de programar en una aplicación las colas de mensajes del sistema, la manipulación de procesos y servicios básicos, la gestión del registro de eventos, y en lo referente al tratamiento de datos, la posibilidad de manipular la información de un servidor desde el propio IDE de Visual Studio, sin la necesidad de acudir a una herramienta adicional, y la generación y manejo de informes, gracias a una nueva versión de la librería **Crystal Reports**, que ahora se presenta como un conjunto de jerarquías de clases muy potente, capaz de generar incluso vistas de datos en páginas HTML, ASP.NET y similares.

Hay que destacar igualmente, que, al tratarse la gestión de datos de algo especialmente sensible en la construcción de aplicaciones, se ha añadido a este explorador la posibilidad de visualizar las conexiones de datos válidas en la máquina donde se esté desarrollando, de forma que podamos manipular las bases de datos de forma remota, sin necesidad de mantener abierto un gestor de administración de bases de datos, al estilo del de SQL-Server, que posibilita incluso la creación de objetos en el servidor, tales como la creación de tablas, Diagramas, Vistas, Procedimientos Almacenados, *Triggers*, etc.

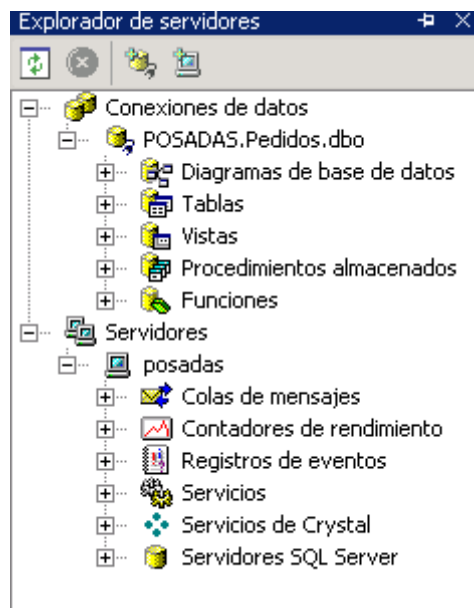


Figura 43. Explorador de Servidores

Por supuesto, además de esta nueva ventana, disponemos de un Cuadro de Herramientas que es, igual que todo el IDE, sensible al contexto, lo que significa que –dependiendo del proyecto seleccionado– nos ofrecerá distintas vistas de conjuntos de controles disponibles (Controles Windows, Controles Web, componentes de datos, etc.).

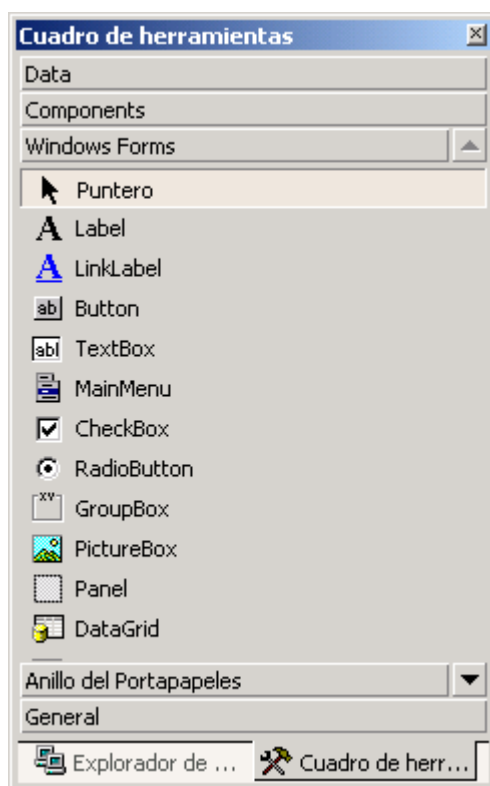


Figura 44. Cuadro de Herramientas mostrando las subcarpetas disponibles

Hay que comentar, llegados aquí que todos los controles han sido reconstruidos para su funcionamiento en .NET y que –de hecho- el propio IDE de Visual Studio, ha sido programado en un 90% en lenguaje C#, y un 10% en C++.

Formularios Windows

Una vez seleccionada una aplicación Windows,. El IDE nos ofrece de forma predeterminada una ventana de nombre **Form1** (todos esos aspectos y, prácticamente, todos, pueden modificarse y configurarse). Esta ventana es un puente entre el **Diseñador de Windows** y el código fuente de dicha ventana (para los programadores de Visual Basic, la diferencia fundamental es que ahora, todo lo que se diseña, se traduce en *código fuente* visible en la ventana de edición).

Al objeto de evitar colisiones entre un modo de programar (diseño visual) y otro (código fuente puro), la ventana generada nos recuerda que existe una zona de código fuente, que no deberíamos de tocar, ya que se corresponde con el trabajo hecho por el diseñador. Esta novedad, llamada **region de código**, consiste en una directiva de preprocesado (*#region*) que define una zona de código colapsable, de forma que el usuario pueda centrarse especialmente en el código fuente propio (No obstante, y –aun a riesgo de que se tenga que recompilar- el usuario puede modificar cualquier zona de código, por supuesto).

Una vez eliminados todos los comentarios, excepto el del propio Diseñador de Código, una ventana vacía, predeterminada, viene dada por el Código fuente 131.

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

namespace WindowsApplication2
{
    public class Form1 : System.Windows.Forms.Form
    {
        private System.ComponentModel.Container components = null;

        public Form1()
        {
            InitializeComponent();
        }
        protected override void Dispose( bool disposing )
        {
            if( disposing )
            {
                if (components != null)
                {
                    components.Dispose();
                }
            }
            base.Dispose( disposing );
        }

        #region Windows Form Designer generated code
        /// <summary>
        /// Método necesario para admitir el Diseñador, no se puede modificar
        /// el contenido del método con el editor de código.
        /// </summary>
        private void InitializeComponent()
        {
```

```

        {
            this.components = new System.ComponentModel.Container();
            this.Size = new System.Drawing.Size(300,300);
            this.Text = "Form1";
        }
    #endregion

    [STAThread]
    static void Main()
    {
        Application.Run(new Form1());
    }
}

```

Código fuente 131

Donde hay una serie de puntos a reseñar, respecto a este código predeterminado:

- Se hacen referencia a varios espacios de nombres, de los que –en principio- sólo se usa uno: ***System.Windows.Forms***.
- Se asume un espacio calificado (*namespace*) que contiene todo el código, con el nombre del proyecto.
- Todos las ventanas (o formularios) heredan de la clase ***System.Windows.Forms.Form*** (no existe la distinción artificial que se daba en Visual Basic 6.0 entre formularios MDI y formularios *normales*).
- El método constructor ***Form1()***, llama de forma predeterminada a la función ***InitializeComponent()***, que es la encargada de instanciar todos los objetos de la ventana.
- Se sobreescribe el método ***Dispose()***, encargado de indicar al *Recolector de Basura*, que puede ocuparse de los recursos ocupados por la ventana. La llamada al destructor se realiza inmediatamente antes de destruirlo mediante la recolección de elementos no utilizados y la reclamación de la memoria que ocupa¹⁹. Dado que existe un retardo entre el momento en que un usuario termina con el componente y el momento en que se llama a su destructor, se introdujo un paso adicional en el ciclo de vida de los componentes .NET: si el componente adquiere recursos del sistema, como conexiones de base de datos o controladores de objetos del sistema Windows, se recomienda implementar la interfaz ***IDisposable*** y proporcionar un método ***Dispose***, de modo que los usuarios del componente puedan elegir cuándo se liberan estos recursos.
- Si el componente hereda de ***System.ComponentModel.Component***, se proporciona una implementación predeterminada de ***Dispose***. El usuario puede situar en ese método las acciones adicionales que crea convenientes, en caso de haber utilizado recursos propios, que de forma predeterminada no sean manejados por el Recolector de Basura.

¹⁹ Una vez que la recolección de elementos no utilizados determina que el componente ya no está al alcance de ningún código en ejecución, Common Language Runtime llama al destructor del componente. Esto sucede si se liberaron todas las referencias al componente o si las únicas referencias al componente las mantienen objetos aislados –de forma similar a él- de todo el código en ejecución, por ejemplo, en el caso de referencias circulares.

- Los controles de una ventana, se almacenan en un objeto contenedor especial, que es una instancia de la clase ***System.ComponentModel.Container***. (Esto va a posibilitar que posteriormente, se puedan aplicar a los controles las nuevas propiedades de redimensionamiento dinámico, anclaje, etc.)
- Se crea un punto de entrada de la aplicación, definiendo un método ***Main()*** estático, y se califica dicho método con el atributo **[STAThread]**, que indica que se trata del punto de entrada de la aplicación (téngase en cuenta que podríamos disponer de varios métodos **Main()**, pero sólo uno se identifica como entrada del programa)²⁰:
- Respecto a la configuración predeterminada del entorno en lo que concierne a la configuración de la aplicación, se asume que estamos en modo **Debug**, aunque cada compilación generará – en realidad- dos ejecutables, uno en el directorio `\bin\debug` y otro `\obj\debug`, específico para opciones de depuración. Toda la estructura de directorios, así como la asignación de un icono predeterminado y la creación de un fichero de recursos asociado a la ventana (**Form1.resx**), son generados al iniciar una nueva solución, como podemos comprobar en los botones del Explorador de Soluciones, seleccionando la opción *"Mostrar todos los Archivos"* (Ver Figura 45).

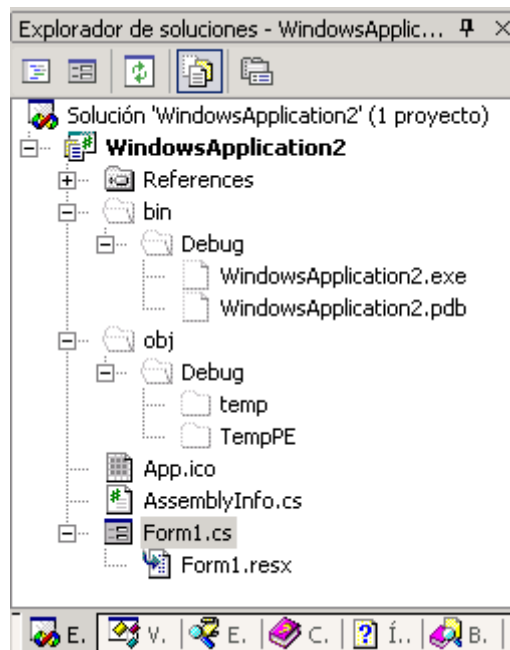


Figura 45. Conjunto de recursos predeterminado a una aplicación Windows

- La forma de iniciar la aplicación es mediante una llamada al método ***Run()*** del objeto ***Application***, que inicia la ejecución de un bucle de mensajes de aplicación estándar de Windows en el subproceso actual. En este caso, se llama a un **overload** que hace que el

²⁰ Windows Forms utiliza el modelo de apartamento de un único subproceso (STA) porque se basa en las ventanas nativas de Win32 que tienen subprocesos de tipo apartamento de forma inherente. El modelo STA implica que es posible crear una ventana en cualquier subproceso, pero que no se puede cambiar de subproceso una vez creada, y todas las llamadas de función a ella deben realizarse en su subproceso de creación. Fuera de Windows Forms, las clases de .NET Framework utilizan el modelo de subprocesamiento libre.

formulario especificado esté visible (fíjese el lector, que de no hacerlo así, la ventana se abriría, para cerrarse inmediatamente después, al concluir la ejecución del método `Main()`).

- La clase ***Application*** proporciona propiedades y métodos *estáticos* para administrar una aplicación (por ejemplo, los métodos para iniciarla ***-Run()***- y detenerla ***-Exit()***-), para procesar mensajes de Windows, y propiedades para obtener información acerca de una aplicación. No se puede heredar esta clase, pero suministra valores útiles acerca de la instancia en ejecución. Si se desean más datos sobre el contexto de ejecución de una aplicación, consúltese la clase estática ***System.Environment***.
- Otros métodos dignos de mención de la clase ***Application***, son:
 - o ***DoEvents()***, para procesar mensajes de sistema mientras se ejecuta un bucle
 - o ***ExitThread()***, para detener un bucle de mensajes iniciado como un subproceso
 - o ***AddMessageFilter()***, que permite realizar de forma sencilla un proceso de subclasificación, esto es, permite capturar los eventos dirigidos a la aplicación actual, para poder determinar si implementamos una acción no predeterminada.

Otra cuestión a tener en cuenta se refiere al enorme conjunto de propiedades, métodos, y eventos disponibles para los objetos de la clase `Form`. Entre otros, existen métodos que hacen tremendamente simples las tareas más complejas que antes requerían del uso de las MFC, (o de uso de API's avanzadas en Visual Basic) como los mecanismos de subclasificación de objetos, que permiten controlar los mensajes que Windows les dirige, al objeto de aportar un comportamiento personalizado, según las necesidades.

Ejemplo: Subclasificación de ventanas

En .NET, la tarea es tan sencilla como la sobreescritura del procedimiento de ventana al que van dirigidos los mensajes: ***WndProc()***, que recibe el mensaje de Windows como argumento. Por ejemplo, si queremos controlar cualquier acción del botón derecho del ratón, nos basta con determinar qué mensajes le corresponden (en este caso, son los números 512 al 514), y proporcionar una implementación de ese método en la clase `Form1()`, que controle y asigne otro comportamiento a cualquier pulsación del botón.

```
//Filtrar mensajes de Windows
protected override void WndProc(ref Message m)
{
    // bloquea los mensajes relativos al botón izquierdo del ratón
    if (m.Msg >= 513 && m.Msg <= 515)
    {
        MessageBox.Show("Pulsado botón izquierdo: " + m.Msg);
    }
    base.WndProc(ref m);
}
```

Código fuente 132

Hay que recordar que, en éstas técnicas, conviene siempre llamar al procedimiento asociado de la clase padre, (***base***), para evitar comportamientos extraños.

Comportamiento básico de formularios estándar y MDI

Como se ha indicado antes, y a diferencia de lo que sucedía en Visual Basic, una ventana MDI no es una ventana especial, sino simplemente, una ventana con una propiedad establecida. (*IsMDIContainer* = *true*). En el otro sentido una ventana hija no es una ventana con una propiedad específica, sino que el hecho de ser o no ser hija de otra ventana de la aplicación, es algo que se resuelve en tiempo de ejecución, dependiendo del modo en que la ventana sea lanzada. Veamos esto con más detalle.

Si queremos que una ventana sea lanzada como hija de otra, -a la que previamente le habremos asignado su propiedad *IsMDIContainer* a *true*- basta con asignar su propiedad *MDIParent* en tiempo de ejecución a la ventana MDI que hará de contenedora.

```
// Si disponemos de la ventana MDI llamada Form1
// y otra ventana no-MDI de nombre Form2, podemos
// lanzar Form2 como hija de Form1, mediante:
Form2 f = new Form2();
f.MDIParent = this // asignamos a la ventana actual (contenedora), la propiedad //
                  // MDIParent de la ventana contenida
f.Show();
```

Código fuente 133

De no incluir la línea de asignación de la propiedad *MDIParent*, la ventana se lanzaría igualmente, pero no sería hija de la primera. Otra posibilidad, es lanzar la segunda ventana como una caja de diálogo, mediante el método *ShowDialog()*.

Aparte de la flexibilidad de este comportamiento, ésta segunda posibilidad tiene –en .NET- algunas implicaciones interesantes:

- Todo formulario lanzado como caja de diálogo, se comporta como una ventana modal de aplicación, no permitiendo ninguna otra operación sobre la aplicación hasta que no sea cerrada.
- El mismo formulario puede ser lanzado como modal o no dependiendo de las circunstancias.
- Todo formulario lanzado como caja de diálogo puede devolver valores relacionados con la respuesta del usuario, asociando mediante código o diseño propiedades predeterminadas a algunos controles de la misma (típicamente, botones, aunque podría ser cualquier otro). En caso de que esto se haga así los valores devueltos son siempre alguno de los predeterminados en la clase estática **DialogResult**, que contiene propiedades para todas las posibles respuestas estándar permitidas (Si, No, Aceptar, Ignorar, Reintentar, Cancelar y Abortar).

Por ejemplo, si asociamos al botón **Botón1** la propiedad **Cancel** y al **Botón2** la propiedad **OK**, podríamos utilizar el Código fuente 134 para saber la respuesta del usuario.

```
// Si Form1 dispone de dos botones Botón1 y Botón2
// y asociamos las propiedades Cancel y OK a dichos botones
// al lanzar Form1 como Caja de Dialogo podemos hacer lo siguiente:

Form1 f = new Form1();

if (f.ShowDialog() == DialogResult.OK)
{
```

```
MessageBox.Show("Botón OK pulsado"); //Se ha pulsado Botón2  
};
```

Código fuente 134

Por lo demás, el diseño de ventanas permite asociar la mayor parte de las propiedades sin necesidad de escribir código, de forma muy similar a como hacían los programadores de Visual Basic en la versión 6.0 de Visual Studio. Vamos a revisar algunas de las más significativas de esas propiedades (por orden alfabético):

Algunas propiedades destacadas de los formularios

Ubicación y Tamaño

- Propiedades **Location** y **Size**: En lugar de Top y Left la ubicación pasa a establecerse mediante un objeto **Location**, con las propiedades X e Y en su lugar).
- Respecto al tamaño, Width y Height se convierten en un objeto **Size**, que dispone de dichos valores como propiedades. Además, se han incluido las propiedades **MinimumSize**, que gestiona otro objeto **Size**, y controla que la ventana no se pueda dimensionar por debajo de los límites establecidos en él y su homónima, **MaximumSize**.

Botones de la Ventana

- Vienen determinados por la propiedad **FormBorderStyle** (que determina el estilo de la ventana, y con ello gran parte de su apariencia, incluyendo los botones propios de cada estilo).
- Específicamente, las propiedades **ControlBox** (que determina si la ventana tiene o no botones de alguna clase), **MaximizeBox** (Botón de Maximizar), **MinimizeBox** (Botón de Minimizar) y **HelpButton** (Si debe aparecer el botón de Ayuda (símbolo ?) asociado a algunas ventanas), permiten el ajuste fino del diseño de botones.

Aspecto Visual

Este apartado podría englobar a las propiedades:

- **ForeColor** (Color de primer plano, para texto y gráficos)
- **BackColor** (Color de fondo, para texto y gráficos)
- **BackgroundImage** (Imagen de fondo del formulario)
- **Icon** (especifica el icono de la ventana)
- **Opacity** (indica el grado de transparencia de la ventana: 0 -> invisible, 1-> opaca. Esto implica la desaparición de la propiedad **visible**, que pasa a ser un caso particular de lo anterior)
- **Text** (texto de la ventana. Anteriormente, llamada *Caption*)

Comportamiento

Podemos indicar aquí algunas novedades interesantes:

- **DockPadding** (que determina el tamaño del borde de los controles asociados)
- **StartPosition** (determina la posición inicial del formulario al lanzarse)
- **TopMost**: (establece si el formulario debe de permanecer en primer plano, independiente de que pierda el foco).
- **WindowState**: (Establece el estado de la ventana: Maximizada, minimizada o normal)

Otros elementos de la interfaz de usuario

Sistema de Menús: Los menús se gestionan ahora mediante controles. Para ello, existen controles asociados del tipo **MainMenu** (Menú principal), **MenuItem** (Submenú) y el propio formulario dispone de la propiedad **ContextMenu**, mediante la cual podemos asociar el elemento de menú que se mostrará cuando pulsemos botón derecho sobre el formulario.

Estos menús, son editables directamente, como se muestra en la Figura 46, y se corresponden con objetos de las clases de las que instancian.

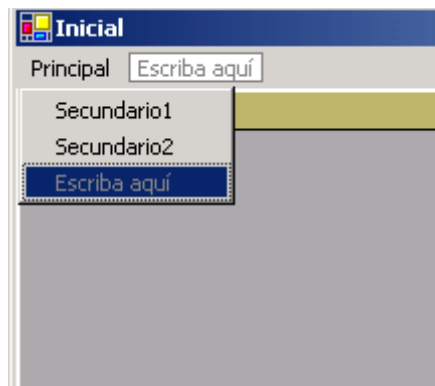


Figura 46. Sistema de edición de controles *MenuItem*.

Respecto a su programación, cabe indicar, que –como cualquier otro control de la ventana- se declara como una instancia del objeto correspondiente, pudiéndose heredar de ellos (para crear nuestro propio sistema de menús con un aspecto visual modificado, por ejemplo), y, por tanto, modificar su funcionamiento mediante mecanismos de herencia.

El código fuente asociado a los controles anteriores es el que muestra el Código fuente 135.

```
//Declaración de los objetos menú

private System.Windows.Forms.MainMenu mainMenu1;
private System.Windows.Forms.MenuItem menuItem1;
private System.Windows.Forms.MenuItem menuItem2;
private System.Windows.Forms.MenuItem menuItem3;

//Llamada al constructor (en el método del formulario InitializeComponent())
```

```

this.mainMenu1 = new System.Windows.Forms.MainMenu();
this.menuItem1 = new System.Windows.Forms.MenuItem();
this.menuItem2 = new System.Windows.Forms.MenuItem();
this.menuItem3 = new System.Windows.Forms.MenuItem();

// mainMenu1
//
this.mainMenu1.MenuItems.AddRange(new System.Windows.Forms.MenuItem[] {
    this.menuItem1});

// menuItem1
//
this.menuItem1.Index = 0;
this.menuItem1.MenuItems.AddRange(new System.Windows.Forms.MenuItem[] {
    this.menuItem2,
    this.menuItem3});
this.menuItem1.Text = "Principal";

// menuItem2
//
this.menuItem2.Index = 0;
this.menuItem2.Text = "Secundario1";

// menuItem3
//
this.menuItem3.Index = 1;
this.menuItem3.Text = "Secundario2";

//Finalmente, se asocia el sistema de menús a la propiedad Menu del formulario:
this.Menu = this.mainMenu1;

```

Código fuente 135

Hay que notar que el verdadero manejador de menús es el objeto **MainMenu**, que alberga en la propiedad colectiva **MenuItems** a todo el conjunto de menús. Eso significa que podríamos hacer un tratamiento de los menús basado en las matrices contenidas en este objeto, o sea, referirnos a los menús a través de la colección.

```

foreach (MenuItem m in mainMenu1.MenuItems)
{
    MessageBox.Show(m.Text);
}

```

Código fuente 136

Respecto a la programación, el entorno diseña automáticamente la declaración formal del procedimiento de evento correspondiente, y asigna el manejador apropiado, cuando hacemos doble clic sobre alguno de ellos.

```

//Declaración del manejador de eventos para el MenuItem

this.menuItem1.Click += new System.EventHandler(this.menuItem1_Click);

//Procedimiento de evento
private void menuItem1_Click(object sender, System.EventArgs e)
{

```

```

    MessageBox.Show("Pulsado el primer ítem de Menú");
}

```

Código fuente 137

Barras de botones y Barras de estado

Al igual que en la versión anterior, disponemos de los controles **ToolBar** y **StatusBar**, que implícitamente soportan colecciones internas de objetos **Button** y **Panel**, respectivamente.

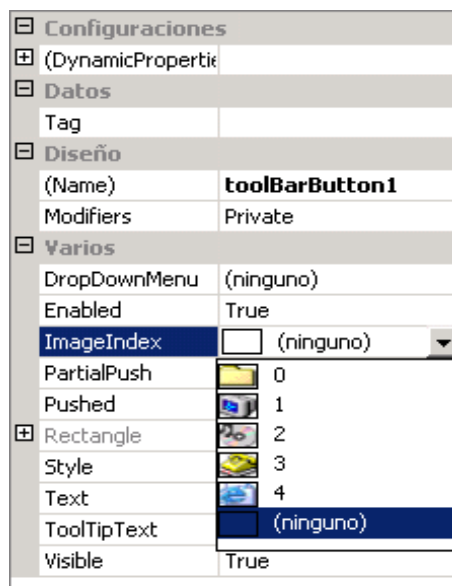


Figura 47. Asignación de propiedades de imagen a un botón

El aspecto visual de los botones se define mediante sus propiedades, una de las cuales se asocia a un control contenedor de imágenes —en concreto, un control **ImageList**—. Una vez que el control se rellena de imágenes y se asocia a la propiedad genérica **ImageList** del propio **ToolBar**, cada botón dispone de una propiedad **ImageIndex** que permite —desde el IDE de Visual Studio— asociar visualmente la imagen deseada.

Por otro lado, un mismo control **ImageList** puede servir de contenedor de imágenes para diferentes controles simultáneamente. El diseño anterior podría tomar un aspecto similar al que muestra la Figura 49.

Donde siguen existiendo la mayor parte de las propiedades ya conocidas desde versiones anteriores, pudiendo, no obstante, manipular el objeto haciendo uso de la herencia, como sucede con todo el conjunto de controles disponibles desde el entorno de desarrollo.

(Consideramos innecesario detallar más aspectos del diseño de estos controles, por resultar evidentes para cualquier usuario acostumbrado al trabajo con alguna herramienta de desarrollo visual).

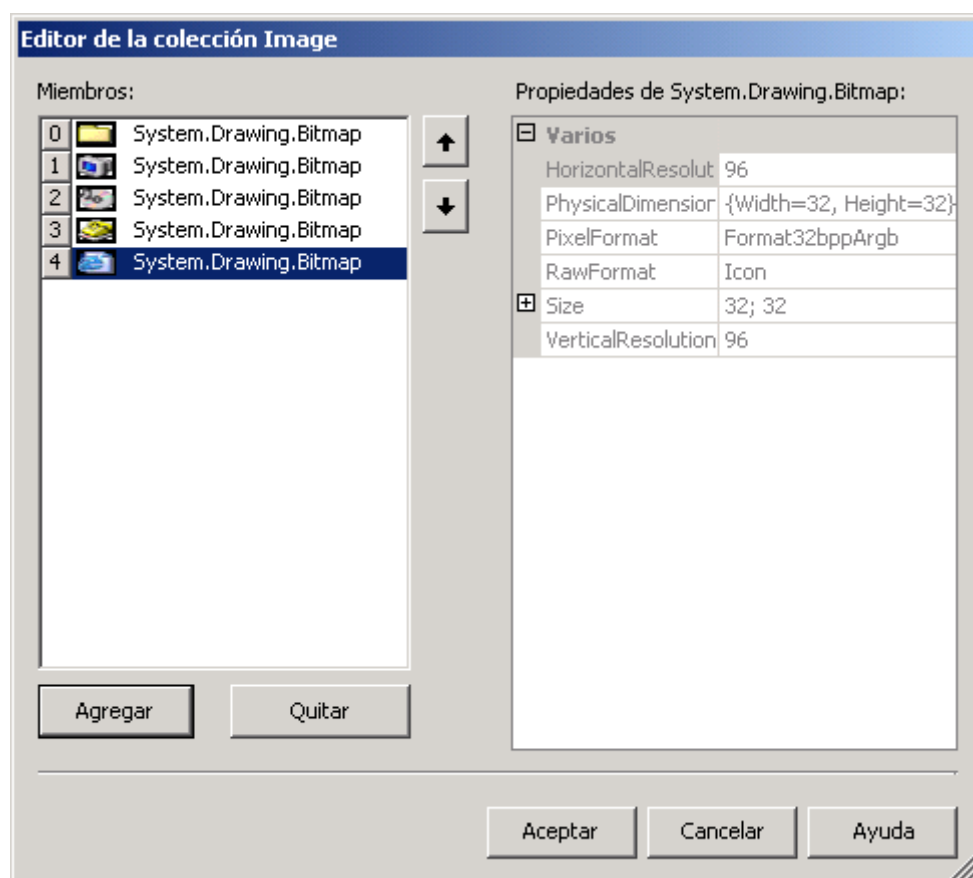


Figura 48. Control ImageList mostrando la lista de imágenes

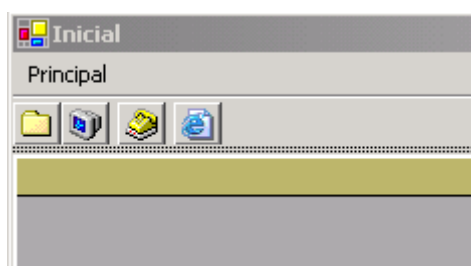


Figura 49. Aspecto final de la Barra de Botones

Otras novedades en los controles de interfaz de usuario

El conjunto completo de controles de la interfaz de usuario ha sido rediseñado mediante tecnología .NET, primero estructurales, y además, para permitir que se puedan utilizar las técnicas de orientación a objetos lo que nos permite la herencia visual, la sobrecarga de métodos de controles existentes, y un largo etcétera de posibilidades nuevas.

No obstante, la librería *VB6.Compatibility* alberga las versiones anteriores de un grupo de esos controles, y también podemos utilizar otros mediante las técnicas de Interop, ya mencionadas. El lector no tendrá problemas en comprender su funcionamiento, a partir de la propia interfaz de usuario.

El lector comprobara este hecho al programar cualquiera del amplio conjunto de controles incluido por defecto (y también del no incluido), al desplegar en el Examinador de Objetos la jerarquía de clases a que pertenezca.

Componentes

Otro aspecto modificado es de la construcción de componentes. En .NET, como ya hemos comentado, se utiliza otra tecnología radicalmente distinta a ActiveX, lo que significa que ciertos tipos de componentes que podíamos crear mediante COM+, desaparecen, como los *ActiveX Documents*, y no existe una diferencia operativa práctica entre componentes y controles. La diferencia es solamente conceptual.

Estrictamente hablando, un componente es una clase que no posee un punto de entrada, y que se compila como DLL. No obstante, y al objeto de aportar alguna funcionalidad añadida propia de cada tipo de objeto, Visual Studio .NET establece algunas diferencias internas respecto a la herencia, para permitir que se obtenga funcionalidad predeterminada.

Así, cuando queremos construir un control, el esqueleto predeterminado contiene algunos elementos que merece la pena reseñar.

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Drawing;
using System.Data;
using System.Windows.Forms;

namespace ControlLibrary1
{
    /// <summary>
    /// Descripción breve de UserControl1.
    /// </summary>

    public class UserControl1 : System.Windows.Forms.UserControl
    {
        /// <summary>
        /// Variable del diseñador requerida.
        /// </summary>
        private System.ComponentModel.Container components = null;

        public UserControl1()
        {
            // Llamada requerida por el Diseñador de formularios
            InitializeComponent();

            // TODO: agregar cualquier inicialización después de llamar a
            // InitializeComponent

        }

        /// <summary>
        /// Limpiar los recursos que se estén utilizando.
        /// </summary>
        protected override void Dispose( bool disposing )
        {
            if( disposing )
            {
                if( components != null )
                    components.Dispose();
            }
            base.Dispose( disposing );
        }

        #region Component Designer generated code
```

```

        private void InitializeComponent()
        {
            components = new System.ComponentModel.Container();
        }
    #endregion
}

```

Código fuente 138

Observamos que la clase hereda de **System.Windows.Forms.UserControl**, lo que aporta al control de usuario la funcionalidad predeterminada que todo control debe tener, al menos, en tiempo de diseño. Un vistazo a dicha clase y sus elementos, nos aclara rápidamente en propósito de esta derivación.

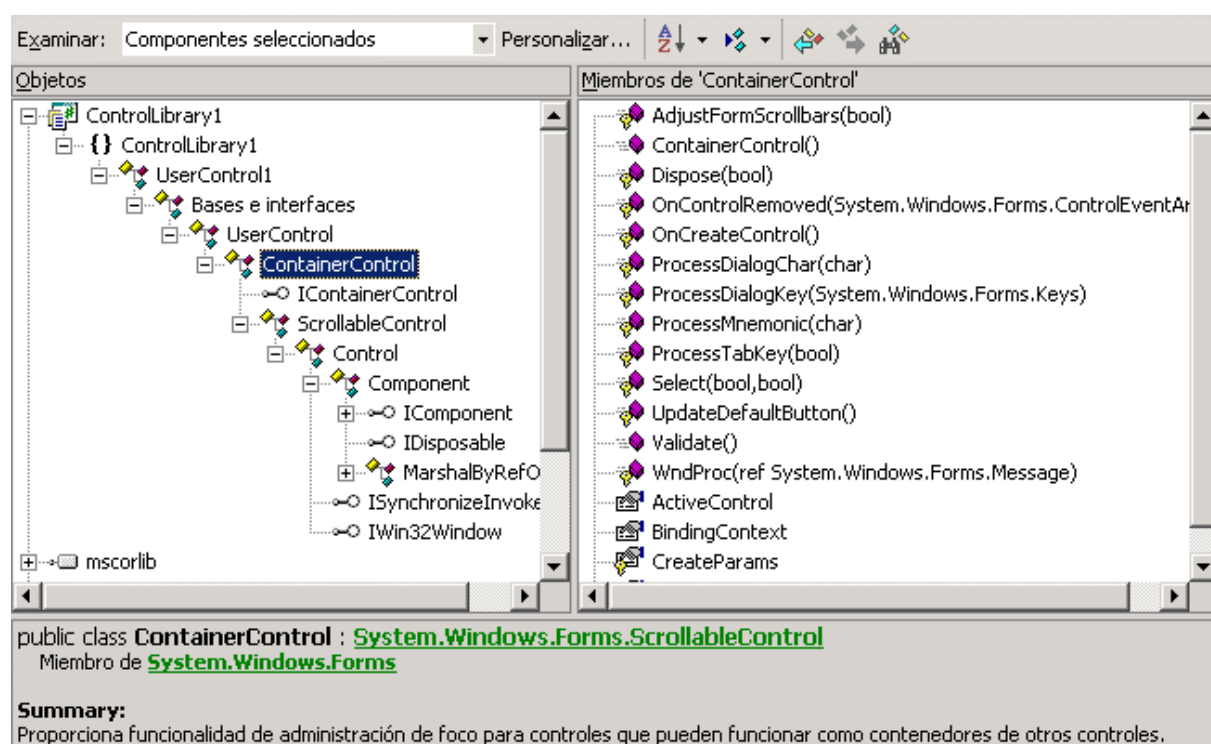


Figura 50. Secuencia de clases de los controles de usuario

Observando la jerarquía apreciamos que **UserControl**, deriva de **ContainerControl**, que dispone de una plétora de funcionalidad heredada, y éste, a su vez, de **ScrollableControl**, **Control** y **Component**.

Me gustaría recordar aquí que la libertad de programación que tenemos en .NET hace posible que comencemos nuestro *viaje programático* por cualquier rama de la jerarquía, siempre que sepamos que la funcionalidad que estamos ignorando, no la vamos a necesitar posteriormente: eso quiere decir que podemos construir un control que herede directamente de **ScrollableControl**, por ejemplo, siempre que las aportaciones que vienen por el lado de las clases derivadas de ella no sean necesarias.

En el caso de los componentes desatendidos (que no necesitan una interfaz de usuario), la libertad es incluso mayor, ya que se puede convertir cualquier clase en componente con tal de que lo establezcamos de tal forma en las propiedades del proyecto y, por tanto, compilemos dicho proyecto como una DLL y no como un ejecutable (y, no incluyamos un punto de entrada).

Un par de casos prácticos

Supongamos un componente realmente básico: tan simple como una clase que disponga de un método público (o propiedad pública), que devuelva algún valor. Según esto, el Código fuente 139 devuelve el día de la semana en la propiedad `DiaSemana`.

```
using System;

namespace ClassLibrary1
{
    public class ComponenteDiaSemana
    {
        public string DiaSemana
        {
            get { return System.DateTime.Today.DayOfWeek.ToString(); }
        }
    }
}
```

Código fuente 139

Dado que la forma de acceso al dato es de sólo lectura, y no se precisa ninguna otra comparación ni operación compleja, el usuario puede decidir si implementa la funcionalidad como propiedad o como función. Una vez compilada, la DLL se puede referenciar como cualquier otro componente, tal y como hemos visto en capítulos anteriores:

Supongamos que la librería se crea con el nombre **ClassLibrary1**. Una vez referenciada en la ventana estándar Agregar Referencias, pasa a formar parte de nuestro proyecto –como se ve en la Figura 51–, y sus miembros no se diferencian en posibilidades funcionales y de diseño de los de cualquier otro elemento de la propia jerarquía de clases de .NET Framework.

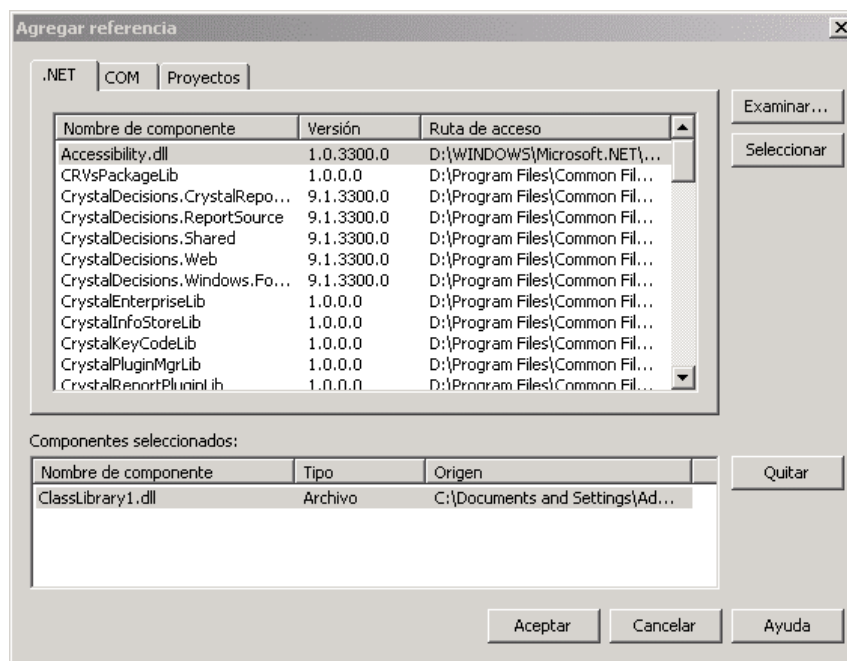


Figura 51. Selección de componentes de usuario en Visual Studio

Una vez hecho esto, podemos utilizarlo a partir de una instancia de la clase contenedora.

```
private ClassLibrary1.ComponenteDiaSemana disem = new
ClassLibrary1.ComponenteDiaSemana() ;
MessageBox.Show("Hoy es " + disem.DiaSemana());
```

Código fuente 140

La construcción de controles, sin embargo, supone una problemática añadida debida al proceso de diseño del control. Recordemos que los controles tienen dos formas de ejecución (una mientras se diseñan por el usuario, y otra en el propio tiempo de ejecución). No obstante, podemos heredar de forma visual de otros controles, y sencillamente, añadir, cambiar o quitar funcionalidad, utilizando las técnicas mencionadas.

Construcción de un sencillo control con herencia visual

Supongamos que deseamos construir nuestro propio control, que utilice todo lo que tiene una caja de texto, pero añadiendo una funcionalidad, quitando otra, y modificando una tercera. Lo primero, sería construir un proyecto Librería de Controles, y añadir el control a heredar a la superficie de diseño, donde asignamos algunas propiedades, tanto al contenedor (color de fondo, tipo de letra, etc., como a la propia caja de texto contenida, hasta obtener algún diseño diferente.



Figura 52. Diseño del control CajaHeredada

Además podemos hacer que el control interno (txtCajaInterna) cambie las dimensiones proporcionalmente con su contenedor, CajaHeredada. Basta que asignemos las propiedades **Anchor** a los cuatro lados del contenedor y su propiedad Multiline a **True** y ya tenemos la parte básica. Hasta aquí la parte de herencia visual. Se pueden hacer controles que agrupen conjuntos de controles, y como vamos a ver se pueden añadir propiedades muy fácilmente.

Si necesitamos algo más de control durante el diseño, tendremos que codificar el comportamiento. Por ejemplo, si la propuesta es que el control interno se mantenga siempre a una distancia proporcional al del contenedor, y además nos informe de sus dimensiones según vamos cambiando el tamaño, nos bastará con codificar el evento **Resize** del contenedor.

```
private void CajaHeredada_Resize(object sender, System.EventArgs e)
{
    txtCajaInterna.Height = this.Height - 12;
    txtCajaInterna.Text=txtCajaInterna.Width.ToString() + ", " +
txtCajaInterna.Height.ToString();
}
```

Código fuente 141

Utilizando otro proyecto añadido y volcando el control a un formulario, veremos automáticamente ese comportamiento.

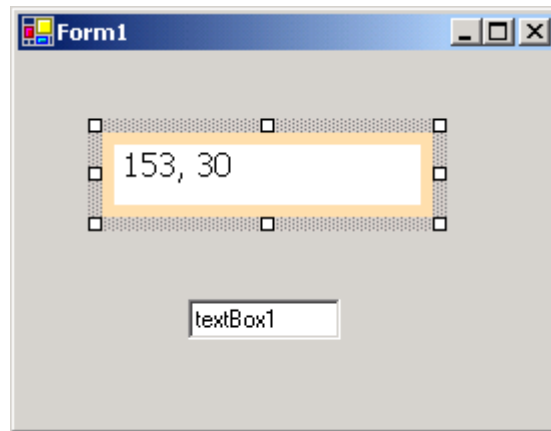


Figura 53. Comportamiento del control en tiempo de diseño

Imaginemos que nos interesa –además– que nuestro control pueda anular esta funcionalidad (la de mostrar los valores de tamaño, mediante la caja de propiedades del control, en tiempo de diseño. Nuestro control tendrá que exponer la funcionalidad, añadiendo alguna propiedad pública, como puede ser **TamañoEnDiseño**. Programaremos la propiedad modificando el código, como vemos en el Código fuente 142.

```
//Definición de la variable de estado TamañoEnDiseño
bool _TamañoEnDiseño = true;

//Cambiamos el código del evento Resize, para hacerlo depender del valor de la
//propiedad
private void CajaHeredada_Resize(object sender, System.EventArgs e)
{
    txtCajaInterna.Height = this.Height - 12;
    if (_TamañoEnDiseño == true)
        txtCajaInterna.Text = txtCajaInterna.Width.ToString() + ", " +
txtCajaInterna.Height.ToString();
    else
        txtCajaInterna.Text = "";
}
//Y codificamos el acceso a la propiedad
public bool TamañoEnDiseño
{
    get { return _TamañoEnDiseño; }
    set { _TamañoEnDiseño = value; }
}
```

Código fuente 142

Finalmente, si deseamos anular alguna funcionalidad, como por ejemplo, el comportamiento predeterminado de las cajas de texto al pulsar el botón derecho del ratón, no necesitamos utilizar subclasificación, sino que nos basta con codificar el evento correspondiente del objeto a controlar.

```
private void txtCajaInterna_MouseDown(object sender,
System.Windows.Forms.MouseEventArgs e)
{
    if (e.Button.ToString() == "Right")
        MessageBox.Show("Funcionalidad anulada");
}
```

Código fuente 143

Algunas consideraciones sobre el aspecto visual de los controles

Todo el dibujo de elementos visuales en Windows, es responsabilidad del API GDI (**Graphics Device Interface**), que permite utilizar los mismos mecanismos de programación para cualquier tarjeta gráfica, sin necesidad de conocer ningún detalle de la implementación ni de las características de hardware o software propias del dispositivo. Esta API, ha ido evolucionando, y a partir de la versión 2000 de Windows (y posteriores), utiliza una extensión de la anterior, denominada GDI+.

.NET ofrece un completo juego de posibilidades de alto nivel para permitir la fácil programación de elementos visuales, de forma que el usuario pueda utilizar la funcionalidad existente (la aportada por Windows) y añadir la que se desee. Vamos a realizar algunas pequeñas modificaciones a nuestro control para ilustrar estas posibilidades.

Cuando Windows tiene que dibujar utiliza un conjunto de elementos básicos para hacerlo que reciben nombres que se han conservado desde la versión Windows 95:

- **Contexto de Dispositivo (*Device Context*)**: es un área de la pantalla donde puede dibujarse. En .NET está representada por un objeto **Graphics**. Dicho objeto aparece como propiedad de la clase **EventArgs** en cada evento relacionado con el dibujo, aunque también puede crearse independientemente de cualquier evento.
- **Pen**: Se trata de la "herramienta" utilizada para dibujar. Podemos elegir el color, el grosor, si se trazan puntos o líneas sólidas, etc.
- **Brush**: Es una herramienta de relleno de superficies. La mayoría de las instrucciones de dibujo sólo trazan líneas huecas. Los **brushes** se utilizan para rellenar superficies. En .NET existe como clase abstracta, de la que puede derivarse, aunque también hay implementaciones concretas listas para usarse, como **SolidBrush**.
- **Point**: Ubicación absoluta o relativa a otro objeto en la pantalla, dada por un par de coordenadas X,Y.
- **Size**: Objeto que guarda valores relacionados con el tamaño de una superficie. Muchas funciones de dibujo la solicitan junto con los objetos puntos para ubicar el dibujo en pantalla.

Un caso práctico para aprovechar el trabajo anterior podría ser añadir una propiedad, que modifique el aspecto visual del control para que tenga sombra, al estilo de las cajas de texto de Access.

Para ello, podemos crear una propiedad Sombra3D (booleana), y hacer que cuando el usuario seleccione el valor True, el control txtCajaInterna modifique su posición hacia el origen de coordenadas del contenedor, y que el color del contenedor y su tamaño se modifiquen ligeramente para hacer los efectos de sombra.

El Código fuente 144 consigue los objetivos *primarios*.

```

//Definición de la variable de estado Sombra3D
bool _Sombra3D = false;

//Propiedad de acceso a _Sombra3D
public bool Sombra3D
{
    get { return _Sombra3D; }
    set
    {
        _Sombra3D = value;
        if (_Sombra3D == true) {
            txtCajaInterna.Location = new Point(0,0);
            this.Width= txtCajaInterna.Width + 5;
            this.Height= txtCajaInterna.Height + 3;
            txtCajaInterna.Size = new Size(this.Width-5, this.Height-5);
            //Crea un contexto gráfico para retocar la sombra
            Graphics Frag = this.CreateGraphics();
            Frag.DrawRectangle(new
Pen(Color.LightGray,4),0,txtCajaInterna.Height,5,5);
            Frag.DrawRectangle(new
Pen(Color.LightGray,4),txtCajaInterna.Width,0,5,5);
        }
    }
}

//Evento Resize modificado
private void CajaHeredada_Resize(object sender, System.EventArgs e)
{
    if (_Sombra3D == true)
        txtCajaInterna.Height = this.Height - 3;
    else
        txtCajaInterna.Height = this.Height - 12;
    if (_TamañoEnDiseño == true)
        txtCajaInterna.Text=txtCajaInterna.Width.ToString() + ", " +
txtCajaInterna.Height.ToString();
    else
        txtCajaInterna.Text = "";
}

```

Código fuente 144

Este código funciona de forma un tanto primaria y parcial. Utiliza un objeto gráfico para dibujar un pequeño rectángulo de forma que la sombra aparezca con los recortes propios de la proyección de un rectángulo que recibiese la luz del origen de coordenadas (punto 0,0 de la ventana contenedora).

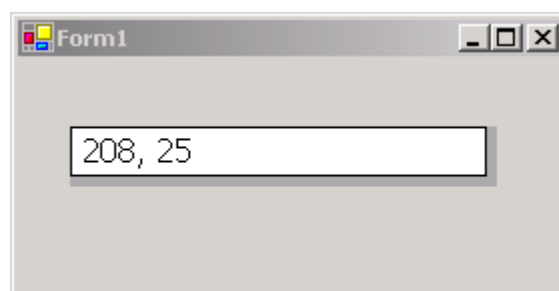


Figura 54. El control con sombra (defectuosa) en la ventana de diseño

El inconveniente es que –aun cuando al cambiar de tamaño al control, el comportamiento es correcto–, dicha funcionalidad no se soporta en otros eventos, y al ejecutarlo, se pierde la propiedad. Una

solución es la utilización del evento `Paint()`. En el Código fuente 144, la llamada al método `Invalidate()` del control, es una indicación para que Windows entienda que el control necesita repintarse. Si programamos la misma acción en el evento `Paint()` del control, dejando que `Resize()` se encargue solamente de asignar los valores de las coordenadas, podremos conseguir el efecto deseado.

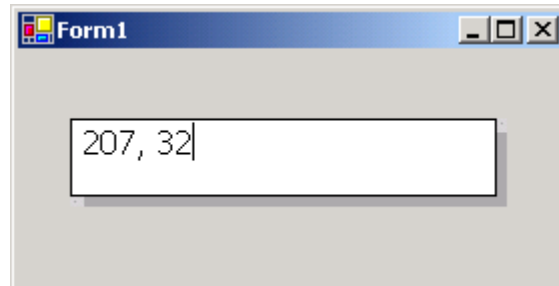


Figura 55. El control con sombra (corregida), en ejecución

Por lo demás, y siguiendo las mismas técnicas, podríamos permitir que el usuario seleccione el color de la sombra, su tamaño e intensidad, etc. Basta con tener en mente, (o comprobar), la secuencia de eventos que se produce, y elegir el lugar adecuado para el retoque gráfico.

(Aunque —obviamente— el control necesitaría algún que otro ajuste para su correcto funcionamiento, nos sirve a propósitos didácticos).

Algunas clases específicas para el tratamiento gráfico

Este ejemplo, integraba algunas capacidades gráficas básicas como el manejo de elementos básicos de dibujo. El conjunto completo de posibilidades va mucho más allá del ámbito de esta obra, y por si solo requeriría una obra separada. No obstante, para dar una idea de algunas de estas capacidades, vamos a incluir aquí algunos ejemplos, de tratamiento de gráficos, como el uso de gradientes, figuras irregulares, etc.

Uso de Gradientes

Dentro del espacio calificado ***System.Drawing***, encontramos otros subespacios muy interesantes para el tratamiento gráfico, como ***System.Drawing.Drawing2D***. Encontramos aquí clases como *LinearGradientBrush*, *Blend*, *GraphicsPath*, etc.

Por ejemplo, el sencillo código que aparece en el Código fuente 145, dibuja un curioso degradado en el formulario a partir de 2 colores seleccionados y una dirección de degradación.

```
LinearGradientBrush linGrBrush = new LinearGradientBrush(new
    Rectangle(100,100,100,100),Color.AntiqueWhite,
    Color.BlueViolet,LinearGradientMode.ForwardDiagonal );
e.Graphics.FillRectangle(linGrBrush, 0, 0, 300, 300);
```

Código fuente 145

La salida en pantalla (no se aprecian aquí más que grados de gris), sería como muestra la Figura 56.

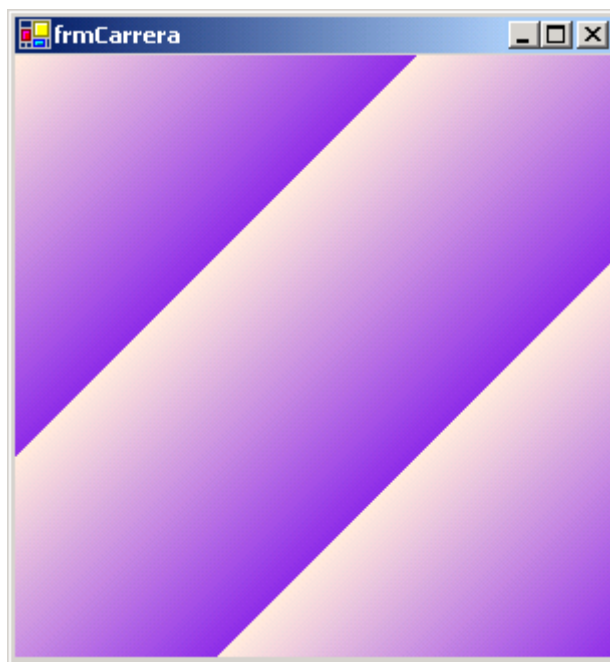


Figura 56. Ejemplo de uso de gradientes.

Dibujando Cadenas

Otro tanto sucede con las cadenas. Los objetos **Font** deben considerarse como tipos especiales de gráficos. En .NET podemos crear objetos Font a partir del método constructor de la clase correspondiente, y a partir de ahí utilizarlos para escribir, con el color deseado en unas coordenadas de pantalla.

```
Font Fuente = new Font("Tahoma",16);  
e.Graphics.DrawString("Bienvenido al mundo de los gráficos GDI", Fuente, new  
SolidBrush(Color.Navy), 20,50);
```

Código fuente 146

Un ejemplo completo mediante un visor de gráficos

Para concluir este capítulo dedicado a la programación en Windows, vamos a incluir un ejemplo completo consistente en un formulario Visor de Gráficos. Mediante 3 de los controles más comunes (**ComboBox**, **TreeView** y **ListBox**), diseñaremos un mecanismo para recorrer unidades y directorios, y una vez que visualicemos ficheros de gráficos, los mostraremos en un cuarto control (**PictureBox**), al seleccionar cada fichero. Esto ilustrará también el uso de algunos de los controles más utilizados en el entorno Windows.

Así pues, el diseño del formulario tendrá el aspecto que muestra la Figura 57.

El objetivo es —en primer lugar— asignar la información a visualizar al contenido de los controles (El *ComboBox*, la información de Unidades de Almacenamiento, el *TreeView*, la lista de directorios y el *ListBox*, la lista de ficheros).

A continuación deberemos sincronizar los tres controles, y finalmente, programar el evento del control *ListBox*, para que al seleccionar un fichero gráfico, se muestre en el control *PictureBox*

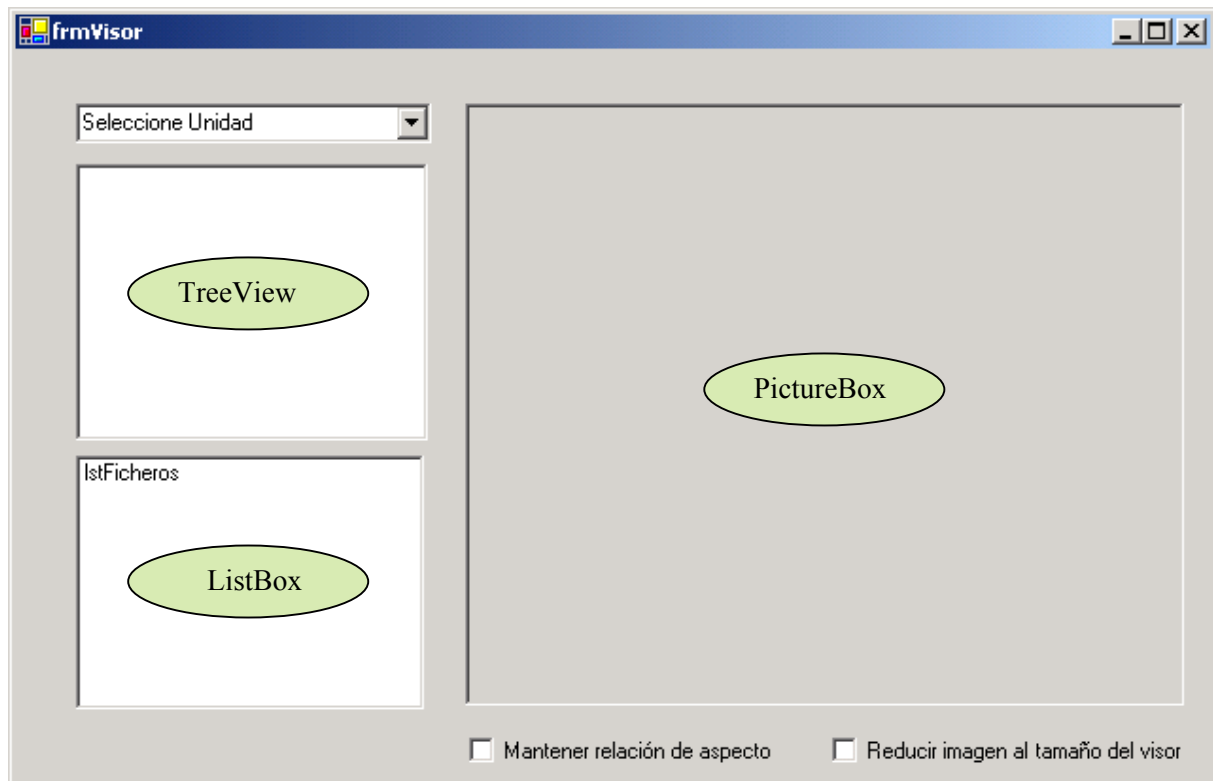


Figura 57. Diseño del visor de gráficos

Vayamos por partes: lo primero, el ComboBox, debe obtener la lista de unidades disponibles:

```
string[] Unidades = System.Environment.GetLogicalDrives();
foreach (string u in Unidades)
{
    cboDrives.Items.Add(u.ToUpper());
}
```

Código fuente 147

A continuación, deberemos programar el evento `SelectedIndexChanged` del Combo `cboDrives`, para obtener la información de directorios para esa unidad:

```
private void cboDrives_SelectedIndexChanged(object sender, System.EventArgs e)
{
    try
    {
        System.IO.DirectoryInfo tDirInfo = new
        System.IO.DirectoryInfo(cboDrives.Text);
        DirectoryInfo[] Directorios = tDirInfo.GetDirectories();
        //Vaciamos el TreeView para presentar únicamente la información
        // de un único dispositivo
        tvDirs.Nodes.Clear();
        foreach (DirectoryInfo di in Directorios)
        {
            TreeNode nodo = new TreeNode(di.FullName, 3, 4);
            tvDirs.Nodes.Add (nodo);
        }
    }
}
```

```

    }
    catch
    {
        MessageBox.Show("El dispositivo no está listo");
    }
}

```

Código fuente 148

El paso siguiente, es la programación del control *TreeView tvDirs*. Debemos responder a las selecciones del usuario, y cuando la selección contenga nuevos directorios, añadirlos al árbol seleccionando los iconos de visualización y el resto de elementos visuales, incluido un filtro para mostrar únicamente ficheros gráficos.

```

private void tvDirs_AfterSelect(object sender,
System.Windows.Forms.TreeViewEventArgs e)
{
    DirectoryInfo tDirInfo = new DirectoryInfo(e.Node.FullPath);
    DirectoryInfo[] Directorios = tDirInfo.GetDirectories();

    foreach (DirectoryInfo di in Directorios)
    {
        TreeNode nodo = new TreeNode(di.Name, 3, 4);
        e.Node.Nodes.Add(nodo.Text);
    }

    FileInfo [] ficheros = tDirInfo.GetFiles();
    lstFicheros.Items.Clear();
    foreach( System.IO.FileInfo fic in ficheros)
    {
        if(fic.Extension.ToUpper() == ".BMP" || fic.Extension.ToUpper()
== ".JPG" || fic.Extension.ToUpper() == ".GIF" || fic.Extension.ToUpper() ==
".ICO" || fic.Extension.ToUpper() == ".PNG" || fic.Extension.ToUpper() == ".WMF" ||
fic.Extension.ToUpper() == ".JPEG" || fic.Extension.ToUpper() == ".TIFF" ||
fic.Extension.ToUpper() == ".TIF" || fic.Extension.ToUpper() == ".CUR" )
        {
            lstFicheros.Items.Add(fic.Name);
        }
    }
}

```

Código fuente 149

Finalmente, viene el proceso de selección de ficheros y su interpretación gráfica. Para ello, cuando el usuario seleccione un gráfico, invocaremos el evento *Invalidate()* del control *PictureBox*, con el objeto de que Windows, provoque un repintado del control, a través del evento *Paint()* del *PictureBox*. Dentro de *Paint()*, si el usuario ha seleccionado un nodo válido, creamos un objeto *Bitmap* a partir de la información del fichero y dibujamos ese *bitmap* en el control mediante el objeto *Graphics* proporcionado por el evento *Paint()*.

```

private void lstFicheros_SelectedValueChanged(object sender, System.EventArgs e)
{
    //El método Invalidate() informa al CLR que la (imagen gráfica actual
    //correspondiente a un objeto ya no es válida, (rendering) con lo que
    //se fuerza la llamada al evento Paint() de ese control
    pictureBox1.Invalidate();
}

```

```
private void pictureBox1_Paint(object sender, System.Windows.Forms.PaintEventArgs e)
{
    if( tvDirs.SelectedNode != null)
    {
        try {
            Bitmap b = new Bitmap(tvDirs.SelectedNode.FullPath + @"\" +
            lstFicheros.SelectedItem.ToString());
            Graphics g = e.Graphics;
            if (chkComprimir.Checked == false)
            {
                g.DrawImage(b,0,0, b.Width,b.Height);
            }
            else
            {
                g.DrawImage(b,pictureBox1.ClientRectangle);
            }
        }
        catch { MessageBox.Show("Error al mostrar el gráfico"); }
    }
}
```

Código fuente 150

Hemos añadido un tratamiento básico de errores, práctica que es recomendable siempre que se produzca un acceso a recursos de este tipo en el sistema (directorios, ficheros de gráficos, unidades, etc.).

La salida final ofrece un buen rendimiento y el tratamiento posterior de la imagen, como –por ejemplo- comprimirlo a la superficie del control que lo muestra, etc, son operaciones que pueden realizarse sencillamente, utilizando ese polivalente objeto Graphics del que esta última parte del capítulo ha servido de introducción.

La salida final en ejecución, tiene el aspecto que muestra la Figura 58.

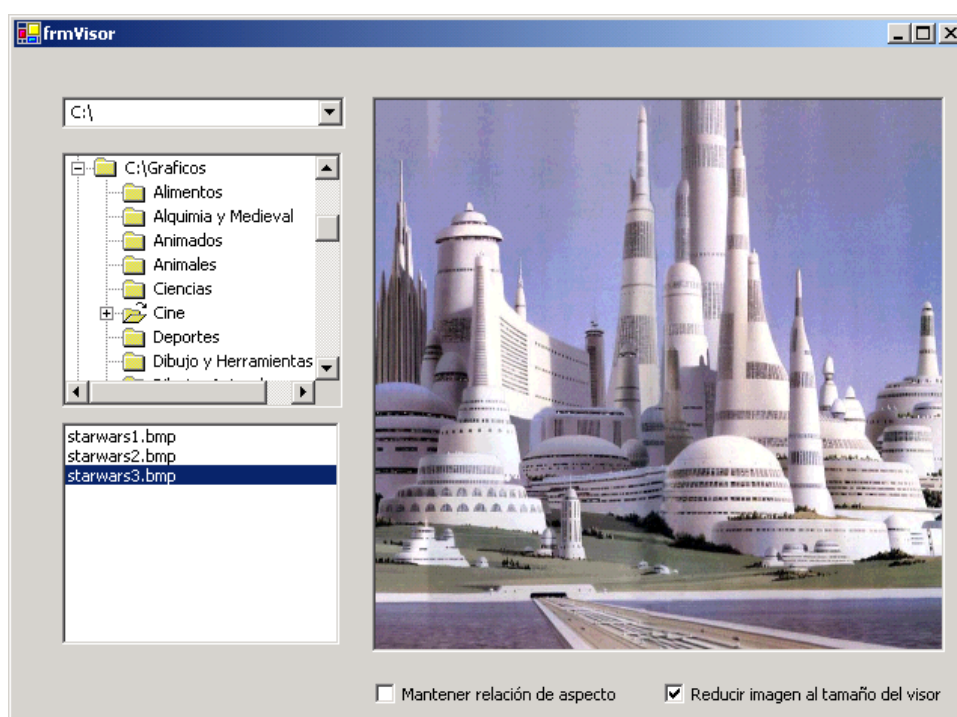


Figura 58. Visor de gráficos en funcionamiento con la opción de reducir imagen al tamaño del contenedor.

Programación con ASP .NET

Introducción²¹

En .NET, la programación para Internet, se ha unificado, en cierto sentido, respecto a la programación Windows, no existiendo esa separación que aparecía –incluso a nivel de herramientas- entre una y otra. No existe, por lo tanto una versión de Visual Interdev para la programación con el nuevo modelo ASP.NET, sino que se convierte en un tipo más de proyecto dentro de Visual Studio.

En concreto, existen 3 tipos de aplicaciones específicas que podemos crear con Visual Studio .NET, en relación con la Web: **Aplicaciones ASP.NET** estándar, **Servicios Web XML**, y **Controles Web**.

Este capítulo pretende ofrecer una visión básica de la tecnología ASP.NET de forma que resulte suficientemente ilustrativa para el lector que se quiera adentrar en su programación utilizando el lenguaje C#. Asumimos que el lector conoce alguna de las versiones anteriores de ASP, al menos en lo concerniente a nociones básicas. Dejaremos, no obstante, la parte dedicada a Servicios Web para un capítulo aparte, pues, aunque se trate de tecnología asociada a Internet, no se puede considerar que ese sea su único ámbito, por una parte, y las tecnologías implicadas son –en principio- algo distintas, y basadas en estándares propios sobre gramáticas XML.

²¹ Todo este capítulo es una versión muy resumida, adaptada y corregida para éste de la obra de **Angel Esteban** "*Programación con ASP.NET*", editada también por Grupo ÉIDOS,. El lector puede encontrarlo en las mismas fuentes que esta obra.

El mecanismo de funcionamiento de ASP.NET

Aunque hemos indicado que existen tres tipos de aplicaciones para la Web, en realidad, la tecnología subyacente es la misma en los tres casos: ASP.NET, si bien la utilizamos para generar una funcionalidad diferente.

Algunas de las características principales de ASP.NET son:

- Cada página se compila a código intermedio para su posterior ejecución.
- La compilación de las páginas sólo se produce en la primera petición que se realiza sobre la página, o bien cuando ha cambiado el código fuente y necesite actualizarse.
- El resultado de esta compilación permanece en caché para poder ser reutilizado.
- La gestión de los eventos de cliente se produce, de forma predeterminada, en el servidor
- Las librerías de ASP.NET son capaces de discriminar el tipo de cliente que realiza la petición, y enviar como respuesta el código HTML que, ésa versión, de ése navegador, es capaz de comprender.
- El código funcional (o código operativo) se puede escribir en cualquier lenguaje de los que soporta la plataforma, y puede situarse en un fichero separado del código HTML de presentación.

El fundamento de ASP.NET

La filosofía de ASP .NET es simple: las páginas ASP .NET, ó páginas activas, contienen código HTML, *script* de cliente y un código de sintaxis propia que se ejecuta en el servidor, dando como resultado código HTML. Al cargar una página ASP .NET en un navegador, en realidad estamos cargando el resultado de la ejecución de la página, es decir, el resultado que una maquinaria de interpretación produce como código HTML.

El cambio que aporta ASP .NET es sobre todo en la forma de desarrollar aplicaciones Web, es decir, la forma en la que vamos a utilizar los distintos componentes y servicios que nos ofrece ASP .NET, y también la forma de programar. Desde ASP .NET podemos utilizar acceso a datos ADO.NET, estudiado en el capítulo anterior, de la misma forma a como hacíamos en Windows.

ASP .NET es completamente compatible con ASP 3.0, y podemos tener aplicaciones Web basadas en ASP y ASP .NET funcionando en el mismo servidor Web, que será Internet Information Server 5.0

Diferencias fundamentales entre ASP .NET y ASP 3.0

Vamos a ver algunas de las más notables diferencias respecto a las versiones anteriores de ASP, y especialmente de la versión 3.0.

- **Código compilado:** Esta es la primera diferencia que encontramos entre las páginas ASP y las páginas ASP .NET, las páginas ASP eran interpretadas línea a línea por la DLL ASP.DLL, pero ahora las páginas ASP .NET cuando reciben la primera petición se compilan automáticamente al lenguaje intermedio. Esta característica consigue grandes ventajas en lo que a rendimiento en tiempo de ejecución se refiere, ya que la compilación de las páginas sólo

se produce en la primera petición que se realiza sobre la página, o bien cuando el código fuente de la misma se ha modificado y necesita por lo tanto actualizarse, además el resultado de esta compilación permanece en caché para poder ser reutilizada. El código intermedio recuerda en cierta forma al *bytecode* generado en el lenguaje Java.

- **Bloques de código:** Otro aspecto diferente de ASP.NET, es que los delimitadores de *script* de servidor, es decir los delimitadores `<% %>` han pasado a utilizarse en un segundo plano, siendo más común la utilización de los bloques `<script></script>`. Los delimitadores `<% %>` se puede seguir utilizando, pero no se podrán utilizar a la hora de definir un procedimiento a una función dentro de una página, para esa labor debemos utilizar los bloques `<script language="C#" runat="server"></script>`. Por lo tanto, se dispone de tres delimitadores de código de servidor: los delimitadores `<script></script>` para declarar métodos, los delimitadores `<% %>` para ejecutar sentencias y los delimitadores `<%= %>` para mostrar el resultado de una expresión.
- **Directivas:** En versiones anteriores de ASP solo podíamos utilizar una única directiva en la página, por ejemplo para indicar el lenguaje de programación utilizado. Sin embargo, en ASP .NET podemos utilizar varias sentencias con directivas en la página. Además ASP introduce nuevas directivas, como **Page**, **Control**, **Import**, **Register**, **Assembly**, **OutputCache** y **WebService** (a continuación vamos a pasar a comentar brevemente cada una de estas directivas).
- Se abandona el método **Server.CreateObject()** para la creación de objetos, debiendo declararse el objeto del tipo específico y llamar a su constructor, de forma que la sentencia

Set obj=Server.CreateObjec("progID") adoptaría en C# la forma:

progID obj = new progID();

- Ya no es necesario obtener una referencia al objeto **ObjectContext**, que se introducía en ASP 2.0, sino que ASP .NET ofrece el nuevo objeto **Context** que permite acceder al entorno de ejecución de las páginas ASP .NET.
- El fichero GLOBAL.ASA a pasado a denominarse GLOBAL.ASAX y permite capturar nuevos eventos como pueden ser: `Application_Error`, `Application_Begin_Request`, `Application_AuthorizeRequest`, etc., hasta un total de 16 eventos frente a los 4 de las versiones anteriores de ASP.
- La configuración de una aplicación ASP .NET no se recomienda realizarla a través de la consola de administración del servidor Web Internet Information Server, sino a través de un fichero especial de configuración en formato XML denominado CONFIG.WEB. Para los lectores conocedores de JSP (Java Server Pages), la finalidad de este fichero es muy similar al fichero de configuración WEB.XML de la tecnología de las páginas de servidor de Java.

Revisión básica de las directivas ASP.NET

Page: La directiva mas compleja, y también la más utilizada es la directiva **Page**. Define una serie de atributos específicos para cada página que serán utilizados por el compilador a la hora de generar el código intermedio (IL). Soporta todos los atributos existentes para la única directiva que se podía utilizar en ASP, es decir, ofrece los atributos `Language`, `Transaction`, etc. La directiva `Page` es la directiva por defecto, así por ejemplo la directiva `<@Language="VBScript"%>` de ASP 3.0, tendría el equivalente `<@ Page Language="VB">` en ASP .NET. Otros atributos o propiedades de la directiva son: `Trace`, `Inherits`, `Src`, etc.

Control: La directiva **Control**, se utiliza para definir atributos específicos de controles definidos por el usuario.

Import: permite importar un espacio con nombre (*namespace*) en la página actual. Para los lectores que conozcan Java, podemos decir que se trata de un mecanismo similar al de los paquetes, para los cuales Java utiliza la sentencia *import*..

Register: La nueva directiva **Register** asocia un alias con un *namespace*, este alias será el prefijo que se utilice en las etiquetas que hagan uso del namespace correspondiente.

Assembly: La directiva **Assembly** nos permite establecer una referencia dentro de una página a un ensamblado, pudiendo utilizar plenamente su contenido.

OutputCache permite especificar la política del caché de salida de la página, pudiendo indicar el número de segundos que el resultado de la ejecución de una página se va a mantener en caché.

Finalmente, la directiva **WebService**, indica que el código fuente de referencia está concebido para ser usado como servicio Web.

Orientación a objetos

ASP .NET se encuentra dentro de un modelo de programación orientado a objetos. La plataforma .NET se basa en un conjunto bastante amplio de clases organizadas mediante los espacios con nombre (NameSpaces), y ASP .NET al estar incluido dentro de la plataforma .NET utiliza también este conjunto de clases.

En ASP .NET cada página es un objeto, (una instancia de la clase **Page**), con sus atributos, tales como **Request** (objeto de la clase **HttpRequest** que representa una petición que se ha realizado sobre una página ASP .NET) y lo mismo ocurre con el miembro **Response**, que es una instancia de la clase **HttpResponse** que representa la salida que la página envía al navegador Web del usuario. Ambas clases se encuentran dentro del espacio con nombre System.Web. Es decir, en ASP .NET los objetos intrínsecos se obtienen a través de miembros de la clase Page. La orientación a objetos también se hace patente a la hora de utilizar los controles de servidor, ya que se utilizan las mismas técnicas en su construcción.

Aspectos sintácticos

Ya hemos indicado que los delimitadores para indicar el código de servidor en ASP .NET son tres: `<% %>` que indica código de script de servidor, y sirve tanto en la declaración de directivas, como en sentencias (por ejemplo, las asignaciones, como `<% lagarto="Juancho"%>` para asignar el valor *Juancho* a la variable *lagarto*).

El segundo tipo de delimitadores `<%= %>` permite expresiones que devuelve algún valor para ser mostrado en la salida de la página, al estilo de cómo se utilizaba en ASP 3.0. Así, por ejemplo la expresión `<%= lagarto %>` devolverá al navegador el valor *Juancho*. Estos delimitadores solo admiten una sentencia por línea. El tercer tipo de delimitador es una etiqueta como la que vemos en el Código fuente 151.

```
<script language="C#" runat="server"></script>
```

Código fuente 151

Esta etiqueta es utilizada únicamente para declarar métodos (procedimientos y funciones) de la página, y el delimitador `<%%>` sólo para escribir código que se ejecute dentro de la página fuera de procedimientos y funciones, es decir, el código que se ejecuta en línea.

Veamos unos fragmentos de código en los que se utilizan cada uno de los delimitadores comentados: En el Código fuente 152 se puede ver la utilización de los delimitadores `<%%>` para definir un objeto de la clase `DateTime` con la fecha y hora actual. Y también se utilizan los delimitadores `<%=%=>` para mostrar la fecha y hora actual en el navegador del usuario.

```
<%DateTime ahora=DateTime.Now;%>
<%=ahora.ToString()%>
```

Código fuente 152

Dentro de los delimitadores de código de servidor se pueden encontrar también instrucciones del lenguaje de correspondiente, así por ejemplo puede aparecer una instrucción `if...else` de C# como puede verse en el Código fuente 153.

```
<%String nombre="", variable="";
if(nombre==""){
    variable="Nombre desconocido";
}else{
    variable="Hola amigo "+nombre;
}%>
<font color="green"><%=variable%></font>
```

Código fuente 153

Se puede incluir código HTML entre las instrucciones del lenguaje de servidor, aunque no es recomendable de cara a una lectura sencilla del código. Un ejemplo equivalente al código anterior lo podemos ver en el Código fuente 154.

```
<font color="green">
<%String nombre="";
if(nombre==""){%>
    Nombre desconocido
<%}else{%>
    Hola amigo <%=nombre%>
<%}%>
</font>
```

Código fuente 154

También se recomienda utilizar los delimitadores del código de servidor encerrando varias líneas de código en lugar de un par de delimitadores por cada línea. Así, en lugar de escribir el código que se muestra en el Código fuente 155, es recomendable utilizar el Código fuente 156.

```
<%String strNombre="";%>
<%Session.Add("nombre","Pórtamos");%>
```

```
<%strNombre=Session["nombre"].ToString();%>
<%Response.Write(strNombre);%>
```

Código fuente 155

```
<%String strNombre="";
Session.Add("nombre", "Pórtamos");
strNombre=Session["nombre"].ToString();
Response.Write(strNombre);%>
```

Código fuente 156

En el caso de tener línea simple de script, los delimitadores se deben encontrar en la misma línea ().

```
<%strNombre=Session["nombre"].ToString();%>
```

Código fuente 157

Si tenemos que definir un método (procedimiento o función) utilizaremos las etiquetas `<script>` como se puede ver en el Código fuente 158.

```
<script language="c#" runat="server">
void Pulsado(object sender, EventArgs args){
    etiqueta.Text="¡Hola Mundo!";
}
</script>
```

Código fuente 158

Como todo .NET, la sintaxis de ASP .NET también está muy relacionada con la sintaxis XML. Eso sucede con los controles de servidor, que se instancian utilizando una sintaxis basada en XML. En el fragmento del Código fuente 159, se puede ver la sintaxis utilizada para instanciar dos controles Web de servidor; en este caso se crea una etiqueta y un botón, dentro de un formulario Web.

```
<form id="formulario" method="post" runat="server">
    <asp:label id="etiqueta" runat="Server"></asp:label>
    <asp:button id="boton" onclick="Pulsado" runat="server"
        text="Pulsa"></asp:button>
</form>
```

Código fuente 159

El lector observará que los controles ASP.NET llevan en su etiqueta el prefijo asp:. Esto es debido a que en XML, esa es la forma de declarar un elemento como perteneciente a un espacio de nombres asociado a un intérprete. En este caso, el espacio de nombres indica que el intérprete asociado de esos controles es el correspondiente a ASP.NET.

Sin embargo la sintaxis utilizada para crear controles HTML de servidor desde las páginas ASP .NET es distinta. En este caso se utiliza una sintaxis prácticamente idéntica a la del lenguaje HTML.

```
<form id="controlesHTML" method="post" runat="server">
<input type="button" id="boton" runat="server" value="Pulsar"
onserverclick="Pulsado">
<div id="etiqueta" runat="server"></div>
</form>
```

Código fuente 160

Según vayamos avanzado en el texto veremos otros aspectos sintacticos en más detalle. A continuación vamos a revisar lo fundamental de los formularios Web (Web Forms).

Web Forms

Los formularios Web son una nueva característica de ASP .NET que ofrece un modelo de programación para generar contenidos dinámicos de una forma más sencilla. Los Web Forms vienen a sustituir e integrar los formularios HTML dentro del entorno de programación de ASP .NET. Un Web Form en última instancia va a generar el código HTML correspondiente al formulario que estamos diseñando.

Nota: La idea es –cada vez más– que el programador de la web se preocupe más por la lógica asociada a los objetos programables, y menos por las características puntuales que se deriven de un navegador o lenguaje de implementación concretos.

Una de las grandes ventajas que ofrecen los Web Forms es que el modelo de programación es muy similar al de los tradicionales formularios de Visual Basic –o Delphi–, incluso podemos realizar el tratamiento de eventos del cliente desde código del servidor.

Ventajas que nos ofrecen los Web Forms

- Reutilización de los controles de interfaz de usuario (UI) que encapsulan su funcionalidad, reduciendo de esta forma el código que el desarrollador debe escribir.
- Código más limpio y claro en la página, ya que no aparece mezclado con código HTML como ocurría en las versiones anteriores de ASP, en las que era bastante usual intercalar código HTML con código ASP, y, aunque en ASP.NET esto también se permite por compatibilidad, si nos ceñimos al modelo de programación que ofrecen los Web Forms no es necesario.
- Toda la interfaz del Web Form puede diseñarse a través de herramientas o entornos visuales de desarrollo. Visual Studio .NET nos ofrece una vista de diseño para los Web Forms, que incluso permite diseñar los Web Forms situando sus controles según las posiciones x e y de la pantalla.
- Integra el tratamiento de eventos del cliente con el código correspondiente del servidor. El tratamiento de eventos es muy similar al que ya se utilizaba en los formularios de las anteriores versiones de Visual Basic.
- Realizan un mantenimiento del estado del formulario (Web Form) de forma automática entre distintas peticiones, es decir, si seleccionamos una serie de valores en el Web Form, al

enviarlo de forma automática se van a mantener estos valores en el caso de que se vuelva a recargar el formulario.

- Ofrece una mayor separación entre la lógica de la aplicación y la presentación de la misma.
- Permite utilizar un gran conjunto de controles de servidor que generarán el código HTML necesario para mostrar el Web Form en la página cargada por el cliente en su navegador. Después veremos que dentro de los Web Forms podemos encontrar varios tipos de controles.

A continuación se ofrece un código muy sencillo, que muestra la utilización de un *Web Form* dentro de una página ASP .NET. El ejemplo consiste en un Web Form que ofrece una lista de destinos en una lista desplegable con sus precios, y al pulsar el botón del formulario se indica en una etiqueta el nombre y precio del destino seleccionado.

```
<html><head>
<script language="C#" runat="server">
void Pulsado(Object sender, EventArgs args){
etiqueta.Text="<br>El precio para el destino " + lista.SelectedItem.Text +
" es de " + lista.SelectedItem.Value + " euros";
}
</script>
</head>
<body>
<form id="WebForm" method="post" runat="server">
<asp:DropDownList id="lista" runat="server">
<asp:listitem value="1000" text="Isla de Pascua"></asp:listitem>
<asp:listitem value="900" text="Morea"></asp:listitem>
<asp:listitem value="750" text="Papeete"></asp:listitem>
<asp:listitem value="850" text="Bora-Bora"></asp:listitem>
</asp:DropDownList>
<asp:Button id="boton" runat="server" Text="Ver precio"
onclick="Pulsado"></asp:Button>
<asp:Label id="etiqueta" runat="server"></asp:Label>
</form>
</body></html>
```

Código fuente 161

En la Figura 59 se puede ver un ejemplo de ejecución de esta página ASP .NET.

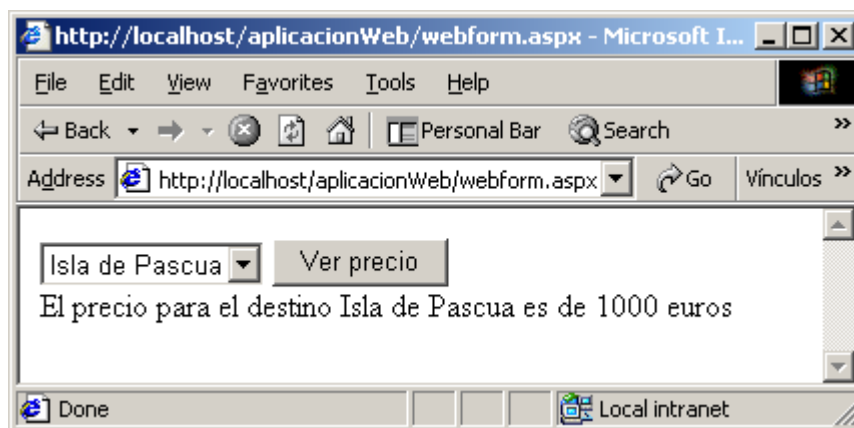


Figura 59

A la vista del código del ejemplo merece la pena comentar que los controles ASP .NET del Web Form tienen una notación en formato XML, a excepción de la etiqueta que se corresponde con el propio formulario, y como se puede comprobar todos ellos poseen el atributo **runat** con el valor **server**, para indicar que son controles de servidor.

Si lanzamos la página se puede ver que, entre distintas peticiones, se va conservando el valor que se seleccione en la lista de destinos, sin escribir ningún código específico para realizar este comportamiento (que sería necesario en versiones anteriores de ASP). El propio Web Form realiza esta labor por nosotros generando el código HTML necesario, que consiste en un campo oculto llamado **__VIEWSTATE**. El código HTML que **genera** la página del ejemplo se puede ver en el Código fuente 162.

```
<html><head>

</head>
<body>
<form name="WebForm" method="post" action="webform.aspx" id="WebForm">
<input type="hidden" name="__VIEWSTATE"
value="dDwtMTY2NzM0NjY1Njtp0PDtsPGk8Mj47PjtsPHQ8O2w8aTw1Pjs+O2w8dDxwPHA8bDxUZXh0Oz47
bDxcPGJyXD5FbCBwcmVjaW8gcGFyYSBlbCBkZXN0aW5vIElzbGEgZGUgUGFzY3VhIGVzIGRlIDewMDAgZXV
yb3M7Pj47Pjs7Pjs+Pjs+Pjs+" />

<select name="lista" id="lista">
    <option selected="selected" value="1000">Isla de Pascua</option>
    <option value="900">Morea</option>
    <option value="750">Papeete</option>
    <option value="850">Bora-Bora</option>
</select>
<input type="submit" name="boton" value="Ver precio" id="boton" />
<span id="etiqueta"><br>El precio para el destino Isla de Pascua es de 1000
euros</span>
</form>
</body></html>
```

Código fuente 162

Otro punto importante es que, como se puede comprobar, no hemos utilizado la colección Form del objeto Request para obtener los valores de los elementos enviados en el formulario, sino que simplemente hemos accedido a las propiedades de los controles a través del nombre con el que los hemos declarado, es decir, al estilo de los formularios de Visual Basic.

Esto es posible gracias a que cada control del formulario es un objeto (objetos de la página ASP .NET), es decir una instancia de una clase determinada, y por lo tanto cada uno de ellos poseerá sus propiedades y métodos. Así por ejemplo la lista desplegable que hemos indicado en el Web Form se corresponde con un objeto de la clase DropDownList que se encuentra dentro del espacio con nombre (Namespace) System.Web.UI.WebControls presente en el .NET Framework. Lo mismo ocurre con el objeto boton que es de la clase Button y el objeto etiqueta que es una instancia de la clase Label. Existe un gran número de controles de servidor y cada uno de ellos pertenece a una clase del .NET Framework. La filosofía es, por tanto, totalmente coherente con lo utilizado hasta aquí para otros entornos.

Veamos algunos de los controles de servidor que nos ofrece ASP .NET.

Controles ASP .NET

Directamente relacionados con los Web Forms tenemos los distintos controles de servidor que se puede utilizar dentro de los Web Forms y en las páginas ASP .NET en general. Todos estos controles generarán el correspondiente código HTML para que el usuario pueda utilizarlos en la página cargada en su navegador. Los numerosos controles de servidor incluidos en ASP .NET se pueden agrupar en cuatro categorías o familias:

- Controles **intrínsecos**, que a su vez se subdividen en dos grupos, y que representan elementos HTML.
- Controles de **lista**, utilizados para distribuir y mostrar datos en una página.
- Controles **ricos**, ofrecen funcionalidades avanzadas de interfaz de usuario.
- Controles de **validación**, ofrecen distintos tipos de validación de entrada de datos.

Controles intrínsecos

Son los que hemos utilizado en el ejemplo del apartado anterior. Se encargan de generar elementos HTML en el cliente, y cada uno de ellos tiene su etiqueta HTML correspondiente, como puede verse al comprobar el código generado. Dentro de este grupo de controles podemos distinguir dos tipos, los llamados controles HTML y los controles Web.

Controles HTML

Los controles HTML son muy similares a las etiquetas HTML que representan y sus propiedades son casi idénticas. Se diferencian de las etiquetas HTML en que su disponen de una propiedad **runat** con el valor "server".

En el Código fuente 163 de ejemplo se muestra una página que posee un Web Form y dos controles HTML, un botón (*HtmlButton*) y una etiqueta (*HtmlGenericControl*) que contiene un texto. Al pulsar el botón aparecerá la fecha y hora actual en la etiqueta. Estos controles se encuentran dentro del espacio con nombre **System.Web.UI.HtmlControls**.

```
<html>
<body>
<script language="C#" runat="server">
void Pulsado(Object sender, EventArgs args){
    DateTime ahora;
    ahora = DateTime.Now;
    etiqueta.InnerText = "El momento actual es: " + ahora.ToString();
}
</script>
<form id="controlesHTML" method="post" runat="server">
<input type="button" id="boton" runat="server" value="Pulsar"
        onclick="Pulsado" NAME="boton">
<div id="etiqueta" runat="server"></div>
</form>
</body>
</html>
```

Código fuente 163

Estos controles permiten mantener el estado de forma automática entre distintas llamadas a una página. Para acceder al control –desde dentro de la página- éste debe identificarse mediante el atributo **ID** de la etiqueta HTML correspondiente.

Estos controles se ofrecen por compatibilidad con versiones anteriores de ASP, así que si queremos que una etiqueta HTML pase a ser un control HTML simplemente debemos indicar la propiedad **runat** con el valor "*server*", que es la que normalmente se utiliza en el otro tipo de controles intrínsecos, los controles Web.

Controles Web

Estos controles ASP .NET cumplen la misma funcionalidad que los anteriores, en este caso muestran una sintaxis basada en XML, y no existe una correlación directa entre los controles y las etiquetas HTML que se generan (La abstracción desde el punto de vista de la OOP es mauor aquí).

Los controles Web pertenecen al espacio calificado **System.Web.UI.WebControls**, y contiene al resto de familias de controles ASP .NET, que a continuación veremos:

Si rescribimos el ejemplo anterior para utilizar controles Web tendremos el Código fuente 164 en nuestra página ASP .NET.

```
<html>
<body>
<script language="C#" runat="server">
void Pulsado(Object sender, EventArgs args){
    DateTime ahora;
    ahora = DateTime.Now;
    etiqueta.Text = "El momento actual es: " + ahora.ToString();
}
</script>
<form id="controlesWeb" method="post" runat="server">
<asp:button id="boton" text="Pulsar" onclick="Pulsado" runat="Server"></asp:button>
<asp:label id="etiqueta" runat="server"></asp:label>
</form>
</body>
</html>
```

Código fuente 164

Como puede verse, existen algunas diferencias a la hora de utilizar controles HTML y controles Web, además de las más evidentes relativas a la distinta sintaxis a la hora de declarar los controles.

Algunos nombres de las propiedades de los controles de servidor cambian, como el control **HtmlButton**, que posee la propiedad *value*, y sin embargo el control Web equivalente (**Button**), posee la propiedad *text*. A la hora de indicar los métodos para el tratamiento de eventos de un botón, en los controles HTML utilizamos la propiedad *onserverclick*, y en los controles Web utilizamos la propiedad *onclick*. De igual forma, en los controles HTML se debe distinguir si se trata de un evento de cliente o de servidor, ya que podemos tratar ambos tipos, sin embargo desde los controles Web sólo tenemos la posibilidad de utilizar eventos de servidor, o sea, eventos que son tratados por rutinas en el servidor, métodos de la página pensados para el tratamiento de eventos.

Los controles Web ofrecen un mayor nivel de abstracción que los controles HTML, y su modelo de objetos no refleja la sintaxis HTML necesariamente. Cuando la página se carga en el navegador, el control Web determina el tipo de navegador que ha realizado la petición, y de acuerdo con esta

información genera el código HTML apropiado, podemos decir que en este aspecto se trata de controles inteligentes.

En la siguiente tabla (Tabla 27) se muestra la correspondencia existente entre los controles Web y las etiquetas HTML que se generan:

Control Web	Código HTML
<asp:textbox>	<input type=text>
<asp:button>	<input type=submit>
<asp:imagebutton>	<input type=image>
<asp:checkbox>	<input type=checkbox>
<asp:radiobutton>	<input type=radiobutton>
<asp:listbox>	<select size="5"></select>
<asp:dropdownlist>	<select></select>
<asp:hyperlink>	
<asp:image>	
<asp:label>	
<asp:panel>	
<asp:table>	<table></table>

Tabla 27

Respecto a cuándo utilizar unos y otros, podemos asumir las siguientes recomendaciones generales: Si preferimos usar un modelo de programación similar a Visual Basic (o Delphi), estamos desarrollando Web Forms que deben ser mostrados por varios tipos de navegadores, o se necesita una funcionalidad específica, como puede ser un calendario o un rotador de anuncios, o simplemente queremos asumir y utilizar la nueva funcionalidad, lo recomendable son los nuevos controles. Por el contrario, los controles HTML es preferible utilizarlos en las siguientes situaciones:

- Preferimos un modelo de objetos similar al lenguaje HTML.
- Estamos trabajando con páginas Web existentes y las queremos migrar a Web Forms.
- El control debe interactuar con script de cliente y de servidor.

A continuación vamos a pasar a comentar las restantes familias de controles ASP .NET.

Controles de lista

Pensados para mostrar listados de datos en las páginas ASP.NET, facilitan las tareas de relacionadas con los listados de datos, tarea que suele ser bastante común en la mayoría de las aplicaciones Web (como se comentó en el apartado dedicado a ADO.NET). Generan de forma automática el interfaz de usuario que muestra la información permitiendo la paginación, filtrado y otras operaciones sobre los contenidos.

Los controles de tipo lista se suelen utilizar en combinación con otra nueva característica ofrecida por el entorno de los Web Forms, se trata de Data Binding, es decir, la conexión de los datos con los controles en los que se van a mostrar.

Existen tres controles estándar dentro de esta familia: **Repeater**, **DataList** y **DataGrid**. Relacionado con los controles de lista, más estrechamente con los controles Repeater y DataList, nos encontramos un interesante mecanismo que nos ofrece ASP .NET denominado plantillas (**templates**), que permite personalizar el aspecto de la información que se muestra.

El control es absoluto, existiendo para ello cinco tipos distintos de plantillas, que los comentaremos en detalle en el capítulo correspondiente. Estas plantillas se utilizan en combinación con los controles ASP .NET DataList y Repeater.

El control **Repeater** procesa plantillas de forma iterativa. Mediante las plantillas podemos definir el estilo de la cabecera, elementos, pies y separadores utilizados por el control. En realidad este componente no tiene ningún resultado visual, para obtener un interfaz de usuario debemos aplicar las plantillas que consideremos necesarias.

El control **DataList** también nos ofrece una funcionalidad parecida al control Repeater, pero en este caso permite personalizar aun más el resultado mostrado en pantalla correspondiente a la información que contiene. En el control DataList podemos indicar si los elementos se van a distribuir de manera horizontal o vertical, esto nos permite tener columnas de elementos de información. además podemos indicar el número de columnas en las que deseamos que se distribuyan los elementos.

El último de los controles de lista, y el más complejo, es el control **DataGrid**, este control muestra la información en forma de tabla con celdas y columnas, generando el código HTML equivalente a una tabla. Se puede utilizar de forma muy sencilla, o podemos complicar su uso para llegar a altos niveles de personalización a la hora de mostrar la información.

Controles ricos

En este grupo de controles (*rich controls*) se encuentran una serie de controles avanzados que ofrecen una gran funcionalidad: el control *rotador de anuncios* y el *control calendario*, se espera que se añadan nuevos controles en siguientes versiones de la tecnología. Pertenecen también al espacio de nombres **System.Web.UI.WebControls**.

El control rotador de anuncios (**AdRotator**), no es ninguna novedad de ASP .NET, ya que existía en versiones anteriores de ASP a través de un componente de servidor. Permite mostrar una serie de anuncios dentro de una página ASP, alternando la aparición de los mismos. Para indicar las imágenes y enlaces de los anuncios utilizamos un fichero especial en formato XML.

El control calendario (Calendar), resulta muy sencillo de utilizar aunque ofrece un alto grado de personalización.

Controles de validación

Este grupo de controles permiten validar entradas del usuario a través de los controles de un Web Form. Esta validación se puede realizar en el cliente o en el servidor. Si el navegador lo soporta (navegador de alto nivel) se realizará en el cliente a través de script de cliente, pero si el navegador no lo soporta (navegador de bajo nivel), la validación se llevará a cabo en el servidor, lo que supone una conexión más con el servidor. *Es posible forzar que la validación se realice en el servidor.*

ASP .NET ofrece seis controles de validación que pertenecen también al NameSpace System.Web.UI.WebControls, y son los siguientes:

- **RequiredFieldValidator:** este control valida si se ha rellenado un campo con un valor, es decir, valida los campos requeridos.
- **CompareValidator:** comprueba si un control contiene un valor determinado o coincide con el contenido de otro control, se utiliza con un operador de comparación.
- **RangeValidator:** valida si el valor indicado por el usuario se encuentra dentro de un rango determinado.
- **RegularExpressionValidator:** es el más versátil de los controles de validación, valida si el valor indicado por el usuario se corresponde con una expresión.
- **CustomValidator:** este control permite especificar una función de validación personalizada.
- **ValidationSummary:** este control contiene todos los mensajes de error que se han producido como resultado de las validaciones realizadas y muestra una lista de ellos en la página.

La validación se realiza sobre los controles de un Web Form, y un mismo control puede tener distintos tipos de validación, es decir, se pueden asociar distintos controles de validación sobre un mismo control y por lo tanto, establecer validaciones múltiples.

Data Binding

Una nueva característica, relacionada con ADO .NET es la técnica del **Data Binding**.

El mecanismo de Data Binding (conexión con los datos o enlace con los datos) es una nueva facilidad que ofrece ASP.NET, pero que debe ser familiar para los lectores que conozcan entornos de desarrollo visuales. Mediante este mecanismo podemos asociar datos a un control de interfaz de usuario (control ASP.NET). Se puede establecer el origen de los datos de los controles ASP.NET ofreciendo compatibilidad con todos los navegadores.

La idea del mecanismo de Data Binding es muy sencilla: se conectan controles Web a una fuente de datos y ASP.NET muestra la información de forma automática, es una forma muy sencilla de organizar la información en listas, cajas de texto, *grids*, etc.

La forma más sencilla de establecer el origen de los datos de un control enlazado (ListBox, Repeater, DataGrid, etc.) es utilizar la propiedad DataSource del control correspondiente para indicar la fuente de los datos y a continuación llamar al método **DataBind()** del control. También es posible llamar al método DataBind() de la página ASP .NET, con lo que se establecen todos los enlaces con los datos existentes en todos los controles de servidor de la página.

En la propiedad DataSource del control Web podemos especificar un amplio tipo de fuentes de datos: un array, un objeto DataView de ADO.NET, una expresión, etc. En el Código fuente 165 vamos a mostrar un ejemplo sencillo de Data Binding, consistente en utilizar como origen de los datos un array. En esta página, mostraremos el contenido del array en una lista desplegable (control **DropDownList**).

```
<html>
<%
    ArrayList datos=new ArrayList();
    datos.Add("Bora-bora");
    datos.Add("Morea");
    datos.Add("Pascua");
    datos.Add("Papeete");
    datos.Add("Santiago de Chile");
    lista.DataSource=datos;
    lista.DataBind();
%>
<body>
<form id="DataBindingSencillo" method="post" runat="server">
Destinos: <asp:DropDownList id="lista" runat="server"></asp:DropDownList>
</form>
</body></html>
```

Código fuente 165

El array se crea y se enlaza con el control cuando se carga la página ASP .NET, el resultado es el que aparece en la Figura 60.

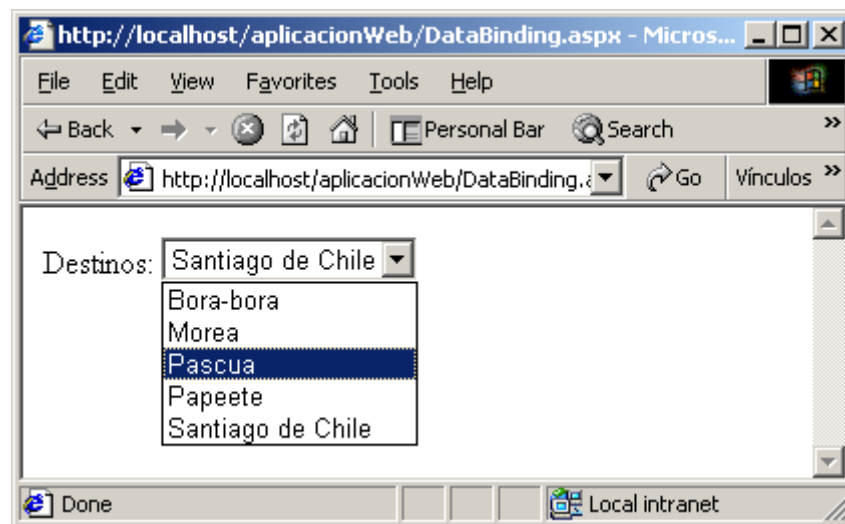


Figura 60

Este mecanismo de enlace con los datos es especialmente útil cuando estamos desarrollando páginas ASP .NET que utilizan bases de datos, es decir, páginas que hagan uso de ADO .NET.

Eventos de la página

ASP .NET ofrecen una serie de eventos que nos permiten estructurar el código de las páginas. Dichos eventos no existían en versiones anteriores de ASP. Los eventos se producen en un orden determinado,

el primero de estos eventos es el evento `Page_Init`, al que se sigue el evento `Page_Load`, que se produce siempre que se carga la página. A continuación se producirán todos los eventos que pertenecen a los controles (controles de servidor ASP .NET) de los Web Forms, y cuyo orden dependen de la interacción del usuario con los mismos.

Cuando se finaliza la ejecución de la página se lanza el método `Page_UnLoad`, este será siempre el último evento de una página.

Relacionada con los eventos de las páginas nos encontramos la propiedad `IsPostBack` del objeto `Page`, esta propiedad tendrá el valor verdadero si la página ha sido enviada al servidor alguna vez, por ejemplo mediante un botón de submit. Así si deseamos indicar si una página ha sido ya enviada al servidor o no, utilizaremos el Código fuente 166.

```
<script language="C#" runat="server">
void Page_Load(Object fuente,EventArgs evento){
    if (Page.IsPostBack)
        mensaje.Text="Página enviada al servidor";
    else
        mensaje.Text="La página no se ha enviado al servidor";
}
</script>
<html>
<body>
<form method="POST" id="formulario" runat="server">
<asp:label id="mensaje" runat="Server"></asp:label>
<asp:button id="boton" runat="Server" text="Enviar"/>
</form>
</body>
</html>
```

Código fuente 166

Si nos remontamos al ejemplo del apartado anterior, podemos modificar la página ASP .NET para que únicamente se realice la conexión con los datos (Data Binding) la primera vez que se cargue la página, es decir, se va a utilizar el método `Page_Load`. Esto se puede ver en el.

En el próximo apartado se va a comentar otra característica de ASP .NET que permite acometer el desarrollo de aplicaciones Web de forma más sencilla, se trata de la posibilidad de utilizar la separación de código, para que la página ASP :NET sea lo más sencilla posible y se ocupe de mostrar el interfaz al usuario, estando separada de la lógica de la aplicación.

Separación de código

ASP .NET permite realizar una clara separación entre la lógica de las páginas y el interfaz, para ello se pueden utilizar varios mecanismos: Code-Behind, controles de usuario y componentes. Cualquiera de estos mecanismos nos ofrecen las siguientes ventajas:

- La posibilidad de realizar una clara división del trabajo entre los programadores y los diseñadores.
- Permite a los desarrolladores o programadores utilizar sus entornos preferidos.
- Los autores de código HTML pueden utilizar sus herramientas de diseño de páginas Web.

La más sencilla de las posibilidades es Code-Behind (código por detrás o código oculto). Este nuevo mecanismo nos permite separar el interfaz de usuario de la lógica de la aplicación, de esta forma el código fuente puede residir en un fichero separado, que puede ser una clase de C#, este código es el que se denomina Code-Behind, ya que permanece oculto.

Para indicar que una página ASP .NET va a utilizar una clase determinada (como Code-Behind), se utilizan los atributos Inherits y Src de la directiva Page. En el atributo Inherits debemos indicar el nombre de la clase y en el atributo Src la ruta del fichero de la clase. Si no indicamos la ruta ASP .NET buscará la clase en el directorio bin de la aplicación. Una vez indicado en la página ASP .NET la clase que se va a utilizar, únicamente nos queda definir e implementar dicha clase.

La otra forma de separar el código de la presentación es mediante los controles de usuario. Los controles de usuario permiten de una manera sencilla reutilizar y separar funcionalidades comunes a través de una aplicación Web completa. Los controles de usuario se utilizan dentro de las páginas ASP .NET de la misma que forma que se utilizan los controles de servidor. Para incluir y utilizar un control de usuario en una página se debe hacer uso de la directiva Control.

Los controles de usuario vienen a sustituir a los ficheros include que se utilizaban en versiones anteriores de ASP. La diferencia más notable entre los controles de usuario y el código oculto (Code-Behind) es que los controles permiten generar parte del interfaz de usuario y el código oculto únicamente permite la utilización de métodos a través del mecanismo de herencia.

Y la última de las posibilidades de separación de código es mediante el uso de componentes. Un componente básicamente es una clase que puede ser instanciada desde una página ASP .NET que debe importar el componente. Por lo tanto para utilizar un componente se debe utilizar la directiva Import.

Hola Mundo con ASP .NET

Una vez realizada la introducción a ASP .NET y a algunas de las características que ofrece, vamos a centrarnos en realizar una página ASP .NET que va a mostrar el famoso mensaje “Hola Mundo” cuando pulsemos sobre el botón incluido en la página.

El Código fuente 167 se corresponde con el código de la página ASP .NET, primero se muestra el código completo y a continuación lo vamos a comentar detenidamente.

```
<html>
<body>
<script language="c#" runat="server">
void Pulsado(Object fuente, EventArgs args){
    etiqueta.Style["font-size"]="20";
    etiqueta.Text="¡Hola Mundo!";
}
</script>
<form id="formulario" method="post" runat="server">
    <asp:label id="etiqueta" runat="Server"></asp:label>
    <asp:button id="boton" onclick="Pulsado" runat="server"
text="Pulsa"></asp:button>
</form>
</body>
</html>
```

Código fuente 167

Si ejecutamos la página ASP .NET del ejemplo y pulsamos sobre el botón mostrará el aspecto de la Figura 61.

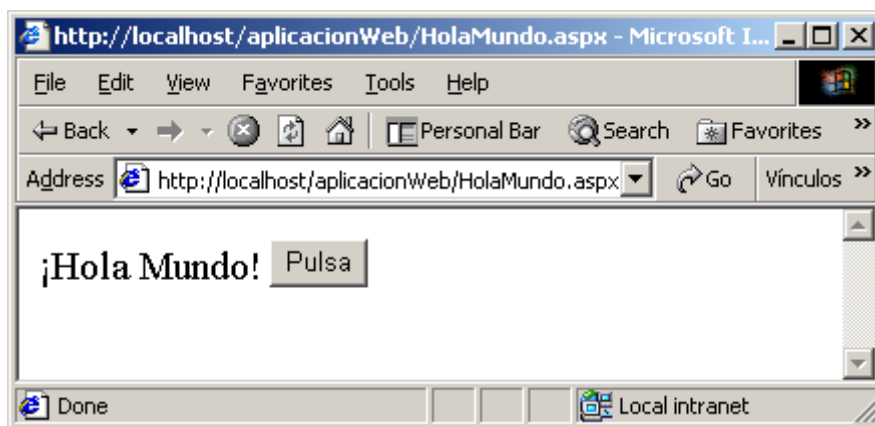


Figura 61

Lo primero que nos puede llamar la atención del Código fuente 167 es el fragmento de código que parece entre las etiquetas `<script>`. Este fragmento de código escrito en el lenguaje C# define un método que se ejecutará cuando el usuario pulse sobre el botón que pertenece al Web Form contenido en la página ASP .NET. El Web Form lo trataremos más tarde, ahora nos vamos a centrar en la definición del método `Pulsado()`, que es el método de tratamiento del evento de pulsación del botón.

Una vez que indicamos el lenguaje que se va a utilizar en la etiqueta `<script>` se pasa a definir la cabecera del método `Pulsado()`. En este caso este método no devuelve ningún valor (`void`) y posee dos parámetros, que en este caso nos ofrecen información acerca del evento que se ha producido, no se debe olvidar que el método que estamos definiendo es un método utilizado para el tratamiento de eventos.

Este método posee dos sentencias. La primera de ellas accede al estilo de la etiqueta del Web Form para darle un tamaño determinado, y la segunda establece el valor de la propiedad `Text` de la etiqueta con el mensaje “¡Hola mundo!”.

El método `Pulsado()` es muy breve, pero dentro de él se pueden apreciar algunas de las características del lenguaje C#, como el delimitador de sentencias, el punto y coma (;), y el delimitador de bloques de código, los corchetes (`{}`).

Una vez finalizada la definición del método y del cierre de las etiquetas `<script>`, nos encontramos con la definición de un Web Form (formulario Web), que será el que va a contener el botón. La sintaxis es muy similar a la utilizada para definir un formulario típico de HTML, la diferencia se encuentra en la propiedad `runat` que posee el valor `server`, es decir, indica que es un formulario de servidor, un Web Form.

Entre las etiquetas `<form>` que definen el formulario Web, se encuentran las etiquetas con sintaxis XML que instancian dos controles ASP .NET. Se trata de dos controles Web que pertenecen a las clases `Label` y `Button` del espacio calificado `System.Web.UI.WebControls`.

El control `Label` representa a la etiqueta que mostrará el mensaje en pantalla y el objeto `Button` es el botón que invocará el método `Pulsado()` cuando el botón sea pulsado por el usuario. Los dos controles Web poseen un atributo `id`, este atributo es el nombre que se utilizará para acceder a ellos, en el método `Pulsado()` utilizamos el identificador “etiqueta” para acceder al objeto `Label`.

El objeto Button posee la propiedad onclick en la que indicamos el método responsable de tratar el evento de pulsación del botón, en este caso se trata del ya conocido método Pulsado().

Una vez creados los controles ASP .NET se cierra el Web Form con la etiqueta correspondiente.

Como se puede observar en el método Pulsado() no se utiliza el objeto Request ni su colección Form para acceder a los elementos del formulario, ya que mediante los Web Forms tenemos acceso a todos los controles, ya que forman parte de la página comportándose como propiedades de la misma.

Para poder ejecutar esta página ASP .NET se escribirá todo el código en un fichero con extensión .aspx que residirá en un directorio de publicación en Internet que además posea el permiso de ejecución de secuencias de comandos (scripts).

Para poder probar este primer ejemplo debemos tener instalado el servidor Web Internet Information Server 5.0. El directorio de publicación en Internet, por defecto, de este servidor es C:\inetpub\wwwroot. Por lo tanto si creamos una subcarpeta llamada C:\inetpub\wwwroot\prueba y copiamos la página ASP .NET llamada HolaMundo.aspx a este directorio, para ejecutarla escribiremos en el navegador <http://nombreServidor/prueba/HolaMundo.aspx>. Se debe recordar que una página ASP .NET debe ser siempre ejecutada a través del servidor Web.

Con este último apartado damos por finalizado este capítulo de introducción a la tecnología ASP .NET. En el siguiente capítulo veremos una serie de conceptos generales sobre la orientación a objetos y los lenguajes orientados a objetos, y en un tercer capítulo trataremos el lenguaje que vamos a utilizar en las páginas ASP .NET, es decir, el nuevo lenguaje de Microsoft C# (C Sharp).

Creación de una primera aplicación ASP.NET

Una vez seleccionado un nuevo proyecto, en la ventana de tipos de aplicación usaremos el tipo ASP .NET Web Application, utilizando el lenguaje C#.

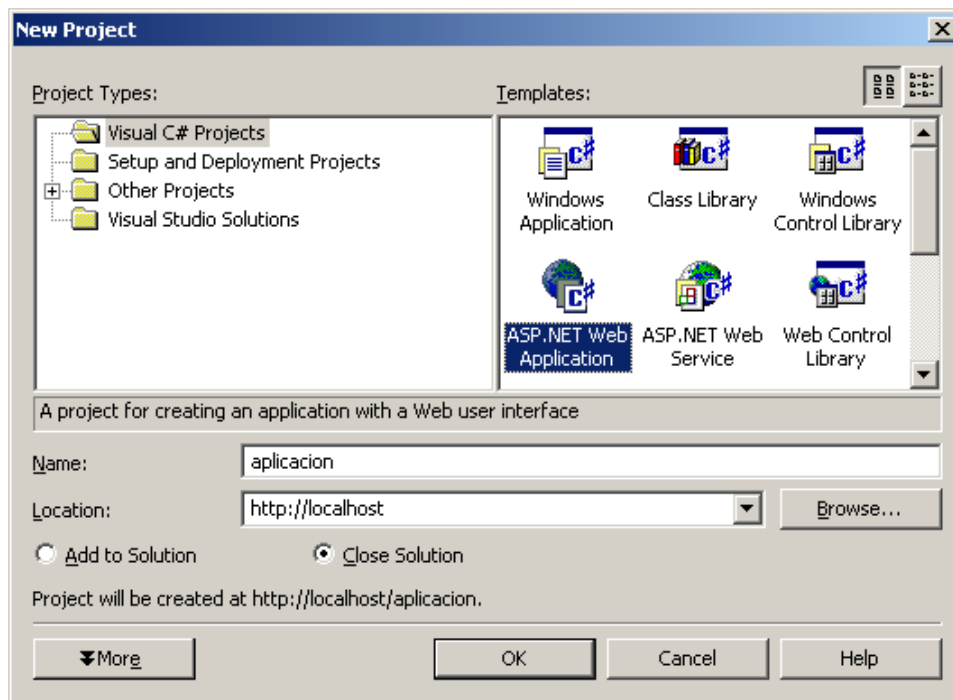


Figura 62

También debemos indicar el nombre del directorio que va a contener la aplicación y el servidor en el que se va a desarrollar. En nuestro caso como se puede ver en la Figura 62, la aplicación se llama aplicacion y se va a crear en la URL `http://localhost/aplicacion`, localhost va a ser el nombre genérico con el que se hace referencia a la máquina local, es decir, a nuestro propio equipo.

Cuando pulsamos el botón etiquetado como OK Visual Studio .NET procede a crear la aplicación ASP .NET en el lugar y con el nombre indicados. De forma automática se crean una serie de ficheros en la aplicación ASP .NET, que se pueden observar en la Figura 63, esta figura muestra el explorador de la solución (Solution Explorer) de Visual Studio .NET.

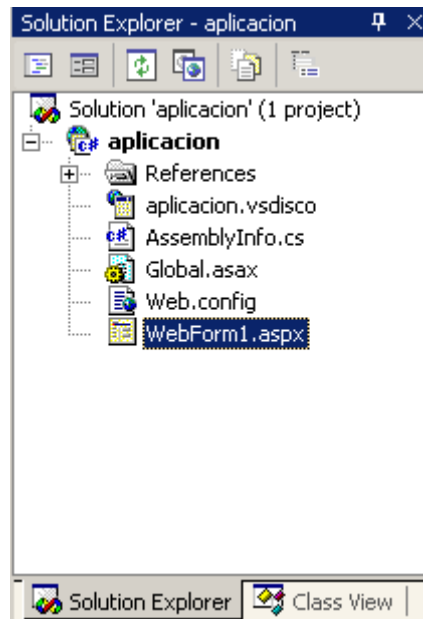


Figura 63

Los ficheros creados son: el fichero especial GLOBAL.ASAX, con funciones similares la fichero GLOBAL.ASA de las versiones anteriores de ASP, es decir, la de definir los eventos propios de la aplicación; el fichero WEB.CONFIG que va a contener la configuración de la aplicación ASP .NET, y una página ASP .NET llamada WEBFORM1.ASPX, que va a ser una página que contiene un Web Form y que podremos utilizar en nuestra aplicación ASP .NET de la forma que deseemos.

Si pulsamos sobre este fichero podemos acceder a la página ASP .NET para añadir un sencillo código fuente que nos permita mostrar la fecha y hora actuales. El código de la página se puede ver en el Código fuente 168.

```
<%@ Page language="c#"%>
<html>
<body>
<%DateTime ahora=DateTime.Now;%>
<%=ahora%>
</body>
</html>
```

Código fuente 168

Para ejecutar nuestra página ASP .NET debemos escribir la URL <http://localhost/aplicación/webform1.aspx> en nuestro navegador Web, el resultado de la ejecución de esta página ASP .NET se puede ver en la Figura 64.

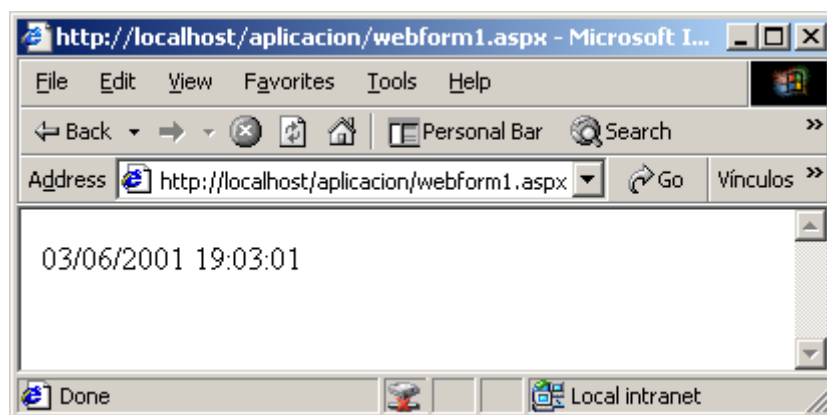


Figura 64

A grandes rasgos se han comentado los distintos elementos que nos van a permitir desarrollar aplicaciones ASP .NET.

Introducción a los Web Forms

Los Web Forms (formularios Web) es una característica de ASP .NET que ofrece un nuevo modelo de programación que nos va a permitir generar contenidos dinámicos de una forma más sencilla. Esta característica no existía en versiones anteriores de ASP.

Los Web Forms vienen a sustituir e integrar los formularios HTML dentro del entorno de programación de ASP .NET. Un Web Form en última instancia va a generar el código HTML correspondiente al formulario que estamos diseñando.

Una de las grandes ventajas que ofrecen los Web Forms es que ofrecen un modelo de programación muy similar al de los tradicionales formularios de Visual Basic, incluso podemos realizar el tratamiento de eventos del cliente desde código del servidor. Se puede decir que vamos a realizar el tratamiento de los eventos del cliente con código ASP .NET, es decir, con código que será ejecutado en el servidor Web.

Los Web Forms tienen tanta importancia dentro de ASP .NET que incluso en el entorno de desarrollo Visual Studio .NET cuando queremos añadir una nueva página, no parece la opción de añadir nueva página ASP .NET, sino que se muestra la posibilidad de añadir un nuevo Web Form, es decir, en algunos lugares las páginas ASP .NET también se denominan Web Forms.

A continuación se van a describir de forma general las ventajas que nos ofrecen los Web Forms, en los siguientes capítulos se verán en detalle el modelo de programación ofrecidos por los Web Forms junto con los controles ASP .NET de servidor. Las ventajas que nos ofrecen los Web Forms son las siguientes:

- Nos permiten reutilizar controles de interfaz de usuario (UI) que encapsulan su funcionalidad, reduciendo de esta forma el código que el desarrollador debe escribir.

- Ofrece un código más limpio y claro en la página, ya que no aparece mezclado con código HTML como ocurría en las versiones anteriores de ASP, en las que era bastante usual intercalar código HTML con código ASP, aunque en ASP .NET esto también se permite por compatibilidad, pero si nos ceñimos al modelo de programación que ofrecen los Web Forms no es necesario.
- Podemos construir todo el interfaz del Web Form a través de herramientas o entornos de desarrollo, al igual que construimos los formularios con Visual Basic arrastrando y soltando los controles del interfaz de usuario. Visual Studio .NET nos ofrece una vista de diseño para los Web Forms, que incluso nos permite diseñar los Web Forms situando sus controles según las posiciones x e y de la pantalla.
- Integra el tratamiento de eventos del cliente con el código correspondiente del servidor. El tratamiento de eventos es muy similar al que ya se utilizaba en los formularios de las anteriores versiones de Visual Basic.
- Realizan un mantenimiento del estado del formulario (Web Form) de forma automática entre distintas peticiones, es decir, si seleccionamos una serie de valores en el Web Form, al enviarlo de forma automática se van a mantener estos valores en el caso de que se vuelva a recargar el formulario.
- Ofrece una mayor separación entre la lógica de la aplicación y la presentación de la misma.
- Permite utilizar un gran conjunto de controles de servidor que generarán el código HTML necesario para mostrar el Web Form en la página cargada por el cliente en su navegador. Después veremos que dentro de los Web Forms podemos encontrar varios tipos de controles.

Para crear un formulario Web o Web Form únicamente debemos utilizar una etiqueta <FORM>, al estilo de las etiquetas del lenguaje HTML. Pero presenta la particularidad que debe poseer la propiedad runat con el valor server. En el Código fuente 169 se puede ver un fragmento de código que todo Web Form debe poseer.

```
<form id="WebForm" method="post" runat="server">
</form>
```

Código fuente 169

Como se puede comprobar es muy similar al código de un formulario de HTML. Por lo tanto el código necesario para convertir un formulario HTML en un Web Form es bastante sencillo, eso sí, sin tener en cuenta los controles de servidor que puede contener el Web Form.

Dentro del Web Form se encontrarán los distintos controles de servidor de ASP .NET necesarios para recoger la información correspondiente. Es decir, un Web Form se va a encontrar compuesto por los distintos controles de servidor de ASP .NET, que realizarán las mismas funciones que los típicos controles del lenguaje HTML que constituían los campos de un formulario.

Estos controles de servidor están muy unidos a los Web Form, ya que es dentro de los formularios Web donde vamos a hacer uso de estos controles. Los controles de servidor, también denominados controles de servidor ASP .NET o simplemente controles ASP .NET, suponen otra nueva y revolucionaria funcionalidad ofrecida por ASP .NET. Los controles ASP .NET combinados con los Web Forms nos van a permitir construir páginas ASP .NET dinámicas de forma muy sencilla.

Podemos decir que los Web Form junto con los controles ASP .NET ofrecen un nuevo modelo de programación dentro de ASP .NET.

Introducción a los controles ASP .NET

Los controles ASP .NET son una serie de objetos de servidor que generarán el correspondiente código HTML para que el usuario pueda utilizarlos en la página cargada en su navegador. Estos controles nos van a permitir manejar eventos generados por el cliente, como puede ser una pulsación de un botón, con código de servidor.

Gracias a la los controles ASP .NET ya no será necesario utilizar los típicos campos de formulario de HTML, es decir, los controles del lenguaje HTML, sino que al ejecutarse la página ASP .NET correspondiente los controles de servidor de ASP .NET generarán todo el código equivalente en el lenguaje HTML, para que así pueda ser interpretado el resultado de la ejecución de la página por cualquier navegador Web.

Los controles ASP .NET nos una muestra más del interesante cambio en la filosofía de desarrollo que aporta ASP .NET, ahora el código de servidor, va a estar separado de forma más clara, que en el pasado, del código del cliente, dando lugar a un código mucho más sencillo, estructurado y por lo tanto legible.

Cada control se corresponde con una clase determinada perteneciente a un espacio con nombre determinado. Existe un gran número de controles de servidor y cada uno de ellos pertenece a una clase del .NET Framework.

Los numerosos controles de servidor incluidos en ASP .NET se pueden agrupar en cuatro categorías o familias:

- Controles intrínsecos, que a su vez se subdividen en dos grupos, y que representan elementos HTML. Los dos grupos que representan los controles intrínsecos se denominan: controles HTML y controles Web, más adelante veremos las diferencias y utilización de estos dos tipos de controles.
- Controles de lista, utilizados para distribuir y mostrar datos en una página.
- Controles ricos, ofrecen funcionalidades avanzadas de interfaz de usuario.
- Controles de validación, ofrecen distintos tipos de validación de entrada de datos.

Las tres últimas categorías se pueden decir que ese encuentran reunidas dentro de los controles Web, es decir, los controles HTML están limitados únicamente a los controles intrínsecos.

A lo largo de los siguientes apartados vamos a ir comentando brevemente los distintos tipos de controles de servidor.

Además de los controles HTML estándar que representan campos de un formulario, los controles de servidor de ASP .NET permiten a los desarrolladores utilizar controles más ricos y complejos, como puede ser el control de rotación de anuncios (AdRotator), para algunos lectores este control les será familiar de las versiones anteriores de ASP .NET.

Cada control ASP .NET es capaz de ofrecer un completo modelo de objetos que contiene propiedades, métodos y eventos, no debemos olvidar que los controles ASP .NET son instancias de objetos que

podemos encontrar en distribuidos en dos espacios con nombre del .NET Framework, se trata de los espacios con nombre: System.Web.UI.HtmlControls y System.Web.UI.WebControls.

En algunos casos nos puede interesar utilizar controles Web, y en otros controles HTML, los controles Web deberán ser usados en las siguientes situaciones:

- Preferimos un modelo de programación similar a Visual Basic.
- Estamos desarrollando Web Forms que deben ser mostrados por varios tipos de navegadores.
- Se necesita una funcionalidad específica, como puede ser un calendario o un rotador de anuncios.

Y los controles HTML es preferible utilizarlos en las siguientes situaciones:

- Preferimos un modelo de objetos similar al lenguaje HTML.
- Estamos trabajando con páginas Web existentes y las queremos migrar a Web Forms.
- El control debe interactuar con script de cliente y de servidor.

Hola Mundo con Web Forms

Antes de seguir comentando más aspectos de los Web Forms y de los controles de servidor de ASP .NET vamos a mostrar y comentar un sencillo ejemplo que utiliza un Web Form con varios controles ASP .NET, se podría considerar que es el típico ejemplo “Hola Mundo” pero con un Web Form.

El ejemplo consiste en crear un Web Form que contenga un botón que al ser pulsado muestre por pantalla el mensaje “Hola Mundo”. En el Código fuente 170 se puede ver el código de la página ASP .NET que contiene este formulario Web.

```
<html>
<body>
<script language="C#" runat="server">
void Pulsado(Object sender, EventArgs args){
    etiqueta.InnerText ="Hola Mundo";
}
</script>
<form id="controlesHTML" method="post" runat="server">
<input type="button" id="boton" runat="server" value="Pulsar"
        onserverclick="Pulsado" NAME="boton">
<div id="etiqueta" runat="server"></div>
</form>
</body>
</html>
```

Código fuente 170

En la Figura 65 se puede ver el resultado de la ejecución de esta página ASP .NET.

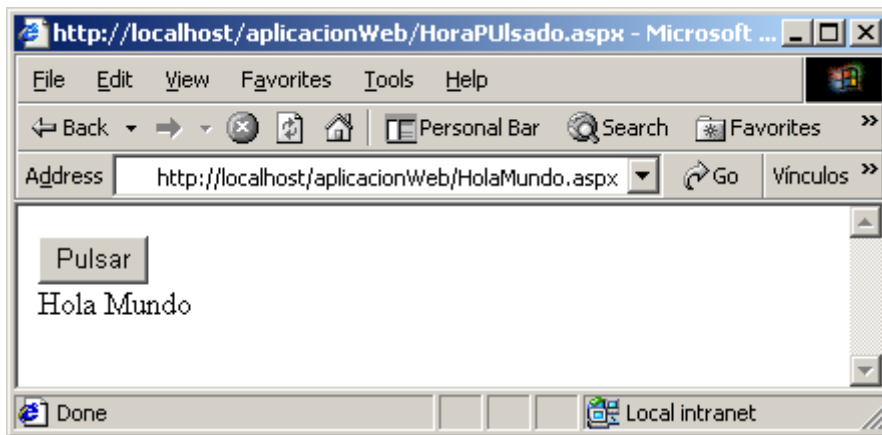


Figura 65

El Web Form contiene dos controles ASP .NET, un botón que será un objeto de la clase `HtmlInputButton` y una etiqueta que será un objeto de la clase `HtmlGenericControl`, ambas clases se encuentran en el espacio de nombres `System.Web.UI.HtmlControls`. En este caso el Web Form para interactuar con el usuario utiliza controles HTML.

Además, a la vista del código del ejemplo se pueden hacer los siguientes comentarios. La sintaxis que ofrecen los controles HTML de servidor es muy similar a la sintaxis de los controles clásicos del lenguaje HTML. Para identificar de forma única un control ASP .NET se debe utilizar la propiedad `id`, mediante este identificador podemos hacer referencia al objeto correspondiente en la página. Para indicar que en la pulsación del botón se debe ejecutar un método determinado, es decir, el método que realiza el tratamiento del evento, se utiliza la propiedad `onserverclick`, esto es específico de los controles HTML, luego veremos las diferencias de sintaxis con los controles Web, en el caso de los controles HTML la sintaxis de los mismo le será muy familiar al lector ya que es al estilo del lenguaje HTML, sin embargo, como veremos más adelante, la sintaxis de los controles Web está basada en XML.

Si queremos utilizar controles Web en nuestro ejemplo anterior en lugar de controles HTML podemos utilizar el Código fuente 171.

```
<html>
<body>
<script language="c#" runat="server">
void Pulsado(Object fuente, EventArgs args){
    etiqueta.Text="Hola Mundo";
}
</script>
<form id="formulario" method="post" runat="server">
    <asp:label id="etiqueta" runat="Server"></asp:label>
    <asp:button id="boton" onclick="Pulsado" runat="server"
text="Pulsa"></asp:button>
</form>
</body>
</html>
```

Código fuente 171

En este nuevo ejemplo se puede comprobar la diferencia existente entre ambos tipos de controles ASP .NET, por lo menos en lo que se refiere a nivel de sintaxis. Como se puede observar el código de servidor, es decir, el código escrito en C# permanece idéntico en ambos ejemplos.

Para terminar con el comentario de este ejemplo de Hola Mundo con un Web Form, podemos ver el código fuente de la página HTML que se ha generado como resultado de la ejecución de nuestra página ASP .NET, este código se ofrece a continuación (Código fuente 172).

```
<html>
<body>
<form name="formulario" method="post" action="Holamundo.aspx" id="formulario">
<input type="hidden" name="__VIEWSTATE"
value="dDwtOTk1MjE0NDA4O3Q8O2w8aTwxPjs+O2w8dDw7bDxpPDE+Oz47bDx0PHA8cDxsPFRleHQ7Pjts
PEhvbGEgTXVuZG87Pj47Pjs7Pjs+Pjs+Pjs+" />
        <span id="etiqueta">Hola Mundo</span>
        <input type="submit" name="boton" value="Pulsa" id="boton" />
</form>
</body>
</html>
```

Código fuente 172

Como se puede comprobar se han generado de forma automática las etiquetas HTML correspondientes, pero también se ha añadido dentro del formulario un campo oculto denominado `__VIEWSTATE` con un extraño valor. Este campo oculto, que volveremos a comentar en próximos ejemplos, permite al Web Form mantener el estado entre distintas llamadas, es decir, en distintas peticiones se va conservando el valor de los campos del formulario.

La clase Page

Cada vez que una página ASP .NET es demandada desde el servidor Web, es decir, un cliente a través de un navegador Web solicita una URL que indica un fichero .ASPX en el servidor, se crean una serie de componentes que constituyen la página solicitada y que se compilan en una unidad. Estos componentes pueden ser: el fichero .ASPX que ha sido solicitado, la clase .NET que contiene el código de la página, controles de usuario de la página, etc.

La unidad formada por los distintos componentes compilados da lugar a un objeto de la clase `Object.Contro.TemplateControl.Page`, esta clase que se crea de forma dinámica se instanciará cada vez que se realice una petición del fichero .ASPX. Una vez que se ha instanciado un objeto de la clase esta clase, este objeto será utilizado para procesar las sucesivas peticiones y devolver los datos correspondientes al cliente que haya realizado la solicitud.

El proceso de compilación de la página únicamente se realiza cuando se modifica el código fuente de la página ASP .NET, de esta forma el proceso resultante es bastante eficiente.

Dentro del código de una página ASP .NET siempre vamos a tener acceso a la clase `Page`, esta clase va a servir como contenedor de todos los componentes que constituyen la página, nos permite el acceso a distintos aspectos de la página, como pueden ser sus propiedades, métodos y eventos.

En resumen y de manera sencilla podemos decir que el objeto instancia de la clase `Page`, va a representar a la página ASP .NET actual.

A lo largo del presente capítulo vamos a comentar los distintos eventos, propiedades y métodos que nos ofrece la clase `Page`, también veremos otros aspectos relacionados con la clase `Page`, como pueden ser la directivas de la página y la separación de código fuente.

Eventos de la página

ASP .NET ofrecen una serie de eventos que nos permiten estructurar el código en las páginas, estos eventos no existían en versiones anteriores de ASP. Los eventos se producen en un orden determinado o en un momento determinado.

A continuación se van a enumerar los distintos eventos que posee la clase Page:

- **Init:** este evento es lanzado cuando la página es inicializada, cuando es lanzado este evento todavía no se han creado por completo los distintos controles de la página. Este evento es tratado en el método `Page_Init`.
- **Load:** este evento se lanzaría a continuación del método `Init` y es lanzado cuando la página se ha cargado, en este caso todos los controles de la página ya han sido creados. Este evento se lanzará cada vez que la página ASP .NET es ejecutada. Este evento es tratado en el método `Page_Load`.
- **PreRender:** el evento se lanzará justo antes de enviar la información al cliente. Este evento es tratado en el método `Page_PreRender`, y siempre es lanzado después del evento `Load`.
- **Unload:** este otro evento se lanzará en último lugar, y tiene lugar cuando la página ha finalizado de procesarse, es decir, cuando se ha terminado la ejecución de la página y toda la información ha sido enviada al cliente. Este evento es tratado en el método `Page_UnLoad`.
- **AbortTransaction:** los cuatro eventos anteriores se ejecutarán siempre, sin embargo los tres eventos que quedan por comentar se producirán en un momento determinado cuando una cierta situación tenga lugar. Este nuevo evento será lanzado cuando la transacción en la que está participando la página actual es abortado.
- **CommitTransaction:** este evento se lanzará cuando la transacción en la que participe la página se lleve a cabo sin errores.
- **Error:** este evento se lanzará cuando se produzca una excepción no tratada dentro de la página. El método `Page_Error` se utilizará cuando deseemos realizar nuestro propio tratamiento de errores, esto lo veremos más en detalle en el capítulo dedicado al tratamiento de errores y depuración.

En el Código fuente 173 se muestra una sencilla utilización de algunos de estos métodos. El ejemplo muestra una serie de mensajes en el cliente para ilustrar el orden de ejecución de los distintos eventos.

```
<html>
<script language="C#" runat="server">
void Page_Init(Object sender, EventArgs e) {
    Page.Response.Write("Evento Init<br>");
}

void Page_Load(Object sender, EventArgs e) {
    Page.Response.Write("Evento Load<br>");
}

void Page_UnLoad(Object sender, EventArgs e) {
}

void Page_PreRender(Object sender, EventArgs e) {
```

```
Page.Response.Write("Evento PreRender<br>");  
}  
</script>  
<body>  
<asp:Label runat="server" id="texto">Eventos de la página</asp:Label>  
</body>  
</html>
```

Código fuente 173

En la Figura 66 se puede ver el resultado de la ejecución de esta página ASP. NET.

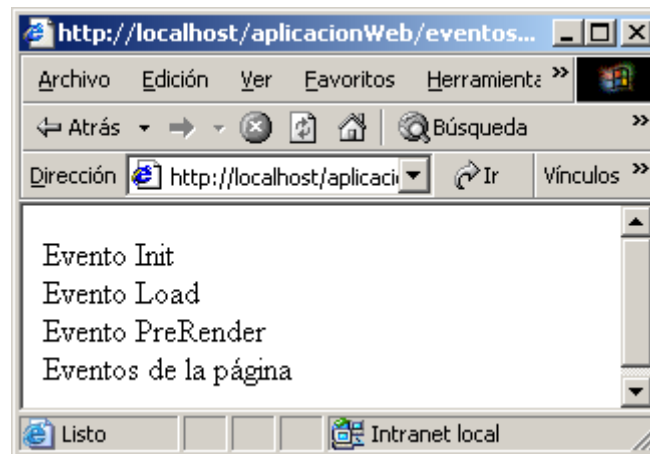


Figura 66

En este código de ejemplo accedemos a una propiedad de la clase Page llamada Response, que es una referencia a un objeto de la clase System.Web.HttpResponse, este objeto permite acceder a la respuesta del protocolo HTTP que se envía al cliente, en este caso el cliente va a ser un navegador Web, es decir, permite manipular la salida que se va a enviar al navegador, en este caso utilizamos el método Write() para escribir una cadena que identifique el evento que se ha producido.

A los lectores conocedores de versiones anteriores de ASP les resultará familiar la propiedad Response de la clase Page, ya que se corresponde con el objeto integrado Response de las otras versiones de la tecnología ASP, más adelante veremos en detalle esta propiedad de la clase Page.

Pero volvamos al tema principal de este apartado que son eventos de la clase Page. A la hora de utilizar los distintos métodos que manejan los ya comentados eventos de la página hay que tener en cuenta una serie de puntos:

- El evento Load se lanza cada vez que la página ASP .NET se ejecuta.
- Una vez cargada la página se pueden ir lanzando los distintos eventos que pueden ser emitidos por los distintos controles Web presentes en la página actual.
- Durante el evento Init se puede acceder a los controles Web de la página ASP .NET, pero estos controles tendrán sus valores por defecto no los valores anteriores que hayan podido ser establecidos antes de que se produjera el envío de la página, es decir, no se ha conservado el estado de los controles del formulario ya que todavía no se encuentra el campo VIEWSTATE del formulario Web, recuerdo a los que este campo se trata de un campo oculto del formulario denominado _VIEWSTATE que permite mantener el estado entre distintas llamadas, es decir, en distintas peticiones se va conservando el valor de los campos del formulario.

- En el evento UnLoad no se tiene acceso al objeto Page, ya que este evento se produce cuando se destruye dicho objeto.

Con estas consideraciones terminamos el apartado dedicado a los eventos de la página ASP .NET, a continuación vamos a comentar las distintas propiedades de la clase Page.

Propiedades de la clase Page

Para acceder a una propiedad de la clase Page lo podemos hacer de dos formas: indicando delante de la propiedad que se trata de la clase Page o bien utilizando la propiedad directamente sin anteponer nada delante de la misma, en el se muestran estas dos posibilidades, se debe indicar que no existe ningún tipo de mejora el rendimiento si se utiliza una u otra de las dos posibilidades.

```
Page.Response.Write("Es equivalente");  
Response.Write("Es equivalente");
```

Código fuente 174

A continuación se va a comentar brevemente las distintas propiedades que nos ofrece la clase Page, no vamos a comentar en detalle cada una de ellas, sino que más adelante, a medida que se siga avanzando en el desarrollo del texto, se comentarán las más relevantes, aquí únicamente las vamos a enumerar a modo de referencia rápida y para tener una visión completa y general de la clase que nos ocupa.

Siempre que sea aplicable y necesario se comentará la correspondencia de la propiedad con el antiguo modelo de objetos integrado que presentaban las versiones anteriores de ASP. Esto se hace así para que los lectores que hayan utilizado alguna versión previa de ASP se sientan más cómodos con la versión ASP .NET.

Teniendo en cuenta todo lo anterior, podemos decir que las propiedades principales de la clase Page son las siguientes:

- **Application:** esta propiedad nos va a ofrecer una referencia a un objeto de la clase System.Web.HttpApplicationState, este objeto nos va a permitir almacenar y acceder a información que va a ser común a toda la aplicación Web, es decir, es una información compartida por todos los clientes de una aplicación Web determinada. Esta propiedad es equivalente al objeto integrado Application de anteriores versiones de ASP.
- **Cache:** esta propiedad permite acceder a un objeto de la clase System.Web.Caching.Cache. La propiedad Cache no tiene ninguna equivalencia con los objetos integrados de las anteriores versiones de ASP, ya que ofrece una nueva funcionalidad que permite almacenar información para que pueda ser mantenida durante distintas peticiones sobre distintas páginas ASP .NET, esta información será global para toda la aplicación. Pero también existe un segundo tipo de caché denominado caché de salida, que no utiliza el objeto Cache, sino que almacena la salida generada por las páginas ASP, y utiliza esta copia almacenada para las siguientes peticiones de la página.
- **ClientTarget:** esta propiedad de la clase String nos permite sobrescribir la detección automática de navegador Web ofrecida por ASP .NET, de esta forma podemos indicar un cliente Web específico con unas propiedad y capacidades determinadas.

- **EnableViewState**: propiedad de tipo Boolean que nos permite desactivar el mantenimiento del estado entre distintas llamadas de los controles Web de la página. Su valor por defecto es true.
- **ErrorPage**: propiedad de la clase String que permite indicar la página de error que se va a utilizar en el caso de que se produzca una excepción no tratada en la página ASP .NET actual. Por defecto esta propiedad no tiene ningún valor, por lo que se utiliza una página ASP .NET genérica para el tratamiento de errores.
- **IsPostBack**: esta propiedad de la clase Boolean devolverá true si la página ASP .NET actual ya ha sido enviada al servidor en alguna ocasión. Si tiene el valor false indicará que la página es la primera vez que se carga y nos servirá de indicador para poder inicializar los controles Web de la página o bien realizar otras labores de inicialización. Esta propiedad ya la hemos utilizado en diversos ejemplos dentro del entorno de los Web Forms.
- **IsValid**: esta propiedad también de tipo Boolean, presentará el valor verdadero si todos los controles de validación presentes en la página actual han realizado sus operaciones de validación con éxito. Si alguna de las validaciones de los controles Web de validación falla esta propiedad tendrá el valor falso.
- **Request**: al igual que sucedía con otras propiedades ya comentadas de la clase Page, esta propiedad ofrece una referencia a un antiguo objeto integrado del modelo de objetos de versiones anteriores de ASP, en este caso nos devuelve una instancia de un objeto de la clase System.Web.HttpRequest. Este objeto va a permitir acceder a toda la información necesaria de la petición del protocolo HTTP que ha sido utilizada para demandar la página de el servidor Web.
- **Response**: nos seguimos encontrando con referencias a los objetos integrados de versiones anteriores de ASP. El objeto de la clase System.Web.HttpResponse nos va a permitir manipular la respuesta devuelta al cliente que ha realizado la petición sobre la página ASP .NET actual, este objeto representa la respuesta HTTP que se envía al cliente.
- **Server**: ofrece una referencia a un objeto de la clase System.Web.HttpServerUtility, este objeto tiene la misma funcionalidad que el objeto Server de anteriores versiones de ASP, es decir, es un compendio de utilidades que permiten realizar una serie de operaciones sobre las peticiones recibidas. Una de las funcionalidades que se ofrecía como principal en las versiones anteriores de ASP, ahora en ASP .NET no tiene dicha importancia se trata de la posibilidad de crear componentes existentes en el servidor, ya hemos visto que en ASP .NET los objetos se crean de manera distinta, ahora los instanciamos mediante el operador new e importando el espacio con nombre de sea necesario.
- **Session**: al igual que sucedía con las últimas propiedades descritas, esta propiedad permite acceder a un objeto incluido dentro del antiguo modelo de objetos de ASP, en este caso se obtiene una instancia de la clase System.Web.HttpSessionState, este objeto nos va a permitir almacenar información entre diferentes páginas ASP incluidas en una misma aplicación ASP .NET. La diferencia con el objeto Application se encuentra en el ámbito de las variables, cada variable del objeto Session es particular a una sesión de un usuario determinado, no a toda la aplicación. De esta forma, cada usuario tendrá sus variables y sus valores, sin dar lugar a problemas de concurrencia, tampoco se podrá acceder a distintas variables de sesión, cada usuario tiene su espacio de almacenamiento. Las variables de aplicación son valores globales y comunes a toda la aplicación, y las variables de sesión son particulares para cada usuario de la aplicación.
- **Trace**: esta propiedad ofrece una referencia a un objeto de clase System.Web.TraceContext, que nos va a permitir acceder a la información detallada de la ejecución de la página ASP

.NET actual. Este objeto lo comentaremos en detalle cuando nos ocupemos del tratamiento de errores y al utilización de trazas dentro de ASP .NET. El objeto Trace es un nuevo objeto de ASP .NET y no tiene ninguna correspondencia con los anteriores objetos del modelo de objetos de ASP.

- TraceEnabled: propiedad de tipo booleano que contendrá el valor true si está activado el mecanismo de trazas dentro de la página, cuando tratemos las directivas de la página veremos como activar las trazas dentro de una página ASP .NET.
- User: propiedad que devuelve información sobre el usuario que ha realizado la petición de la página ASP .NET.
- Validators: esta propiedad es una referencia a una colección de todos los controles Web de validación presentes en la página ASP .NET actual.

Una vez comentadas las propiedades y eventos principales de la clase Page nos queda comentar los métodos de dicha clase, para ello no hay nada más que pasar al siguiente apartado.

Métodos de la clase Page

A continuación vamos a pasar a describir brevemente algunos de los métodos que nos ofrece la clase Page:

- DataBind: este método, que ya lo hemos utilizado en algunos ejemplos en capítulos anteriores, establece el enlace de datos de todos los controles Web incluidos en la página ASP .NET actual. Este método no recibe ningún tipo de parámetro y no devuelve ningún tipo de valor.
- FindControl: este método realiza una búsqueda en la página del control Web cuyo nombre se pasa como parámetro, y si lo encuentra devuelve dicho control. Este método recibe como parámetro un objeto de la clase String que va a representar el nombre del control que deseamos localizar, y si el control se encuentra en la página devuelve el objeto correspondiente de la clase Control, sobre el que deberemos aplicar el mecanismo de casting adecuado para poder utilizar el control Web de la forma conveniente.
- LoadControl: carga de forma dinámica un control de usuario (en un futuro capítulo se comentará como crear controles definidos por el usuario). Como parámetro recibe un objeto de la clase String que representa la ruta del fichero .ASCX que va a ser el tipo de fichero dónde se definen los controles de usuario. Este método devuelve una instancia de dicho control de usuario mediante un objeto de la clase UserControl.
- MapPath: este método devuelve un objeto de la clase String que representa una ruta virtual construida a partir de la ruta física que se indica por parámetro, también como un objeto de la clase String. Este método tiene la misma función que el método MapPath del objeto Server de versiones anteriores de ASP.
- ResolveUrl: este método convierte una URL virtual, que le indicamos por parámetro, en una URL absoluta.
- Validate: método que no posee ningún parámetro y que tampoco devuelve ningún valor, cuando es lanzado se realizan las validaciones correspondientes de todos los controles Web de validación incluidos en la página ASP .NET.

En el siguiente apartado vamos a comentar otro aspecto importante de ASP .NET y que se encuentra estrechamente relacionado con la clase Page, se trata de las directivas que podemos utilizar en una página ASP .NET.

El atributo Code-Behind y los controles de usuario

Vamos a comentar las distintas posibilidades que nos ofrece ASP .NET para poder realizar una separación de código fuente frente al contenido o presentación, es decir, se pretende separar la lógica de la aplicación de la lógica de la presentación.

ASP .NET nos ofrece cuatro mecanismos que nos van a permitir realizar esta separación de código y contenido:

- Mediante el uso de ficheros Code-Behind, que son módulos precompilados escritos en cualquier lenguaje de la plataforma .NET Framework.
- Utilizando controles de usuario definidos en ficheros .ASCX.
- Creando componentes para el .NET Framework definidos en unidades denominadas assembly, los ensamblados ya se comentaron en el capítulo anterior.
- Construyendo nuestros propios controles de servidor de ASP .NET, al igual que en el caso anterior estos controles de servidor irán compilados en assemblies. Aunque la construcción de controles Web tiene que ver también con la reutilización de código y con la creación de controles de servidor visuales, que generan contenido, se ha creído conveniente incluir este mecanismo que nos ofrece ASP .NET en esta enumeración.

A lo largo del presente capítulo se comentarán los dos primeros mecanismos de separación de código, también se mostrarán diversos ejemplos de utilización. Y en el próximo capítulo veremos como realizar la construcción de componentes .NET y de controles de servidor de ASP .NET.

Code-Behind

Como ya hemos adelantado ASP .NET ofrece a las páginas un mecanismo denominado Code-Behind (código oculto). Este nuevo mecanismo nos permite separar el interfaz de usuario de la lógica de la aplicación, de esta forma el código fuente puede residir en un fichero separado, que puede ser una clase de Visual Basic .NET, C#, etc. Este código es el que se denomina Code-Behind, ya que permanece oculto. Las ventajas que obtenemos al utilizar este mecanismo son:

- La posibilidad de realizar una clara división del trabajo entre los programadores y los diseñadores.
- Permite a los desarrolladores o programadores utilizar sus entornos preferidos.
- Los autores de código HTML pueden utilizar sus herramientas de diseño de páginas Web.

Se debe señalar que un fichero Code-Behind es al fin y al cabo una clase definida en el lenguaje .NET correspondiente.

Para indicar que una página ASP .NET va a utilizar una clase determinada (como Code-Behind), se utilizan los atributos Inherits y Src de la directiva Page, esta directiva se veía con detenimiento en el capítulo anterior.

En el atributo Inherits debemos indicar el nombre de la clase Code-Behind y en el atributo Src la ruta del fichero de la clase. No es necesario hacer uso del atributo Src, si no indicamos la ruta al fichero que define la clase, en el caso del lenguaje C# se trata de un fichero con la extensión .CS, ASP .NET buscará la clase en el directorio bin de la aplicación.

Una vez indicado en la página ASP .NET la clase que se va a utilizar, mediante la directiva Page, únicamente nos queda definir e implementar la clase Code-Behind.

Vamos a realizar una clase muy sencilla en C# que simplemente muestra un saludo mediante un método. Para realizar la llamada de un método perteneciente a una clase Code-Behind desde una página ASP .NET únicamente debemos escribir el nombre del método como si estuviera definido en la propia página.

El aspecto que presenta una clase que va a ser utilizada desde una página ASP .NET es muy similar al resto de las clases tradicionales, lo que la hace particular es que debe importar una serie de NameSpaces correspondientes al entorno de ASP .NET. Estos espacios de nombres son System y System.Web, que nos van a permitir utilizar los objetos del entorno ASP .NET. Para importar los espacios de nombres se va a utilizar la instrucción using.

Y si también deseamos acceder a los controles Web definidos en la página ASP .NET desde la clase Code-Behind tenemos que importar los espacios de nombres necesarios, como puede ser System.Web.UI.WebControls.

La otra diferencia con las clases convencionales es que nuestra clase Code-Behind debe heredar de la clase System.Web.UI.Page de ASP .NET. Para indicar la clase de la que hereda nuestra clase Code-Behind, debemos situar a continuación del nombre de la clase actual el nombre de la clase padre separado mediante dos puntos (:). De esta forma el código de nuestra clase Code-Behind sería el mostrado en el Código fuente 175.

```
using System;
using System.Web;

namespace aplicacionWeb.Ejemplos
{
    public class ClaseSencilla: System.Web.UI.Page
    {
        public void muestraSaludo()
        {
            Response.Write("<b>¡Hola amigo, que tal...!</b><br>");
        }
    }
}
```

Código fuente 175

En este ejemplo hemos creado una clase en C# que hereda de la clase Page, se llama ClaseSencilla y se encuentra en el espacio con nombre aplicacionWeb.Ejemplos. Se ha definido un método público llamado muestraSaludo() que únicamente utiliza el método Write() de la propiedad Response de la clase. No se debe olvidar que tenemos acceso completo a la clase Page, ya que la clase que estamos definiendo es clase hija de la clase Page.

En el Código fuente 176 se muestra el código que contendría la página ASP .NET que desea utilizar la clase Code-Behind anterior, como ya hemos comentado, si copiamos el fichero .CS que contiene el código fuente en C# de la clase Code-Behind en el directorio bin especial de la aplicación ASP .NET actual, no tenemos que utilizar el atributo Src de la directiva Page. Más adelante en este texto veremos la estructura y configuración de una aplicación ASP .NET, de momento cabe destacar la importancia del directorio bin de la aplicación, que también será utilizado para definir los componentes de servidor.

```
<%@ Page Language="C#" Src="ClaseSencilla.cs"
Inherits="aplicacionWeb.Ejemplos.ClaseSencilla"%>
<html>
<head>
<script runat="server">
void Page_Load(Object objFuente,EventArgs args){
    muestraSaludo();
}
</script>
<title>Code-Behind</title>
</head>

<body>
Página que utiliza una clase Code-Behind
</body>

</html>
```

Código fuente 176

En el código de la página ASP .NET se ha asignado el valor ClaseSencilla.cs al atributo Src porque en este caso la clase Code-Behind se encuentra en el mismo directorio que la página. Si no deseamos utilizar este atributo podemos copiar la clase al directorio bin de la aplicación ASP .NET, pero para que todo funcione correctamente debemos seleccionar la opción de generar la aplicación ASP .NET (proyecto) dentro de Visual Studio .NET, en este caso si modificamos la clase Code-Behind, se debe volver a generar el proyecto, ya que en el directorio bin se debe encontrar la clase precompilada.

Como se puede comprobar en el código de la página ASP .NET a la hora de lanzar el método definido en la clase Code-Behind no debemos hacer nada especial, no es necesario compilar previamente la clase Code-Behind, al ejecutar la página ASP .NET que hace uso de la clase, se realiza la compilación de forma automática.

En la Figura 67 se puede ver el resultado de la ejecución de este ejemplo de utilización Code-Behind.

Utilizando los controles Web de la página

Este ejemplo de Code-Behind es muy sencillo y únicamente muestra un mensaje en el navegador, pero en otros casos nos puede interesar operaciones más complejas, como puede ser acceder a controles Web, desde la clase Code-Behind, definidos en una página ASP .NET.

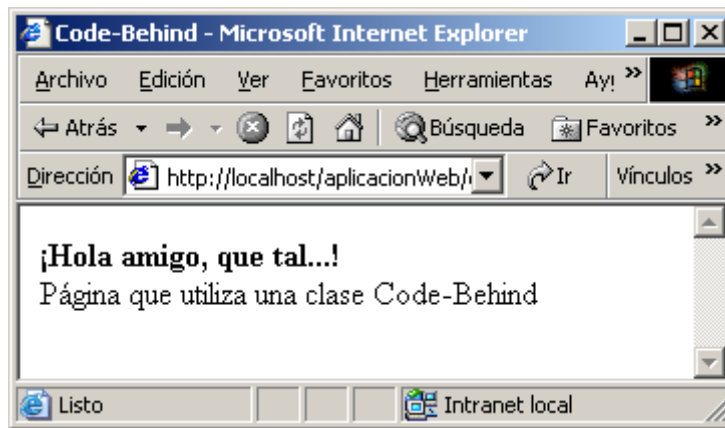


Figura 67

Vamos a realizar otro ejemplo de clase Code-Behind, en este caso se trata de ampliar la clase anterior añadiendo un método que muestre la hora y fecha actuales en un control Web de la clase Label de la propia página ASP .NET.

En este nuevo ejemplo, en nuestra clase Code-Behind debemos importar el Namespace `System.Web.UI.WebControls`, para así poder acceder a los controles Web que se encuentran en la página ASP .NET. Otra diferencia que resulta notable es que definimos una variable dentro de la clase que va a ser de la misma clase que el control Web que deseamos manipular, y además debe tener el mismo nombre que se le ha dado al control Web en la página ASP .NET.

Estas variables que van a corresponderse con controles Web de la página ASP .NET no es necesario instanciarlas, ya que esto lo va a hacer la propia página, no debemos olvidar que coincida el nombre de la variable con el del control Web.

El Código fuente 177 es el código de la clase Code-Behind que incorpora esta otra funcionalidad que nos permiten este tipo de clases.

```
using System;
using System.Web;
using System.Web.UI.WebControls;

namespace aplicacionWeb.Ejemplos
{
    public class ClaseSencillaDos: System.Web.UI.Page
    {
        public Label etiqueta;
        public void muestraSaludo()
        {
            Response.Write("<b>¡Hola amigo, que tal...!</b><br>");
        }

        public void muestraHora()
        {
            DateTime ahora=DateTime.Now;
            etiqueta.Text=ahora.ToString();
        }
    }
}
```

Código fuente 177

Y el Código fuente 178 es el código de la página ASP .NET que utiliza a la clase Code-Behind anterior, en este caso la única diferencia con el ejemplo anterior es el nombre de la clase de la que hereda la página, el nombre del fichero de código fuente y la llamada al nuevo método muestraHora().

```
<%@ Page Language="C#" Src="ClaseSencillaDos.cs"
Inherits="aplicacionWeb.Ejemplos.ClaseSencillaDos"%>
<html>
<head>
<script runat="server">
void Page_Load(Object objFuente,EventArgs args){
    muestraSaludo();
    muestraHora();
}
</script>
<title>Code-Behind</title>
</head>

<body>
Página que utiliza una clase Code-Behind
<br>
<form runat="server">
<asp:Label ID="etiqueta" runat="server"/>
</form>
</body>

</html>
```

Código fuente 178

Tratamiento de eventos

Podemos complicar algo más la clase Code-Behind, en esta nueva ampliación vamos a incluir el tratamiento de eventos de una pulsación (evento OnClick) de un botón que se encuentra en la página ASP .NET, al pulsar el botón se mostrará en el objeto Label la fecha y hora actuales.

En el Código fuente 179 se puede ver el nuevo aspecto que presentará la clase, se ha añadido el método pulsado() que recibe los parámetros adecuados para realizar el tratamiento de eventos del botón que se ha pulsado en la página ASP .NET.

```
using System;
using System.Web;
using System.Web.UI.WebControls;

namespace aplicacionWeb.Ejemplos
{
    public class ClaseSencillaTres: System.Web.UI.Page
    {
        public Label etiqueta;
        public void muestraSaludo()
        {
            Response.Write("<b>¡Hola amigo, que tal...!</b><br>");
        }

        private void muestraHora()
        {
            DateTime ahora=DateTime.Now;
            etiqueta.Text=ahora.ToString();
        }
    }
}
```

```
    }

    public void pulsado(Object sender, EventArgs args)
    {
        muestraHora();
    }
}
}
```

Código fuente 179

También se ha optado por cambiar el acceso que la página va a tener al método `muestraHora()`, en este ejemplo se ha indicado que este método es privado, por lo tanto únicamente se va a poder lanzar a través del método `pulsado()`.

La página ASP .NET presenta el Código fuente 180, como se puede observar para indicar el método que va a tratar el evento del control Web Button, se hace de la misma forma que cuando se encuentra el código para el tratamiento del evento en la misma página.

```
<%@ Page Language="C#" Src="ClaseSencillaTres.cs"
Inherits="aplicacionWeb.Ejemplos.ClaseSencillaTres"%>
<html>
<head>
<script runat="server">
void Page_Load(Object objFuente,EventArgs args){
    muestraSaludo();
}
</script>
<title>Code-Behind</title>
</head>

<body>
Página que utiliza una clase Code-Behind
<br>
<form runat="server" ID="Form1">
<asp:Label ID="etiqueta" runat="server"/>
<asp:Button ID="boton" Runat="server" OnClick="pulsado" Text="pulsa"/>
</form>
</body>
</html>
```

Código fuente 180

En la Figura 68 se puede ver el nuevo aspecto de la página ASP .NET del ejemplo.

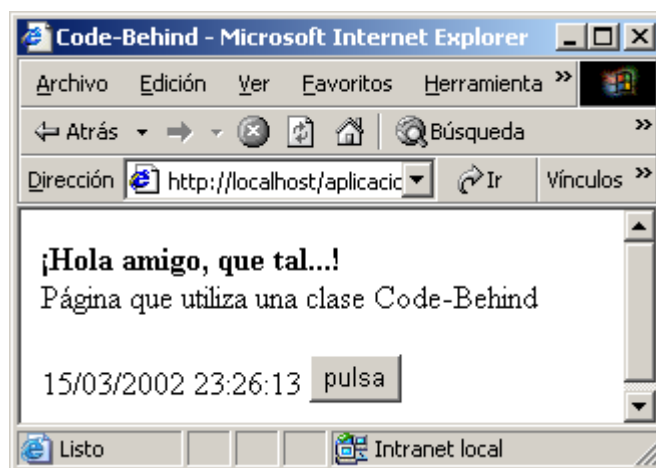


Figura 68

Controles de usuario

Los controles de usuario (user controls) pueden concebirse como páginas ASP .NET que serán utilizadas por otras páginas como controles de servidor. Los ficheros en los que se definan los controles de usuario poseen una extensión .ASCX, gracias a esta extensión se evita que el control de usuario pueda ser ejecutado de forma aislada como si se tratara de una extensión .ASPX.

Los controles de usuario nos puede servir para ampliar los controles ofrecidos por ASP .NET. Con una serie de cambios mínimos cualquier página ASP .NET puede convertirse en un control de usuario y así poder ser reutilizada por otra página.

Podríamos decir que los controles de usuario reemplazan de largo a los ficheros include de versiones anteriores de ASP, los controles de usuario van a intentar solventar los mismos problemas, como puede ser la creación de cabeceras, barras de navegación, pies, bloques de código repetitivos, etc., aunque ofrecen una funcionalidad mucho más avanzada que los ficheros de *include*.

Básicamente un control de usuario se compone de HTML y código, pero debido a que un control de usuario siempre va a estar incluido dentro de una página ASP .NET, no se pueden utilizar en los ficheros ASCX las etiquetas <html>, <body> y <form>.

Un control de usuario va a participar en el ciclo completo de la ejecución de cada petición de la página en la que se encuentran incluidos, y además pueden manejar sus propios eventos, encapsulando de esta forma la lógica frente a la página ASP .NET en la que se utilizan, así por ejemplo un control de usuario va a poder implementar su método Page_Load.

Vamos a comenzar por un ejemplo muy básico de control de usuario, se va a tratar de un simple texto para que sea incluido en las páginas que sean necesarias, en este caso se trata de un copyright que puede ser incluido en el pie de una página. El Código fuente 181 muestra el sencillo código que presenta el fichero ASCX que contiene al control, como se puede comprobar se trata de código HTML sencillo.

```
<hr>  
<font size=1>© 2002 Grupo EIDOS. Todos los derechos reservados.</font>
```

Código fuente 181

Una vez que tenemos definido nuestro control de usuario en el fichero ASCX correspondiente, para incluirlo en una página ASP .NET debemos hacer uso de la directiva Register, que ya comentamos en el capítulo anterior.

En la directiva Register debemos indicar un NameSpace para el control de usuario, es decir, el prefijo que se incluirá delante del nombre del control dentro de la página, esto se hace mediante el atributo TagPrefix de la directiva Register. Varios controles de usuario pueden pertenecer al mismo espacio de nombres, es decir, pueden tener el mismo valor para el atributo TagPrefix.

También debemos indicar un valor para el atributo TagName y Src. En el atributo TagName indicaremos el nombre del control, es decir, el nombre de la etiqueta que va a identificar al control de forma única, y en el atributo Src indicaremos la ruta virtual al fichero ASCX que contiene al control de usuario.

Una vez registrado el control vamos a poder utilizarlo como si se tratara de un control Web de ASP .NET, se debe indicar que la clase que representa al control de usuario es System.Web.UI.UserControl que hereda directamente de la clase System.Web.UI.Control. Cuando ejecutamos la página ASP .NET que contiene al control, el entorno de ejecución del .NET Framework compila el control de usuario y lo hace disponible para la página.

El Código fuente 182 es el que presentaría la página ASP .NET que hace uso del control de usuario definido anteriormente, en este caso el NameSpace del control es controles y su etiqueta es pie.

```
<%@ Page language="c#"%>
<%@ Register TagPrefix="controles" TagName="pie" Src="ControlPie.ascx"%>
<HTML>
<head><title>Controles de usuario</title></head>
<body>
<h1>Página ASP .NET que utiliza un control de usuario</h1>
<controles:pie runat="server" id="pieUno"/>
</body>
</HTML>
```

Código fuente 182

Y en la Figura 69 se puede observar el resultado de la ejecución de esta página ASP .NET.

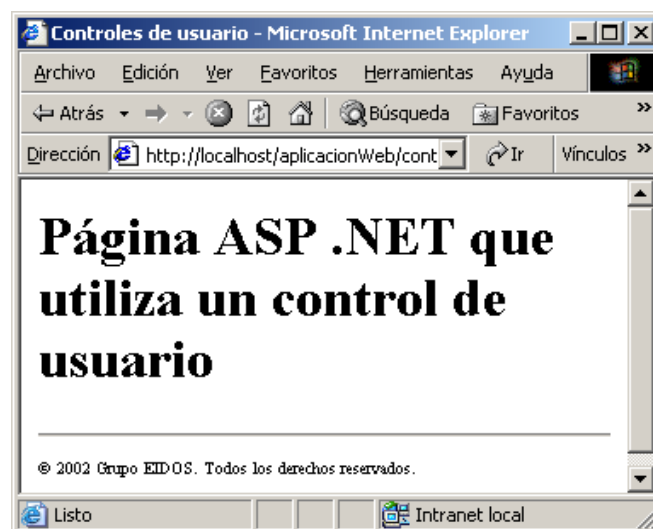


Figura 69

Si deseamos registrar más de un control de usuario en una misma página ASP .NET se deberá utilizar una directiva Register para cada uno de ellos, así si queremos incluir un nuevo control de usuario (Código fuente 183) que funcione como una cabecera deberemos escribir el Código fuente 184 para la página ASP .NET correspondiente.

```
<font size=1>El momento actual es: <%=DateTime.Now.ToString()%></font>
<hr>
```

Código fuente 183

```
<%@ Page language="c#"%>
<%@ Register TagPrefix="controles" TagName="pie" Src="ControlPie.ascx"%>
<%@ Register TagPrefix="controles" TagName="cabecera" Src="ControlCabecera.ascx"%>
<HTML>
<head><title>Controles de usuario</title></head>
<body>
<controles:cabecera runat="server" id="cabeceraUno"/>
<h1>Página ASP .NET que utiliza controles de usuario</h1>
<controles:pie runat="server" id="pieUno"/>
</body>
</HTML>
```

Código fuente 184

Como se puede apreciar en este nuevo ejemplo, el nuevo control de cabecera pertenece al mismo espacio de nombres que el control anterior, y en este caso se ha incluido código en C# para obtener la fecha y horas actuales.

El resultado se puede observar en la Figura 70.

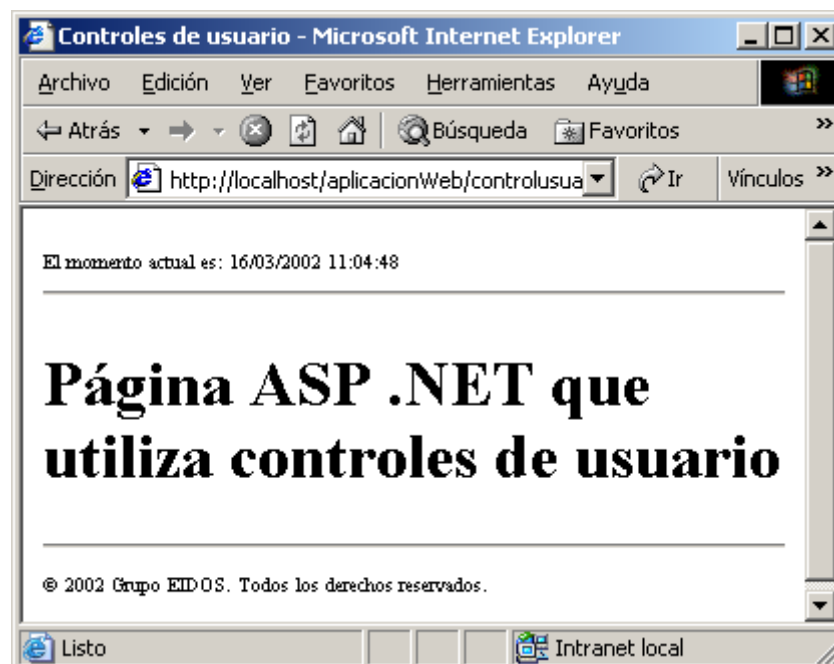


Figura 70

Vamos a realizar un ejemplo un poco más complejo de control de usuario, en este caso el control va a contener controles Web, como si de una página ASP .NET se tratara. El control de usuario va a contener un objeto Text con dos controles de validación, uno del tipo RequiredFieldValidator y otro del tipo RangeValidator. El control de usuario nos a requerir introducir un número entre 1 y 100. En el Código fuente 185 se puede ver como se ha construido este control de usuario.

```
<asp:TextBox Id="valor" runat="server"/><br>
<asp:RequiredFieldValidator id="validadorRequerido" runat="server"
    ControlToValidate="valor" Display="Dynamic"
    Font-Italic="True" Font-Name="Courier" Font-Size="7"
    ErrorMessage="Debe indicar un número">
</asp:RequiredFieldValidator>
<asp:RangeValidator id="validadorRango" runat="server"
    ControlToValidate="valor"
    ErrorMessage="Debe indicar un entero en el rango 1-100"
    Display="Dynamic" MinimumValue="1" MaximumValue="100" Type="Integer"
    Font-Italic="True" Font-Name="Courier" Font-Size="7">
</asp:RangeValidator>
```

Código fuente 185

Como se puede comprobar en el control de usuario no se ha utilizado ningún Web Form que contenga a los controles Web, será responsabilidad de la página ASP .NET que utilice el control de usuario incluirlo si es necesario dentro de un formulario. No debemos olvidar que no es posible realizar la anidación de formularios.

La página ASP .NET que utiliza este control (Código fuente 186), va a ser muy sencilla, únicamente contiene un formulario Web con un control Web de la clase Button y una instancia de nuestro control, al pulsar el botón se realizará la correspondiente validación, como se puede comprobar el tipo de validación que se realiza es transparente a la página ASP .NET.

```
<%@ Page language="c#" %>
<%@ Register TagPrefix="controles" TagName="validador"
Src="ControlValidacion.ascx" %>
<html>
<head><title>Controles de usuario</title></head>
<body>

<form id="formulario" method="post" runat="server">
Valor:<controles:validador Id="control" runat="server"/>
<br>
<asp:Button ID="boton" runat="server" Text="Enviar datos"></asp:Button>
</form>

</body>
</html>
```

Código fuente 186

En la Figura 71 se puede ver el resultado de la ejecución de esta página ASP .NET

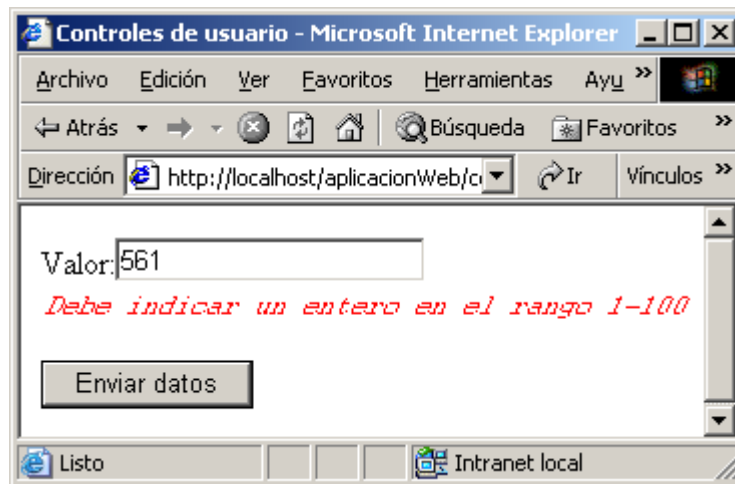


Figura 71

La directiva @Control

Los controles de usuario presentan una directiva, similar a la directiva Page, llamada Control, esta directiva únicamente se puede utilizar en los fichero ASCX y sólo puede existir una directiva Control por cada control de usuario.

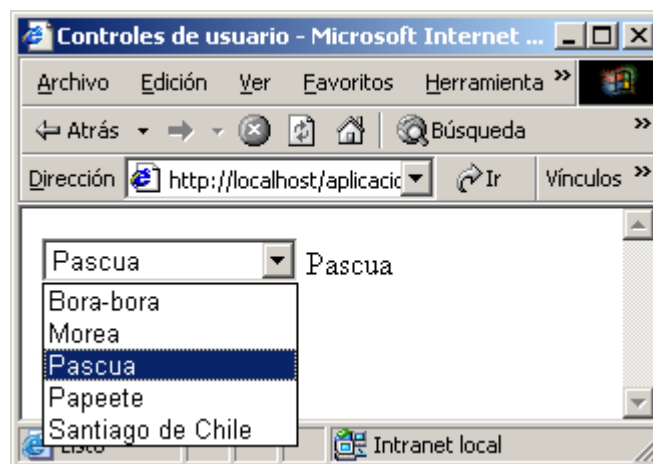


Figura 72

La directiva Control soporta algunos de los atributos que soportaba la directiva Page, algunos de estos atributos son `AspCompat` y `Trace`, para establecer las trazas para un control de usuario, se deben activar en la página ASP .NET que contiene a dicho control.

A continuación vamos a comentar los distintos atributos que presenta la directiva Control, vamos a ser muy breves ya que tienen la misma funcionalidad que en la directiva Page pero aplicados al control de usuario actual:

- `AutoEventWireup`: indica si los eventos (`UnLoad`, `Load`, `Init`, etc.) se van a lanzar de forma automática.
- `ClassName`: esta propiedad va a presentar como valor una cadena que indica el nombre de la clase del control actual.

- **CompilerOptions**: este atributo es una cadena que indica una lista de opciones para el compilador.
- **Debug**: permite activar la depuración para el control de usuario actual
- **Description**: cadena que muestra una descripción del control, se utiliza a modo de documentación.
- **EnableSessionState**: este atributo permite activar o desactivar el estado de sesión, es decir, permite o no la utilización de la propiedad Session de la clase Page para almacenar información común a la sesión actual del usuario con la aplicación Web.
- **EnableViewState**: permite activar o desactivar el mantenimiento automático de los valores de los controles Web dentro de un formulario, por defecto tiene el valor true.
- **Explicit**: mediante este atributo indicaremos si la declaración de variables es obligatoria o no, es equivalente al modo Option Explicit de Visual Basic.
- **Inherits**: este atributo indica el nombre de una clase de la que el control hereda, este atributo es utilizado para realizar la separación del código de la presentación de la información.
- **Language**: el lenguaje de la plataforma .NET utilizado en el control de usuario.
- **Src**: la ruta del fichero de la clase Code-Behind utilizada por el control.
- **Strict**: indica si se va utilizar el modo Option Strict de Visual Basic a la hora de compilar el control.
- **WarningLevel**: este atributo indica el nivel en el que la compilación será interrumpida, los valores que puede presentar son de 0 a 4.

Creación de componentes y controles de servidor

En primer lugar nos vamos a ocupar de la creación de componentes para la plataforma .NET Framework, veremos las diferencias y mejores que existen respecto al modelo de componentes de COM+, veremos también como construir componentes y como distribuirlos y utilizarlos dentro de páginas ASP .NET.

En segundo lugar, vamos a tratar la creación de controles de servidor de ASP .NET. Gracias a la creación de nuestros propios controles ASP .NET de servidor podemos encapsular interfaces personalizados y más funcionalidades en controles que pueden reutilizarse en las páginas ASP .NET. No se debe confundir la creación de controles ASP .NET de servidor con los controles de usuario que veíamos en el capítulo anterior. Veremos lo sencillo, en comparación con otros lenguajes, que resulta en ASP .NET escribir nuestros propios controles de servidor.

Introducción a los componentes .NET

Los componentes dentro de ASP .NET los vamos a utilizar para que contengan la lógica de negocio, es decir, para que sean objetos con lógica de negocio (business objects). Estos componentes van a contener bloques de código con una funcionalidad concreta.

Los componentes de software son unidades de código ejecutable que proporcionan una funcionalidad específica para una aplicación. A la aplicación que utiliza código de un componente, creando objetos y llamando a sus propiedades y métodos, se denomina cliente o contenedora.

Las ventajas derivadas de la utilización de los componentes en el desarrollo de software pueden compararse a las de la producción industrial en cadena: en vez de ser un único fabricante el que se encargue, por ejemplo, de la construcción de todas y cada una de las piezas de un coche, existen varios fabricantes, cada uno de ellos especializado en una parte concreta (motor, amortiguadores, motores eléctricos, chapa, caja de cambios, ...), que tienen funcionalidades diferentes. Cada uno de estos fabricantes trabaja de forma independiente a los demás, pero siguiendo unos estándares industriales para que luego las piezas encajen entre sí para formar una única unidad funcional: el coche.

Del mismo modo, la utilización de los componentes en una aplicación informática permite que varias personas distintas, de la misma o distinta empresa, trabajando con distintos lenguajes de programación, puedan crear pequeños bloques de código con una funcionalidad específica. Cada uno de estos componentes deben ajustarse a unos estándares o protocolos, para que luego puedan encajar entre sí y formar una aplicación con funcionalidad completa.

Pero en este capítulo no vamos a entrar en más detalles en lo que a teoría de componentes se refiere, sino que vamos a ponernos manos a la obra y vamos a ir viendo la creación de componentes .NET a través de ejemplos, aunque antes de hacer esto debemos tener en cuenta una serie de puntos acerca de los componentes .NET.

Un problema que tenía el antiguo modelo de componentes (Component Object Model, COM) para los componentes de una aplicación Web es que estos componentes debían ser registrados en el servidor antes de poder ser utilizados desde una aplicación ASP. A menudo la administración remota de estos componentes no resultaba posible, debido a que la aplicación que permitía el registro de los componentes debía ser local al servidor.

Otras de las dificultades que plantea el uso de componentes es que permanecen bloqueados en el disco después de ser utilizados por la aplicación Web correspondiente, y en muchos casos era necesario parar el servidor Web para poder liberar el componente y así poder ser reemplazado por otro o eliminado.

El siguiente paso de COM es COM+. COM+ es un conjunto de servicios que combina COM con Microsoft Transaction Server (MTS) en sistemas Windows 2000. De esta forma el servidor transaccional MTS ha dejado de existir como una entidad separada en Windows 2000, toda su funcionalidad es ahora una parte básica del sistema operativo.

En el sistema operativo Windows 2000 Server la administración de componentes se ha visto mejorada a través de la herramienta administrativa de Servicios de componentes, que permite configurar las aplicaciones COM+.

Los Servicios de componentes ofrecen las siguientes características que están relacionadas con el desarrollo de componentes en la versión 3.0 de ASP:

- Plena integración con Internet Information Server 5.0. Lo cual redundará en una serie de características relacionadas con la creación de aplicaciones ASP, como son:
 - Ejecución de secuencias de comandos de páginas ASP dentro de una transacción gestionada por Servicios de componentes.

- Protección contra errores de las aplicaciones Web de IIS. Al poder ejecutarse cada aplicación dentro de su propia aplicación COM+ de Servicios de componentes, se permite un aislamiento de los procesos implicados.
 - Inclusión de eventos transaccionales en las páginas ASP.
- Características relacionadas con la administración:
 - La utilidad explorador de Servicios de componentes, que permitirá administrar las aplicaciones COM+.
 - Cierre de aplicaciones COM+ de manera individual, sin necesidad de cerrar el proceso del servidor.
 - Configuración de la activación de aplicaciones COM+.
 - Desplazamiento de componentes entre aplicaciones COM+.
- Características relacionadas con la programación:
 - Se puede llevar a cabo, de manera automática, procedimientos administrativos como la instalación de aplicaciones COM+, mediante objetos de administración programables a través de secuencias de comandos.

Ahora en la plataforma .NET Framework y en ASP .NET se ha tratado de solventar todos los problemas y dificultades que planteaba el desarrollo de componentes en anteriores versiones de las tecnologías implicadas.

Para registrar un componente en ASP .NET debemos situarlo en un directorio determinado de la aplicación ASP .NET, los componentes se van a distribuir de una manera muy sencilla, simplemente se debe copiar al directorio BIN de la aplicación, este directorio debe encontrarse inmediatamente debajo del directorio raíz de la aplicación ASP .NET.

A continuación se comentan las ventajas de almacenar los componentes en el directorio BIN:

- No es necesario registrar el componente en el servidor Web. No es necesario el registro de componentes para hacerlos disponibles para una aplicación ASP .NET. Como ya hemos comentado los componentes se pueden distribuir copiando la DLL del componente (assembly) al directorio BIN de la aplicación correspondiente, esta operación puede ser realizada a través del protocolo FTP. En este punto cabe hacer una aclaración sobre el término DLL y assembly o ensamblado. Podemos establecer una analogía entre un ensamblado y una DLL, ya que ambos contienen clases, que se exponen a otras aplicaciones. Por dicho motivo, a un ensamblado también se le da el nombre de *DLL lógica*; el término *DLL* se emplea porque tiene un comportamiento similar al de las DLL's tradicionales, y el término *lógica* porque un ensamblado es un concepto abstracto, ya que se trata de una lista de ficheros que se referencian en tiempo de ejecución, pero que no se compilan para producir un fichero físico, a diferencia de lo que ocurre con las DLL's tradicionales.
- No es necesario reiniciar el servidor Web. Cuando cambiamos un componente reemplazando la antigua DLL del mismo por una nueva versión únicamente tenemos que sobrescribir el fichero y de forma automática la aplicación o aplicaciones ASP .NET que estén utilizando el componente comenzará a hacer uso de la nueva versión modificada del componente .NET. No es necesario reiniciar el servidor Web, ni el servicio Web ni descargar de memoria las aplicaciones ASP .NET que hacen uso del componente.

- No existen conflictos de espacios de nombre. Cada componente cargado desde el directorio BIN se encuentra limitado al ámbito de la aplicación que se está ejecutando. Esto quiere decir que muchas aplicaciones pueden potencialmente utilizar diferentes componentes con el mismo nombre de NameSpace o de clase, sin ocasionarse por ello ningún tipo de conflicto.

Al nivel más básico un componente va a ser simplemente una clase que podrá ser instanciada desde una página ASP .NET que importa el componente.

Podemos comparar el mecanismo de Code-Behind, que veíamos en el capítulo anterior, con al creación de componentes, de esta comparación se extraen las siguientes conclusiones:

- Las clases Code-Behind por defecto son ficheros sin compilar, aunque vimos en el capítulo anterior que podíamos compilarlas para que se incluyeran en un assembly que puede utilizar la aplicación ASP .NET.
- Las páginas ASP .NET utilizan a las clases Code-Behind a través del mecanismo de herencia, sin embargo los componentes se utilizan importándolos en las páginas correspondientes.
- Para utilizar una clase Code-Behind no hay que instanciarla, únicamente hay que heredar de ella, ya que gracias al mecanismo de herencia tendremos acceso a las propiedades y métodos de la clase padre. Sin embargo para hacer uso de un componente si es necesario realizar la instanciación el mismo, será sobre el objeto que es instancia de la clase del componente sobre el que lanzaremos los métodos correspondientes.
- Las clases Code-Behind se incluyen en una página ASP .NET mediante la directiva @Page, y los componentes se incluyen utilizando la directiva @Import, como si estuviéramos importando una clase o un NameSpace del .NET Framework.

Una vez realizada esta introducción más o menos exhaustiva sobre los componentes .NET vamos a pasar al siguiente apartado, en el que vamos a crear un sencillo componente para ASP .NET.

Creación de controles ASP .NET de servidor

Mediante al construcción de nuestros propios controles ASP .NET de servidor podemos encapsular la lógica y el interfaz de usuario para que puedan ser reutilizados en distintas páginas ASP .NET, como ya hemos comentado no se deben confundir los controles ASP .NET de servidor con los controles de usuario, ofrecen unos mecanismos distintos y se construyen de maneras muy diferentes, un control de usuario se va a construir normalmente a partir de una página ASP .NET, y un control de servidor se va a construir a partir de una clase.

A diferencia de otros entornos como puede ser C/C++, la creación de controles visuales personalizados en ASP .NET resulta muy sencilla. Y como el movimiento se demuestra andando, vamos a crear nuestro primer control ASP .NET de servidor, este control de servidor únicamente va a mostrar un texto.

Para crear un control de servidor sencillo debemos crear una clase que herede de la clase System.Web.UI.Control y que sobrescriba su método Render(). El método Render() recibe como parámetro un objeto de la clase System.Web.UI.HtmlTextWriter, sobre este objeto lanzaremos el método Write() pasándole por parámetro una cadena (String) el código HTML que deseamos que genere nuestro control ASP .NET de servidor.

El método `Render()` se ejecutará cuando se instancie el control en la página ASP .NET, y generará el código que será mostrado al cliente en la página. El proceso de traducción o rendering del control es una de las fases del ciclo de vida de un control de servidor.

Nuestra clase que va a dar lugar al control de servidor también debe importar los espacios de nombre que resulten necesarios, normalmente se deben importar los siguientes:

- **System:** este Namespace contiene las clases básicas del sistema, como puede ser la clase `String`. El código del control de servidor que vamos a realizar en el ejemplo podrá compilar sin este espacio de nombres, pero es recomendable incluirlo siempre.
- **System.Web:** este es el espacio de nombres padre para todas las clases de ASP .NET. Contiene clases como `HttpRequest`, que se corresponde con el objeto `Request` de ASP .NET, y `HttpResponse`, que se corresponde con el objeto `Response` de ASP .NET. De nuevo nuestro código podría compilar sin este Namespace, pero es una buena práctica incluirlo, ya que en otros casos si que será necesario.
- **System.Web.UI:** este Namespace contiene las clases de los controles ASP .NET, las cuales se encuentran divididas en espacios de nombres atendiendo a su familia (`System.Web.UI.WebControls`, `System.Web.UI.HtmlControls`, etc.). Esta referencia es necesaria, ya que estamos utilizando las clases `Control` y `HtmlTextWriter`.

En el Código fuente 187 se ofrece el código de la clase que nos va a permitir crear un control ASP .NET de servidor.

```
using System;
using System.Web;
using System.Web.UI;

namespace Controles.Ejemplos
{
    public class Texto:Control
    {
        protected override void Render(HtmlTextWriter salida)
        {
            salida.Write("<h1>Control ASP .NET de servidor</h1>");
        }
    }
}
```

Código fuente 187

Como se puede observar se ha utilizado la palabra reservada `override` para indicar que deseamos sobrescribir el método `Render()` de la clase `Control` de la que heredamos.

Una vez tenemos la clase de nuestro control debemos compilarla para que se genere en el assembly correspondiente, al igual que sucedía en el apartado anterior, a la hora de compilar la clase de un componente .NET, también tenemos las mismas opciones para compilar la clase de nuestro control ASP .NET de servidor. Cuando se ha generado el assembly de nuestro control de servidor lo debemos copiar al ya conocido directorio `BIN` de la aplicación ASP .NET en la que deseamos utilizar el control ASP .NET de servidor.

Para hacer uso del control de servidor en una página debemos utilizar la directiva `@Register`, esta misma directiva la utilizábamos en el capítulo anterior para registrar los controles de usuario en una página ASP .NET, pero ahora la vamos a utilizar para registrar controles ASP .NET de servidor.

En el atributo `TagPrefix` indicaremos la etiqueta que precederá al nombre de la clase del control de servidor cuando queramos utilizarlo en la página ASP .NET, en el atributo `NameSpace` indicamos el espacio de nombres del control y en la propiedad `Assembly` el nombre del fichero .DLL que contiene assembly, pero sin extensión. Como se puede ver en el Código fuente 188 para hacer referencial al control ASP .NET de servidor utilizamos la sintaxis `<TagPrefix:NombreClase>`.

```
<%@ Page language="C#"%>
<%@ Register TagPrefix="Controles" NameSpace="Controles.Ejemplos"
Assembly="ControlesServidor"%>
<html>
<head><title>Controles ASP .NET de servidor</title></head>
<body>
<controles:Texto runat="server"/>
</body>
</html>
```

Código fuente 188

En la Figura 73 se puede ver el resultado de la ejecución de la página ASP .NET que hace uso del control que hemos construido.

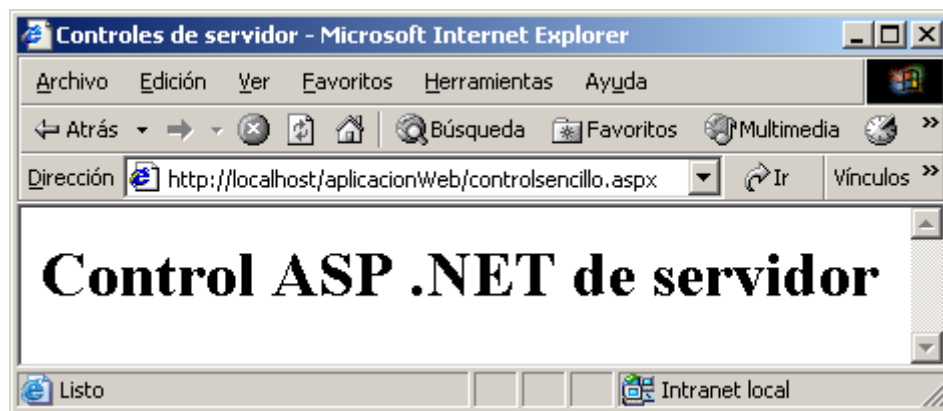


Figura 73

Y el Código fuente 189 es el código HTML que se generaría.

```
<html>
<head><title>Controles de servidor</title></head>
<body>
<h1>Control ASP .NET de servidor</h1>
</body>
</html>
```

Código fuente 189

Ya hemos visto con un sencillo ejemplo los distintos pasos que hay que seguir para construir un control ASP .NET de servidor, en los siguientes subapartados vamos a ir viendo aspectos más avanzados relacionados con la construcción de controles de servidor.

Propiedades de los controles de servidor

Para que un control de servidor sea útil debe permitir que el desarrollador de las páginas ASP .NET pueda tener algún tipo de influencia sobre la forma en la que el control se presenta en la página. En la forma más común para realizar esta tarea es mediante atributos.

La creación de atributos a través de propiedades de la clase del control ya lo vimos para el caso de los controles de usuario, para los controles ASP .NET de servidor va a ser muy similar.

Cuando el .NET Framework ejecuta una página ASP .NET y procesa los atributos de un control los hace corresponder con propiedades públicas de la clase en la que se encuentra definido el control de servidor. Los valores de las propiedades se establecen en la fase de inicialización de la página.

Durante la fase de inicialización de la página, una vez que las propiedades del control se han establecido con los valores de los atributos indicados en las etiquetas, se llama al método OnInit() de cada control. En el método OnInit() del control podemos validar las propiedades o realizar otras operaciones como las de establecer los valores por defecto para las propiedades.

Vamos a crear un control muy similar al anterior, pero en este caso va a presentar una propiedad de la clase String llamada Texto, que nos va a permitir indicar al control el texto que deseamos mostrar. Para implementar la propiedad debemos utilizar la sintaxis de propiedades del lenguaje C# tal como hacíamos en otros ejemplos.

El Código fuente 190 es el código del nuevo control ASP .NET de servidor.

```
using System;
using System.Web;
using System.Web.UI;

namespace Controles.Ejemplos
{
    public class Etiqueta:Control
    {
        private String _texto;

        public String Texto
        {
            get
            {
                return _texto;
            }

            set
            {
                _texto=value;
            }
        }

        protected override void OnInit(EventArgs evento)
        {
            base.OnInit(evento);
            if(_texto==null)

```

```
        {  
            _texto="Texto por defecto";  
        }  
    }  
  
    protected override void Render(HtmlTextWriter salida)  
    {  
        salida.Write("<h3>" + _texto + "</h3>");  
    }  
}
```

Código fuente 190

Como se puede comprobar este control se encuentra en el mismo espacio de nombres que el del ejemplo anterior. Otro aspecto a destacar es la sobrescritura del método `OnInit()`, como ya hemos comentado utilizaremos este método para realizar tareas de inicialización de propiedades y de comprobación y validación de los valores de las mismas.

En el método `OnInit()` se comprueba si la propiedad `Texto` tiene algún valor, si no posee ningún valor se le asigna un texto por defecto. En este método también se hace referencia a la clase padre, la clase `Control`, a través de la palabra reservada `base`, y sobre la clase padre se lanza el método `OnInit()`, ya que lo estamos sobrescribiendo en la clase actual.

En el método `Render()` de nuestra nueva clase se escribe en el objeto `HtmlTextWriter` la propiedad `Texto` entre las etiquetas HTML correspondientes. Como se puede observar la clase internamente utiliza una variable privada para manejar la propiedad `Texto`.

Realizamos las labores ya conocidas sobre la clase del componente para hacerla disponible en nuestra aplicación ASP .NET, es decir, se compila y se copia el assembly al directorio BIN de la aplicación. En este caso vamos a crear en la página ASP .NET (Código fuente 191) tres controles de servidor, uno de ellos va a mostrar el valor por defecto de su propiedad `Texto`, y los otros dos mostrarán el valor especificado en la misma.

```
<%@ Page language="C#" %>  
<%@ Register TagPrefix="Controles" Namespace="Controles.Ejemplos"  
Assembly="ControlesServidor"%>  
<html>  
<head><title>Controles de servidor</title></head>  
<body>  
<controles:Etiqueta runat="server" ID="Etiqueta1"/>  
<controles:Etiqueta runat="server" ID="Etiqueta2" Texto="Hola"/>  
<controles:Etiqueta runat="server" ID="Etiqueta3" Texto="Mundo"/>  
</body>  
</html>
```

Código fuente 191

En la Figura 74 se puede ver el resultado de la ejecución de esta página ASP .NET.

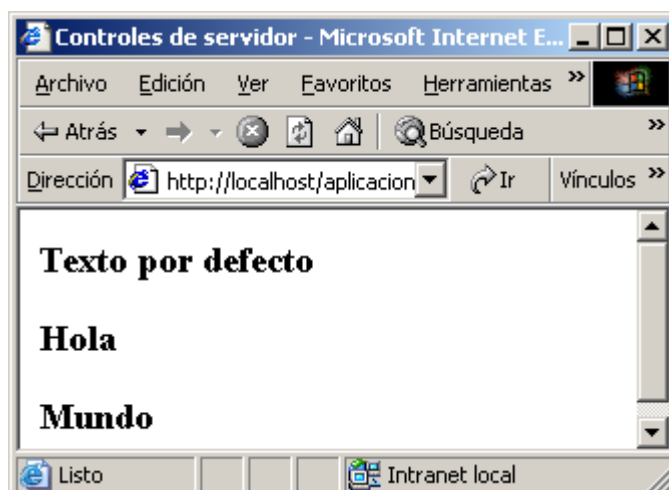


Figura 74

Propiedades de clase

Las propiedades de clase son aquellas propiedades definidas en la clase de un control ASP .NET de servidor que son objetos de otra clase. A su vez, estos objetos tienen propiedades, que se denominan subpropiedades.

Para acceder a estas propiedades desde las etiquetas de los controles de servidor de una página ASP .NET se debe seguir una sintaxis determinada, esta sintaxis consiste en separar las subpropiedades de las propiedades mediante un guión (-), y también tenemos la posibilidad de utilizar elementos anidados dentro de las etiquetas principales del control ASP .NET servidor, a continuación veremos estas dos posibilidades mediante un ejemplo.

Vamos a partir de la ya conocida clase que define una etiqueta como control de servidor, en este caso nuestra clase va a tener dos propiedades simples como son Repetición y Texto (con el mismo significado que en ejemplos anteriores), pero además va a tener una propiedad llamada Formato de la clase Formateador, que se encuentra definida en el mismo fichero .CS.

La propiedad Formato, objeto de la clase Formateador, posee dos subpropiedades denominadas Color y Tamaño, que respectivamente se van a utilizar para definir el color y tamaño de la etiqueta de texto del control.

En el Código fuente 192 se ofrece el código completo del fichero .CS que va a contener a la clase Formateador y a la clase EtiquetaCuatro. Como se puede observar en este código la definición de la propiedad Formato de la clase EtiquetaCuatro no posee ninguna dificultad ni ninguna diferencia respecto al resto de las propiedades definidas en dicha clase. La única diferencia es que la propiedad no tiene un método set asociado. Para indicar los valores por defecto de la propiedad Formato se ha utilizado el constructor de la clase a la que pertenece.

```
using System;
using System.Web;
using System.Web.UI;
using System.Drawing;

namespace Controles.Ejemplos
{
    public class Formateador
```

```

    {
        private int _tamaño;
        private String _color;

        public Formateador(int _tamaño, System.String _color)
        {
            this._tamaño=_tamaño;
            this._color=_color;
        }
        public String Color
        {
            get {return _color;}
            set {_color = value;}
        }
        public int Tamaño
        {
            get {return _tamaño;}
            set { _tamaño= value;}
        }
    }

    [ParseChildren(true)]
    public class EtiquetaCuatro:Control
    {
        private String _texto;
        private int _repetición=1;
        private Formateador _formato=new Formateador(2,"Orange");

        public Formateador Formato
        {
            get{return _formato;}
        }

        public String Texto
        {
            get{return _texto;}
            set{_texto=value;}
        }

        public int Repetición
        {
            get{return _repetición;}
            set{_repetición=value;}
        }

        protected override void OnInit(EventArgs evento)
        {
            base.OnInit(evento);
            if(_texto==null)
            {
                _texto="Texto por defecto";
            }
        }

        protected override void Render(HtmlTextWriter salida)
        {
            int i;
            for(i=0;i<_repetición;i++)
            {
                salida.AddAttribute("color",Formato.Color);
                salida.AddAttribute("size",Formato.Tamaño.ToString());
                salida.RenderBeginTag("font");
                salida.Write(_texto);
                salida.RenderEndTag();
                salida.RenderBeginTag("br");
            }
        }
    }

```

```
}
}
```

Código fuente 192

En el código del fichero .CS se puede observar que antes de la declaración de la clase del control de servidor se ha indicado el atributo ParseChildren asignándole el valor true, esto es necesario para que sea posible utilizar uno de los tipos de sintaxis de acceso a las subpropiedades, en concreto se trata de la sintaxis que permite anidar las subpropiedades con sus atributos correspondientes, en el ejemplo se verán los dos tipos de sintaxis.

Otro punto interesante del código del ejemplo es la utilización de los servicios de presentación del objeto HtmlTextWriter para generar la etiqueta font y añadirle dos propiedades, como se puede ver para la etiqueta de salto de línea (etiqueta
 de HTML) sólo se ha aplicado el método de apertura de etiqueta, ya que esta etiqueta no posee cierre.

Una vez visto el código del control ASP .NET de servidor vamos a comentar la forma en la que se utilizaría dentro de una página ASP .NET. Como ya hemos adelantado, para hacer referencia a las subpropiedades del control EtiquetaCuatro (propiedades de la clase Formateador), disponemos de dos posibilidades.

La primera de estas posibilidades consiste en separar mediante un guión las propiedades de las subpropiedades, así para acceder a la propiedad Color tendríamos que escribir el siguiente atributo dentro de la etiqueta del control: Formato-Color.

La otra posibilidad es la de anidar la propiedad Formato dentro de las etiquetas de apertura y cierre del control y en la etiqueta de la propiedad Formato acceder a las subpropiedades como atributos de dicha etiqueta, esta otra sintaxis es la que requiere que el atributo ParseChildren de la clase del control ASP .NET de servidor posea el valor true.

En el Código fuente 193 se muestra distintos usos del control EtiquetaCuatro, en el primero se utilizan los valores por defecto de las propiedades y subpropiedades, en el segundo se indican las subpropiedades utilizando la sintaxis de separación mediante guiones y en el tercero se anidan las etiquetas del control de servidor y de su propiedad de clase.

```
<%@ Page language="C#" %>
<%@ Register TagPrefix="Controles" Namespace="Controles.Ejemplos"
Assembly="ControlesServidor"%>
<html>
<head><title>Controles de servidor</title></head>
<body>
<controles:EtiquetaCuatro runat="server" ID="Etiqueta1"/>
<controles:EtiquetaCuatro runat="server" ID="Etiqueta2"
Repetición="3" Texto="Hola" Formato-Color="Green" Formato-Tamaño="6"/>

<controles:EtiquetaCuatro runat="server" ID="Etiqueta3" Repetición="2"
Texto="Mundo">
    <Formato Color="Red" Tamaño="4"/>
</controles:EtiquetaCuatro>
</body>
</html>
```

Código fuente 193

En la Figura 75 se puede ver el resultado de la ejecución de esta página.



Figura 75

El Código fuente 194 es el código HTML que genera la ejecución de la página ASP .NET.

```
<html>
<head><title>Controles de servidor</title></head>
<body>
<font color="Orange" size="2">Texto por defecto</font><br>

<font color="Green" size="6">Hola</font><br>
    <font color="Green" size="6">Hola</font><br>
    <font color="Green" size="6">Hola</font><br>

<font color="Red" size="4">Mundo</font><br>
    <font color="Red" size="4">Mundo</font><br>

</body>
</html>
```

Código fuente 194

Trazas y depuración en ASP .NET

El mecanismo de trazas

Este mecanismo es una de las nuevas características que nos ofrece ASP .NET, en versiones anteriores el mecanismo más común de trazas era la utilización de la sentencia `Response.Write`, pero ASP .NET nos ofrece un mecanismo de trazas automático.

Las trazas se pueden habilitar o deshabilitar a nivel una página ASP .NET determinada o bien al nivel de una aplicación ASP .NET completa, en este último caso deberemos acudir de nuevo al ya cada vez más conocido fichero de configuración de la aplicación, es decir, al fichero Web.config.

A continuación vamos a ver las dos posibilidades que nos ofrece ASP .NET a la hora de utilizar trazas.

Trazas a nivel de página

Para hacer uso del mecanismo de trazas dentro de una página ASP .NET determinada no tenemos nada más que asignar el valor True al atributo Trace de la directiva @Page.

En el Código fuente 195 se ofrece una página ASP .NET muy sencilla, que únicamente pretende mostrar la utilización de las trazas dentro de una página.

```
<%@ Page Trace="True" %>
<html>
<head>
<title>Trazas</title>
</head>
<body>
<b>Se activan las trazas.</b>
</body>
</html>
```

Código fuente 195

La ejecución de esta página se puede ver en la Figura 76.

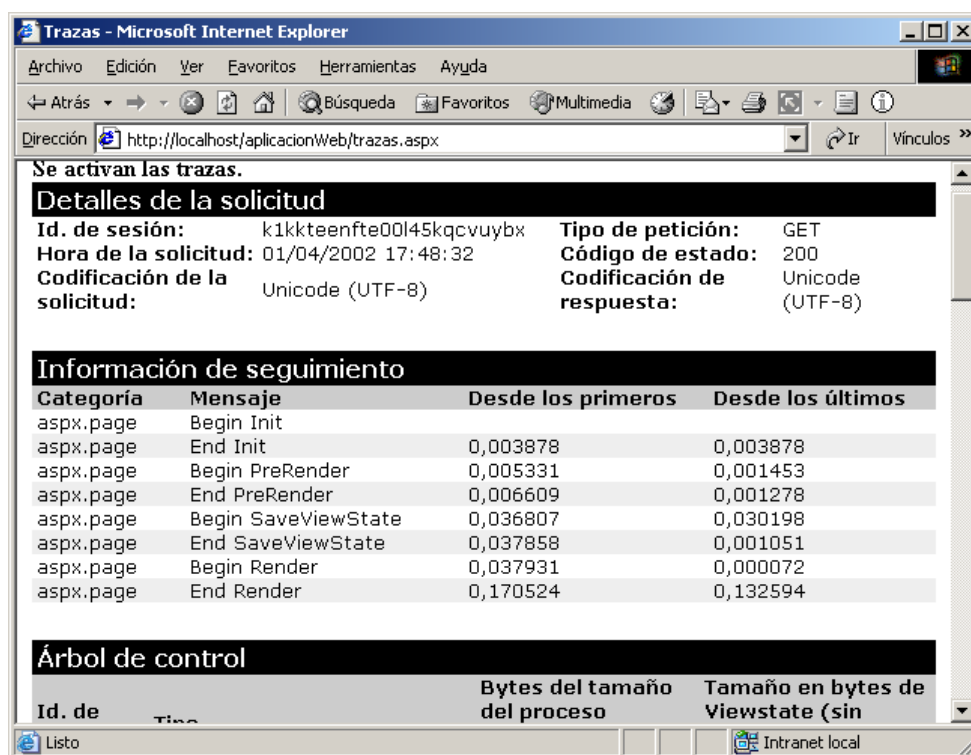


Figura 76

Como se puede comprobar, el entorno del .NET Framework nos ofrece una gran cantidad de información de forma automática con sólo activar el mecanismo de trazas. Esta información se encuentra dividida en varias secciones, que vamos a comentar a continuación:

- **Detalles de la solicitud (Request Details):** esta sección ofrece información sobre la petición realizada a la página ASP .NET, alguna de esta información es el tipo de petición, el código de estado, la hora de la petición, el identificador de sesión, etc.
- **Información de seguimiento (Trace Information):** en esta sección se puede observar el orden de ejecución de la página ASP .NET, se indica el tiempo empleado en la ejecución de cada una de las fases de la página ASP .NET. Si añadimos mensajes personalizados en nuestras trazas (algo que veremos en breve), parecerá también en esta sección.
- **Árbol de control (Control Tree):** aquí aparece una lista de la jerarquía de todos los controles incluidos en la página ASP .NET actual. Para cada control se indica el tamaño que ocupa en bytes.
- **Estado de sesión (Session State):** indica los detalles de cada elemento presente en el estado de la sesión actual. El objeto Session se verá más adelante en este mismo texto. Si no existe ningún elemento almacenado en la sesión esta sección no aparecerá.
- **Estado de aplicación (Application State):** esta sección tiene un cometido similar al anterior, pero este caso para los elementos almacenados en la aplicación.
- **Colección de cookies (Cookies Collection):** esta sección muestra los detalles de cada cookie existente en el momento actual.
- **Colección de encabezados (Headers Collection):** esta sección contiene los encabezados del protocolo HTTP utilizados para realizar la petición de la página.
- **Colección de formularios (Forms Collection):** contiene el nombre y valor de cada uno de los campos de los un formulario Web, sino existe ningún Web Form en la página esta sección no aparecerá.
- **Colección del QueryString (QueryString Collection):** contiene el nombre y valor de cada uno de los contenidos de la cadena de consulta (QueryString) que se ha utilizado para solicitar la página ASP .NET, si no existe el QueryString esta sección no aparecerá como información de traza de la página.
- **Variables del servidor (Server Variables):** esta última sección ofrece una completa lista de las variables de servidor y su contenido correspondiente.

Para desactivar las trazas de una página, únicamente debemos asignar el valor False al atributo Trace de la directiva @Page, o simplemente no indicar este atributo de la directiva @Page, ya que False es su valor por defecto.

En el siguiente apartado vamos a comentar como se puede escribir información en las trazas de una página ASP .NET.

Escribiendo en las trazas

Además de la información automática que nos ofrece ASP .NET mediante la activación del mecanismo de trazas, también podremos añadir nuestra propia información de trazas a la sección de Información de seguimiento.

Esto se puede hacer gracias a la propiedad Trace del objeto Page, esta propiedad es un objeto de la clase System.Web.TraceContext. Esta clase nos ofrece el método Write() para escribir información en las trazas de la página.

En el Código fuente 196 se puede ver una página ASP .NET que escribe información en las trazas indicando el evento de la página que se está produciendo en cada caso.

```
<%@ Page Language="C#" Trace="True" %>
<html>

<script runat="server">
void Page_Init(Object sender, EventArgs e) {
    Page.Trace.Write("Evento Init");
}

void Page_Load(Object sender, EventArgs e) {
    Page.Trace.Write("Evento Load");
}

void Page_PreRender(Object sender, EventArgs e) {
    Page.Trace.Write("Evento PreRender");
}
</script>

<head>
<title>Trazas</title>
</head>
<body>
<asp:Label runat="server" id="texto">Eventos de la página con trazas</asp:Label>
</body>
</html>
```

Código fuente 196

Estos mensajes se pueden apreciar en la sección Información de seguimiento tal como se puede comprobar en la Figura 77, que es un fragmento del resultado de la ejecución del código anterior.

Información de seguimiento			
Categoría	Mensaje	Desde los primeros	Desde los últimos
aspx.page	Begin Init		
	Evento Init	0,000063	0,000063
aspx.page	End Init	0,000105	0,000042
	Evento Load	0,000139	0,000034
aspx.page	Begin PreRender	0,000173	0,000034
	Evento PreRender	0,000206	0,000033
aspx.page	End PreRender	0,000239	0,000033
aspx.page	Begin SaveViewState	0,000458	0,000219
aspx.page	End SaveViewState	0,000492	0,000034
aspx.page	Begin Render	0,000523	0,000031
aspx.page	End Render	0,000792	0,000269

Figura 77

La propiedad Trace del objeto Page nos va a ofrecer acceso al contexto de las trazas de la página ASP .NET actual, para ello la clase TraceContext, de la que es instancia la propiedad Trace, nos ofrece dos propiedades y dos métodos que son los siguientes:

- **IsEnabled**: esta propiedad de tipo booleano indica si el mecanismo de trazas se encuentra activado en la página ASP .NET. Es una propiedad de lectura/escritura y se corresponde con el valor del atributo Trace de la directiva @Page.
- **TraceMode**: propiedad que indica la forma de ordenación de los mensajes que aparecen dentro de la sección Información de seguimiento, esta propiedad puede tener los valores SortByCategory (ordenación por categoría) y SortByTime (ordenado por el momento en el que se ha producido). Esta propiedad se corresponde también con un atributo de la directiva @Page, en este caso con el atributo TraceMode. Su valor por defecto es SortByTime.
- **Write**: este método, ya utilizado en el ejemplo anterior, permite escribir información en la sección Información de seguimiento.
- **Warn**: este método tiene la misma funcionalidad que el anterior, pero en este caso los mensajes aparecen en color rojo.

Los métodos Write() y Warn() se encuentran sobrecargados, presentando los parámetros del Código fuente 197 en cada una de sus versiones.

```
Trace.Write(Mensaje)
Trace.Write(Categoría, Mensaje)
Trace.Write(Categoría, Mensaje, InformaciónError)

Trace.Warn(Mensaje)
Trace.Warn(Categoría, Mensaje)
Trace.Warn(Categoría, Mensaje, InformaciónError)
```

Código fuente 197

A continuación pasamos a comentar los distintos parámetros de estos métodos.

- **Categoría**: cadena que se corresponde con la categoría que se le va a signar al mensaje. Esta categoría nos sirve para describir y agrupar mensajes que poseen un significado similar.
- **Mensaje**: cadena que representa el mensaje que se va a mostrar en la sección Información de seguimiento.
- **InformaciónError**: objeto de la clase System.Exception, permite mostrar información relativa a una excepción que se haya producido, esta información aparecerá a continuación del mensaje de la traza.

En el Código fuente 198 se ofrece una página ASP .NET que utiliza a modo de resumen práctico los métodos y propiedades de la clase TraceContext. Además esta página muestra como se puede activar y desactivar de forma dinámica el mecanismo de trazas.

```
<%@ Page Language="C#" %>
<html>
```

```

<script runat="server">
void Page_Load(Object sender, EventArgs e) {
    if (activar.Checked) {
        Trace.IsEnabled=true;
    }else{
        Trace.IsEnabled=false;
    }
    if (orden1.Checked) {
        Page.Trace.TraceMode=TraceMode.SortByCategory;
    }else if (orden2.Checked) {
        Page.Trace.TraceMode=TraceMode.SortByTime;
    }
    Page.Trace.Write("Eventos de la página","Evento Load");
}

void Page_PreRender(Object sender, EventArgs e) {
    Page.Trace.Write("Eventos de la página","Evento PreRender");
}

void PulsadoOrden(Object Sender, EventArgs e) {
    Trace.Warn("Eventos de los controles","Se ha pulsado un RadioButton");
}

void PulsadoActivar(Object Sender, EventArgs e) {
    Trace.Warn("Eventos de los controles","Se ha pulsado el CheckBox");
}

</script>

<head>
<title>Trazas</title>
</head>

<body>
<asp:Label runat="server" id="texto">Eventos de la página con
trazas</asp:Label><br>

<form runat="server" ID="Formulario">
<asp:RadioButton id="orden1" Text="Ordenar por categoría" runat="server"
GroupName="orden" OnCheckedChanged="PulsadoOrden" AutoPostBack="True"/><br>
<asp:RadioButton id="orden2" Text="Ordenar por tiempo" runat="server"
GroupName="orden" OnCheckedChanged="PulsadoOrden" AutoPostBack="True"/><br>
<asp:CheckBox id="activar" Text="Activar trazas" runat="server"
OnCheckedChanged="PulsadoActivar" AutoPostBack="True"/>
</form>

</body>
</html>

```

Código fuente 198

Aunque el mecanismo de trazas se encuentre desactivado no es necesario eliminar el código que muestra información de trazas, es decir, la llamada a los métodos Write() y Warn() del objeto Trace no generará ningún error aunque las trazas se encuentren desactivadas, únicamente estos mensajes no se tendrán en cuenta.

La página ASP .NET del ejemplo muestra un formulario como el de la Figura 78, que permite indicar el orden en el que se va a mostrar la información presente en la sección Información de seguimiento, y además de estos objetos RadioButton también se ofrece un objeto CheckBox que permite activar y desactivar el mecanismo de trazas de la página.

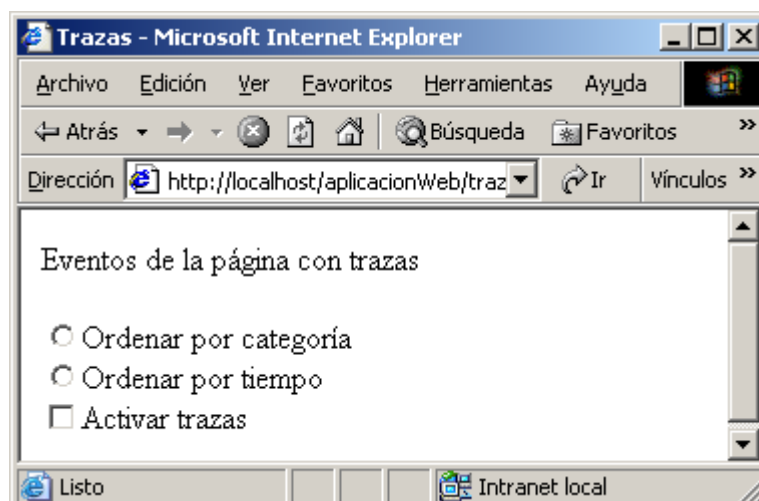


Figura 78

Para mostrar información relativa a los eventos de la página se utiliza el método Write(), y para los eventos de los controles se utiliza el método Warn().

En la Figura 79 se puede ver un momento en la ejecución de esta página ASP .NET, en este caso se ha seleccionado que la información de las trazas se ordene por su categoría, y evidentemente en este caso se ha activado el mecanismo de trazas.

Información de seguimiento			
Categoría	Mensaje	Desde los primeros	Desde los últimos
aspx.page	Begin ProcessPostData Second Try	0,000069	0,000069
aspx.page	End ProcessPostData Second Try	0,000144	0,000075
aspx.page	Begin Raise ChangedEvents	0,000181	0,000037
aspx.page	End Raise ChangedEvents	0,000261	0,000039
aspx.page	Begin RaisePostBackEvent	0,000297	0,000036
aspx.page	End RaisePostBackEvent	0,000339	0,000042
aspx.page	Begin PreRender	0,000376	0,000037
aspx.page	End PreRender	0,000466	0,000052
aspx.page	Begin SaveViewState	0,001253	0,000787
aspx.page	End SaveViewState	0,001359	0,000107
aspx.page	Begin Render	0,001409	0,000050
aspx.page	End Render	0,002087	0,000678
Eventos de la página	Evento Load		
Eventos de la página	Evento PreRender	0,000414	0,000038
Eventos de los controles	Se ha pulsado un RadioButton	0,000222	0,000041

Figura 79

Trazas a nivel de aplicación

En este caso el mecanismo de trazas se va a aplicar a toda la aplicación ASP .NET, para ello debemos editar y modificar el fichero de configuración de la aplicación, es decir, el fichero Web.config.

Para configurar el mecanismo de trazas de una aplicación ASP .NET debemos añadir un elemento <trace> dentro de la sección <system.web> del fichero de configuración Web.config.

La etiqueta trace, que nos va a permitir habilitar u configurar el mecanismo de trazas a nivel de una aplicación ASP .NET, presenta los siguientes atributos:

- **enabled:** este atributo indica si el mecanismo de trazas se encuentra activado a o no, tiene por lo tanto la misma funcionalidad que el atributo de mismo nombre de la directiva `@Page`. Su valor por defecto es `False`.
- **requestLimit:** en este atributo indicaremos el número máximo de peticiones HTTP de las que se va a almacenar información de trazas, las trazas van a ser almacenadas en un registro de trazas mediante un mecanismo circular en las que permanecerán las últimas `n` peticiones. Por defecto el valor que presenta este atributo es 10.
- **pageOutput:** indica si la información de trazas se va a mostrar al final de cada página ASP .NET, tal como se hacía con las trazas a nivel de página. Aunque la información de las trazas no se muestre en la página, quedará constancia de las mismas en el registro de trazas. Su valor por defecto es `False`.
- **traceMode:** este atributo nos permite indicar el modo de ordenación de los mensajes de trazas en la sección Información de seguimiento, tiene por lo tanto la misma funcionalidad que el atributo de mismo nombre de la directiva `@Page`. Puede presentar los valores `SortByCategory` y `SortByTime`, su valor por defecto es `SortByTime`.
- **localOnly:** indica si la información de trazas se muestra únicamente a los clientes locales o por el contrario se muestra también a los clientes remotos. Su valor por defecto es `False`.

En el Código fuente 199 se muestra un fragmento del fichero `Web.config` de una aplicación ASP .NET, en este caso se está activando el mecanismo de trazas para la aplicación actual, pero la información de las trazas no se va a mostrar al final de cada página y únicamente se va a permitir acceder al registro de trazas de la aplicación a los clientes locales.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.web>
    <trace
      enabled="true"
      requestLimit="10"
      pageOutput="false"
      traceMode="SortByTime"
      localOnly="true"
    />
  </system.web>
</configuration>
```

Código fuente 199

Debido a que existe la posibilidad de no mostrar la información de las trazas al final de cada página, como ocurre en el caso del ejemplo, debemos tener una herramienta que nos permita consultar la información de trazas de las distintas páginas ASP .NET que forman parte de la aplicación Web actual, para ello el entorno de ASP .NET nos ofrece una URL especial que nos va a permitir consultar el registro de trazas de la aplicación ASP .NET actual.

Para hacer uso de esta herramienta debemos invocarla navegando al recurso `trace.axd` desde el directorio raíz de la aplicación ASP .NET en la que se ha activado el mecanismo de trazas. Este fichero no existe realmente en la aplicación, sino que se trata de una URL especial que ASP .NET va a interceptar para mostrarnos el registro de trazas de la aplicación.

La llamada a este recurso nos va a mostrar una página similar a la de la Figura 80, en la que aparecen una serie de peticiones realizadas a las páginas ASP .NET de la aplicación actual. El número de peticiones dependerá del valor de la propiedad requestLimit de la etiqueta trace dentro del fichero Web.config.

Al pulsar sobre el enlace Ver detalles de cada una de las peticiones de las páginas ASP .NET, veremos la información de trazas de la página correspondiente. También se nos ofrece la posibilidad de eliminar el registro de trazas actual.

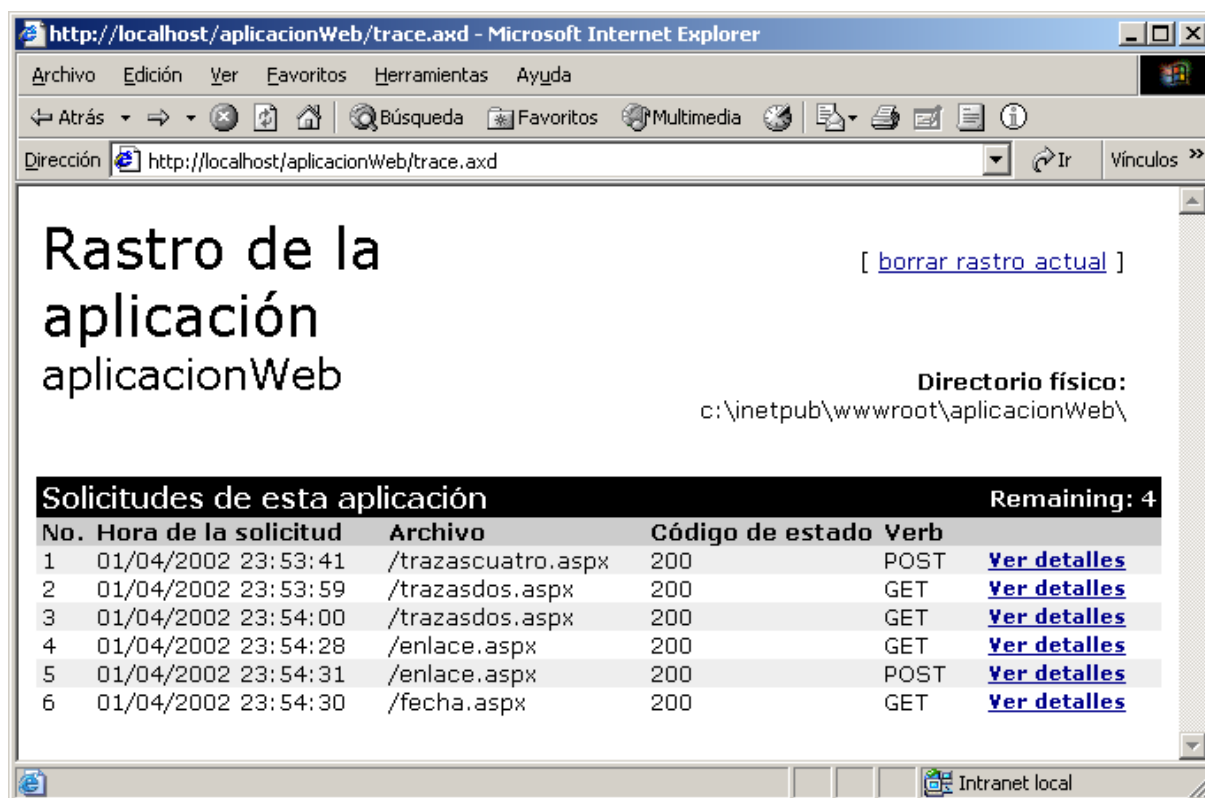


Figura 80

Mediante el mecanismo de trazas a nivel de aplicación podemos monitorizar toda la información de trazas de las páginas de la aplicación sin que los usuarios sean conscientes de ello.

Con este apartado finalizamos la parte del presente capítulo dedicado a la utilización de trazas dentro de ASP .NET, en lo que resta de capítulo comentaremos el mecanismo de depuración que ofrece ASP .NET.

Depuración en ASP .NET

Además de ofrecer un mecanismo de trazas, como novedad ASP .NET también ofrece herramientas para realizar la depuración de páginas. En versiones anteriores de ASP la mejor forma de depurar era, al igual que sucedía con las trazas, utilizar la instrucción Response.Write, aunque existía una herramienta para depurar llamada Script Debugger, conseguir depurar con la misma era bastante complejo y los resultados no eran los más satisfactorios.

Todo estos inconvenientes han terminado en .NET, ya que el CLR (Common Language Runtime), ofrece la depuración integrada para todo tipo de aplicaciones, incluidas las aplicaciones ASP .NET.

Dentro del entorno .NET Framework existen dos formas de depurar, la primera de ellas es utilizar la herramienta SDK debugger, esta herramienta se incluye en el SDK del .NET Framework, y la segunda posibilidad es utilizar el entorno de desarrollo Visual Studio .NET.

Depurando con el SDK Debugger

Esta herramienta, que nos va a permitir la depuración de páginas ASP .NET, se encuentra incluida como parte de la instalación del SDK del .NET Framework. El ejecutable de este depurador es el fichero DbgCLR.exe y lo podemos encontrar en el directorio C:\Archivos de programa\Microsoft.NET\FrameworkSDK\GuiDebug. Este depurador también es llamado Depurador Microsoft CLR.

Este depurador ofrece unas funcionalidades muy similares al depurador de Visual Studio .NET, exceptuando la depuración remota y la edición y continuación con la ejecución del código.

Como ya vimos en la primera parte de este capítulo, si queremos utilizar las trazas dentro de una página ASP .NET determinada debemos utilizar un atributo de la directiva @Page, para activar la depuración de una página debemos hacer algo similar. En este caso se debe utilizar el atributo Debug de la directiva @Page, este atributo admite los valores True/False, por defecto presenta el valor False, es decir, por defecto no se encuentra habilitada la depuración de páginas.

Al activar la depuración de una página ASP .NET estamos indicando al compilador que genere los símbolos de depuración necesarios para realizar el proceso de depuración cuando se estime necesario.

Si además de habilitar la depuración en las páginas ASP .NET, también deseamos depurar los componentes .NET utilizados en las mismas, debemos indicar a la hora de compilar los componentes que vamos a depurarlos. Para ello se utiliza el parámetro /debug con el compilador, si estamos utilizando el compilador del Microsoft .NET Framework SDK (Software Development Kit), es decir, la herramienta CSC. EN capítulos anteriores ya vimos otros parámetros que podíamos utilizar con el compilador de C#.

Si compilamos nuestros componentes desde Visual Studio .NET, no debemos hacer nada adicional, ya que por defecto el entorno de desarrollo compila los componentes con la opción de depuración habilitada.

A continuación vamos ver como se puede depurar una página ASP .NET y los componentes utilizados en la misma mediante el depurador ofrecido por el SDK del .NET Framework, para ello vamos a utilizar un ejemplo. Se trata de una página ASP .NET, cuyo código es el Código fuente 200, que hace uso de un sencillo componente .NET que muestra la fecha y hora actuales.

```
<%@ Page Debug="True"%>
<%@ Import Namespace="Componentes.Ejemplos" %>
<html>
<script language="C#" runat="server">
ComponenteSencillo componente=new ComponenteSencillo();
void Pulsado(Object origen, EventArgs argumentos){
    etiqueta.Text =componente.muestraHora();
}
</script>
<head><title>Depuración en ASP .NET</title></head>
<body>

<form id="formulario" method="post" runat="server">
<asp:button id="boton" text="Pulsar" onclick="Pulsado" runat="Server"/>
```

```
<asp:label id="etiqueta" runat="server"/>
</form>

</body>
</html>
```

Código fuente 200

Como se puede ver en el código anterior se ha habilitado la depuración en la página, el código de la página es muy sencillo, presenta un Web Form con una etiqueta y un botón, la pulsar el botón se instanciará un objeto de la clase Componentes.Ejemplos.ComponenteSencillo y se llamará al método muestraHora(). En el Código fuente 201 se puede ver el código de la clase del componente, se supone que este componente se ha compilado con el parámetro debug, para así permitir su depuración.

```
using System;

namespace Componentes.Ejemplos
{
    public class ComponenteSencillo
    {
        public ComponenteSencillo()
        {
        }

        public String muestraHora()
        {
            DateTime ahora=DateTime.Now;
            return ahora.ToString();
        }
    }
}
```

Código fuente 201

En la Figura 81 se puede ver la ejecución de esta página que hace uso del componente.

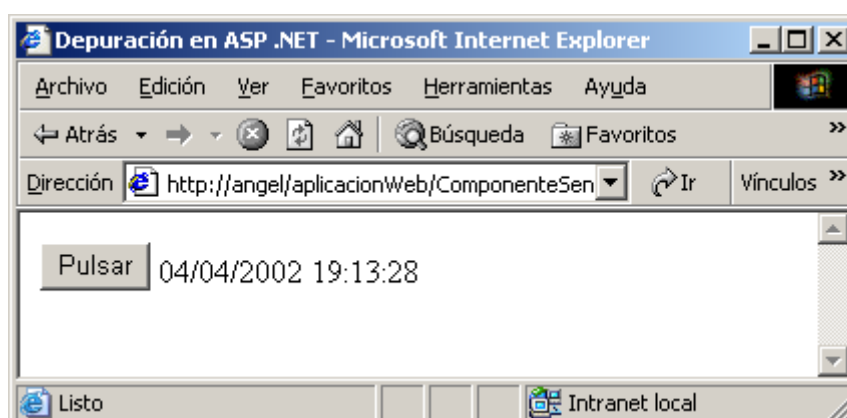


Figura 81

Se debe señalar que antes de utilizar el depurador es necesario ejecutar la página ASP .NET dentro del navegador correspondiente, de esta forma los símbolos de depuración para la página serán cargados para que los pueda utilizar el depurador en su proceso de depuración.

Ya tenemos preparada la página ASP .NET y el componente que deseamos depurar, el siguiente paso es ejecutar el depurador (fichero DbgCLR.exe), que se encuentra en la ruta C:\Archivos de programa\Microsoft.NET\FrameworkSDK\GuiDebug.

Una vez que se ha iniciado el depurador y con la página ASP .NET cargada en el navegador debemos acceder ala opción de menú Herramientas|Procesos de depuración. En ese momento aparecerá un diálogo similar al de la .En ese diálogo debemos marcar la casilla Mostrar procesos del sistema, de esta forma podremos seleccionar el proceso que se corresponde con la ejecución de la página ASP .NET que vamos a depurar.

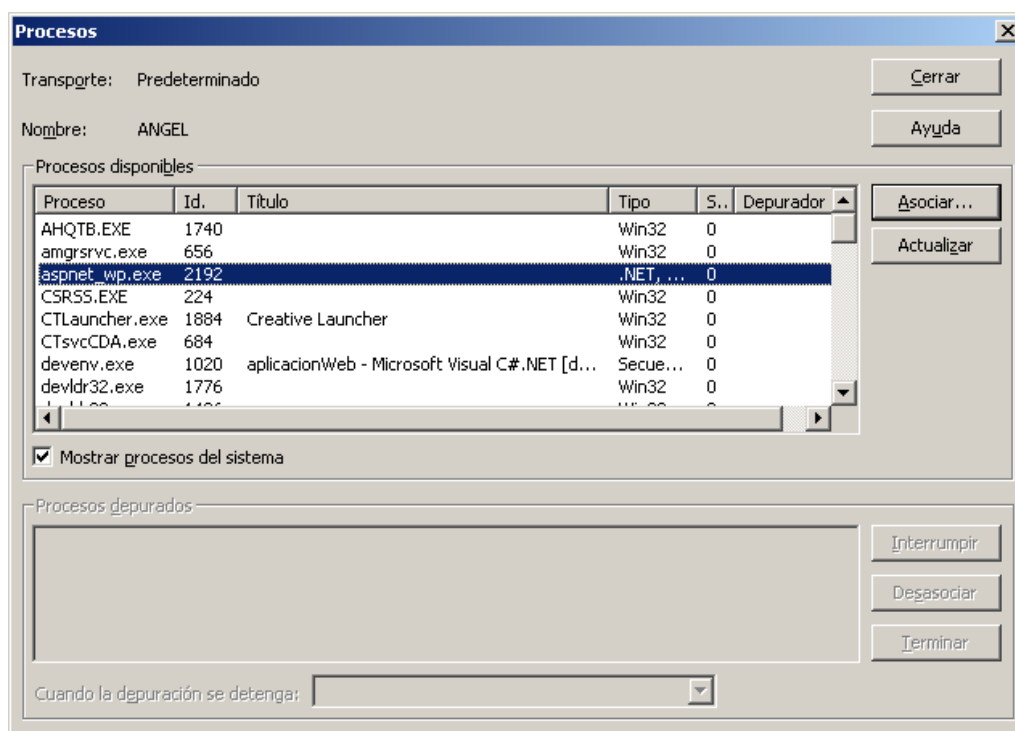


Figura 82

El proceso del sistema que se corresponde con la ejecución de las páginas ASP .NET es aspnet_wp.exe. Una vez seleccionado este proceso entre la lista de procesos del sistema, pulsamos el botón Asociar, de esta forma indicamos que el depurador va a depurar el proceso indicado, una vez hecho esto podemos cerrar el diálogo.

Una vez preparado el SDK debugger ya sólo tenemos que indicarle la página ASP .NET que deseamos depurar, para ello se utiliza la opción de menú Archivo|Abrir. En el diálogo que aparece indicaremos la página correspondiente.

Con la página ASP .NET cargada en el depurador podemos añadirle un punto de ruptura. Para añadir un punto de ruptura en el código fuente, únicamente debemos pulsar en el margen gris de la línea correspondiente. En este caso vamos a añadir el punto de ruptura en la línea en la que se llama al método muestraHora() del objeto componente de la clase ComponenteSencillo. Al ejecutarse esta línea de código de la página ASP .NET se detendrá al ejecución de la misma en el punto de ruptura, es

decir, el SDK debugger funciona de la misma forma que los depuradores convencionales de otros entornos. En la Figura 83 se puede ver el aspecto que tendría la página dentro del depurador CLR.

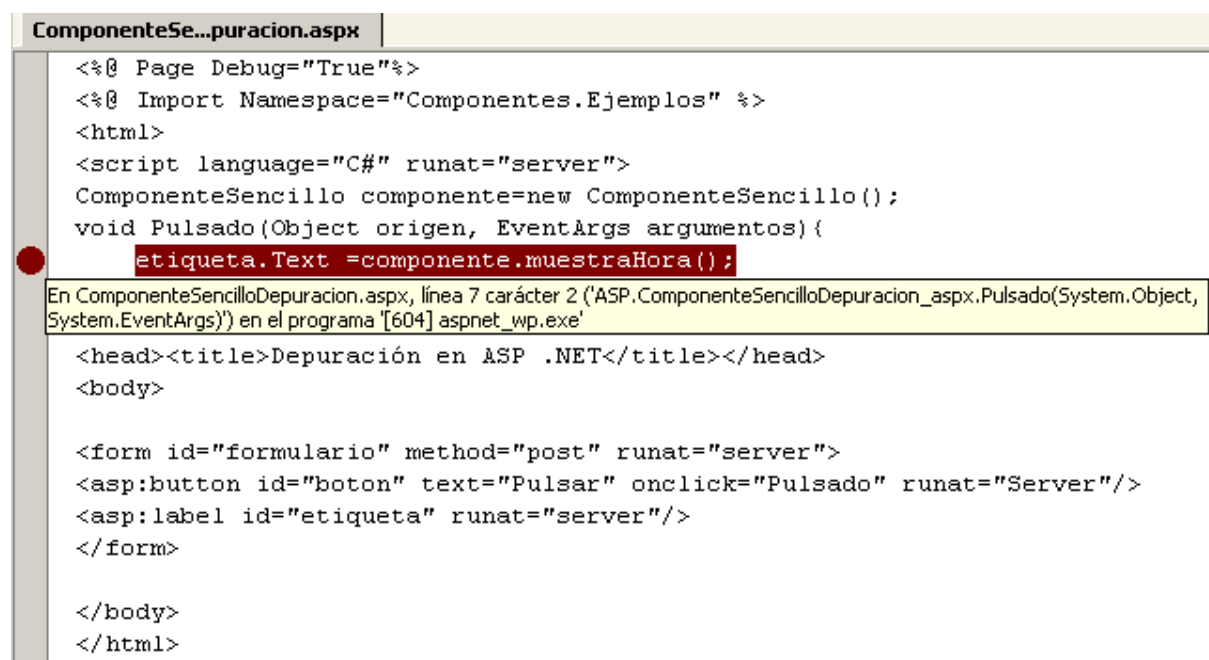


Figura 83

Una vez que se ha detenido la ejecución en el punto de interrupción, podemos controlar lo que deseamos hacer a continuación. Podemos acceder al código fuente del método dentro del componente si pulsamos la opción Ir a instrucciones, en este caso se abrirá una nueva ventana de código, similar a la de la Figura 84 con el código fuente del componente .NET, y se indicará en color amarillo en que línea nos encontramos.

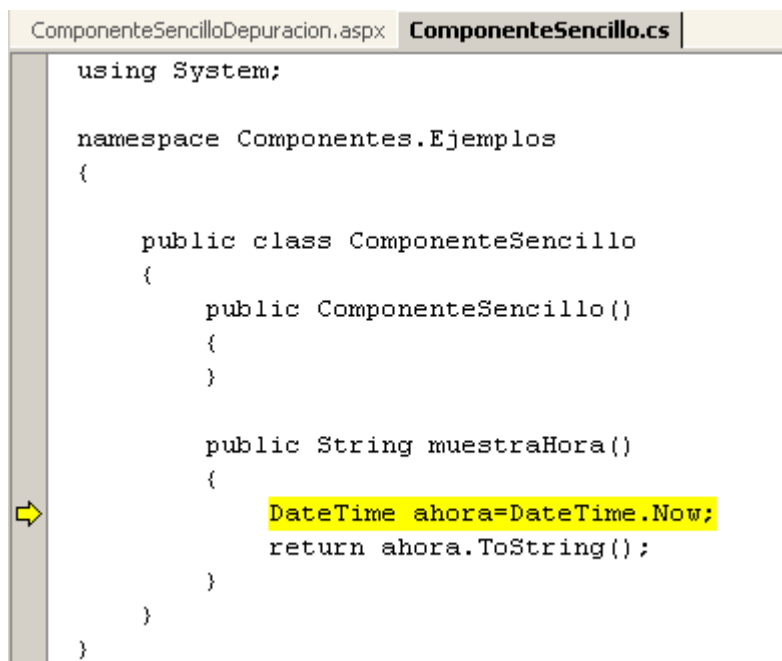


Figura 84

Una vez que se termina de ejecutar el código del componente .NET se vuelve a la página, es decir, tenemos un control completo sobre los pasos en la ejecución de la página, ya que podemos indicar también salir del componente mediante la opción Paso a paso para salir.

Para aquellos lectores que hayan utilizado versiones anteriores de ASP comprobarán las grandes facilidades que nos ofrece este depurador, la depuración de páginas ASP .NET supone todo un avance a la hora de localizar errores en nuestras aplicaciones.

El depurador CLR ofrece una serie de características muy similares a las de otros depuradores de otros entornos de desarrollo, por lo que no nos vamos a entretener mucho más comentando este depurador, aunque si merece la pena enumerar las posibilidades que nos ofrece:

- Control completo sobre la ejecución de las instrucciones del código fuente, permitiendo Ir a instrucciones, Paso a paso por procedimientos y Paso a paso para salir.
- Establecimiento de puntos de ruptura en líneas específicas.
- Establecimiento de puntos de ruptura en excepciones específicas.
- Establecimiento de puntos de ruptura en expresiones.
- Visualización del contenido de variables.
- Visualización de la pila de llamadas.

Pero hay dos características, que ya avanzábamos anteriormente, que no las soporta este depurador, se trata de la depuración de procesos remotos y la posibilidad de editar el código y continuar. Por esto último se dice que el depurador CLR es un depurador de sólo lectura, debido a esto no podemos editar el código y continuar con el proceso de depuración. Si deseamos estas características adicionales debemos acudir al depurador de Visual Studio .NET.

Conclusión

Como vemos, todo un nuevo entramado de construcción de aplicaciones para Internet con un sinfín de posibilidades, y notables mejoras. El próximo capítulo lo dedicaremos a una breve introducción a los Servicios Web XML.

Bibliografía y recursos

Bibliografía

"A programmers introduction to C#", Eric Gunnerson, Ed. Apress, 2000

"Inside C#", Tom Archer, Ed. Microsoft Press, 2001

"Professional C#", Simon Robinson et. al. Wrox Press, 2001

"C# and the .NET Platform", Andrew Troelsen. Ed. Apress, 2001

"El lenguaje de programación C#". Jose Antonio González Seco. Ed. Digital.

"Programación para Internet con ASP.NET", Angel Esteban, Ed. Grupo EIDOS, 2002

"Introducing .NET Framework", James Connard et. al. Ed. Wrox Press, 2001

"Moving to VB.NET: Strategies, Concepts and Code", Daniel Appleman, Ed. Apress, 2001

"XML IE5: programmers reference", Alex Homer, Ed. Wrox. 1999

Recursos en Internet

Genéricos sobre .NET

<http://www.mindcracker.com/csharp/>
<http://www.devx.com/dotnet/resources/>
<http://www.fawcette.com/archives/magazines/dotnetmag/>
<http://www.passport.com/sdkdocuments/sdk21/default.htm>
<http://advisor.com/>
<http://www.dotnet101.com/>
<http://www.fcharte.com/> (en Español)
<http://www.sjoerd.org/projects/enhance/>
<http://www.codeproject.com/winhelp/msdnintegrator.asp>
<http://www.xpdotnet.com/>
<http://uddi.microsoft.com/developer/default.asp>
<http://www.dnjonline.com/webwatch/index.html>

Sobre el lenguaje C#

<http://csharpindex.com/>
<http://manowar.lsi.us.es/~csharp/>
<http://www.develop.com/dm/default.asp>
<http://www.learnsharp.cjb.net/>
<http://www.c-sharpcorner.com/Code.asp>

Revistas gratuitas

<http://www.algoritmodigital.com> (en español)
<http://msdn.microsoft.com/msdnmag/>
<http://www.portalvb.com> (en Español)
<http://www.devx.com/dotnet/default.asp>

<http://www.vb2themax.com/>

<http://guille.costasol.net/> (en Español)

Componentes en .NET

<http://www.vbpj.com/upload/free/features/vbpj/2001/06jun01/ce0106/ce0601-1.asp>

<http://www.componentsource.com/BuildComponents/Resources/TechnicalArticles.asp>

<http://www.developerdex.com/csharp/default.asp>

Si quiere ver más textos en este formato, visítenos en: <http://www.lalibreriadigital.com>.

Este libro tiene soporte de formación virtual a través de Internet, con un profesor a su disposición, tutorías, exámenes y un completo plan formativo con otros textos. Si desea inscribirse en alguno de nuestros cursos o más información visite nuestro campus virtual en: <http://www.almagesto.com>.

Si quiere información más precisa de las nuevas técnicas de programación puede suscribirse gratuitamente a nuestra revista **Algoritmo** en: <http://www.algoritmodigital.com>. No deje de visitar nuestra revista **Alquimia** en <http://www.eidos.es/alquimia> donde podrá encontrar artículos sobre tecnologías de la sociedad del conocimiento.

Si quiere hacer algún comentario, sugerencia, o tiene cualquier tipo de problema, envíelo a la dirección de correo electrónico lalibreriadigital@eidos.es.