Desarrollo de software

Texto diseñado para enseñar en profundidad a desarrollar aplicaciones basadas en la plataforma .NET Framework, utilizando Visual Basic .NET como lenguaje.

El texto cubre tanto aspectos básicos como avanzados, abordando el lenguaje, programación orientada a objetos (OOP), el entorno de desarrollo de aplicaciones (IDE) de Visual Studio .NET, etc.

Se trata de un manual de muy amplio contenido en documentación, además de numerosos ejemplos listos para ser utilizados desde Visual Basic .NET.

Entre los requisitos previos, basta con conocimientos de fundamentos de programación, conocer al menos un lenguaje, sea este el que sea y conocer el S.O. Windows a nivel de usuario.

PROGRAMACIÓN EN VISUAL BASIC .NET

LUIS MIGUEL BLANCO

GRUPO Fidos

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)-91-5012824). 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.

© Luis Miguel Blanco, 2002
© Grupo EIDOS Consultaría y Documentación Informática, S.L., 2002

ISBN 84-88457-53-7

Programación en Visual Basic .NET

Luis Miguel Blanco

Responsable de la edición
Paco Marín (fmarin@eidos.es)

Coordinación de la edición
Antonio Quirós (aquiros@eidos.es)

Autoedición
Magdalena Marín (mmarin@eidos.es)
Luis Miguel Blanco (lmblanco@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

A Olga y David,

por las horas que les he robado

para escribir este texto que ahora

tienes en tus manos

A mis padres

A Roberto y Carlos,

mis hermanos pequeños, en edad, pero no en altura

Índice

ÍNDICE **7**

INTRODUCCIÓN **21**

UNA VERSIÓN LARGO TIEMPO ESPERADA 21
AQUELLOS DIFÍCILES TIEMPOS DE LA PROGRAMACIÓN EN WINDOWS 22
SÓLO PARA LOS ELEGIDOS 22
EL PROYECTO THUNDER 23
UN PRODUCTO REVOLUCIONARIO 23
EL PASO A OLE Y ODBC 23
PASO A 32 BITS Y ORIENTACIÓN A OBJETO 23
ACTIVEX Y ADO 24
PROGRAMACIÓN EN INTERNET 24
LOS PROBLEMAS PARA EL DESARROLLO EN LA RED 25
EL PANORAMA ACTUAL 25
LAS SOLUCIONES APORTADAS POR VB.NET 25
EL PRECIO DE LA RENOVACIÓN 25
COMENZAMOS 26

LA EVOLUCIÓN HACIA .NET **27**

LAS RAZONES DEL CAMBIO 27
LA DIFÍCIL DECISIÓN DE ABANDONAR ANTERIORES TECNOLOGÍAS 28
LA PROBLEMÁTICA DE WINDOWS DNA 28
ASP 28
ADO 28

VISUAL BASIC 29
CONFLICTOS CON DLL'S 29
TRAS LOS PASOS DE COM 29

.NET FRAMEWORK, NUEVOS CIMIENTOS PARA LA NUEVA GENERACIÓN DE APLICACIONES 31

ALGO ESTÁ CAMBIANDO 31
¿QUÉ ES .NET? 32
.NET FRAMEWORK 34
EL CLR, COMMON LANGUAGE RUNTIME 35
EL CTS, COMMON TYPE SYSTEM 36
¿Qué es un tipo dentro de .NET Framework? *36*
Los tipos de datos son objetos *37*
Categorías de tipos *39*
La disposición de los datos en la memoria *39*
Embalaje y desembalaje de tipos por valor *42*
METADATA (METADATOS) 44
SOPORTE MULTI-LENGUAJE 44
EL CLS (COMMON LANGUAGE SPECIFICATION) 45
EJECUCIÓN ADMINISTRADA 45
Código administrado *46*
Datos administrados *46*
Recolección de memoria no utilizada *46*
Recolección de memoria en VB6 y versiones anteriores *47*
Recolección de memoria en .NET Framework *47*
LA EJECUCIÓN DE CÓDIGO DENTRO DEL CLR 47
El IL, Intermediate Language *47*
Compilación instantánea del IL y ejecución *48*
Compilación bajo demanda *49*
Independencia de plataforma *50*
DOMINIOS DE APLICACIÓN 50
SERVIDORES DE ENTORNO 51
NAMESPACES 52
LA JERARQUÍA DE CLASES DE .NET FRAMEWORK 55
ENSAMBLADOS 56
La problemática tradicional de los componentes *57*
Ensamblados, una respuesta a los actuales conflictos *57*
Tipos de ensamblado según modo de creación *57*
El contenido de un ensamblado *58*
El manifiesto del ensamblado *58*
Tipos de ensamblado según contenido *59*
Tipos de ensamblado según ámbito *61*
Ubicación de ensamblados compartidos *61*
Identificación mediante claves integradas en el propio ensamblado *62*
Versiones de ensamblados *62*
Compatibilidad a nivel de versión *63*
Ejecución conjunta de ensamblados *63*
Ficheros de configuración *64*
Localización de ensamblados por parte del CLR *64*
Optimización de la carga de ensamblados *65*

INSTALACIÓN DE VISUAL STUDIO .NET 67

PREPARACIÓN DEL ENTORNO DE TRABAJO 67
.NET Framework SDK *67*
Visual Studio .NET *67*

REQUISITOS HARDWARE 68
SISTEMA OPERATIVO 68
RECOMENDACIONES PREVIAS 68
INSTALACIÓN DE VISUAL STUDIO .NET 69
BASES DE DATOS DE EJEMPLO 75

LA PRIMERA APLICACIÓN 79

UN HOLA MUNDO DESDE VB.NET 79
INICIAR EL IDE DE VS.NET 79
CREAR UN NUEVO PROYECTO 81
OBJETOS, PROPIEDADES Y MÉTODOS 82
FORMULARIOS 82
EL FORMULARIO COMO UN OBJETO 83
ACCESO A LAS PROPIEDADES DE UN FORMULARIO 83
CONTROLES 85
LABEL 86
EJECUTANDO LA APLICACIÓN 87
EL CÓDIGO DE LA APLICACIÓN 88
DISECCIONANDO EL CÓDIGO FUENTE DEL FORMULARIO 90
La clase del formulario 91
El método constructor New( ) 91
Configuración del formulario y creación de controles 92
Liberación de recursos del formulario 93
ESTRUCTURA Y GRABACIÓN DEL PROYECTO 93

ESCRITURA DE CÓDIGO 95

ESCRIBIR CÓDIGO, EL PAPEL CLÁSICO DEL PROGRAMADOR 95
UN PROGRAMA ESCRIBIENDO SU CÓDIGO 95
CREAR EL PROYECTO 96
UN NUEVO MÓDULO DE CÓDIGO 96
LA UBICACIÓN FÍSICA DEL CÓDIGO 97
COMENTARIOS DE CÓDIGO 98
PROCEDIMIENTOS 98
EL PUNTO DE ENTRADA AL PROGRAMA 99
LA CLASE MESSAGEBOX 99
CONFIGURAR EL PUNTO DE ENTRADA DEL PROYECTO 100
VARIABLES 102
INPUTBOX( ) 102
COMPLETANDO EL PROCEDIMIENTO 103
FINALIZAMOS EL PROGRAMA 104

UNA APLICACIÓN CON FUNCIONALIDAD BÁSICA 105

INTEGRANDO LO VISTO HASTA EL MOMENTO 105
UN PROGRAMA MÁS OPERATIVO 105
DISEÑO DEL FORMULARIO 105
CONTROLES Y EVENTOS 108
OTRO MODO DE ESCRIBIR EL CÓDIGO DE UN EVENTO 109
GRABANDO TEXTO EN UN FICHERO 110
UNA PUNTUALIZACIÓN SOBRE LOS EVENTOS 112

EL ENTORNO DE DESARROLLO INTEGRADO (IDE), DE VISUAL STUDIO .NET 113

EL IDE, UN ELEMENTO A VECES MENOSPRECIADO 113
EL LARGO CAMINO HACIA LA CONVERGENCIA 114
VISUAL STUDIO .NET, EL PRIMER PASO DE LA TOTAL INTEGRACIÓN 114
LA PÁGINA DE INICIO 114

PRINCIPALES ELEMENTOS EN NUESTRO ENTORNO DE TRABAJO 116
VENTANA PRINCIPAL DE TRABAJO 117
MANEJO DE VENTANAS ADICIONALES DEL IDE 119
EL EXPLORADOR DE SOLUCIONES 124
Agregar nuevos elementos a un proyecto *126*
Propiedades del proyecto *127*
Propiedades de la solución *128*
Agregar proyectos a una solución *128*
EL MENÚ CONTEXTUAL 130
EL DISEÑADOR DEL FORMULARIO 130
LA VENTANA DE PROPIEDADES 131

EL IDE DE VISUAL STUDIO .NET. HERRAMIENTAS Y EDITORES 135

EL CUADRO DE HERRAMIENTAS 135
Organización en fichas *136*
Manipulación de fichas *137*
Organización de controles *137*
Manipulación de controles *139*
Agregar controles *140*
El cuadro de herramientas como contenedor de código fuente *141*
LAS BARRAS DE HERRAMIENTAS 142
Barras de herramientas personalizadas *143*
Acople de barras de herramientas *145*
OPCIONES ADICIONALES DE PERSONALIZACIÓN 145
VENTANA DE RESULTADOS 146
EL EDITOR DE CÓDIGO FUENTE 147
Ajuste de fuente y color *148*
Números de línea *149*
Búsqueda y sustitución de código *149*
Ajuste de línea *151*
Dividir el editor de código *152*
Marcadores *153*
Mostrar espacios en blanco *154*
Esquematización *154*
Regiones *155*
Comentarios de código en bloque *156*
Ir a la definición de un procedimiento *156*
IntelliSense *156*
Cambiar a mayúsculas y minúsculas *158*

EL IDE DE VISUAL STUDIO .NET. ELEMENTOS COMPLEMENTARIOS Y AYUDA 159

EDITORES DE IMÁGENES 159
LISTA DE TAREAS 160
Definición de símbolos para tareas *161*
Creación de tareas *161*
Ventana Lista de tareas *162*
Eliminación de tareas *163*
MOSTRAR LA PANTALLA COMPLETA 163
LA VISTA DE CLASES 163
EL EXPLORADOR DE OBJETOS 164
MACROS 165
El Explorador de macros *166*
Ejecución de macros *167*
Grabación de macros *167*
Manipulación de proyectos de macros *168*

El IDE de macros ... *169*
Escritura de macros ... *170*
Macro para comentar líneas de código determinadas ... *172*
EL SISTEMA DE AYUDA ... 173
Ayuda dinámica ... *174*
Contenido ... *175*
Índice ... *176*
Buscar ... *177*
Ayuda externa ... *178*
Mantener temas de ayuda disponibles ... *179*
Otros modos de acceso a la ayuda ... *180*

APLICACIONES DE CONSOLA ... 181

CREACIÓN DE UN PROYECTO DE TIPO APLICACIÓN DE CONSOLA ... 181
LA CLASE CONSOLE ... 182
ESCRITURA DE INFORMACIÓN ... 183
ESCRITURA DE MÚLTIPLES VALORES EN LA MISMA LÍNEA ... 185
LECTURA DE INFORMACIÓN ... 187

EL LENGUAJE ... 189

EL LENGUAJE, PRINCIPIO DEL DESARROLLO ... 189
ESTRUCTURA DE UN PROGRAMA VB.NET ... 190
MAIN( ) COMO PROCEDIMIENTO DE ENTRADA AL PROGRAMA ... 191
VARIABLES ... 191
Declaración ... *191*
Denominación ... *192*
Avisos del IDE sobre errores en el código ... *192*
Lugar de la declaración ... *192*
Tipificación ... *193*
Declaración múltiple en línea ... *195*
Asignación de valor ... *195*
Valor inicial ... *197*
Declaración obligatoria ... *198*
Tipificación obligatoria ... *201*
ARRAYS, CONCEPTOS BÁSICOS ... 203
Declaración ... *203*
Asignación y obtención de valores ... *204*
Modificación de tamaño ... *205*
Recorrer un array ... *206*
CONSTANTES ... 206
CONCEPTOS MÍNIMOS SOBRE DEPURACIÓN ... 208

OPERADORES DEL LENGUAJE ... 211

ARITMÉTICOS ... 211
Potenciación: ^ ... *211*
*Multiplicación: ** ... *212*
División real: / ... *212*
*División entera: * ... *213*
Resto: Mod ... *213*
Suma: + ... *213*
Resta: - ... *214*
CONCATENACIÓN: &, + ... 215
OPERADORES ABREVIADOS DE ASIGNACIÓN ... 215
Potencia: ^= ... *215*
*Multiplicación: *=* ... *216*

División real: /= ... *216*
División entera: \= ... *216*
Suma: += ... *217*
Resta: -= ... *217*
Concatenación: &= ... *218*
COMPARACIÓN ... 218
Comparación de cadenas ... *219*
La función Asc( ) ... *220*
La función Chr( ) ... *221*
Comparación de cadenas en base a un patrón. El operador Like ... *221*
Comparación de objetos. El operador Is ... *224*
LÓGICOS Y A NIVEL DE BIT ... 225
And ... *225*
Uso de paréntesis para mejorar la legibilidad de expresiones ... *226*
Not ... *227*
Or ... *228*
Xor ... *229*
AndAlso ... *230*
OrElse ... *231*
PRIORIDAD DE OPERADORES ... 231
USO DE PARÉNTESIS PARA ALTERAR LA PRIORIDAD DE OPERADORES ... 233

RUTINAS DE CÓDIGO ... 235

DIVISIÓN DE UNA LÍNEA DE CÓDIGO ... 235
ESCRITURA DE VARIAS SENTENCIAS EN LA MISMA LÍNEA ... 236
PROCEDIMIENTOS ... 236
Sintaxis de un procedimiento Sub ... *237*
Llamada a un procedimiento Sub ... *238*
Sintaxis de un procedimiento Function ... *238*
Llamada a un procedimiento Function ... *240*
Paso de parámetros a procedimientos ... *241*
Protocolo de llamada o firma de un procedimiento ... *241*
Tipo de dato de un parámetro ... *241*
Paso de parámetros por valor y por referencia ... *242*
Paso por valor (ByVal) ... 242
Paso por referencia (ByRef) ... 243
Paso de parámetros por posición y por nombre ... *244*
Parámetros opcionales ... *245*
Array de parámetros ... *246*
Sobrecarga de procedimientos ... *247*
Lista desplegable "Nombre de método", en el editor de código ... *251*

BIFURCACIÓN Y ÁMBITO DEL CÓDIGO ... 253

ESTRUCTURAS DE CONTROL ... 253
Selección ... *253*
If...End If ... 253
Select Case...End Select ... 257
Repetición ... *259*
While...End While ... 259
Do...Loop ... 260
For...Next ... 262
For Each...Next ... 264
ORGANIZACIÓN DEL PROYECTO EN FICHEROS Y MÓDULOS DE CÓDIGO ... 265
Agregar un nuevo módulo (y fichero) de código ... *266*
Crear un nuevo módulo dentro de un fichero existente ... *267*

Cambiar el nombre de un fichero de código ... *268*
Añadir al proyecto un fichero de código existente ... *269*
Lista desplegable "Nombre de clase", en el editor de código ... *269*
Excluir y eliminar ficheros de código del proyecto ... *270*
REGLAS DE ÁMBITO ... 271
Ámbito de procedimientos ... *271*
Público ... 271
Privado ... 273
Ámbito de variables ... *274*
Ámbito a nivel de procedimiento ... 274
Ámbito a nivel de bloque ... 275
Ámbito a nivel de módulo ... 276
Ámbito a nivel de proyecto ... 277
PERIODO DE VIDA O DURACIÓN DE LAS VARIABLES ... 278
VARIABLES STATIC ... 278

FUNCIONES COMPLEMENTARIAS DEL LENGUAJE ... 281

CONVENCIONES DE NOTACIÓN ... 281
FUNCIONES DE COMPROBACIÓN DE TIPOS DE DATOS ... 283
FUNCIONES DEL LENGUAJE ... 285
Numéricas ... *285*
Cadena de caracteres ... *286*
Fecha y hora ... *292*
CREAR MÚLTIPLES ENTRADAS AL PROGRAMA MEDIANTE DISTINTOS MAIN( ) ... 292

PROGRAMACIÓN ORIENTADA A OBJETO (OOP) ... 295

LAS VENTAJAS DE LA PROGRAMACIÓN ORIENTADA A OBJETO ... 295
DEL ENFOQUE PROCEDURAL AL ENFOQUE ORIENTADO A OBJETO ... 295
ABORDANDO UN PROBLEMA MEDIANTE PROGRAMACIÓN PROCEDURAL ... 295
LOS FUNDAMENTOS DE LA PROGRAMACIÓN ORIENTADA A OBJETO ... 297
OBJETOS ... 298
CLASES ... 298
INSTANCIAS DE UNA CLASE ... 299
CARACTERÍSTICAS BÁSICAS DE UN SISTEMA ORIENTADO A OBJETO ... 300
Abstracción ... *300*
Encapsulación ... *300*
Polimorfismo ... *301*
Herencia ... *301*
JERARQUÍAS DE CLASES ... 302
RELACIONES ENTRE OBJETOS ... 302
Herencia ... *302*
Pertenencia ... *303*
Utilización ... *303*
REUTILIZACIÓN ... 303
ANÁLISIS Y DISEÑO ORIENTADO A OBJETOS ... 303
CREACIÓN DE CLASES ... 304
ORGANIZACIÓN DE CLASES EN UNO O VARIOS FICHEROS DE CÓDIGO ... 305
CÓDIGO DE CLASE Y CÓDIGO CLIENTE ... 306
REGLAS DE ÁMBITO GENERALES PARA CLASES ... 306
INSTANCIACIÓN DE OBJETOS ... 306
MIEMBROS DE LA CLASE ... 307
DEFINIR LA INFORMACIÓN DE LA CLASE ... 307
CREACIÓN DE CAMPOS PARA LA CLASE ... 308
CREACIÓN DE PROPIEDADES PARA LA CLASE ... 309
VENTAJAS EN EL USO DE PROPIEDADES ... 310

ENCAPSULACIÓN A TRAVÉS DE PROPIEDADES 310
PROPIEDADES DE SÓLO LECTURA O SÓLO ESCRITURA 312
PROPIEDADES VIRTUALES 314
NOMBRES DE PROPIEDAD MÁS NATURALES 315
PROPIEDADES PREDETERMINADAS 316
ELIMINACIÓN DE LA PALABRA CLAVE SET PARA ASIGNAR OBJETOS 317

MÉTODOS Y ESPACIOS DE NOMBRE 319

CREACIÓN DE MÉTODOS PARA LA CLASE 319
¿CUÁNDO CREAR UNA PROPIEDAD Y CUÁNDO UN MÉTODO? 323
LA ESTRUCTURA WITH...END WITH 325
RESULTADOS DISTINTOS EN OBJETOS DE LA MISMA CLASE 325
USO DE ME Y MYCLASS PARA LLAMAR A LOS MIEMBROS DE LA PROPIA CLASE 326
SOBRECARGA DE MÉTODOS O POLIMORFISMO, EN UNA MISMA CLASE 327
ENLACE (BINDING) DE VARIABLES A REFERENCIAS DE OBJETOS 329
Enlace temprano 329
Enlace tardío 330
ESPACIOS DE NOMBRES (NAMESPACES) 333
ACCESO A ESPACIOS DE NOMBRE DE OTROS ENSAMBLADOS 337

CONSTRUCTORES Y HERENCIA 341

MÉTODOS CONSTRUCTORES 341
HERENCIA 343
TODAS LAS CLASES NECESITAN UNA CLASE BASE 344
REGLAS DE ÁMBITO ESPECÍFICAS PARA CLASES 345
Protected 346
Friend 347
Protected Friend 348
HERENCIA Y SOBRECARGA DE MÉTODOS 348
MYBASE, ACCESO A LOS MÉTODOS DE LA CLASE BASE 350
HERENCIA Y SOBRE-ESCRITURA DE MÉTODOS 350
DIFERENCIAS ENTRE SOBRECARGA Y SOBRE-ESCRITURA EN BASE AL TIPO DE ENLACE 353
OCULTAMIENTO DE MIEMBROS DE UNA CLASE 355
COMPORTAMIENTO DE LAS PALABRAS CLAVE ME, MYCLASS Y MYBASE ANTE LA SOBRE-ESCRITURA DE MÉTODOS 359
HERENCIA Y MÉTODOS CONSTRUCTORES 360
CLASES SELLADAS O NO HEREDABLES 362
CLASES ABSTRACTAS O NO INSTANCIABLES 362

ELEMENTOS COMPARTIDOS E INTERFACES 365

COMPROBACIÓN DEL TIPO DE UN OBJETO Y MOLDEADO (CASTING) 365
MIEMBROS COMPARTIDOS (SHARED) DE UNA CLASE 368
DEFINIR UNA CLASE COMO PUNTO DE ENTRADA DE LA APLICACIÓN 370
DESTRUCCIÓN DE OBJETOS Y RECOLECCIÓN DE BASURA 371
INTERFACES 373
ESTRUCTURAS 378
Creación y manipulación de estructuras 378
Estructuras o clases, ¿cuál debemos utilizar? 380
La estructura del sistema DateTime 382
ENUMERACIONES 383

APLICANDO UN ENFOQUE ENTERAMENTE OOP EN EL CÓDIGO 387

LOS TIPOS DE DATOS TAMBIÉN SON OBJETOS 387
MANIPULACIÓN DE CADENAS CON LA CLASE STRING 388
OPTIMIZANDO LA MANIPULACIÓN DE CADENAS CON LA CLASE STRINGBUILDER 393

CONVERSIÓN DE TIPOS CON LA CLASE CONVERT 394
LA ESTRUCTURA CHAR 395
EL TIPO DATE (FECHA) 396
OPERACIONES ARITMÉTICAS, LA CLASE MATH 396
FORMATEO DE VALORES 397
Fechas *398*
Modificando el formato estándar para las fechas *400*
Números *402*
Formateando directamente en la consola *403*
Usando la clase String para formatear *403*
Usando una clase para crear formatos personalizados *403*

DELEGACIÓN DE CÓDIGO Y EVENTOS 407

DELEGADOS (DELEGATES) 407
DECLARACIÓN DE DELEGADOS 407
CREACIÓN DE DELEGADOS 408
EXTENDER LAS FUNCIONALIDADES DE UNA CLASE A TRAVÉS DE DELEGADOS 411
EVENTOS. ¿QUÉ ES UN EVENTO? 414
EVENTOS EN .NET 414
PROGRAMACIÓN ESTRICTAMENTE PROCEDURAL 414
UN ESCENARIO DE TRABAJO SIN EVENTOS 414
PROGRAMACIÓN BASADA EN EVENTOS 415
ESQUEMA BÁSICO DE UN SISTEMA ORIENTADO A EVENTOS 415
EL EMISOR DE EVENTOS 416
EL RECEPTOR DE EVENTOS 417
CONEXIÓN DE UN EMISOR DE EVENTOS CON UN MANIPULADOR DE EVENTOS 417
ENLACE ESTÁTICO DE EVENTOS 418
ENLACE DINÁMICO DE EVENTOS 420
UN EVENTO ES UN DELEGADO 421
LA CLASE EVENTARGS, O CÓMO OBTENER INFORMACIÓN DEL OBJETO EMISOR DEL EVENTO 422

ARRAYS 427

ASPECTOS BÁSICOS 427
LA CLASE ARRAY 428
ADECUACIÓN DE LOS ARRAYS EN VB CON LOS ARRAYS DE LA PLATAFORMA .NET 428
El primer índice de un array debe ser siempre cero *429*
No es posible crear arrays con rangos de índices *429*
Todos los arrays son dinámicos *429*
DECLARACIÓN 430
ASIGNACIÓN Y OBTENCIÓN DE VALORES 431
RECORRER EL CONTENIDO 431
MODIFICACIÓN DE TAMAÑO 433
USO DEL MÉTODO CREATEINSTANCE( ) PARA ESTABLECER EL NÚMERO DE ELEMENTOS EN UN ARRAY 434
PASO DE ARRAYS COMO PARÁMETROS, Y DEVOLUCIÓN DESDE FUNCIONES 435
CLONACIÓN 435
COPIA 436
INICIALIZACIÓN DE VALORES 437
ORDENACIÓN 439
BÚSQUEDA 439
ARRAYS MULTIDIMENSIONALES 440

COLECCIONES 443

COLECCIONES, LA ESPECIALIZACIÓN DE LOS ARRAYS 443
EL ESPACIO DE NOMBRES SYSTEM.COLLECTIONS 444

LA CLAVE SE HALLA EN LOS INTERFACES 444
LA CLASE ARRAYLIST 445
Instanciación de objetos ArrayList *445*
Agregar valores a un ArrayList *445*
Recorrer y obtener valores de un ArrayList *446*
Capacidad y valores en una colección ArrayList *447*
Obtención de subarrays a partir de un objeto ArrayList *449*
Búsquedas en colecciones ArrayList *451*
Borrado de elementos en una colección ArrayList *451*
Ordenar elementos en un objeto ArrayList *453*
LA CLASE HASHTABLE 453
Manejo básico de colecciones Hashtable *453*
Operaciones varias con colecciones Hashtable *455*
Traspaso de elementos desde una colección Hashtable a un array básico *457*
LA CLASE SORTEDLIST 458
LA CLASE QUEUE 459
Manipulación de valores en una colección Queue *459*
LA CLASE STACK 462

COLECCIONES PERSONALIZADAS 465

CUANDO EL TIPO DE ARRAY QUE NECESITAMOS NO EXISTE 465
UTILIZANDO LA HERENCIA PARA CREAR UNA NUEVA COLECCIÓN 465
IMPLEMENTANDO UN INTERFAZ PARA CREAR UNA NUEVA COLECCIÓN 467

MANIPULACIÓN DE ERRORES 473

ERRORES, ESE MAL COMÚN 473
Errores de escritura *473*
Errores de ejecución *474*
Errores lógicos *474*
ERRORES Y EXCEPCIONES 474
MANIPULADORES DE EXCEPCIONES 475
TIPOS DE TRATAMIENTO DE ERROR EN VB.NET 475
MANIPULACIÓN ESTRUCTURADA DE ERRORES 475
La estructura Try...End Try *475*
La clase Exception *478*
Captura de excepciones de diferente tipo en el mismo controlador de errores *479*
Establecer una condición para un manipulador de excepciones *480*
La influencia del orden de los manipuladores de excepciones *482*
Forzar la salida de un controlador de errores mediante Exit Try *483*
Creación de excepciones personalizadas *484*
MANIPULACIÓN NO ESTRUCTURADA DE ERRORES 486
El objeto Err *486*
On Error *486*
On Error Goto Etiqueta *486*
On Error Resume Next *487*
Creación de errores con el objeto Err *488*
On Error Goto 0 ..... *488*

OPERACIONES DE ENTRADA Y SALIDA (I/O). GESTIÓN DEL SISTEMA DE ARCHIVOS 491

LA REMODELACIÓN DEL VIEJO ESQUEMA DE ENTRADA Y SALIDA 491
SYSTEM.IO, EL PUNTO DE PARTIDA 491
OBJETOS STREAM 492
LAS CLASES TEXTREADER Y TEXTWRITER 492
LA CLASE STREAMWRITER 492

LA CLASE STREAMREADER 494
LAS CLASES STRINGWRITER Y STRINGREADER 496
LA CLASE STREAM (FLUJO DE DATOS) 496
LA CLASE FILESTREAM 496
MANEJO DE DATOS BINARIOS 498
MANIPULACIÓN DE ARCHIVOS MEDIANTE FILE Y FILEINFO 498
MANIPULACIÓN DE ARCHIVOS MEDIANTE DIRECTORY Y DIRECTORYINFO 500
LA CLASE PATH 502
MONITORIZACIÓN DEL SISTEMA DE ARCHIVOS CON FILESYSTEMWATCHER 503
Ajuste preciso de filtros para el monitor de archivos 505
Establecer el procedimiento de evento con AddHandler 505
Consideraciones sobre la ruta de archivos 506
DETECCIÓN CON ESPERA, DE EVENTOS PRODUCIDOS SOBRE ARCHIVOS 507
MANIPULACIÓN DE ARCHIVOS MEDIANTE FUNCIONES ESPECÍFICAS DE VISUAL BASIC 507

FORMULARIOS WINDOWS 509

INTERFACES DE VENTANA. FORMULARIOS Y CONTROLES 509
SYSTEM.WINDOWS.FORMS 510
LA CLASE FORM 510
CREACIÓN DE UN FORMULARIO BÁSICO 510
EL CÓDIGO DEL FORMULARIO 512
CAMBIANDO EL NOMBRE DEL FORMULARIO 513
CREACIÓN DE FORMULARIOS DESDE CÓDIGO 514
INICIAR EL FORMULARIO DESDE MAIN( ) 515
TRABAJO CON CONTROLES 517
EL CUADRO DE HERRAMIENTAS 517
INSERTAR UN CONTROL EN EL FORMULARIO 518
AJUSTE DE LA CUADRÍCULA DE DISEÑO DEL FORMULARIO 519
ORGANIZACIÓN-FORMATO MÚLTIPLE DE CONTROLES 520
ANCLAJE DE CONTROLES 522
ACOPLE DE CONTROLES 523

CONTROLES WINDOWS 525

CONTROLES MÁS HABITUALES 525
BUTTON 526
CODIFICACIÓN DE LOS EVENTOS DE CONTROLES 527
CODIFICANDO OTROS EVENTOS DE UN CONTROL 528
ESCRITURA DEL MANIPULADOR DE EVENTO SIN USAR EL NOMBRE PROPORCIONADO POR EL EDITOR 530
RESPONDIENDO A LOS EVENTOS DE UN FORMULARIO 530
LABEL 531
FOCO DE ENTRADA 532
TEXTBOX 532
ORDEN DE TABULACIÓN DE CONTROLES 535
SELECCIÓN DE TEXTO EN UN TEXTBOX 535
CHECKBOX 538
RADIOBUTTON Y GROUPBOX 540
LISTBOX 542
COMBOBOX 547

CODIFICACIÓN AVANZADA DE CONTROLES Y HERENCIA VISUAL 549

COMPARTIENDO CÓDIGO ENTRE CONTROLES 549
CREACIÓN DE CONTROLES DESDE CÓDIGO 553
Código para el interfaz de usuario 553
Código para eventos del formulario, conectando con Handles 555

Código para eventos de controles, conectando con Handles ... *555*
Código para eventos de controles, conectando con AddHandler ... *556*
Código para eventos de controles, asociando y separando dinámicamente con AddHandler y RemoveHandler ... *557*
RECORRIENDO LOS CONTROLES DE UN FORMULARIO ... 559
TEMPORIZADORES ... 560
CREAR UNA CLASE DERIVADA DE UN CONTROL ... 563
HERENCIA VISUAL ... 565
El formulario base ... *566*
Agregar un proyecto con un formulario derivado ... *567*
Crear un formulario heredado desde un proyecto independiente ... *570*

MENÚS ... 575

CONTROLES DE TIPO MENÚ ... 575
Menú Principal. MainMenu ... *575*
Menú Contextual. ContextMenu ... *580*
Creación de menús desde código ... *582*

PROGRAMACIÓN CON HEBRAS ... 585

MANIPULACIÓN DE HEBRAS DE EJECUCIÓN ... 585
LA CLASE THREAD ... 586
EJECUTAR UN PROCESO EN UNA HEBRA ... 586
CONTROL DE PROCESOS INDEFINIDOS ... 588
EJECUCIÓN MULTIHEBRA ... 590
EJECUCIÓN MULTIHEBRA DE MÚLTIPLES PROCESOS ... 591
DETECTANDO EL ESTADO DE FINALIZACIÓN ... 593
EJECUCIÓN PARALELA DE PROCESOS A DISTINTOS RITMOS ... 594
SINCRONIZACIÓN DE HEBRAS ... 596
CREAR UN PROCESO DE MONITORIZACIÓN ... 597
INICIOS DE APLICACIÓN CON DOS FORMULARIOS, EMPLEANDO HEBRAS ... 599

FORMULARIOS DE INTERFAZ MÚLTIPLE (MDI) ... 601

APLICACIONES DE ESTILO SDI ... 601
APLICACIONES DE ESTILO MDI ... 601
CREACIÓN DE MENÚS DE TIPO VENTANA, EN FORMULARIOS MDI ... 604
BLOQUEO DE OPCIONES DE MENÚ EN FORMULARIOS MDI ... 606
RECORRER LOS FORMULARIOS HIJOS DE UN MDI ... 607
COMPORTAMIENTO NO MODAL (MODELESS) DE FORMULARIOS ... 608
COMPORTAMIENTO MODAL DE FORMULARIOS ... 608
CONTROLES DE CUADROS DE DIÁLOGO DEL SISTEMA ... 611
ColorDialog ... *611*
FontDialog ... *612*
SaveFileDialog ... *614*
OpenFileDialog ... *615*

FORMULARIOS DEPENDIENTES Y CONTROLES AVANZADOS ... 617

FORMULARIOS DEPENDIENTES Y FIJOS EN PRIMER PLANO ... 617
VALIDACIÓN DE CONTROLES ... 626
CONTROLES AVANZADOS ... 627
IMAGELIST ... 628
TOOLBAR ... 629
STATUSBAR ... 631
DATETIMEPICKER ... 633
NUMERICUPDOWN ... 634
DOMAINUPDOWN ... 635

MONTHCALENDAR 635
LINKLABEL 636
CREACIÓN Y MANIPULACIÓN DE ELEMENTOS EN EJECUCIÓN 637
NOTIFYICON 638

GDI+. ACCESO AL SUBSISTEMA GRÁFICO DE WINDOWS 641

SYSTEM.DRAWING 642
DIBUJO CON LAS CLASES GRAPHICS Y PEN 642
LA CLASE BRUSH 646
DIBUJO DE TEXTO EN EL FORMULARIO 649
PERSONALIZACIÓN DE LA IMAGEN DE FONDO DEL FORMULARIO 650
Manipulación de los eventos de pintado en la clase Form 650
Empleo del control PictureBox 651
MANIPULANDO EL GRADO DE OPACIDAD DEL FORMULARIO 652
MODIFICACIÓN DE LA FORMA DEL FORMULARIO 655
INTEGRANDO ELEMENTOS. UN VISUALIZADOR DE GRÁFICOS 656

ACCESO A DATOS CON ADO .NET 663

COMPARATIVA DE ADO /ADO .NET 664
BENEFICIOS DE ADO .NET 666
Interoperabilidad 666
Mantenimiento 666
Programación 666
Rendimiento 667
Escalabilidad 667
ARQUITECTURA DE DATOS DESCONECTADOS 667
DataSet 669
ADO .NET y XML 669
UNA VISIÓN GENERAL DE ADO .NET 670
ESPACIOS DE NOMBRES Y CLASES EN ADO .NET 671
LAS CLASES CONNECTION 673
LAS CLASES COMMAND 676
LAS CLASES DATAREADER 680

CONJUNTOS DE DATOS Y ENLACE (DATA BINDING) 683

LA CLASE DATASET 683
LAS CLASES DATAADAPTER 686
NAVEGACIÓN Y EDICIÓN DE REGISTROS EN MODO DESCONECTADO 689
DATA BINDING. ENLACE DE DATOS A CONTROLES 694
Tipos de Data Binding 694
Elementos integrantes en un proceso de Data Binding 694
EMPLEO DE DATA BINDING SIMPLE PARA NAVEGAR Y EDITAR DATOS 694

EL CONTROL DATAGRID, RELACIONES Y VISTAS 701

DATAGRID 701
CREACIÓN DE UN DATAGRID A TRAVÉS DE LOS ASISTENTES DEL IDE 703
CONFIGURAR LAS PROPIEDADES DEL DATAGRID 708
CONFIGURAR POR CÓDIGO LAS PROPIEDADES DEL DATAGRID 708
SELECCIÓN DE TABLA EN EL DATAGRID 710
RELACIONES ENTRE TABLAS MEDIANTE OBJETOS DATARELATION 711
Obtener tablas relacionadas mediante código 711
Visualizar datos relacionados en modo maestro-detalle en un DataGrid 713
Mostrar una relación maestro-detalle en dos DataGrid 714
RELACIÓN MAESTRO-DETALLE EN MÚLTIPLES DATAGRID 715
VISTAS Y ORDENACIÓN DE DATOS CON LA CLASE DATAVIEW 716

VISTAS POR CÓDIGO Y DEFAULTVIEW 717
FILTROS CON OBJETOS DATAVIEW 718
BÚSQUEDAS CON DATAVIEW 720
ORDENACIÓN DE FILAS MEDIANTE DATAVIEW 721
OBTENER EL ESQUEMA DE UN DATASET 723

1

Introducción

Una versión largo tiempo esperada

Con la aparición de .NET Framework, y de Visual Basic .NET, como una de las herramientas estrella para el desarrollo sobre esta nueva plataforma de trabajo, estamos asistiendo a una evolución/revolución sin precedentes en el mundo de la informática, que sitúa a este clásico de la programación en una posición difícil de igualar y menos aún de superar.

Visual Basic .NET (VB.NET a partir de ahora), como cada nueva versión de las que han aparecido en el mercado de este producto, incorpora, como es natural, un buen conjunto de novedades. Sin embargo, la inclusión de Visual Basic en el entorno de .NET, añade también un compendio de drásticos cambios para los programadores de versiones anteriores, derivados en su conjunto, de la necesidad de afrontar con garantías de éxito el desarrollo de la nueva generación de aplicaciones para Internet, objetivo perseguido por todas las herramientas de desarrollo actuales.

Tales cambios, como decimos, son necesarios para la plena integración de Visual Basic con el resto de lenguajes del entorno de .NET; un alto porcentaje, suponen la mejora sobre ciertas características del lenguaje y la eliminación de aspectos obsoletos, arrastrados por una compatibilidad, que en ocasiones como la actual, es necesario dejar atrás; en otros casos, se trata de adaptar nuestras costumbres a nuevos modos y hábitos de programar.

Para comprender un poco mejor, la razón que ha llevado a los diseñadores de Microsoft al punto actual, hagamos un breve repaso histórico a la programación con Windows y Visual Basic.

Aquellos difíciles tiempos de la programación en Windows

La aparición de Windows a mediados de los años ochenta, sobre todo a raíz del lanzamiento de la versión 3.1, supuso una gran revolución en el mundo del PC. Los usuarios de esta plataforma, disponían ahora de un entorno gráfico de trabajo, que facilitaba en gran medida su labor y dejaba atrás paulatinamente la aridez del trabajo en el modo comando (la interfaz MS-DOS); ya no era necesario migrar a la plataforma Macintosh para disponer de un entorno de trabajo avanzado.

Sin embargo, el desarrollo de aplicaciones para el nuevo modo gráfico de MS-DOS (aún no era propiamente un sistema operativo), distaba mucho de ser una tarea sencilla y rápida. Aquellos aventurados programadores, que se embarcaban en la gesta de desarrollar una aplicación para Windows, debían prácticamente, hacer borrón y cuenta nueva sobre todo lo que sabían, y comenzar casi, desde cero. Tan radical era el cambio, que hacer el más sencillo programa para que funcionara en Windows, se convertía en la más traumática de las experiencias.

Hasta ese momento, y en líneas generales, todo era más simple en la programación para MS-DOS: la aplicación tomaba el control del sistema operativo, el cuál esperaba las instrucciones del programa para ir ejecutándolo; sólo podíamos tener en ejecución una aplicación en cada momento; el modo gráfico era proporcionado por librerías específicas del lenguaje que estuviéramos utilizando, etc.

Pero la nueva arquitectura de programación de Windows cambiaba todos los esquemas que pudiera conocer el programador: programación basada en eventos y orientada a objetos; modo gráfico proporcionado y gestionado por el sistema y no por el lenguaje; múltiples aplicaciones funcionando simultáneamente; y lo más novedoso, y también más traumático para los programadores, el hecho de que el sistema enviaba información mediante mensajes a nuestra aplicación, a los que debíamos dar una adecuada respuesta, lo que suponía que a partir de ese momento, era el sistema el que controlaba a la aplicación, con lo que se acabaron los tiempos en los que nuestro programa tomaba el control absoluto del sistema operativo.

Sólo para los elegidos

En estos primeros tiempos de la programación para Windows, sólo los llamados *gurús* de C y Windows, que conocían perfectamente todos los trucos y la arquitectura del nuevo entorno operativo de Microsoft, eran capaces de desarrollar las nuevas aplicaciones, para el asombro de los más modestos *programadores de a pie*.

Uno de los grandes problemas para el programador, consistía en que debía centrarse excesivamente en el desarrollo de la parte del interfaz de la aplicación, controlando hasta el más mínimo detalle de lo que el usuario pudiera hacer con una ventana: captura y envío de mensajes desde y hacia las ventanas de la aplicación, gestión de manipuladores de ventanas y contextos de dispositivos para el dibujo de todos los elementos de la aplicación, escritura de los procedimientos de ventana, etc.; el más simple programa que mostrara un mensaje tenía un gran número de líneas de código.

En un escenario como este, en la mayor parte de casos, se desviaba la atención de lo verdaderamente importante en la aplicación: la funcionalidad que necesitábamos dar al usuario. Programar una simple entrada de datos para almacenar en un fichero era toda una odisea.

Por añadidura, tampoco existían herramientas de desarrollo que facilitaran la labor del programador, todo consistía en un puñado de aplicaciones independientes que funcionaban en modo comando: compilador, enlazador, editor de código, etc., lo que hacía que un programador no pudiera alcanzar el mismo nivel de productividad que tenía desarrollando las aplicaciones MS-DOS de aquel entonces.

Esto suponía un grave inconveniente para Microsoft, puesto que el paso previo para popularizar su nuevo entorno de usuario para ordenadores personales, pasaba por la existencia de una comunidad de programadores lo más amplia posible, todos escribiendo aplicaciones para Windows; sin embargo, dada su dificultad, pocos eran los que se lanzaban a tal osado intento.

El proyecto Thunder

Conscientes del problema que entrañaba el que los desarrolladores no migraran de forma masiva a la creación de programas para Windows, Microsoft puso en marcha un proyecto con el nombre clave Thunder (Trueno), encaminado a crear una herramienta de desarrollo que facilitara la escritura de programas para Windows. En 1991, este proyecto dio como fruto la primera versión de Visual Basic (VB a partir de ahora).

Un producto revolucionario

VB 1.0 suponía una forma de encarar el desarrollo de aplicaciones Windows totalmente diferente a lo conocido hasta aquel entonces. Mediante un entorno de desarrollo integrado (IDE) ejecutado desde el propio Windows, cualquier programador, sin necesidad de conocer los aspectos intrincados de Windows y con una mínima curva de aprendizaje, podía crear aplicaciones que hasta esa fecha era potestad reservada sólo a unos pocos.

En esa época, resultaba asombroso cómo de forma prácticamente intuitiva, creábamos un formulario, añadíamos controles, y en definitiva, diseñábamos el interfaz de usuario sin escribir una sola línea de código. La parte correspondiente al código, quedaba reservada para los eventos de los controles que necesitábamos que respondieran a las acciones del usuario.

El gran inconveniente en esta versión y en VB 2.0, era que adolecía de un soporte nativo para manipular bases de datos, puesto que uno de los pilares de las aplicaciones de gestión lo constituye su capacidad de comunicarse con bases de datos para almacenar y recuperar información.

El paso a OLE y ODBC

La siguiente versión, VB 3.0, aportó dos novedades importantes: nos liberó de los limitados controles VBX, hacia el más robusto y flexible modelo de controles OLE, también conocidos como controles OCX; y proporcionó soporte para manejar bases de datos a través de ODBC. Esto supuso, la entrada de Visual Basic en el mundo de las aplicaciones de gestión. Ahora ya no había excusa para desarrollar un mantenimiento de datos bajo Windows de una forma relativamente rápida y sencilla.

Paso a 32 bits y orientación a objeto

Con Windows 95 y su arquitectura basada en 32 bits, se hacía necesario una herramienta de programación, que permitiera aprovechas las características de lo que había dejado de ser una mera capa gráfica sobre MS-DOS, y se había convertido en un sistema operativo por derecho propio.

Tal era el motivo de VB 4.0, una versión que consiguió cambiar muchas opiniones dentro del mundo de la programación.

Hasta el momento, VB era considerado poco más que un juguete dentro de la comunidad de programadores; permitía desarrollar aplicaciones fácilmente sí, pero no era considerado una herramienta *seria*, había muchos aspectos en las aplicaciones que todavía necesitaban ser escritos en C.

La versión 4.0 disponía a su vez de versiones para crear aplicaciones que se ejecutaran para 16 o 32 bits, de forma que ya podíamos crear aplicaciones para el nuevo sistema operativo.

Permitía la programación orientada a objetos, aunque limitada en algunos aspectos. Podíamos crear nuestras propias clases, pero no disponíamos de herencia. Dichas clases podían ser compiladas como servidores OLE, lo que actualmente conocemos como componentes. Esto abría las puertas al programador de VB hacia un nuevo aspecto del desarrollo, ya que hasta la fecha, la creación de componentes estaba reservada a los programadores en C++. Como ventaja añadida, evitaba al programador el esfuerzo y la inversión de tiempo en el aprendizaje de C++.

En cuanto a la gestión de datos, se proporcionaba la jerarquía de objetos RDO (Remote Data Objects), que permitía la creación de aplicaciones de alto rendimiento con acceso a bases de datos cliente/servidor, lo que situaba a VB en el grupo de herramientas de nivel empresarial.

ActiveX y ADO

La versión 5.0 permitía la compilación de las aplicaciones a código nativo, superando la más lenta de versiones anteriores, basada en pseudo-código; como resultado, nuestros programas podían ejecutarse casi tan velozmente como los de C++.

Otro área del desarrollo hasta ese momento reservado a C++ era la creación de controles ActiveX. La versión 5.0 introdujo la posibilidad de crear controles Actives, con lo que ya no era necesario recurrir a C++ para crear nuestros propios controles, superando una nueva limitación.

Respecto al manejo de bases de datos, se incluía una nueva jerarquía de objetos para datos: DAO (Data Access Objects), que facilitaba la manipulación de bases de datos Jet, el formato utilizado por Access.

VB 6 incluía un nuevo modelo de acceso a datos mejorado: ADO (ActiveX Data Objects), cuya finalidad era la de reemplazar a los medios existentes hasta ese momento: RDO y DAO, por una única jerarquía de objetos de acceso a datos de cualquier tipo y en cualquier situación: bases de datos locales, cliente/servidor, acceso a datos a través de Internet, etc. Este modelo de objetos para datos, si bien se conserva en .NET, ha sido profundamente renovado para atender a las exigencias de las aplicaciones actuales.

Programación en Internet

En los últimos tiempos, y más concretamente durante el periodo en el que aparecieron las versiones 5.0 y 6.0 de VB, el desarrollo de aplicaciones para Internet ha tomado un auge espectacular. VB no ha sido ajeno a este factor, y en la versión 6.0, se incluían elementos que intentaban proporcionar al programador, capacidades de acceso a Internet para evitar su cambio a otras herramientas o lenguajes más específicos para la Red.

Los Documentos ActiveX y las Web Classes fueron un buen intento de llevar la programación de Internet a VB, pero su rendimiento en ejecución y complejidad en algunos casos, distaban mucho de

ser la solución idónea a este problema, y el programador que necesitaba crear aplicaciones web, hubo de cambiar a soluciones más específicas, como la programación de páginas ASP.

Por otro lado, un punto fuerte de la programación web, en el que VB sí ha tenido éxito, ha sido el desarrollo de componentes, que encapsulan reglas de negocio, y pueden ser llamados desde páginas ASP. Estos componentes, compilados en formato de DLL, se ejecutan en la capa intermedia del esquema de funcionamiento en tres capas de una aplicación en Internet.

Los problemas para el desarrollo en la Red

A pesar de los intentos de dotarle de capacidades para el desarrollo de aplicaciones web, VB adolecía de algunos aspectos que han influido en que no haya podido entrar en este sector de la programación.

Algunas de estas características son la falta de un pleno soporte para la programación orientada a objetos, en concreto, la falta de herencia; la creación y manipulación multihebra; poca interacción con otros lenguajes como C++; una pobre gestión de errores, etc.

El panorama actual

La entrada en una nueva generación de aplicaciones para Internet, basada cada vez más en dispositivos y servicios trabajando en conjunto para ofrecer un mayor y mejor número de soluciones, hacía cada vez más patente el hecho de que VB necesitaba un cambio (una nueva versión), que le permitiera afrontar todos estos nuevos retos: VB.NET es la respuesta a todas estas necesidades.

Las soluciones aportadas por VB.NET

VB.NET aporta un buen número de características que muchos programadores de VB hemos demandado desde hace largo tiempo. En cierto modo, algunas de estas incorporaciones hemos de agradecérselas a la plataforma .NET, ya que al integrar VB dentro del conjunto de lenguajes de .NET Framework, dichos cambios han sido necesarios, no ya porque los necesitara VB, sino porque eran requisitos derivados de la propia arquitectura de .NET.

Entre las novedades aportadas por VB.NET tenemos plenas capacidades de orientación a objetos (Full-OOP), incluyendo por fin, herencia; Windows Forms o la nueva generación de formularios para aplicaciones Windows; soporte nativo de XML; gestión de errores estructurada; un modelo de objetos para acceso a datos más potente con ADO.NET; posibilidad de crear aplicaciones de consola (ventana MS-DOS); programación para Internet mediante Web Forms; un entorno de desarrollo común a todas las herramientas de .NET, etc.

El precio de la renovación

Pero todas las mejoras efectuadas en VB.NET, han hecho que esta herramienta sufra una renovación tan profunda, que marcan un punto de inflexión importante, haciendo que muchos programadores opinen que estamos ante un nuevo lenguaje, más que una nueva versión.

A pesar de ciertas opiniones negativas, procedentes de los sectores más conservadores de la comunidad VB, debemos recordar que el paso de VB3 a VB4 también supuso importantes y profundos cambios en el modo en el que se desarrollaban las aplicaciones por aquel entonces; sin embargo, todas

aquellas innovaciones han sido asumidas por el colectivo de desarrolladores y en la actualidad sería impensable abordar la realización de un programa sin ellas.

Otro punto a favor de VB.NET consiste en el hecho de que proporciona una utilidad de migración de aplicaciones creadas con versiones anteriores de VB que según las pruebas realizadas es capaz de migrar hasta el 95% del código de una aplicación creada en VB6.

Y lo que es más importante, no es obligatoria la migración de una aplicación escrita por ejemplo en VB6; podemos seguir ejecutando tales programas dentro de .NET Framework, con el inconveniente de que al no ser código gestionado por el entorno de .NET no podrá aprovecharse de sus ventajas.

Muchos programadores argumentarán: -¿Y por qué no incorporar programación web, dejando la facilidad de uso que siempre ha tenido VB?-. La respuesta hemos de buscarla en el apartado anterior.

Si queríamos programación en Internet y todo el nuevo espectro de servicios que se avecinan, era necesario integrar VB como lenguaje del entorno .NET, pero los lenguajes que formen parte de esta plataforma están obligados a cumplir una serie de requisitos, no porque lo pueda necesitar el lenguaje, sino porque es la plataforma la que obliga a ello para poder sacar partido de todas las ventajas de .NET.

Si reflexionamos además, mirando hacia anteriores cambios de versiones, podemos comprobar que desde VB4, todos los cambios han sido en buena medida profundos, para poder adaptarse a las necesidades de los programas en cada momento. Bien es cierto, que esta versión incorpora un cambio más traumático que las otras, pero si sopesamos las nuevas funcionalidades y potencia que obtendrán nuestras aplicaciones, suponemos que la inversión efectuada en adaptarnos merecerá la pena.

Comenzamos

Nos encontramos en un momento muy importante en la historia de la informática en general, y la programación en particular; estamos en el punto de partida de una nueva generación de aplicaciones, que demandan una nueva tecnología, y que gracias al entorno .NET y a VB.NET, como una de sus herramientas integrantes, vamos a poder afrontar con plenas garantías de éxito.

Desde esta obra, intentaremos hacer que la curva de aprendizaje de VB.NET, sea una experiencia grata y amena, tanto para los programadores que se acercan por primera vez a este lenguaje, como para los veteranos, ya curtidos en las lides del mundo de Visual Basic. Bienvenidos a todos.

2

La evolución hacia .NET

Las razones del cambio

Los motivos que han llevado a Microsoft al desarrollo de .NET han sido tanto tecnológicos como estratégicos.

Respecto a las motivaciones tecnológicas, la necesidad de poner a disposición del programador una plataforma de desarrollo con plena potencia para abarcar los requerimientos de las nuevas aplicaciones que están a punto de llegar, y que no soporte incómodos lastres derivados de antiguos modelos de programación, ha desembocado en una tecnología totalmente nueva, que no arrastra pesadas incompatibilidades, pero que sin embargo, permite la ejecución de componentes basados en el anterior modelo de programación. Esto es .NET, una nueva arquitectura para el futuro del desarrollo de aplicaciones, y no, como en un principio pudiera pensarse, una operación más de marketing, que proporciona las herramientas ya conocidas con algunas remodelaciones y *lavados de cara*.

En cuanto a las causas estratégicas, gracias a .NET y a su modelo de distribución de software basado en servicios, Microsoft se sitúa en una posición clave en un mercado que evoluciona hacia la creación de servicios para la web, que serán utilizados por otras aplicaciones mediante un sistema de suscripción o alquiler. Se espera que en este potencial mercado, comiencen a aparecer empresas dedicadas a la producción y publicación de servicios en Internet. La propia Microsoft, ha expresado en este sentido, su intención de convertirse en proveedor de servicios.

La difícil decisión de abandonar anteriores tecnologías

Los herméticos y poco flexibles modelos de programación actuales, impiden cada vez más al programador el abordaje de proyectos para Internet, que le permitan la creación de aplicaciones distribuidas más potentes.

Estos sistemas de trabajo, han evolucionado desde un esquema que integra diversas tecnologías como COM, ASP, ADO, etc., la mayor parte de ellas no pensadas inicialmente para ser ejecutadas en la Red, o que en el caso de ser diseñadas para Internet, arrastran elementos que no estaban pensados para funcionar en la web.

Todos estos elementos, conforman la arquitectura Windows DNA (Distributed interNet Architecture), que hasta la actualidad ha sido el modelo de programación para Internet propugnado por Microsoft.

Por los motivos comentados en el apartado anterior, este modelo ha sido dejado a un lado para dar paso a .NET; lo que no supone una evolución de la actual arquitectura Windows DNA, sino que por el contrario, significa el nuevo comienzo de una arquitectura pensada para la Red.

Antes de describir en qué consiste .NET, hagamos un breve repaso de los problemas que plantea Windows DNA, de manera que podamos comprender mejor, los motivos por los cuales es necesaria la migración hacia la nueva plataforma de Microsoft.

La problemática de Windows DNA

Cuando a mediados de los años 90, Microsoft reorientó su estrategia hacia Internet, carecía de una herramienta de desarrollo potente y rápida para dicho entorno, a diferencia de lo que sucedía dentro de Windows con Visual Basic, a quien nadie podía hacer sombra.

Sin embargo necesitaba un producto para la programación en la Red y lo necesitaba ya. El resultado fue Windows DNA, que era bastante aceptable dado el apremio con el que debía dar respuesta a este sector del desarrollo de aplicaciones, aunque siempre ha adolecido de una falta de integración y facilidad de manejo, siendo un gran calvario para el desarrollador.

ASP

Las páginas ASP (Active Server Pages) son el medio con el que en Windows DNA, podemos programar aplicaciones para Internet utilizando la tecnología de Microsoft.

Aun cuando el resultado conseguido es satisfactorio, el hecho de ser código interpretado, carecer de una herramienta de depuración y poca estructuración suponen un grave paso atrás, máxime cuando todas las herramientas de desarrollo tienden progresivamente hacia un modelo orientado a objetos.

ADO

Este modelo de objetos para el acceso a datos fue diseñado inicialmente para ASP, pero dado su éxito, se traslado también a Visual Basic, para superar los inconvenientes que presentaban los obsoletos DAO y RDO.

El hecho de que se creara en un principio para ASP, puede hacernos pensar que es el medio perfecto para el acceso a datos en Internet; sin embargo, su diseño no se basa totalmente en un modo de acceso desconectado a los datos, ya que para que funcionara con mejor rendimiento dentro del mundo cliente/servidor de las aplicaciones VB, también se puede utilizar estableciendo una conexión permanente con el origen de datos del servidor, lo que supone un claro lastre a la hora de trasladarlo al mundo de Internet, en el que la conexión se establece sólo durante el tiempo que dura la operación a realizar con los datos (obtención, modificación)

Visual Basic

El papel de VB dentro de Windows DNA ha sido fundamentalmente, el de la escritura de componentes para su uso por parte de las páginas ASP de una aplicación web; de hecho, es el lenguaje preferido para el desarrollo de componentes debido a su ya larga tradición como lenguaje sencillo y de fácil manejo.

Microsoft hizo un intento de dotar de un mayor número de características a Visual Basic para que pudiera convertirse en una herramienta de desarrollo integral para Internet; para ello, incorporó las Web Classes, los documentos ActiveX y controles ActiveX, aunque ninguno de ellos obtuvo plena aceptación.

Por un lado, las Web Classes tenían el complejo modelo de programación, mientras que los documentos ActiveX arrojaban unos pobres rendimientos de ejecución. Con respecto a los controles ActiveX, necesitaban de cierto proceso de instalación por parte del servidor, lo que los hacía en muchas situaciones poco operativos. Estas circunstancias han impedido que VB pudiera convertirse en la herramienta de desarrollo para Internet de Microsoft.

Otros factores decisivos que han limitado la plena entrada de VB en la programación web han sido la falta de capacidades multihebra, inexistencia de un interfaz de usuario específico para aplicaciones web, falta de herencia y otras características orientadas a objeto, escasa integración con otros lenguajes, deficiente gestión de errores, etc., aspectos todos, solucionados en VB.NET.

Conflictos con DLL's

La instalación y mantenimiento de los componentes compilados en forma de DLL es otro de los importantes problemas existentes en la actualidad. La actualización de una DLL, cuando se produce un cambio en la misma y los conflictos de versión entre componentes, llevan a una inversión muy importante y grave de tiempo en corregir estos problemas.

Tras los pasos de COM

Una observación de la evolución de COM resulta reveladora y ayuda a comprender el camino que ha llevado hasta la creación de .NET.

El modelo de objetos basado en componentes (COM), se introdujo a mediados de los años 90 como una vía para conseguir un mayor aprovechamiento del código, al situarlo en componentes reutilizables por más de una aplicación.

El gran despegue de COM se produjo con el lanzamiento de VB4, la primera versión de VB que incorporaba algunas características de orientación a objetos (OOP). Gracias a ello, la escritura de componentes se popularizó de una forma muy notable.

A pesar de constituir un gran avance en el mundo de la programación, carecía de herencia, un aspecto muy importante y al que Microsoft anunció un próximo soporte, además de otras características, como el poder disponer de un modelo de objetos unificado que podría ser utilizado en diferentes plataformas; de hecho, se especuló con un cambio de nombre hacia *Common Object Model*, lo cual era muy significativo.

Sin embargo, y en contra de las expectativas, la siguiente versión, DCOM, siguió sin incorporar las características anunciadas, aunque eso no significaba que el equipo de desarrollo de COM no estuviera trabajando en ello.

Para la nueva versión, denominada COM+, se anunciaban cambios radicales en el panorama del desarrollo de componentes, en donde habría plenas capacidades de orientación a objetos (herencia incluida), los componentes se podrían escribir en un amplio abanico de lenguajes soportados por COM, la ejecución se realizaría en un entorno común que se haría cargo de la gestión de memoria y objetos, etc.

Aproximadamente en el mismo espacio de tiempo, otro equipo de desarrollo de Microsoft, después de la finalización de IIS 4, acumuló un conjunto de ideas para la creación de una nueva arquitectura, que provisionalmente se definió como Next Generation Windows Services (NGWS) o Nueva Generación de Servicios para Windows.

Al proyecto NGWS se incorporó Visual Studio y COM+ junto con MTS; sobre estos dos últimos, se comenzó a trabajar en todas las características comentadas antes, de forma que permitieran un entorno de ejecución común para todos los lenguajes de Visual Studio. El resultado fue .NET, y debido a los profundos cambios sufridos por la integración de todos los elementos que lo forman, esta arquitectura no ha derivado directamente de COM, aunque muestra las principales características anunciadas para COM+.

Por todo lo anteriormente comentado, se puede afirmar que .NET es una nueva tecnología, y no una evolución del modelo Windows DNA; construida sin el peso de la compatibilidad hacia tecnologías anteriores, pero que ha sabido aprovechar las mejores ideas de los elementos existentes en la actualidad.

3

.NET Framework, nuevos cimientos para la nueva generación de aplicaciones

Algo está cambiando

El mundo del desarrollo de aplicaciones se encuentra sumido en una nueva etapa de transformación y evolución hacia nuevos esquemas de trabajo.

Los factores determinantes de dicho cambio los podemos encontrar en la necesidad de utilizar Internet como vehículo de intercambio por parte de diversos sectores de la economía.

Las empresas requieren establecer relaciones comerciales más dinámicas con sus clientes, de modo que su volumen de negocio se incremente a través del canal de ventas electrónico (el denominado comercio electrónico o e-commerce). Por otro lado también necesitan unas relaciones empresariales más ágiles en este mismo marco del ciberespacio (el llamado B2B o Bussiness to bussiness).

Aparte de todos estos elementos, nos encontramos con que el usuario de este medio, Internet, dispone de dispositivos cada vez más sofisticados para desplazarse por la Red, no sólo el PC; y además, exige que todos ellos permitan un acceso rápido y sencillo, a múltiples aplicaciones simultáneamente, con un mayor grado de interacción, y obteniendo información de un amplio conjunto de fuentes de datos; todo esto, naturalmente, sin los tradicionales esfuerzos de configuración que requieren algunas aplicaciones.

Con el paso del tiempo, Internet se ha convertido en el principal entorno de trabajo para el desarrollo de aplicaciones que gestionan información, haciendo que su alcance sea mayor que ningún otro medio

hasta el momento. Baste pensar, que con un simple dispositivo que tenga acceso a Internet (léase un PC) y un programa navegador, es posible acceder a infinidad de sitios web basados en este paradigma.

Sin embargo, actualmente, la comunicación entre servidores es complicada (sobre todo si residen en plataformas distintas), y la integración de aplicaciones en dispositivos que no sean el típico PC, es limitada con las herramientas disponibles hasta la fecha. Pero no desesperemos, nos encontramos en un momento crucial, en el que todos esos inconvenientes pueden ser salvados gracias a un nuevo avance tecnológico: Microsoft .NET.

¿Qué es .NET?

.NET es toda una nueva arquitectura tecnológica, desarrollada por Microsoft para la creación y distribución del software como un servicio. Esto quiere decir, que mediante las herramientas de desarrollo proporcionadas por esta nueva tecnología, los programadores podrán crear aplicaciones basadas en servicios para la web.

Las características principales que conforman .NET son las siguientes:

- La plataforma .NET Framework, que proporciona la infraestructura para crear aplicaciones y el entorno de ejecución para las mismas.

- Los productos de Microsoft enfocados hacia .NET, entre los que se encuentran Windows .NET Server, como sistema operativo que incluirá de forma nativa la plataforma .NET Framework; Visual Studio .NET, como herramienta integrada para el desarrollo de aplicaciones; Office .NET; b.Central para .NET, etc.

- Servicios para .NET desarrollados por terceros fabricantes, que podrán ser utilizados por otras aplicaciones que se ejecuten en Internet.

Existen adicionalmente un conjunto de productos, que bajo la etiqueta de Servidores Empresariales para .NET (.NET Enterprise Servers) se incluyen dentro de la estrategia .NET. Entre estos productos podemos encontrar a SQL Server 2000, BizTalk Server, Commerce Server 2000, etc. Sin embargo, hemos de hacer una puntualización importante: estos productos no están basados en .NET Framework, pueden funcionar dentro del entorno de ejecución de .NET Framework, pero el único producto actualmente desarrollado bajo el nuevo entorno es Visual Studio .NET.

Gracias a .NET y a su modelo de desarrollo basado en servicios, se flexibiliza y enriquece el modo en el que hasta ahora se construían aplicaciones para Internet. La idea que subyace bajo esta tecnología, es la de poblar Internet con un extenso número de aplicaciones, que basadas en servicios para la web (Web Services), formen un marco de intercambio global, gracias a que dichos servicios están fundamentados en los estándares SOAP y XML, para el intercambio de información.

En este sentido, un programador puede crear Web Services para que sean utilizados por sus propias aplicaciones a modo de componentes (pero de una forma mucho más avanzada que empleando el modelo COM clásico), siguiendo una estructura de programación ya conocida. Ver Figura 1.

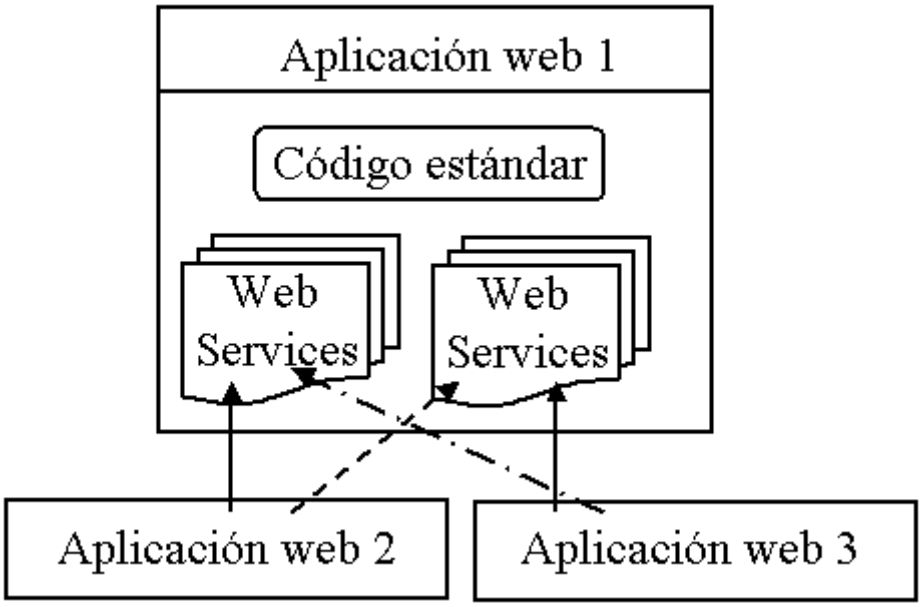


Figura 1. Esquema de funcionamiento de aplicación web incluyendo Web Services.

Sin embargo, los Web Services traen de la mano un nuevo modelo de distribución del software; el basado en el desarrollo y publicación de Web Services y en la suscripción a los mismos por parte de otras aplicaciones, potenciales usuarios de tales servicios. Ver Figura 2.

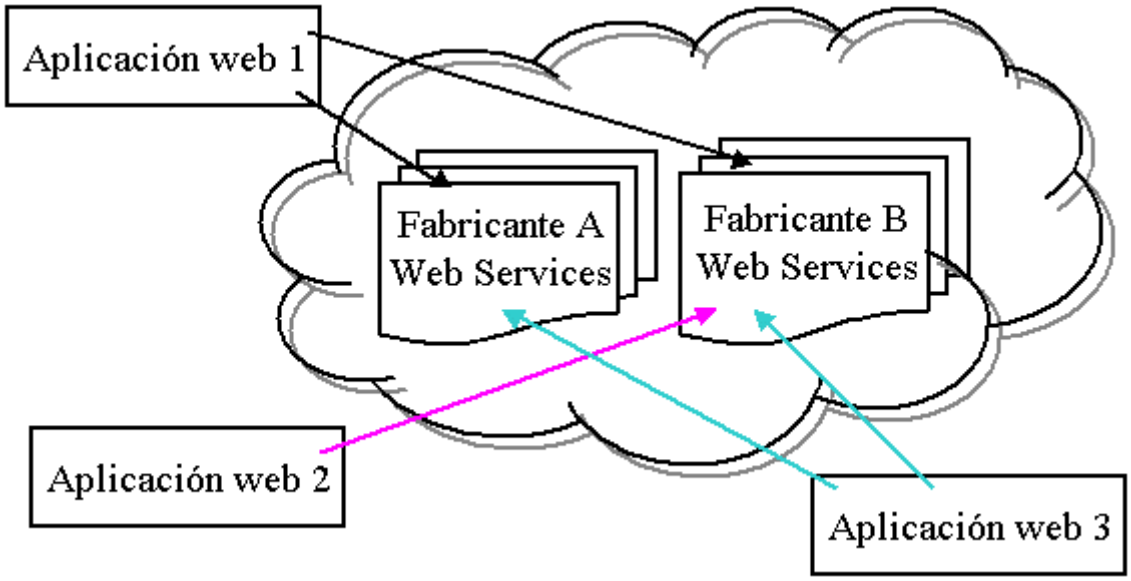


Figura 2. Interacción de aplicaciones con Web Services publicados en Internet.

Los fabricantes de software, pueden de esta manera, dedicarse a la creación de servicios web y a su alquiler. Nace de esta manera, la figura del proveedor de servicios web.

Dado el esquema anterior, el programador puede construir sus aplicaciones a base de Web Services, reduciendo significativamente el tiempo y esfuerzo en el desarrollo.

.NET Framework

.NET Framework constituye la plataforma y elemento principal sobre el que se asienta Microsoft .NET. De cara al programador, es la pieza fundamental de todo este nuevo modelo de trabajo, ya que proporciona las herramientas y servicios que necesitará en su labor habitual de desarrollo.

.NET Framework permite el desarrollo de aplicaciones a través del uso de un conjunto de herramientas y servicios que proporciona, y que pueden agruparse en tres bloques principales: el Entorno de Ejecución Común o Common Language Runtime (CLR a partir de ahora); la jerarquía de clases básicas de la plataforma o .NET Framework Base Classes; y el motor de generación de interfaz de usuario, que permite crear interfaces para la web o para el tradicional entorno Windows, así como servicios para ambos entornos operativos. La Figura 3 muestra un diagrama con la distribución de elementos dentro del entorno de .NET Framework.

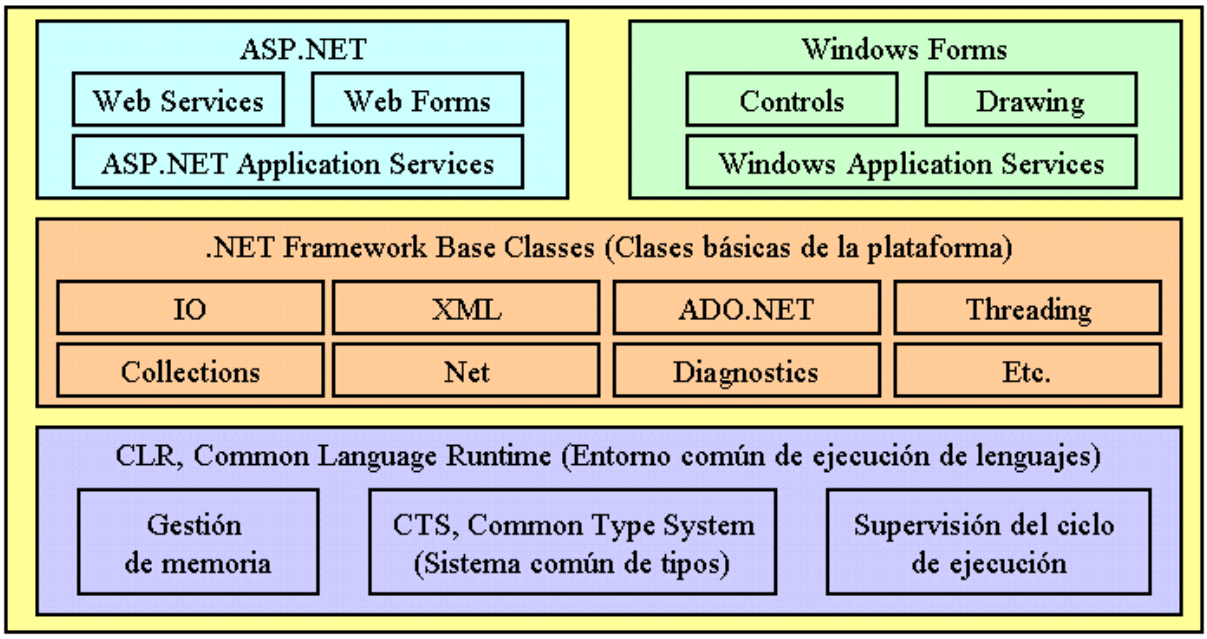


Figura 3. Esquema de componentes dentro de la plataforma .NET Framework.

En la base del entorno de ejecución, se encuentra el CLR, que constituye el núcleo de .NET Framework, encargándose de la gestión del código en cuanto a su carga, ejecución, manipulación de memoria, seguridad, etc.

En el nivel intermedio, se sitúa la jerarquía de clases básicas del entorno de ejecución, que constituyen un sólido API de servicios a disposición del programador, para multitud de tareas como, gestión del sistema de ficheros, manipulación multihebra, acceso a datos, etc.

Finalmente, en el nivel superior, encontramos las clases que permiten el diseño del interfaz de usuario de nuestras aplicaciones. Si necesitamos desarrollar aplicaciones para Internet, utilizaremos ASP.NET, que nos provee de todo lo necesario para crear aplicaciones para la Red: web forms, web services, etc.

Y no piense el programador tradicional de Windows, que todo en .NET Framework es programación para Internet. La plataforma no se ha olvidado de este colectivo de programadores, que necesitan desarrollar programas para este sistema operativo, y pone a su disposición los denominados Windows Forms, la nueva generación de formularios, con características avanzadas y muy superiores a las del motor de generación de formularios de VB6.

Adicionalmente, existe la posibilidad de que necesitemos servicios del sistema que no requieran interfaz de usuario en absoluto. Este aspecto también está contemplado por la plataforma, permitiéndonos, por ejemplo, la creación de servicios para Windows 2000 y NT.

El CLR, Common Language Runtime

El Entorno de Ejecución Común de Lenguajes o CLR (Common Language Runtime), representa el alma de .NET Framework y es el encargado de la ejecución del código de las aplicaciones.

A continuación se enumeran algunas de las características de este componente de la plataforma:

- Proporciona un desarrollo de aplicaciones más sencillo y rápido gracias a que gran parte de las funcionalidades que tradicionalmente debía de crear el programador, vienen implementadas en el entorno de ejecución.
- Administra el código en tiempo de ejecución, en todo lo referente a su carga, disposición en memoria, recuperación de memoria no utilizada a través de un recolector de memoria, etc.
- Implementa características de gestión a bajo nivel (administración de memoria, por ejemplo), que en ciertos lenguajes, eran labor del programador.
- Proporciona un sistema común de tipos para todos los lenguajes del entorno.
- Gestiona la seguridad del código que es ejecutado.
- Dispone de un diseño abierto a lenguajes y herramientas de desarrollo creadas por terceros fabricantes.
- Facilita enormemente la distribución e instalación de aplicaciones, ya que en teoría, es posible instalar una aplicación simplemente copiando los ficheros que la componen en uno de los directorios del equipo en el que se vaya a ejecutar, eliminando los temibles conflictos de versiones entre librerías, problema conocido también con el nombre de *Infierno de las DLL* o *DLL Hell*.

La Figura 4 muestra un esquema de la organización interna del CLR.

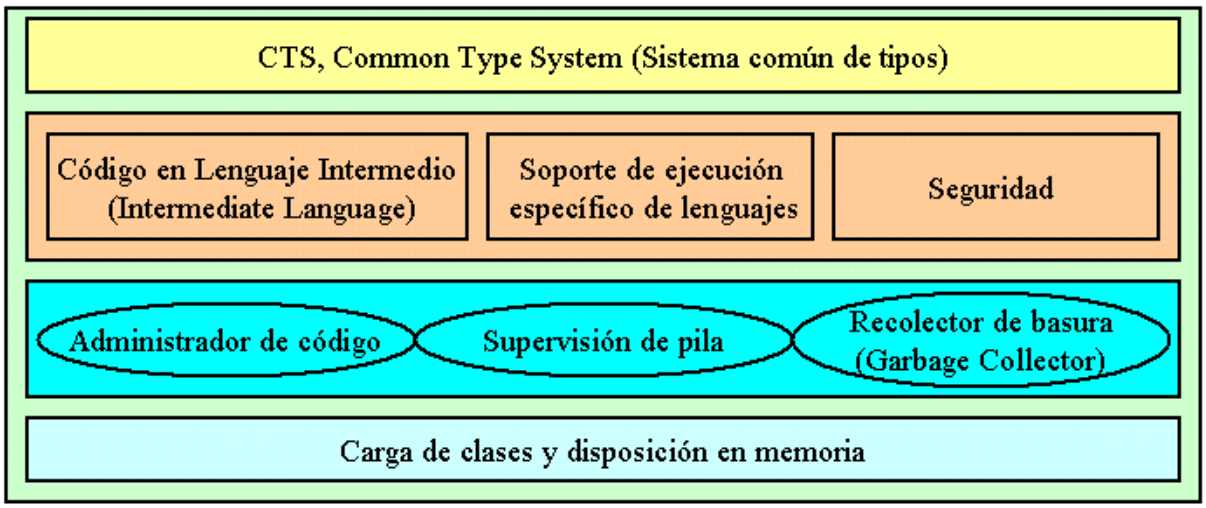


Figura 4. Esquema de elementos dentro del CLR.

En los siguientes apartados, haremos una descripción de los elementos y características más destacables del CLR, que permitan al lector obtener una visión global del mismo, y de las ventajas de escribir programas para este entorno de ejecución.

El CTS, Common Type System

El Sistema Común de Tipos o CTS (Common Type System), es el mecanismo del CLR que permite definir el modo en que los tipos serán creados y manipulados por el entorno de ejecución de .NET Framework.

Entre las funcionalidades que comprende, podemos destacar la integración de código escrito en diferentes lenguajes; optimización del código en ejecución; un modelo de tipos orientado a objeto, que soporta múltiples lenguajes; y una serie de normas que aseguran la intercomunicación entre objetos.

El sistema común de tipos (CTS a partir de ahora), como hemos indicado, permite definir o diseñar el modo en cómo el código de la aplicación será ejecutado, pero no se encarga directamente de su ejecución; dicho de otro modo, el CTS le dice al CLR cómo quiere que sea ejecutado el código.

Un ejemplo de las ventajas del CTS, consiste en que desde un lenguaje como VB.NET, podemos instanciar un objeto de una clase escrita en otro lenguaje como C#; y al hacer una llamada a uno de los métodos del objeto, no es necesario realizar conversiones de tipos en los parámetros del método, funcionando todo de forma transparente.

¿Qué es un tipo dentro de .NET Framework?

Al mencionar el sistema de tipos de la plataforma .NET, podemos pensar de un modo inmediato, que se trata sólo del conjunto de tipos de datos con que podemos declarar variables en nuestro código; sin embargo, el CTS, va mucho más allá y se extiende a cualquier elemento que pueda ser ejecutado dentro del entorno.

Por tal motivo, en el contexto de .NET Framework, un tipo se puede definir como una entidad de código ejecutada dentro del CLR; entendiendo por entidad de código, aquella a partir de la cual creamos una instancia y manejamos posteriormente en el programa como un objeto.

De todo lo anterior podemos obtener dos conclusiones:

- Todas las implementaciones de clases, interfaces, estructuras, etc., ya sean nativas de la plataforma o creadas por el programador, se pueden considerar tipos válidos de .NET.

- Todos los tipos que manipulamos dentro de .NET Framework son objetos.

En la Figura 5 se muestra un esquema de funcionamiento del CTS; en él, tenemos una aplicación en VB.NET y otra en C#, en las que ambas declaran y crean dos variables; una pertenece a un tipo de dato de la plataforma y otra a una clase. En esta situación, el CTS se encarga de dar las oportunas instrucciones sobre como instanciar y proporcionar el dato y el objeto a cada una de las aplicaciones cuando sean ejecutadas, con la ventaja de que no es necesario tener una implementación específica para cada lenguaje, al disponer de un sistema de tipos unificado, y un motor de manipulación de esos tipos, que es el CTS.

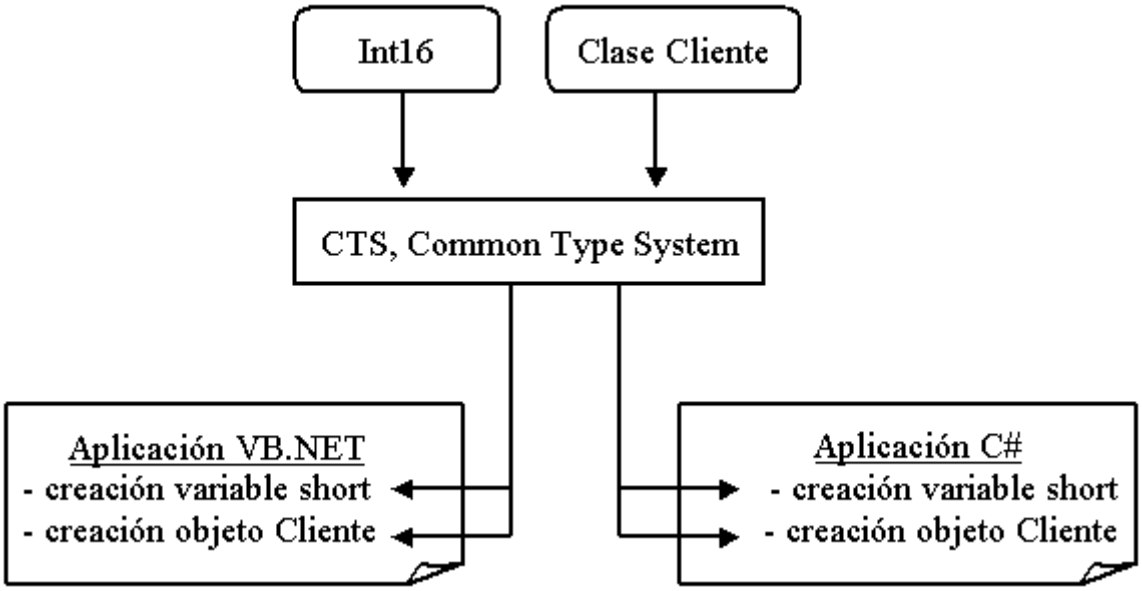


Figura 5. Esquema de funcionamiento del CTS.

Los tipos de datos son objetos

Dentro de .NET Framework, todos los tipos de datos están implementados como clases, de ahí el hecho de que cuando declaremos una variable en el código, esa variable sea además, un objeto de la clase relacionada con el tipo de dato que contiene, disponiendo de propiedades y métodos al igual que cualquier otro objeto. Ver Código fuente 1.

```
Dim sNombre As String
sNombre = "coche"
MessageBox.Show(sNombre.Length) ' devuelve 5
MessageBox.Show(sNombre.ToUpper()) ' devuelve COCHE
```

Código fuente 1. Manipulación de variable como objeto

En el Código fuente 1, escrito en VB.NET, declaramos una variable de tipo String (cadena de caracteres), y a continuación le asignamos un valor; hasta aquí, todo igual que en versiones anteriores. Pero ahora viene lo novedoso, ya que manipulamos la variable igual que un objeto, obteniendo la longitud de su valor mediante la propiedad Length y convertimos su valor a mayúsculas ejecutando el método ToUpper(); en ambos casos mostramos el resultado usando un objeto MessageBox.

En este fuente y otros escritos en VB.NET que utilicemos a lo largo de este tema, el lector percibirá cambios en la sintaxis del lenguaje, motivados por la nueva versión de VB. Todas estas novedades se comentarán en los temas dedicados al lenguaje y su implementación orientada a objeto.

La Tabla 1 muestra una relación de los principales tipos de datos de .NET Framework y su correspondencia específica con VB.NET.

Tipo de dato (Nombre de clase)	Tipo de dato en VB.NET	Descripción
Byte	Byte	Entero sin signo de 8 bit
SByte	SByte (tipo de dato no nativo)	Entero sin signo de 8 bit (tipo no acorde con el CLS)
Int16	Short	Entero con signo de 16 bit
Int32	Integer	Entero con signo de 32 bit
Int64	Long	Entero con signo de 64 bit
UInt16	UInt16 (tipo de dato no nativo)	Entero sin signo de 16 bit (tipo no acorde con el CLS)
UInt32	UInt32 (tipo de dato no nativo)	Entero sin signo de 32 bit (tipo no acorde con el CLS)
UInt64	UInt64 (tipo de dato no nativo)	Entero sin signo de 64 bit (tipo no acorde con el CLS)
Single	Single	Numero con coma flotante de precisión simple, de 32 bit
Double	Double	Numero con coma flotante de precisión doble, de 64 bit
Boolean	Boolean	Valor lógico
Char	Char	Caracter Unicode de 16 bit
Decimal	Decimal	Valor decimal de 96 bit
IntPtr	IntPtr (tipo de dato no nativo)	Entero con signo con tamaño dependiente de la plataforma: 32 bit en plataformas de 32 bit y 64 bit en plataformas de 64 bit (tipo no acorde con el CLS)
UIntPtr	UIntPtr (tipo de dato no nativo)	Entero sin signo con tamaño dependiente de la plataforma: 32 bit en plataformas de 32 bit y 64 bit en plataformas de 64 bit (tipo no acorde con el CLS)
String	String	Cadena de caracteres

Tabla 1. Tipos de datos de .NET Framework con sus correspondencias en VB.NET.

Debemos aclarar, no obstante, que el tipo String no se englobaría dentro de los tipos primitivos del lenguaje, ya que realmente, una variable de tipo String, lo que contiene es un array de tipos Char; sin embargo, nosotros podemos seguir manipulando cadenas de caracteres del mismo modo en el que lo

hacíamos en versiones anteriores de VB, ya que el entorno se encarga de gestionar el array de valores Char que una cadena contiene.

Categorías de tipos

Los tipos creados por el CTS pueden clasificarse en dos grupos principales, según el modo en el que se almacenan y manipulan en memoria:

- **Tipos por valor.** Un tipo creado por valor, almacena un dato que puede ser accedido de forma directa. Los tipos por valor se organizan a su vez en varios subgrupos, como son los tipos de datos nativos de la plataforma .NET, tipos de datos creados por el programador y tipos enumerados.

- **Tipos por referencia.** Un tipo creado por referencia, contiene la dirección de memoria en donde reside un dato. Para acceder a dicho dato, lo hacemos de forma indirecta utilizando esa dirección de memoria o referencia. Los tipos por referencia se organizan a su vez en varios subgrupos, como son las clases propias de la plataforma, las clases creadas por el programador, interfaces, delegates, etc.

La Figura 6 muestra un esquema con la organización de tipos de .NET Framework.

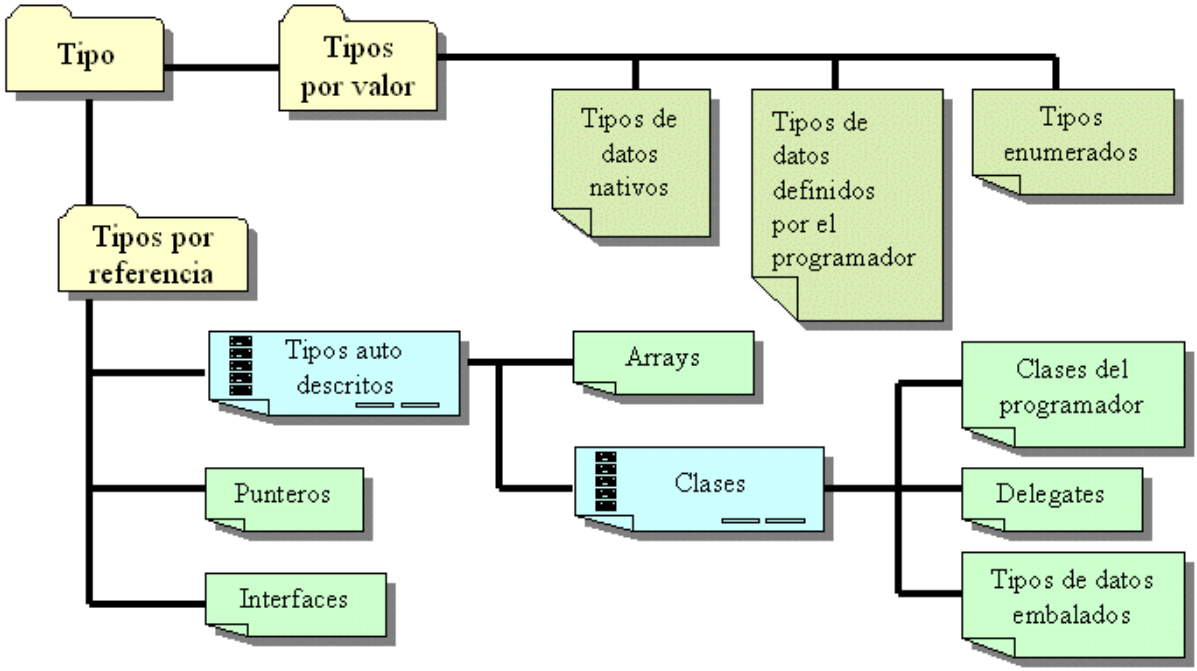


Figura 6. Organización de tipos de .NET Framework.

La disposición de los datos en la memoria

Explicado de un modo muy elemental, cuando ejecutamos una aplicación, los datos que utiliza dicha aplicación se sitúan en memoria. La memoria se divide en dos zonas: una denominada *pila* o *stack*, pequeña y compacta pero de muy veloz acceso a datos y rápido proceso; y otra denominada *montón* o *heap*, de mayor tamaño pero más lenta.

El modo en como el CLR maneja los tipos en la memoria, tiene una gran importancia de cara a conseguir el mayor rendimiento posible en la ejecución del programa. A pesar de que esta es una tarea que gestiona de forma automática el entorno de ejecución, pensamos que es interesante que el lector conozca su técnica, cuanto menos, de una forma genérica.

Cuando se crea un tipo por valor, una variable de tipo numérico, por ejemplo; el valor de dicha variable se sitúa en la pila, de forma que su acceso es directo. Al destruir un tipo por valor, el valor que se almacena en la pila también es destruido. Si asignamos una variable de estas características a otra, se crea en memoria una copia del valor original, con lo que tenemos en este caso dos tipos o variables con valores iguales. Un tipo por valor no puede tener un valor nulo.

Cuando creamos un tipo por referencia, la instancia de una clase (un objeto) que asignamos a una variable, por ejemplo; dicho tipo se sitúa en el montón. Una variable de este tipo contiene la referencia a un valor, no el propio valor, por lo que si asignamos una variable que contiene un tipo por referencia a otra variable, se dice que ambas apuntan o se refieren al mismo valor. Un tipo por referencia sí puede contener un valor nulo.

El Código fuente 2 muestra un ejemplo de creación y asignación de valores a cada uno de estos tipos.

```
Public Class Cliente

    Public Calculo As Long

End Class

Module Gestion

    Public Sub Main()
        ' declarar dos variables (tipos por valor)
        Dim ImportePrim As Long
        Dim ImporteSeg As Long

        ImportePrim = 100
        ImporteSeg = ImportePrim

        ' las dos variables tienen ahora el mismo valor
        ' ImportePrim --> 100
        ' ImporteSeg --> 100

        ' asignamos un nuevo valor a una
        ' de las variables
        ImporteSeg = 728

        ' las dos variables tienen ahora distinto valor
        ' ImportePrim --> 100
        ' ImporteSeg --> 728

        ' ------------------------------------------
        ' declarar dos objetos (tipos por referencia)
        Dim oClienteUno As New Cliente()
        Dim oClienteDos As Cliente

        oClienteUno.Calculo = 85000
        ' al asignar un objeto a otra variable
        ' ambas variables apuntan al mismo objeto
        ' o dirección de memoria
        oClienteDos = oClienteUno

        ' los dos objetos tienen el mismo valor en la propiedad
        ' oClienteUno.Calculo --> 85000
        ' oClienteDos.Calculo --> 85000
```

```
        ' asignamos un nuevo valor a la propiedad
        ' en uno de los objetos
        oClienteDos.Calculo = 120000

        ' los dos objetos tienen el mismo valor
        ' en la propiedad,
        ' ya que ambos apuntan
        ' a la misma referencia en memoria
        ' oClienteUno.Calculo --> 120000
        ' oClienteDos.Calculo --> 120000

    End Sub

End Module
```

Código fuente 2. Creación y manipulación de tipos por valor y referencia en código.

Como acabamos de observar, las variables Long, son tipos por valor que contienen valores independientes; por el contrario, las variables con los objetos de la clase Cliente, por el hecho de haber asignado una de ellas a la otra, apuntan al mismo lugar o referencia.

Otro detalle importante a destacar de este fuente es el manejo de valores nulos. Como hemos comentado, los tipos por valor no pueden tener valores nulos, por lo que aunque no se aprecie en el fuente, las variables Long, al ser creadas, tienen como valor inicial el cero, mientras que las variables con los objetos Cliente, al ser instanciadas, sí contienen un valor nulo o Nothing, como se denomina en VB.NET.

Representado de una forma gráfica, la disposición en memoria del anterior código fuente quedaría como se muestra en el esquema de la Figura 7.

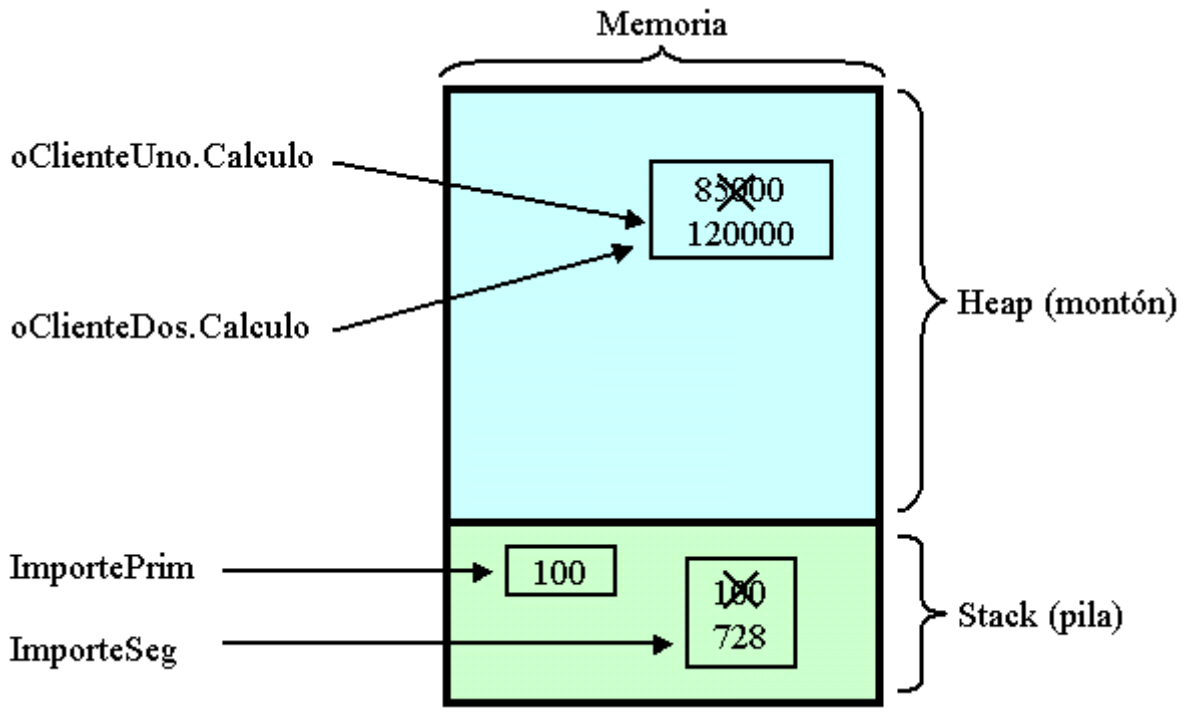


Figura 7. Disposición en memoria de tipos por valor y referencia.

Como podemos comprobar, la relación que la memoria tiene con respecto a los tipos de .NET es muy importante, ya que dependiendo de donde sean ubicados, se conseguirá un rendimiento mas o menos óptimo en la ejecución del programa.

Embalaje y desembalaje de tipos por valor

La operación de *embalaje* de un tipo por valor, también denominada *boxing*, consiste en la conversión de un tipo por valor a un tipo Object. El resultado es una variable Object almacenada en la pila, y una copia del tipo por valor almacenado en el montón, al que apunta el objeto. La Figura 8 muestra un esquema de funcionamiento de esta técnica.

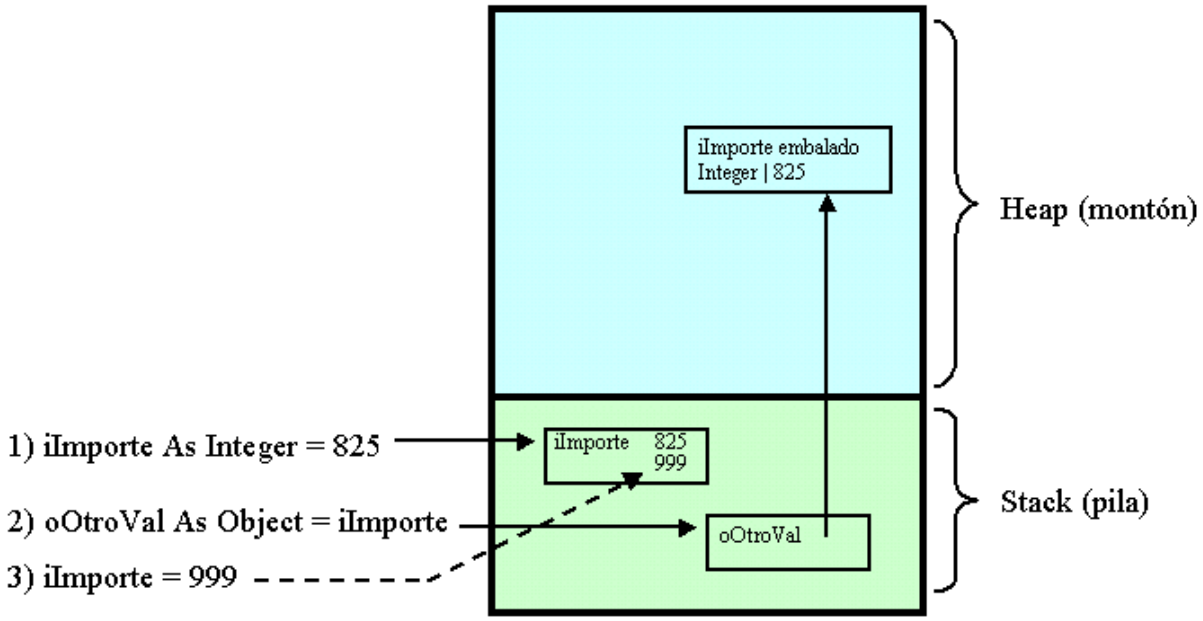


Figura 8. Embalaje o boxing de un tipo por valor.

Traducido a código VB.NET, el anterior esquema quedaría como muestra el Código fuente 3.

```
Public Class Embalar
    Public Shared Sub Main()
        ' tipo por valor
        Dim iImporte As Integer = 825

        ' embalaje:
        ' asignamos el tipo por valor a
        ' una variable Object
        Dim oOtroVal As Object = iImporte

        ' si cambiamos el contenido del
        ' tipo por valor, la variable Object
        ' mantiene el valor original ya que
        ' los valores de ambas variables
        ' son copias independientes
        iImporte = 999

        ' los valores actuales de la variables
        ' serían los siguientes:
```

```
        ' iImporte --> 999
        ' oOtroVal --> 825
    End Sub

End Class
```

Código fuente 3. Embalaje de un tipo por valor.

El proceso opuesto al anterior, denominado *desembalaje* o *unboxing*, consiste en tomar un tipo Object y convertirlo a un tipo por valor.

Tomando el ejemplo anterior, si queremos volver a convertir la variable Object a un tipo por valor, creamos un nuevo tipo por valor y le asignamos la variable Object, creándose una copia del valor en el nuevo tipo por valor. La Figura 9 muestra como quedaría este proceso.

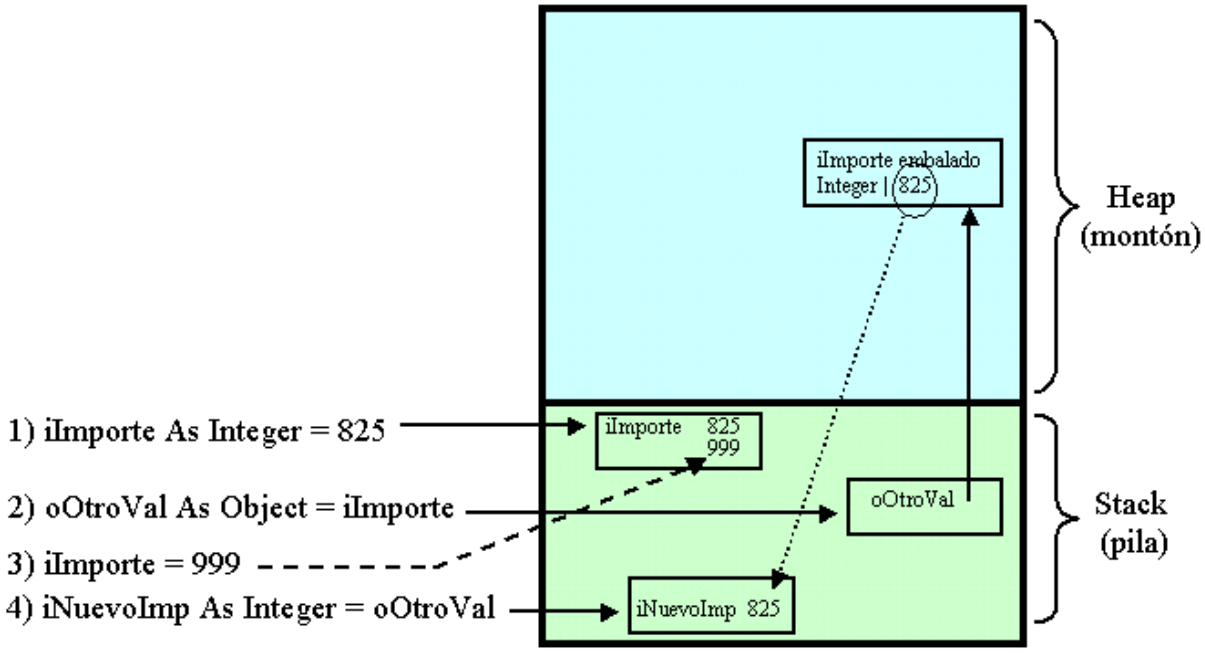


Figura 9. Desembalaje o unboxing de un tipo Object.

El código necesitaría también ser completado como se muestra en el Código fuente 4.

```
Public Class Embalar
    Public Shared Sub Main()
        ' tipo por valor
        Dim iImporte As Integer = 825

        ' embalaje:
        ' asignamos el tipo por valor a
        ' una variable Object
        Dim oOtroVal As Object = iImporte

        ' si cambiamos el contenido del
        ' tipo por valor, la variable Object
        ' mantiene el valor original ya que
        ' los valores de ambas variables
        ' son copias independientes
```

```
        iImporte = 999

        ' los valores actuales de la variables
        ' serían los siguientes
        ' iImporte --> 999
        ' oOtroVal --> 825


        '-----------------------
        ' desembalaje:

        ' crear un nuevo tipo por valor
        ' y asignarle el tipo Object
        Dim iNuevoImp As Integer = oOtroVal

    End Sub

End Class
```

Código fuente 4. Desembalaje o unboxing de un tipo Object.

Metadata (metadatos)

Durante el diseño de .NET Framework, se hizo evidente que una aplicación necesitaba, además de su propio código ejecutable, información adicional sobre la propia aplicación, que pudiera ser utilizada por el entorno de ejecución para funcionalidades diversas.

Para resolver este problema, se optó por incluir toda esta información complementaria dentro de la propia aplicación. A la información que va incluida en la aplicación pero que no forma parte del código ejecutable se le denomina metadatos, y con esta técnica obtenemos aplicaciones o componentes auto-descritos.

Los metadatos son creados por el compilador del lenguaje utilizado en cada caso y grabados dentro del fichero resultante (EXE o DLL) en formato binario, siendo el CLR el encargado de recuperarlos en el momento que los necesite.

Algunos de los datos proporcionados por los metadatos de una aplicación son la descripción del ensamblado (trataremos los ensamblados posteriormente) junto a su versión, clave y tipos que lo componen (clases, interfaces, etc.).

Soporte multi-lenguaje

Uno de los puntos clave del CLR es que está diseñado para soportar múltiples lenguajes, permitiendo así unos elevados niveles de integración entre los mismos. Con tal motivo, .NET Framework proporciona los siguientes lenguajes con sus correspondientes compiladores para la escritura de aplicaciones:

- VB.NET.
- C#.
- C++ con Extensiones Administradas.
- JScript.NET.

Por integración de lenguajes podemos definir algo tan poderoso como el hecho de escribir una clase en C#, y heredar de dicha clase desde VB.NET. Esto permite formar grupos de trabajo heterogéneos, en los que cada integrante del grupo, puede escribir el código de una aplicación en el lenguaje de su preferencia. Gracias a que el entorno de ejecución es común, y el código compilado no pasa directamente a código ejecutable puro, sino a un código intermedio (lo veremos más adelante), podemos crear nuestros programas en el lenguaje con el que nos sintamos más cómodos en cuanto a sintaxis y prestaciones, por ejemplo VB.NET; con la ventaja de que la velocidad de ejecución será muy parecida a la obtenida habiendo escrito el código en otro lenguaje en principio más rápido como C++ o C#.

El CLS (Common Language Specification)

La integración entre lenguajes mencionada en el anterior apartado, puede llevar a preguntarnos cómo es posible conseguir que lenguajes de distinta naturaleza y sintaxis *se entiendan*.

La respuesta la hallamos en la Especificación Común de Lenguajes o CLS (Common Language Specification), que consiste en un conjunto de características comunes, que deben cumplir todos los lenguajes de la plataforma, para poder integrarse entre sí.

Esto tiene varias finalidades, que describimos a continuación:

- **Independencia del lenguaje**. En muchas ocasiones el programador se ve obligado a escribir el código en un lenguaje que no es de su agrado; la causa de ello es que dicho lenguaje le provee de funcionalidades de las cuales carece su lenguaje preferido. Con .NET, esto no ocurre, puesto que es la propia plataforma la que proporciona la funcionalidad de modo independiente al lenguaje, por lo que podemos escribir nuestras aplicaciones utilizando el lenguaje con el que nos sintamos más cómodos, ya que el resultado será el mismo.

- **Integración entre lenguajes**. Es posible escribir, por ejemplo, una librería de clases en un lenguaje, y utilizarla desde otro lenguaje distinto (siempre que ambos lenguajes cumplan con las normas del CLS). Este concepto no es nuevo, hasta ahora también podíamos escribir una librería en C++ y utilizarla desde VB, pero gracias al CLS, se extiende y se potencia este modo de trabajo, ya que al basarse los lenguajes en un conjunto de reglas comunes, el acceso en el caso antes mencionado, a una librería de clases, se facilita enormemente desde cualquier otro lenguaje creado en base al CLS.

- **Apertura a nuevos lenguajes**. Finalmente, al ser esta, una especificación abierta, es posible incorporar a .NET Framework nuevos lenguajes, aparte de los actualmente disponibles, y no sólo creados por Microsoft, sino por cualquier otro fabricante. Mediante el CLS, un fabricante de software sabe qué requisitos debe observar un nuevo lenguaje que él desarrolle, para poder integrase en el entorno de .NET Framework.

  Terceros fabricantes ya han anunciado en este sentido, su intención de proporcionar nuevos lenguajes para .NET; de esta forma aparecerán progresivamente versiones para esta plataforma de Cobol, Perl, Smalltalk, etc., en una lista en la que actualmente figuran más de veinte lenguajes candidatos.

Ejecución administrada

La ejecución administrada se trata de un conjunto de elementos existentes en .NET Framework, que supervisan el código del programa durante su ejecución dentro del CLR, asegurándose de que el

código cumple todos los requisitos para poder hacer uso de los servicios proporcionados por el entorno de ejecución, y beneficiarse de sus ventajas.

Código administrado

El código que escribamos orientado a utilizar todas las cualidades del CLR se denomina código administrado. Por defecto el código escrito en VB.NET, C# y JScript.NET es administrado, con lo que el programador no debe preocuparse en configurar de manera especial su proyecto.

Por el contrario, el código escrito en C++ no es administrado por defecto, lo que significa que el entorno no lo supervisa y no garantiza su fiabilidad al ser ejecutado por el CLR. Si el programador de C++ quiere que su código sea administrado debe utilizar las extensiones administradas que la plataforma proporciona para este lenguaje y activarlas a través de una opción del compilador.

El hecho de que el entorno realice labores de comprobación sobre el código, supone evidentemente, una labor extra que repercute sobre el rendimiento final a la hora de ejecutar el programa. Sin embargo, las pruebas realizadas ofrecen como resultado una pérdida de un 10% en el rendimiento del código administrado con respecto al código no administrado.

Teniendo en cuenta los niveles de seguridad que nos ofrece el código administrado y dado que la velocidad de los procesadores evoluciona, esta pérdida de rendimiento que supone la ejecución administrada posiblemente no sea significativa en un corto plazo de tiempo.

Datos administrados

De forma similar al código, los datos administrados son datos los datos de la aplicación gestionados en memoria por el CLR a través de un mecanismo denominado recolector de basura.

Al igual que en el punto anterior, los datos son administrados por defecto en las aplicaciones escritas en VB.NET, C# y JScript.NET. Si utilizamos en cambio C++, los datos de la aplicación no son administrados por defecto, debiéndolo indicar en el código del programa.

Recolección de memoria no utilizada

Durante la ejecución de un programa, los datos cargados en memoria por dicho programa dejan de ser utilizados en algún momento, por lo que ocupan un espacio que puede ser muy necesario para otros procesos.

En muchos lenguajes, la gestión de memoria es tarea del programador, el cual debe preocuparse de asignar y liberar memoria para el programa, escribiendo el código necesario.

En el caso de VB, tenemos la ventaja de que siempre ha sido la herramienta quien se ha encargado de la gestión de memoria, por lo que nunca ha sido necesario preocuparse de ella al escribir el código de los programas. En VB.NET tampoco es necesario ya que también se ocupa el entorno de ejecución de la memoria. ¿Qué necesidad hay pues de preocuparse ahora si nunca nos ha afectado?

Bien, en efecto, el programador de VB.NET no debe preocuparse de este aspecto, ya que es gestionado por el CLR; sin embargo, y ya que la gestión de memoria ahora es común a todos los lenguajes del entorno, conviene conocer, aunque sea de someramente como es manipulada y liberada la memoria durante la ejecución. Para ello, veamos las técnicas de recolección de memoria en versiones anteriores

de VB y en la actualidad, de forma que podamos apreciar mejor el trabajo realizado en .NET Framework.

Recolección de memoria en VB6 y versiones anteriores

En estas versiones, para liberar la memoria se utiliza un método denominado Finalización Determinista (Deterministic Finalization), que consiste en lo siguiente:

Cada vez que se instancia un objeto de una clase, se lleva un contador de instancias o contador de referencias; de esta forma se sabe si la memoria utilizada por un objeto es necesaria o no.

Cuando se destruye la última instancia del objeto, se ejecuta el evento Terminate( ) del mismo, y se libera la memoria que estaba utilizando; esto también permite saber al programador cuando se ha liberado la memoria de un objeto.

Recolección de memoria en .NET Framework

En lo que respecta a .NET Framework, el CLR implementa un supervisor de memoria denominado Garbage collector o recolector de basura, que se ocupa de hallar los objetos (datos administrados) de la aplicación que no son utilizados y liberar la memoria que usan, realizando de manera adicional, una compactación sobre la memoria, para optimizar el rendimiento sobre el área de trabajo de la aplicación.

En VB.NET no disponemos de un método Terminate( ) que indique cuando se ha finalizado el uso de un objeto, ya que el sistema de liberación de memoria es distinto:

Cuando un objeto ya no es necesario, debemos ejecutar su método Dispose( ), en el cual se deberá incluir el código que se ocupe de liberar los recursos que utilice el objeto; esta acción comunica al CLR que hay un objeto cuya memoria no es necesitada por la aplicación, aunque esto no quiere decir que dicha memoria se libere inmediatamente.

La liberación de memoria se producirá cuando el CLR así lo requiera; esto sucede cuando la zona de memoria reservada para las instancias del objeto, denominada *montón administrado*, se llene; en ese momento, el CLR activará el recolector de basura que se encargará de liberar y compactar la memoria no utilizada.

La ejecución de código dentro del CLR

El proceso de ejecución del código en el entorno de .NET Framework, varía notablemente respecto al modo de ejecución tradicional de programas. En este apartado, realizaremos un repaso de los elementos y técnicas que intervienen en dicho proceso, de modo que el lector tenga un conocimiento más detallado de lo que sucede con el código, desde que termina de escribirlo, y hasta el resultado obtenido tras su ejecución.

El IL, Intermediate Language

Durante el proceso de compilación, el código fuente es tomado por el compilador del lenguaje utilizado para su escritura, y convertido, no directamente a código binario, sino a un lenguaje intermedio, que recibe el nombre de Microsoft Intermediate Languaje (MSIL o IL).

Este lenguaje o código intermedio (IL a partir de ahora), generado por el compilador, consiste en un conjunto de instrucciones que son independientes del sistema operativo o procesador en el que vaya a ejecutarse el programa, y que se ocupan de la manipulación de objetos, accesos a memoria, manejo de excepciones, etc.

La Figura 10 muestra un diagrama con el proceso de generación de lenguaje intermedio a partir del código fuente.

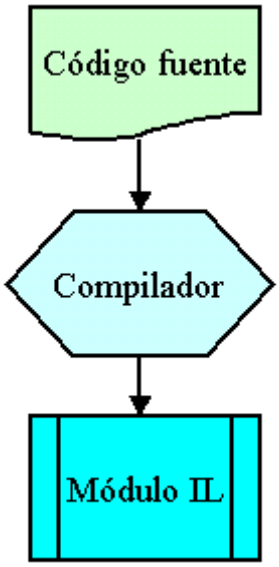


Figura 10. Obtención de lenguaje intermedio a partir de código fuente.

Además del código en IL, el compilador genera también metadatos, que como se ha explicado en un apartado anterior, contienen información adicional, incluida en la propia aplicación, y que serán utilizados por el CLR al ejecutar el programa.

Tanto el código en IL, como los metadatos generados, se guardan en un fichero de tipo EXE o DLL, basado en la especificación tradicional de Microsoft para ficheros con formato de ejecutable transportable (Portable Executable o PE) y objeto común (Common Object File Format o COFF). Con el desarrollo de la tecnología .NET, esta especificación ha sido ampliada para dar cabida, además de código binario, código IL y metadatos.

Ya que el código obtenido en IL es independiente del procesador, en su estado actual no es posible todavía ejecutarlo, debido a que el IL no ha sido diseñado para conocer las instrucciones específicas del procesador en el que se va a ejecutar. La ejecución se lleva a cabo, realizando el paso final de compilación que se detalla seguidamente.

Compilación instantánea del IL y ejecución

Como acabamos de indicar, el código en lenguaje intermedio no es directamente ejecutable, ya que desconoce la arquitectura del procesador sobre la cual va a funcionar.

Antes de realizar la ejecución, el código en IL debe ser convertido a código máquina, utilizando lo que se denomina un compilador instantáneo o compilador Just-In-Time (JIT compiler), que es el encargado de generar el código binario específico para el procesador en el que el programa será

ejecutado. La Figura 11 muestra un esquema con el proceso de compilación llevado a cabo por el compilador Just-In-Time (JIT a partir de ahora).

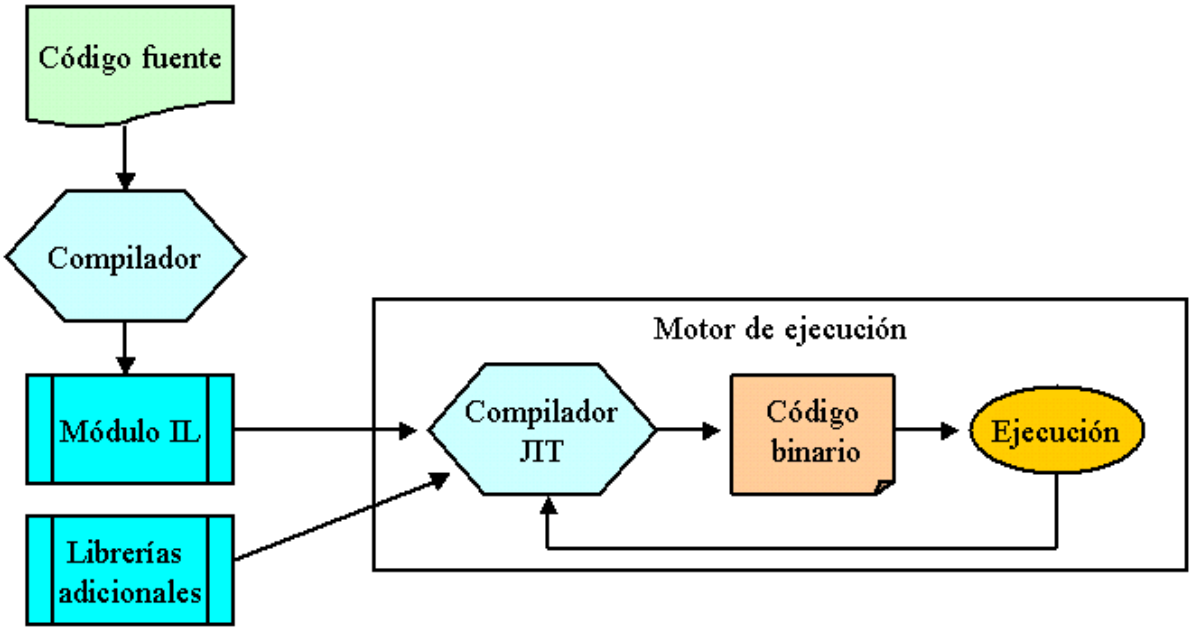


Figura 11. Proceso de compilación instantánea de código IL.

Compilación bajo demanda

Para optimizar la ejecución y mejorar su velocidad, el compilador JIT se basa en el hecho de que es posible que ciertas partes del código que compone la aplicación nunca sean ejecutadas. Por este motivo, al ejecutar la aplicación, no se toma todo su IL y se compila, sino que sólo se compila el código según se va necesitando y se almacena el código máquina resultante de modo que esté accesible en las siguientes llamadas. Veamos con un poco más de detalle este proceso.

Durante la carga de la aplicación, el cargador de código del CLR, toma cada tipo incluido en el programa, y para cada uno de los métodos que componen el tipo, crea y pega una etiqueta indicativa de su estado.

En la primera llamada a un método, se comprueba su estado de compilación a través de la etiqueta de estado; como aún no está compilado, se pasa el control al JIT, que compila el código IL a código máquina. A continuación se modifica la etiqueta de estado, de modo que en las próximas llamadas a ese método, la etiqueta de estado informa que el método ya ha sido compilado, por lo que se evita repetir el proceso de compilación, ejecutando directamente el código máquina creado con anterioridad. Esta técnica optimiza notablemente la velocidad de ejecución. Ver Figura 12.

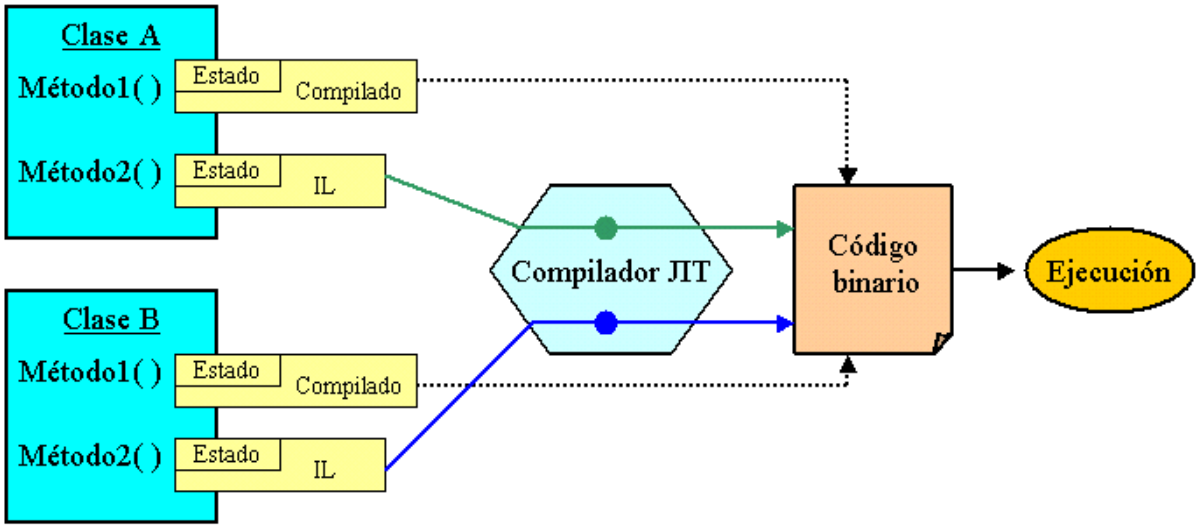


Figura 12. Compilación bajo demanda en .NET Framework.

Independencia de plataforma

Ya que el código máquina ejecutable, es obtenido a través de un compilador JIT, con las instrucciones adecuadas para un procesador determinado, .NET Framework proporciona varios compiladores JIT para cada una de las plataformas que soporta, consiguiendo así que la aplicación, una vez escrita, pueda funcionar en distintos sistemas operativos, y haciendo realidad el objetivo de que nuestro código sea independiente de la plataforma en la que se vaya a ejecutar, actuando .NET Framework como una capa intermedia, que aísla el código del sistema operativo. Ver Figura 13.

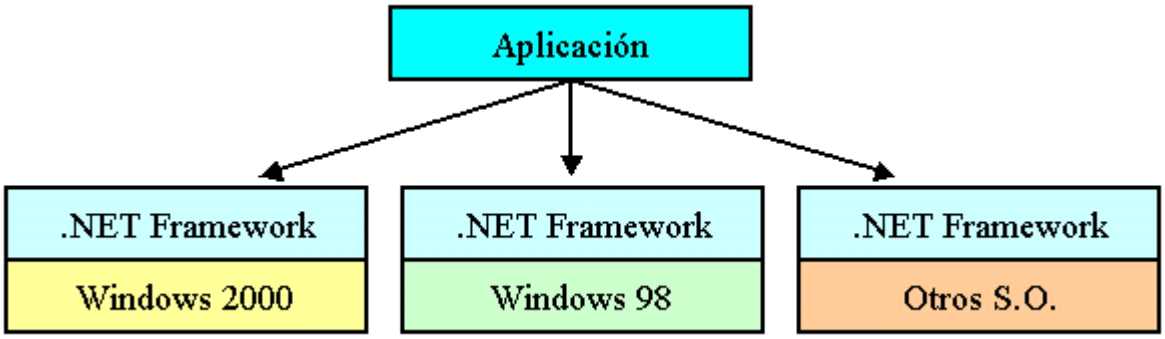


Figura 13. Una misma aplicación se ejecuta en distintos sistemas a través de .NET Framework.

Dominios de aplicación

En .NET Framework se han reforzado las características de seguridad y aislamiento hasta un nivel que permite la ejecución de múltiples aplicaciones en un mismo proceso. A este contexto de ejecución de un programa se le denomina *dominio de aplicación* (Application Domain).

La técnica utilizada tradicionalmente para conseguir aislar las aplicaciones, de modo que no se produzcan colisiones entre las mismas, ha sido a través de procesos. Cada aplicación se carga en un proceso separado, que proporciona el adecuado nivel de aislamiento; de este modo, se evitan posibles conflictos entre las direcciones de memoria utilizadas por cada programa. Sin embargo, esto supone un

gran consumo de recursos, cuando las aplicaciones deben hacer llamadas a otras aplicaciones que residan en procesos distintos, debido a que se debe de realizar un traspaso de procesos entre la aplicación que realiza la llamada y la aplicación destino. Esta técnica ha sido mejorada en .NET, de modo que se consigue tener en un mismo proceso, varias aplicaciones en ejecución.

El código administrado en .NET Framework, para poder ser considerado como seguro, debe pasar en primer lugar una fase de comprobación, efectuada por el CLR, que asegure el hecho de que no realice ningún acceso no permitido a direcciones de memoria u otras operaciones que puedan provocar un fallo del sistema. Una vez superada dicha comprobación, el código es marcado como seguro a nivel de tipos (type-safe), y la aplicación ejecutada.

Superada esta fase de verificación, el programa se ejecutará en un dominio de aplicación, que como hemos comentado antes, consiste en una técnica que permite ejecutar varias aplicaciones en un único proceso, con el mismo nivel de aislamiento que si se estuvieran ejecutando en procesos separados, y la ventaja de eliminar la sobrecarga producida cuando distintas aplicaciones están situadas en diferentes procesos y deben hacerse llamadas entre sí. Cada aplicación se ejecuta en su propio dominio de aplicación

Los dominios de aplicación incrementan notablemente la capacidad de crecimiento de los servidores al ejecutar múltiples aplicaciones en un mismo proceso. La Figura 14 muestra un esquema del proceso de carga y ejecución de aplicaciones en sus correspondientes dominios de aplicación.

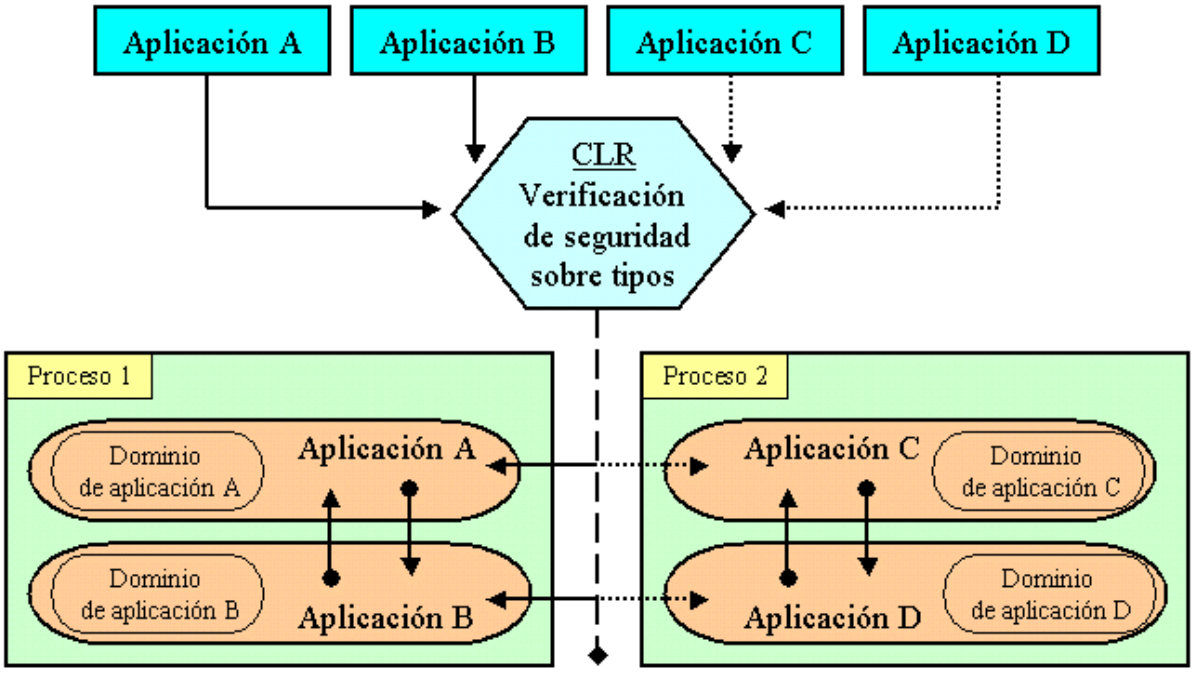


Figura 14. Carga de aplicaciones y creación de dominios de aplicación en procesos.

Servidores de entorno

Un servidor de entorno o Runtime Host es el encargado de ejecutar un dominio de aplicación dentro del CLR, aprovechando las ventajas proporcionadas por este último.

Cuando el CLR se dispone a ejecutar una aplicación, un servidor de entorno crea el entorno de ejecución o shell para dicha aplicación, y lo carga en un proceso; a continuación, crea un dominio de aplicación en ese proceso y por último carga la aplicación en el dominio.

.NET Framework dispone entre otros, de los servidores de entorno relacionados a continuación:

- **ASP.NET**. Carga el entorno en un proceso preparado para gestionarse en la web; creando también, un dominio de aplicación para cada aplicación de Internet ejecutada en un servidor web.

- **Internet Explorer**. Crea un dominio de aplicación por cada sitio web visitado, en el que se ejecutan controles administrados basados en el navegador.

- **Windows Shell**. Crea un dominio de aplicación con interfaz Windows, para cada programa que es ejecutado.

La Figura 15 muestra un diagrama del trabajo realizado por un servidor de entorno.

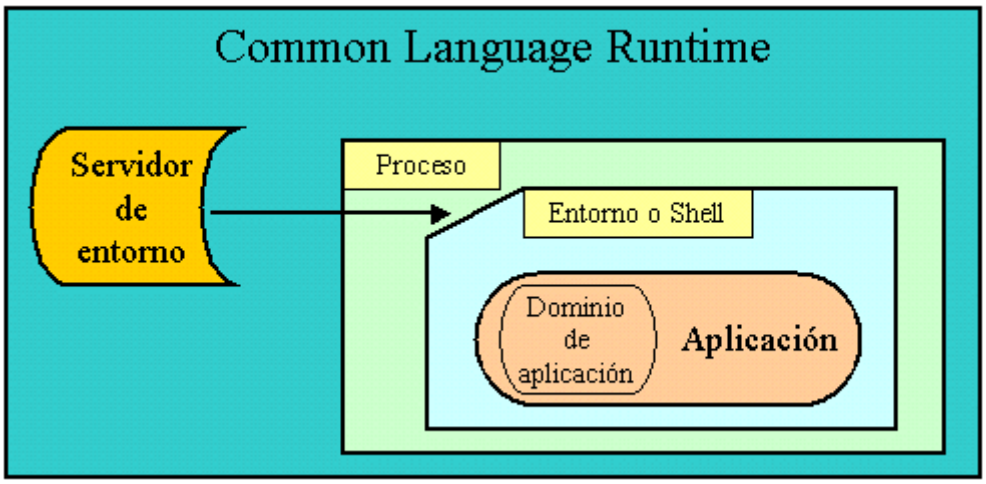


Figura 15. Servidor de entorno creando un entorno para un dominio de aplicación.

Namespaces

Otro de los pilares que forman los cimientos de .NET Framework es el concepto de *espacio de nombres* o *namespaces*.

Un namespace o espacio de nombres, también denominado nombre calificado, es el medio proporcionado por la plataforma para organizar las clases dentro del entorno, agrupándolas de un modo más lógico y jerárquico. Para comprender mejor este concepto veamos un ejemplo:

Estamos desarrollando un conjunto de clases para las operaciones de gestión contable y facturas de una empresa. Podemos ir escribiendo todas las clases y situarlas dentro de una misma aplicación o DLL. Actualmente tenemos dos clases para operaciones contables, denominadas Balance y LibroIVA, y otras dos clases para operaciones con facturas, denominadas Albaran y Factura.

Pero necesitamos añadir una clase más para las facturas que registre el libro de IVA de las facturas emitidas. El nombre más idóneo sería LibroIVA, pero ya está siendo utilizado, así que para evitar

problemas de duplicidad de nombres, debemos elegir otro que puede no se ajuste a definir la funcionalidad de la clase.

Mediante el uso de espacios de nombre este problema sería solucionado, con el añadido de poder organizar mejor cada clase, asignándole un nombre jerárquico para la funcionalidad que desempeña. Para ello, deberíamos crear un namespace con el nombre Gestion, que contuviera otros dos namespaces llamados Contabilidad y Facturación, para finalmente incluir en cada uno de ellos las clases correspondientes. La Figura 16 muestra un diagrama organizativo de las clases de este ejemplo utilizando espacios de nombre.

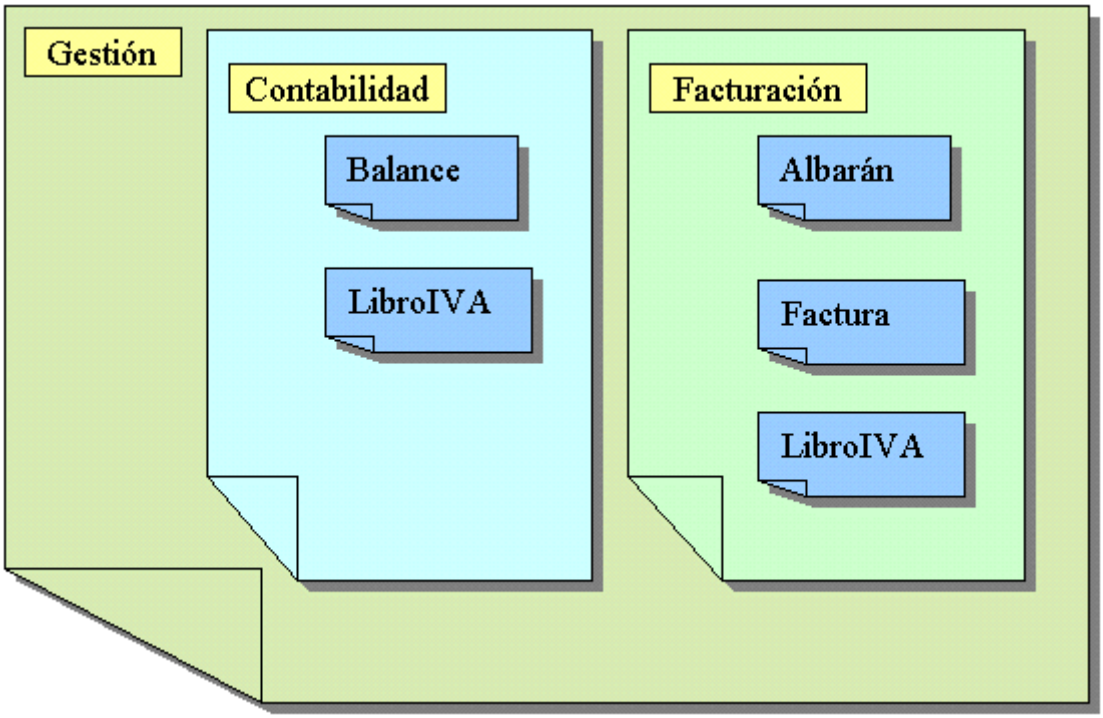


Figura 16. Estructura de un namespace con namespaces contenidos y clases dependientes.

Cuando creamos un proyecto dentro de Visual Studio .NET, esta herramienta ya se encarga de crear de forma automática un namespace con el mismo nombre del proyecto. En el caso de que sea el programador quien quiera crear un namespace de forma explícita, puede hacerlo mediante la palabra clave Namespace dentro del código del proyecto.

Para acceder desde el código de una aplicación, a una clase contenida dentro de un espacio de nombre, debemos indicarlo en la aplicación realizando una operación que en VB.NET se denomina *Importar*. Existen dos medios para importar un espacio de nombre: usar la palabra clave Imports en la cabecera del módulo de código junto al nombre del namespace y clase a la que queremos acceder; o bien usar la descripción calificada completa en cada momento que necesitemos hacer referencia a la clase. El Código fuente 5 muestra algunos ejemplos:

```
Imports Gestion.Contabilidad
Imports System.Windows.Forms

Public Class Cliente

    Public Shared Sub Main()
```

```
        Dim oBal As New Balance()
        Dim oFactu As New Gestion.Facturacion.Factura()
        Dim oPulsado As New Button()

        '............
        '............
        '............

    End Sub

End Class
```

Código fuente 5. Acceso a clases a través de espacios de nombre.

La convención sintáctica para hacer referencia a una clase contenida en un espacio de nombre, es como acabamos de ver, el espacio de nombre y la clase separados por un punto. En el caso de acceder a una clase que se encuentra con varios espacios de nombre de profundidad, especificaremos dichos espacios de nombre separados por un punto, e igualmente al final, la clase. La inclusión al final del nombre de la clase, depende de si instanciamos directamente el objeto usando la lista de espacios de nombre o importamos dicha lista.

En el caso de instanciar un objeto directamente en el código, escribiremos los espacios de nombre y al final, el nombre de la clase. Si importamos los espacios de nombre, no debemos poner el nombre de la clase, sino que debemos terminar con el espacio de nombres que contiene la clase que necesitamos.

De esta forma, la línea mostrada en el Código fuente 6, nos permitirá instanciar en el código del módulo donde esté declarada, objetos de la clase File, que está en el namespace IO, este último a su vez contenido en el namespace System.

```
Imports System.IO
```

Código fuente 6. Referencia a una clase a través de varios espacios de nombre.

En el ejemplo del Código fuente 5, al importar una clase contenida en un namespace, en este caso Balance o Button, cuando instanciamos un objeto de ella, no es necesario poner el namespace completo. No ocurre lo mismo con Factura, ya que al no haber importado el namespace que la contiene, debemos indicarlo en el momento de instanciar el objeto.

Todas las clases de la plataforma .NET están contenidas dentro de espacios de nombre, por lo que siempre que necesitemos instanciar un objeto, deberemos hacerlo usando la convención de espacios de nombre y puntos explicada anteriormente.

Las clases principales de .NET Framework están, por consiguiente, incluidas también en sus correspondientes namespaces. Como muestra el ejemplo anterior, si queremos instanciar un objeto para un formulario (Button, TextBox, etc.) debemos usar el espacio System.Windows.Forms, y dentro de este la clase que necesitemos. Como habrá podido adivinar el lector, el namespace System constituye el espacio raíz, a partir del cual, descienden el resto de espacios de nombre y clases de la plataforma, como IO, Threading, Collections, etc.

La jerarquía de clases de .NET Framework

El entorno de ejecución integra toda la funcionalidad y servicios necesarios a través de la jerarquía de clases base de la plataforma. La mayor parte de las necesidades básicas del programador están cubiertas por este amplio conjunto de clases, que permiten dotar a las aplicaciones de todas las características necesarias.

El desarrollador experimentado puede estar preguntándose la necesidad de implementar una nueva jerarquía de clases si las actuales ya cumplen con su cometido. Entre las posibles razones, queremos destacar las siguientes:

- El nuevo sistema de clases está mucho mejor organizado, y provee al programador de una potencia y versatilidad para sus aplicaciones nunca antes lograda en versiones anteriores de Visual Studio.

- Podemos crear una nueva clase, heredando de una clase propia de la plataforma, para extender su funcionalidad.

- Desplazando la funcionalidad de las clases fuera de los lenguajes, y haciéndolas por lo tanto, independientes de los mismos, simplifica el proceso de desarrollo.

- Al ser las clases de .NET Framework, comunes a todos los lenguajes, se eliminan las barreras tradicionales que impedían a los programadores abordar ciertos proyectos por el hecho de usar un lenguaje que no disponía de cierta funcionalidad que sí tenía otro lenguaje. Ahora cualquier programador, con independencia del lenguaje que elija, tiene pleno acceso a todas las funcionalidades que le brinda la plataforma .NET.

El ejemplo del Código fuente 7 muestra la declaración y asignación de valor a una variable desde VB.NET y C#. Con las salvedades particulares de cada lenguaje, en ambos casos se instancia una variable de la misma clase o tipo: Integer.

```
' código VB.NET
Dim MiDato As Integer = 20

// código C#
int MiDato=20;
```

Código fuente 7. Instanciación de objetos de la misma clase de .NET Framework desde distintos lenguajes.

Dentro de .NET Framework, System designa al espacio de nombre principal o raíz, a partir del cual, descienden todos los espacios de nombre y clases de la plataforma.

Además de las clases que proporcionan acceso a los tipos de datos intrínsecos de .NET Framework, System nos permite el acceso a otros servicios entre los que se encuentran los mostrados en la Tabla 2.

Descripción	Espacio de nombre
Servicios básicos	System.Collection System.IO System.Threading

Interfaz de usuario	System.Windows.Forms System.Drawing
Acceso a datos	System.Data System.XML
Manejo de ensamblados	System.Reflection

Tabla 2. Algunas de las clases básicas de .NET Framework.

Ensamblados

Un *ensamblado* o *assembly*, consiste en un conjunto de tipos y recursos, reunidos para formar la unidad más elemental de código que puede ejecutar el entorno de .NET Framework.

De igual forma que los edificios se crean a base de la unión de un conjunto de materiales, dentro de la tecnología .NET, los ensamblados se presentan como los *bloques de construcción* software, que se unen o ensamblan para crear aplicaciones. Una aplicación desarrollada para .NET Framework debe estar compuesta por uno o varios ensamblados, ver Figura 17.

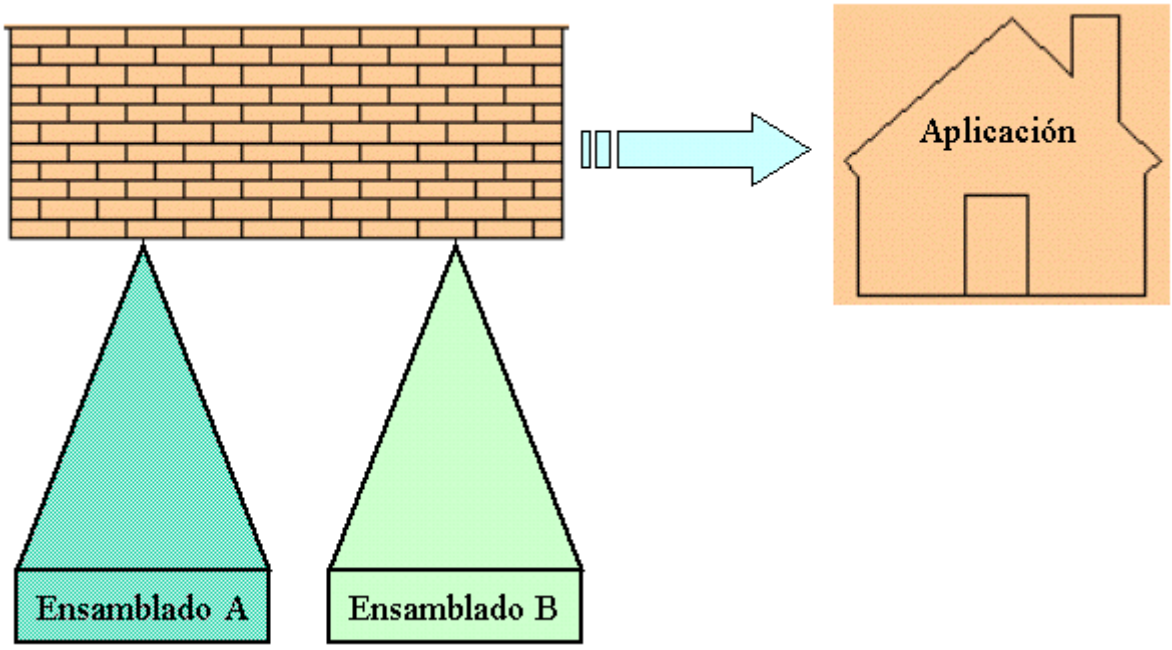


Figura 17. Los ensamblados forman bloques de construcción de aplicaciones.

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.

Sin embargo, un ensamblado extiende sus funcionalidades a un horizonte mucho más amplio, ya que puede contener otros elementos aparte de clases, como son recursos, imágenes, etc.

Por otro lado, simplifican los tradicionales problemas de instalación y control de versiones sobre los programas, uno de los objetivos de la tecnología .NET, en la que en teoría, para instalar una aplicación, sólo sería necesario copiar los ficheros que la componen en un directorio de la máquina que la vaya a ejecutar.

Cuando creamos un nuevo proyecto en VB.NET desde Visual Studio .NET, dicho proyecto es ya un ensamblado, creado de forma implícita.

La problemática tradicional de los componentes

De todos son conocidos los problemas que puede acarrear la instalación de una aplicación, en la que uno de sus elementos, sea un componente que sobrescribe otro ya existente de una versión anterior, pero que en su interior no guarda compatibilidad con ciertos aspectos de versiones anteriores, provocando el que otras aplicaciones que también hacen uso de ese componente, fallen.

Este inconveniente ha sido solucionado en parte por Windows 2000, ya que permite el desarrollo de programas, cuyos componentes puedan ser instalados en el mismo directorio del programa, de forma que al ejecutarse, la aplicación busque inicialmente en su directorio los componentes necesarios, y en caso de no encontrarlos, se dirija a las rutas habituales del sistema. Adicionalmente, los componentes propios del sistema operativo, permanecen bloqueados para evitar ser reemplazados accidentalmente por la instalación de terceras aplicaciones.

A pesar de que los anteriores aspectos constituyen un importante avance, no han resuelto del todo el problema, fundamentalmente, porque las cuestiones que tienen que ver con las versiones de componentes y sus correspondientes reglas, residen en lugares distintos del propio componente: librerías de tipos, el registro del sistema, etc.

Ensamblados, una respuesta a los actuales conflictos

Para solucionar las cuestiones planteadas en la anterior sección, se han desarrollado los ensamblados. Un programador debe indicar en el interior del ensamblado, mediante metadatos que se almacenan en el manifiesto del ensamblado, toda la información acerca de la versión a la que pertenece. El ensamblado, por su parte dispone de la maquinaria que supervisa el cumplimiento de las normas de versión; y gracias a su diseño, es posible ejecutar varias versiones del mismo ensamblado simultáneamente sin provocar errores en el sistema, esta es una de las grandes innovaciones que introducen.

Tipos de ensamblado según modo de creación

Para crear un ensamblado, podemos emplear Visual Studio .NET, la utilidad AL.EXE que proporciona el SDK de .NET Framework o las clases situadas en el espacio de nombre Reflection.Emit.

Cuando creamos desde VB.NET un nuevo proyecto, como configuración por defecto, dicho proyecto es al mismo tiempo un ensamblado privado. Trataremos más adelante sobre el ámbito de ensamblados.

Según la forma en que son creados, los ensamblados se dividen en dos tipos:

- **Estáticos**. Es el tipo más corriente de ensamblado, y es creado por el programador en tiempo de diseño.

- **Dinámicos**. Son ensamblados creados en tiempo de ejecución.

El contenido de un ensamblado

Un ensamblado está compuesto por los siguientes elementos:

- Manifiesto del ensamblado, que contiene información acerca de los elementos que forman el ensamblado.
- Metadatos sobre los tipos que contiene el ensamblado.
- Módulos de código con los tipos compilados en IL.
- Recursos adicionales.

Figura 18. Estructura de un ensamblado.

El manifiesto del ensamblado

Ya que uno de los imperativos de la tecnología .NET, radica en que todos los componentes que se ejecuten dentro de la plataforma sean auto descritos, esto es, que no necesiten de elementos exteriores al propio componente para obtener información acerca del mismo, la forma que tienen los ensamblados de proporcionar esta información, es a través de metadatos contenidos en su interior. Los metadatos de un ensamblado reciben el nombre de *manifiesto*.

Un manifiesto contiene la siguiente información:

- **Nombre**. Una cadena con el nombre del ensamblado.
- **Versión**. Número de versión.
- **Cultura**. Información sobre idioma y otros elementos culturales que soporta el ensamblado.
- **Nombre seguro**. En el caso de ensamblados compartidos, este nombre permite identificar al ensamblado a través de una clave.
- **Lista de ficheros**. Los nombres y un resumen de cada uno de los ficheros que forman el ensamblado.

- **Referencia de tipos**. Información que usa el entorno para localizar el código IL de cada uno de los tipos que contiene el ensamblado.

- **Ensamblados referenciados**. Lista de los ensamblados con los que el actual mantiene dependencias.

De los puntos que acabamos de describir, los cuatro primeros forman lo que se denomina *la identidad del ensamblado*.

El manifiesto se almacena, dependiendo del modo de creación del ensamblado, de las siguientes maneras:

- Si el ensamblado es de fichero único, la información del manifiesto se añade al fichero ejecutable .EXE o DLL (con formato PE) resultante.

- Si el ensamblado es de fichero múltiple, la información del manifiesto se graba en un fichero independiente (también con formato PE, pero que en este caso no es ejecutable), que sólo contiene los datos del manifiesto (aunque también se puede añadir a uno de los ficheros DLL o EXE que componen el ensamblado).

La siguiente sección, nos proporciona una información más detallada del lugar en el que es grabado el manifiesto dependiendo del tipo de ensamblado.

Tipos de ensamblado según contenido

Según el modo en que agrupemos sus elementos, obtendremos uno de los siguientes tipos de ensamblado:

- **Ensamblado de fichero único**. Está compuesto por un sólo fichero con formato PE (ejecutable transportable), bien .EXE o .DLL, en el que se incluyen todos los elementos necesarios. En este tipo de ensamblado, el manifiesto se encuentra integrado dentro del propio fichero. Ver Figura 19.

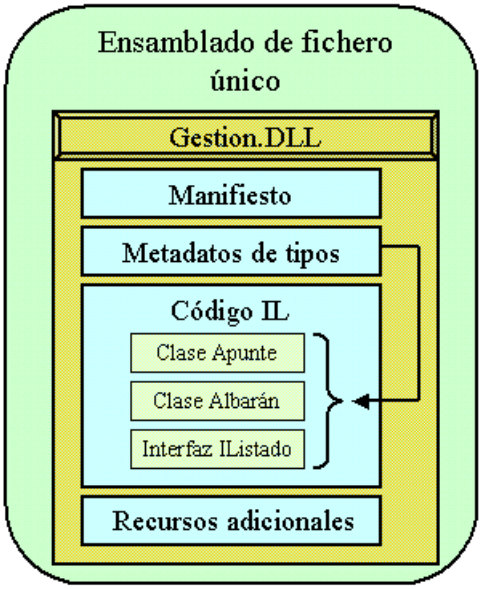


Figura 19. Estructura de un ensamblado de fichero único.

- **Ensamblado de múltiples ficheros**. Esta compuesto por un conjunto de ficheros. La ventaja de este tipo de ensamblado, reside en que podemos diseñarlo para optimizar la descarga de sus tipos, en función de la frecuencia con que sean utilizados.

Por ejemplo, podemos desarrollar una aplicación en la que existan varias clases, de las que algunas son utilidades generales, y otras son específicas de la aplicación; por último tenemos también un gráfico que usaremos en el interfaz de usuario.

En una situación como esta, podemos agrupar las clases específicas en una librería y las de utilidad en un módulo independiente, dejando el gráfico en su propio fichero, que sólo consumirá recursos cuando sea referenciado, optimizando de esta manera el rendimiento de la aplicación. El manifiesto en este caso, puede ser creado en un fichero aparte.

La Figura 20 muestra un esquema de ensamblado con múltiples ficheros.

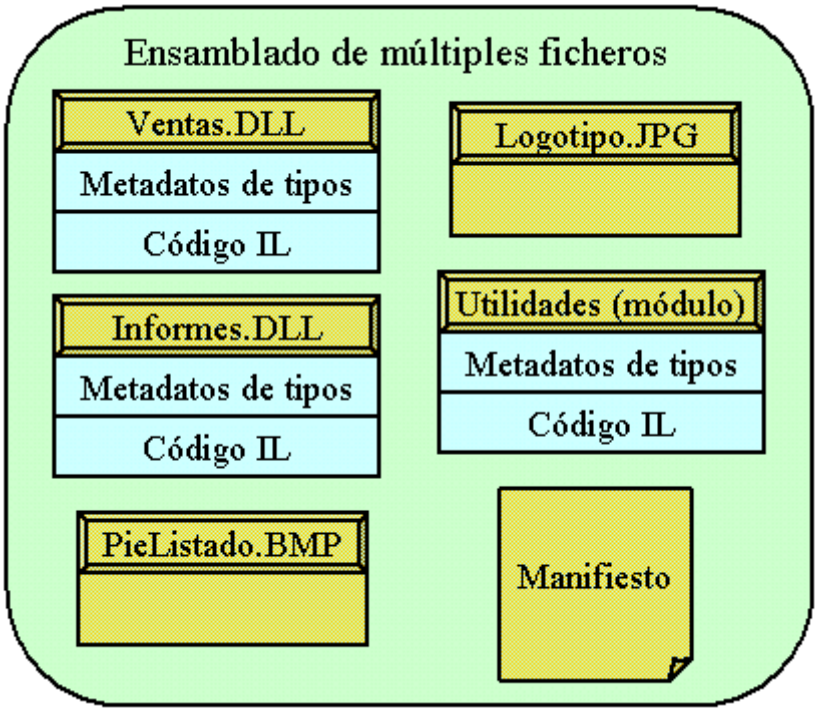


Figura 20. Estructura de un ensamblado de múltiples ficheros.

Un aspecto muy importante a tener en cuenta con referencia a este tipo de ensamblados, consiste en que los ficheros que lo componen, no están conectados físicamente (no se compilan a un fichero destino); es el manifiesto del ensamblado el que se encarga de mantener las referencias, de manera que el CLR al ejecutar el ensamblado, lee el manifiesto para averiguar que elementos lo forman, y así poder manipular el ensamblado como una entidad ejecutable única.

Este aspecto de la arquitectura de los ensamblados es de suma importancia, ya que un mismo fichero, conteniendo uno o varios módulos compilados en IL, puede formar parte al mismo tiempo de varios ensamblados, al no estar conectado físicamente con ninguno de ellos, sólo a través del manifiesto. Ver Figura 21.

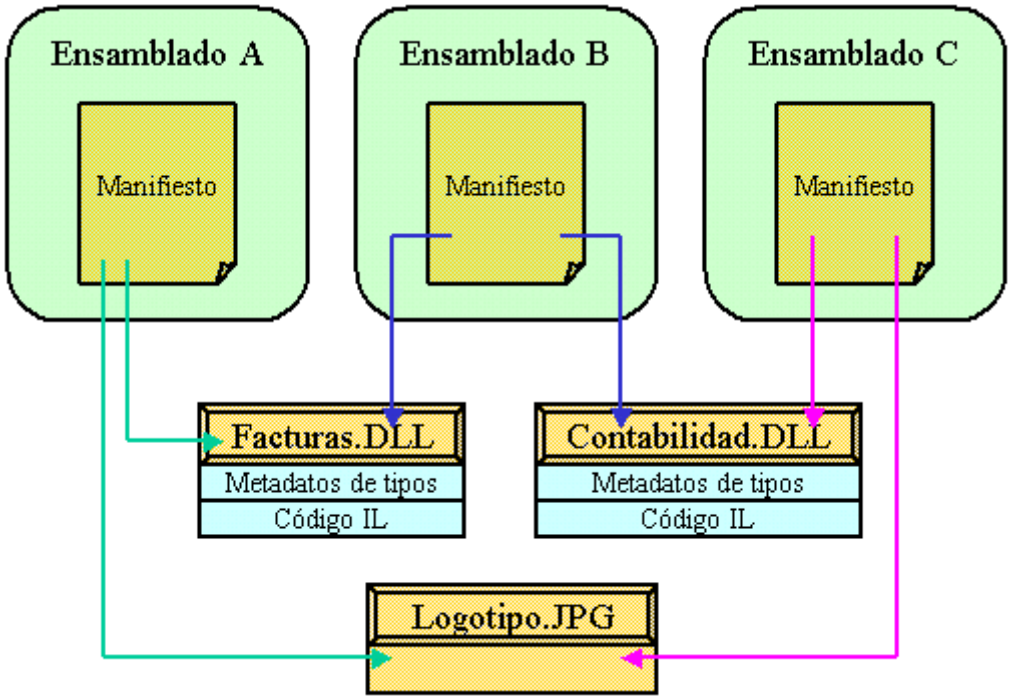


Figura 21. El manifiesto de cada ensamblado maneja las referencias a los ficheros que componen el ensamblado.

Tipos de ensamblado según ámbito

En función del lugar desde el que podamos acceder a un ensamblado, podemos clasificarlos en dos categorías:

- **Privados**. Un ensamblado privado es aquél que sólo es empleado por la aplicación, quedando situado en su mismo directorio. Como hemos indicado anteriormente, este es el modo por defecto en que se crean los ensamblados.

- **Compartidos**. Un ensamblado compartido, como su propio nombre indica, puede ser utilizado por varias aplicaciones. A diferencia de un ensamblado privado, que sólo es visible desde la propia aplicación, y se encuentra instalado en el directorio de esta, un ensamblado compartido debe exponer su funcionalidad al exterior; este motivo, hace que deban ser tenidos en cuenta factores adicionales, como la localización del ensamblado, seguridad en cuanto a accesos, versiones, etc., para que no entren en conflicto con el resto de ensamblados compartidos del sistema. Estos aspectos serán tratados a continuación

Ubicación de ensamblados compartidos

Los ensamblados compartidos se sitúan en la Caché Global de Ensamblados o Global Assembly Cache, que es una caché especial del CLR que administra este tipo de ensamblados.

Para instalar un ensamblado compartido en la caché global, utilizaremos alguna de las herramientas que permiten realizar tal operación, como Windows Installer, la utilidad GACUTIL.EXE proporcionada en el SDK de .NET Framework o el Explorador de Windows.

Durante la ejecución de un programa, cuando sea necesario el uso de un ensamblado compartido, el CLR realizará la búsqueda de dicho ensamblado empleando su clave de identificación y el número de versión, para localizar el correcto ensamblado en la caché global.

Identificación mediante claves integradas en el propio ensamblado

Para asegurarnos de que un ensamblado compartido no va a ocasionar conflictos de identificación con otros ensamblados residentes en la caché global, al crearlo, debemos proporcionarle una clave identificativa única, denominada en el argot de .NET *Nombre Seguro* o *Strong Name*.

Un nombre seguro está compuesto por la identidad del ensamblado (nombre, versión, etc), junto a una clave pública y firma digital; dicho de otro modo, una combinación de valores accesibles y encriptados, con los que nos aseguramos que nuestro ensamblado sea único a nivel global.

Podemos generar nombres para ensamblados compartidos empleando Visual Studio.NET, o alguna de las utilidades de .NET Framework, como AL.EXE y SN.EXE.

Los nombres seguros nos proporcionan los siguientes beneficios:

- **Unicidad**. A través de las dos claves únicas que componen el nombre del ensamblado, nos aseguramos de que dicho ensamblado también es único, es decir, ningún otro programador podrá generar un nombre igual.

- **Descendencia asegurada**. Gracias a los nombres seguros, nadie puede crear siguientes versiones de nuestro ensamblado.

- **Integridad a nivel de contenido**. Las comprobaciones de seguridad que realiza el entorno de ejecución, nos aseguran que el ensamblado no ha sido modificado desde que fue generado. A pesar de todo, el propio nombre seguro no constituye en sí mismo, un elemento de confianza sobre la integridad del ensamblado; dicha confianza es alcanzada a través de la firma digital con el certificado.

Para alcanzar un nivel de confianza en un ensamblado, además del nombre seguro, es necesario utilizar una herramienta como SIGNCODE.EXE, que proporciona una firma digital para el ensamblado. Esta utilidad requiere una autoridad de emisión de certificados, que podemos integrar en el ensamblado para cumplir los requerimientos de confianza necesarios en ciertos niveles.

Versiones de ensamblados

Todos los ensamblados deben disponer de su correspondiente versión, que es almacenada en el manifiesto. Los datos de la versión de un ensamblado se indican de dos maneras:

- **Número de versión**. Consiste en un valor numérico representado bajo el siguiente formato:

  <Número superior>.<Número inferior>.<Número de construcción>.<Revisión>

  Un ejemplo de número de versión podría ser el siguiente: 5.2.176.0

  El número de versión se graba dentro del manifiesto, junto a otros datos, como el nombre del ensamblado, clave pública, referencias a otros ensamblados, etc., que son utilizados por el entorno para cargar la correcta versión del ensamblado cuando se ejecuta.

- **Descripción de versión**. Cadena de caracteres con información adicional sobre el ensamblado, que no es utilizada por el entorno, sólo se proporciona a efectos informativos.

Compatibilidad a nivel de versión

Cada una de las partes que componen el número de versión, tiene un valor que define el grado de compatibilidad del ensamblado a la hora de su ejecución, por lo que la versión de un ensamblado dispone de los niveles de compatibilidad relacionados a continuación:

- **Incompatible**. Este nivel viene determinado por los números superior e inferior de la versión, lo que quiere decir, que al realizar un cambio en cualquiera de esos números, el ensamblado se hace incompatible con otras versiones del mismo ensamblado, que difieran en estos números.

- **Posiblemente compatible**. Este nivel viene determinado por el número de construcción, y significa que una nueva versión del mismo ensamblado, en la que haya un cambio sólo en este número, permite el uso de dicha versión, aunque esto no implica que puedan aparecer ciertas incompatibilidades.

- **Actualización rápida**. Este tipo de compatibilidad, también denominado QFE (Quick Fix Engineering) se indica mediante el número de revisión, e indica al entorno que se trata de una actualización de urgencia para resolver problemas puntuales importantes.

Ejecución conjunta de ensamblados

La *ejecución conjunta* o *Side-by-Side Execution* de ensamblados es una de sus características mas potentes, y consiste en la capacidad del entorno de ejecución de poder tener simultáneamente, varias versiones del mismo ensamblado en ejecución en el mismo equipo y en el mismo proceso si fuera necesario.

Un aspecto de este tipo de ejecución, consiste en que las distintas versiones de un mismo ensamblado no deben mantener una rigurosa compatibilidad.

Supongamos que disponemos de un ensamblado compartido que es utilizado por diferentes aplicaciones y creamos una nueva versión del ensamblado con algunos cambios que necesita una nueva aplicación. En este escenario, ejecutando de forma conjunta las dos versiones del ensamblado, las aplicaciones originales seguirían haciendo llamadas a la versión mas antigua del ensamblado, mientras que la nueva aplicación llamaría a la versión más reciente del ensamblado. Ver Figura 22.

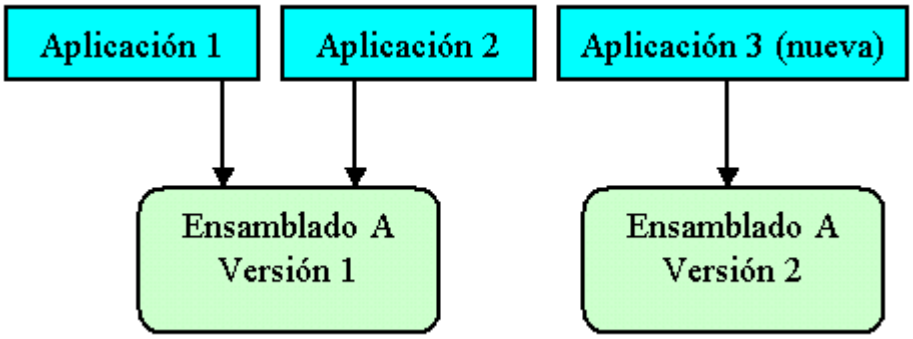


Figura 22. Ejecución conjunta de ensamblados.

Ficheros de configuración

Cuando el CLR necesita hacer uso de un ensamblado, toma su número de versión del manifiesto, realiza una búsqueda del ensamblado y lo ejecuta en caso de encontrarlo. Este es el comportamiento por defecto de entorno de ejecución.

Sin embargo, puede haber ocasiones en las que se haga necesario el uso de una versión diferente del ensamblado, para lo cual, debemos *redirigir* al CLR hacia dicha versión específica que deseamos ejecutar, en lugar de la versión por defecto.

Esto lo conseguimos a través de los ficheros de configuración, que son unos ficheros con extensión .CFG, basados en etiquetas XML, en los que a través de un conjunto de etiquetas clave ordenamos al entorno la ejecución de una determinada versión del ensamblado. En el ejemplo del Código fuente 8 indicamos mediante un fichero de configuración el uso de una versión específica de un ensamblado.

```
<configuration>
   <runtime>
      <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
         <dependentAssembly>
            <assemblyIdentity name="GestVentas"
                              publickeytoken="32ab4ba45e0a69a1"
                              culture="sp" />
            <bindingRedirect oldVersion="1.0.0.0"
                             newVersion="2.0.0.0"/>
            <codeBase version="2.0.0.0"
                      href="http://www.AcmeFac.com/GestVentas.dll"/>
         </dependentAssembly>
      </assemblyBinding>
   </runtime>
</configuration>
```

Código fuente 8. Contenido de un fichero de configuración de ensamblado.

Localización de ensamblados por parte del CLR

Una vez que un ensamblado compartido ha sido debidamente creado con su correspondiente número de versión, claves, e instalado en la caché global de ensamblados, el entorno de ejecución, para utilizar dicho ensamblado, debe realizar una serie de operaciones que le permitan localizar con éxito el ensamblado, de modo que sea la versión exacta que necesita ejecutar. A continuación, se realiza una breve descripción de cada uno de esos pasos que lleva a cabo el entorno

- **Verificación de versión**. En primer lugar, el CLR comprueba si la versión del ensamblado es correcta, examinando la existencia de posibles ficheros de configuración.

  Los ficheros de configuración permiten el establecimiento o cambio de valores de configuración del ensamblado en tres niveles: aplicación, políticas de autor, máquina. Los posibles ficheros de configuración son buscados también en este orden.

- **Ensamblados previamente utilizados**. A continuación, se verifica si el ensamblado ya ha sido cargado por una llamada anterior. En caso afirmativo, se utiliza dicho ensamblado, evitando tener que cargarlo de nuevo.

- **Comprobación de la caché global de ensamblados**. Si no se emplea un ensamblado ya cargado, se busca seguidamente en la caché global.

- **Localización mediante codebases o sondeo**. Una vez averiguada la versión del ensamblado, se intenta localizar el ensamblado a través de sus ficheros de configuración, en primer lugar mediante un codebase, que consiste en una de las etiquetas que pueden incluirse en este tipo de ficheros y que proporcionan información sobre la versión.

  Si no existe un codebase, se aplica una técnica denominada sondeo (probing), que consiste en realizar una búsqueda por aproximación utilizando los siguientes elementos:

  - Ruta en la que se está ejecutando la aplicación.
  - Datos culturales del ensamblado.
  - Nombre del ensamblado.
  - Lista de subdirectorios situados en la ruta de ejecución de la aplicación.

Optimización de la carga de ensamblados

Cuando el CLR carga un ensamblado compartido para su ejecución, su código es compilado por el correspondiente compilador JIT, para cada dominio de aplicación que necesite dicho ensamblado. Esto puede suponer un gran consumo de recursos.

Para solucionar este problema, se puede modificar el modo de carga del ensamblado, estableciendo que pueda ser usado por múltiples aplicaciones, de manera que optimice su rendimiento en este sentido.

Cuando se establece un ensamblado como utilizable por múltiples aplicaciones, el compilador JIT sólo compila una vez el código del ensamblado, generando una única copia del mismo.

Seguidamente, cuando una aplicación solicita uno de los tipos (clases) del ensamblado, se crea una referencia de dicho tipo (dicho tipo es mapeado) para todos los dominios de la aplicación, en lugar de cargar una copia del tipo en cada dominio; con esto se consigue consumir menos recursos, teniendo en cuenta que cada dominio de aplicación comparte el código, pero tiene su propia copia estática de los datos. Ver Figura 23.

La desventaja en este tipo de ejecución de ensamblados radica en que el código resultante del compilador JIT es mayor que para un ensamblado no optimizado para carga, y el acceso a los datos resulta más lento ya que debe realizarse de modo indirecto.

Existen tres niveles de configuración para la carga de ensamblados, que se relacionan a continuación:

- **Dominio único**. El ensamblado no está optimizado para ser usado por múltiples aplicaciones.
- **Dominio múltiple**. El ensamblado está optimizado para ser usado por múltiples dominios de aplicación, en los que cada aplicación de cada dominio ejecuta el mismo código.
- **Dominio múltiple de entorno**. El ensamblado se optimiza para ser usado por múltiples dominios de aplicación, en los que cada dominio ejecuta diferente código.

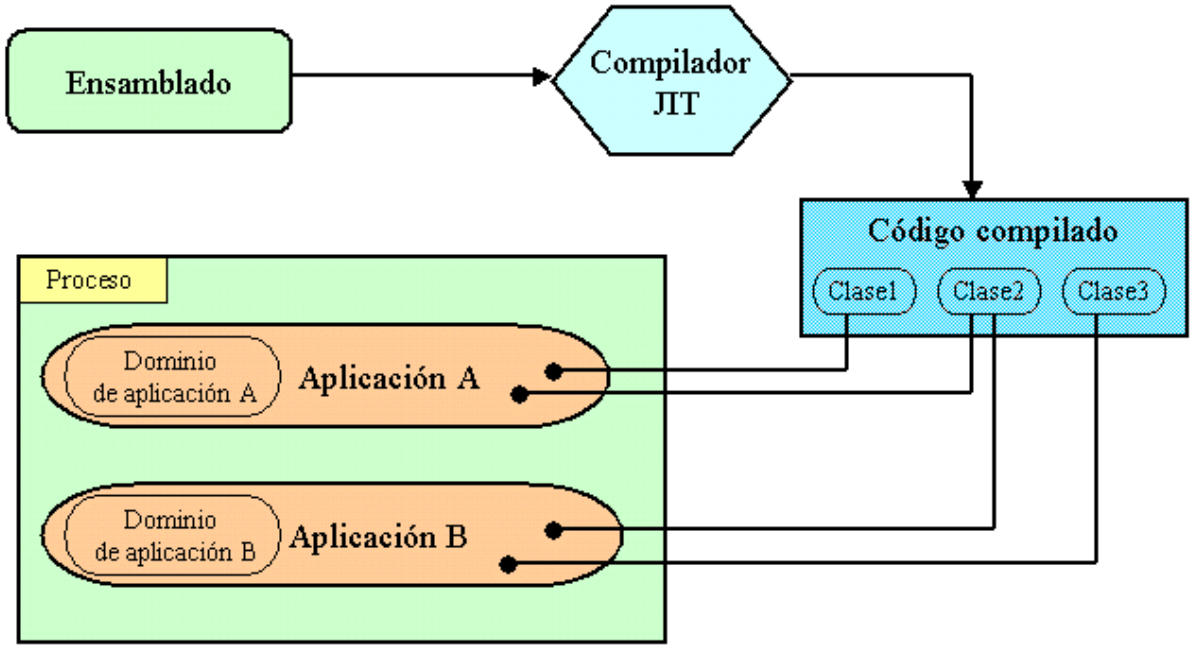


Figura 23. Ensamblado utilizado para múltiples aplicaciones.

4

Instalación de Visual Studio .NET

Preparación del entorno de trabajo

Antes de poder comenzar a escribir aplicaciones para .NET Framework, debemos instalar en nuestra máquina de trabajo las herramientas que nos permitirán el desarrollo de programas para este entorno de ejecución.

.NET Framework SDK

Se trata del kit de desarrollo de software para .NET Framework (Software Development Kit o SDK), que contiene la propia plataforma .NET y un conjunto de herramientas independientes, algunas 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

Es la nueva versión de la familia de herramientas de desarrollo de software de Microsoft, naturalmente orientadas hacia su nuevo entorno de programación: .NET Framework.

Si bien es posible la escritura de programas empleando sólo el SDK de .NET Framework, este último, al estar compuesto de herramientas independientes, constituye un medio más incómodo de trabajo.

Visual Studio .NET (VS.NET a partir de ahora), al tratarse de un entorno de desarrollo integrado (Integrated Development Environment o IDE como también lo denominaremos a lo largo del texto), aúna todas las herramientas del SDK: compiladores, editores, ayuda, etc., facilitando en gran medida la creación de programas. Por este motivo, todas las explicaciones y ejemplos desarrollados a lo largo de este texto se harán basándose en este entorno de programación.

Requisitos hardware

La Tabla 3 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	3 GB	

Tabla 3. Requerimientos hardware para instalar Visual Studio .NET.

Sistema operativo

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

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

Para aprovechar todo el potencial de desarrollo de la plataforma, es recomendable usar como sistema operativo Windows 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.

Recomendaciones previas

Es recomendable realizar la instalación sobre un equipo *limpio*, es decir, un equipo con el software mínimo para poder realizar pruebas con .NET Framework, o con otro tipo de aplicaciones sobre las que estemos seguros de que no se van a producir conflictos con el entorno.

En este sentido, una buena práctica consiste en crear en nuestro disco duro una partición que utilizaremos para el trabajo cotidiano con el ordenador, y otra partición en la que instalaremos VS.NET.

Para ayudar al lector a formarse una idea más aproximada en cuanto a configuraciones hardware y software, el equipo utilizado para realizar las pruebas mostradas en este texto ha sido un Pentium III a 933 MHz, con 256 MB de memoria y disco duro de 18 GB.

En cuanto a sistemas operativos, se han realizado dos particiones sobre el disco duro; en la partición primaria se ha asignado un tamaño de 2 GB y se instalado Windows 98. En el resto de espacio en disco se ha creado una unidad lógica sobre la que se ha instalado Windows 2000 Server y el Service Pack 2 para este sistema operativo.

Respecto a las aplicaciones utilizadas, aparte naturalmente de VS.NET, hemos instalado Visual Studio 6.0 que puede perfectamente convivir en el mismo equipo en el que esté instalado .NET Framework y por ende VB.NET, de esta forma podemos hacer pruebas con la herramienta de migración de aplicaciones de VB.NET que convierte aplicaciones escritas en VB6 a la nueva versión de VB. Como base de datos se ha utilizado SQL Server 2000 y como conjunto de herramientas adicionales Office 2000.

El orden más conveniente de instalación en el equipo del software antes mencionado, de forma que evitemos posibles conflictos ha sido el siguiente:

- Windows 2000 Server.
- Service Pack 2 para Windows 2000.
- Office 2000.
- Visual Studio 6.0.
- SQL Server 2000.
- Visual Studio .NET.

Y ya sin mas preámbulos, pasemos al proceso de instalación de .NET.

Instalación de Visual Studio .NET

En el momento de escribir este texto, se ha empleado Visual Studio .NET, Beta 2, versión española (número de versión 7.0.9254), que se compone de los tres CDs de instalación del producto más uno de actualización de componentes del sistema operativo (Windows Component Update)

Procederemos insertando el disco de instalación rotulado como CD1, el cuál detectará si es necesario actualizar algún componente a nivel del sistema operativo; en caso afirmativo, pulsaremos sobre el paso 1 *Windows Component Update*, en el que se nos pedirá el disco rotulado con el mismo nombre. Ver Figura 24.

Una vez insertado el disco de actualización de componentes para Windows, se mostrará la pantalla de la Figura 25. En caso de aceptar el contrato, haremos clic sobre *Continuar*, para que el instalador detecte qué componentes faltan por actualizar.

Figura 24. Selección de actualización de componentes de Windows.

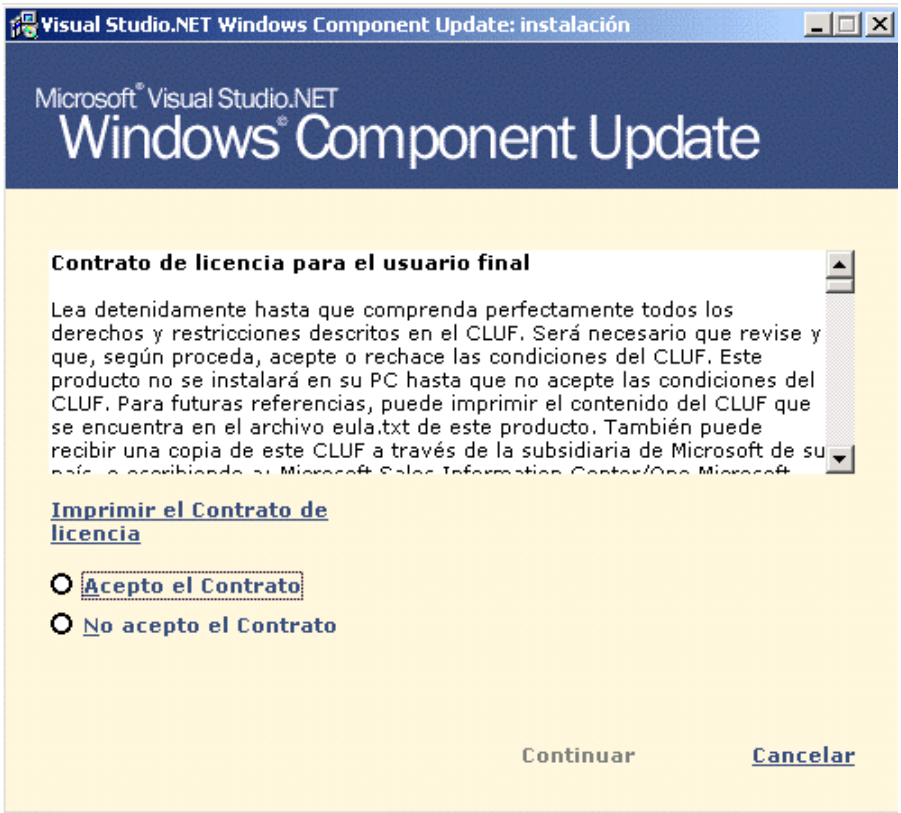


Figura 25. Contrato de instalación para Windows Component Update.

Una vez detectados los componentes que necesitan actualización, serán mostrados a continuación en la lista de la Figura 26, donde volveremos a pulsar sobre *Continuar*.

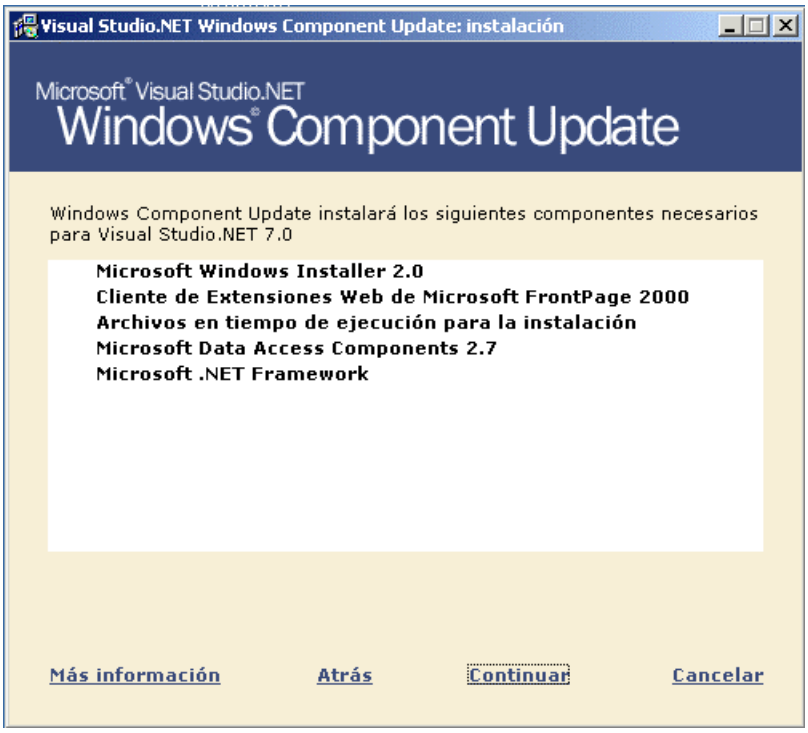


Figura 26. Lista de componentes que se necesita actualizar.

Ya que es posible que el programa de instalación reinicie el equipo una o más veces, a continuación estableceremos, en el caso de que existan en nuestro equipo, las claves de acceso al sistema, para que los reinicios sean automáticos. Ver Figura 27.

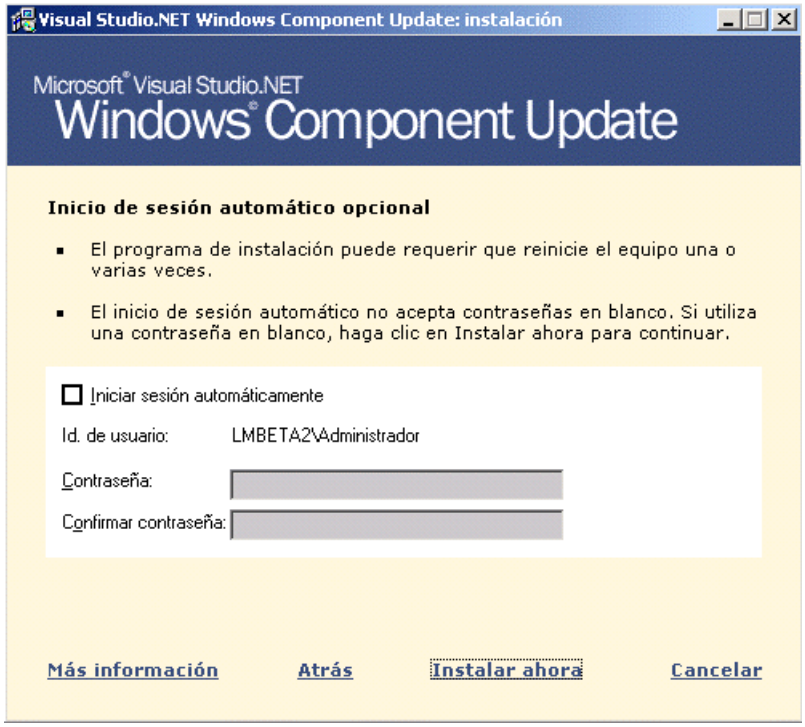


Figura 27. Valores para realizar reinicios automáticos del equipo.

Pulsaremos a continuación sobre *Instalar ahora*, con lo que se procederá a la actualización de los componentes de la lista. Una vez terminada esta actualización, aceptaremos la ventana final de Windows Component Update y seguiremos con la instalación normal de VS.NET, lo que nos requerirá de nuevo la introducción del CD1.

Puesto que ya hemos actualizado los componentes del sistema, el siguiente paso será ya la instalación de VS.NET, que pondremos en marcha al hacer clic sobre el paso 2 de la instalación, que tiene el nombre de *Visual Studio .NET*. Ver Figura 28.

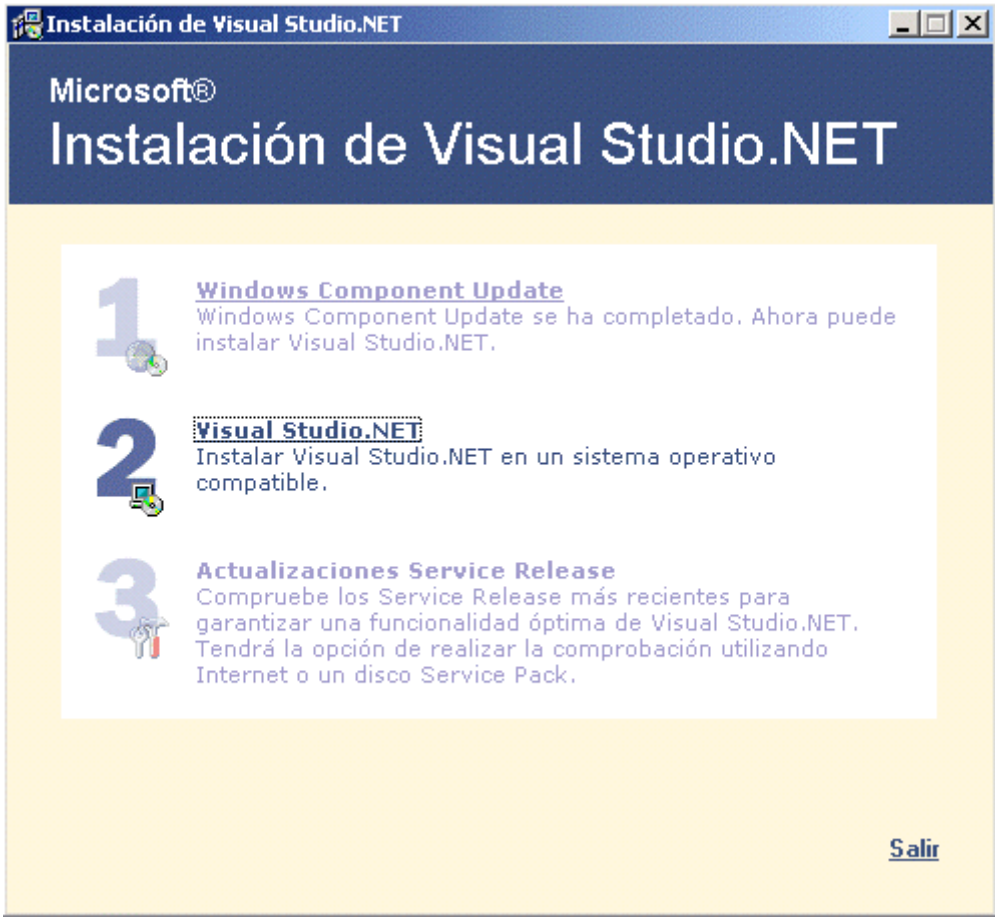


Figura 28. Instalación de Visual Studio .NET.

Se mostrará pues, la pantalla con los datos de licencia, producto y usuario. En el caso de estar de acuerdo con todos estos términos y aceptar el contrato, haremos clic sobre *Continuar*. Ver Figura 29.

A continuación debemos seleccionar aquellos elementos del producto que deseamos instalar, el entorno de ejecución, lenguajes, utilidades, ayuda, etc., y su ubicación en el disco duro, como muestra la Figura 30. Terminada la selección, pulsaremos sobre *Instalar ahora* para que comience el proceso.

Durante la instalación, el programa nos solicitará progresivamente los discos rotulados como CD2 y CD3.

Este proceso de instalación nos indica el archivo que se está instalando en cada momento, así como la información de su estado a través de una barra de progreso y el tiempo estimado restante, aunque por las pruebas realizadas, este último valor no es totalmente fiable. Para que el lector se forme una idea, en el equipo en el que se realizó la instalación, esta llevo un tiempo aproximado de dos horas. Ver Figura 31.

Figura 29. Información de licencia de Visual Studio .NET.

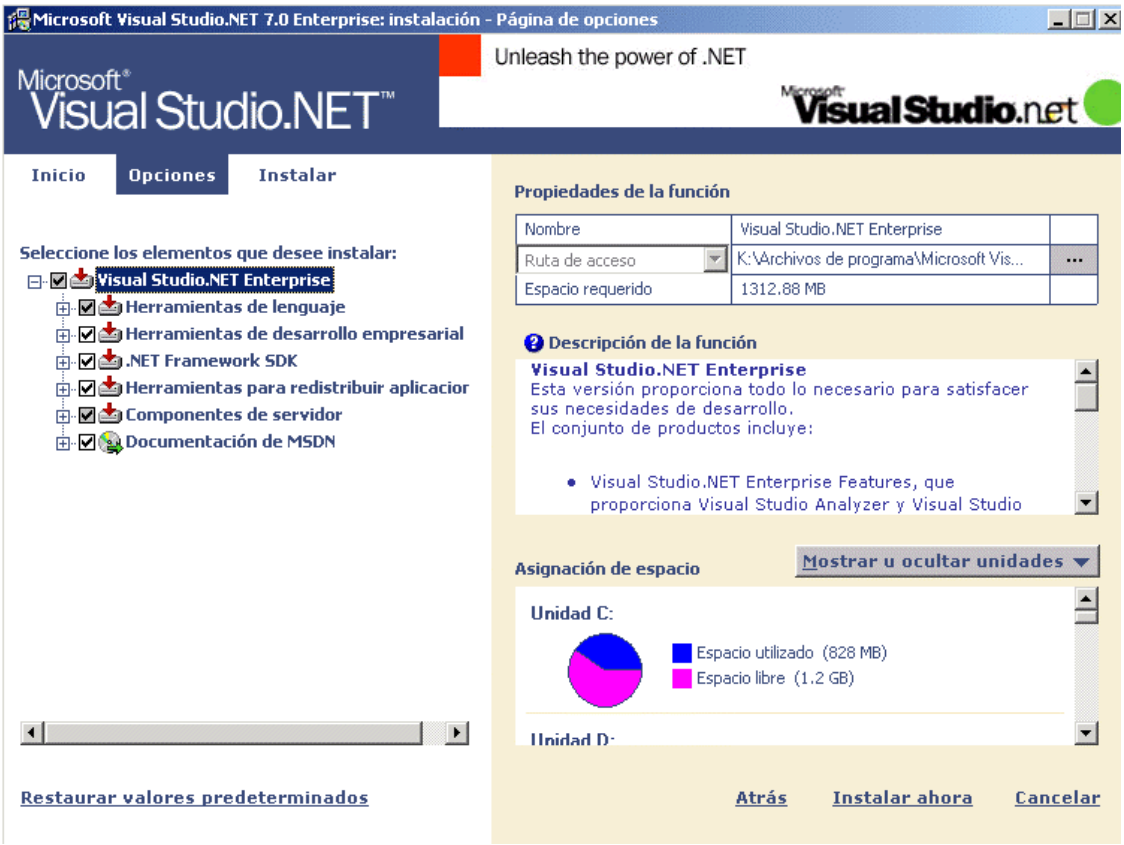


Figura 30. Selección de componentes a instalar de Visual Studio .NET.

Figura 31. Información sobre el progreso de la instalación.

Concluida la instalación, el programa nos informará de si se produjo alguna incidencia. En caso de que no se hayan producido errores, finalizaremos haciendo clic sobre *Listo*, con lo que ya tendremos instalado Visual Studio .NET en nuestro ordenador. Ver Figura 32.

Figura 32. Final de la instalación.

Bases de datos de ejemplo

El programa de instalación también copia varias bases de datos SQL Server de ejemplo en nuestro equipo que pueden ser utilizadas desde SQL Server 7 o posteriores. En el caso de disponer de SQL Server 2000, probablemente habrá creado la siguiente ruta: \Archivos de programa\Microsoft SQL Server\MSSQL$NetSDK\Data, y en ella habrá depositado las bases de datos típicas de ejemplo: Northwind y pubs, más dos adicionales con el nombre de GrocerToGo y Portal.

En el caso de que estas bases de datos no se hayan incorporado al servidor SQL, las adjuntaremos manualmente realizando los siguientes pasos:

- Iniciar el Administrador corporativo de SQL Server (Enterprise Manager) y conectar con nuestro servidor de trabajo. Ver Figura 33.

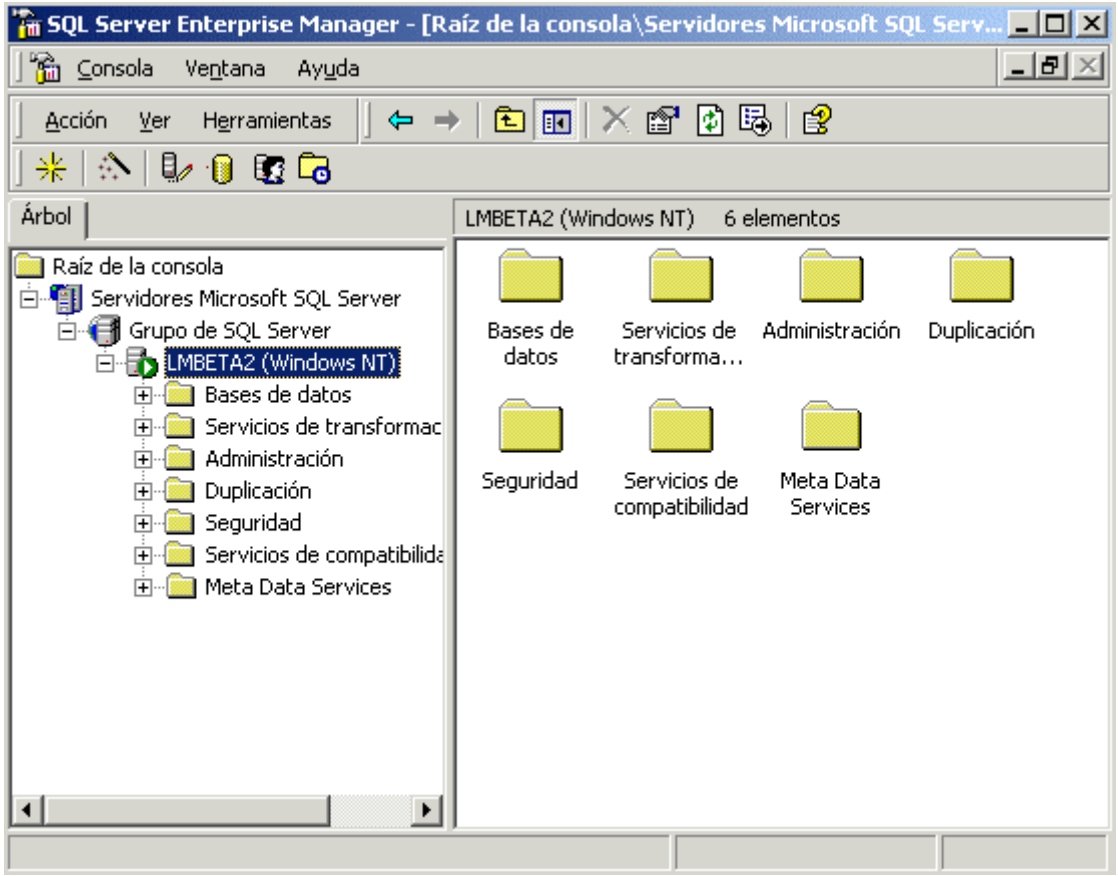


Figura 33. Administrador corporativo de SQL Server.

- A continuación haremos clic derecho sobre el elemento *Bases de datos* e iremos abriendo los sucesivos menús contextuales hasta seleccionar la opción *Adjuntar base de datos* que se muestra en la Figura 34.

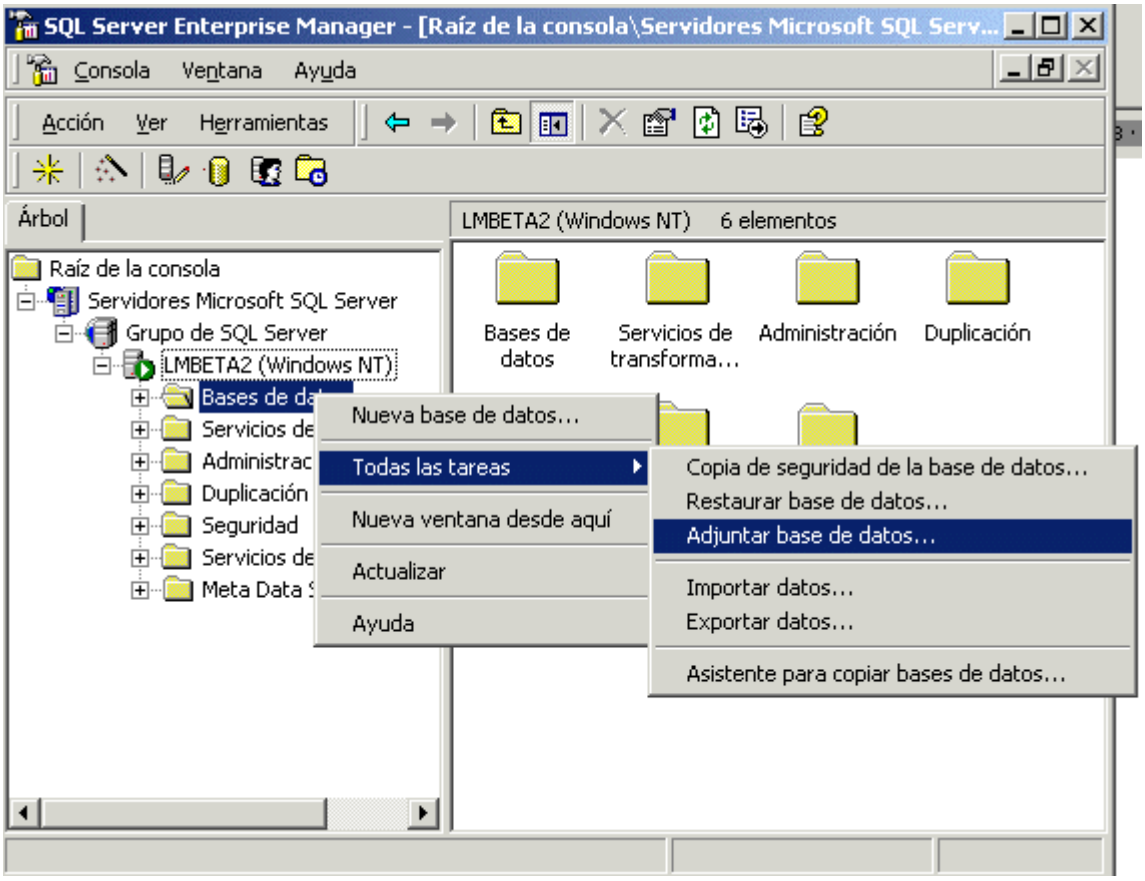


Figura 34. Seleccionar opción para adjuntar una base de datos a SQL Server.

- Se mostrará un cuadro de diálogo para seleccionar la ruta en donde reside la base de datos que queremos adjuntar. Por lo que haciendo clic en el botón con los puntos suspensivos, nos desplazaremos a la ruta en la que se han situado las nuevas bases de datos de ejemplo y seleccionaremos el fichero de datos de una de ellas, por ejemplo: Portal.MDF, como muestra la Figura 35.

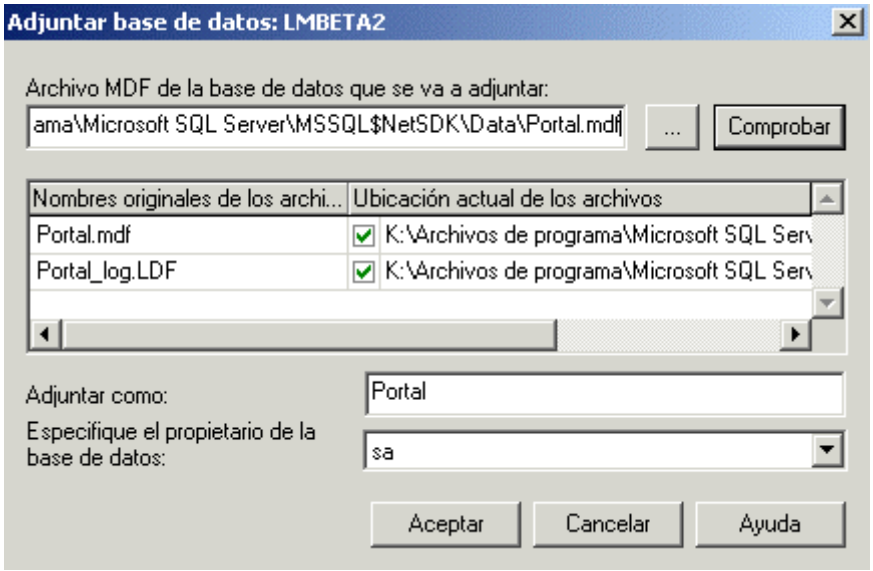


Figura 35. Selección de base de datos para adjuntar a SQL Server.

- Pulsaremos *Aceptar* y si todo es correcto, se adjuntará la base de datos a nuestro servidor mostrándose ya en el Administrador corporativo. Ver Figura 36.

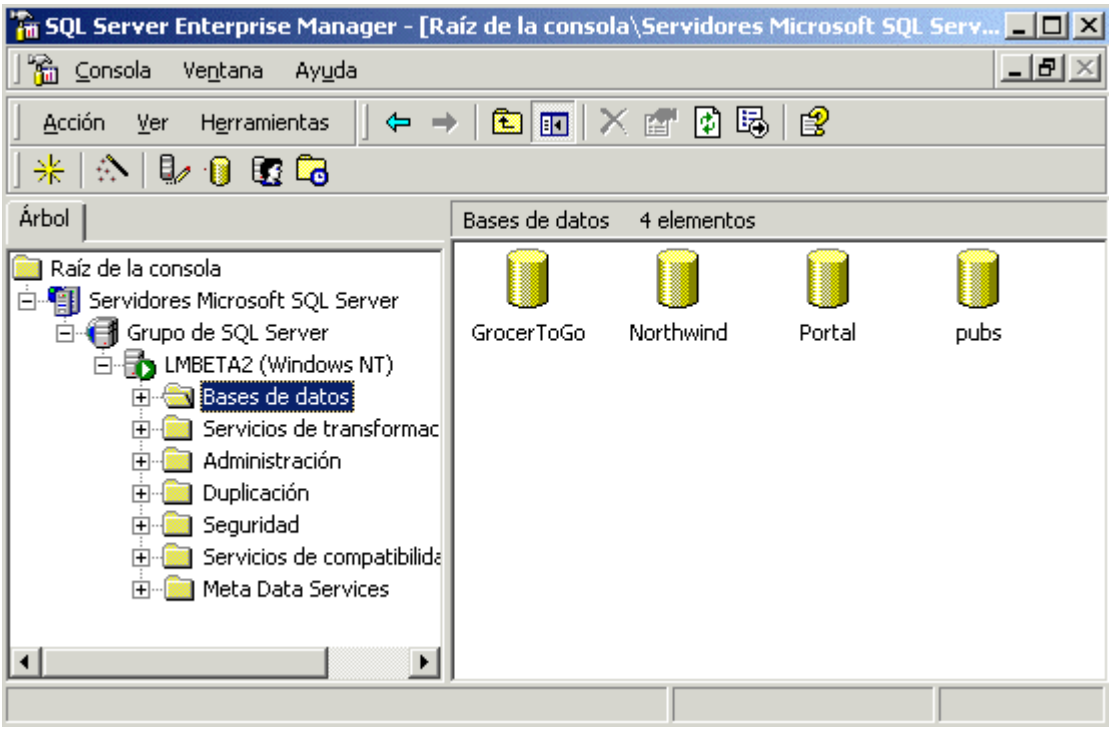


Figura 36. Bases de datos adjuntadas a SQL Server.

Concluidos todos los pasos de instalación, estamos preparados para comenzar a trabajar con nuestro nuevo entorno de trabajo, y para ello, nada mejor que escribir un pequeño programa de prueba, que mostraremos en el siguiente tema.

5

La primera aplicación

Un Hola Mundo desde VB.NET

Una de las primeras acciones que solemos realizar cuando nos enfrentamos a un nuevo lenguaje de programación, es la escritura de un programa que muestre el clásico mensaje “Hola Mundo” por pantalla. Esto nos permite dar nuestro primer paso con el entorno y empezar a familiarizarnos con el mismo.

Así que cumpliendo con la tradición, vamos a proceder a escribir nuestro primer programa para VB.NET. A lo largo de los siguientes apartados de este tema, describiremos los pasos necesarios para la confección de una sencilla aplicación. El lector podrá comprobar que esta labor no encierra grandes complejidades, simplemente requiere un poco de entrenamiento y la adaptación a un nuevo conjunto de modos de programación.

Iniciar el IDE de VS.NET

El primer paso a dar es arrancar el entorno de desarrollo de VS.NET, para lo cual, seleccionaremos en la estructura de menús de Windows, la opción de menú situada en *Inicio + Programas + Microsoft Visual Studio .NET 7.0 + Microsoft Visual Studio .NET 7.0*, que ejecutará el IDE y nos mostrará el área principal de trabajo con la pestaña *Página de inicio*. Ver Figura 37 y Figura 38.

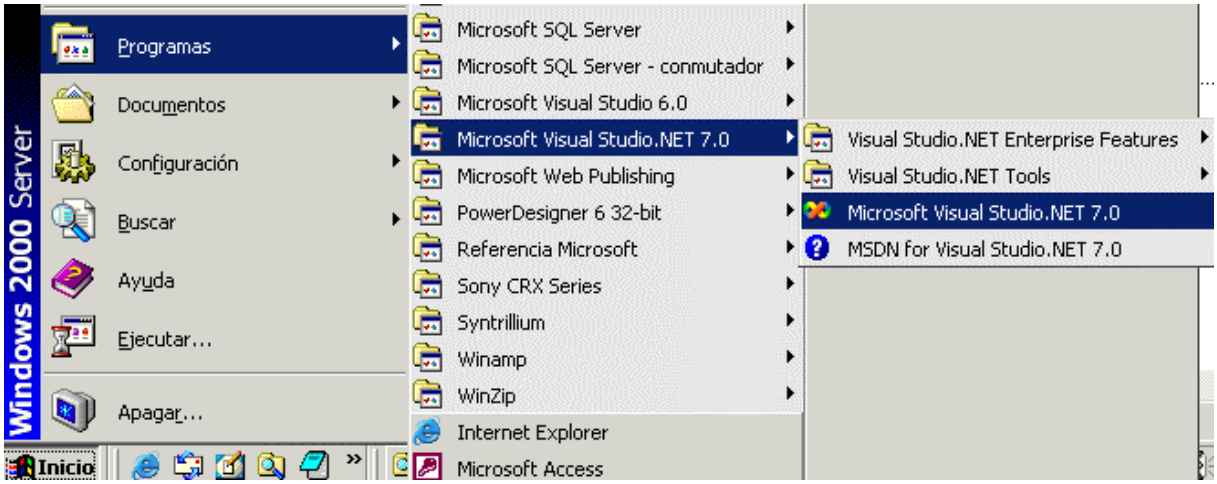


Figura 37. Opción de menú para acceder a Visual Studio .NET.

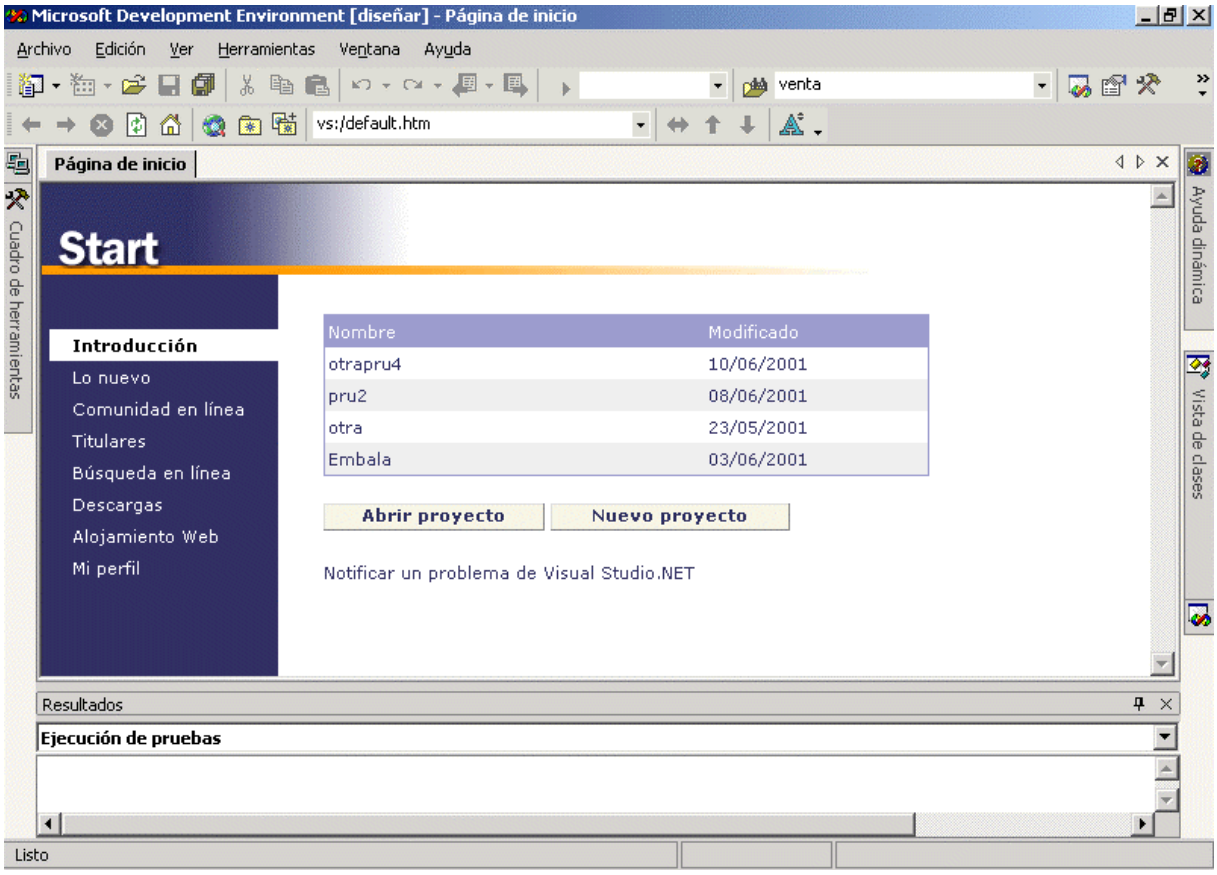


Figura 38. Pantalla inicial del IDE de Visual Studio .NET.

Este IDE es común para todos los lenguajes, como habrá podido observar el lector al iniciarlo, ya que a diferencia de versiones anteriores, no se selecciona la herramienta de trabajo y esta abre el entorno de programación, sino que directamente se abre el IDE y posteriormente elegiremos el lenguaje con el que vamos a escribir el programa.

Otro punto de diferencia con versiones anteriores reside en la disposición de los elementos dentro del IDE: el sistema de menús y barras de herramientas no ha variado, pero algunos componentes se hallan en pestañas desplegables, cuyo contenido se expande al situar el cursor del ratón sobre la pestaña.

Igualmente el área principal de trabajo se organiza también en base a una ventana con pestañas, que nos permite cambiar de contenido pulsando la pestaña correspondiente, en lugar de tener ventanas independientes. No vamos a extendernos aquí en cuestiones de configuración del IDE, ya que estos aspectos se tratan en un tema específico, por lo que vamos a seguir creando nuestro primer programa.

Crear un nuevo proyecto

A continuación, pulsaremos dentro de la página de inicio el botón *Nuevo proyecto*, que nos mostrará un cuadro de diálogo para seleccionar el lenguaje a usar y el tipo de aplicación que queremos obtener. Ver Figura 39.

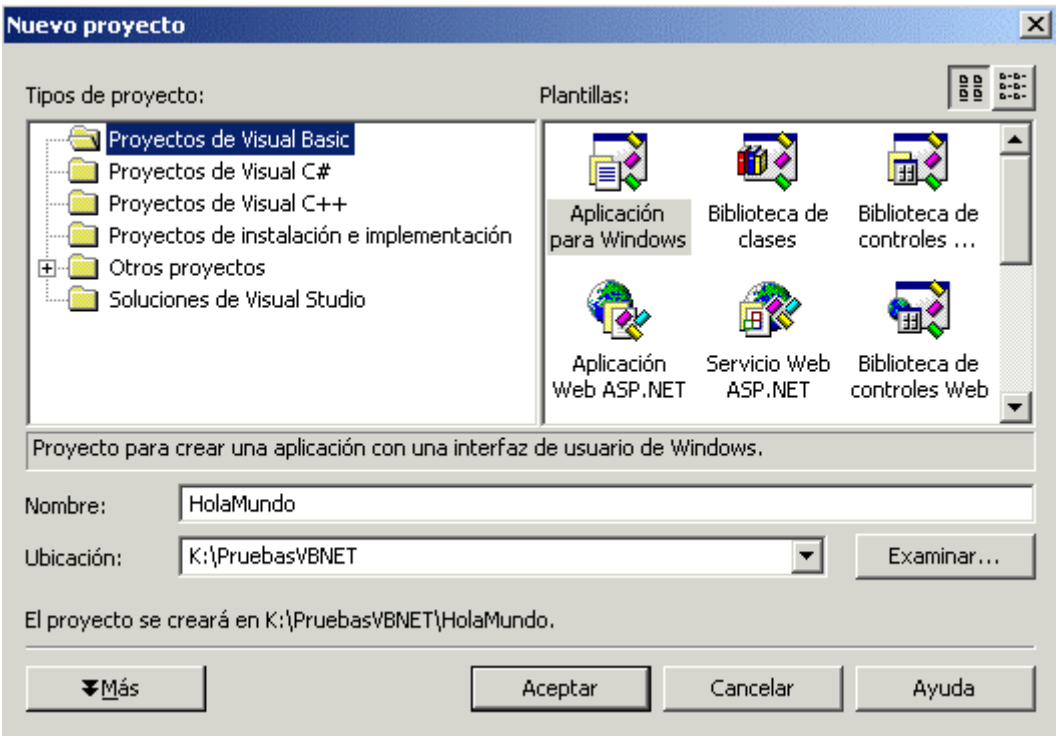


Figura 39. Selección de tipo de proyecto a crear.

Como podemos observar, en este cuadro de diálogo hay varios puntos a tener en cuenta que describimos seguidamente.

En la lista *Tipos de proyecto* podemos seleccionar el lenguaje en el que vamos a codificar el programa: Visual Basic, C#, C++; así como otra serie de asistentes de instalación, todo ello agrupado en diferentes carpetas. En este caso elegiremos *Proyectos de Visual Basic*.

Una vez que sabemos el lenguaje a usar, debemos elegir el tipo de aplicación en la lista *Plantillas*. Seleccionaremos *Aplicación para Windows* ya que vamos a crear un programa con interfaz típica de Windows.

La gran ventaja de las plantillas radica en que al crear la aplicación, nos proporciona la funcionalidad básica de la misma, que de otro modo tendríamos que codificar manualmente.

Por último, en el campo *Nombre* escribiremos *HolaMundo* como nombre para nuestra aplicación y en el campo *Ubicación* estableceremos la carpeta del disco duro que contendrá los ficheros del proyecto. Pulsando *Aceptar* se creará el nuevo proyecto.

Objetos, propiedades y métodos

Desde la versión 4.0, el lenguaje Visual Basic ha ido adoptando progresivamente principios del paradigma de la Programación Orientada a Objeto (Object Oriented Programming u OOP a partir de ahora), aunque con ciertas restricciones.

VB.NET es la primera versión de este lenguaje que incorpora plenas, excelentes y muy potentes características de orientación a objetos, esto es debido a que el lenguaje forma parte de la plataforma .NET, la cual está diseñada en su totalidad siguiendo un modelo de orientación a objetos, basado en un conjunto de especificaciones que obligan a todos los lenguajes que operen bajo este entorno a seguir los dictados de dichas normas. Por este motivo, todos los elementos que usemos en una aplicación VB.NET serán considerados objetos, que deberemos manipular a través de sus propiedades y métodos.

A partir de ahora, y a lo largo de todo el texto, se harán continuas referencias relacionadas con los fundamentos, terminología y técnicas de programación a objetos, ya que es tal la integración de estas características en todo el entorno, que es imposible realizar un mínimo acercamiento inicial sin tratar estos aspectos.

Somos conscientes de que el lector puede no tener una experiencia previa en OOP, por ello, le recomendamos la consulta de los temas dedicados a programación OOP, para resolver las cuestiones sobre fundamentos de orientación a objetos que se presenten.

-¿Y por qué no ir explicando la programación orientada a objetos desde el primer ejemplo?-. Suponemos que esta será una pregunta que se formulará el lector. Bien, el motivo es por intentar simplificar al máximo los primeros pasos a realizar con VB.NET. Por ello, en estos ejemplos iniciales se darán las mínimas explicaciones puntuales sobre los objetos que se utilicen, para evitar añadir una innecesaria complejidad, nada recomendable al comenzar a trabajar con un nuevo lenguaje.

Formularios

Una vez creado el proyecto, se añade un formulario al mismo, apareciendo una nueva pestaña en el área principal del IDE, que corresponde al diseñador del formulario. Ver Figura 40.

Dentro de una aplicación VB.NET, el término *formulario* designa a una ventana estándar de las que utilizamos habitualmente en Windows para comunicarnos con el usuario, mientras que el diseñador del formulario representa a la plantilla de una ventana, sobre la cuál añadiremos controles y modificaremos si es necesario su aspecto inicial.

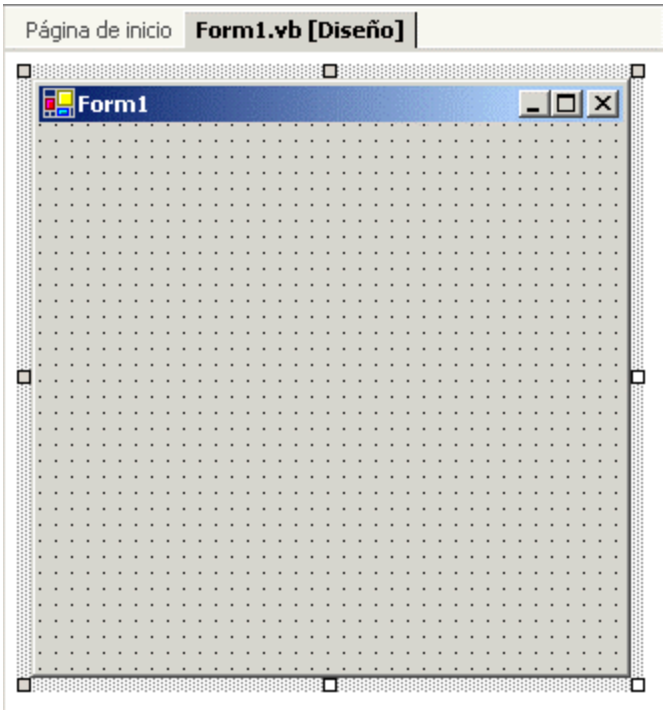


Figura 40. Diseñador de formulario de VB.NET.

El formulario como un objeto

Un formulario es, al igual que la gran mayoría de elementos en el entorno de .NET, un objeto, y como tal, la forma de manipularlo pasa por asignar y obtener valores de sus propiedades, y por la ejecución de sus métodos.

Debido a que un formulario dispone de un elevado número de propiedades y métodos, durante el texto nos centraremos sólo sobre los que vayamos a trabajar, pudiendo el lector, consultar el resto a través de la ayuda de VS.NET; esto es aplicable a todos los objetos con los que tratemos.

Acceso a las propiedades de un formulario

Para acceder a las propiedades de un formulario, podemos hacerlo de una de las siguientes maneras:

- Seleccionar la opción *Ver + Ventana Propiedades* del menú de VS.NET.
- Pulsar [F4].
- Hacer clic en el botón de la barra de herramientas correspondiente a la ventana de propiedades. Ver Figura 41

Figura 41. Botón Propiedades de la barra de herramientas.

- Situar el ratón en la pestaña *Propiedades,* que se halla generalmente en el margen derecho del IDE, que al expandirse, nos mostrará la ventana *Propiedades* para el objeto que tengamos en ese momento activo en el proyecto. Ver Figura 42.

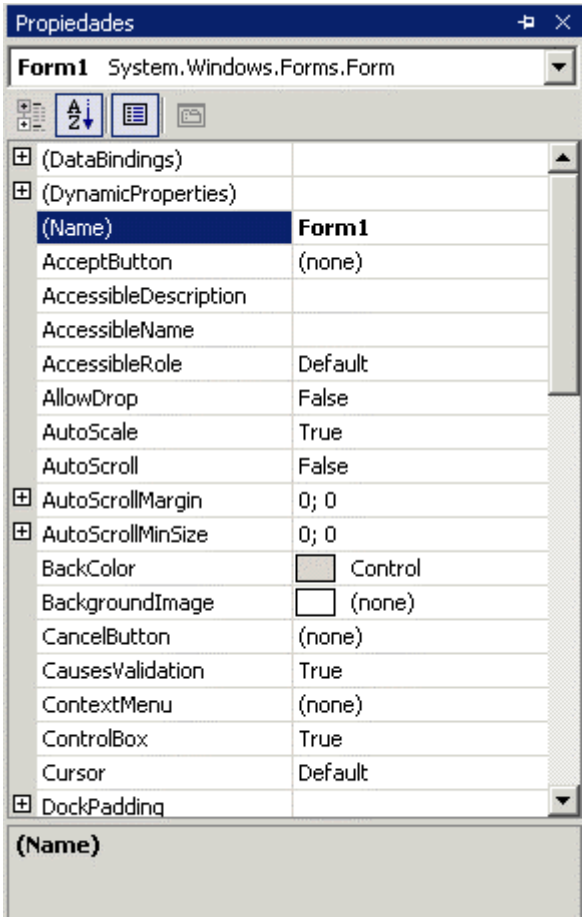


Figura 42. Ventana Propiedades de VS.NET, mostrando las propiedades de un formulario.

En esta ventana, los nombres de las propiedades aparecen en la columna izquierda y su valor en la derecha. Por defecto, las propiedades aparecen ordenadas por categorías, pero para acceder más rápidamente a ellas a través del nombre, vamos a ordenarlas alfabéticamente, pulsando el segundo botón de esta ventana comenzando por la izquierda. Ver Figura 43.

Figura 43. Botón para ordenar las propiedades alfabéticamente.

La primera propiedad a mencionar, y la más importante para cualquier objeto es *Name*, que contiene el nombre del objeto que luego nos va a permitir manipularlo en el código del programa. VB.NET asigna nombres por defecto a los formularios y controles que agreguemos a la aplicación. En este caso, el nombre que ha asignado al formulario es *Form1*. Podemos modificar estos nombres por otros que sean más significativos para el programador, sin embargo, para simplificar este ejemplo, mantendremos los nombres que sean asignados por defecto.

El formulario en su estado actual, muestra como título el mismo que tiene para el nombre. La propiedad que contiene el título del formulario es *Text*, y vamos a cambiarla por un valor que describa mejor la funcionalidad que queremos dar al programa.

Para ello, haremos clic sobre el valor de la propiedad Text y cambiaremos el literal que aparece por el siguiente: *Programa de prueba*. Al pulsar [INTRO], el diseñador del formulario mostrará el nuevo título.

Otro aspecto es referente a la posición del formulario en pantalla cuando ejecutemos el programa. Actualmente es Windows quien calcula dicha posición, apareciendo en la zona superior izquierda de la pantalla. Podemos modificar también esta posición, para ello haremos clic en la propiedad *StartPosition*, que mostrará un botón que al ser pulsado abrirá una lista con los posibles valores disponibles. Seleccionaremos *CenterScreen*, y cada vez que ejecutemos el programa, el formulario aparecerá siempre en el centro de la pantalla.

Controles

Los controles constituyen aquellos elementos que insertamos dentro de un formulario, y que permiten al mismo interactuar con el usuario, tales como botones de pulsación, cajas de texto, casillas de verificación, cajas con listas de valores, etc.; al igual que un formulario, son objetos con sus propiedades y métodos, y se manejan de la misma forma.

Para añadir un control a un formulario, en primer lugar situaremos el ratón sobre la pestaña *Cuadro de herramientas*, que al expandirse mostrará los controles disponibles, que podemos incluir en un formulario. Ver Figura 44.

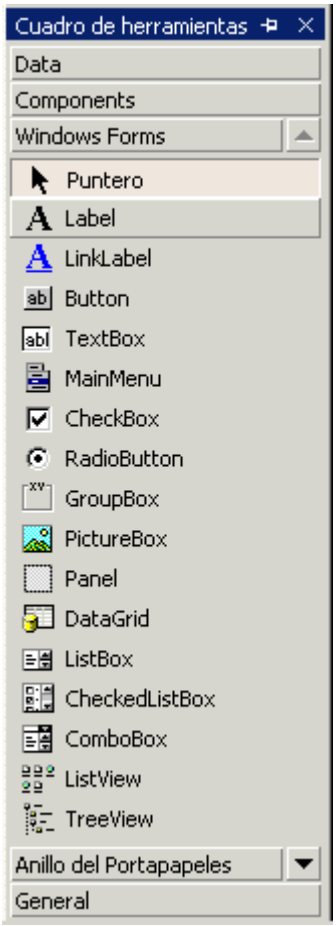


Figura 44. Cuadro de herramientas de VB.NET.

La operación de añadir un control a un formulario se denomina *dibujo de control,* y podemos realizarla de varias maneras:

- Haciendo doble clic sobre el icono del control, lo que situará una copia del control en el formulario con una posición y tamaño predefinidos por VB.NET.

- Haciendo clic sobre el icono del control, y a continuación clic sobre la superficie del formulario. El nuevo control se insertará desde el punto en que hemos pulsado, extendiéndose hacia la derecha y abajo con un tamaño predeterminado por el diseñador.

- Haciendo clic sobre el icono del control, y seguidamente clic sobre la posición del formulario en la que queramos comenzar a dibujar el control, arrastraremos y soltaremos dando nosotros el tamaño requerido al control.

Label

Un control *Label* o *Etiqueta* es un control estático. Eso quiere decir que no realiza ninguna interacción con el usuario, puesto que sólo muestra un texto informativo.

Dibujaremos sobre el formulario un control de este tipo del modo descrito anteriormente, al que el diseñador le asignará el nombre Label1. A continuación, con el control seleccionado, pasaremos a la ventana de propiedades. En la propiedad Text escribiremos *Hola Mundo*, lo cual se reflejará también en el control dentro del diseñador de formularios. Ver Figura 45.

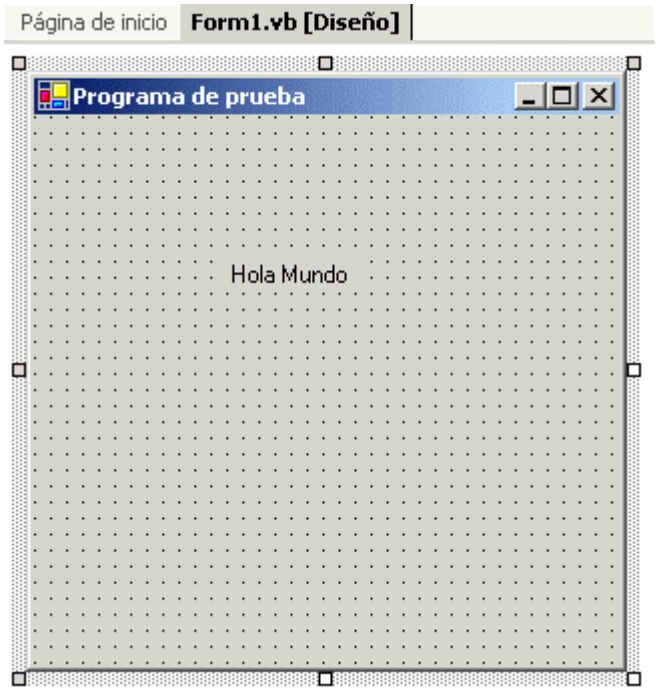


Figura 45. Control Label añadido a un formulario.

Ya que el tamaño de Label1 con respecto al formulario es insignificante, aumentaremos dicho tamaño haciendo clic sobre el control; esto mostrará alrededor del mismo una serie de recuadros o guías de redimensión. Haciendo clic sobre cualquiera de ellas y arrastrando el ratón, variaremos el tamaño del Label hasta conseguir uno más adecuado.

También podemos hacer clic sobre el control y arrastrarlo, cambiando la posición en la que lo habíamos dibujado originalmente.

Ahora debemos cambiar el tamaño del tipo de letra, y para ello emplearemos la propiedad *Font* o Fuente del control. Pasaremos pues a la ventana de propiedades, observando como esta propiedad muestra en su valor el nombre del fuente actual. Ver Figura 46.

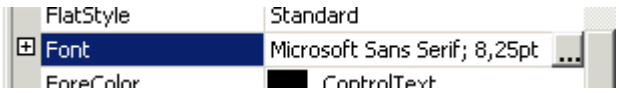


Figura 46. Ventana de propiedades con Font seleccionada.

Haciendo clic sobre Font, aparecerá un botón con puntos suspensivos, que al ser pulsado, abrirá el cuadro de diálogo estándar del sistema para selección de tipos de letra. Ver Figura 47

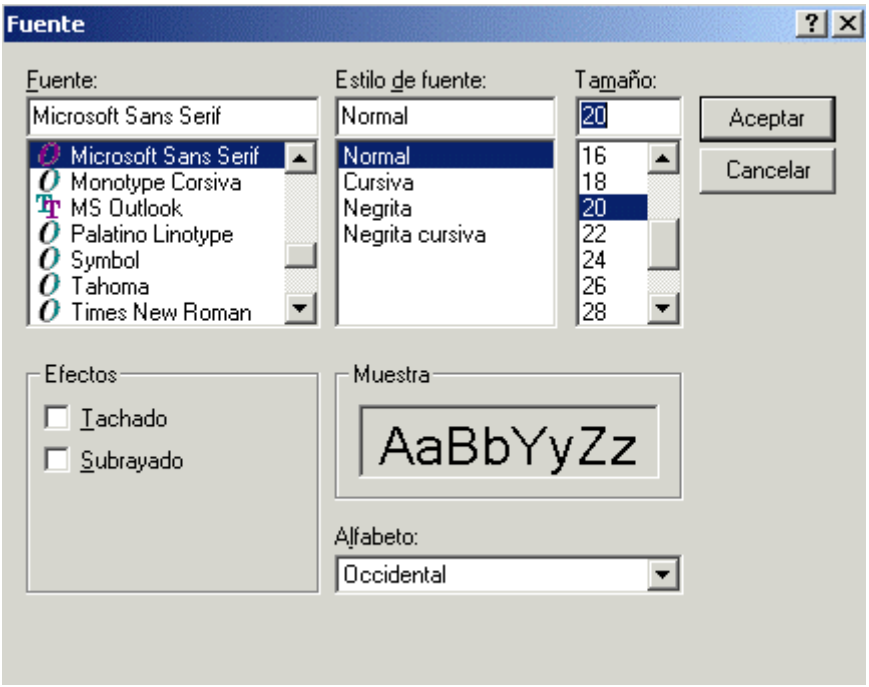


Figura 47. Selección del tipo de fuente para un control del formulario.

Cambiando el tamaño del tipo de letra a 20 y pulsando *Aceptar*, aumentará la letra del Label que tenemos en el formulario.

Ejecutando la aplicación

En este punto del desarrollo, daremos por concluida la aplicación. Ahora debemos ejecutarla para comprobar que todo funciona correctamente. Podemos hacerlo empleando una de las siguientes formas:

- Seleccionar la opción *Depurar* + *Iniciar* en el menú de VS.NET.
- Pulsar [F5].

- Pulsar el botón Iniciar de la barra de herramientas. Ver Figura 48.

Figura 48. Botón Iniciar de la barra de herramientas de VS.NET.

Esta acción compilará el proyecto y generará la aplicación, ejecutándola desde el propio IDE. El resultado será la visualización del formulario de la Figura 49.

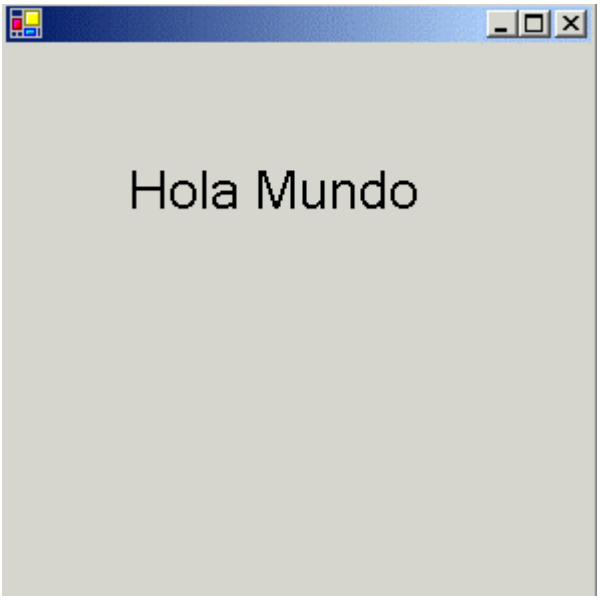


Figura 49. La aplicación Hola Mundo en ejecución.

Este formulario podemos manipularlo de igual forma que cualquier otro de los que existan en el sistema: redimensionarlo, maximizarlo, minimizarlo, etc.

Como ventaja añadida, observe el lector que para crear este programa no ha necesitado escribir ni una sola línea de código, todo lo ha realizado a través de los diseñadores y demás elementos del IDE.

El código de la aplicación

¿Quiere lo anterior decir que un formulario no tiene código?. La respuesta es no, toda aplicación VB.NET tiene su correspondiente código, lo que ocurre en el caso del formulario que acabamos de crear, es que al haberlo hecho desde el diseñador de formulario, su código lo genera el IDE por nosotros, lo que supone una gran ayuda.

Para acceder al código fuente del formulario, hemos de hacerlo de alguna de las siguientes maneras:

- Seleccionar la opción *Ver + Código* en el menú de VS.NET.
- Pulsar [F7].
- Hacer clic derecho sobre el formulario y elegir la opción *Ver código* del menú contextual que aparece.

Cualquiera de las anteriores acciones abre una nueva pestaña en la zona principal del IDE, mostrando el *editor de código* del formulario. Ver Figura 50.

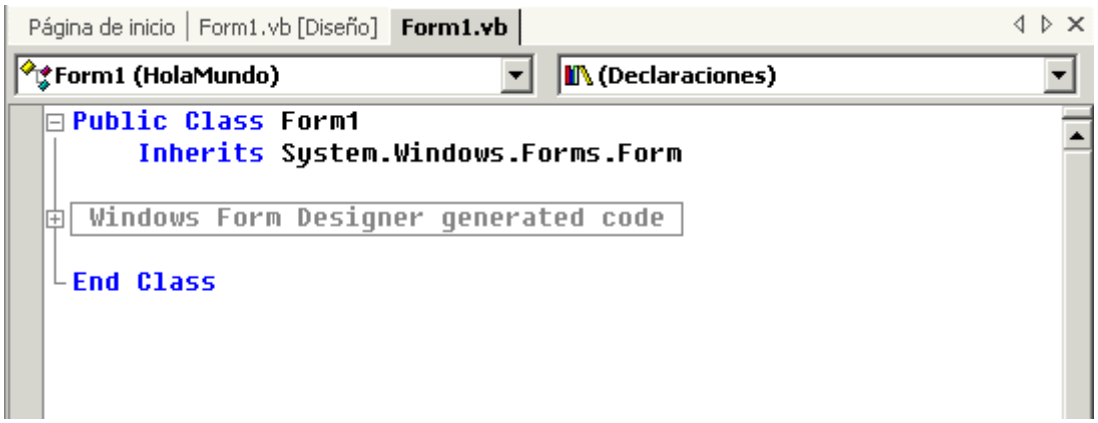


Figura 50. Ventana de código del formulario.

Sin embargo falta un pequeño detalle, ya que evidentemente, no es posible que un formulario tenga tan pocas líneas de código. Lo que ocurre es que el código generado por el diseñador, está oculto por una técnica denominada *Esquematización (Outlining)*, que permite definir zonas y regiones de código que pueden ser expandidas o contraídas desde el editor de código, haciendo clic en los indicadores de la región correspondiente.

En este caso, el diseñador ha creado una región con el nombre *Windows Form Designer generated code*, y la ha contraído. Podemos reconocer una región de código contraída porque su nombre se encuentra dentro de un rectángulo.

Para expandir una región de código, haremos clic en el signo + que se encuentra a la izquierda de su nombre, lo que mostrará su contenido al completo. En nuestro caso accederemos al código del formulario Form1, que reproducimos en el Código fuente 9.

```
Public Class Form1
    Inherits System.Windows.Forms.Form

#Region " Windows Form Designer generated code "

    Public Sub New()
        MyBase.New()

        'This call is required by the Windows Form Designer.
        InitializeComponent()

        'Add any initialization after the InitializeComponent() call

    End Sub

    'Form overrides dispose to clean up the component list.
    Protected Overloads Overrides Sub Dispose(ByVal disposing As Boolean)
        If disposing Then
            If Not (components Is Nothing) Then
                components.Dispose()
            End If
        End If
        MyBase.Dispose(disposing)
    End Sub
```

```
    Friend WithEvents Label1 As System.Windows.Forms.Label

    'Required by the Windows Form Designer
    Private components As System.ComponentModel.Container

    'NOTE: The following procedure is required by the Windows Form Designer
    'It can be modified using the Windows Form Designer.
    'Do not modify it using the code editor.
    <System.Diagnostics.DebuggerStepThrough()> Private Sub InitializeComponent()
        Me.Label1 = New System.Windows.Forms.Label()
        Me.SuspendLayout()
        '
        'Label1
        '
        Me.Label1.Font = New System.Drawing.Font("Microsoft Sans Serif", 20.25!,
System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, CType(0,
Byte))
        Me.Label1.Location = New System.Drawing.Point(56, 56)
        Me.Label1.Name = "Label1"
        Me.Label1.Size = New System.Drawing.Size(184, 40)
        Me.Label1.TabIndex = 0
        Me.Label1.Text = "Hola Mundo"
        '
        'Form1
        '
        Me.AutoScaleBaseSize = New System.Drawing.Size(5, 13)
        Me.ClientSize = New System.Drawing.Size(292, 273)
        Me.Controls.AddRange(New System.Windows.Forms.Control() {Me.Label1})
        Me.Name = "Form1"
        Me.StartPosition = System.Windows.Forms.FormStartPosition.CenterScreen
        Me.Text = "Programa de prueba"
        Me.ResumeLayout(False)

    End Sub

#End Region

End Class
```

Código fuente 9.

El lector puede estar sorprendido ante tal cantidad de código para un simple formulario, por lo que vamos a analizar brevemente cada una de las partes que componen este fuente para comprobar que todas son necesarias.

Diseccionando el código fuente del formulario

Todo este código que vemos por primera vez nos puede parecer muy complejo. En el caso de un programador de VB6, puede hacer retroceder la vista hacia dicha versión y en su sencillez a la hora de crear un formulario, -si es una nueva versión de VB ¿por qué es, o al menos parece más complicado, cuando debería de ser todo lo contrario?-.

Bien, si analizamos la situación y reflexionamos un poco, veremos como esta aparente dificultad no es tal. Por una parte, pensemos en las ventajas de poder crear un formulario desde código, esto abre la puerta a muchas posibilidades que teníamos cerradas en VB6, donde el código de creación del formulario no era accesible por el programador. Por otro lado, todo el código del formulario ha sido creado automáticamente por el diseñador, de manera que en circunstancias normales, no tendremos que preocuparnos por escribirlo, sino que será el propio diseñador quien lo genere por nosotros.

A continuación, y para poder asimilar mejor todas estas novedades, vamos a proceder a examinar el anterior código fuente por partes y de un modo general, por lo que ciertas instrucciones, palabras clave, técnicas, etc., no serán tratadas aquí, explicándose cada una en su correspondiente tema del texto.

La clase del formulario

Todo formulario es un objeto y el código de un objeto se escribe en una clase. Por dicho motivo, debemos crear una nueva clase usando las palabras clave Class...End Class y heredar esta clase de la clase Form con la palabra clave Inherits. La clase Form se encuentra en el espacio de nombres System.Windows.Forms, y contiene todas las clases relacionadas con el formulario y controles. Ver Código fuente 10.

```
Public Class Form1
    Inherits System.Windows.Forms.Form
    . . .
    . . .
End Class
```

Código fuente 10

Por el mero hecho de establecer una relación de herencia entre Form y Form1, toda la funcionalidad de la clase Form (que representa la clase padre o base) pasa a la nueva clase Form1 (que representa a la clase hija o derivada), a la que podremos añadir nuevas características propias, de forma similar a como se establece una relación real padre-hijo.

Para más información sobre los espacios de nombres, consulte el lector el tema dedicado a .NET Framework.

El método constructor New( )

Toda clase debe tener lo que se denomina un *método constructor*, que es el primero que se ejecuta cuando es instanciado un objeto de la clase, y en él se escribe el código de inicialización para el objeto.

Dentro de VB.NET el método constructor para una clase se debe llamar New( ). Ver Código fuente 11.

```
    Public Sub New()
        MyBase.New()

        'This call is required by the Windows Form Designer.
        InitializeComponent()

        'Add any initialization after the InitializeComponent() call

    End Sub
```

Código fuente 11

Puesto que nuestra clase Form1 hereda de la clase Form, en el método constructor de Form1 llamamos al método constructor de la clase base con MyBase.New( ). La palabra clave MyBase representa al objeto de la clase padre. Por último llamamos al método InitializeComponent( ), que como su nombre indica, se encarga de inicializar los componentes del formulario: configura las propiedades del formulario, crea y añade los controles.

Configuración del formulario y creación de controles

Siguiendo un orden lógico de ejecución, pasemos ahora al método InitializeComponent( ). Antes de comenzar a ejecutarlo se declaran dos variables: Label1, un objeto que representa al control Label que hemos incluido en el formulario, y components, que se trata de un objeto en el que se incluirán los elementos no visuales del formulario. En nuestro ejemplo no se realiza uso de este objeto

Es importante destacar la advertencia que incluye el diseñador del formulario en el código, justo antes de comenzar este método, indicando que no se modifique dicho método mediante código, sino que se utilice el diseñador para tal labor.

Dentro del código de InitializeComponent( ) propiamente dicho, la palabra clave *Me* nos permite referirnos a cualquier elemento del formulario (propiedades, métodos, etc), desde dentro del propio formulario. Aunque no es necesario el uso de Me, se recomienda por hacer más fácil la lectura del código; el propio IDE al generar el código utiliza esta palabra, lo cual es indicativo de su importancia.

Se instancia el control Label1, y se asigna valor a sus propiedades, para más adelante, agregarlo a la lista de controles del formulario con el método AddRange( ) de la colección Controls del formulario.

También se establecen valores para las propiedades del formulario, y durante todo este tiempo, para evitar efectos extraños de visualización, esta es suspendida para el formulario, y se vuelve a reanudar al final. Esto se consigue con los métodos SuspendLayout( ) y ResumeLayout( ) del formulario. Ver el Código fuente 12.

```
    Friend WithEvents Label1 As System.Windows.Forms.Label

    'Required by the Windows Form Designer
    Private components As System.ComponentModel.Container

    'NOTE: The following procedure is required by the Windows Form Designer
    'It can be modified using the Windows Form Designer.
    'Do not modify it using the code editor.
    <System.Diagnostics.DebuggerStepThrough()> Private Sub InitializeComponent()
        Me.Label1 = New System.Windows.Forms.Label()
        Me.SuspendLayout()
        '
        'Label1
        '
        Me.Label1.Font = New System.Drawing.Font("Microsoft Sans Serif", 20.25!,
System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, CType(0,
Byte))
        Me.Label1.Location = New System.Drawing.Point(56, 56)
        Me.Label1.Name = "Label1"
        Me.Label1.Size = New System.Drawing.Size(184, 40)
        Me.Label1.TabIndex = 0
        Me.Label1.Text = "Hola Mundo"
        '
        'Form1
        '
        Me.AutoScaleBaseSize = New System.Drawing.Size(5, 13)
        Me.ClientSize = New System.Drawing.Size(292, 273)
```

```
        Me.Controls.AddRange(New System.Windows.Forms.Control() {Me.Label1})
        Me.Name = "Form1"
        Me.StartPosition = System.Windows.Forms.FormStartPosition.CenterScreen
        Me.Text = "Programa de prueba"
        Me.ResumeLayout(False)

    End Sub
```

Código fuente 12

Liberación de recursos del formulario

El método Dispose( ) del formulario, sirve para indicarle al entorno de ejecución de .NET Framework, que el formulario ya no se va a utilizar y que todos los recursos que ha estado usando, quedan de nuevo a disposición del entorno para que el recolector de basura de la plataforma, los recupere cuando considere necesario. Esta acción se emplea tanto para los componentes, representados por la variable components, como para el propio formulario, en la llamada que hace a este mismo método en su clase base. Ver Código fuente 13.

```
    'Form overrides dispose to clean up the component list.
    Protected Overloads Overrides Sub Dispose(ByVal disposing As Boolean)
        If disposing Then
            If Not (components Is Nothing) Then
                components.Dispose()
            End If
        End If
        MyBase.Dispose(disposing)
    End Sub
```

Código fuente 13

Estructura y grabación del proyecto

Al crear un nuevo proyecto en VB.NET, se genera en disco, partiendo de la ruta especificada en la ventana de creación del proyecto, una estructura de directorios, que contiene los archivos que forman parte del proyecto. La Figura 51 muestra la estructura correspondiente al programa de ejemplo Hola Mundo.

Figura 51. Estructura de directorios de un proyecto VB.NET.

Si modificamos los elementos del proyecto (formularios, clases, módulos, etc), debemos grabar los cambios en alguna de las siguientes maneras:

- Opción *Generar* + *Generar* del menú de VS.NET.

- Opción *Depurar* + *Iniciar* del menú de VS.NET.

- Pulsando [F5] al ejecutar el proyecto en modo de prueba. Los elementos del proyecto que se hayan modificado y no se hayan grabado, se grabarán automáticamente.

Un proyecto VB.NET está compuesto por un conjunto de ficheros, cuyos tipos han variado notablemente desde VB6. Clasificados por su extensión, a continuación se relacionan algunos de estos ficheros:

- **VB**. Código fuente escrito en lenguaje Visual Basic. A diferencia de VB6, en el que había diferentes tipos de ficheros en función de si se trataba de un formulario, clase, módulo de código, etc., un fichero con extensión VB puede contener cualquier tipo de código en VB: clases, módulos de código, etc.

- **VBPROJ**. Proyecto de VB. Contiene información sobre todos los elementos que forman parte de un proyecto: ficheros de código, referencias, etc.

- **SLN** (Solución). Una solución es el medio que utiliza VS.NET para agrupar varios proyectos escritos en el mismo o en diferentes lenguajes de los que integran la plataforma .NET.

- **VBPROJ.USER**. Información sobre las opciones de usuario del proyecto.

- **RESX**. Plantilla de recursos en formato XML.

- **EXE**. Aplicación ejecutable.

- **PDB**. Información sobre depuración de la aplicación

En el directorio *bin* del proyecto se genera el fichero ejecutable, que contiene nuestra aplicación y que en principio es lo único que necesitamos para ejecutar el programa en cualquier otro equipo, que naturalmente, también tenga instalado la plataforma .NET Framework. Ello nos evita problemas y ahorra tiempo de instalación.

Una vez grabado el proyecto a disco, podemos dar por concluido el desarrollo de nuestro primer programa Hola Mundo.

6

Escritura de código

Escribir código, el papel clásico del programador

En el tema anterior hemos realizado un primer acercamiento al desarrollo de programas en VB.NET, creando el típico Hola Mundo, y utilizando VS.NET como herramienta de trabajo; comprobando también, algunas de las múltiples facilidades que nos proporciona el IDE para la construcción de aplicaciones.

Sin embargo, por muy sofisticado que sea un entorno de trabajo, hay un aspecto que siempre ha sido propio del programador, y que no podrá ser eliminado: la escritura del código.

Es cierto que la generación de código automática por parte de los asistentes supone una gran ayuda, pero nunca se ajustará a las necesidades específicas que requieren los programas, por lo que el programador será el que tenga siempre la última palabra y deba retocar o añadir el código necesario para que el comportamiento de la aplicación sea el requerido.

Un programa escribiendo su código

En este tema vamos a desarrollar un sencillo programa, en el que a diferencia del ejemplo mostrado en el tema anterior, escribiremos nosotros el código en lugar de dejar al IDE que lo haga de forma automática.

Será un programa como hemos dicho muy sencillo, no diseñaremos formularios, solamente tomaremos un valor que introducirá el usuario por pantalla y lo mostraremos posteriormente. La idea principal es

que el lector aprenda como configurar el proyecto para establecer el lugar por el que se iniciará el programa, y el modo de escribir código para el mismo.

Crear el proyecto

En primer lugar, iniciaremos el IDE de VS.NET y crearemos un nuevo proyecto en VB.NET de la misma forma que la explicada en el tema anterior. El nombre que daremos al proyecto será EscribirCodigo.

Al igual que en el ejemplo HolaMundo, este tipo de proyecto crea un formulario vacío, pero no vamos a hacer uso del mismo. A continuación, agregaremos un nuevo módulo al proyecto para el código que vamos a escribir.

Un nuevo módulo de código

Mediante la opción de menú de VS.NET *Proyecto + Agregar módulo*, se abrirá la caja de diálogo *Agregar nuevo elemento*, con la que podremos añadir a nuestro proyecto un módulo (como este caso), formulario, clase, etc., seleccionando dicho elemento del panel derecho. Ver Figura 52.

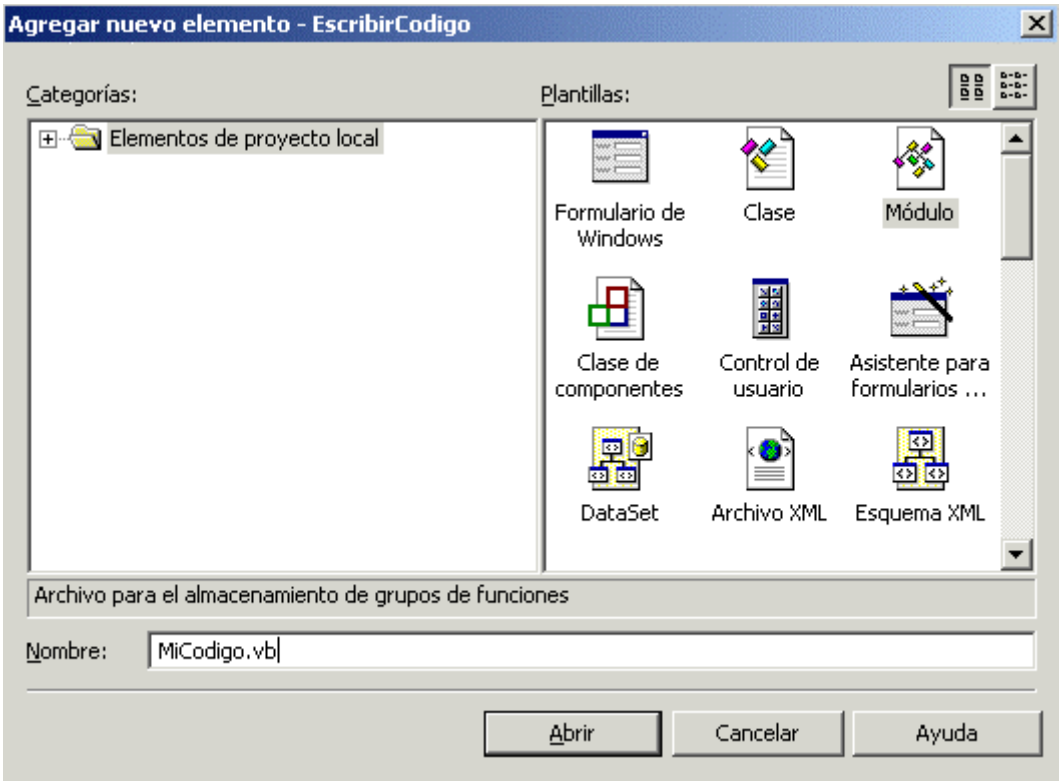


Figura 52. Añadir un nuevo módulo al proyecto.

Daremos el nombre MiCodigo.VB al módulo, con lo que se creará el nuevo módulo en un fichero y se mostrará una nueva pestaña en la ventana principal del IDE con el editor de código para el módulo. Ver Figura 53. Un módulo se define mediante las palabras clave *Module...End Module,* que indican respectivamente el comienzo y fin del módulo, y entre ellas escribiremos el código que va a contener: procedimientos, declaraciones, etc.

Figura 53. Editor de código para el módulo.

La ubicación física del código

Como vimos en el ejemplo anterior, el código de un programa se escribe en ficheros con extensión .VB. Ahora hemos añadido un módulo al proyecto que se ha creado en un nuevo fichero de este tipo.

Los ficheros .VB pueden contener cualquier instrucción en lenguaje VB.NET, de manera que es posible tener un único fichero .VB con todo el código de la aplicación, en el caso de que esta no sea muy extensa, tal y como muestra de forma esquemática el Código fuente 14.

```
CodigoProg.VB
=============
Class Cliente
    ' código de la clase
    ' ............
    ' ............
End Class

Module General
    ' código del módulo
    ' ............
    ' ............
End Module

Class Facturas
    ' código de la clase
    ' ............
    ' ............
End Class
```

Código fuente 14

Aunque también podemos añadir todos los ficheros .VB que necesitemos al proyecto, para tener nuestro código organizado por procedimientos generales, procedimientos específicos de la aplicación, clases de formularios, clases no visuales, etc., este el modo recomendado de organización cuando el programa es muy grande. Ver Código fuente 15. En cualquier caso, disponemos de una gran flexibilidad a este respecto.

```
CodigoClases.VB
=============
Class Cliente
```

```
    ' código de la clase
    ' ............
    ' ............
End Class

Class Facturas
    ' código de la clase
    ' ............
    ' ............
End Class
**************************************************

RutinasVarias.VB
================
Module General
    ' código del módulo
    ' ............
    ' ............
End Module

Module Listados
    ' código del módulo
    ' ............
    ' ............
End Module
```

Código fuente 15

Comentarios de código

Para introducir un comentario aclaratorio en el código de un programa utilizaremos la comilla simple (‘), seguida del texto correspondiente al comentario. Podemos insertar comentarios desde el comienzo de línea o a continuación de código ejecutable. Ver Código fuente 16.

```
Sub Prueba()
    ' este es un comentario desde el principio de línea
    Dim Valor As Integer

    Valor = 158   ' este es un comentario junto a una línea de código

End Sub
```

Código fuente 16

Procedimientos

Dentro de cualquier lenguaje de programación, un procedimiento o rutina de código es aquella entidad que guarda un conjunto de líneas de código que serán ejecutadas al llamar al procedimiento desde cualquier otro punto del programa.

Para crear un procedimiento en el programa usaremos las palabras clave *Sub...End Sub*, y entre las mismas escribiremos las instrucciones del procedimiento. El Código fuente 17 muestra un ejemplo.

```
Sub Prueba()
```

```
    ' instrucción1
    ' instrucción2
    ' instrucción3
    ' ...........
    ' ...........
    ' ...........
    ' instrucciónN
End Sub
```

Código fuente 17.

Los procedimientos podemos incluirlos en cualquier lugar dentro del programa. En el ejemplo actual, escribiremos un procedimiento en el módulo MiCodigo, al que daremos el nombre de Main( ) y que representa el procedimiento por el cual se comienza a ejecutar la aplicación.

El punto de entrada al programa

Todo programa debe tener un punto de entrada, o elemento que sea el que se comienza a ejecutar en primer lugar.

En el caso de una aplicación con estilo Windows, lo primero que comienza a ejecutarse por defecto es el formulario. Sin embargo, puede haber ocasiones en que no queramos ejecutar el formulario en primer lugar, bien porque debamos establecer valores de inicialización para el programa, en un procedimiento que sea el que da paso al formulario, o simplemente, puede que sólo queramos ejecutar uno o varios procedimientos sin usar formularios.

En esta situación, debemos escribir un procedimiento especial al que daremos el nombre Main( ), y que en VB.NET representa el punto de entrada a la aplicación, antes incluso que el propio formulario. También debemos configurar el proyecto, para que conozca la existencia de dicho procedimiento y lo ejecute en primer lugar.

En el módulo de código, escribiremos por lo tanto este procedimiento, aunque de momento vacío. Ver Código fuente 18.

```
Module MiCodigo

    Sub Main()


    End Sub

End Module
```

Código fuente 18

La clase MessageBox

Queremos mostrar un aviso cuando empecemos a ejecutar la aplicación, por lo que podemos usar la clase MessageBox. Esta es una clase del sistema, que permite mostrar un mensaje en pantalla al usuario mediante su método Show( ), y una cadena de caracteres que pasaremos como parámetro a dicho método.

Se trata de una clase *no instanciable*, es decir, no permite que creemos objetos a partir de ella. Al utilizarla, es el entorno el encargado de crear un objeto compartido, que usaremos para llamar a sus miembros compartidos o shared.

Para usar esta clase en nuestro procedimiento Main( ), podemos hacerlo como se muestra en el Código fuente 19.

```
Sub Main()

    MessageBox.Show("Empieza el programa")

End Sub
```

Código fuente 19

Configurar el punto de entrada del proyecto

Si ahora ejecutamos el proyecto, seguirá apareciendo el formulario y no el mensaje que esperábamos. Esto es debido a que no hemos configurado el proyecto para que se inicie por Main( ).

Para establecer qué elemento del proyecto debe ejecutarse en primer lugar, debemos acudir a las propiedades del proyecto, a las que accedemos a través de la ventana *Explorador de soluciones*, de alguna de las siguientes formas:

- Opción *Ver* + *Explorador de soluciones*, del menú de VS.NET.
- Situando el cursor del ratón sobre la pestaña *Explorador de soluciones*, se expandirá su ventana.
- Haciendo clic sobre el botón de la barra de herramientas correspondiente a esta opción. Ver Figura 54.

Figura 54. Botón para abrir el Explorador de soluciones.

Cualquiera de estas vías, nos llevará a la ventana de la Figura 55.

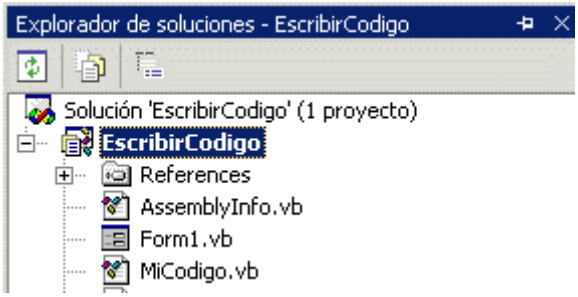


Figura 55. Explorador de soluciones de VS.NET.

Seguidamente haremos clic sobre el nombre del proyecto, y para acceder a sus propiedades emplearemos alguna de las siguientes formas:

- Haremos clic sobre el tercer botón de esta ventana, que corresponde a las propiedades del elemento seleccionado. Ver Figura 56.

Figura 56. Botón de propiedades del Explorador de soluciones.

- Seleccionaremos la opción de menú de VS.NET *Proyecto + Propiedades*.
- Haremos clic sobre el nombre del proyecto en el Explorador de soluciones, y seleccionaremos la opción *Propiedades* del menú contextual.

Cualquiera de estas acciones nos mostrará la ventana de propiedades del proyecto en ella, debemos abrir la lista desplegable del elemento *Objeto inicial*, que actualmente mostrará el nombre del formulario como objeto inicial, y seleccionar *Sub Main*. Pulsaremos Aceptar y a partir de ese momento, el entorno de ejecución buscará un procedimiento con el nombre Main( ), para ejecutar en primer lugar. Ver Figura 57.

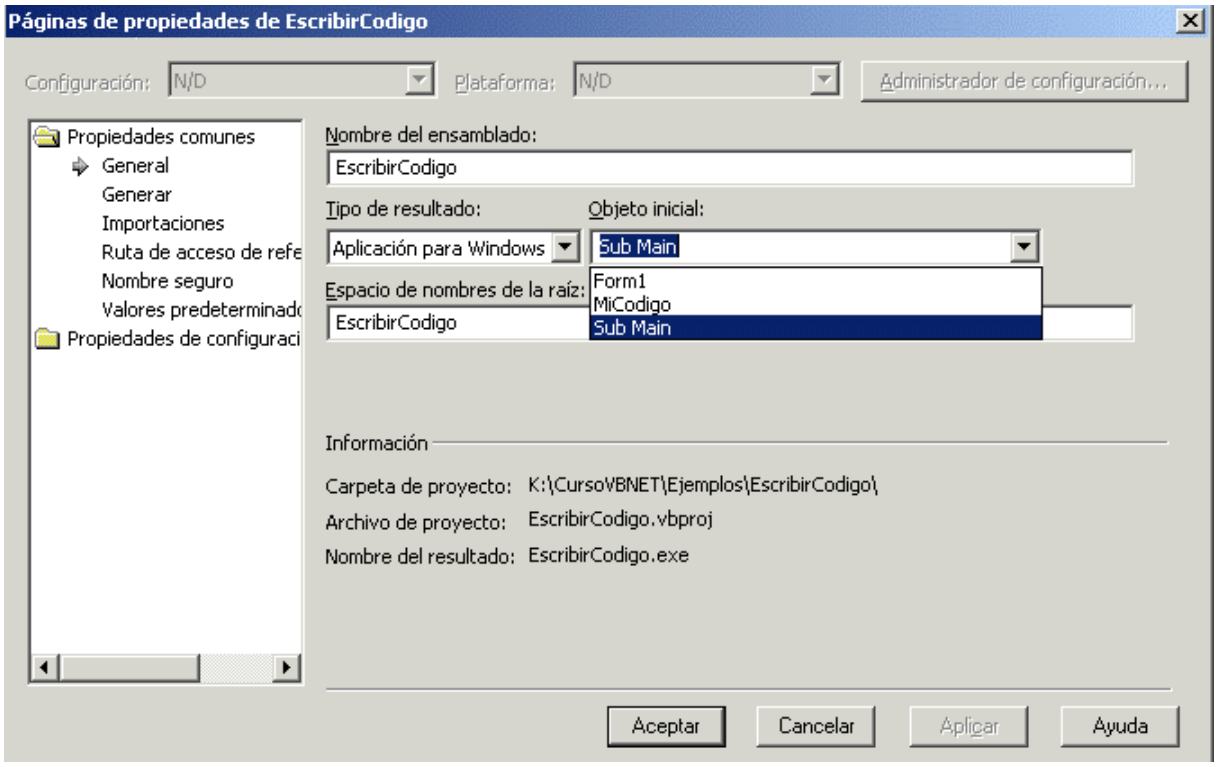


Figura 57. Ventana de propiedades del proyecto.

El resultado ahora, al ejecutar el programa, será el mensaje que mostramos a través de MessageBox. Ver Figura 58.

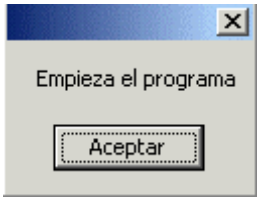


Figura 58. Mensaje mostrado desde Main( ).

Variables

Creemos que mostrar un simple mensaje es insuficiente en este ejemplo, por lo que vamos además, a pedir al usuario que introduzca un valor, que mostraremos en otro mensaje posterior. Dicho valor lo almacenaremos en una variable del programa.

Para declarar variables en VB.NET utilizaremos la instrucción *Dim*, seguida del nombre de la variable y el tipo de dato que queremos asignarle. En Main( ) declararemos una variable como muestra el Código fuente 20.

```
Sub Main()

    MessageBox.Show("Empieza el programa")

    Dim Nombre As String

End Sub
```

Código fuente 20

InputBox( )

InputBox( ) es una función que muestra una caja de diálogo en la que el usuario puede introducir un valor, que será devuelto al aceptar dicha caja. El Código fuente 21 muestra el formato de InputBox( ).

```
InputBox(Mensaje,Título,RespuestaDefecto,XPosicion,YPosicion)
```

Código fuente 21

- **Mensaje**. Obligatorio. Cadena de caracteres con el texto que va a mostrar la caja de diálogo.
- **Título**. Opcional. Título que aparecerá en la caja de diálogo.
- **RespuestaDefecto**. Opcional. Cadena de caracteres con el valor que devolverá esta función, en el caso de que el usuario no escriba nada.
- **XPosicion, YPosicion**. Opcionales. Valores numéricos que indican las coordenadas en donde será mostrada la caja. Si se omiten, se mostrará en el centro de la pantalla.

Completando el procedimiento

Llegados a este punto del ejemplo, tenemos todos los ingredientes para completarlo. Necesitamos que el usuario introduzca su nombre utilizando InputBox( ), volcar dicho nombre en la variable que hemos declarado y mostrarlo usando otro MessageBox. Todo ello lo vemos en el Código fuente 22.

```
Sub Main()

    MessageBox.Show("Empieza el programa")

    Dim Nombre As String

    Nombre = InputBox("Escribe tu nombre")

    MessageBox.Show("El nombre del usuario es: " & Nombre, "Programa de prueba")

End Sub
```

Código fuente 22

Describamos los últimos pasos que hemos dado:

Después de la declaración de la variable Nombre, llamamos a la función InputBox( ). Como dicha función devuelve una cadena de caracteres con el valor que haya escrito el usuario, necesitamos recuperarla de alguna forma, y esta es asignando el resultado de la llamada a la función en la variable. La Figura 59 muestra la caja de diálogo resultante de InputBox( ).

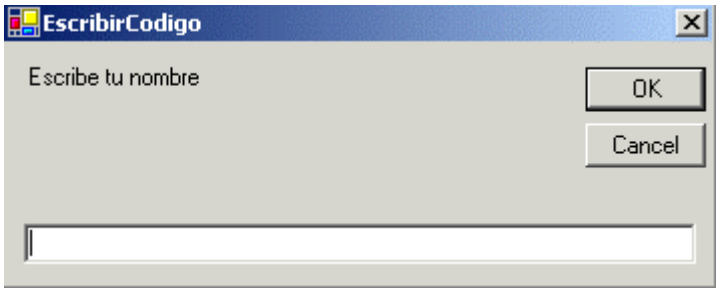


Figura 59. Ejecución de la función InputBox( ).

Después de escribir su nombre en el campo de la caja, si el usuario pulsa OK, InputBox( ) devolverá el valor de dicho campo a la variable Nombre. Por último, mostraremos el valor de la variable usando el método Show( ) de MessageBox, pero con algunas variaciones respecto a la primera vez que utilizamos esta clase en Main( ). En este caso concatenamos una cadena de caracteres al nombre, para ello debemos utilizar el operador &, y empleamos un segundo parámetro, que muestra un título en la ventana del mensaje. Ver Figura 60.

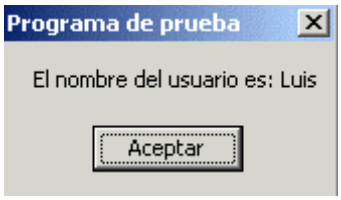


Figura 60. Mensaje con el valor resultante obtenido al llamar a InputBox( ).

Finalizamos el programa

Tras la ejecución del programa para comprobar que todo funciona correctamente, grabamos si es necesario, los últimos cambios del proyecto y podemos dar por concluida la aplicación.

7

Una aplicación con funcionalidad básica

Integrando lo visto hasta el momento

Los ejemplos de los temas anteriores constituyen un buen comienzo, y nos han permitido dar nuestros primeros pasos tanto con el lenguaje como con el IDE, pero evidentemente, no nos van a llevar muy lejos si lo que pretendemos es crear aplicaciones con algo más de contenido.

En este tema no vamos a entrar todavía en los detalles del IDE ni en el lenguaje. Para que el lector siga familiarizándose con el entorno, daremos unos pequeños pasos iniciales más; con ello pretendemos que se adquiera una mejor visión global tanto del lenguaje VB.NET como de su herramienta de trabajo: Visual Studio .NET.

Un programa más operativo

En este tema vamos a escribir una aplicación algo más completa, que consistirá en un formulario en el que introduciremos el nombre de un fichero y un pequeño texto, que seguidamente grabaremos en nuestro equipo. Así que, una vez esbozado el objetivo a conseguir... manos a la obra.

Diseño del formulario

Después de iniciar VS.NET, crearemos un nuevo proyecto al que daremos el nombre de EscritorTexto (para acceder a EscritorTexto, el proyecto de este ejemplo, hacer clic aquí). En el formulario del proyecto, Form1, añadiremos los controles que permitirán al usuario escribir un texto, grabar dicho

texto en un fichero, etc. En concreto añadiremos dos controles Label, dos TextBox y dos Button, cuya ubicación en la ventana del Cuadro de herramientas mostramos en la Figura 61.

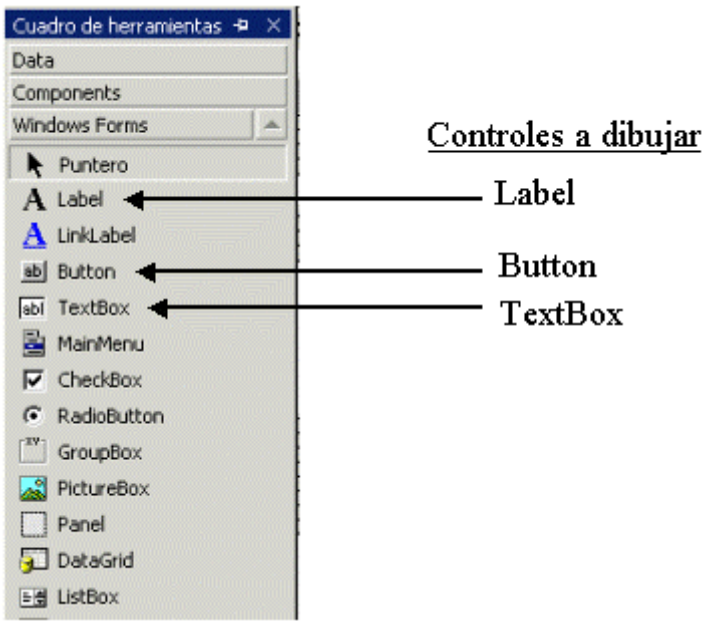


Figura 61. Controles que debemos dibujar en el formulario del ejemplo.

La forma de dibujar un control en un formulario ya ha sido explicada anteriormente, por lo que directamente mostramos en la Figura 62, el formulario resultante con los controles ya insertados, en donde indicamos el tipo de control y el nombre que hemos asignado a cada control en su propiedad Name.

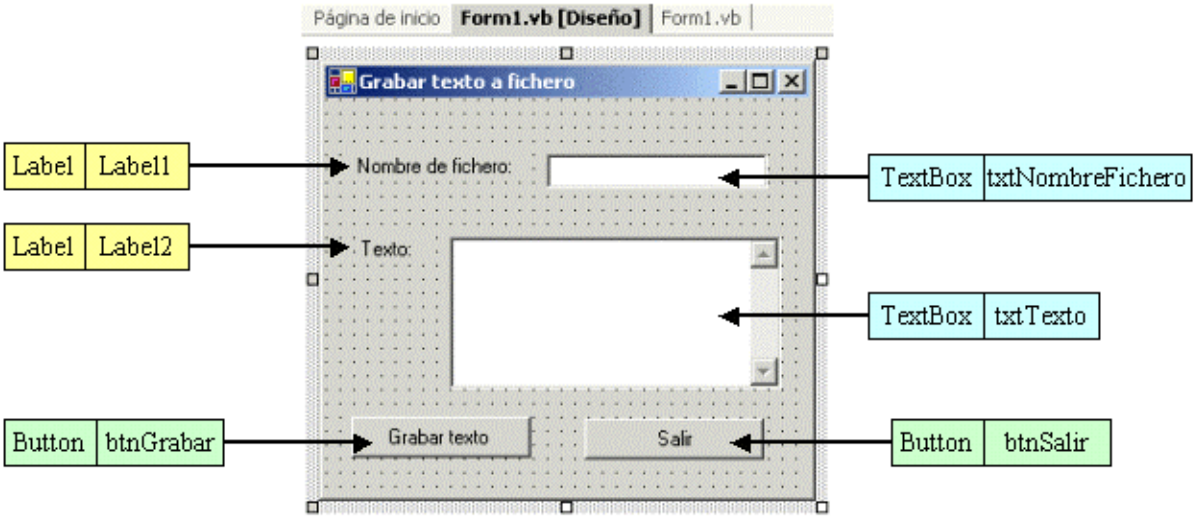


Figura 62. Formulario para la grabación de un texto en un fichero.

A continuación detallamos brevemente la funcionalidad de cada uno de los controles de este formulario:

- **Label1, Label2**. Muestran un simple literal que indica al usuario lo que debe introducir en los controles de texto.

- **txtNombreFichero**. Contiene el nombre que daremos al fichero en el que grabaremos el texto.

- **txtTexto**. Contiene el texto que se va a guardar en un fichero. La diferencia de este control, con el otro control de tipo TextBox del formulario, reside en que permite escribir varias líneas de texto, gracias a que hemos asignado a su propiedad Multiline el valor True. La propiedad Multiline por defecto contiene False, lo que indica que un TextBox sólo permite introducir el texto en una única línea.

- **btnGrabar**. Al pulsar este botón, se tomará el texto del control txtTexto y se grabará en un fichero con el nombre que contenga el control txtNombreFichero. Veremos como escribir el código para un control más adelante.

- **btnSalir**. Al pulsar este botón, se finalizará la ejecución del programa, de igual forma que si pulsáramos el botón de cierre del formulario o [ALT+F4].

Observe el lector que al asignar el nombre de algunos controles, hemos utilizado un prefijo. Así, para un TextBox utilizamos el prefijo txt (txtNombreControl); para un Button, btn (btnNombreControl), etc.

Esta técnica, denominada *convenciones de notación*, consiste en una serie de normas no obligatorias, utilizadas a la hora de escribir el código, y que son pactadas generalmente en equipos de trabajo, de manera que cuando un programador debe tomar parte de un proyecto que ha estado desarrollando otro programador, la interpretación del código se facilita, y el desarrollo del proyecto en este sentido, se dinamiza.

El programador independiente puede igualmente utilizar este tipo de convenciones, ya que gran parte del código fuente que circula en artículos, demos, aplicaciones shareware, etc., emplean una serie de convenciones genéricas de notación, por lo que si necesita en algún momento compartir su código, la legibilidad del mismo se facilita.

La Tabla 4 muestra una serie de convenciones para la codificación de los nombres de controles, que proponemos como ejemplo, para que el lector utilice estas o alguna similar.

Control	Prefijo
Label	lbl
Button	btn
TextBox	txt
CheckBox	chk
ListBox	lst
ComboBox	cbo
RadioButton	rbt

MainMenu	Mnu
GroupBox	grp
MainMenu ContextMenu	mnu
FontDialog ColorDialog y demás controles de caja diálogo	dlg

Tabla 4. Convenciones de notación para controles de formulario.

Para el formulario podemos utilizar el prefijo frm.

Controles y eventos

Windows es un sistema basado en eventos. Esto quiere decir que cualquier cosa que ocurra dentro de un formulario, bien sobre el propio formulario, o a cualquiera de los controles que lo componen, se detecta a través de un suceso o evento: pasar el ratón sobre un control, hacer clic, escribir en un control de texto, cerrar el formulario, redimensionarlo, etc. Cualquier acción de este tipo provoca el evento correspondiente.

En nuestro ejemplo actual, si ejecutamos el programa y hacemos clic sobre alguno de los botones del formulario, no ocurrirá nada en absoluto. ¿Quiere esto decir que los botones no tienen eventos?, la respuesta es no, los botones sí tienen eventos, así como los demás controles, y aunque no lo percibamos, se están produciendo constantemente, lo que sucede, es que una vez que dibujamos un control en un formulario, dicho control inicialmente no está programado para responder a los eventos, por lo que debemos escribir el código para el evento correspondiente, en respuesta a la acción del usuario.

Siguiendo con los botones del formulario, vamos a elegir el más sencillo de codificar, btnSalir. Para escribir el código del evento correspondiente a la pulsación de este control, la forma mas rápida es haciendo doble clic sobre él en el diseñador del formulario, lo que abrirá el editor de código y nos situará en un procedimiento vacío que mostramos en el Código fuente 23.

```
Private Sub btnSalir_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnSalir.Click

End Sub
```

Código fuente 23. Procedimiento para el evento Click de un Button.

Se trata de un procedimiento cuyo nombre, btnSalir_Click, compuesto del nombre del control y del evento, lo proporciona automáticamente el IDE. Recibe dos parámetros: sender y e, con información adicional proporcionada por el entorno. Pero lo verdaderamente importante está al final de la declaración: “Handles btnSalir.Click”. La palabra clave Handles, seguida del nombre de un control, un punto y el nombre de un evento de ese control, le indica al entorno de .NET Framework que debe ejecutar este procedimiento cuando se produzca el evento para ese control. No realizaremos en este

momento una explicación más detallada puesto que trataremos este aspecto con más profundidad en temas posteriores del texto.

Este procedimiento será ejecutado cada vez que pulsemos el control btnSalir, por lo que si en tal situación, queremos cerrar el formulario, sólo será necesario incluir la línea de código mostrada en el Código fuente 24.

```
Private Sub btnSalir_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnSalir.Click

    ' cerrar el formulario
    Me.Close()

End Sub
```

Código fuente 24. Código del evento Click para cerrar un formulario.

El método Close( ) del formulario, produce su cierre, y por ende, la finalización del programa. El efecto es el mismo que si pulsamos el botón de cierre del formulario o la combinación [ALT+F4]. La palabra clave Me indica que estamos haciendo usando una propiedad o método del formulario desde el interior de la clase del propio formulario; esta palabra clave será explicada con más detalle en el tema dedicado a la programación orientada a objeto.

Ahora nos formularemos una interesante cuestión: -¿Y por qué sabía VS.NET cuál evento quería codificar y me lo ha mostrado directamente?-. Pues sencillamente, VS.NET no lo sabía, lo que sucede es que cuando hacemos doble clic sobre un control en el diseñador del formulario, se abre el editor de código y nos sitúa en el evento por defecto del control. Y sucede que el evento por defecto de un control Button es Click( ).

Otro modo de escribir el código de un evento

En el apartado anterior hemos mostrado el medio más fácil y rápido para escribir el código de un evento. Sin embargo, existe otra forma de realizar esta operación, ya que en muchas ocasiones tendremos que codificar un evento distinto al evento por defecto de un control o formulario, y en ese caso deberemos seleccionarlo manualmente. Veamos los pasos a realizar cuando se plantee dicha situación.

En el formulario tenemos todavía pendiente de codificar el botón btnGrabar. Lo que necesitamos que haga este botón, es grabar el texto escrito en el control txtTexto, en un fichero al que daremos como nombre el valor que se haya escrito en el control txtNombreFichero.

Primeramente debemos acceder al editor de código del formulario, y abrir la lista desplegable *Nombre de clase*, situada en la parte superior izquierda. Ver Figura 63.

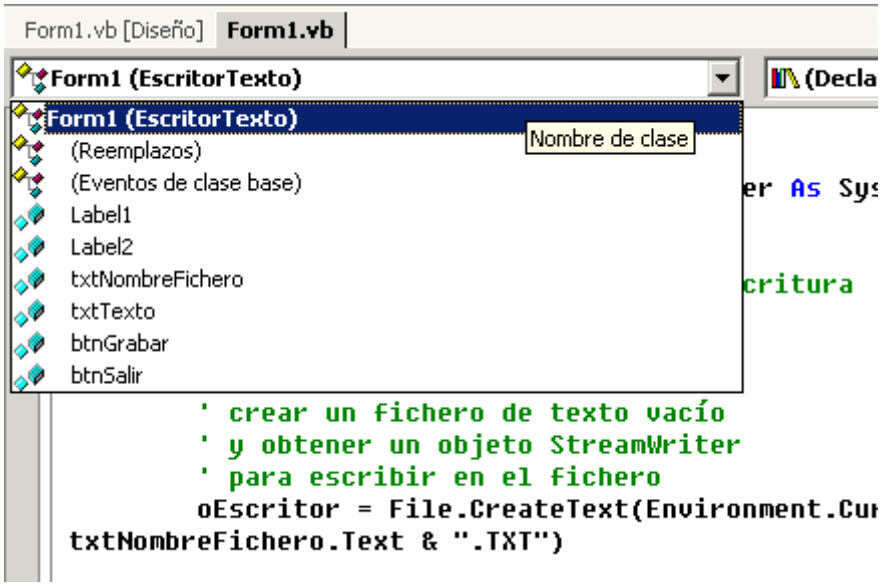


Figura 63. Editor de código mostrando la lista de nombres de clases.

Esta lista muestra el nombre del formulario y los controles que hemos incluido en él, o dicho de otro modo, todos los objetos del formulario, teniendo en cuenta que el propio formulario también es un objeto.

Seleccionaremos el control para el que vamos a codificar un evento: btnGrabar, y a continuación abriremos la lista desplegable *Nombre de método*, situada esta vez en la parte superior derecha del editor de código. Ver Figura 64.

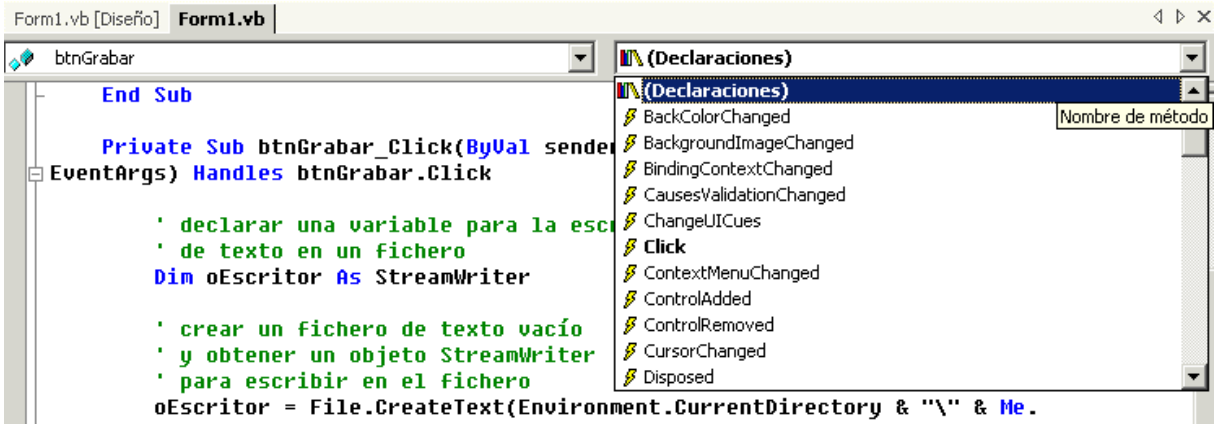


Figura 64. Editor de código mostrando la lista de métodos.

Esta lista muestra todos los eventos que podemos codificar para el control que hemos seleccionado en el formulario. Al seleccionar el evento Click( ), se mostrará la declaración de dicho procedimiento de evento vacía para escribir las acciones descritas anteriormente.

Grabando texto en un fichero

Una vez seleccionado el procedimiento de evento para el botón btnSalir, escribiremos las líneas mostradas en el Código fuente 25 que comentaremos detalladamente.

```
Private Sub btnGrabar_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnGrabar.Click

    ' declarar una variable para la escritura
    ' de texto en un fichero
    Dim oEscritor As StreamWriter

    ' crear un fichero de texto vacío
    ' y obtener un objeto StreamWriter
    ' para escribir en el fichero
    oEscritor = File.CreateText(Environment.CurrentDirectory & "\" &
Me.txtNombreFichero.Text & ".TXT")

    ' escribir en el fichero el contenido del
    ' control TextBox
    oEscritor.Write(Me.txtTexto.Text)

    ' cerrar el objeto, lo que cierra también el
    ' fichero, y eliminar el objeto
    oEscritor.Close()
    oEscritor = Nothing

End Sub
```

Código fuente 25. Evento Click() del botón btnGrabar.

En primer lugar declaramos la variable oEscritor de tipo StreamWriter. Este tipo de objetos nos permitirán realizar la escritura de un flujo (stream) de caracteres sobre un fichero del disco.

A continuación, vamos a crear un fichero de texto en nuestro disco duro, en la misma ruta en la que se está ejecutando la aplicación. Esto lo conseguimos llamando al método compartido CreateText( ), del objeto File (observe el lector que al ser un método compartido no necesitamos instanciar un objeto de la clase File y pasarlo a una variable).

El método CreateText( ) recibe como parámetro una cadena de caracteres con la ruta y el nombre del fichero a crear. Para componer esta ruta utilizamos por un lado el objeto Environment y su propiedad compartida CurrentDirectory que devuelve la ruta del directorio actual en donde se está ejecutando la aplicación, y por otro lado la propiedad Text del control txtNombreFichero, que contiene el valor que el usuario haya escrito en dicho TextBox. Estos dos elementos los unimos, formando una sola cadena, mediante el operador de concatenación de VB: &.

La llamada a CreateText( ) devuelve un objeto de tipo StreamWriter, que depositamos en la variable oEscritor, con lo que ya tenemos en la variable un objeto para escribir texto.

El siguiente paso consiste en llamar al método Write( ) de oEscritor y pasarle como parámetro la propiedad Text del control txtTexto, que contiene el texto escrito por el usuario. Este texto es grabado en el fichero.

Para finalizar, cerramos el objeto oEscritor llamando a su método Close( ) y le asignamos la palabra clave Nothing para liberar los recursos del sistema que pudiera estar utilizando.

Observe el lector, como el formato de manipulación de objetos se basa en la variable que contiene el objeto o el propio nombre del objeto (si es compartido), un punto y el nombre del método a llamar o propiedad de la que recuperamos o a la que asignamos un valor, tal y como muestra de forma esquemática el Código fuente 26.

```
' objetos instanciados
'---------------------
oVar.Propiedad = valor   ' asignar valor a propiedad
variable = oVar.Propiedad   ' recuperar valor de propiedad
oVar.Metodo([ListaParametros])  ' llamar a método
variable = oVar.Metodo([ListaParametros])  ' llamar a método y recuperar valor

' objetos compartidos
'---------------------
Objeto.Propiedad = valor   ' asignar valor a propiedad
variable = Objeto.Propiedad   ' recuperar valor de propiedad
Objeto.Metodo([ListaParametros])   ' llamar a método
Variable = Objeto.Metodo([ListaParametros]) ' llamar a método y recuperar valor
```

Código fuente 26. Modos de manipulación propiedades y métodos de objetos.

Ya sólo queda ejecutar el programa, escribir valores en los controles, y generar el fichero de texto para comprobar que todo funciona correctamente, con ello habremos conseguido crear un programa que tenga una aplicación algo más práctica que el típico hola mundo.

Una puntualización sobre los eventos

En un apartado anterior hemos explicado que si pulsábamos sobre un nuevo control Button, este no realizaba ninguna acción porque no habíamos codificado todavía su evento correspondiente. Esto puede dar lugar a confusiones, si por ejemplo, al ejecutar un programa, en el caso de un formulario, pulsamos su botón de cierre, lo redimensionamos, etc.; o en el caso de una lista desplegable, pulsamos sobre el botón que abre la lista de valores.

¿Por qué el formulario y controles responden a esos eventos si el programador no ha escrito código para ellos?. Bien, en este caso estamos ante unos eventos que el programador, en principio, no necesita codificar, ya que forman parte intrínseca del sistema operativo, siendo el propio sistema el que se ocupa de que el formulario o control se comporten de acuerdo con tales eventos.

8

El entorno de desarrollo integrado (IDE), de Visual Studio .NET

El IDE, un elemento a veces menospreciado

El disponer en la actualidad de un entorno de desarrollo (Integrated Development Environment o IDE) para una herramienta de programación, puede parecer algo natural o incluso elemental para el propio lenguaje. Ello hace que en ocasiones no se conceda la importancia que realmente tiene a este aspecto del desarrollo.

Las cosas no han sido siempre tan fáciles en este sentido, ya que en tiempos no muy lejanos, los programadores debían de realizar de una forma digamos *artesanal*, todos los pasos en la creación de una aplicación.

Cuando utilizamos el término artesanal, nos referimos a que el programador debía escribir el código fuente en un editor y diseñar el interfaz con un programa generador de pantallas. Después ejecutaba otro programa que contenía el compilador, sobre el código fuente, para obtener los módulos compilados. Finalmente, debía de enlazar los módulos compilados con las librerías del lenguaje y terceras librerías de utilidades si era necesario, para obtener el ejecutable final de la aplicación. Todo se hacía a base de pasos independientes.

Tal dispersión de elementos a la hora de desarrollar resultaba incómoda en muchas ocasiones, por lo que los fabricantes de lenguajes de programación, comenzaron a incluir en sus productos, además del propio compilador, enlazador y librerías, una aplicación que permitía al programador realizar todas las fases del desarrollo: los editores de código, diseñadores visuales, compilador, etc., estaban incluidos

en el mismo entorno a modo de escritorio de trabajo o taller de programación, se trataba de los primeros IDE.

El largo camino hacia la convergencia

En lo que respecta a los lenguajes de Microsoft, los programadores disponemos desde hace ya tiempo del IDE de Visual Studio.

Los diseñadores del entorno de programación de Microsoft, sobre todo desde su versión 5 han tenido un objetivo principal: hacer que el IDE de cada uno de los lenguajes sea lo más similar posible al resto, de forma que el desarrollo con varios lenguajes no suponga un cambio traumático en el entorno de trabajo.

Esto quiere decir que, por ejemplo, un programador que debe desarrollar aplicaciones tanto en Visual Basic como en Visual C++, cada vez que abra el entorno de trabajo de alguna de estas herramientas, va a encontrar, salvo las particularidades impuestas por el lenguaje, un IDE casi igual, lo que evita un entrenamiento por separado para cada lenguaje y una mayor productividad, al acceder a los aspectos comunes de igual manera.

A pesar de todas las similitudes, y hasta la versión 6, cada lenguaje seguía teniendo su propio IDE.

Visual Studio .NET, el primer paso de la total integración

Con la llegada de la tecnología .NET, el panorama ha cambiado sustancialmente, ya que al estar todos los lenguajes bajo el abrigo de un entorno de ejecución común, se ha podido desarrollar también un IDE común.

Ya no debemos elegir en primer lugar el lenguaje y abrir su IDE particular. Todo lo contrario, ahora debemos iniciar el IDE de Visual Studio .NET y después, elegir el lenguaje con el que vamos a trabajar. Esto materializa la idea de disponer de un IDE único para diversos de lenguajes. Este concepto es además extensible, ya que al ser .NET Framework una plataforma multilenguaje, los lenguajes desarrollados por terceros fabricantes también podrán engrosar la lista de los disponibles a través del IDE.

La descripción del IDE se abordará en esta obra de dos formas: una de ellas será en el tema actual, dentro del que se explicarán los aspectos generales. La otra forma será a lo largo de los distintos temas del texto que así lo requieran, ya que ciertos aspectos específicos del IDE es recomendable describirlos en el tema con el que guardan una mayor relación.

La página de inicio

Nada más iniciar VS.NET, se muestra la página de inicio del IDE. Ver Figura 65.

Desde esta página podemos realizar una primera configuración del entorno, ya que si hacemos clic en el vínculo *Mi perfil*, situado en la columna izquierda, accederemos a una pantalla en la que podemos establecer una configuración adaptada al lenguaje con el que vamos a programar. Ver Figura 66.

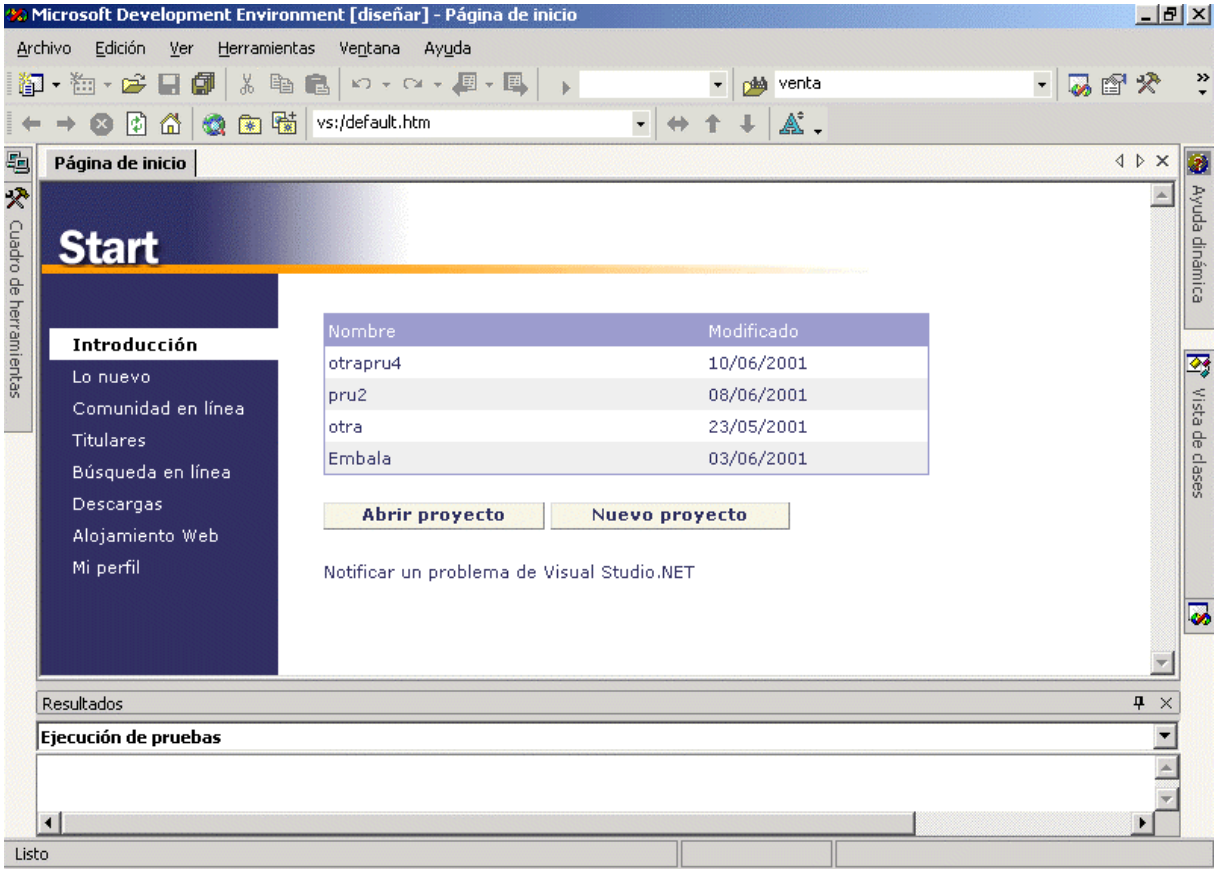


Figura 65. Página de inicio de VS.NET.

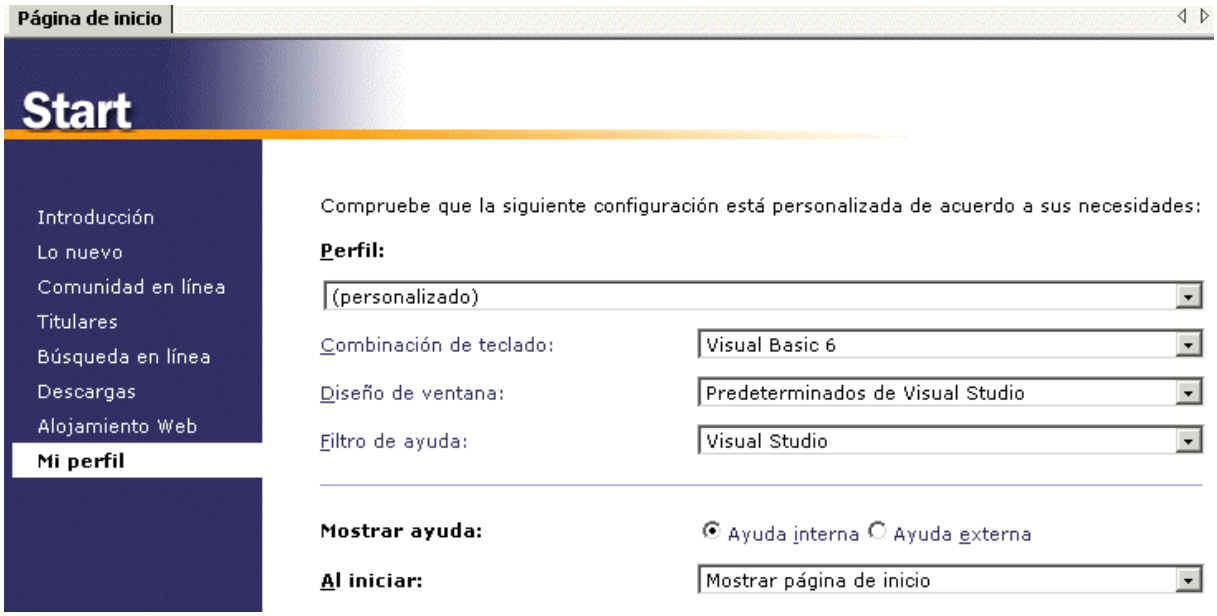


Figura 66. Estableciendo un perfil de programador.

Como puede comprobar el lector, podemos configurar el perfil general para adaptar a nuestra comodidad la totalidad del IDE, o bien hacerlo sólo sobre ciertos elementos como el teclado, diseño de ventana, etc.

Establezca el lector el perfil que prefiera, por el momento recomendamos elegir sólo la combinación de teclado adaptada a un perfil de programador de Visual Basic 6. El resto de elementos los dejaremos como estaban por defecto, ya que si adaptamos la totalidad del IDE al perfil de VB, se expandirán muchas de las ventanas ocultas adicionales, dejando poco espacio en la ventana principal de trabajo.

Configurado el perfil del programador, haremos clic en el vínculo Introducción, de la columna izquierda, para volver al punto inicial, en el que crearemos un nuevo proyecto de VB.NET, de la forma explicada en el tema *La primera aplicación*, que nos servirá para hacer las pruebas sobre los diferentes aspectos del IDE.

Si por cualquier motivo, cerramos la página de inicio, podemos volver a visualizarla utilizando alguna de las siguientes formas:

- Opción de menú *Ayuda + Mostrar página de inicio* o bien con *Ver + Explorador Web + Inicio*.
- Opción de menú *Ver + Explorador Web + Inicio.*
- Tecleando la dirección *vs:/default.htm*, en el campo *Dirección URL*, de la barra de herramientas Web.

Principales elementos en nuestro entorno de trabajo

Una vez abierto un proyecto en el IDE, los elementos básicos para nuestra tarea habitual de desarrollo se muestran en la Figura 67.

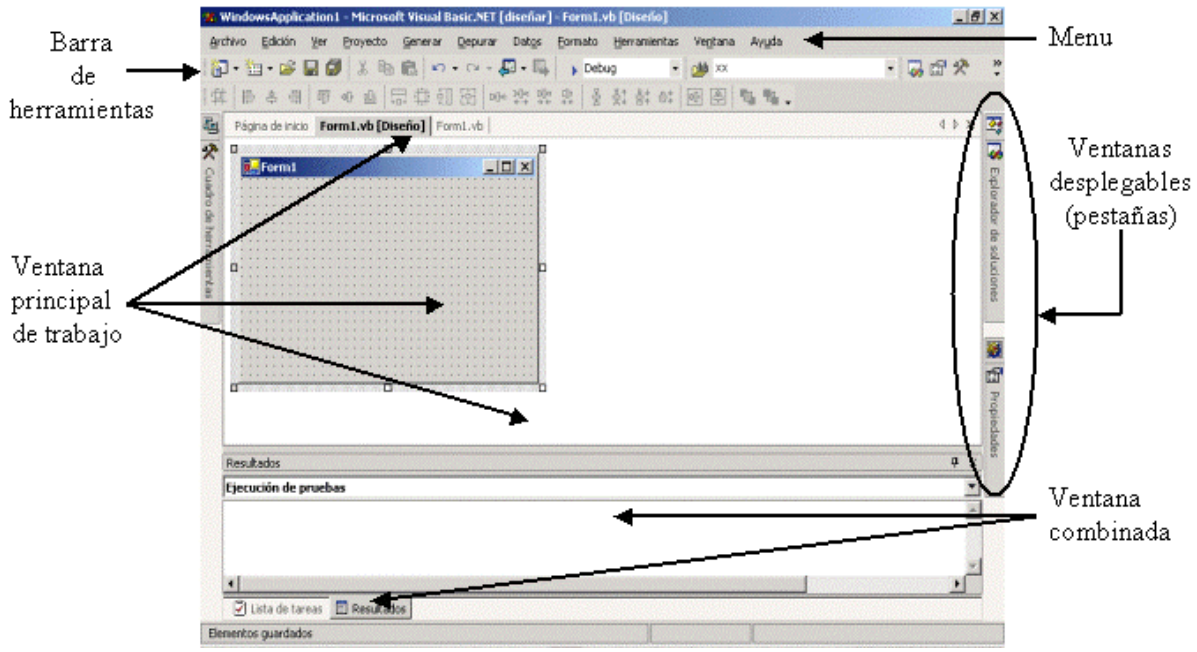


Figura 67. Elementos principales del IDE.

En los siguientes apartados realizaremos una descripción más detallada de algunos de estos componentes del IDE y el modo de trabajar con ellos.

Ventana principal de trabajo

De forma predominante y ocupando la mayor parte del IDE, encontramos la ventana o zona principal de trabajo. Esta ventana principal contiene todos los editores de código y diseñadores que vayamos abriendo, organizados en base a unas fichas o pestañas, que nos permiten trasladarnos de uno a otro cómodamente.

Para comprobarlo, vamos a añadir un nuevo formulario al proyecto mediante la opción de menú *Proyecto + Agregar formulario de Windows*, y un módulo de código con la opción *Proyecto + Agregar módulo*. En ambos casos, dejaremos a estos elementos los nombres que asigna por defecto el IDE. Si además, en los dos diseñadores de formulario que deberemos tener actualmente, seleccionamos la opción *Ver + Código*, se añadirán los correspondientes editores de código a la ventana, que mostrará un aspecto similar al de la Figura 68.

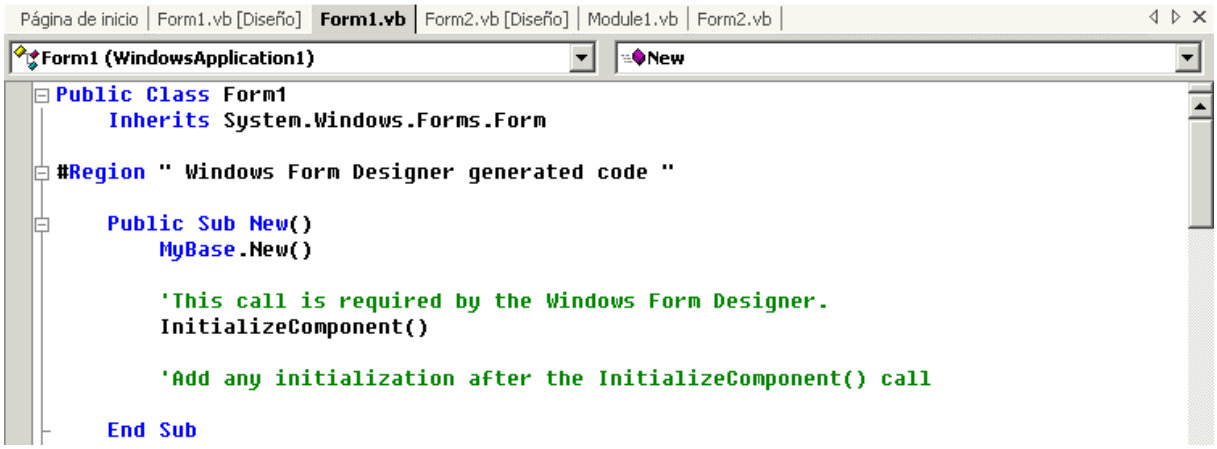


Figura 68. Ventana principal de trabajo mostrando varias fichas.

Podemos cambiar de diseñador con un simple clic sobre su ficha correspondiente o la combinación de teclas [CTRL.+TAB]. Cuando la ventana se llene totalmente de fichas, podemos desplazarnos entre las mismas mediante los dos iconos de la parte superior derecha que muestra unas flechas de dirección. Si queremos cerrar alguna de las fichas, podemos hacerlo igualmente pulsando el icono de cierre de esta ventana o la combinación [CTRL.+F4].

Para cambiar la posición de las fichas, debemos hacer clic sobre la ficha que queramos cambiar y arrastrar hacia una nueva posición.

La organización en fichas, supone un importante cambio en el modo de trabajo respecto a VB6, que aporta una mayor comodidad a la hora de tener abiertos simultáneamente diversos editores y diseñadores. Sin embargo, si el programador se siente más confortable con la antigua organización basada en ventanas, puede cambiar a dicha configuración seleccionando la opción de menú *Herramientas + Opciones*, que mostrará la ventana de opciones de configuración del IDE. Ver Figura 69.

En el caso de que no estemos posicionados inicialmente, debemos seleccionar en la parte izquierda de esta ventana, la carpeta Entorno y su apartado General. A continuación pulsaremos sobre la opción *Entorno MDI* y pulsar Aceptar, ver Figura 70. Debemos tener en cuenta, que los cambios no se reflejarán hasta la próxima vez que iniciemos VS.NET. Si queremos volver al modo inicial, tendremos que pulsar sobre *Organización por fichas*.

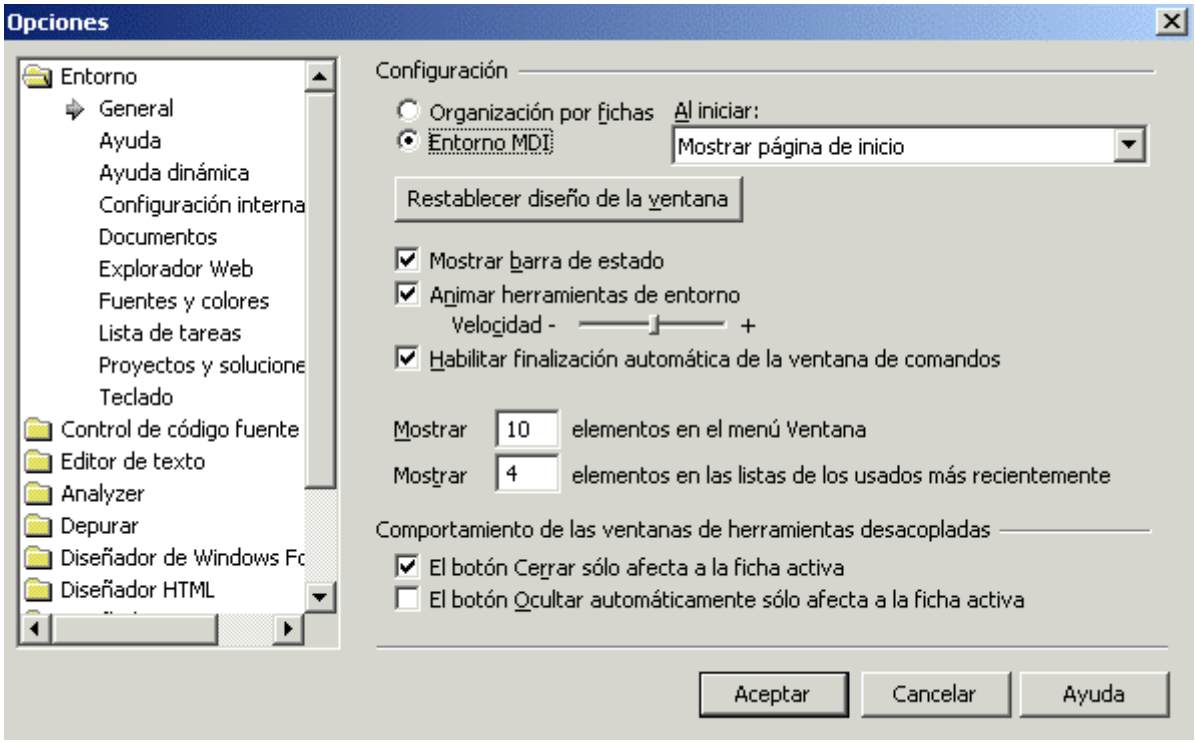


Figura 69. Opciones de configuración del IDE.

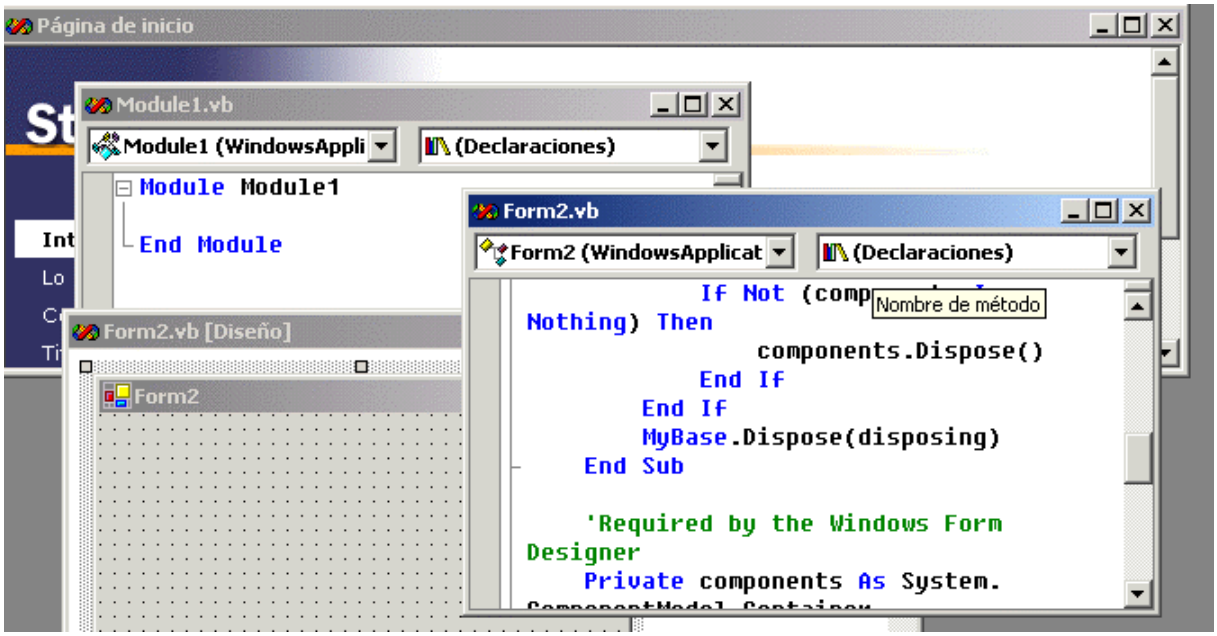


Figura 70. Organización del IDE en ventanas.

Es posible crear una nueva ventana para albergar fichas, usando la opción de menú *Ventana + Nuevo grupo de fichas horizontal*, o la opción *Ventana + Nuevo grupo de fichas vertical,* a la que podremos mover fichas desde la ventana original con sólo arrastrar y soltar. Ver Figura 71.

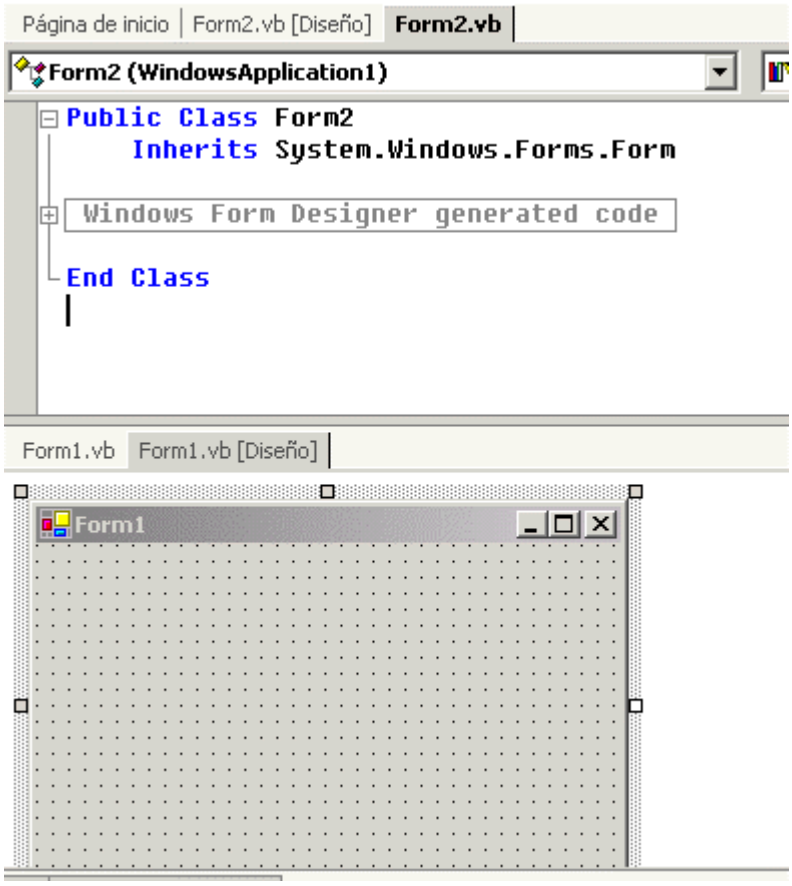


Figura 71. Fichas organizadas en grupos diferentes.

Manejo de ventanas adicionales del IDE

Aparte de la ventana principal de trabajo, el IDE dispone de una serie de ventanas suplementarias, que sirven de apoyo a la tarea del desarrollo.

En este apartado trataremos el modo en que podemos manipular y organizar dichas ventanas, no de su contenido en sí, dejando este aspecto para próximos apartados.

Las ventanas adicionales disponen de un estado de visualización que por defecto es *Ocultar automáticamente*, y se encuentran situadas en los laterales del IDE, mostrándose una ficha indicativa de la misma. Al situar el cursor del ratón sobre la ficha, la ventana se expande. Veamos un ejemplo en la Figura 72.

Si hacemos clic derecho sobre el lateral, aparecerá un menú contextual con el nombre de todas las fichas dispuestas en el lateral.

Una vez que expandimos una de estas ventanas, y hacemos clic en alguno de sus elementos, podemos trabajar con ella normalmente. Cuando volvamos a hacer clic sobre la ventana principal del IDE, se ocultará automáticamente la que habíamos expandido. Esto supone una gran comodidad, ya que nos ahorra tener que cerrar explícitamente la ventana suplementaria cuando no la necesitemos, y además brinda un mayor espacio de trabajo sobre la zona principal del IDE.

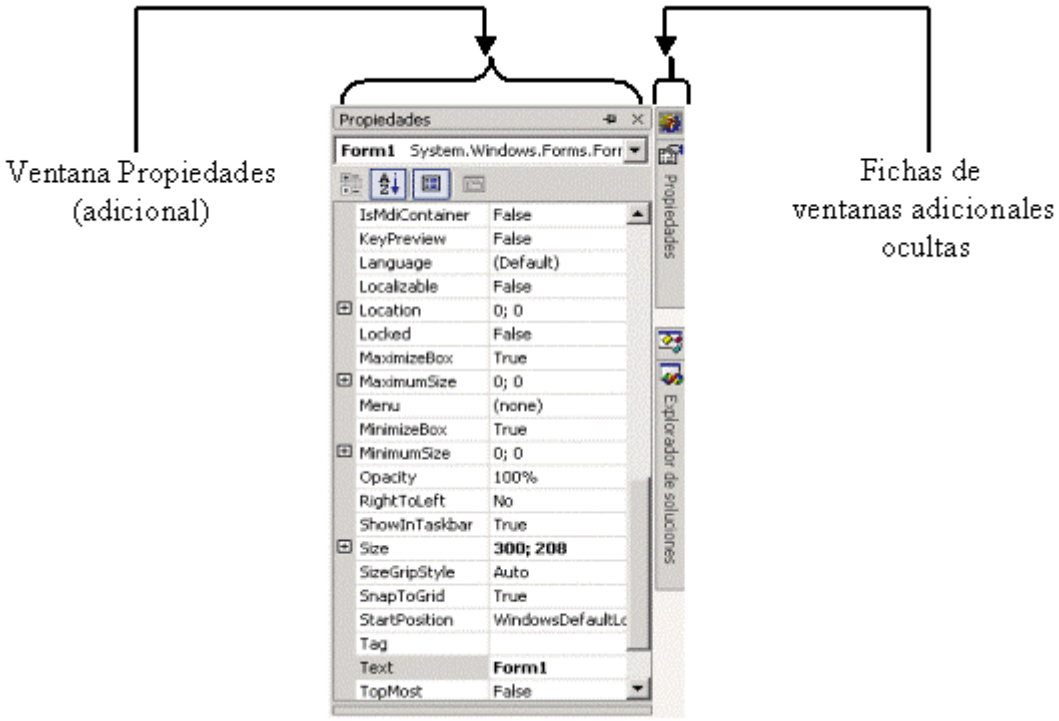


Figura 72. Ventana adicional expandida y fichas de ventanas ocultas.

Una ventana del IDE dispone de cuatro estados de visualización: Acoplable, Ocultar, Flotante y Ocultar automáticamente. Para verificar el estado de una ventana, debemos hacer clic derecho sobre su barra de título, que nos mostrará un menú contextual con el estado actualmente activo. Ver Figura 73.

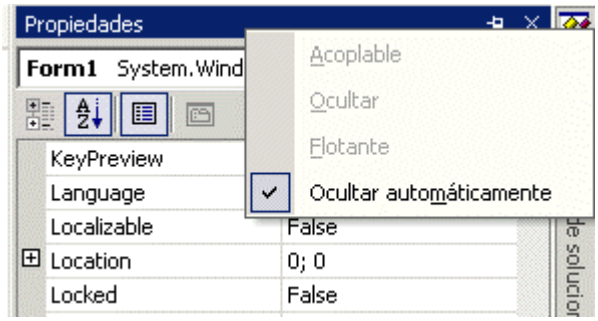


Figura 73. Menú de estado de una ventana adicional del IDE.

Habrá ocasiones en que necesitaremos tener permanentemente visible una ventana adicional. Para ello, y una vez que tengamos visible la ventana, debemos cambiar su estado a Acoplable, haciendo clic sobre el icono en forma de chincheta que aparece en la barra de título. Esto hará que cambie dicho icono de forma indicando el nuevo estado. Ver Figura 74 y Figura 75.

Figura 74. Ocultar automáticamente una ventana.

Figura 75. Ventana en estado acoplable.

Una ventana acoplable o fija no se oculta cuando pasamos a cualquier otra ventana del IDE.

También puede ser útil en algunas situaciones, permitir que una ventana pueda moverse libremente por todo el área del IDE, para lo que en tal caso, haremos clic derecho sobre su título y elegiremos la opción Flotante, lo que dejará a dicha ventana libre para ser situada en cualquier lugar del IDE, sin la obligación de estar ajustada a ningún borde del entorno de desarrollo. Ver Figura 76.

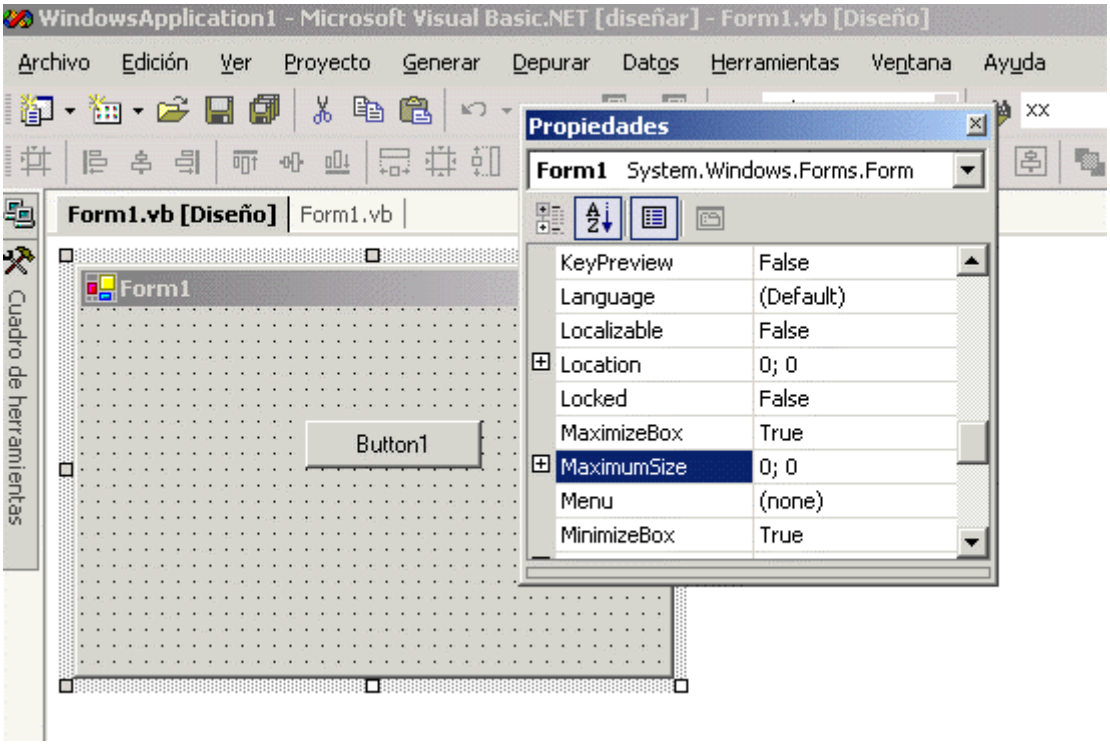


Figura 76. Ventana de propiedades en modo flotante.

Para ocultar una de estas ventanas, haremos clic en su icono de cierre o en su menú contextual de estado, opción Ocultar.

El acople de una ventana no es obligatorio realizarlo siempre en los laterales, también podemos ajustar una de estas ventanas a la parte inferior o superior del IDE. Para ello hemos de arrastrar la ventana hacia uno de los bordes del IDE hasta el momento en que se muestre un rectángulo que representa la guía o modo en cómo se va a acoplar dicha ventana. Ver Figura 77.

Al soltar en el momento en que aparece la guía de acople, la ventana quedará fijada en concordancia. Ver Figura 78.

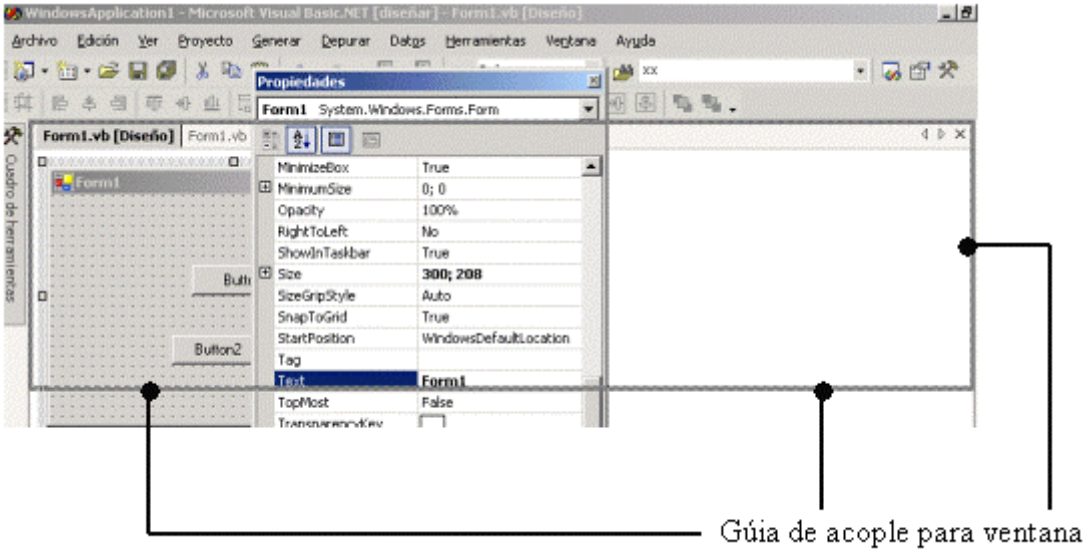


Figura 77. Guía de acople al desplazar una ventana por el IDE.

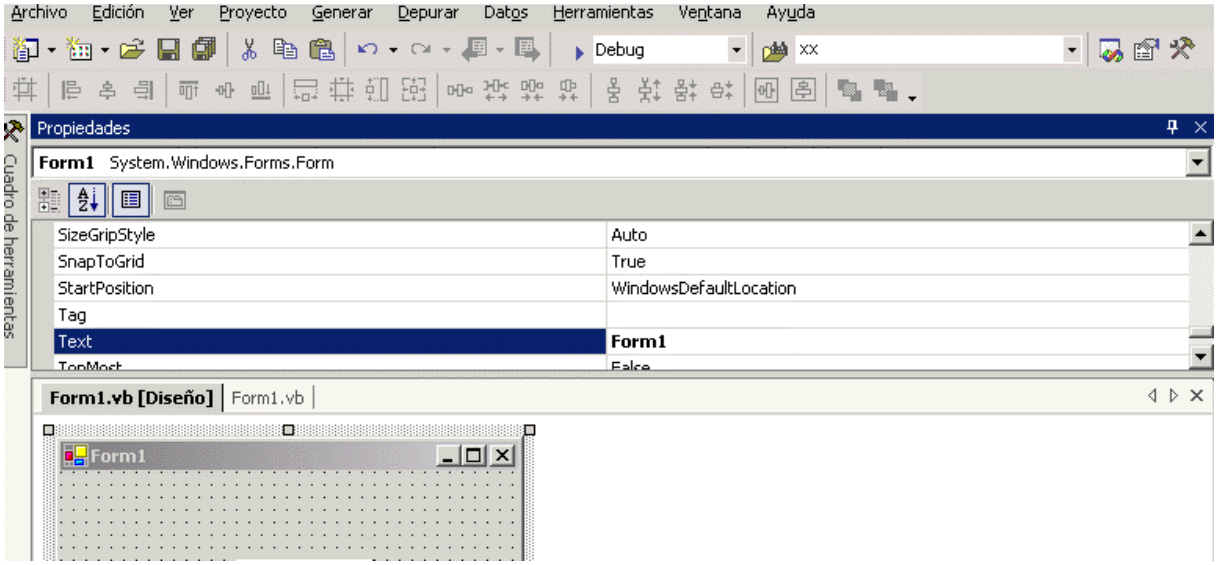


Figura 78. Ventana de propiedades acoplada a la parte superior del IDE.

Podemos conseguir un acople entre múltiples ventanas, arrastrando una de ellas hacia la zona de trabajo de otra y soltando en el momento en que aparezca la guía de acople. La Figura 79 muestra tres ventanas con diferentes acoples realizados entre ellas.

Finalmente, es posible también acoplar múltiples ventanas pero organizarlas mediante fichas, de modo que sólo se visualice una ventana a la vez haciendo clic en la ficha con el título de la ventana. Al acoplar una ventana para que se muestre de esta forma, debemos situarla sobre el título de otra, apareciendo la guía de acople como muestra la Figura 80.

En la Figura 81 tenemos múltiples ventanas acopladas organizadas de esta manera. Podemos cambiar entre ellas haciendo clic en la ficha mostrada en la parte inferior.

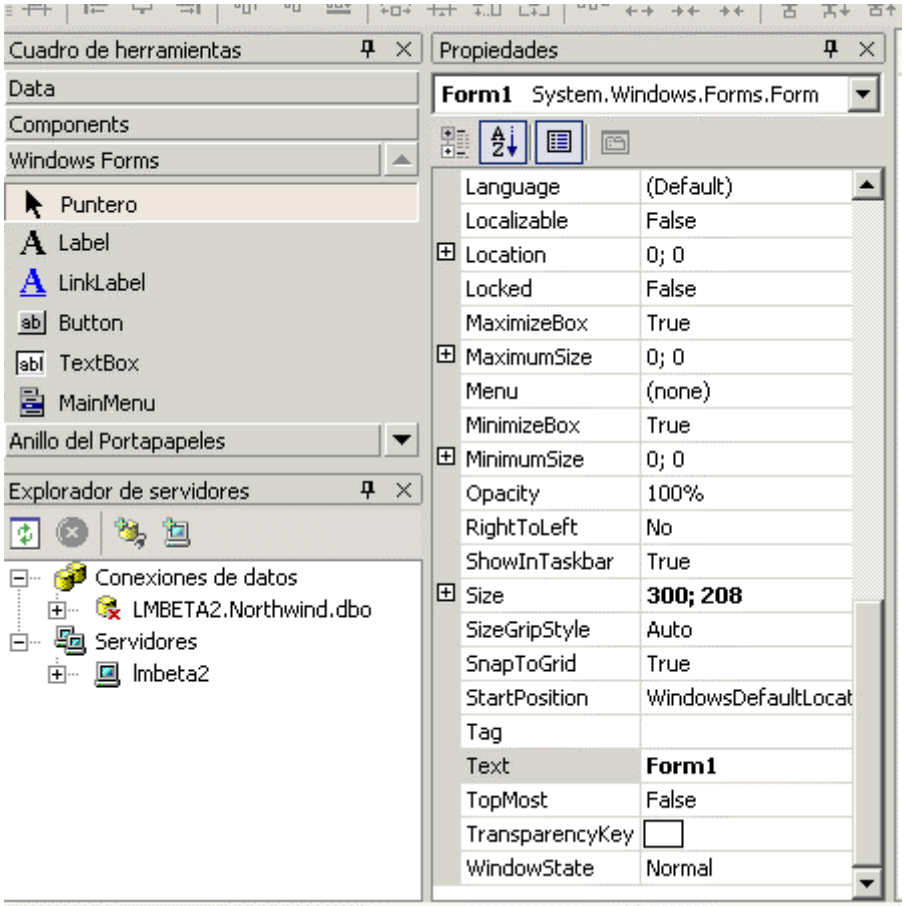


Figura 79. Ventanas de propiedades, herramientas y servidores, con diferentes tipos de acople entre ellas.

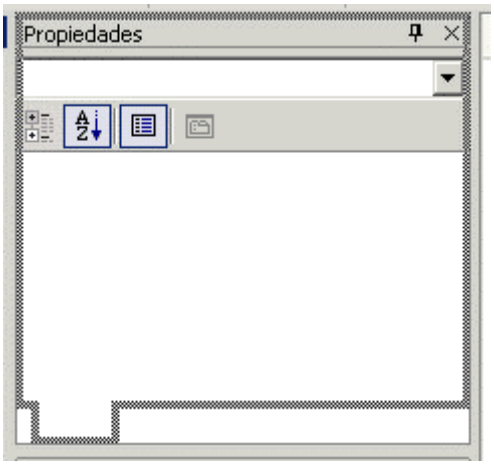


Figura 80. Acople de múltiples ventanas en modo ficha.

Para separar cualquiera de estas ventanas, basta con hacer clic sobre su ficha y arrastrar hacia el exterior de la ventana contenedora.

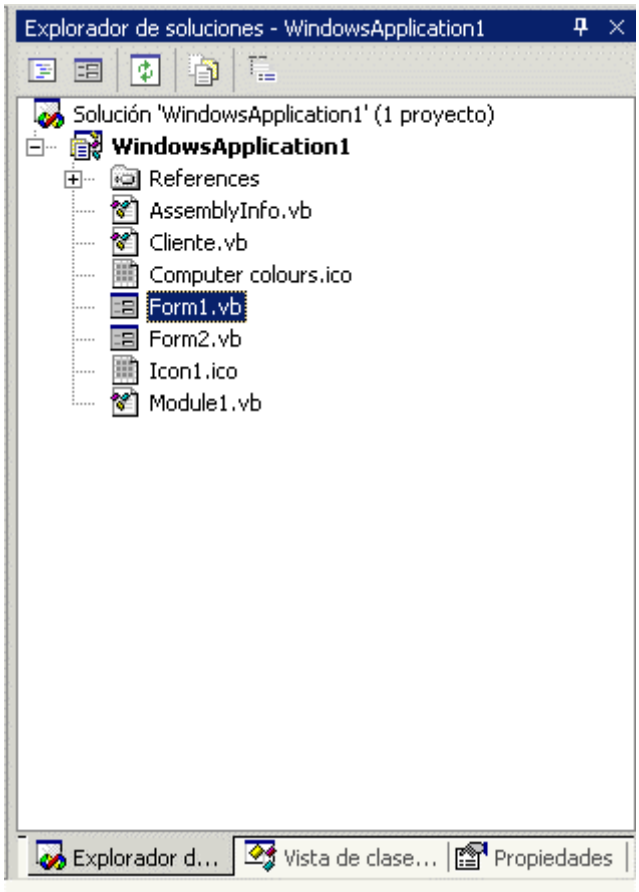


Figura 81. Múltiples ventanas adicionales acopladas en fichas.

El Explorador de soluciones

Al desarrollar una aplicación en VB.NET, los elementos que contiene: formularios, módulos, clases, recursos, referencias, etc., se organizan dentro de un proyecto.

También es posible tener varios proyectos abiertos simultáneamente en la misma sesión de trabajo del IDE. Dichos proyectos se organizan dentro de lo que en VS.NET se denomina una solución.

Una solución puede contener proyectos desarrollados en los diferentes lenguajes de la plataforma .NET, y el medio más cómodo para manejarlos es a través de la ventana *Explorador de soluciones*. La Figura 82 muestra el aspecto típico de esta ventana con una solución que contiene un proyecto, en el que a su vez hay contenido un formulario.

Los modos de abrir a esta ventana son los siguientes:

- Expandir la ficha lateral, si existe para esta ventana.
- Opción de menú del IDE *Ver + Explorador de soluciones*.
- [CTRL.+R].
- Pulsar el botón de la barra de herramientas para esta opción. Ver Figura 83.

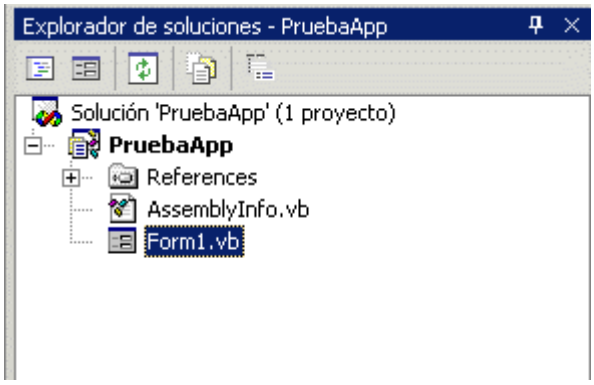


Figura 82. Explorador de soluciones del IDE.

Figura 83. Botón del Explorador de soluciones en la barra de herramientas del IDE.

La carpeta References contiene las referencias que están establecidas dentro del proyecto, hacia los diferentes espacios de nombre que pueden ser necesarios a la hora de escribir el código. Al expandir esta carpeta tendremos referenciados entre otros: System, System.Windows.Forms, etc.

Al crear un nuevo proyecto desde VS.NET, dichas referencias son establecidas automáticamente por el IDE, facilitando el trabajo del programador que no necesita preocuparse por los espacios de nombres esenciales necesita para su aplicación.

Las referencias establecidas por el IDE varían en función del estilo de proyecto elegido: aplicación de Windows, de consola, etc. El programador puede, naturalmente, establecer referencias adicionales en función de las necesidades del programa.

Respecto al fichero ASSEMBLYINFO.VB, contiene información adicional del ensamblado, fundamentalmente en forma de atributos, para el entorno de ejecución. Podemos editar este fichero para cambiar ciertos parámetros del ensamblado. Ver Código fuente 27.

```
Imports System.Reflection
Imports System.Runtime.InteropServices

' General Information about an assembly is controlled through the following
' set of attributes. Change these attribute values to modify the information
' associated with an assembly.

' Review the values of the assembly attributes

<Assembly: AssemblyTitle("")>
<Assembly: AssemblyDescription("")>
<Assembly: AssemblyCompany("")>
<Assembly: AssemblyProduct("")>
<Assembly: AssemblyCopyright("")>
<Assembly: AssemblyTrademark("")>
<Assembly: CLSCompliant(True)>

'The following GUID is for the ID of the typelib if this project is exposed to COM
<Assembly: Guid("C0158A80-9226-4712-A38C-17233E5767CE")>

' Version information for an assembly consists of the following four values:
'
```

```
'       Major Version
'       Minor Version
'       Build Number
'       Revision
'
' You can specify all the values or you can default the Build and Revision Numbers
' by using the '*' as shown below:

<Assembly: AssemblyVersion("1.0.*")>
```

Código fuente 27. Contenido del fichero ASSEMBLYINFO.VB creado por el IDE.

El modo de apertura de un proyecto explicado hasta el momento consiste en iniciar VS.NET y abrir después el proyecto. Sin embargo, podemos hacer directamente doble clic sobre el fichero del proyecto (fichero con extensión .VBPROJ), y esta acción abrirá el IDE y cargará el proyecto en un solo paso.

Agregar nuevos elementos a un proyecto

Una vez creado un nuevo proyecto, necesitaremos con toda probabilidad añadir formularios, clases, etc., adicionales. Para ello, seleccionaremos alguna de las opciones del menú Proyecto, que comienzan por *Agregar <NombreOpción>*, y que nos permiten agregar un nuevo elemento al proyecto. Ver Figura 84.

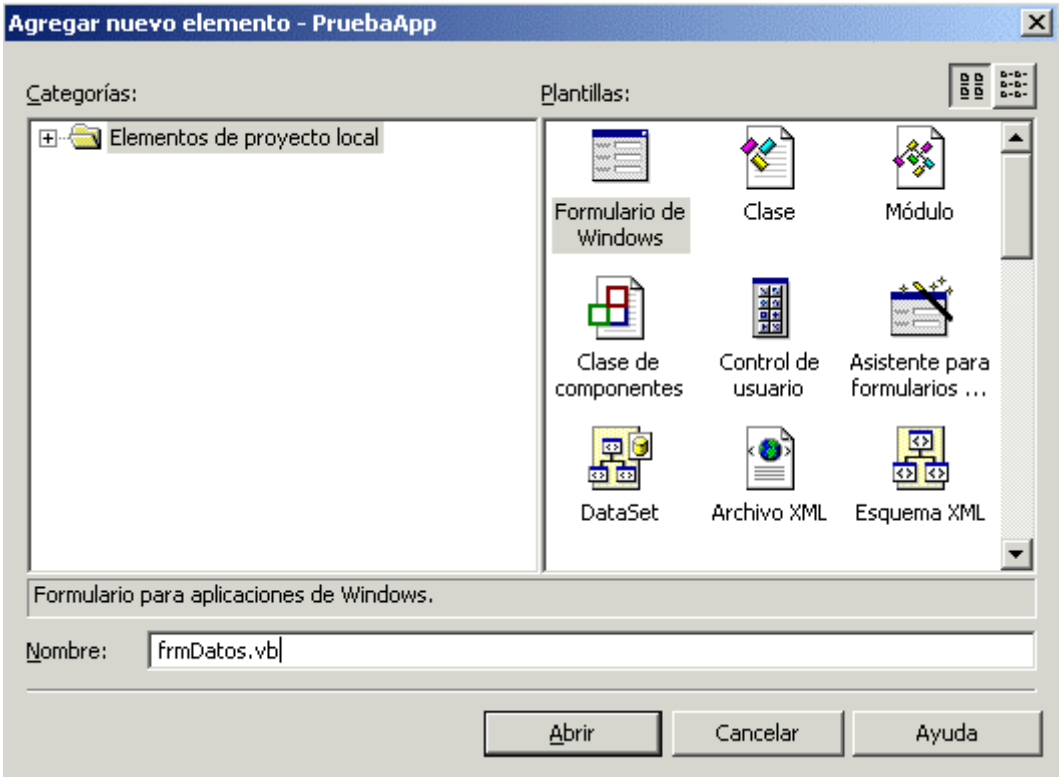


Figura 84. Agregando un nuevo formulario al proyecto.

Estas opciones hacen uso común de la caja de diálogo *Agregar nuevo elemento*. Lo que sucede, es que si elegimos añadir por ejemplo un formulario, la caja de diálogo se abre posicionándonos ya en la plantilla correspondiente al elemento que queremos añadir, ello supone un atajo y nos ahorra el paso

de selección del elemento. Pero podemos cambiar libremente dicho elemento, seleccionando uno diferente en el panel derecho de este diálogo. La Figura 85 muestra un proyecto en el que además del formulario por defecto, Form1, se ha agregado el formulario, frmDatos, la clase Factura, el módulo, General, y el fichero de texto TextFile1.

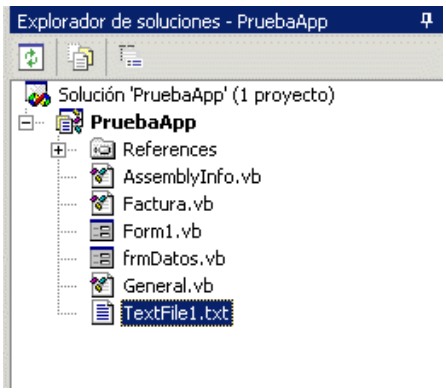


Figura 85. Proyecto con diversos elementos.

Haciendo clic en el segundo botón de la barra de herramientas de esta ventana, *Mostrar todos los archivos*, observará el lector como se visualizan las carpetas del proyecto que contienen los ficheros inicialmente ocultos, como el ejecutable.

Propiedades del proyecto

Si necesitamos variar los parámetros de configuración del proyecto, deberemos abrir la ventana de propiedades del proyecto, haciendo clic sobre el mismo y seleccionando la opción de menú *Proyecto + Propiedades* (este punto se explicó en el tema *Escritura de código*). Ver Figura 86.

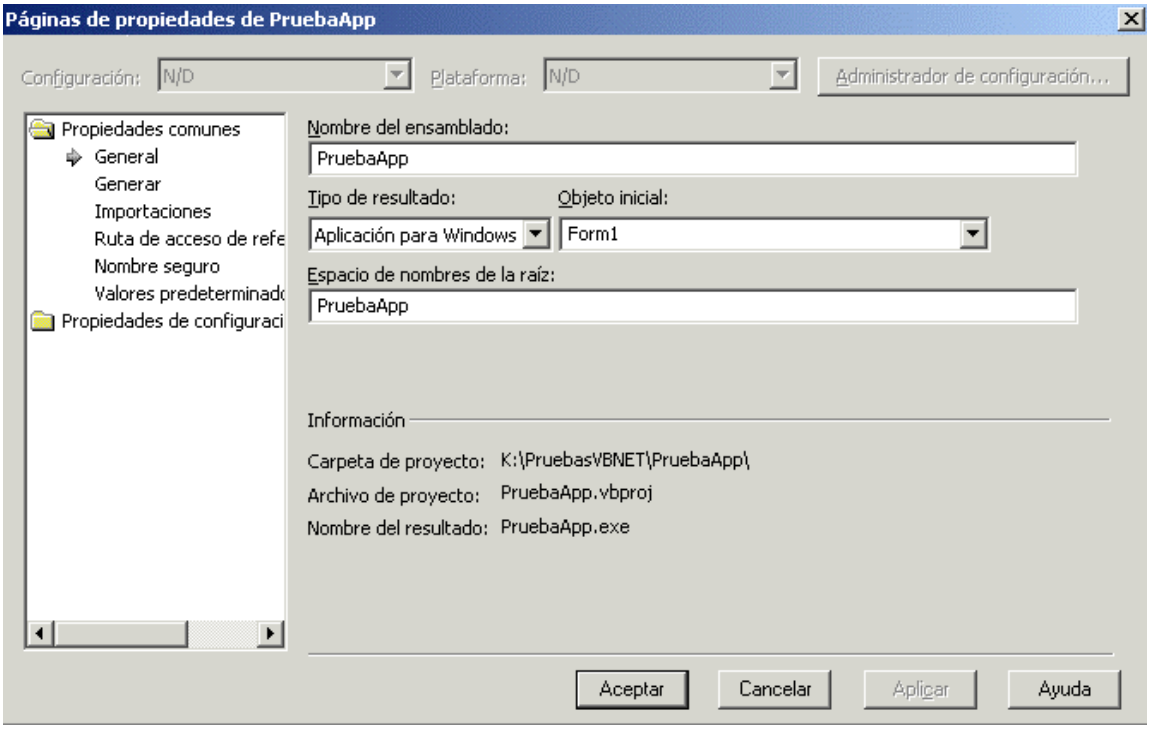


Figura 86. Ventana de propiedades del proyecto.

Esta ventana de propiedades está organizada en dos carpetas: *Propiedades comunes* y *Propiedades de configuración*, situadas en la parte izquierda, y que contienen cada una, los diversos apartados en los que se organizan las propiedades. Al hacer clic en cada apartado, la parte derecha cambiará mostrando las propiedades relacionadas. De esta forma podemos configurar aspectos tales como el tipo de aplicación resultante, el punto de entrada, nombre de ensamblado, espacios de nombres importados, generación de nombres para ensamblados compartidos, ruta de generación de ficheros del proyecto, etc.

Propiedades de la solución

Al igual que el proyecto, si hacemos clic sobre el nombre de la solución y pulsamos el botón Propiedades de la barra de herramientas, se abrirá la ventana de propiedades de la solución, en la que podremos configurar aspectos tales como el proyecto de inicio, en el caso de una solución con varios proyectos; establecer las dependencias de un proyecto, etc. Ver Figura 87.

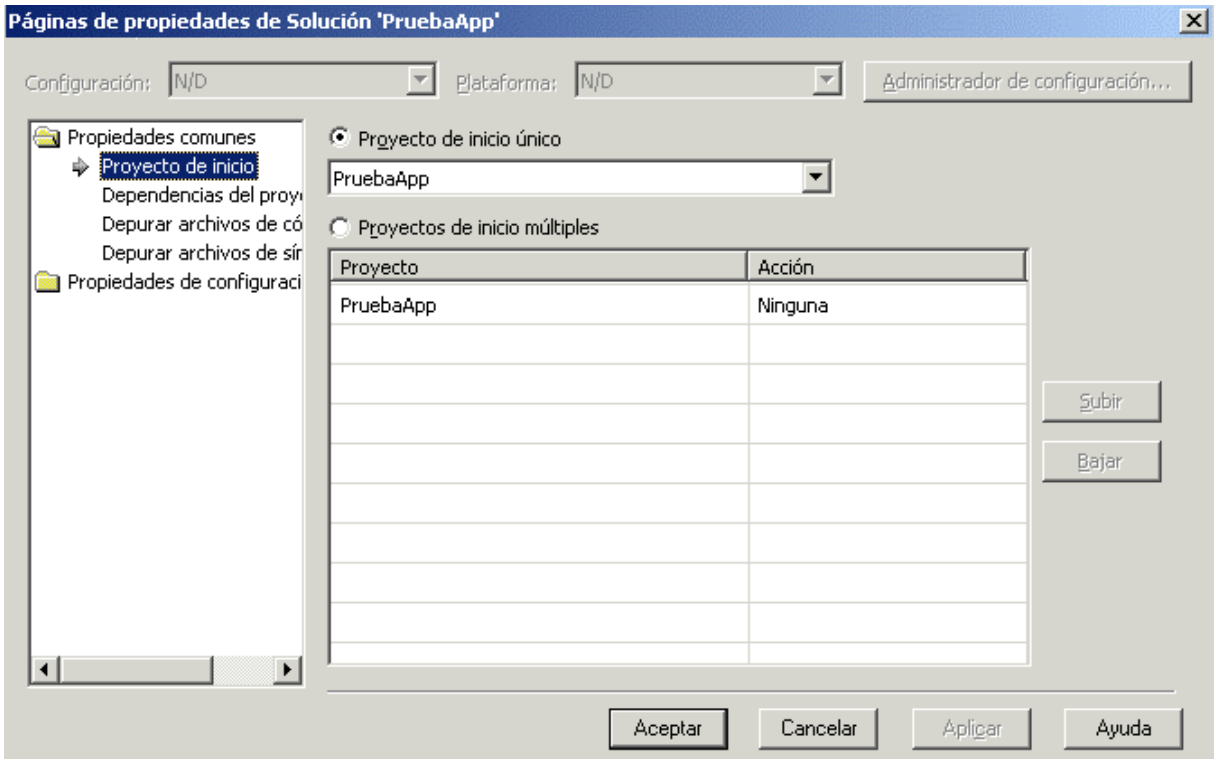


Figura 87. Ventana de propiedades de la solución.

Agregar proyectos a una solución

Como hemos indicado anteriormente, una solución puede contener varios proyectos. Para agregar un nuevo proyecto o uno existente a una solución, seleccionaremos la opción del menú de VS.NET *Archivo* + *Nuevo* + *Proyecto* o *Archivo* + *Abrir* + *Proyecto* respectivamente. También podemos hacer clic derecho sobre la solución, en el explorador de soluciones, y elegir del menú contextual una de las opciones de Agregar. La Figura 88 muestra la caja de diálogo para añadir un nuevo proyecto a una solución.

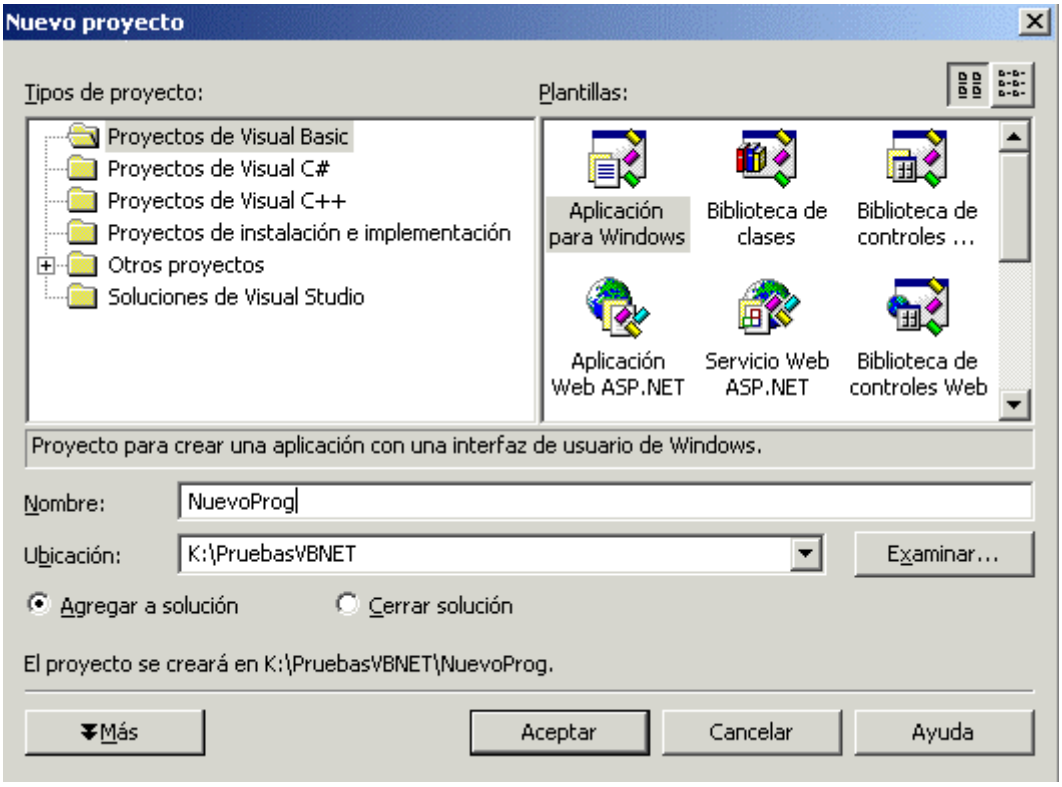


Figura 88. Añadir un nuevo proyecto a la solución actual.

Observe el lector, en el caso de la figura anterior, que para que el nuevo proyecto sea agregado a la solución en la que nos encontramos, debemos marcar la opción *Agregar a solución*.

Una vez añadido un proyecto a una solución, formándose una solución multiproyecto, el explorador de soluciones mostrará un aspecto parecido al de la Figura 89.

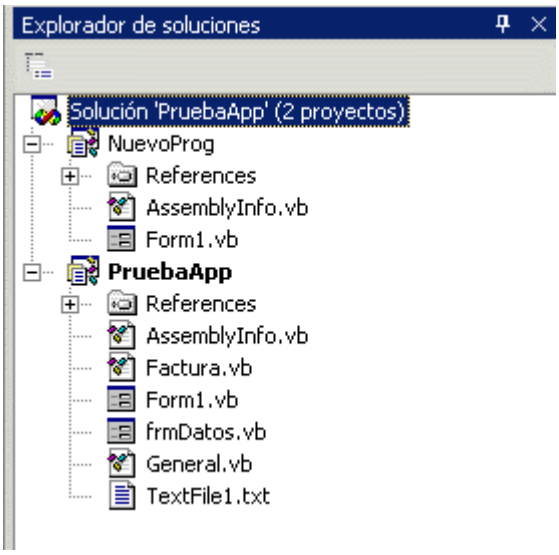


Figura 89. Solución conteniendo varios proyectos.

El menú contextual

Tanto si nos encontramos en la ventana del explorador de soluciones como en cualquier otra, podemos acceder de un modo rápido a múltiples opciones de los elementos situados en la ventana, haciendo clic derecho sobre un elemento, de modo que se abrirá el menú contextual correspondiente, en el que podremos elegir operaciones relacionadas con el elemento seleccionado. La Figura 90 muestra el menú contextual de un proyecto.

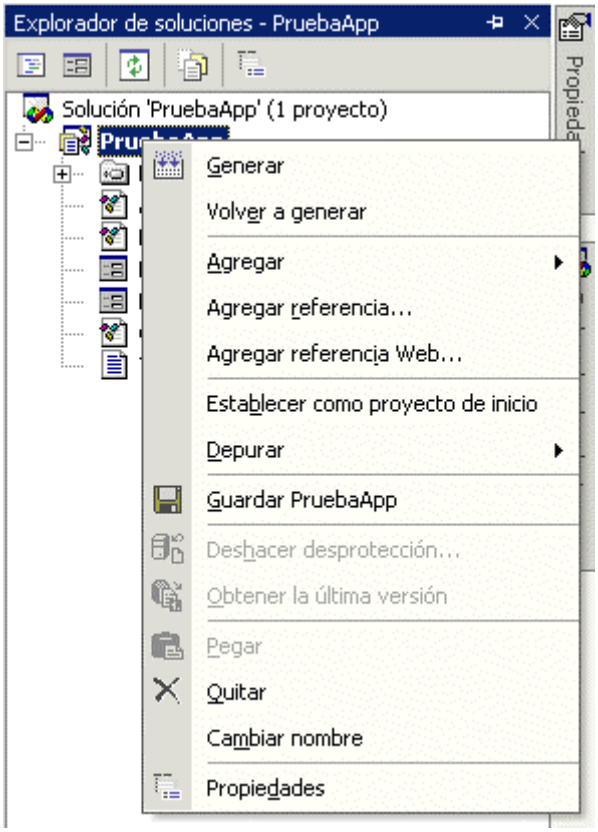


Figura 90. Menú contextual de un proyecto.

El diseñador del formulario

Contenido en la ventana principal del IDE, el diseñador del formulario es uno de los elementos más importantes del entorno de desarrollo, y aunque haremos una revisión en profundidad en el tema dedicado al trabajo con formularios, no podemos dejar de mencionarlo en este tema dedicado al IDE. Ver Figura 91.

Este diseñador muestra la representación de un formulario del proyecto, en el que a modo de plantilla, iremos situando los controles que componen el interfaz de usuario de la ventana.

A la hora de ubicar controles en un formulario, si empleamos controles que no disponen de interfaz de usuario, aparecerá en la parte inferior de esta ventana un panel con dichos controles. Ver Figura 92.

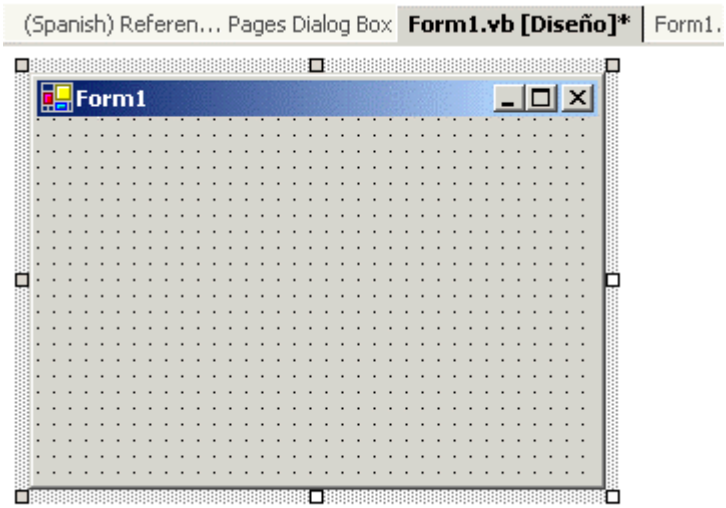


Figura 91. Diseñador del formulario.

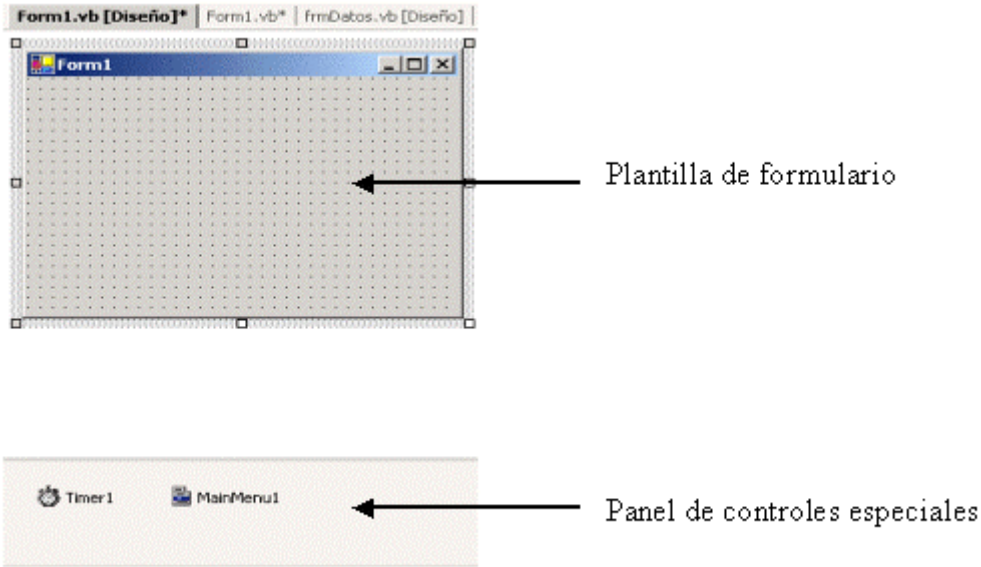


Figura 92. Diseñador de formulario con panel de controles.

La ventana de propiedades

Cuando estamos diseñando un formulario, esta ventana muestra las propiedades del objeto que tengamos seleccionado en el diseñador: bien un control o el propio formulario. La Figura 93 muestra esta ventana indicando sus elementos principales.

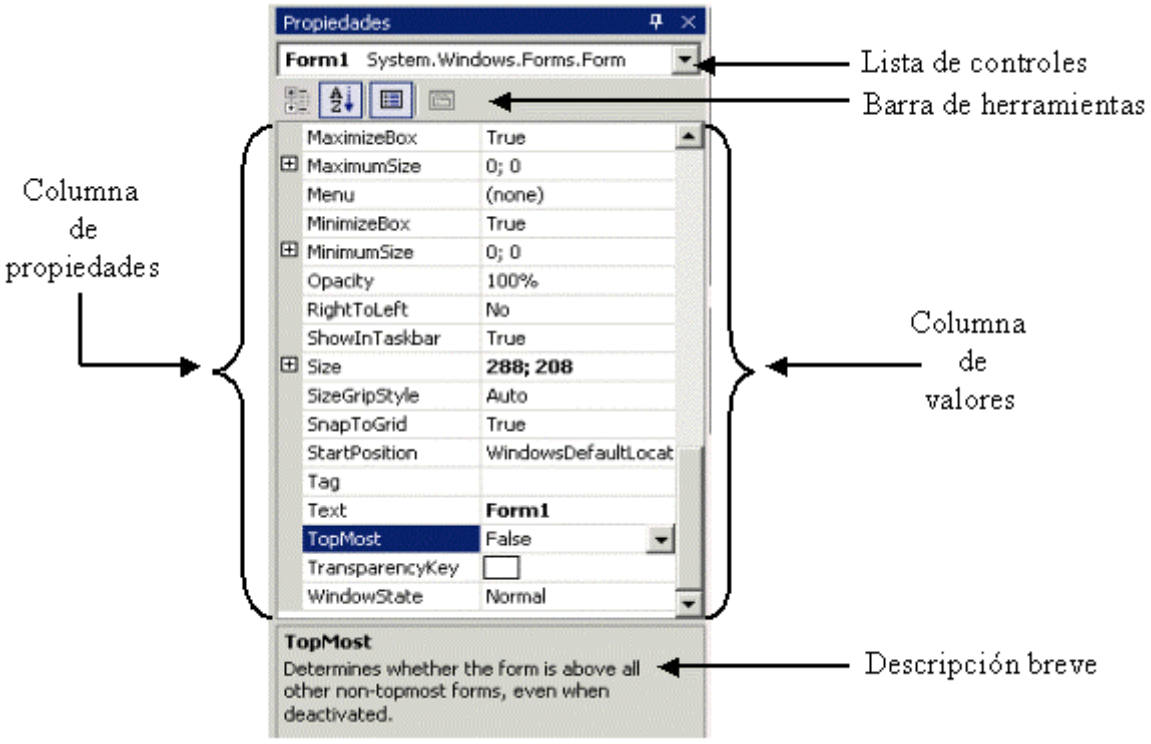


Figura 93. Ventana de propiedades de VS.NET.

Como vemos en la figura anterior, las propiedades se organizan en dos columnas: una contiene los nombres de las propiedades y otra sus valores. Las propiedades compuestas de varios miembros, incluyen en el lateral izquierdo un signo + para expandirlos.

Ciertas propiedades contienen una lista de valores, que podemos abrir con el botón que figura en el valor de la propiedad. Ver Figura 94.

Figura 94. Propiedad con lista de valores

Existe otras propiedades cuyo valor es seleccionado mediante una caja de diálogo. En esta propiedades, se muestra en su valor, un botón con puntos suspensivos indicando que debemos pulsarlo para modificar su valor. Ver Figura 95.

Figura 95. Propiedad modificable mediante caja de diálogo.

Podemos hacer clic sobre un control del formulario para pasar a continuación a ver sus propiedades, o bien podemos elegir el control de la lista desplegable de controles. La Figura 96 muestra esta lista con el propio formulario y varios controles adicionales.

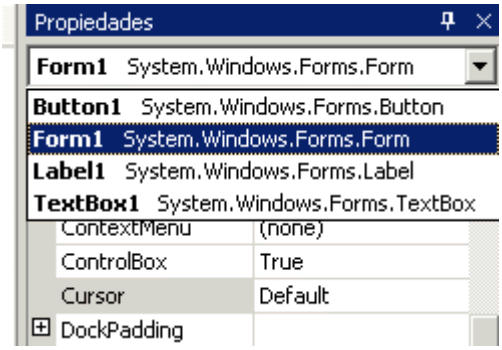


Figura 96. Lista de controles de la ventana de propiedades.

Los dos primeros botones de la barra de herramientas de esta ventana, nos permiten respectivamente, ordenar las propiedades por categoría o alfabéticamente. Mientras que en la parte inferior, se visualiza una descripción resumida de la propiedad que tengamos seleccionada. Si no deseamos ver dicha descripción, haremos clic derecho sobre la ventana, seleccionando la opción de menú Descripción.

9

El IDE de Visual Studio .NET. Herramientas y editores

El Cuadro de herramientas

Situado habitualmente como una ficha expandible en el lateral izquierdo del IDE, la ventana *Cuadro de herramientas* contiene todos los controles que podemos insertar en un formulario para construir el interfaz de usuario de la aplicación. Ver Figura 97.

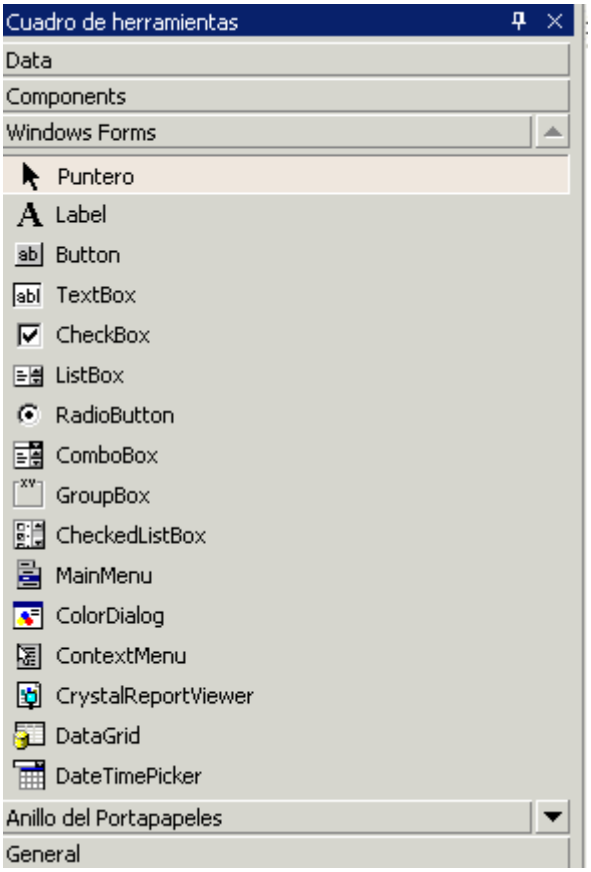


Figura 97. Cuadro de herramientas de VS.NET.

Organización en fichas

Esta ventana está organizada en base a una serie de fichas en forma de barras, en las que al hacer clic se despliegan sus elementos. Por defecto, cuando abrimos el cuadro de herramientas, se muestra abierta la ficha *Windows Forms*, conteniendo los controles que habitualmente utilizaremos en los formularios, aunque también disponemos de las fichas *Data*, *Components*, etc.

Ya que habitualmente no haremos uso de todos los controles en un programa, algunas fichas se encuentran ocultas, por lo que si queremos tener todas disponibles, haremos clic derecho sobre el cuadro de herramientas y elegiremos la opción de menú *Mostrar todas las fichas*, con lo que fichas como *HTML*, *XML Schema*, *Dialog Editor*, etc., que hasta ese momento no estaban disponibles, podrán ser usadas por el programador.

Para seleccionar un control, sólo hemos de desplazarnos por la lista de controles de la ficha que tengamos abierta con las teclas de desplazamiento o los botones de la ventana que realizan también dicha función y que se encuentran en el título de la ficha actual y la siguiente. Ver Figura 98.

Figura 98. Botones de desplazamiento del cuadro de herramientas.

Manipulación de fichas

Las fichas del cuadro de herramientas se muestran en un orden predeterminado por el IDE, pero podemos cambiar la posición de cualquier ficha haciendo clic sobre la misma y arrastrando arriba o abajo hasta soltarla en una nueva posición.

Esta flexibilidad también se extiende al hecho de que podemos crear nuestras propias fichas para situar en ella controles ya existentes en el cuadro de herramientas o nuevos controles.

Para crear una nueva ficha, que llamaremos MisControles, haremos clic derecho en el cuadro de herramientas y seleccionaremos la opción de menú *Agregar ficha*, ello creará una nueva ficha vacía en la que escribiremos su nombre y pulsaremos [INTRO] para terminar, véase la Figura 105. El modo de añadir controles a una ficha será explicado en un próximo apartado.

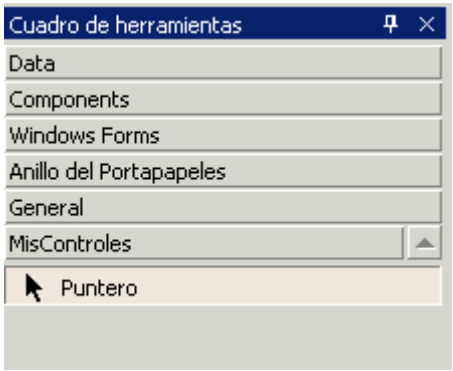


Figura 99. Nueva ficha creada por el programador.

Es posible también, cambiar de nombre y eliminar las fichas existentes, haciendo clic derecho sobre su título y eligiendo del menú contextual, la opción *Eliminar ficha* o *Cambiar nombre de ficha*. Debemos tener en cuenta que sólo podremos eliminar las fichas que hayamos creado nosotros, puesto que las que originalmente se muestran en el IDE, pertenecen al entorno de desarrollo y no pueden eliminarse.

Organización de controles

Los controles dentro de una ficha se visualizan por defecto con el icono del control y su nombre, en el llamado modo *Vista de lista*. No obstante, podemos hacer que los controles se muestren en vista de iconos, al estilo de VB6, haciendo clic derecho sobre el cuadro de herramientas y seleccionando la opción de menú *Vista de lista*, que se encontrará marcada, desmarcándola de esa forma, y quedando esta ventana con un aspecto parecido al de la Figura 100.

Aunque esta vista pueda ser inicialmente de mayor agrado para los programadores de VB6, creemos que la vista de lista, al incluir el nombre del control es más práctica de manejar, por lo que vamos a dejar de nuevo esta ventana con dicha vista.

Podemos ordenar los controles de una ficha por su nombre, haciendo clic derecho sobre esta ventana y eligiendo la opción de menú *Ordenar elementos alfabéticamente*.

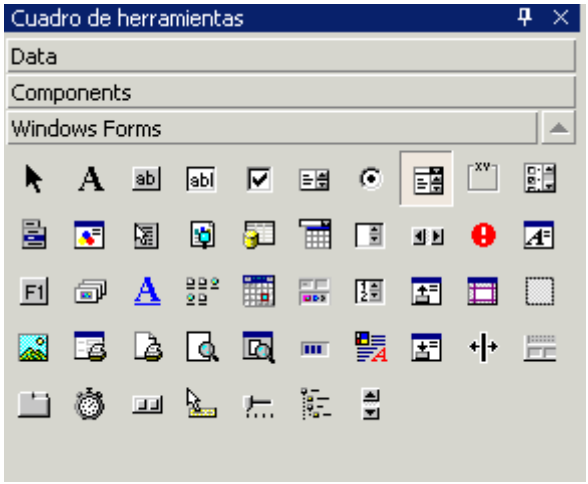


Figura 100. Cuadro de herramientas mostrando controles con vista de iconos.

El orden alfabético puede tener el inconveniente de situar algunos controles de uso frecuente (por ejemplo TextBox), al final de la lista. Para remediar este problema, podemos cambiar la posición de los controles dentro de la lista de dos maneras: haciendo clic sobre el control y arrastrándolo hasta la posición que deseemos; o bien, haciendo clic derecho sobre la lista de controles y eligiendo las opciones *Subir* o *Bajar*. En este último modo podemos lograr una ordenación mixta, en parte alfabética, pero con nuestro controles de mayor uso en posiciones preferentes. La Figura 101 muestra un ejemplo de este orden combinado.

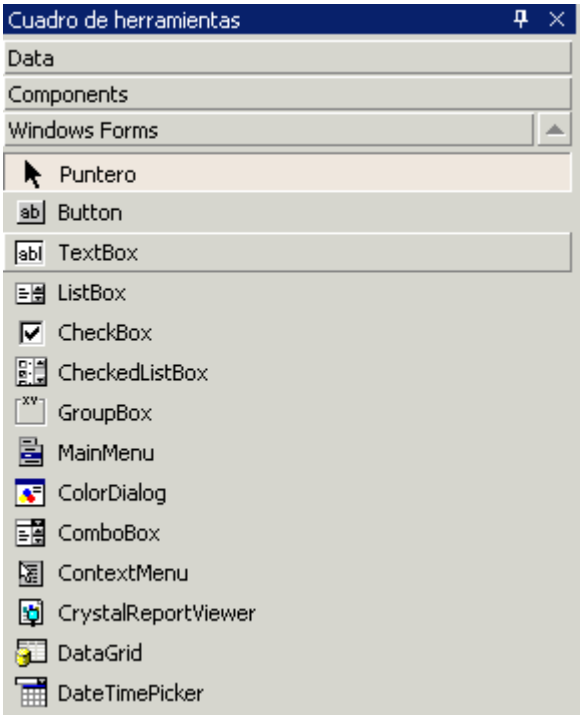


Figura 101. Controles con orden mixto: parte alfabética y parte preferente.

Manipulación de controles

Todos los controles aparecen con su nombre original en el cuadro de herramientas. No obstante, es posible cambiar dicho nombre por otro que sea más fácilmente identificable por nosotros, o con el que nos sintamos más cómodos.

Por ejemplo, supongamos que queremos cambiar el nombre del control StatusBar. Para ello, haremos clic derecho sobre dicho control y seleccionaremos la opción *Cambiar nombre de elemento*, lo que nos permitirá editar directamente el nombre en la lista de controles. Introduciremos como nuevo nombre *Barra de estado* y pulsaremos [INTRO], el resultado se refleja en la Figura 102.

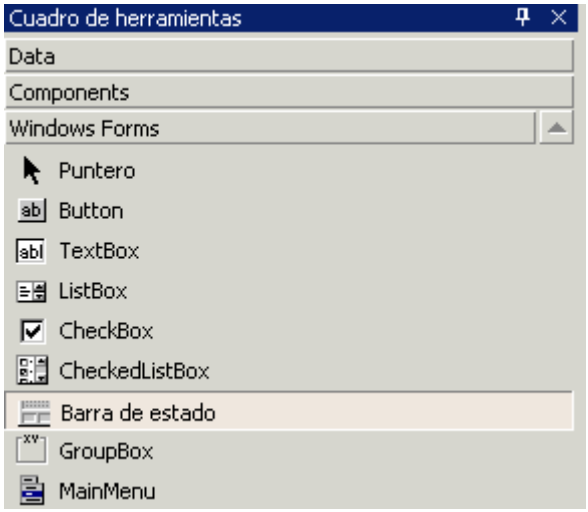


Figura 102. Cambio de nombre para un control.

Es importante remarcar, que este tipo de cambio sólo es a nivel del literal mostrado por el control en el cuadro de herramientas, ya que cuando añadamos una copia del control al formulario, el nombre que le asignará el IDE seguirá siendo su nombre original: StatusBar.

Mediante las operaciones estándar de Windows: Cortar, Copiar y Pegar, podemos cambiar de ficha los controles, mientras que si hacemos clic y arrastramos un control hacia otra ficha, moveremos dicho control de su ficha original, depositándolo en una nueva. La Figura 103 muestra la ficha General, con los controles Panel y ListView, originarios de la ficha Windows Forms: el primero se ha copiado y el segundo se ha movido a esta ficha.

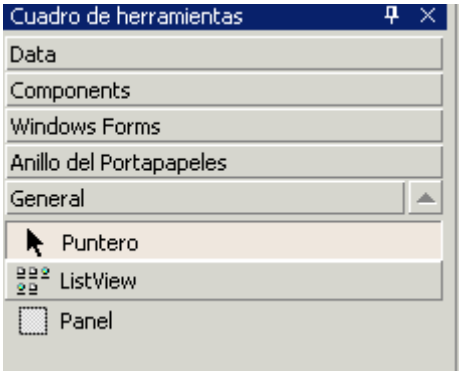


Figura 103. Controles cambiados de ficha.

Agregar controles

Además de poder organizar los controles existentes, es posible añadir nuevos controles a cualquiera de las fichas del cuadro de herramientas. Para ello, y una vez situados sobre una ficha, debemos hacer clic derecho y seleccionar la opción de menú *Personalizar cuadro de herramientas*, que nos mostrará la caja de diálogo del mismo título de la Figura 104.

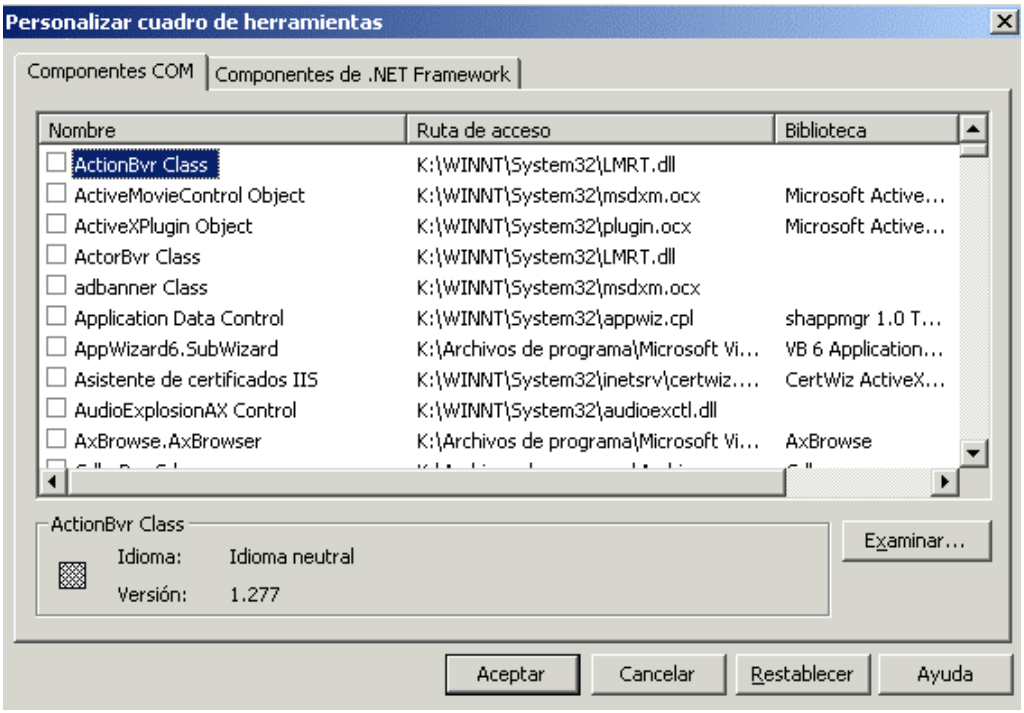


Figura 104. Ventana para personalizar el cuadro de herramientas.

En esta ventana, y dependiendo de la ficha seleccionada, podemos añadir al cuadro de herramientas tanto controles basados en componentes COM, como componentes de .NET Framework.

Como ejemplo, nos situaremos en la ficha General del cuadro de herramientas, abriendo a continuación la ventana para personalizar el cuadro de herramientas. Haremos clic sobre la ficha de componentes de .NET, y marcaremos la casilla del control DirListBox, pulsando seguidamente el botón Aceptar, con lo que dicho control se añadirá a la ficha. Ver Figura 105.

Figura 105. Control DirListBox añadido al cuadro de herramientas.

La operación inversa, es decir, eliminar un control del cuadro de herramientas, podemos realizarla de dos formas:

- Seleccionar el control y pulsar [SUPR], confirmando su eliminación del cuadro de herramientas.
- Abrir la ventana para personalizar el cuadro de herramientas, localizar el control y quitar la marca de su casilla. Con ello desaparecerá del cuadro de herramientas.

El cuadro de herramientas como contenedor de código fuente

El cuadro de herramientas, mediante su ficha *Anillo del Portapapeles*, permite el intercambio de código fuente entre diversos elementos del proyecto, a través de un acceso visual al Portapapeles del sistema. Veamos un ejemplo a continuación.

Supongamos que en nuestro proyecto hemos creado una clase llamada Factura en la que escribimos el método del Código fuente 28.

```
Public Sub Conteo(ByVal Importe As Long)
    Dim Indicador As Integer
    Dim Total As Long

    For Indicador = 1 To 100
        Total += Importe + 4
    Next
End Sub
```

Código fuente 28. Método Conteo( ) de la clase Factura.

Posteriormente, añadimos un botón al formulario del proyecto y en el código de su evento escribimos la línea del Código fuente 29.

```
MessageBox.Show("Has pulsado un botón")
```

Código fuente 29. Código para el botón de un formulario.

Ambos códigos fuente los seleccionamos y copiamos al Portapapeles en el modo habitual. Hasta aquí nada de particular, pero si abrimos el cuadro de herramientas y hacemos clic en la ficha Anillo del Portapapeles, veremos algo parecido a la Figura 106.

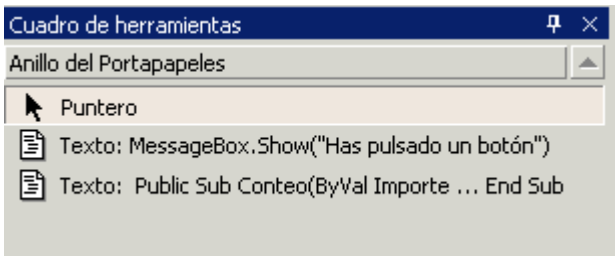


Figura 106. Acceso al código fuente copiado al Portapapeles desde el cuadro de herramientas de VS.NET.

Si a continuación, abrimos el editor de código correspondiente a cualquier elemento del proyecto: formulario, clase, módulo, etc., podemos hacer clic en cualquiera de los elementos del Anillo del Portapapeles y arrastrarlo, copiándolo en dicho editor de código. También conseguiremos el mismo resultado haciendo doble clic en cualquiera de los elementos de esta ficha. Esta característica aporta una gran comodidad en la manipulación del código fuente de la aplicación.

Las barras de herramientas

El IDE de VS.NET dispone de un gran número de barras de herramientas, de las cuales sólo se muestran por defecto la estándar. El programador puede posteriormente, visualizar barras adicionales y crear sus propias barras de herramientas personalizadas.

Para mostrar alguna de las barras de herramientas definidas en el IDE, podemos hacerlo de las siguientes formas:

- Clic derecho sobre una de las barras actualmente visible, lo que mostrará un menú contextual con todas las barras existentes, en el que podremos elegir una. Ver Figura 107.

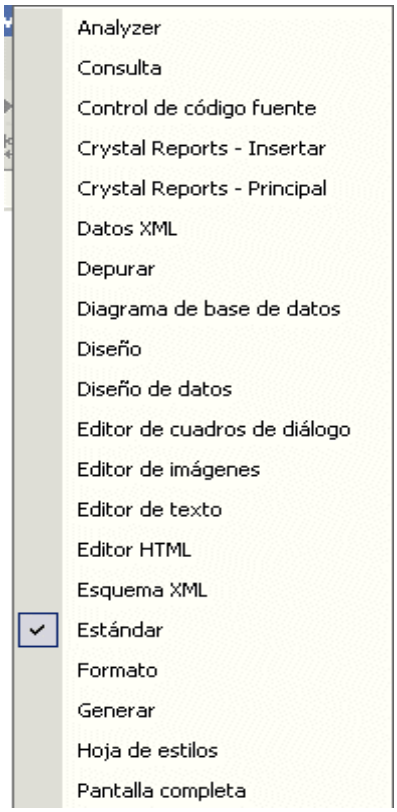


Figura 107. Menú contextual con las barras de herramientas del IDE.

- Opción de menú *Herramientas + Personalizar*, que mostrará la ventana Personalizar, en la que dentro de la ficha *Barras de herramientas*, podremos visualizar y ocultar barras, marcando y desmarcando respectivamente la casilla asociada a dicha barra. Ver Figura 108.

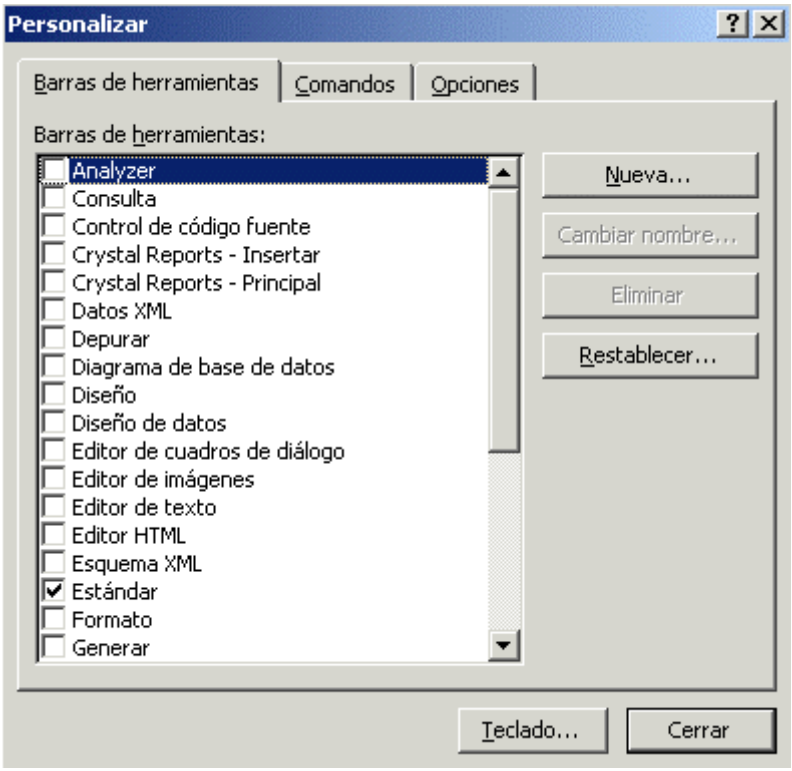


Figura 108. Ventana Personalizar para manejar las barras de herramientas del IDE.

Marcando por ejemplo, la barra *Editor de texto*, se visualizará esta barra, situándose debajo de la estándar. Ver Figura 109.

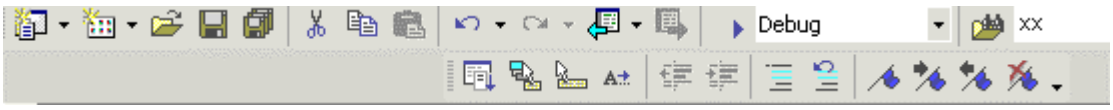


Figura 109. Barra de herramientas Editor de texto debajo de la barra estándar.

Barras de herramientas personalizadas

Durante nuestro trabajo habitual, es muy posible que empleemos con frecuencia acciones situadas en diferentes barras de herramientas. Para no tener una cantidad excesiva y posiblemente innecesaria de barras abiertas al mismo tiempo, podemos crear una barra personalizada (o varias) con nuestro juego de botones favoritos. Los pasos a dar para conseguirlo se describen seguidamente:

Abriremos en primer lugar la ventana Personalizar y pulsaremos el botón Nueva, tras lo que deberemos de introducir el nombre de la nueva barra de herramientas, por ejemplo: MisBotones. Esto creará una nueva barra vacía a la que deberemos añadir botones. Ver Figura 110.

El siguiente paso consiste en hacer clic sobre la ficha Comandos de la ventana Personalizar, para seleccionar en el panel izquierdo la categoría de comando a incluir, y en el panel derecho el comando a insertar. Una vez elegido el comando, lo arrastraremos y soltaremos en nuestra barra de herramientas. Ver Figura 111.

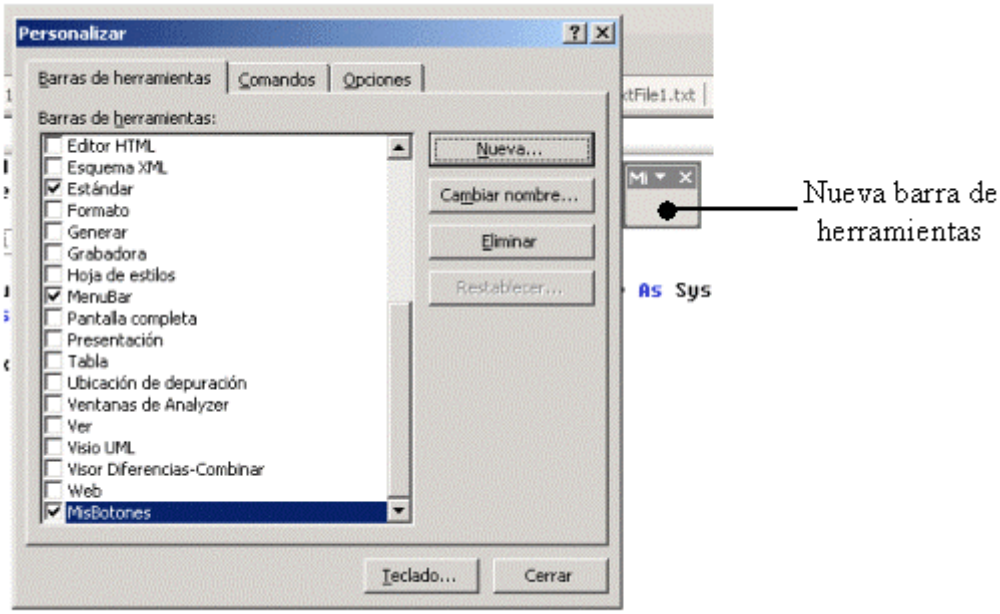


Figura 110. Creación de una nueva barra de herramientas.

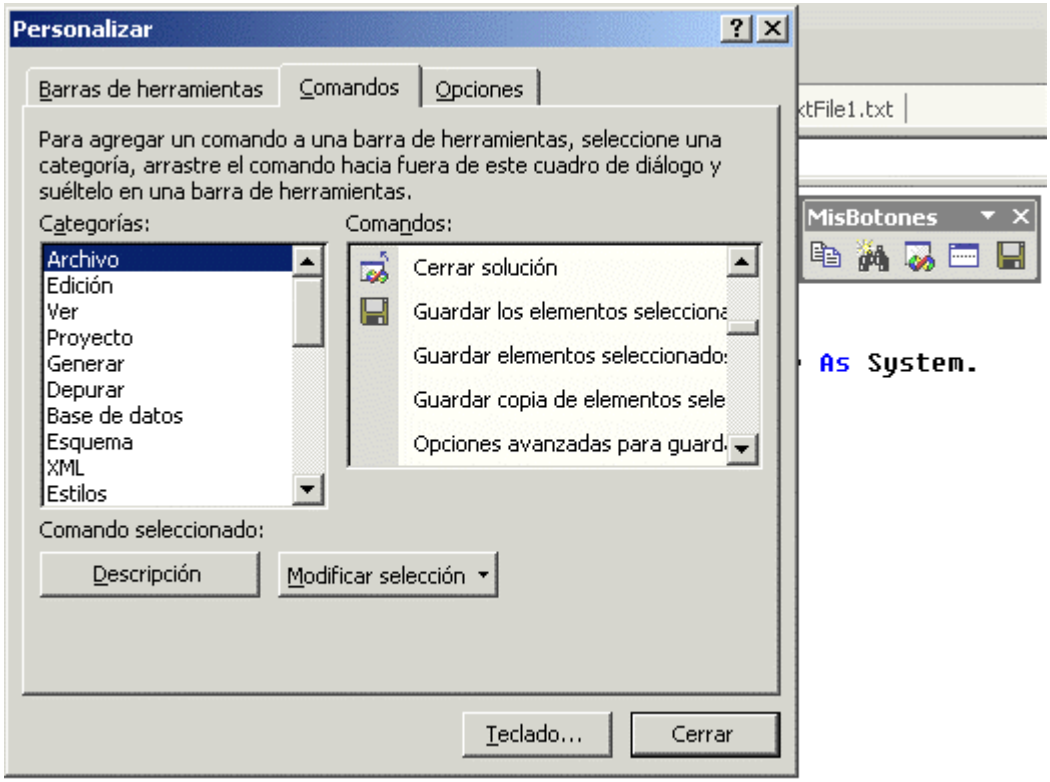


Figura 111. Selección de comandos e inserción en barra de herramientas personalizada.

Terminada la confección de nuestra barra de herramientas, pulsaremos el botón Cerrar de la ventana Personalizar, quedando nuestra barra en modo flotante sobre el IDE.

Acople de barras de herramientas

Al haber finalizado de crear una barra de herramientas, podemos dejarla flotando sobre cualquier área del entorno de trabajo, o bien, acoplarla debajo de las barras ya visibles o en los laterales del IDE. Esta operación la realizaremos de un modo simple: haciendo clic sobre el título de la barra y desplazándola hasta que quede acoplada en el destino elegido. Podemos cambiar de esta forma, la posición de todas las barras de VS.NET.

La Figura 112 muestra la barra estándar y la barra personalizada que acabamos de crear en su posición habitual; la barra de edición de código fuente en un lateral del entorno; mientras que la barra para depurar está flotando en el IDE. El indicador con forma de pequeñas líneas situado en la parte izquierda de cada barra, antes del primer botón, sirve para hacer clic sobre él y mover la barra de lugar.

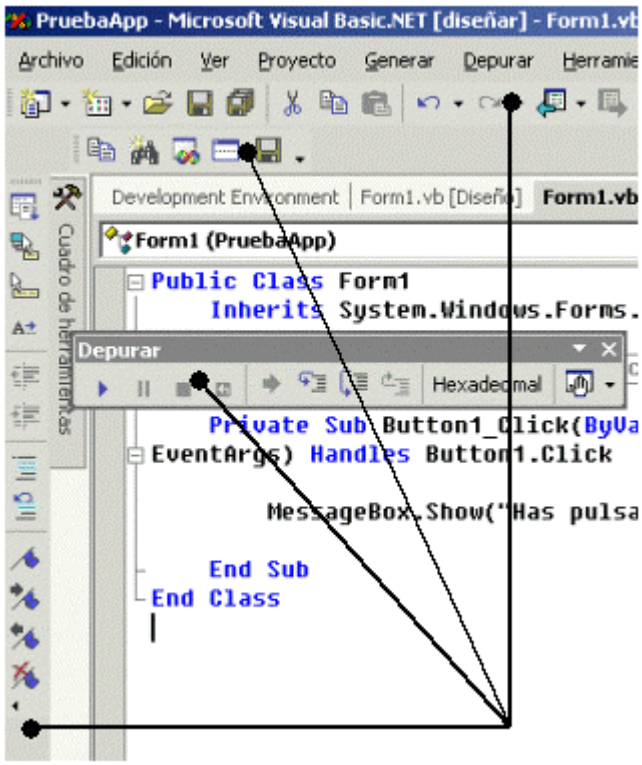


Figura 112. Distintos lugares de acoplamiento para las barras de herramientas.

Opciones adicionales de personalización

También a través de la ventana Personalizar, en su ficha Opciones encontramos un conjunto diverso de selecciones, que nos permiten ampliar el tamaño de los iconos de las barras de herramientas, aplicar efectos a las animaciones de menú, etc.

La Figura 113 muestra esta ficha una vez aplicado el efecto de iconos grandes al IDE.

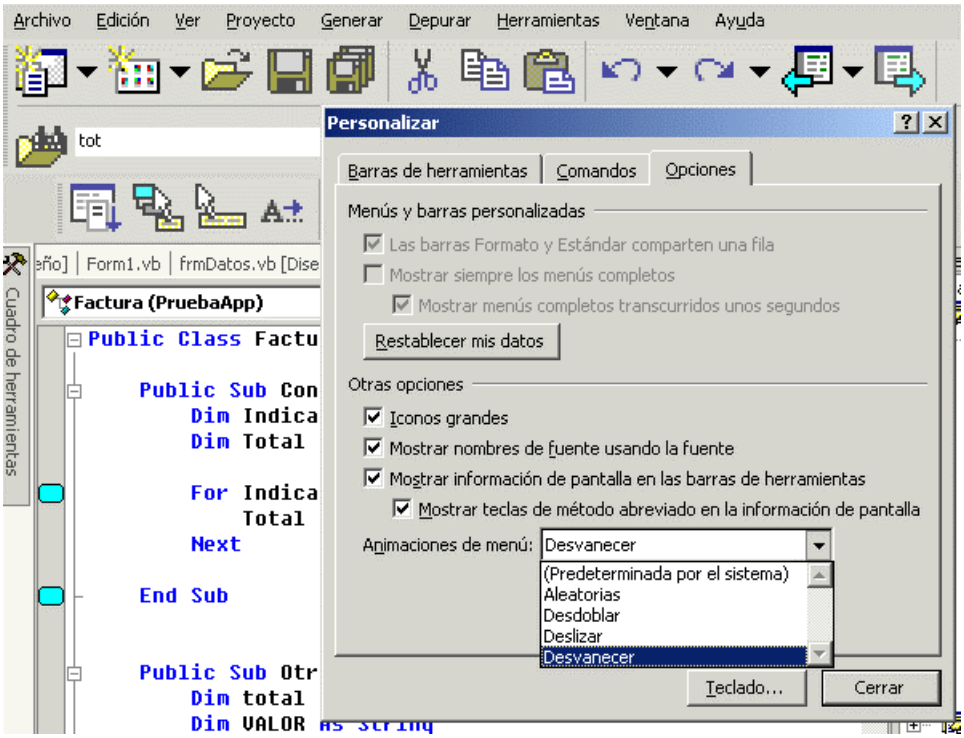


Figura 113. Ventana Personalizar, para opciones adicionales.

Ventana de resultados

Esta ventana se encuentra habitualmente en la parte inferior del IDE, y muestra el producto de acciones diversas, como la compilación previa a la ejecución, generación de ejecutable, etc. La Figura 114 muestra esta ventana en la que aparece el resultado de la ejecución de una aplicación en depuración, es decir, una aplicación ejecutada desde el propio IDE.

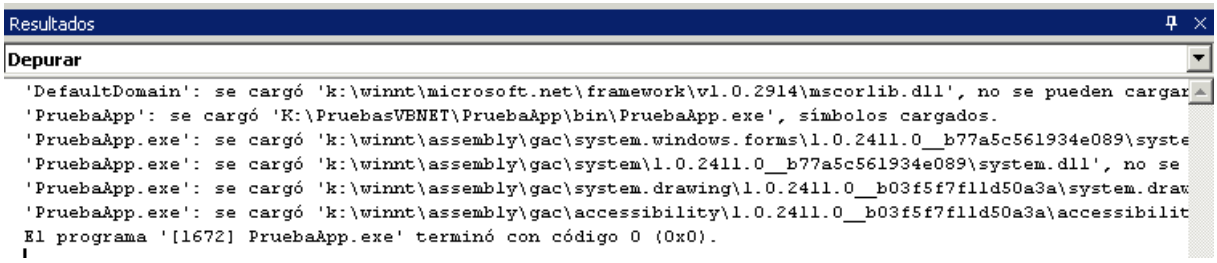


Figura 114. Ventana de resultados.

Si necesitamos visualizar otros resultados, como por ejemplo el de la generación del ejecutable, tenemos que abrir la lista desplegable situada en la parte superior de esta ventana.

Existen otras variantes de esta ventana, que muestran los resultados de búsquedas efectuadas en la ayuda del IDE, búsquedas de símbolos en el examinador de objetos, etc. Todas ellas se sitúan como fichas en la parte inferior del entorno de trabajo. Ver Figura 115.

Resultados de la búsqueda de textbox: 11 temas encontrados

Título	Ubicación	Jerarquía
13.4.3 Interface implementation inheritance (C#)	C# Language Specification	1
mNotepad Sample: Demonstrates Managed Exte...	Visual C++ Samples	2
CHtmlEditCtrlBase::TextBox (MFC)	Visual C++ Libraries	3
Shopping Cart	Visual Studio Samples: Duwamish 7.0	4
13.4 Interface implementations (C#)	C# Language Specification	5
13.4.1 Explicit interface member implementation...	C# Language Specification	6
CHtmlEditCtrlBase Members (MFC)	Visual C++ Libraries	7
Categorical List of Managed Extensions for C++...	Visual C++ Samples	8
Alphabetical List of Managed Extensions for C+...	Visual C++ Samples	9
Calculator Sample: Windows Forms Pocket Calcu...	Visual C++ Samples	10
Property Pages Version Tab	Visual Basic Language Reference	11

Lista de tareas | Resultados | Resultados de la búsqueda de símbolos | Resultados de la búsqueda de textbox: 11 temas encontrados

Figura 115. Fichas de resultados de búsquedas en el IDE.

El editor de código fuente

En esta ventana escribiremos el código del programa, en forma de declaraciones y procedimientos dentro de clases, módulos, etc. Ver Figura 116.

Development Environment | Form1.vb [Diseño] | **Form1.vb** | frmDatos.vb [Diseño] | Factura.vb | General.vb | TextFile1.txt

Form1 (PruebaApp) | **(Declaraciones)**

```
Public Class Form1
    Inherits System.Windows.Forms.Form

#Region " Windows Form Designer generated code "

    Public Sub New()
        MyBase.New()

        'This call is required by the Windows Form Designer.
        InitializeComponent()

        'Add any initialization after the InitializeComponent() call

    End Sub

    'Form overrides dispose to clean up the component list.
    Protected Overloads Overrides Sub Dispose(ByVal disposing As Boolean)
        If disposing Then
            If Not (components Is Nothing) Then
                components.Dispose()
            End If
        End If
```

Figura 116. Editor de código fuente de VS.NET.

Esta ventana dispone de multitud de opciones y características dada su importancia, por lo que en este apartado trataremos las más importantes.

Ajuste de fuente y color

Al haber seleccionado en la página de inicio de VS.NET la configuración de teclado de VB6, la mayoría de las opciones en cuanto a tipo de letra y color de los elementos del editor de código estarán ajustadas correctamente. No obstante, es posible modificar cualquiera de estos valores si queremos realizar una configuración más personalizada todavía.

Por ejemplo, si el tipo de letra no es de nuestro agrado, seleccionaremos la opción *Herramientas + Opciones*, que abrirá la ventana Opciones. Dentro de ella, en la carpeta Entorno, haremos clic en el elemento *Fuentes y colores*, que nos mostrará en la parte derecha la configuración de tipo de letra y colores para el editor de código fuente. Ver Figura 117.

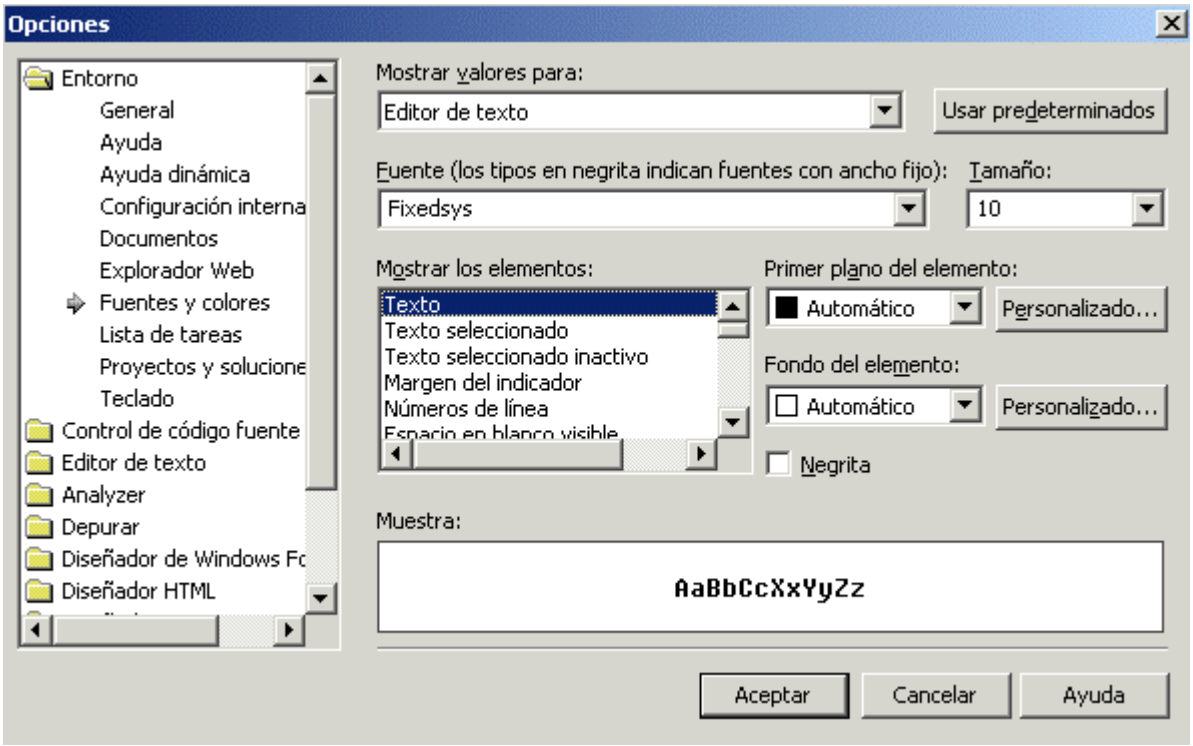


Figura 117. Configuración de fuente y colores para el editor de código.

Aquí podemos cambiar por ejemplo, el tipo de fuente a una de ancho fijo como Fixedsys, más cómoda para trabajar, y el color de algunos elementos de código, como los literales de error, comentarios, etc.

Aparte de estas opciones, la carpeta *Editor de texto* de esta misma ventana, nos permite configurar tanto aspectos generales del editor de código, como particulares para cada lenguaje. Ver Figura 118.

Entre las características del editor para VB.NET que podemos configurar, se encuentra el mostrar la lista de miembros automáticamente para los objetos, visualizar los números de línea, indentación de código, ancho de los tabuladores, finalización automática de estructuras de código, etc.

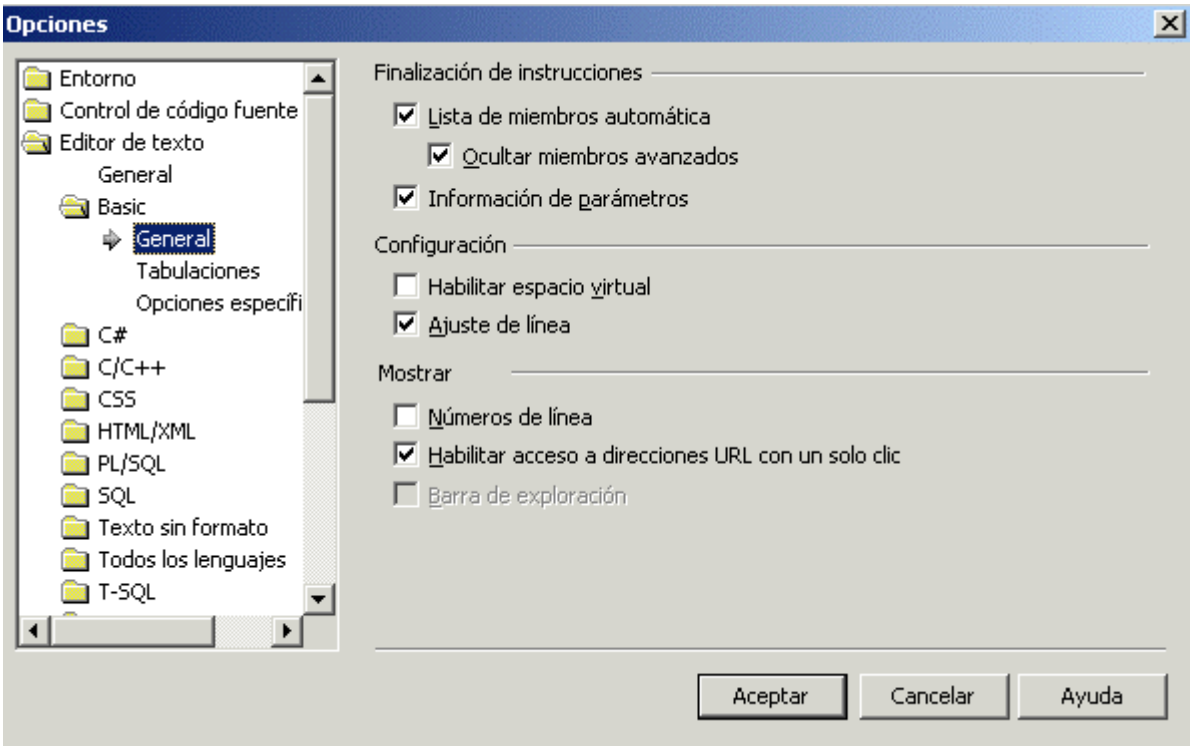


Figura 118. Opciones específicas para el editor de código del lenguaje VB.NET.

Números de línea

En la parte inferior derecha del IDE nos encontramos la barra de estado, que muestra el número de línea, columna y carácter sobre el que estamos situados actualmente. Si en algún momento necesitamos desplazarnos a un número de línea determinado dentro del documento actual, seleccionaremos la opción de menú *Edición + Ir a*, que mostrará la caja de diálogo de la Figura 119 para posicionarnos en una línea en concreto.

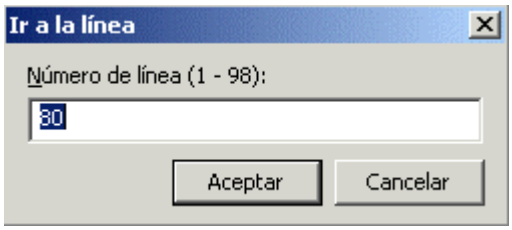


Figura 119. Desplazamiento a un número de línea.

Búsqueda y sustitución de código

Las opciones del menú *Edición + Buscar y reemplazar*, muestran diversas cajas de diálogo que nos permiten realizar una búsqueda de código con diferentes ámbitos: procedimiento actual, módulo actual, todo el proyecto, etc., y opcionalmente, sustituir el texto hallado por otro.

Supongamos que en nuestro programa queremos hacer una búsqueda de la palabra Total. Para ello seleccionaremos de la opción de menú antes mencionada, la subopción Buscar, o pulsaremos [CTRL+F], que nos mostrará la caja de búsqueda de la Figura 120.

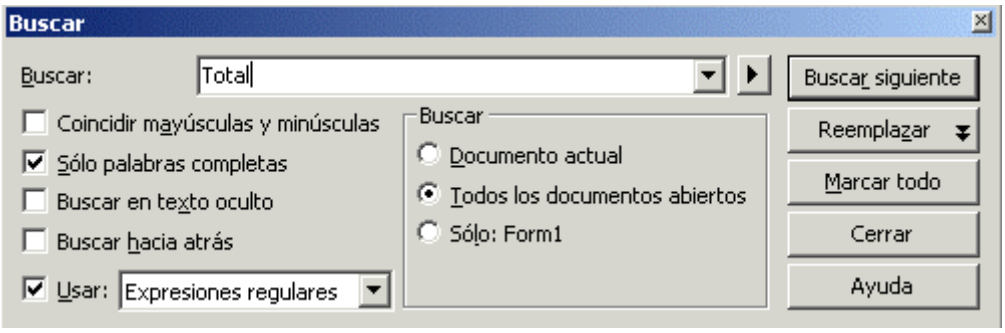


Figura 120. Caja de diálogo para buscar texto en el editor de código.

En esta ventana por un lado, introduciremos el texto en el campo Buscar, después podemos indicar al alcance o ámbito de nuestra búsqueda: el documento actual, todos los documentos, o el procedimiento de código en el que estábamos situados cuando hemos invocado la búsqueda.

Podemos realizar una búsqueda con un alto nivel de precisión, marcando alguna de las opciones que permiten buscar sólo por palabras completas, distinguiendo entre mayúsculas y minúsculas, por expresiones regulares, etc.

Al pulsar el botón *Buscar siguiente*, se comenzará a realizar la búsqueda según los valores establecidos, marcándose el texto hallado cada vez que la búsqueda tenga éxito.

Si pulsamos el botón Reemplazar, cambiará ligeramente el aspecto de la ventana, permitiendo al usuario introducir un texto, con el que reemplazaremos el que vamos a buscar. En la Figura 121 vamos a sustituir las ocurrencias de la palabra Total por la palabra Resultado; podemos ir buscando y reemplazando cada vez que localicemos el texto, o bien realizar una sustitución global en un solo paso.

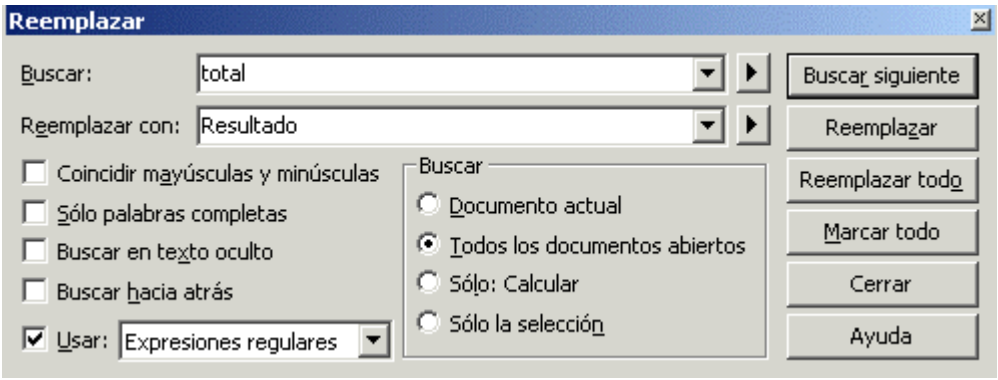


Figura 121. Ventana para búsqueda y reemplazo de texto.

La búsqueda por expresiones regulares es especialmente interesante. Podemos definir una expresión regular como un carácter comodín que nos permitirá realizar una búsqueda extendida. Si marcamos la casilla Usar para expresiones regulares, se habilitará el botón en forma de flecha que hay junto al campo Buscar. Al pulsar dicho botón se mostrará un menú con la lista de expresiones regulares disponibles. Ver Figura 122.

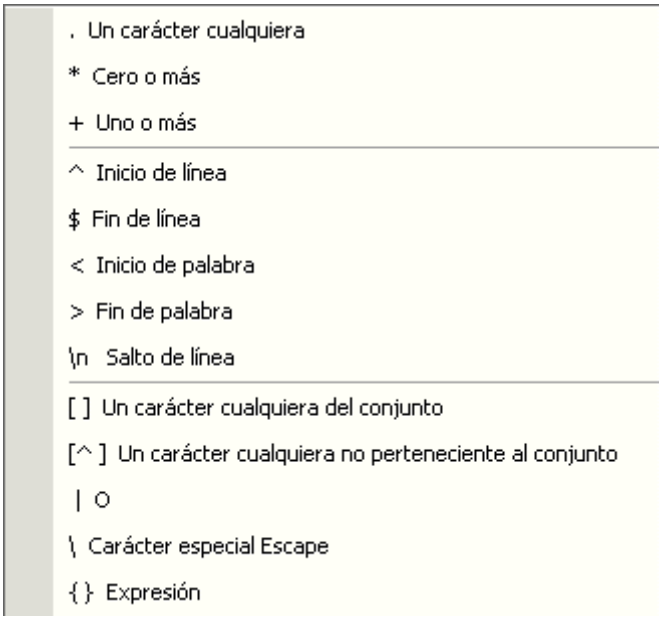


Figura 122. Menú de expresiones regulares para búsqueda de texto.

Un ejemplo del uso de expresiones regulares podría ser el siguiente: supongamos que queremos localizar todas las cadenas de texto que tengan las palabras *Calcular porcentaje*, y en medio de ambas que pueda haber una letra comprendida entre un intervalo. La expresión de búsqueda quedaría así: *Calcular [m-t] porcentaje*.

Otro tipo de búsqueda disponible lo encontramos en la opción de menú *Edición + Avanzadas + Búsqueda incremental*, o combinación de teclado [CTRL. + ALT + I] que una vez seleccionada, realiza una búsqueda dinámica del texto que vayamos introduciendo.

Ajuste de línea

Esta opción, que se encuentra en el menú *Edición + Avanzadas + Ajuste de línea*, si está activada, parte una línea de código muy larga en varias, de forma que no quede oculta en la ventana del editor. Si no está activada, se mantiene una única línea por muy larga que sea, de forma que parte puede quedar oculta. Ver Figura 123.

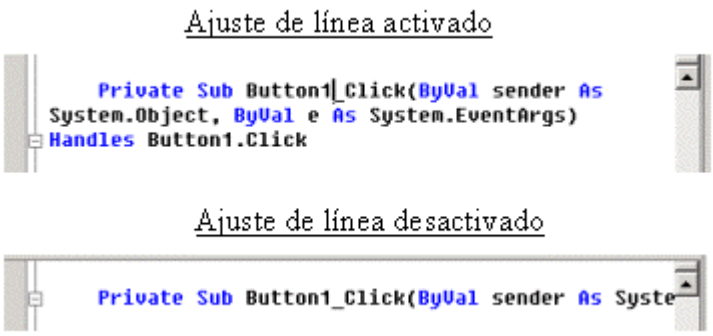


Figura 123. Línea de código con ajuste de línea activado y desactivado.

Activar esta característica puede resultar muy cómodo, ya que evita al programador tener que desplazarse hacia la derecha para ver el contenido de líneas de código muy largas.

Dividir el editor de código

Podemos encontrarnos en una situación en la que es muy importante visualizar en todo momento parte de un fragmento de código, pero la edición debemos realizarla en otro punto del documento diferente, por ejemplo: visualizar las variables de un procedimiento, que hemos declarado en su cabecera, pero editar la parte final del procedimiento.

Para solucionar este problema, podemos dividir horizontalmente la ventana del editor de código mediante la opción de menú Ventana + Dividir. De esta manera es posible disponer de dos vistas totalmente independientes del código. Ver Figura 124.

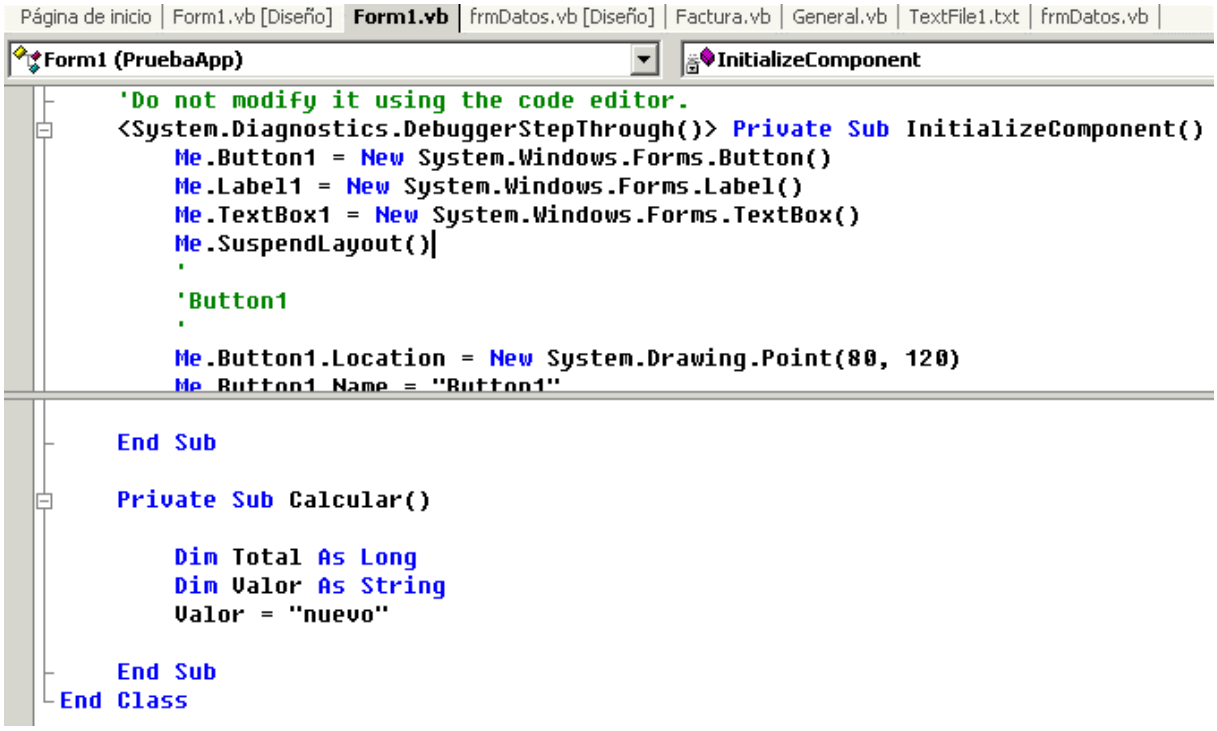


Figura 124. Editor de código dividido en dos paneles.

Para dejar esta ventana con un solo panel de edición seleccionaremos la opción de menú *Ventana + Quitar división*.

Otro modo de establecer la división del editor de código en dos paneles, consiste en hacer clic y arrastrar el indicador de división que se encuentra en la parte superior derecha de esta ventana, soltando en el punto que deseemos. Con esto quedará también dividida la ventana. Ver Figura 125.

Figura 125. Indicador de división del editor de código.

Marcadores

Un marcador consiste en una señal que situamos en una línea de código, de manera que podamos volver rápidamente a ella, sin necesidad de estar buscándola. Esta característica resulta especialmente útil cuando trabajamos con documentos de código muy grandes.

Para establecer un marcador, nos situaremos en la línea a marcar, y seleccionaremos la opción de menú *Edición + Marcadores + Alternar marcador*, o la combinación de teclado [CTRL + K, CTRL + K] Esta acción establecerá la marca correspondiente en el margen del editor de código, consistente en un semicírculo azul. Ver Figura 126.

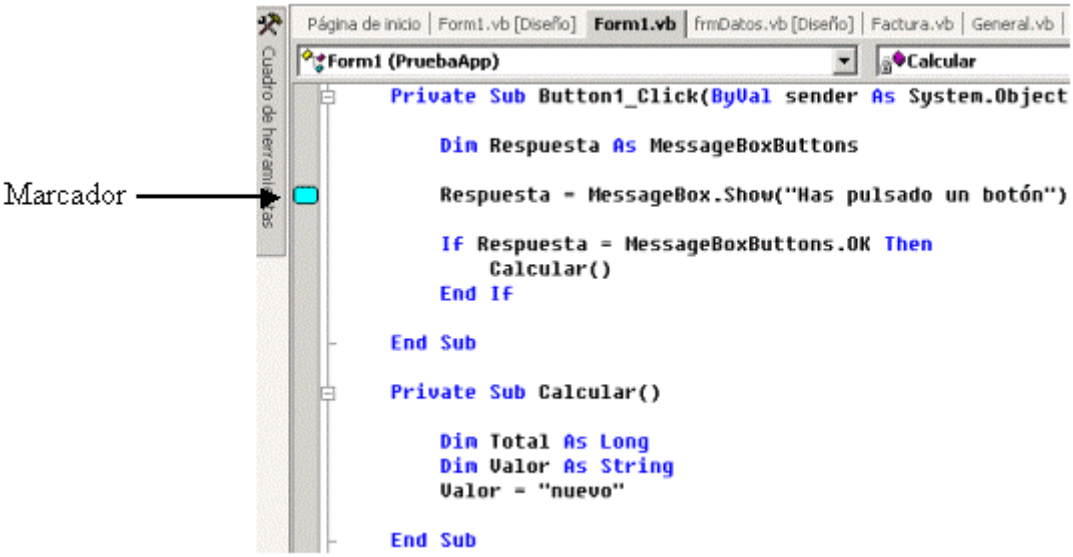


Figura 126. Marcador establecido en el editor de código.

Una vez que hayamos establecido todos los marcadores que consideremos oportunos, podemos desplazarnos de uno a otro con las opciones del menú *Edición + Marcadores*: *Marcador siguiente* o *Marcador anterior*, que corresponden respectivamente a las pulsaciones de teclado [CTRL + K, CTRL + N] y [CTRL + K, CTRL + P].

El desplazamiento entre marcadores se realizará en la ventana del editor actual, es decir, si tenemos varias ventanas de edición de código abiertas, con marcadores también establecidos en ellas, no podremos pasar desde el último marcador de una ventana al primer marcador de otra.

Para eliminar todos los marcadores establecidos en el editor actual, seleccionaremos la opción de menú *Edición + Marcadores + Borrar marcadores*, o la combinación de teclas [CTRL + K, CTRL + L].

Mostrar espacios en blanco

La opción de menú Edición + Avanzadas + Ver espacios en blanco, o combinación de teclado [CTRL + R, CTRL + W], permite mostrar u ocultar un punto en el lugar en el que existe un espacio en blanco, dentro del código fuente. Ver Figura 127.

```
Página de inicio | Form1.vb [Diseño] | Form1.vb | frmDatos.vb [Diseño] | Factura.vb | General.vb | TextFile1.t
Form1 (PruebaApp)                                  Button1_Click

    Private Sub Button1_Click(ByVal sender As System.Object, ByVal
EventArgs) Handles Button1.Click

        Dim Respuesta As MessageBoxButtons

        Respuesta = MessageBox.Show("Has pulsado un botón")

        If Respuesta = MessageBoxButtons.OK Then
            Calcular()
        End If

    End Sub

    Private Sub Calcular()

        Dim Total As Long
        Dim Valor As String
```

Figura 127. Visualizar un punto en la posición de un espacio en blanco.

Esquematización

La Esquematización u Outlining consiste en una característica del editor por la cual podemos expandir o contraer bloques de código, facilitando su lectura.

Cuando la esquematización se encuentra activada (estado por defecto), se muestra una línea o guía en el lateral izquierdo del editor, que discurre paralela al código. Ver Figura 128.

```
Private Sub Comprobar(ByVal Dato As Integer)

    Dim Resultado As Boolean

    If Dato = 2000 Then
        Resultado = True

    Else
        Resultado = False

    End If

End Sub
```

Figura 128. Editor de código con guía de esquematización.

La guía dispone a lo largo del código de una serie de símbolos más (+), y menos (-), que permiten al hacer clic, expandir o contraer respectivamente el código. Cuando hay una parte del código oculto de esta manera, se muestran además unos puntos suspensivos. Ver Figura 129.

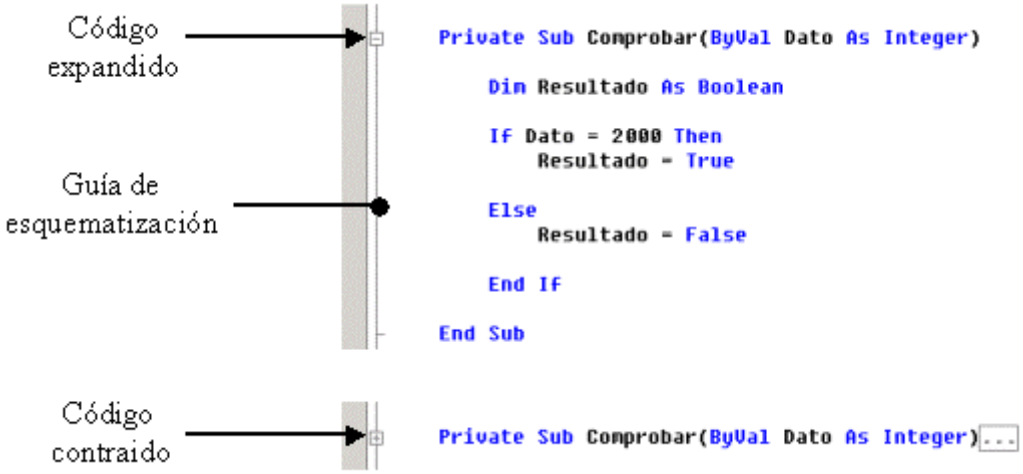


Figura 129. Código expandido y contraído mediante esquematización.

Podemos manipular la esquematización a través de las opciones de menú contenidas en *Edición + Esquematización*, que describimos a continuación de forma breve:

- **Ocultar selección**. [CTRL + M, CTRL + H].Contrae el texto seleccionado en el editor.
- **Alternar expansión de esquematización**. [CTRL + M, CTRL + M]. Expande o contrae el procedimiento de código en el que estamos situados.
- **Alternar toda la esquematización**. [CTRL + M, CTRL + L]. Expande o contrae todo el código existente en el editor.
- **Detener esquematización**. [CTRL + M, CTRL + P]. Elimina la esquematización del documento actual, suprimiendo la guía.
- **Detener ocultar actual**. [CTRL + M, CTRL + U]. Muestra una selección de texto que previamente habíamos contraído.
- **Contraer a definiciones**. [CTRL + M, CTRL + O]. Contrae cada procedimiento existente en el editor.
- **Iniciar esquematización automática**. Reinicia la esquematización que previamente habíamos detenido en el editor.

Regiones

Mediante las directivas del compilador #Region...#End Region, podemos definir bloques de código, que comprendan varios procedimientos, y que a través de esquematización, ocultemos o expandamos colectivamente. Cada región de código debe tener un nombre, que estableceremos en una cadena de caracteres a continuación de la palabra clave Region. Ver Figura 130.

```
#Region "Mis procedimientos"

    Private Sub Imprimir()
        '.....
        '.....
    End Sub

    Private Function Grabar() As String
        '.....
        '.....
        Return "valor"
    End Function

#End Region
```

Figura 130. Región de código definida en el editor.

Al crear un formulario con el diseñador de VS.NET, su editor de código correspondiente añade una región con el nombre "Windows Form Designer generated code", que contiene el código que genera automáticamente el diseñador.

Comentarios de código en bloque

Cuando necesitamos escribir un comentario muy extenso que ocupa varias líneas, o comentar un bloque de código, resulta muy incómodo hacerlo línea a línea. Para evitar este problema podemos seleccionar varias líneas, y mediante las opciones del menú *Edición + Avanzadas*: *Selección con comentarios* ([CTRL + K, CTRL + C]) y *Selección sin comentarios* ([CTRL + K, CTRL + U]), comentar y quitar comentarios respectivamente al bloque seleccionado.

Estas opciones también están disponibles en la barra de herramientas del editor de texto, cuyos botones se muestran en la Figura 131.

Figura 131. Botones para comentar y quitar comentarios del código fuente.

Ir a la definición de un procedimiento

Al hacer clic derecho sobre la llamada a un procedimiento, se muestra un menú contextual, en el que al seleccionar la opción *Ir a definición*, nos sitúa en el código del procedimiento seleccionado.

IntelliSense

Las opciones del menú *Edición + IntelliSense*, suponen una gran ayuda para el programador a la hora de escribir código, facilitando la labor de buscar miembros de clases, informando sobre los parámetros de procedimientos, etc.

A continuación realizaremos un rápido repaso de cada una de estas opciones:

- **Lista de miembros**. [CTRL + J]. Abre una lista que muestra los miembros del identificador seleccionado en el editor: clase, enumeración, etc.

  Nos encontramos en la siguiente situación: hemos creado una aplicación Windows con el típico formulario que se encuentra en una clase con el nombre Form1. En un procedimiento declaramos una variable con el nombre oVentana, de tipo Form1. Cuando escribamos esta variable y pongamos un punto, o bien ejecutemos esta opción de menú, aparecerá la lista de miembros de la clase Form1. Ver Figura 132.

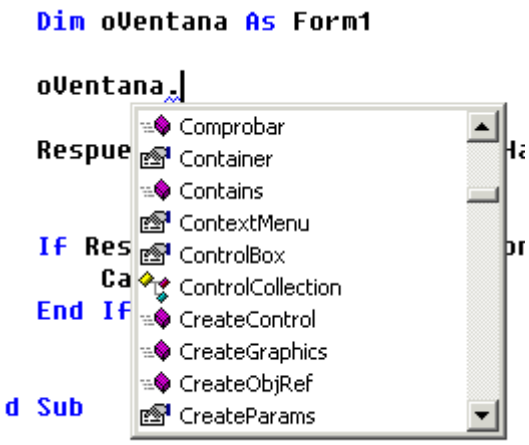


Figura 132. Lista de miembros de un elemento de código.

- **Información de parámetros**. [CTRL + MAYÚS + I]. Al hacer una llamada a un procedimiento, esta opción muestra una viñeta informativa con un resumen de la lista de parámetros necesarios para un procedimiento del programa.

En el siguiente ejemplo realizamos una llamada a un procedimiento llamado Comprobar( ), que recibe como parámetro un valor numérico Integer. Al aplicar esta opción se muestra la Figura 133.

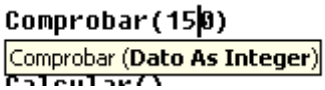


Figura 133. Información de parámetros para un procedimiento.

La definición de este procedimiento se muestra en el Código fuente 30.

```
Private Sub Comprobar(ByVal Dato As Integer)

      Dim Resultado As Boolean

      If Dato = 2000 Then
            Resultado = True
      Else
            Resultado = False
      End If

End Sub
```

Código fuente 30. Procedimiento Comprobar( ).

- **Información rápida**. [CTRL + I]. En función del elemento de código sobre el que sea aplicada, esta opción muestra una viñeta informativa.

Si por ejemplo, ejecutamos esta opción sobre la llamada al método Show( ) de un objeto MessageBox, se muestra la información de la Figura 134. Esto indica que el método requiere un parámetro de tipo String.

```
Respuesta = MessageBox.Show("Has pulsado un botón")
                            String
```

Figura 134. Información rápida.

- **Palabra completa**. [CTRL + ESPACIO]. Ayuda a completar la escritura de una instrucción. Si por ejemplo, comenzamos a escribir en el editor la palabra mess y seleccionamos esta opción, se abrirá una lista de valores en el que se mostrará el valor más parecido. En esta lista podremos ya elegir la instrucción que necesitemos. Ver Figura 135.

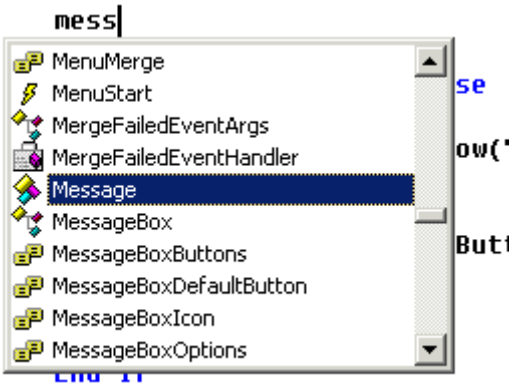


Figura 135. Lista de valores de la opción Palabra completa.

Cambiar a mayúsculas y minúsculas

Las opciones del menú *Edición + Avanzadas*: *Poner en mayúsculas* ([CTRL + MAYÚS + U]) y *Poner en minúsculas* ([CTRL + U]), tal y como indica su nombre, convierten a mayúsculas o minúsculas respectivamente el texto seleccionado en el editor.

10

El IDE de Visual Studio .NET. Elementos complementarios y ayuda

Editores de imágenes

VS.NET permite la creación o modificación de ficheros de imagen en los formatos más usuales: BMP, GIF, JPG, ICO, etc.

Para crear una nueva imagen y añadirla al proyecto, seleccionaremos la opción de menú *Proyecto + Agregar nuevo elemento*, eligiendo en la ventana siguiente, la plantilla *Archivo de mapa de bits*, lo que añadirá una nueva ficha a la ventana principal del IDE con las herramientas adecuadas para dibujar una imagen. Ver Figura 136.

Si en lugar de agregar un nuevo elemento, seleccionamos un fichero de imagen o icono existente, dicho ficheros se añadirá al proyecto, mostrándose en el editor de imagen permitiendo su modificación. La Figura 137 muestra esta situación con un icono.

Al abrir este editor, se muestra también automáticamente la barra de herramientas para la edición de imágenes, que dispone de los elementos necesarios para este trabajo. Ver Figura 138.

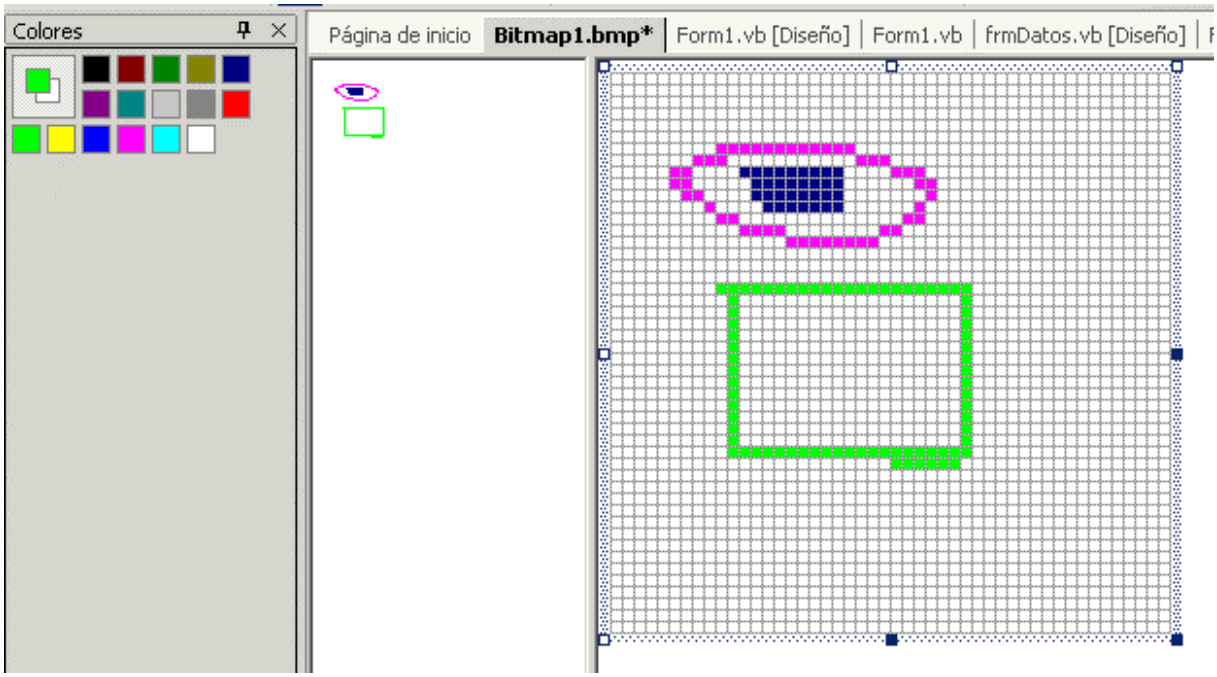


Figura 136. Editor de imágenes de VS.NET.

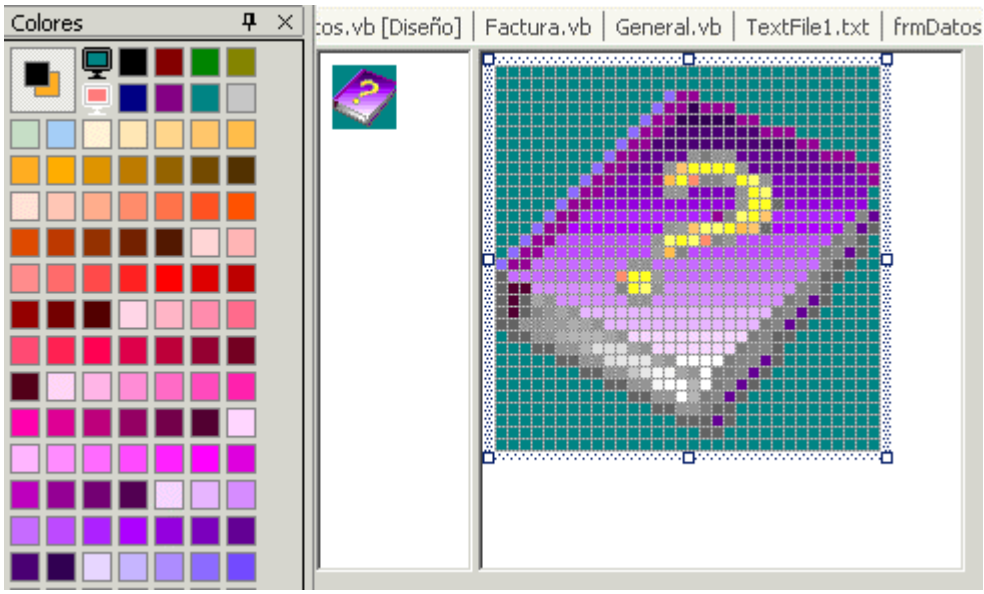


Figura 137. Edición de una imagen existente.

Figura 138. Barra de herramientas Editor de imágenes.

Lista de tareas

Cuando desarrollamos una aplicación, van apareciendo algunas labores que no podemos completar en el momento y debemos posponer para otra ocasión. Cada programador lleva un control más o menos efectivo de esta lista de labores incompletas: apuntarlas en un fichero con el Bloc de notas, una libreta

de apuntes, etc., que cumplen correctamente su misión, si bien, ninguna de ellas está integrada con la herramienta de programación.

VS.NET incorpora un novedoso elemento denominado tarea, que permite definir labores pendientes de realizar en el código de la aplicación, asignarles una prioridad y realizar búsquedas entre todas las definidas en el programa.

En el contexto del IDE, una tarea es un identificador simbólico, que al ser incluido en una línea de comentario es detectado de forma especial, engrosando la lista de tareas pendientes de realizar en el proyecto.

Definición de símbolos para tareas

Al instalar VS.NET se crea un conjunto de símbolos de tareas predeterminado, a los que podemos acceder a través de la opción de menú *Herramientas + Opciones*. En la ventana Opciones, abriremos la carpeta Entorno, y haremos clic sobre *Lista de tareas*. Ver Figura 139.

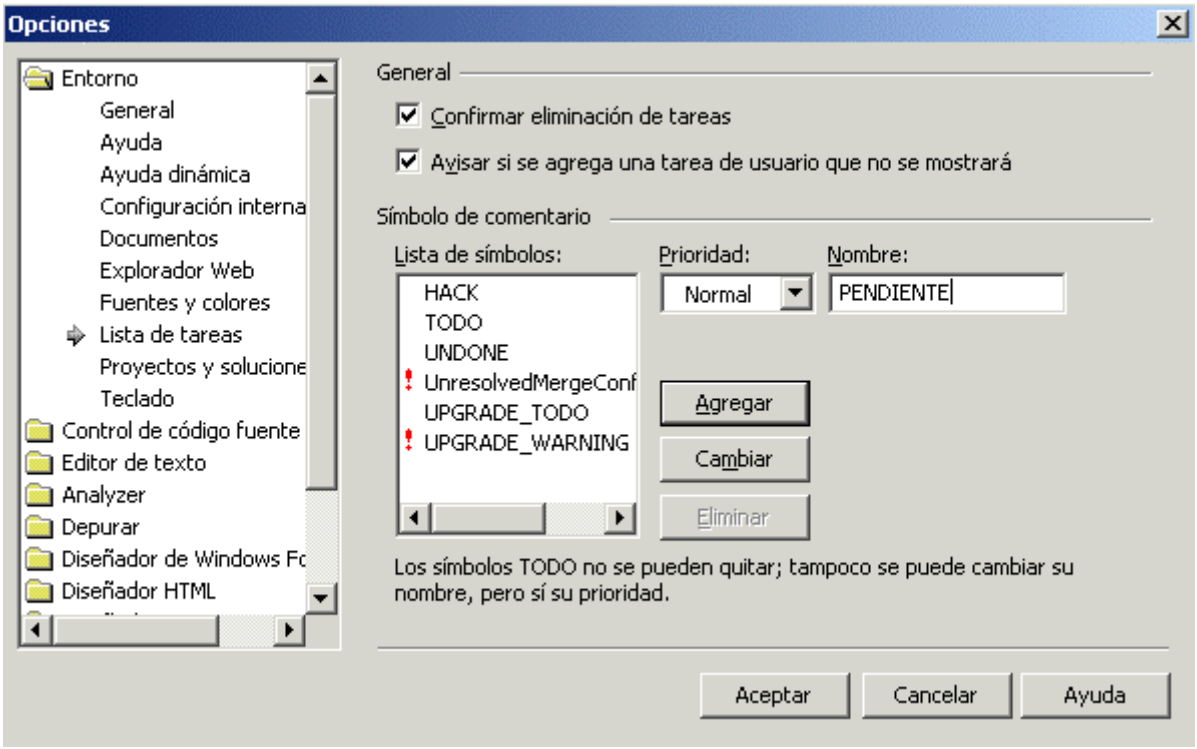


Figura 139. Definición de identificadores de tareas en el IDE.

En esta pantalla de configuración, podemos alterar los valores de los símbolos asociados a tareas, crear nuestros propios símbolos y eliminar todos excepto el símbolo TODO. En este caso, en el campo Nombre escribiremos un nuevo símbolo con el nombre PENDIENTE, y pulsaremos el botón Agregar, para añadirlo a la lista existente.

Creación de tareas

Podemos crear tareas de dos formas: asociando una tarea a un símbolo o directamente a una línea ejecutable.

En el primer caso, una vez creados o configurados los símbolos de tareas, vamos a añadir una tarea en el código fuente, de tipo PENDIENTE, el que acabamos de crear. Situados en un procedimiento, añadiremos el comentario mostrado en el Código fuente 31.

```
Private Sub Calcular()
    Dim Total As Long
    Dim Valor As String

    Total = 500

    ' PENDIENTE: agregar parámetros a este procedimiento
    Valor = "nuevo"

End Sub
```

Código fuente 31. Tarea PENDIENTE añadida en un procedimiento.

De igual forma que hemos agregado esta tarea, podemos utilizar los otros tipos definidos en el IDE, como son TODO, HACK, etc.

En el segundo caso, haremos clic derecho sobre una línea de código y seleccionaremos la opción de menú *Agregar acceso directo a la lista de tareas*, que situará una marca en forma de flecha en el margen del editor de código. Esta opción también está disponible en el menú *Edición* + *Marcadores*, o con la combinación de teclas [CTRL + K, CTRL + H]. Ver Figura 140.

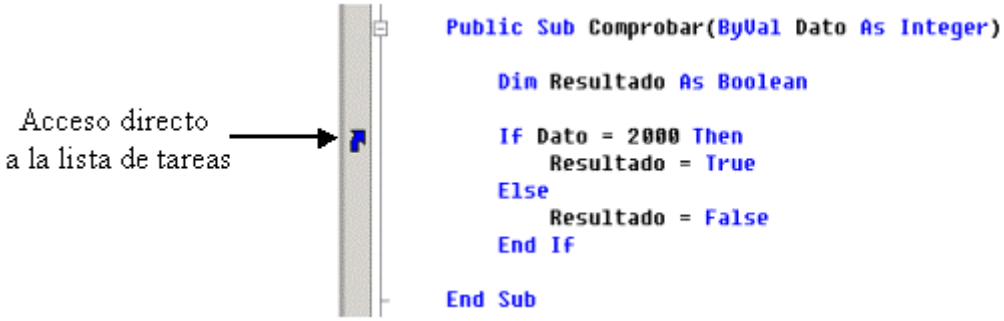


Figura 140. Acceso directo a la lista de tareas.

Ventana Lista de tareas

Podemos situarnos en las tareas definidas utilizando esta ventana a la que accederemos con la opción de menú *Ver* + *Otras ventanas* + *Lista de tareas*, la combinación de teclado [CTRL + ALT + K], o haciendo clic sobre su ficha, que habitualmente se halla en la parte inferior del IDE. Ver Figura 141.

Podemos desplazarnos a una tarea haciendo doble clic sobre la misma, lo que nos situará en el documento de código y línea especificados por la tarea.

La lista de tareas muestra por defecto todos los tipos de tareas creados. Si queremos establecer un filtro sólo por determinadas tareas, seleccionaremos alguna de las opciones del menú *Ver* + *Mostrar tareas*: Comentario, Usuario, Directiva, Todas, etc.

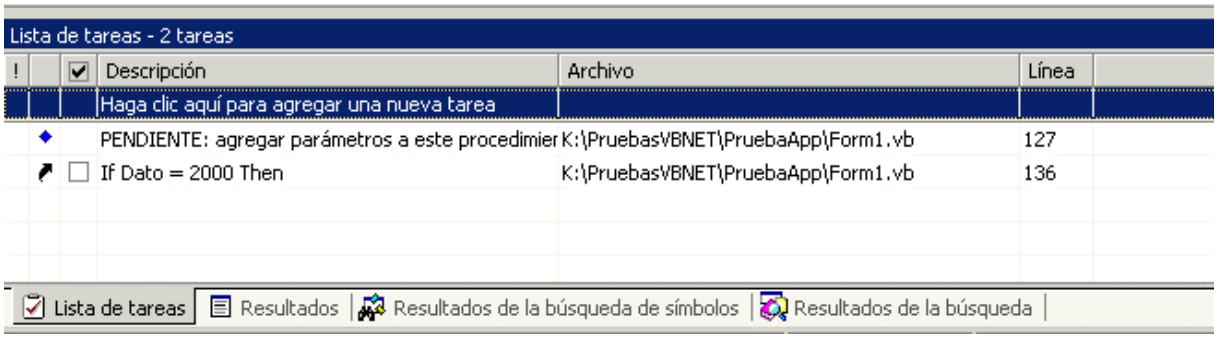


Figura 141. Ventana Lista de tareas.

Eliminación de tareas

Cuando consideremos una tarea completada podemos eliminarla de la lista. El modo de quitar una tarea depende de su tipo:

- Si se trata de una tarea asociada a una línea de código ejecutable, haremos clic sobre ella en la ventana de tareas y pulsaremos [SUPR]. O bien, haremos clic derecho y elegiremos la opción de menú Eliminar.

- Si la tarea está asociada a un símbolo situado en un comentario de código, bastará con borrar la línea del comentario.

Mostrar la pantalla completa

Dada la gran cantidad de elementos existentes en el entorno de desarrollo, y puesto que en ocasiones no es necesario disponer de todos ellos simultáneamente, la opción de menú *Ver + Pantalla completa*, o de teclado [MAYÚS + ALT + INTRO], amplia el espacio ocupado por el elemento con el que estemos trabajando en ese momento (por ejemplo, el editor de código), ocultando otros componentes del IDE, con lo cual, ganamos espacio de trabajo.

La Vista de clases

Esta ventana, a la que accedemos con la opción de menú *Ver + Vista de clases*, o la combinación de teclas [CTRL + MAYÚS + C] proporciona una panorámica de todos los módulos que componen el proyecto: clases, código estándar, etc. Ver Figura 142.

Organizada en forma de árbol jerárquico, podemos expandir los elementos que parten desde el proyecto, hasta llegar a los miembros de cada clase. Una vez expandida una clase, al hacer clic derecho sobre uno de sus métodos o propiedades, se muestra un menú contextual que nos permite ver el código del elemento seleccionado, su organización en el examinador de objetos, etc.

Por defecto, la ordenación de los miembros de las clases se realiza por tipo, aunque el primer botón de la barra de herramientas de esta ventana nos permite alterar dicho orden.

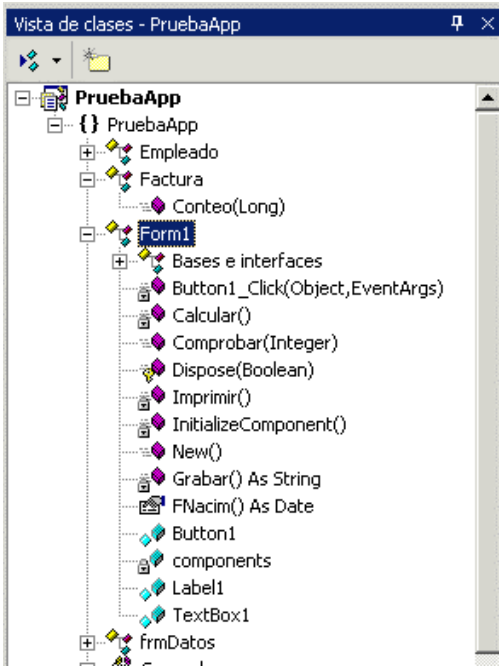


Figura 142. Ventana Vista de clases del IDE.

El Explorador de objetos

Muy relacionada con la vista de clases tenemos esta ventana, que abrimos con la opción de menú *Ver + Otras ventanas + Examinador de objetos*, o pulsando la tecla [F2]. Una vez abierta, se sitúa como una ficha más de la ventana principal del IDE, organizada en tres paneles principales.

El panel izquierdo muestra la organización de espacios de nombres, clases, etc. El panel derecho visualiza los miembros de la clase actualmente seleccionada. Finalmente, el panel inferior muestra la declaración del miembro seleccionado en el panel derecho. Ver Figura 143.

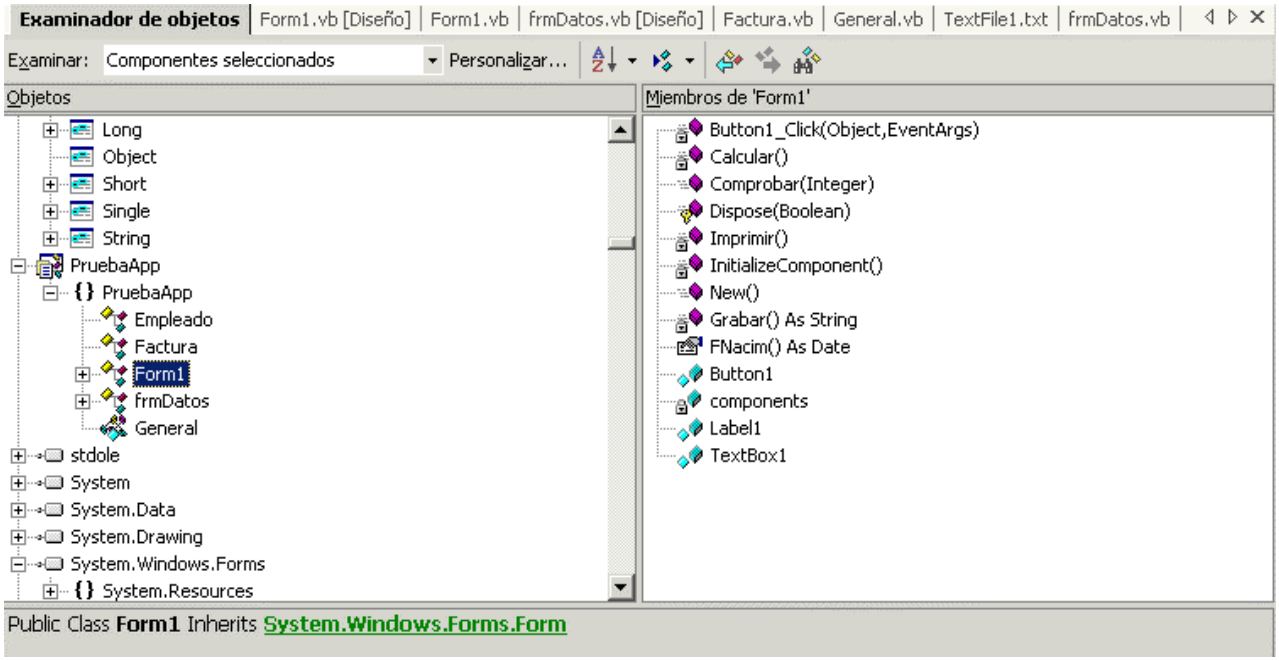


Figura 143. Examinador de objetos de VS.NET.

La diferencia respecto a la vista de clases, reside en que con el examinador de objetos podemos buscar información sobre cualquiera de las clases que componen la plataforma .NET Framework, pulsando sobre el último botón de la barra de herramientas: *Buscar símbolo*, ver Figura 144.

Figura 144. Botón para buscar un símbolo en el examinador de objetos.

Para buscar un símbolo dentro de las clases, se muestra una caja de diálogo en la que introducimos el nombre del símbolo. Al pulsar el botón Buscar, se abre la ventana Resultados, situada habitualmente en la parte inferior del IDE, con la lista de símbolos coincidentes encontrados. Al hacer doble clic sobre alguno de los símbolos encontrados, se actualiza la información del examinador de objetos, mostrando la clase y símbolo seleccionado.

Como ejemplo, en la Figura 145, hemos buscado por el símbolo Button, seleccionando de los valores resultantes el correspondiente a la clase Button, del espacio de nombres System.Windows.Forms.

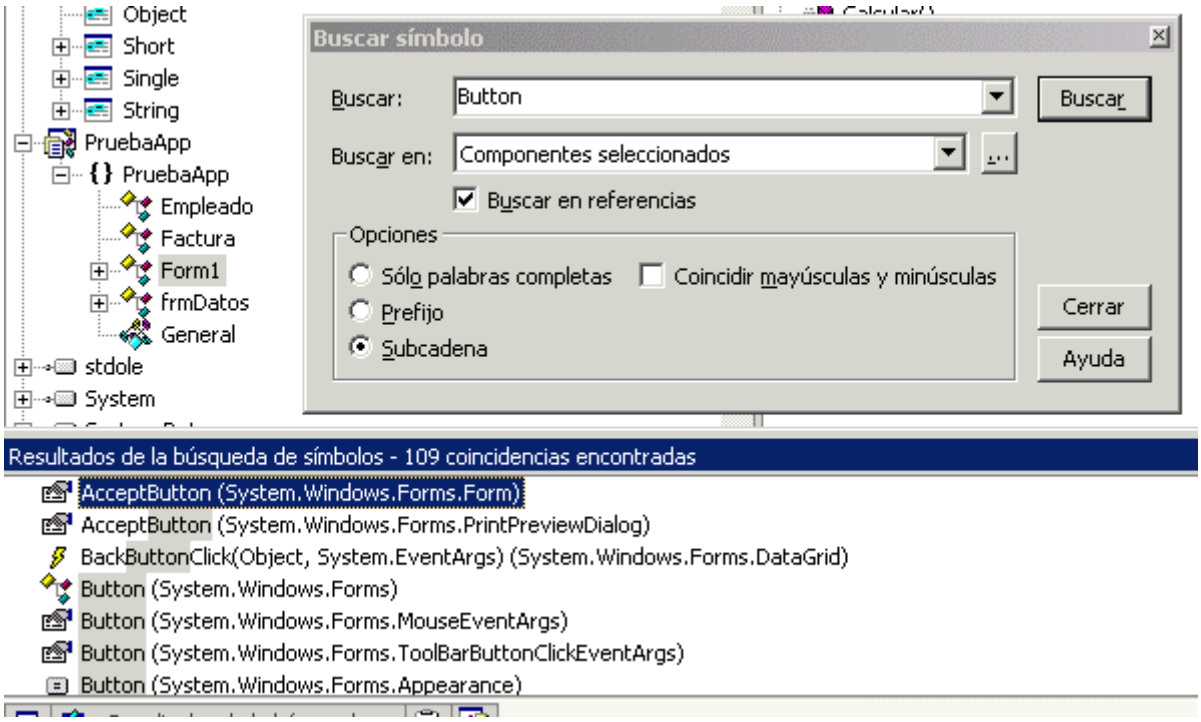


Figura 145. Búsqueda de símbolo en el examinador de objetos.

Macros

Dentro del contexto de las herramientas informáticas, una macro consiste en un conjunto de acciones, de las que habitualmente realizamos con dicha herramienta, que han sido grabadas para ser ejecutadas en un solo paso. Esto supone una gran ayuda al usuario tanto en tiempo como en esfuerzo, ya que unificamos en una sola operación, una lista de tareas que posiblemente tenga que realizar de forma repetitiva un gran número de veces durante una sesión de trabajo.

Otra de las ventajas de las macros radica en que por su peculiar naturaleza, deben poder ser creadas por el propio usuario, de manera que este defina las macros que mejor se adapten a su quehacer cotidiano.

Los lectores que sean usuarios de alguna de las herramientas de la familia Microsoft Office estarán a buen seguro familiarizados con las macros. En Microsoft Word por ejemplo, podemos crear una macro que seleccione el texto del documento que estamos escribiendo, cambie su tipo de letra y estilo. Todas estas operaciones las realizaríamos en un solo paso al ejecutar la macro.

VS.NET incorpora un conjunto de macros predefinidas, con algunas de las operaciones más habituales, así como un entorno adicional para la creación de nuestras propias macros.

Las macros realizan operaciones fundamentalmente sobre el código fuente, por ello recomendamos al lector que una vez creado un nuevo proyecto de VB.NET de tipo aplicación Windows, se posicione sobre una ventana de editor de código para poder comprobar mejor los resultados de las pruebas realizadas.

El Explorador de macros

El primer elemento del IDE que tenemos que utilizar para trabajar con macros es el *Explorador de macros*, al cual accedemos de alguna de las siguientes formas:

- Opción de menú *Ver* + *Otras ventanas* + *Explorador de macros*.
- Opción de menú *Herramientas* + *Macros* + *Explorador de macros*.
- [ALT + F8].

Esta ventana muestra inicialmente las macros definidas por defecto en el IDE. La organización se realiza en base a proyectos de macros, de una forma similar a los proyectos habituales de VB.NET. En el interior de cada proyecto encontramos un conjunto de módulos, dentro de los cuales se encuentran las macros. Ver Figura 146.

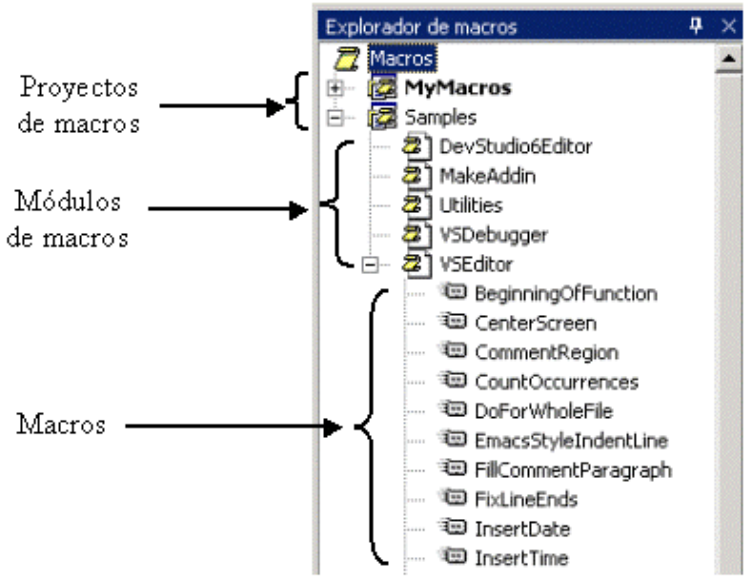


Figura 146. Explorador de macros de VS.NET.

Como podemos observar, partiendo del elemento Macros, y en forma de árbol, tenemos los proyectos predefinidos: Samples y MyMacros. Samples contiene un conjunto de macros ya creadas por defecto para el IDE, mientras que MyMacros es un proyecto que como su nombre indica, está pensado para añadir nuestras propias macros. Podemos asimismo, crear proyectos de macros adicionales.

Ejecución de macros

Una vez familiarizados con el explorador de macros, abriremos el proyecto Samples y su módulo VSEditor. Podemos ejecutar alguna de las macros de este módulo de las siguientes maneras:

- Haciendo clic sobre la macro y pulsando [INTRO].
- Haciendo doble clic sobre la macro.
- Haciendo clic derecho y eligiendo la opción Ejecutar de su menú contextual

Si por ejemplo, ejecutamos la macro InsertDate, se incluirá la fecha actual en la línea del editor en la que estemos posicionados, como muestra el ejemplo del Código fuente 32.

```
Public Function DevuelveImporte() As Long
    Dim Valores As Long

    Valores = 55477
' resultado de la ejecución de la macro InsertDate
jueves, 12 de julio de 2001
    Return Valores

End Function
```

Código fuente 32. Inserción de la fecha en el código con la macro InsertDate.

Grabación de macros

La operativa a seguir para grabar una macro es la misma que la utilizada en las herramientas de Microsoft Office, por lo que a los lectores usuarios de este grupo de aplicaciones, les resultará muy fácil la creación de macros en VS.NET. No obstante, aquellos usuarios no familiarizados con estas aplicaciones comprobarán también que se trata de un proceso muy sencillo.

Tomemos una situación simple: supongamos que con relativa frecuencia en nuestro código, debemos seleccionar un conjunto de líneas de comentario y convertirlas a mayúsculas. Esta operación implica un conjunto de elementales pasos, pero que si nos vemos a repetirlos muy frecuentemente, llegarán a convertirse en una molestia.

Para remediar este problema, grabaremos dichas acciones en una macro, de modo que cada vez que necesitemos realizarlas, ejecutando la macro conseguiremos el mismo resultado en un solo paso.

En primer lugar, situados en el explorador de macros, haremos clic en el proyecto MyMacros, y seleccionaremos la opción de menú *Establecer como proyecto de grabación*. Esto hará que las macros creadas a partir de ese momento se graben en dicho proyecto, dentro de su módulo RecordingModule.

A continuación, seleccionaremos la opción de menú *Herramientas + Macros + Grabar Temporary Macro*, o la combinación de teclas [CTRL + MAYÚS + R]. Esto iniciará la grabación de la macro, con lo que cualquier acción que hagamos a partir de ese momento quedará grabada.

Los pasos que grabaremos en la macro serán los siguientes: seleccionaremos una o varias líneas de comentarios en un editor de código, y después elegiremos la opción de menú *Edición + Avanzadas + Poner en mayúsculas*, que convertirá a mayúsculas las líneas seleccionadas.

Finalizaremos la grabación de la macro seleccionando la opción de menú *Herramientas + Macros + Detener grabación*, la combinación [CTRL + MAYÚS + R], o pulsando el segundo botón de la barra de herramientas de grabación de macros, que se muestra al comenzar el proceso de grabación. Ver Figura 147.

Figura 147. Barra de herramientas para grabación de macros.

La macro quedará grabada con el nombre predeterminado TemporaryMacro dentro del explorador de macros. Podemos cambiarle el nombre por otro más descriptivo, haciendo clic derecho sobre la macro y seleccionando la opción de menú *Cambiar nombre*. Le asignaremos por ejemplo el nombre PasarMay. El modo de ejecución es igual que el explicado en el apartado anterior.

Es muy recomendable cambiar el nombre a una macro recién grabada, ya que al grabar la siguiente, el IDE también le asignará el nombre TemporaryMacro, sobrescribiendo la macro que hubiera previamente.

Manipulación de proyectos de macros

Situados en el explorador de macros, al hacer clic derecho podemos realizar diversas acciones en función del elementos sobre el que estemos situados. A continuación describimos brevemente estas tareas agrupadas por elemento:

- **Macros**. Cargar un proyecto existente, crear un nuevo proyecto, abrir el IDE de macros.
- **Proyecto**. Cambiar el nombre del proyecto, añadir un nuevo módulo, descargar el proyecto, y establecer como proyecto de grabación.
- **Módulo**. Crear una nueva macro, editar el código del módulo, cambiar el nombre, y eliminar el módulo.
- **Macro**. Ejecutar, editar el código de la macro, cambiar su nombre, y eliminar.

Hasta el momento hemos trabajado con los dos proyectos que proporciona el IDE por defecto. Si queremos crear un nuevo proyecto para grabar macros, Haremos clic derecho sobre el elemento Macros, seleccionando la opción de menú *Nuevo proyecto de macros*, que mostrará la ventana de la Figura 148, en la que podremos seleccionar una carpeta del disco y un nombre para el nuevo proyecto.

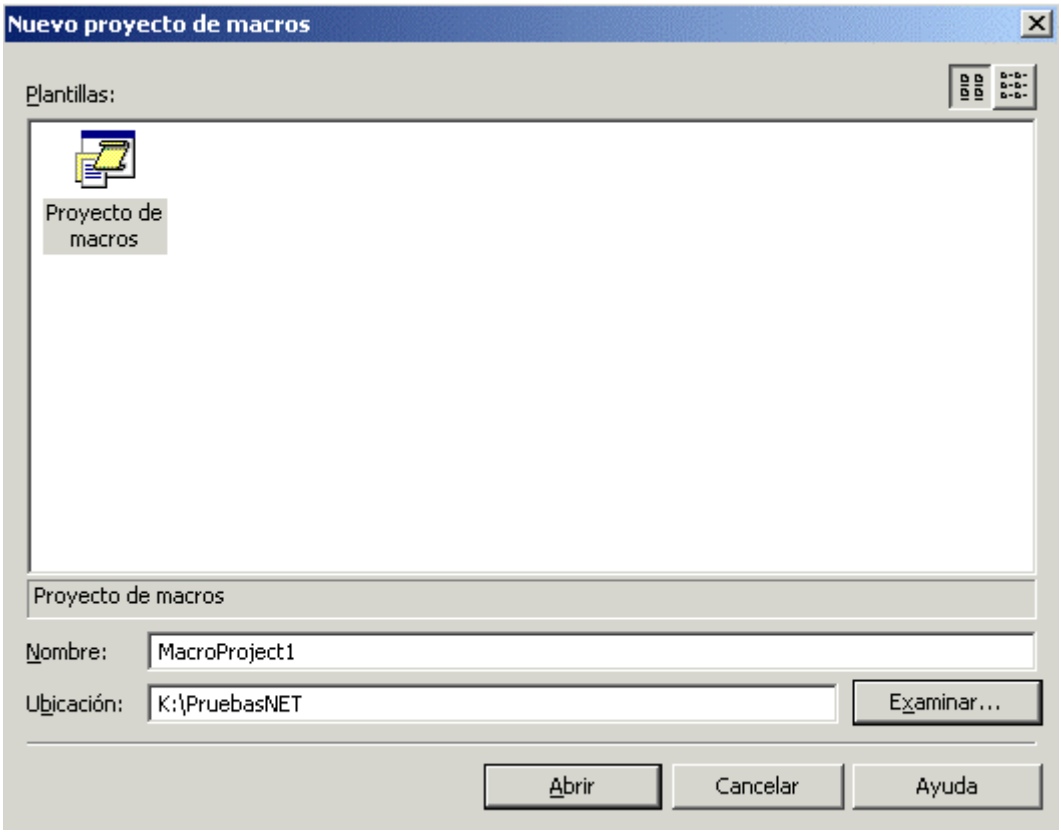


Figura 148. Ventana de creación de un proyecto de macros.

Como resultado, obtendremos un nuevo proyecto con su módulo correspondiente en el explorador de macros. Si queremos grabar macros en dicho proyecto, deberemos establecerlo como proyecto de grabación, operación anteriormente explicada.

El IDE de macros

Hasta el punto actual, imaginamos que todo habrá funcionado correctamente, pero probablemente el lector se pregunte, en el caso de la macro que ha grabado en un apartado anterior, qué clase de magia contiene la macro, que permite la repetición de las acciones antes grabadas en ella.

Para desvelar este misterio, debemos acceder al interior de la macro, lo que conseguimos a través de una versión reducida del IDE de VS.NET, especialmente diseñada para la creación y manipulación de macros denominada *IDE de macros*. Podemos abrir este entorno mediante alguna de las siguientes operaciones:

- Haciendo clic derecho sobre la macro y seleccionando la opción de menú Editar. Este es el medio más directo.
- Opción de menú *Herramientas* + *Macros* + *IDE de macros*.
- Pulsando las teclas [ALT + F11].
- Clic derecho sobre el elemento Macros del explorador de macros.

Como resultado se muestra la ventana de la Figura 149.

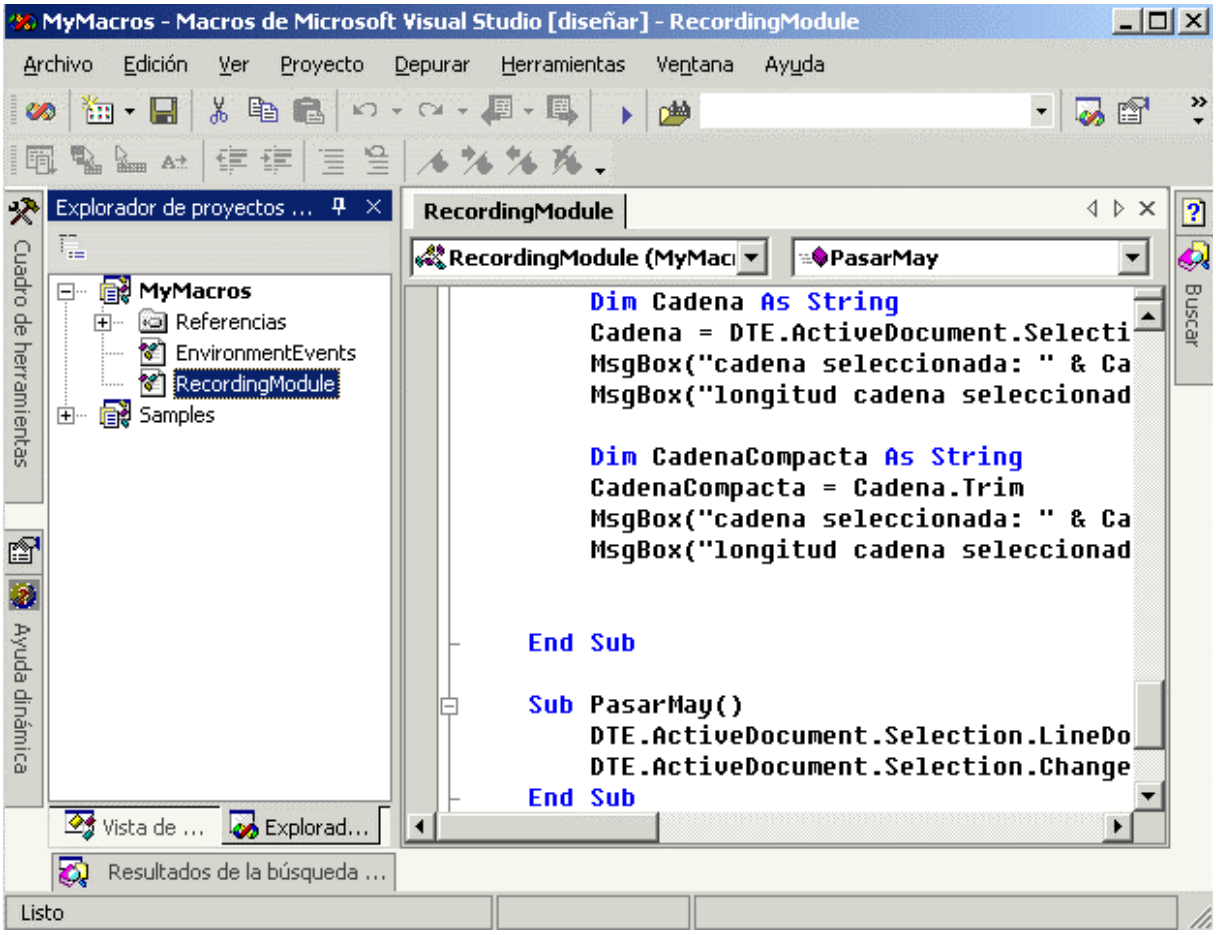


Figura 149. Ventana del IDE de macros.

Si hemos elegido editar directamente la macro PasarMay( ), en el editor de código de macros, podemos comprobar como una macro no es otra cosa que un procedimiento especial, que es ejecutado por VS.NET cada vez que invocamos la macro. Ver Código fuente 33.

```
Sub PasarMay()
    DTE.ActiveDocument.Selection.LineDown(True, 2)
    DTE.ActiveDocument.Selection.ChangeCase(vsCaseOptions.vsCaseOptionsUppercase)
End Sub
```

Código fuente 33. Código correspondiente a la macro PasarMay( ).

Escritura de macros

Cuando el lector vaya creando sus propias macros de prueba, observará como todas están basadas en el objeto DTE, que constituye el objeto principal para la manipulación del código fuente a través de macros.

Conociendo los métodos y propiedades de este objeto, podemos escribir nuestras propias macros en el editor de código del IDE de macros, para efectuar operaciones sobre el código fuente de nuestras aplicaciones como seleccionar, comentar, buscar, etc.

Sin embargo, el conocimiento del objeto DTE puede resultar algo oscuro para el programador si no dispone de la documentación de referencia adecuada, por lo que existe un truco para poder crear nuestras propias macros sin un entendimiento a fondo de dicho objeto.

Establezcamos la siguiente situación: necesitamos escribir una macro que comente un número de líneas de código variable cada vez que sea ejecutada, es decir, que cada vez que invoquemos esta macro, deberemos pedir al usuario, a partir de la línea de código en la que esté situado, cuantas líneas quiere comentar.

Lo primero que necesitamos saber es cómo debemos utilizar el objeto DTE para posicionarnos al principio de la línea de código y comentarla. Para averiguarlo, nada más fácil que crear una macro que haga dicha tarea. Ver Código fuente 34.

```
Sub TemporaryMacro()
    ' posicionar al principio de la línea (en la primera columna)
DTE.ActiveDocument.Selection.StartOfLine(vsStartOfLineOptions.vsStartOfLineOptionsF
irstColumn)

    ' seleccionar la línea de código
    DTE.ActiveDocument.Selection.LineDown(True)

    ' comentar la línea seleccionada
    DTE.ExecuteCommand("Edit.CommentSelection")
End Sub
```

Código fuente 34. Macro para comentar la línea de código actual.

Ahora que sabemos como aplicar el proceso a una línea, escribiremos nuestra propia macro en el IDE de macros, a la que daremos el nombre ComentarBloqueCodigo, y que contendrá el código anterior con las variaciones necesarias para pedir al usuario el número de líneas a comentar, seleccionar dichas líneas, y finalmente comentarlas. Ver Código fuente 35.

```
Sub ComentarBloqueCodigo()
    ' selecciona el número de líneas introducidas
    ' por el usuario y luego las comenta
    Dim NumLineas As Integer
    Dim Contador As Integer

    ' pedir al usuario el número de líneas
    NumLineas = InputBox("Número de lineas a comentar", "Comentar bloque de
código")

    ' posicionar al principio de la línea actual
    DTE.ActiveDocument.Selection.StartOfLine( _
        vsStartOfLineOptions.vsStartOfLineOptionsFirstColumn)

    ' comenzar a seleccionar líneas
    For Contador = 1 To NumLineas
        DTE.ActiveDocument.Selection.LineDown(True)
    Next

    ' comentar las líneas seleccionadas
    DTE.ExecuteCommand("Edit.CommentSelection")

End Sub
```

Código fuente 35. Macro para comentar un número variable de líneas de código.

A partir de aquí, podemos ejecutar nuestra macro del modo explicado anteriormente, para comprobar que su funcionamiento es correcto. Ver Figura 150.

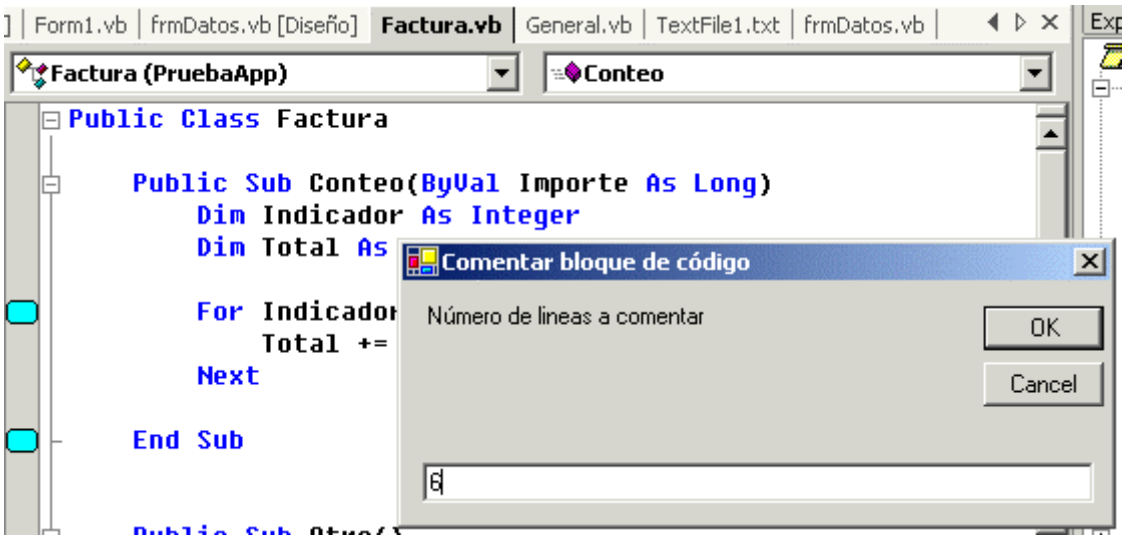


Figura 150. Macro en ejecución.

Macro para comentar líneas de código determinadas

Como ejemplo adicional, se proporciona al lector en el Código fuente 36 una macro con el nombre ComentarIdentificador, cuya misión consiste en comentar líneas de código dentro del procedimiento en el que estemos posicionados. La particularidad de esta macro reside en que sólo vamos a comentar las líneas en las que se halle una determinada palabra o identificador que introduciremos mediante InputBox( ). Los comentarios oportunos sobre cada uno de los pasos que realiza la macro se encuentran en el propio código fuente.

```
Sub ComentarIdentificador()
    ' pedir nombre del identificador a comentar
    Dim Identif As String
    Identif = InputBox("Introducir nombre a buscar", "Comentar identificador")

    ' si no se introduce ningún valor, salir
    If Identif.Length = 0 Then
        Exit Sub
    End If

    ' posicionar al comienzo del procedimiento
    Dim ts As TextSelection = DTE.ActiveWindow.Selection

ts.MoveToPoint(ts.ActivePoint.CodeElement(vsCMElement.vsCMElementFunction).GetStart
Point(vsCMPart.vsCMPartHeader))

    Dim LineaCodigo As String
    Do
        ' bajar a la siguiente línea de código
        DTE.ActiveDocument.Selection.LineDown()

        ' posicionar cursor al comienzo de la línea

DTE.ActiveDocument.Selection.StartOfLine(vsStartOfLineOptions.vsStartOfLineOptionsF
irstColumn)
```

```
        ' seleccionar la línea, pasar su contenido a una
        ' variable, y volver a posicionar cursor al principio
        ' de línea
        DTE.ActiveDocument.Selection.EndOfLine(True)
        LineaCodigo = DTE.ActiveDocument.Selection.Text
        LineaCodigo = LineaCodigo.Trim()  ' eliminar espacios en blanco

DTE.ActiveDocument.Selection.StartOfLine(vsStartOfLineOptions.vsStartOfLineOptionsF
irstText)

        ' si llegamos al final del procedimiento, finalizar macro
        If (LineaCodigo = "End Sub") Then
            Exit Do
        Else
            ' si no estamos al final, comprobar
            ' si existe el identificador en la línea actual
            Dim Posicion As Integer
            Posicion = LineaCodigo.IndexOf(Identif)

            ' si existe el identificador en la línea
            If Posicion > -1 Then
                ' comentar la línea

DTE.ActiveDocument.Selection.StartOfLine(vsStartOfLineOptions.vsStartOfLineOptionsF
irstText)
                DTE.ActiveDocument.Selection.EndOfLine(True)
                DTE.ExecuteCommand("Edit.CommentSelection")
            End If
        End If
    Loop

End Sub
```

Código fuente 36. Macro ComentarIdentificador( ).

El sistema de ayuda

Al igual que ha sucedido con multitud de elementos del IDE, la ayuda o MSDN Library (Microsoft Development Network Library) en esta versión de Visual Studio ha sido ampliamente mejorada. La aplicación utilizada para navegar por la documentación de la ayuda es Microsoft Document Explorer, que permite al programador una enorme flexibilidad a la hora de realizar consultas sobre la ayuda disponible de los diferentes productos de desarrollo que componen la plataforma .NET.

La integración ahora del sistema de ayuda con el IDE es total, siguiendo como es natural, con la tónica establecida en versiones anteriores de proporcionar ayuda contextual siempre que sea posible. Si por ejemplo, nos encontramos diseñando un formulario y hacemos clic sobre un control TextBox, al pulsar [F1] se abrirá una nueva pestaña en la ventana principal del IDE que iniciará la ayuda, y nos posicionará en un documento relativo a dicho control. Ver Figura 151.

La documentación de ayuda está creada en formato HTML, lo cual nos permite, gracias a su elevado grado de integración, guardar los documentos que visitemos dentro del apartado Favoritos de Internet Explorer, y consultarlos igualmente utilizando este navegador de Internet.

Página de inicio | **TextBox Members** | Form1.vb [Diseño] | Form1.vb | Factura.vb

.NET Framework Class Library

TextBox Members

Public Instance Constructors

TextBox Constructor	Initializes a new instance of the **TextBox** class.

Public Instance Properties

AcceptsReturn	Gets or sets a value indicating whether pressing ENTER in a multiline **TextBox** control creates a new line of text in the control or activates the default button for the form.
AcceptsTab (inherited from **TextBoxBase**)	Gets or sets a value indicating whether pressing the TAB key in a multiline text box control types a TAB character in the control instead of moving the focus to the next control in the tab order.
AccessibilityObject (inherited from **Control**)	Gets the AccessibleObject assigned to the control.
AccessibleDefaultActionDescription (inherited from **Control**)	Gets or sets the default action description of the control.

Figura 151. Página de ayuda de VS.NET.

Ayuda dinámica

Esta ventana muestra enlaces a los temas de ayuda del entorno .NET. Podemos acceder a ella mediante la opción de menú *Ayuda + Ayuda dinámica*, pulsando [CTRL + F1], o haciendo clic en su ficha desplegable situada en un lateral del IDE, que tendrá alguna de las formas mostradas en la Figura 152.

Figura 152. Distintas pestaña para acceso a la ayuda de VS.NET.

Tal y como su propio nombre indica, los enlaces que muestra esta ventana son activos y van actualizándose dependiendo del elemento de código u objeto del diseñador en el que estemos posicionados. La Figura 153 muestra un ejemplo de esta ventana cuando estamos posicionados en la palabra clave del lenguaje Property.

Al hacer clic sobre alguno de los enlaces de esta ventana, la ficha del IDE que muestra la ayuda se actualizará igualmente con el tema seleccionado.

Figura 153. Ventana Ayuda dinámica.

Contenido

Esta ventana muestra la documentación al completo de la plataforma .NET Framework organizada en áreas temáticas. Podemos abrirla de las siguientes formas:

- Haciendo clic en el primer botón de la barra de herramientas de la ventana *Ayuda dinámica* (icono con forma de libro).
- Situando el cursor encima de su ficha desplegable situada en el lateral del IDE.
- Menú *Ayuda* + *Contenido*.
- Teclas [CTRL + ALT + F1].

La ventana mostrada tendrá el aspecto de la Figura 154.

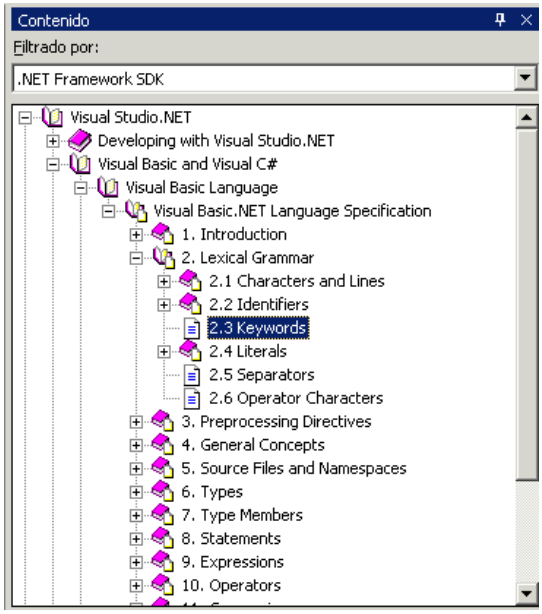


Figura 154. Ventana Contenido, de la ayuda.

El manejo de la ayuda con esta ventana se basa en expandir o cerrar los libros que muestra. Al hacer clic en uno de los iconos en forma de documento, se mostrará su contenido en la ventana de ayuda que tengamos abierta en el IDE.

Índice

Esta ventana nos permite realizar una búsqueda dinámica de un elemento dentro de la ayuda. Podemos acceder a ella de las siguientes formas:

- Haciendo clic en el segundo botón de la barra de herramientas de la ventana *Ayuda dinámica* (icono con forma de interrogación).
- Situando el cursor encima de su ficha desplegable situada en el lateral del IDE.
- Menú *Ayuda + Índice*.
- Teclas [CTRL + ALT + F2].

Según tecleamos un valor en el campo Buscar de esta ventana, se realizará una búsqueda dentro del MSDN, del valor más parecido a lo que hasta ese momento hemos tecleado. Podemos adicionalmente, seleccionar en la lista desplegable *Filtrado por*, un área para acotar la búsqueda Ver la Figura 155.

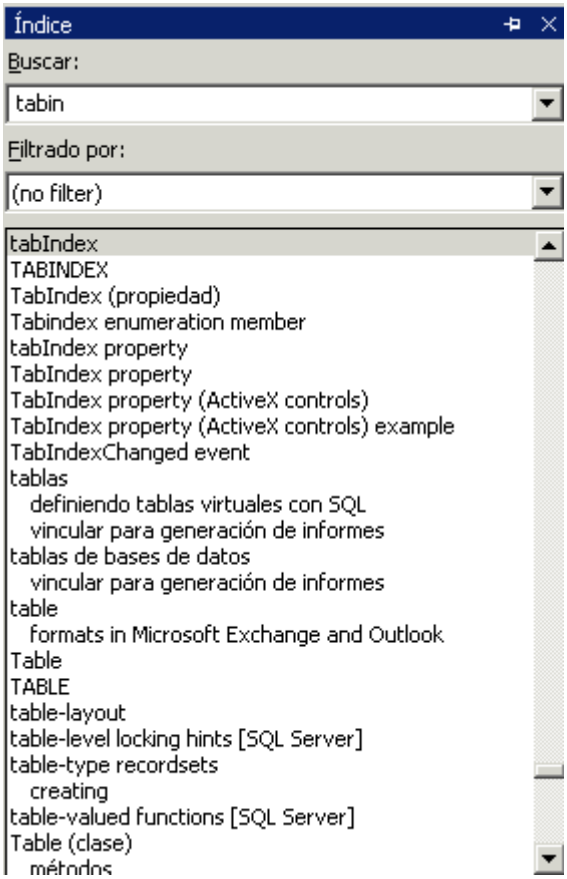


Figura 155. Ventana Índice de la ayuda.

Una vez que localicemos en la lista de elementos, el tema deseado, haremos clic sobre él y se mostrará su contenido en la ventana principal del IDE.

Buscar

En el caso de que la ayuda contextual o a través del índice no aporten la información necesaria, podemos utilizar la ventana Buscar, perteneciente a la ayuda, que explorará en toda la documentación del MSDN y devolverá una lista con los documentos que cumplan la condición de búsqueda. Esta ventana, ver Figura 156, está disponible de las siguientes formas:

- Haciendo clic en el tercer botón de la barra de herramientas de la ventana *Ayuda dinámica* (icono con forma de libros y lupa).
- Situando el cursor encima de su ficha desplegable situada en el lateral del IDE.
- Menú *Ayuda* + *Buscar*.
- Teclas [CTRL + ALT + F3].

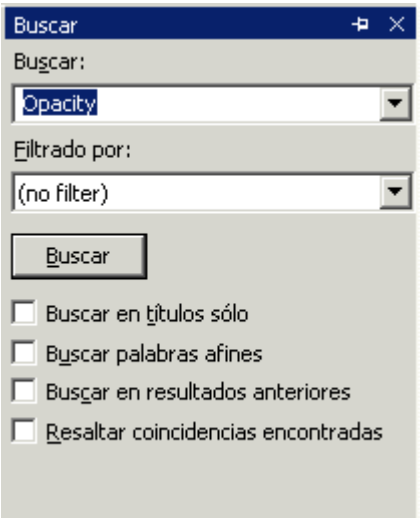


Figura 156. Ventana Buscar en la ayuda.

Al igual que en la ventana de índice, teclearemos un valor en el campo Buscar y estableceremos un filtro sobre un tema si lo necesitamos (no es obligatorio). Podemos restringir además la búsqueda, marcando las casillas correspondientes que limitan aspectos de la búsqueda.

Una vez pulsado el botón Buscar, la ventana Resultados de la búsqueda, mostrará una lista de todos los temas relacionados, ver Figura 157. Al hacer doble clic sobre uno de los elementos, se mostrará el tema en la ventana de ayuda correspondiente.

Resultados de la búsqueda de Opacity: 204 temas encontrados

Título	Ubicación	Jerarquía
Alpha Filter	Web Workshop	1
Filters and Transitions	Web Workshop	2
Style Property	Web Workshop	3
Visual Filters and Transitions Reference	Web Workshop	4
Opacity Property	Web Workshop	5
Opacity Property	Web Workshop	6
Filters and Transitions Interactive Demo	Web Workshop	7

Figura 157. Resultados de una búsqueda en la ayuda.

Ayuda externa

En el caso de que el lector esté más acostumbrado a trabajar con la ayuda como un elemento aparte del entorno de desarrollo, puede configurarla para que sea mostrada de forma externa al IDE de VS.NET.

Para ello debemos situarnos en la página de inicio del IDE y hacer clic en el vínculo *Mi perfil*. A continuación, en el apartado *Mostrar ayuda*, haremos clic en la opción *Ayuda externa*. Los cambios quedarán grabados, pero no se harán efectivos hasta la siguiente ocasión en que iniciemos el IDE.

A partir de ese momento, cuando invoquemos la ayuda en cualquiera de las maneras anteriormente descritas, se abrirá una nueva ventana perteneciente a la colección combinada de Visual Studio .NET, conteniendo el tema de ayuda seleccionado. Ver Figura 158.

Colección combinada de Visual Studio.NET - Button Members

Archivo Editar Ver Herramientas Ventana Ayuda

ms-help://MS.VSCC/MS.MSDNVS.3082/cpr

Button Members

.NET Framework Class Library

Button Members

Public Instance Constructors

Button Constructor	Initializes a new instance of the **Button** class.

Public Instance Properties

AccessibilityObject (inherited from **Control**)	Gets the AccessibleObject assigned to the control.
AccessibleDefaultActionDescription (inherited from **Control**)	Gets or sets the default action description of the control.

Resultados del índice de Stri...

Listo

Figura 158. Ventana de ayuda en ejecución externa al IDE de VS.NET.

La ventaja de usar la ayuda de forma externa reside en que al ejecutarse en su propia ventana, disponemos de más espacio para visualizar cada uno de los temas seleccionados.

Los mecanismos de búsqueda dentro de la ayuda están igualmente disponibles a través de las fichas desplegables situadas en el lateral, o el menú de esta ventana.

Mantener temas de ayuda disponibles

Durante una sesión de trabajo con VB.NET, puede ocurrir que al emplear la ayuda, entre todos los documentos consultados, haya uno en concreto al que necesitemos recurrir con especial frecuencia.

Para facilitar el trabajo con la ayuda en una situación como esta, una vez que hayamos localizado el tema de ayuda que consultaremos en repetidas ocasiones, abriremos una nueva ventana con el menú *Ventana + Nueva ventana*, y a partir de ese momento, toda la navegación por la ayuda que realicemos se reflejará en esa última ventana, permaneciendo el contenido de la otra ventana de ayuda con el tema original.

La Figura 159 muestra un ejemplo en el que dentro de la ventana de ayuda se han abierto dos ventanas o fichas: la primera comenzando por la izquierda contiene un tema que consultaremos en repetidas ocasiones; mientras que la segunda contiene la navegación por la ayuda que vamos realizando durante nuestro trabajo de desarrollo.

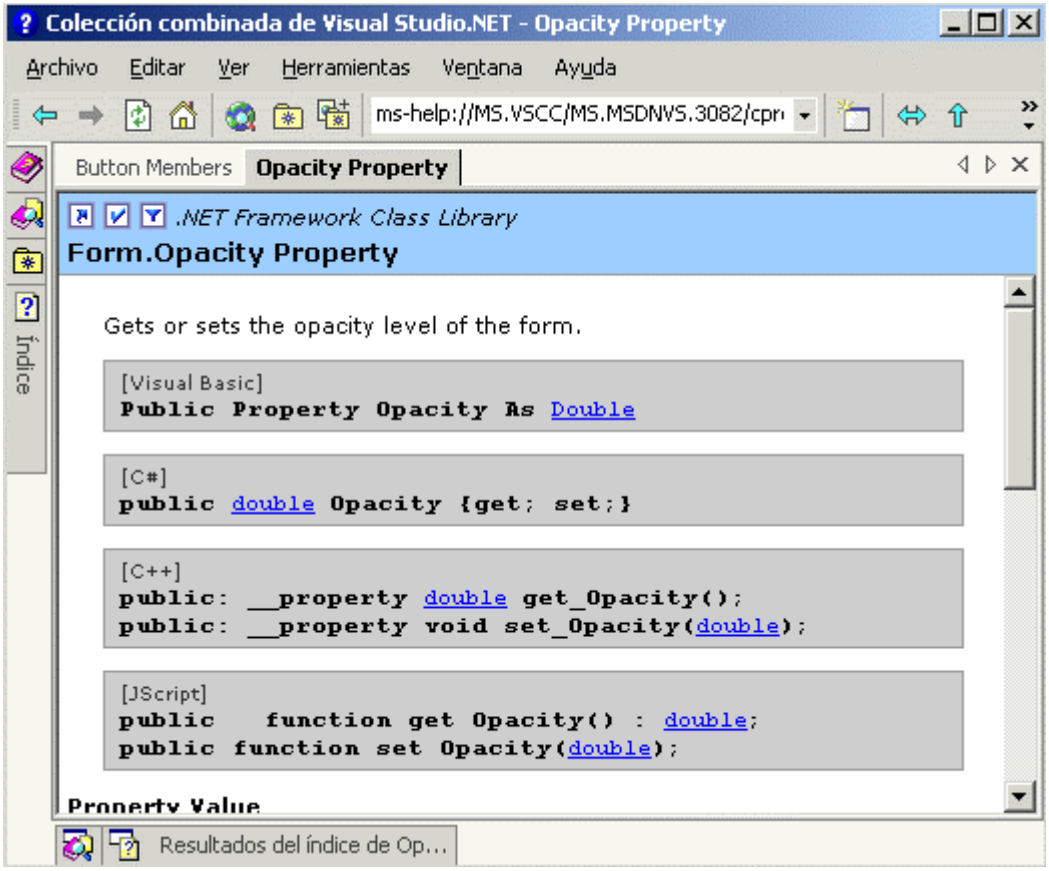


Figura 159. Ventana de ayuda con varias ventanas de temas.

Podemos abrir tantas ventanas adicionales como precisemos, y organizarlas arrastrando y soltando sobre la ficha que contiene su nombre. Debemos tener en cuenta que la ventana hija o ficha activa será la que se actualice cuando nos desplazamos a un tema de ayuda nuevo.

Esta característica está disponible tanto si hemos configurado el IDE para utilizar la ayuda externa como interna.

Otros modos de acceso a la ayuda

La ruta de menú del sistema operativo *Inicio + Programas + Microsoft .NET Framework SDK*, nos lleva a un conjunto de opciones que contienen toda la documentación sobre la plataforma disponible en forma de ayuda, ejemplos, artículos, etc.

Por otra parte el menú también del sistema *Inicio + Programas + Microsoft Visual Studio .NET 7.0 + MSDN for Visual Studio 7.0*, abrirá la ventana de ayuda del IDE sin necesidad de tener abierto el entorno de desarrollo, proporcionándonos un medio adicional de acceso a la ayuda del programador.

11

Aplicaciones de consola

Una aplicación de consola es aquella que se ejecuta dentro de una ventana de línea de comandos. Este tipo de ventana recibe diferentes denominaciones: Símbolo del sistema, Sesión MS-DOS, Ventana de línea de comandos, etc., por lo que a lo largo de esta obra nos referiremos a ella de forma genérica como consola.

Las aplicaciones de consola son muy útiles cuando necesitamos realizar pruebas que no impliquen el uso del modo gráfico del sistema operativo: formularios, controles, imágenes, etc., ya que consumen menos recursos y su ejecución es más veloz.

En nuestro caso particular, debido a que los próximos temas versarán sobre aspectos del lenguaje, y en ellos no necesitaremos obligatoriamente el uso de formularios, utilizaremos aplicaciones de consola para los ejemplos.

Creación de un proyecto de tipo aplicación de consola

Para crear una aplicación de consola básica, después de iniciar el IDE de VS.NET, y seleccionar el menú para crear un nuevo proyecto, elegiremos *Aplicación de consola* en el panel derecho de la ventana *Nuevo proyecto*. El resto de opciones de esta ventana se configuran igual que para una aplicación con formularios Windows. Ver Figura 160.

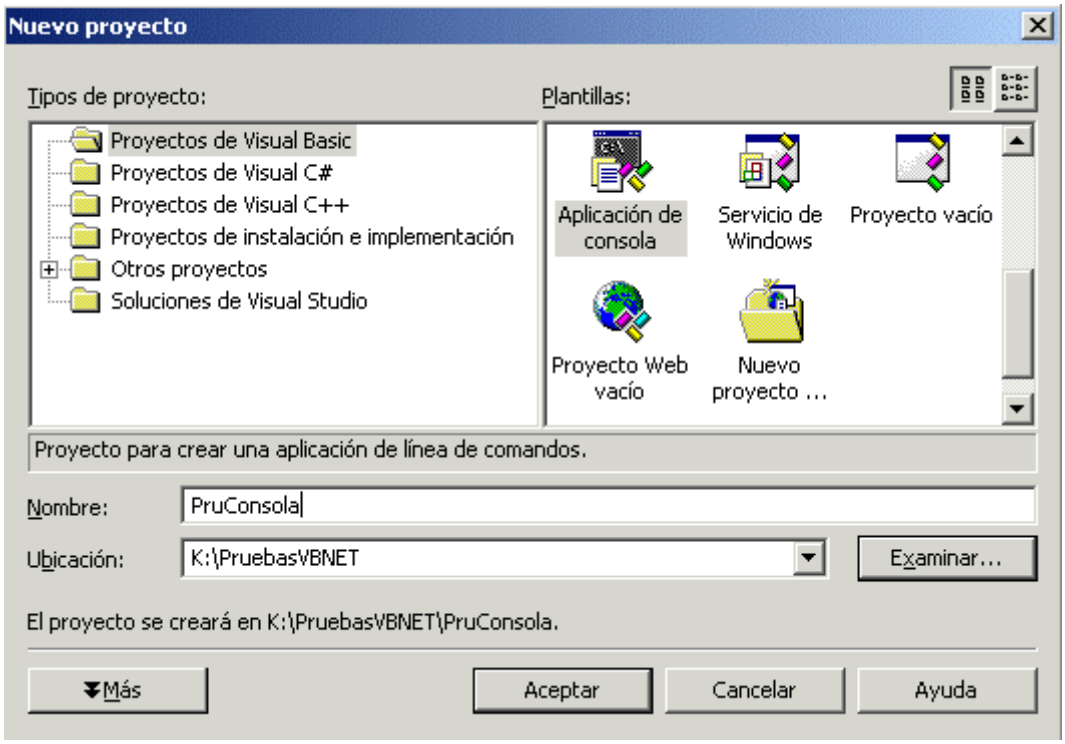


Figura 160. Creación de un proyecto de tipo consola.

Después de pulsar Aceptar se creará el proyecto que contendrá un fichero de código con el nombre MODULE1.VB, en cuyo interior encontraremos un módulo de código conteniendo un procedimiento Main( ) vacío, por el que comenzará la ejecución del programa. Ver Código fuente 37.

```
Module Module1

    Sub Main()

    End Sub

End Module
```

Código fuente 37

La clase Console

Esta clase se encuentra dentro del espacio de nombres System, y nos proporciona a través de sus métodos, acceso a la consola para mostrar u obtener información del usuario.

Debido a que los miembros de esta clase se encuentran compartidos (shared), no es necesario crear una instancia previa de la misma en una variable, pudiendo ejecutar directamente sus métodos sobre el objeto Console. Todo ello se explicará en los siguientes apartados.

Escritura de información

Para mostrar texto utilizaremos el método WriteLine( ) del objeto Console. Este método escribe en la línea actual de la consola el valor que le pasemos como parámetro, añadiendo automáticamente las marcas de retorno de carro y nueva línea, por lo que la siguiente escritura se realizará en una nueva línea. Ver Código fuente 38.

```
Sub Main()
    Console.WriteLine("Hola mundo desde la consola")
    Console.WriteLine("Esta es otra línea nueva")
End Sub
```

Código fuente 38

El código fuente anterior tiene no obstante un inconveniente: cuando el lector lo ejecute observará que se muestra la consola con las líneas de texto, pero inmediatamente vuelve a cerrarse, no dejando apenas tiempo para ver su contenido. Esto es debido a que no utilizamos ninguna instrucción que establezca una parada en la ejecución, que nos permita observar el resultado de lo que hemos escrito en la consola.

Para remediar este problema, utilizaremos el método ReadLine( ), que realiza una lectura de los caracteres que vayamos introduciendo en la línea actual de la consola, e impedirá continuar la ejecución hasta que no pulsemos [INTRO]. Ver Código fuente 39 y el resultado en la Figura 161.

```
Sub Main()
    Console.WriteLine("Hola mundo desde la consola")
    Console.WriteLine("Esta es otra línea nueva")

    Console.ReadLine()
End Sub
```

Código fuente 39

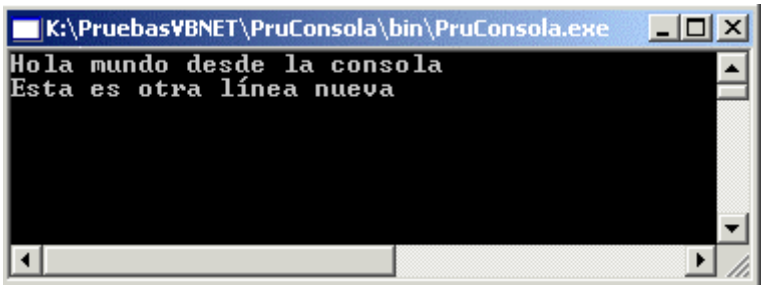


Figura 161. Escritura en la consola con parada de ejecución.

Los valores a mostrar con WriteLine( ) pueden ser de distintos tipos de datos, pudiendo insertar también líneas en blanco, como vemos en el Código fuente 40.

```
Sub Main()
    ' ejemplos con WriteLine()
```

```
    ' escritura de cadenas de caracteres
    Console.WriteLine("Esta es la primera línea")
    Console.WriteLine("Ahora ponemos una línea vacía")

    Console.WriteLine()  ' línea vacía

    ' escritura de números
    Console.WriteLine("A continuación escribimos un número")
    Console.WriteLine(5891)

    Console.WriteLine("Operaciones con números")
    Console.WriteLine(500 + 150)

    Console.WriteLine()  ' otra línea vacia

    ' escritura de valores lógicos
    Console.WriteLine("Resultado de la expresión lógica: 5 > 2")
    Console.WriteLine(5 > 2)

    ' parada y espera respuesta del usuario
    Console.ReadLine()
End Sub
```

Código fuente 40

La Figura 162 muestra la consola con el resultado de la ejecución del anterior fuente.

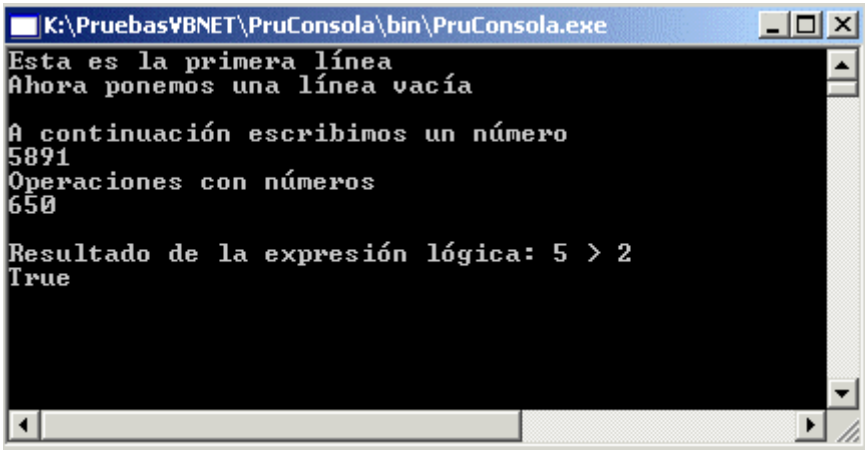


Figura 162. Escritura de valores en la consola con WriteLine( ).

Write( ) es otro método que nos permite también escribir valores en la consola. Su uso es igual que WriteLine( ), aunque debemos tener en cuenta que Write( ) no separa los valores a mostrar. Veamos un ejemplo en el Código fuente 41.

```
Sub Main()
    Console.Write("Hola")
    Console.Write("A")
    Console.Write("Todos")
    Console.Write(3456)
End Sub
```

Código fuente 41

La ejecución del anterior código mostraría en la consola los valores así: HolaATodos3456.

Para evitar que el texto en la consola salga junto, podemos incluir espacios al comienzo y/o al final en las cadenas de caracteres que pasemos como parámetro a Write( ), o bien utilizar este método pasando una cadena vacía. Ver Código fuente 42.

```
Sub Main()
    ' ejemplos con Write()

    Console.Write("Hola ")
    Console.Write("A")
    Console.Write(" Todos")
    Console.Write(" ")
    Console.Write(3456)

    Console.ReadLine()
End Sub
```

Código fuente 42

Escritura de múltiples valores en la misma línea

Al utilizar WriteLine( ) o Write( ) ocurrirá con frecuencia que en el texto a mostrar debamos incluir valores que se encuentran en variables o expresiones, por lo que tendremos que realizar una combinación de la cadena de texto principal con los demás elementos para obtener la cadena final que mostraremos al usuario. Esto lo podemos hacer empleando dos técnicas:

Concatenación.

Concatenando a la cadena principal las variables que contienen los valores a mostrar. Ver Código fuente 43.

```
' concatenar múltiples valores

' declarar variables
Dim Nombre As String
Dim Numero As Integer

' asignar valor a las variables
Nombre = "Luis"
Numero = 15

Console.WriteLine("He visto a " & Nombre & " transportando " & Numero & " cajas")
Console.ReadLine()
```

Código fuente 43

La Figura 163 muestra el resultado.

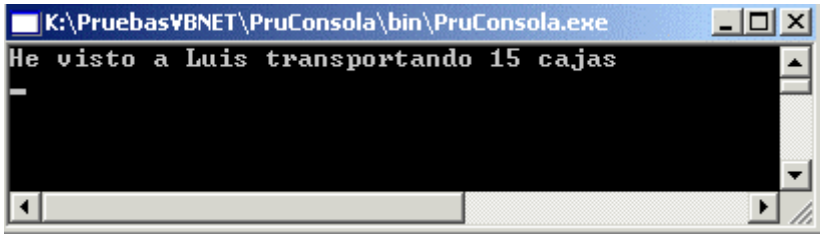


Figura 163. Concatenar valores a mostrar en la consola.

Parámetros sustituibles.

Pasando como primer parámetro la cadena a mostrar, y añadiendo tantos parámetros adicionales como valores debamos mostrar en la cadena principal. En la cadena principal indicaremos el lugar en donde visualizaremos los parámetros poniendo su número entre los símbolos de llave "{ }". El Código fuente 44 muestra la misma situación del ejemplo anterior pero utilizando esta técnica. El resultado final en la consola es el mismo que el del ejemplo anterior.

```
' combinación de múliples valores

' declarar variables
Dim Nombre As String
Dim Numero As Integer

' asignar valor a las variables
Nombre = "Luis"
Numero = 15

' el primer parámetro contiene la cadena a mostrar,
' el resto de parámetros son las variables que se visualizarán
' en alguna posición de la cadena del primer parámetro,
' el contenido de la variable Nombre se situará en el lugar
' indicado por {0}, la variable Numero en la posición de {1} y
' así sucesivamente
Console.WriteLine("He visto a {0} transportando {1} cajas", Nombre, Numero)
Console.ReadLine()
```

Código fuente 44

Como habrá comprobado el lector, los parámetros sustituibles comienzan a numerarse por cero, no estando obligados a mostrarlos en el mismo orden en el que los hemos situado en la llamada al método. El Código fuente 45 muestra dos ejemplos de sustitución de parámetros, uno de ellos con el mismo orden en el que se han situado en WriteLine( ), y otro con un orden distinto.

```
' declaración de variables
Dim Lugar As String
Dim Numero As Integer
Dim Vehiculo As String

' asignación de valor a variables
Vehiculo = "tren"
Lugar = "Alicante"
Numero = 300

' visualización de resultados en consola
Console.WriteLine("El {0} con destino {1} viaja a mas de {2} kms. por hora",
Vehiculo, Lugar, Numero)
```

```
Console.WriteLine()

Console.WriteLine("El {2} con destino {0} viaja a mas de {1} kms. por hora",
Vehiculo, Lugar, Numero)
Console.ReadLine()
```

Código fuente 45

Al ejecutar este fuente, la consola mostrará el aspecto de la Figura 164.

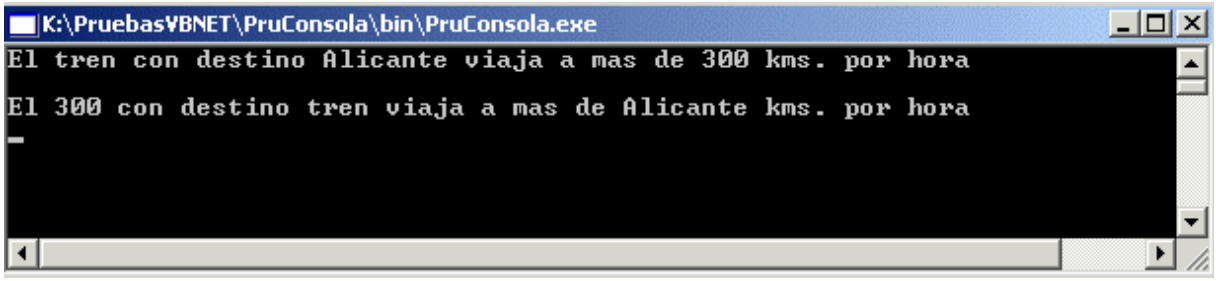


Figura 164. Diversas salidas a la consola con parámetros sustituibles.

Lectura de información

Para obtener el texto escrito por el usuario en la línea actual de la consola y hasta la pulsación de [INTRO] podemos utilizar el método ReadLine( ) del objeto Console.

El Código fuente 46 muestra como volcamos a una variable el contenido de la línea escrita por el usuario y posteriormente exponemos su contenido, también a través de la consola.

```
' declaramos una variable para volcar el contenido
' de una línea de la consola
Dim LineaTexto As String

Console.WriteLine("Introducir un texto")
LineaTexto = Console.ReadLine()  ' el texto se pasa a la variable

' ahora mostramos lo que hemos escrito
Console.WriteLine()
Console.WriteLine("El usuario ha escrito la siguiente línea:")
Console.WriteLine(LineaTexto)

' aquí evitamos cerrar la consola,
' así podemos ver mejor el resultado
Console.ReadLine()
```

Código fuente 46

Read( ) es otro método del objeto Console que permite también la lectura del dispositivo de entrada de la consola, pero en este caso devuelve el código de una sola tecla pulsada por el usuario. Para ilustrar el uso de este método tenemos el ejemplo del Código fuente 47, en el que después de pulsar varias teclas, nos introducimos en un bucle que va extrayendo cada uno de sus códigos, que volvemos a transformar en el carácter correspondiente a la tecla pulsada.

```
' ejemplo con Read()
Dim CodTecla As Integer
Dim NombreTecla As Char

Console.WriteLine("Pulsar varias teclas")
Console.WriteLine()

While True
    ' tomar los códigos de las teclas uno a uno
    CodTecla = Console.Read()

    ' si se ha pulsado intro, salir
    If CodTecla = 13 Then
        Exit While
    End If

    Console.WriteLine("Código de tecla pulsada: {0}", CodTecla)

    ' convertir el código al caracter de la tecla
    NombreTecla = Chr(CodTecla)

    Console.WriteLine("Tecla pulsada: {0}", NombreTecla)
End While

Console.ReadLine()
Console.WriteLine("Ejemplo terminado, pulse intro")
Console.ReadLine()
```

Código fuente 47

12

El lenguaje

El lenguaje, principio del desarrollo

Desde la llegada de los interfaces visuales basados en ventanas, los productos para el desarrollo de aplicaciones, han ido incorporando paulatinamente un mayor número de ayudas y asistentes que hacían a veces olvidar el verdadero soporte de aquello que utilizábamos para programar: el lenguaje.

Con la frase *olvidar el lenguaje*, nos referimos a que en el caso concreto de VB6, cualquier persona con un nivel de usuario medio/avanzado, y sin una base sólida de programación, conociendo un pequeño conjunto de instrucciones del lenguaje, podía escribir programas, dada la gran cantidad de utilidades y apoyo proporcionados por el entorno de programación.

Esto es claramente contraproducente, puesto que no se aprovecha todo el potencial que ofrece la herramienta al desconocer su elemento más importante, el lenguaje, del cual parten el resto de aspectos del producto.

En el caso de VB.NET este aspecto se acentúa, debido al gran trabajo realizado en dotar al lenguaje de un elevado número de características que estaban siendo reclamadas desde hace ya tiempo por la comunidad de programadores.

Estas mejoras no han sido realizadas exclusivamente para VB.NET, ya que este lenguaje se ha beneficiado indirectamente de ellas, puesto que al compartir ahora todos los lenguajes de .NET Framework una especificación común, Visual Basic como lenguaje, ha necesitado ser adaptado para cumplir con dicha normativa, beneficiándose así de la potencia incluida en todas las características de la plataforma .NET.

En este tema realizaremos una introducción al lenguaje, repasando sus aspectos principales: ficheros de código y su organización, módulos, procedimientos, identificadores, operadores, estructuras de control, etc.

Dado el vasto número de características disponibles, realizaremos una exposición elemental de cada una, ampliando allá donde sea necesario en posteriores temas del texto.

Estructura de un programa VB.NET

En el tema *Escritura de código* ya hicimos una descripción de los principales elementos de que consta un programa para VB.NET de tipo Windows. En este caso vamos a crear desde el IDE un nuevo proyecto, en esta ocasión de tipo consola. Los componentes de un programa de estas características serán los mismos, salvando claro está, las diferencias de interfaz de usuario entre ambas clases tipos de aplicación.

La Figura 165 muestra de una manera gráfica la estructura de una aplicación, en forma de niveles.

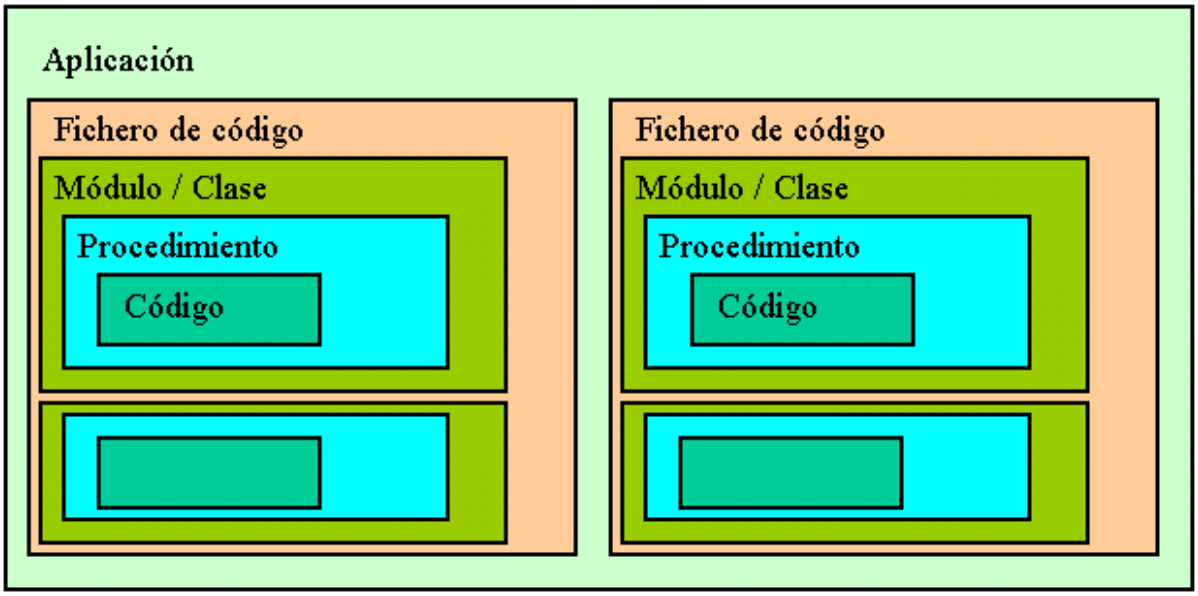


Figura 165. Estructura en niveles de una aplicación VB.NET.

Como muestra el diagrama, una aplicación está formada por uno o más ficheros de código, que a su vez contienen módulos de código o clases, dentro de los que se escriben procedimientos que son los elementos que contienen el código base.

Cuando creamos una aplicación usando VS.NET, es el propio IDE quién se encarga de crear por nosotros la estructura básica del programa: crea un fichero de código conteniendo un módulo que tiene el procedimiento de entrada, sólo falta el código del programador.

Todos los elementos que componen una aplicación VB.NET, son organizados por VS.NET bajo el concepto de proyecto. Un proyecto aglutina los ficheros de código de la aplicación, recursos, referencias a clases globales de la plataforma .NET, etc. Consulte el lector el tema dedicado a la primera aplicación en VB.NET para una descripción general de estos tipos de fichero.

De manera implícita, cada vez que creamos un nuevo proyecto utilizando el IDE, dicho proyecto es al mismo tiempo un ensamblado de ámbito privado, por lo que también podemos referirnos a una aplicación utilizando ambos términos: proyecto o ensamblado.

Main( ) como procedimiento de entrada al programa

Todo programa necesita una rutina o procedimiento de entrada, que sea el primero que se ejecute. En VB.NET ese procedimiento recibe el nombre especial Main( ), y debe estar contenido dentro de un módulo de código, como muestra el Código fuente 48

```
Module Module1

    Sub Main()

    End Sub

End Module
```

Código fuente 48

En el caso de una aplicación de consola creada desde VS.NET, se crea un módulo de forma automática que contiene un procedimiento Main( ) vacío. Dentro de este procedimiento escribiremos el código de los próximos ejemplos.

Variables

Una variable es un identificador del programa que guarda un valor que puede ser modificando durante el transcurso de dicha aplicación.

Declaración

La declaración de una variable es el proceso por el cual comunicamos al compilador que vamos a crear una nueva variable en el programa.

Para declarar una variable utilizaremos la palabra clave Dim, seguida del identificador o nombre que daremos a dicha variable. Ver Código fuente 49

```
Sub Main()
    Dim MiValor
End Sub
```

Código fuente 49

Denominación

Respecto al nombre de la variable, debe empezar por letra, y no puede ser ninguna de las palabras reservadas del lenguaje, ni contener caracteres como operadores u otros símbolos especiales. VerCódigo fuente 50

```
Sub Main()
    Dim MiValor  ' nombre correcto
    Dim Total2   ' nombre correcto
    Dim Mis_Datos  ' nombre correcto
    Dim 7Datos   ' nombre incorrecto
    Dim Nombre+Grande  ' nombre incorrecto
    Dim End  ' nombre incorrecto
End Sub
```

Código fuente 50

Como podemos comprobar en este fuente, y ya explicamos en un tema anterior, incluimos comentarios en el código usando la comilla simple (') seguida del comentario correspondiente.

Avisos del IDE sobre errores en el código

Al declarar una variable con un nombre incorrecto, o si se produce otro tipo de error en la escritura del código, el propio IDE se encarga de avisarnos que existe un problema subrayando el fragmento de código conflictivo y mostrando una viñeta informativa al situar sobre dicho código el cursor. Ver Figura 166

```
Sub Main()
    Dim MiValor  ' nombre correcto
    Dim Total2   ' nombre correcto
    Dim 7Datos   ' nombre incorrecto
    Dim Nombre+Grande  ' nombre incorrecto
                                          Se esperaba un final de instrucción.
```

Figura 166. Código con errores subrayado por el IDE.

Estos avisos constituyen una gran ayuda, ya que permiten al programador observar problemas en la escritura del código, antes incluso de ejecutar el programa.

Existen multitud de avisos de muy diversa naturaleza, teniendo en cuenta que la tónica general consiste en que el código problemático quedará subrayado por el IDE hasta que no modifiquemos la línea en cuestión y la escribamos correctamente.

Lugar de la declaración

Podemos declarar variables en muy diversos lugares del código. El punto en el que declaremos una variable será determinante a la hora del ámbito o accesibilidad a esa variable desde otros puntos del programa. Por ahora, y ciñéndonos a la declaración de variables dentro de procedimientos, recomendamos declarar todas las variables en la cabecera o comienzo del procedimiento, para dar una

mayor claridad al mismo. Después de la declaración, escribiríamos el resto de instrucciones del procedimiento.

Tipificación

La tipificación de una variable es la operación por la cual, al declarar una variable, especificamos qué clase de valores o tipo de datos vamos a poder almacenar en dicha variable.

En VB.NET utilizamos la palabra clave As seguida del nombre del tipo de datos, para establecer el tipo de una variable. Ver Código fuente 51

```
Sub Main()
    Dim Valor As String   ' cadena de caracteres
    Dim Cuenta As Integer ' numérico entero
    Dim FhActual As Date  ' fecha
End Sub
```

Código fuente 51

La Tabla 5 muestra la relación de tipos de datos disponibles en el lenguaje.

Tipo de dato en VB.NET	Tipo correspondiente en el entorno de .NET Framework	Tamaño	Rango de valores
Boolean	System.Boolean	1 byte	True o False (Verdadero o Falso)
Byte	System.Byte	1 byte	0 a 255 (sin signo)
Char	System.Char	2 bytes	0 a 65535 (sin signo)
Date	System.DateTime	8 bytes	1 de enero de 1 a 31 de diciembre de 9999
Decimal	System.Decimal	12 bytes	+/- 79.228.162.514.264.337.593.543.950.335 sin punto decimal; +/-7,9228162514264337593543950335 con 28 posiciones a la derecha del signo decimal; el número más pequeño distinto de cero es +/-0,0000000000000000000000000001
Double (punto flotante con precisión doble)	System.Double	8 bytes	-1,79769313486232E308 a -4,94065645841247E-324 para valores negativos; 4,94065645841247E-324 a 1,79769313486232E308 para valores positivos
Integer	System.Int32	4 bytes	-2.147.483.648 a 2.147.483.647

Long (entero largo)	System.Int64	8 bytes	-9.223.372.036.854.775.808 a 9.223.372.036.854.775.807
Short	System.Int16	2 bytes	-32.768 a 32.767
Single (punto flotante con precisión simple)	System.Single	4 bytes	-3,402823E38 a –1,401298E-45 para valores negativos; 1,401298E-45 a 3,402823E38 para valores positivos
Object	System.Object	4 bytes	Cualquier tipo
String (cadena de longitud variable)	System.String	10 bytes + (2 * longitud de la cadena)	Desde 0 a unos 2.000 millones de caracteres Unicode
Estructura (tipo de dato definido por el usuario)	Hereda de System.ValueType	Suma de los tamaños de los miembros de la estructura	Cada miembro de la estructura tiene un intervalo de valores determinado por su tipo de datos e independiente de los intervalos de valores correspondientes a los demás miembros

Tabla 5. Tipos de datos en VB.NET.

Si al declarar una variable no indicamos el tipo, por defecto tomará Object, que corresponde al tipo de datos genérico en el entorno del CLR, y admite cualquier valor.

Según la información que acabamos de ver, si declaramos una variable de tipo Byte e intentamos asignarle el valor 5899 se va a producir un error, ya que no se encuentra en el intervalo de valores permitidos para esa variable. Esto puede llevar al lector a preguntar: "¿por qué no utilizar siempre Object y poder usar cualquier valor?, o mejor ¿para qué necesitamos asignar tipo a las variables?".

El motivo de tipificar las variables reside en que cuando realizamos una declaración, el CLR debe reservar espacio en la memoria para los valores que pueda tomar la variable, como puede ver el lector en la tabla anterior, no requiere el mismo espacio en memoria una variable Byte que una Date. Si además, declaramos todas las variables como Object, los gastos de recursos del sistema serán mayores que si establecemos el tipo adecuado para cada una, ya que como el CLR no sabe el valor que puede tomar en cada ocasión la variable, debe realizar un trabajo extra de adecuación, consumiendo una mayor cantidad de recursos.

Una correcta tipificación de las variables redundará en un mejor aprovechamiento de las capacidades del sistema y en un código más veloz en ejecución. Cuantos más programas se diseñen optimizando en este sentido, el sistema operativo ganará en rendimiento beneficiándose el conjunto de aplicaciones que estén en ejecución.

VS.NET dispone de una ayuda al asignar el tipo a una variable, que nos muestra la lista de tipos disponibles para poder seleccionar uno sin tener que escribir nosotros el nombre. Al terminar de escribir la palabra As, aparecerá dicha lista, en la que pulsando las primeras letras del tipo a buscar, se irá situando en los más parecidos. Una vez encontrado, pulsaremos la tecla Enter o Tab para tomarlo. Ver Figura 167.

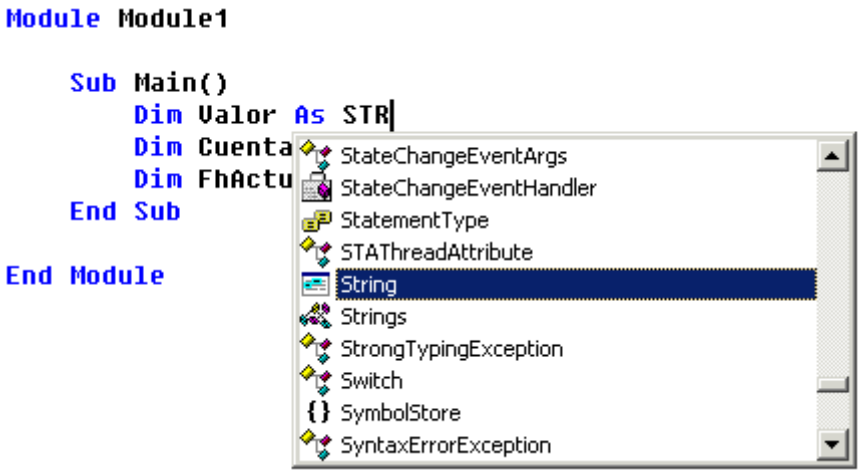


Figura 167. Lista de tipos de datos al declarar una variable.

Declaración múltiple en línea

En el caso de que tengamos que declarar más de una variable del mismo tipo, podemos declararlas todas en la misma línea, separando cada una con una coma e indicando al final de la lista el tipo de dato que van a tener, como vemos en el Código fuente 52

```
Dim Nombre, Apellidos, Ciudad As String
```

Código fuente 52

Con esta técnica de declaración, todas las variables de la línea tienen el mismo tipo de dato, ya que no es posible declarar múltiples variables en la misma línea que tengan distintos tipos de dato.

Asignación de valor

Para asignar un valor a una variable utilizaremos el operador de asignación: el signo igual (=), situando a su izquierda la variable a asignar, y a su derecha el valor. Ver Código fuente 53

```
Dim Cuenta As Integer
Cuenta = 875
```

Código fuente 53

Según el tipo de dato de la variable, puede ser necesario el uso de delimitadores para encerrar el valor que vamos a asignar.

- **Tipos numéricos**. Las variables de tipos de datos numéricos no necesitan delimitadores, se asigna directamente el número correspondiente. Si necesitamos especificar decimales, utilizaremos el punto (.) como carácter separador para los decimales

- **Cadenas de caracteres**. En este caso es preciso encerrar la cadena entre comillas dobles (").

- **Fechas**. Al asignar una fecha a una variable de este tipo, podemos encerrar dicho valor entre el signo de almohadilla (#) o comillas dobles ("). El formato de fecha a utilizar depende del delimitador. Cuando usemos almohadilla la fecha tendrá el formato Mes/Día/Año; mientras que cuando usemos comillas dobles el formato será Día/Mes/Año.

Las fechas pueden contener además información horario que especificaremos en el formato Hora:Minutos:Segundos FranjaHoraria. En el caso de que no indiquemos la franja horaria (AM/PM) y si estamos utilizando el signo almohadilla como separador, el entorno insertará automáticamente los caracteres de franja horaria correspondientes.

- **Tipos lógicos**. Las variables de este tipo sólo pueden tener el valor True (Verdadero) o False (Falso).

Además de asignar valores como acabamos de explicar, podemos asignar el contenido de una variable a otra o el resultado de una expresión, como veremos más adelante en el apartado dedicado a operadores. El Código fuente 54 muestra unos ejemplos de asignación a variables, que después visualizamos en la consola.

```
Sub Main()
    Dim ImporteFac As Integer
    Dim Precio As Double
    Dim Valor As String
    Dim FhActual As Date
    Dim FhNueva As Date
    Dim FhCompletaUno As Date
    Dim FhCompletaDos As Date
    Dim FhHora As Date
    Dim Correcto As Boolean

    ImporteFac = 875
    Precio = 50.75

    Valor = "mesa"

    FhActual = #5/20/2001#  ' mes/día/año
    FhNueva = "25/10/2001"  ' dia/mes/año
    FhCompletaUno = #10/18/2001 9:30:00 AM#
    FhCompletaDos = "7/11/2001 14:22:00"
    FhHora = #5:40:00 PM#

    Dim NuevaCadena As String
    NuevaCadena = Valor  ' asignar una variable a otra

    Correcto = True

    ' mostrar variables en la consola
    Console.WriteLine("Variable ImporteFac: {0}", ImporteFac)
    Console.WriteLine("Variable Precio: {0}", Precio)
    Console.WriteLine("Variable Valor: {0}", Valor)
    Console.WriteLine("Variable FhActual: {0}", FhActual)
    Console.WriteLine("Variable FhNueva: {0}", FhNueva)
    Console.WriteLine("Variable FhCompletaUno: {0}", FhCompletaUno)
    Console.WriteLine("Variable FhCompletaDos: {0}", FhCompletaDos)
    Console.WriteLine("Variable FhHora: {0}", FhHora)
    Console.WriteLine("Variable NuevaCadena: {0}", NuevaCadena)
    Console.WriteLine("Variable Correcto: {0}", Correcto)
```

```
    Console.ReadLine()
End Sub
```

Código fuente 54

Otra cualidad destacable en este apartado de asignación de valores, reside en que podemos declarar una variable y asignarle valor en la misma línea de código, como vemos en el Código fuente 55

```
Dim Valor As String = "mesa"
Dim ImporteFac As Integer = 875
```

Código fuente 55

Valor inicial

Toda variable declarada toma un valor inicial por defecto, a no ser que realicemos una asignación de valor en el mismo momento de la declaración. A continuación se muestran algunos valores de inicio en función del tipo de dato que tenga la variable:

- **Numérico**. Cero (0).
- **Cadena de caracteres**. Cadena vacía ("").
- **Fecha**. 01/01/0001 0:00:00.
- **Lógico**. Falso (False).
- **Objeto**. Valor nulo (Nothing).

El Código fuente 56 muestra un ejemplo de valores iniciales.

```
Sub Main()
    Dim ImporteFac As Integer
    Dim Valor As String
    Dim FhActual As Date
    Dim FhNueva As Date
    Dim ValorLogico As Boolean
    Dim UnObjeto As Object

    ' mostrar variables en la consola
    Console.WriteLine("Variable ImporteFac: {0}", ImporteFac)
    Console.WriteLine("Variable Valor: {0}", Valor)
    Console.WriteLine("Variable FhActual: {0}", FhActual)
    Console.WriteLine("Variable FhNueva: {0}", FhNueva)
    Console.WriteLine("Variable ValorLogico: {0}", ValorLogico)
    Console.WriteLine("Variable UnObjeto: {0}", UnObjeto)
    Console.ReadLine()
End Sub
```

Código fuente 56

Debemos tener en cuenta al ejecutar estas líneas, que en los casos de las variables de tipo cadena y objeto, no se mostrará nada, ya que se considera que están inicializadas pero vacías.

Por otro lado podemos, inversamente, inicializar una variable que ya tiene valor, asignándole la palabra clave Nothing; con ello, la variable pasa a tener el valor por defecto o inicial. Ver el Código fuente 57.

```
Sub Main()
    Dim Valor As String
    Dim FhActual As Date
    Dim ValorLogico As Boolean

    ' asignar valores a variables
    Valor = "mesa"
    FhActual = "10/8/2001"
    ValorLogico = True

    ' inicializar variables
    Valor = Nothing
    FhActual = Nothing
    ValorLogico = Nothing

    ' mostrar variables en la consola
    Console.WriteLine("Variable Valor: {0}", Valor)
    Console.WriteLine("Variable FhActual: {0}", FhActual)
    Console.WriteLine("Variable ValorLogico: {0}", ValorLogico)
    Console.ReadLine()
End Sub
```

Código fuente 57

Declaración obligatoria

Es obligatorio, por defecto, la declaración de todas las variables que vayamos a utilizar en el código. En el caso de que intentemos utilizar una variable no declarada, se producirá un error.

La declaración de variables proporciona una mayor claridad al código, ya que de esta forma, sabremos en todo momento si un determinado identificador corresponde a una variable de nuestro procedimiento, de un parámetro, etc.

Mediante la instrucción *Option Explicit*, y sus modificadores On/Off, podemos requerir o no la declaración de variables dentro del programa.

- **Option Explicit On**. Hace obligatoria la declaración de variables. Opción por defecto.
- **Option Explicit Off**. Hace que no sea obligatoria la declaración de variables.

Podemos aplicar esta instrucción para que tenga efecto a nivel de proyecto y a nivel de fichero de código.

- **Option Explicit a nivel de proyecto**.

Para establecer Option Explicit a nivel de proyecto, debemos abrir la ventana Explorador de soluciones, hacer clic en el nombre del proyecto, y a continuación pulsar el botón de propiedades en esa misma ventana. Esto mostrará la ventana de propiedades del proyecto, en cuyo panel izquierdo haremos clic sobre el elemento Generar. Finalmente abriremos la lista desplegable del elemento Option Explicit, seleccionaremos un valor (On, Off) y pulsaremos Aplicar y Aceptar. Ver Figura 168.

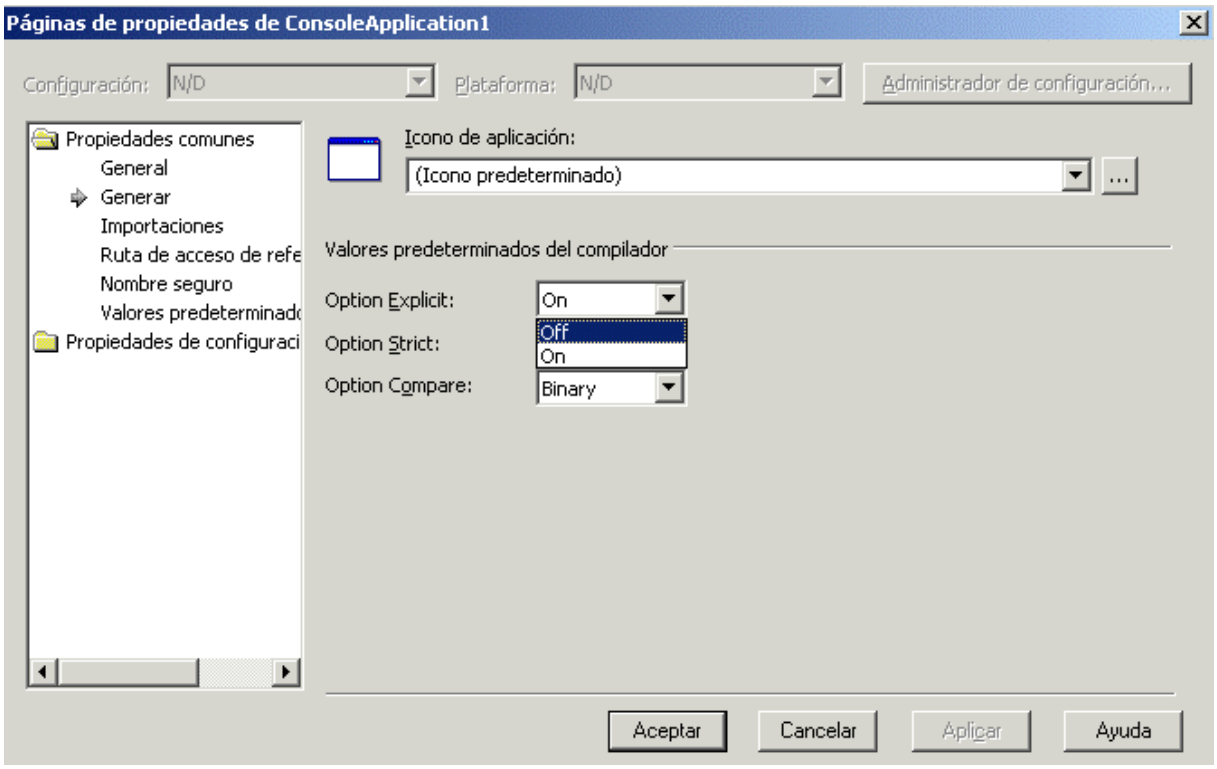


Figura 168. Propiedades del proyecto para modificar la declaración obligatoria de variables.

Con la declaración obligatoria desactivada podríamos escribir código como el mostrado en el Código fuente 58.

```
Sub Main()
    Valor = "coche"
    MiDato = 984

    Console.WriteLine("Variable Valor: {0}", Valor)
    Console.WriteLine("Variable MiDato: {0}", MiDato)
    Console.ReadLine()
End Sub
```

Código fuente 58

En el ejemplo anterior, no hemos declarado las variables en Main( ). Al estar Option Explicit Off esto no produce error, y el CLR al detectar un identificador sin declarar, crea una nueva variable internamente.

Mucho más fácil que tener que declarar las variables ¿verdad?. Pues precisamente esta facilidad es uno de los graves problemas de no declarar variables. En un procedimiento de prueba con poco código, esto no supone una importante contrariedad. Sin embargo pensemos un momento, que en lugar de un pequeño procedimiento, se trata de una gran aplicación con muchas líneas de código, procedimientos, y cientos de variables. Al encontrarnos con una variable de esta forma, no sabremos si esa variable ya la hemos utilizado con anterioridad en el procedimiento, si ha sido pasada como parámetro al mismo, etc. Estas circunstancias provocan que nuestro código se vuelva complejo de interpretar, retrasando la escritura general de la aplicación. Si volvemos a activar Option Explicit On, inmediatamente sabremos

que algo va mal, ya que toda variable no declarada, quedará subrayada por el IDE como un error de escritura. Las ventajas son evidentes.

Option Explicit a nivel de fichero.

Para establecer la declaración obligatoria a nivel de fichero, debemos situarnos al comienzo del fichero de código y escribir la instrucción Option Explicit con el modificador correspondiente. El Código fuente 59 muestra un ejemplo de cómo desactivar esta característica en el fichero de código actual.

```
' desactivar declaración obligatoria de variables
' ahora podemos, dentro de este fichero de código,
' escribir todas las variables sin declarar

Option Explicit Off

Module Module1

    Sub Main()
        Valor = "coche"
        MiDato = 984

        Console.WriteLine("Variable Valor: {0}", Valor)
        Console.WriteLine("Variable MiDato: {0}", MiDato)
        Console.ReadLine()
    End Sub

End Module
```

Código fuente 59

Option Explicit a nivel de fichero, nos permite establecer el modo de declaración de variables sólo para ese fichero en el que lo utilizamos, independientemente del tipo de obligatoriedad en declaración de variables establecido de forma general para el proyecto. Podemos por ejemplo, tener establecido Option Explicit On para todo el proyecto, mientras que para un fichero determinado podemos no obligar a declarar variables escribiendo al comienzo del mismo Option Explicit Off.

El hecho de tener Option Explicit Off no quiere decir que no podamos declarar variables, podemos, por supuesto declararlas, lo que sucede es que el compilador no generará un error al encontrar una variable sin declarar.

El otro grave problema al no declarar variables proviene por la incidencia en el rendimiento de la aplicación. Cuando tenemos Option Explicit Off, el CLR por cada identificador que encuentre sin declarar, crea una nueva variable, y ya que desconoce qué tipo de dato querría utilizar el programador, opta por asignarle el más genérico: Object.

Una excesiva e innecesaria proliferación de variables Object afectan al rendimiento del programa, ya que el CLR debe trabajar doblemente en la gestión de recursos utilizada por dichas variables. En el próximo apartado trataremos sobre la obligatoriedad a la hora de tipificar variables.

Por todo lo anteriormente comentado, a pesar de la engañosa facilidad y flexibilidad de Option Explicit Off, nuestra recomendación es tener configurado *siempre* Option Explicit On a nivel de aplicación, nos ahorrará una gran cantidad de problemas.

Tipificación obligatoria

Cuando declaramos una variable, no es obligatorio por defecto, establecer un tipo de dato para la misma. Igualmente, al asignar por ejemplo, una variable numérica a una de cadena, se realizan automáticamente las oportunas conversiones de tipos, para transformar el número en una cadena de caracteres. Veamos un ejemplo en el Código fuente 60.

```
Sub Main()
    ' no es necesario tipificar la variable, tipificación implícita,
    ' la variable Valor se crea con el tipo Object
    Dim Valor

    ' tipificación explícita
    Dim Importe As Integer
    Dim UnaCadena As String

    ' al asignar una fecha a la variable Valor,
    ' sigue siendo de tipo Object, pero detecta que
    ' se trata de una fecha y guarda internamente
    ' esta información como un subtipo Date
    Valor = #8/20/2001#

    Importe = 590

    ' no es necesario hacer una conversión de tipos previa
    ' para asignar un número a una variable de cadena,
    ' ya que se realiza una conversión implícita,
    ' la variable UnaCadena contiene la cadena "590"
    UnaCadena = Importe

    Console.WriteLine("Variable Valor: {0}", Valor)
    Console.WriteLine("Variable Importe: {0}", Importe)
    Console.WriteLine("Variable UnaCadena: {0}", UnaCadena)
    Console.ReadLine()
End Sub
```

Código fuente 60

Como ya comentábamos en el apartado anterior, si no asignamos el tipo de dato adecuado al declarar una variable, el CLR le asigna el tipo Object, lo que afecta negativamente al rendimiento de la aplicación.

La instrucción *Option Strict*, junto a sus modificadores On/Off, nos permite establecer si en el momento de declarar variables, será obligatoria su tipificación. También supervisa la obligatoriedad de realizar una conversión de tipos al efectuar asignaciones entre variables, o de expresiones a variables.

- **Option Strict On**. Hace obligatoria la tipificación de variables y la conversión de tipos explícita.

- **Option Strict Off**. Hace que no sea obligatoria la tipificación de variables. La conversión de entre tipos distinta en asignaciones y expresiones es realizada automáticamente por el entorno. Opción por defecto.

Podemos configurar Option Strict a nivel de proyecto y de fichero de código, de igual forma que con Option Explicit. En el caso de configurar a nivel de proyecto, deberemos abrir la ventana de propiedades del proyecto, y en su apartado Generar, establecer el valor correspondiente en la lista desplegable Option Strict. Ver Figura 169.

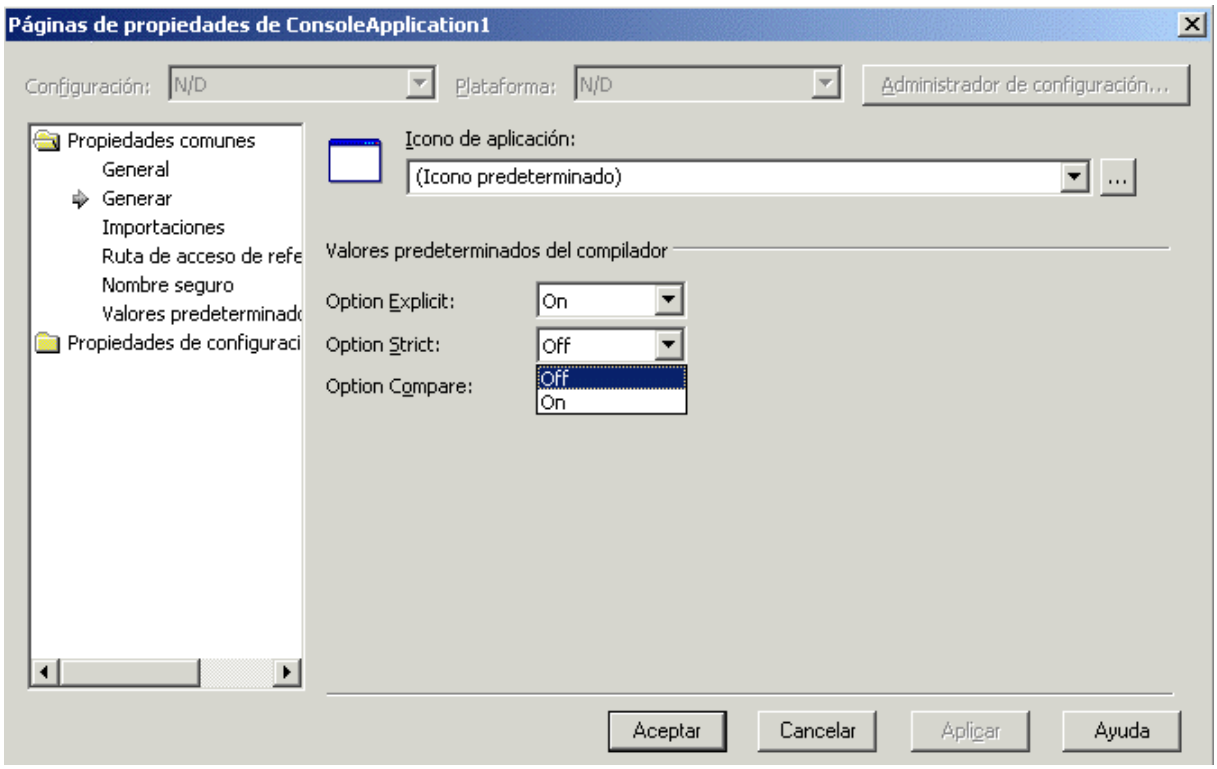


Figura 169. Configuración de Option Strict a nivel de proyecto.

Si configuramos a nivel de fichero de código, escribiremos esta instrucción en la cabecera del fichero con el modificador oportuno. Consulte el lector el anterior apartado para un mayor detalle sobre el acceso a esta ventana de propiedades del proyecto.

En el ejemplo del Código fuente 61, establecemos Option Strict On a nivel de fichero de código, y a partir de ese momento, no podremos asignar un tipo de dato Double a un Integer, o un valor numérico a una variable String, por exponer un par de casos de los más comunes. El código erróneo será marcado por el IDE como un error de sintaxis, e igualmente se producirá un error si intentamos ejecutar el programa.

```
Option Strict On
Module Module1
    Sub Main()
        ' ahora es obligatorio establecer
        ' el tipo de dato a todas las variables
        Dim Valor As Integer
        Dim TotalGeneral As Double
        Dim Dato As String
        TotalGeneral = 500
        Valor = TotalGeneral ' error, no se permite la conversión implícita
        Dato = TotalGeneral  ' error, no se permite la conversión implícita
    End Sub
End Module
```

Código fuente 61

Si queremos que no se produzcan errores de conversión en el anterior código fuente, tendremos que emplear las funciones de conversión de tipo que proporciona el lenguaje. En este caso utilizaremos

CInt( ), a la que pasamos un valor numérico como parámetro, y devuelve un tipo numérico Integer; y CStr( ), que convierte a String el valor que pasemos como parámetro. Veamos el resultado en el Código fuente 62.

```
Sub Main()
    ' ahora es obligatorio establecer
    ' el tipo de dato a todas las variables
    Dim Valor As Integer
    Dim TotalGeneral As Double
    Dim Dato As String

    TotalGeneral = 500
    Valor = CInt(TotalGeneral) ' conversión de tipos
    Dato = CStr(TotalGeneral) ' conversión de tipos
End Sub
```

Código fuente 62

Establecer Option Strict On requiere un mayor trabajo por parte del programador, ya que ha de ser más cuidadoso y escribir un código más correcto y preciso, lo cual es muy conveniente. Sin embargo, ya que la opción por defecto en este sentido es Option Strict Off, los ejemplos realizados a lo largo de este texto se ajustarán en este particular a dicha configuración, con ello ganamos en comodidad, ya que evitaremos la obligación de realizar conversiones de tipos en muy diversas situaciones.

Arrays, conceptos básicos

Un array consiste en una lista de valores asociada a un identificador. Al emplear una variable para contener más de un dato, el modo de acceder a los valores se consigue a través de un índice asociado a la variable, que permite saber con qué elemento o posición de la lista estamos tratando. Otros nombres para referirnos a un array son matriz y vector, aunque en este texto emplearemos el término array de forma genérica.

En este apartado vamos a realizar una introducción muy elemental a los arrays y su uso, que sirva al lector para obtener los conocimientos mínimos necesarios para este tema dedicado al lenguaje. Dado que la gran potencia de los arrays reside en su uso a través de las características de orientación a objetos de que disponen, cubriremos los arrays en profundidad posteriormente, dentro de un tema específico en el que trataremos todos sus aspectos principales.

Declaración

Para declarar un array actuaremos prácticamente igual que para declarar una variable normal, con la diferencia de que utilizaremos los paréntesis junto al nombre de la variable, para indicar que se trata de un array, y opcionalmente, dentro de los paréntesis, indicaremos el número de elementos de que inicialmente va a constar el array. También es posible, asignar valores a los elementos en el mismo momento de su declaración.

Debemos tener en cuenta a la hora de establecer el número de elementos, que el primer índice de un array es el cero, por lo que al ser creado, el número real de elementos en un array será el especificado en la declaración más uno.

La Figura 170 muestra la representación de un array en un modo gráfico.

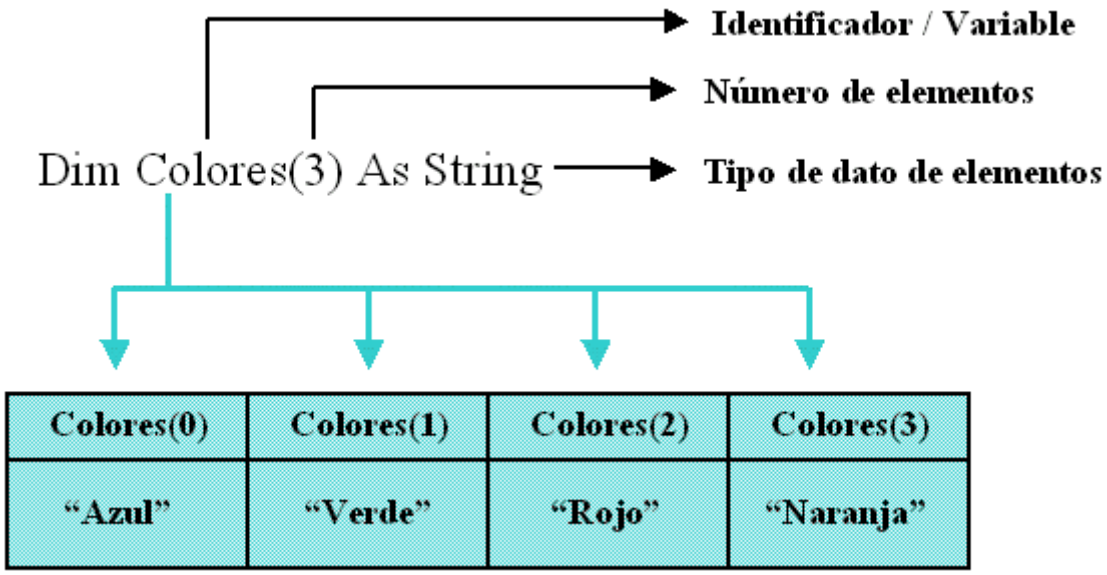


Figura 170. Representación gráfica de un array.

A continuación vemos unos ejemplos de creación de arrays en el Código fuente 63.

```
Sub Main()
    ' array sin elementos
    Dim Colores() As String

    ' array con 4 elementos: de 0 a 3
    Dim Nombres(3) As String

    ' array con 3 elementos, cuyos valores asignamos
    ' en el momento de la declaración del array
    Dim Frutas() As String = {"Manzana", "Naranja", "Pera"}

End Sub
```

Código fuente 63

Al declarar un array, todos sus valores son del mismo tipo de dato. Si necesitamos que dichos valores sean de tipos diferentes, debemos declarar el array como tipo Object, ya que al ser este, el tipo de dato genérico en el entorno de .NET, nos permitirá asignar valores de distintos tipos al array.

Asignación y obtención de valores

Para asignar y obtener valores de los elementos de un array, actuaremos igual que para una variable normal, pero empleando además el índice para indicar qué posición queremos manipular. Ver Código fuente 64.

```
Sub Main()
    ' array con 4 elementos: de 0 a 3
    Dim Nombres(3) As String
```

```
    ' asignar valores al array
    Nombres(0) = "Ana"
    Nombres(1) = "Pedro"
    Nombres(2) = "Antonio"
    Nombres(3) = "Laura"

    ' obtener valores de un array
    Dim ValorA As String
    Dim ValorB As String

    ValorA = Nombres(1)  ' Pedro
    ValorB = Nombres(3)  ' Laura

    ' mostrar los valores obtenidos del array
    Console.WriteLine("Variables: ValorA --> {0}, ValorB --> {1}", ValorA, ValorB)
    Console.ReadLine()
End Sub
```

Código fuente 64

Modificación de tamaño

Para modificar el tamaño o número de elementos de un array, emplearemos la instrucción ReDim, seguido del array a modificar y el nuevo tamaño. En el Código fuente 65, modificamos el tamaño de un array, añadiéndole dos elementos.

```
' array con 4 elementos: de 0 a 3
Dim Nombres(3) As String

' asignar valores al array
Nombres(0) = "Ana"
Nombres(1) = "Pedro"
Nombres(2) = "Antonio"
Nombres(3) = "Laura"

' ampliamos el array con 6 elementos: de 0 a 5
ReDim Nombres(5)
```

Código fuente 65

ReDim no toma el array existente y modifica su número de elementos, sino que internamente crea un nuevo array con el número de elementos indicado, por lo que se pierden los valores del array previo.

Para solucionar este inconveniente, debemos utilizar junto a ReDim, la palabra clave Preserve. Con ello, los valores existentes en el array a modificar son conservados. Ver Código fuente 66.

```
' ampliamos el array con 6 elementos: de 0 a 5
' y los valores de los elementos que hubiera, son conservados
ReDim Preserve Nombres(5)
```

Código fuente 66

Recorrer un array

Para recorrer todos los elementos de un array emplearemos la estructura de control For...Next, que ejecuta un bloque de código un número determinado de veces, y la función del lenguaje Ubound( ), que devuelve el número de elementos del array pasado como parámetro. Ver Código fuente 67.

```
Sub Main()
    ' crear un array y rellenarlo con valores
    Dim Nombres(3) As String

    Nombres(0) = "Ana"
    Nombres(1) = "Pedro"
    Nombres(2) = "Antonio"
    Nombres(3) = "Laura"

    ' recorrer el array y mostrar el contenido
    ' de cada uno de sus elementos
    Dim Contador As Integer
    For Contador = 0 To UBound(Nombres)
        Console.WriteLine("Posición del array: {0}, valor: {1}", _
            Contador, Nombres(Contador))
    Next

    Console.ReadLine()
End Sub
```

Código fuente 67

La estructura For...Next será explicada con más detalle en el apartado dedicado a las estructuras de control del lenguaje.

Constantes

Al igual que las variables, una constante es un elemento del lenguaje que guarda un valor, pero que en este caso y como su propio nombre indica, dicho valor será permanente a lo largo de la ejecución del programa, no pudiendo ser modificado.

Para declarar una constante, debemos utilizar la palabra clave Const, debiendo al mismo tiempo establecer el tipo de dato y asignarle valor. Ver Código fuente 68.

```
Sub Main()
    Const Color As String = "Azul"
    Const ValorMoneda As Double = 120.48
End Sub
```

Código fuente 68

La tipificación de una constante se rige, al igual que las variables, por la configuración que tengamos establecida para la instrucción Option Strict.

Si intentamos asignar un valor a una constante después de su asignación inicial, el IDE nos subrayará la línea con un aviso de error de escritura, y se producirá igualmente un error si intentamos ejecutar el programa. Ver Figura 171

```
Sub Main()
    Const Color As String = "Azul"
    Const ValorMoneda As Double = 120.48

    Color = "Verde"
          Una constante no puede ser el destino de una asignación.
End Sub
```

Figura 171. No es posible asignar valores a constantes después de su creación.

La ventaja del uso de constantes reside en que podemos tener un valor asociado a una constante, a lo largo de nuestro código para efectuar diversas operaciones. Si por cualquier circunstancia, dicho valor debe cambiarse, sólo tendremos que hacerlo en el lugar donde declaramos la constante.

Supongamos como ejemplo, que hemos escrito un programa en el que se realiza una venta de productos y se confeccionan facturas. En ambas situaciones debemos aplicar un descuento sobre el total resultante. Ver Código fuente 69.

```
Sub Main()
    ' venta de productos
    Dim Importe As Double
    Dim TotalVenta As Double
    Console.WriteLine("Introducir importe de la venta")
    Importe = Console.ReadLine()

    ' aplicar descuento sobre la venta
    TotalVenta = Importe - 100
    Console.WriteLine("El importe de la venta es: {0}", TotalVenta)
    Console.WriteLine()
    ' ............
    ' ............
    ' ............

    ' factura de mercancías
    Dim PrecioArt As Double
    Dim TotalFactura As Double
    Console.WriteLine("Introducir precio del artículo")
    PrecioArt = Console.ReadLine()

    ' aplicar descuento a la factura
    TotalFactura = PrecioArt - 100
    Console.WriteLine("El total de la factura es: {0}", TotalFactura)
    Console.WriteLine()
    ' ............
    ' ............
    ' ............
    Console.ReadLine()
End Sub
```

Código fuente 69

En el anterior ejemplo, realizamos el descuento utilizando directamente el valor a descontar. Si en un momento dado, necesitamos cambiar dicho valor de descuento, tendremos que recorrer todo el código e ir cambiando en aquellos lugares donde se realice esta operación.

Empleando una constante para el descuento, y utilizando dicha constante en todos aquellos puntos del código en donde necesitemos aplicar un descuento, cuando debamos modificar el descuento, sólo necesitaremos hacerlo en la línea en la que declaramos la constante. Ver Código fuente 70.

```
Sub Main()
    ' crear constante para calcular descuento
    Const DESCUENTO As Integer = 100

    ' venta de productos
    Dim Importe As Double
    Dim TotalVenta As Double
    Console.WriteLine("Introducir importe de la venta")
    Importe = Console.ReadLine()

    ' aplicar descuento sobre la venta, atención al uso de la constante
    TotalVenta = Importe - DESCUENTO
    Console.WriteLine("El importe de la venta es: {0}", TotalVenta)
    Console.WriteLine()
    ' .............
    ' .............
    ' .............

    ' factura de mercancías
    Dim PrecioArt As Double
    Dim TotalFactura As Double
    Console.WriteLine("Introducir precio del artículo")
    PrecioArt = Console.ReadLine()

    ' aplicar descuento a la factura, atención al uso de la constante
    TotalFactura = PrecioArt - DESCUENTO
    Console.WriteLine("El total de la factura es: {0}", TotalFactura)
    Console.WriteLine()
    ' .............
    ' .............
    ' .............
    Console.ReadLine()
End Sub
```

Código fuente 70

Conceptos mínimos sobre depuración

Para probar los ejemplos en este tema hemos utilizado hasta ahora la salida a consola. Sin embargo, pueden plantearse situaciones en las que la visualización por consola no sea suficiente, requiriendo un seguimiento línea a línea durante la ejecución del programa.

Ante esta tesitura debemos recurrir a un elemento imprescindible, que toda herramienta de desarrollo que se precie debe disponer: el depurador.

Un depurador nos permite introducirnos dentro del código de nuestro programa durante la ejecución del mismo, para observar qué es lo que está ocurriendo: ejecutar línea a línea el programa, observar el valor de las variables, etc., aspectos todos ellos fundamentales para el seguimiento de errores y fallos en la lógica de la aplicación.

VS.NET dispone de un excelente depurador; del que describiremos a continuación sus elementos más básicos, para que el lector pueda realizar un seguimiento más preciso de lo que sucede durante la ejecución de su aplicación.

Para ejecutar el programa en modo de depuración pulsaremos [F8], o seleccionaremos el menú *Depurar + Ir a instrucciones*. Cualquiera de estas acciones iniciarán el programa dentro del contexto del depurador, deteniendo la ejecución en la primera línea de código ejecutable, destacada en color amarillo. La línea marcada en amarillo indica que está a punto de ejecutarse, para ejecutarla y pasar a la siguiente línea pulsaremos de nuevo [F8], y así sucesivamente hasta llegar a la última línea del programa, donde se finalizará el mismo, cerrándose el depurador.

Podemos ver de forma inmediata el valor de una variable simplemente situando el cursor del ratón sobre ella, con lo que se mostrará una viñeta informativa de su valor. Ver Figura 172.

```
Sub Main()
    Dim ImporteFac As Integer
    Dim Precio As Double
    Dim Valor As String
    Dim FhActual As Date
    Dim FhNueva As Date
    Dim FhCompletaUno As Date
    Dim FhCompletaDos As Date
    Dim FhHora As Date

    ImporteFac = 875
    Preci [ImporteFac = 875]

    Valor = "mesa"

    FhActual = #5/20/2001#   ' mes/día/año
    FhNueva = "25/10/2001"   ' dia/mes/año
```

Figura 172. Ejecución del programa en el depurador.

Podemos también ver con detalle el valor que van adquiriendo las variables a lo largo de la ejecución, abriendo la ventana Locales del depurador, mediante el menú *Depurar + Ventanas + Locales*, o la pulsación [CTRL + ALT + V, L]. Ver Figura 173.

Locales

Nombre	Valor	Tipo
FhActual	#5/20/2001#	Date
FhCompletaDos	#12:00:00 AM#	Date
FhCompletaUno	#12:00:00 AM#	Date
FhHora	#12:00:00 AM#	Date
FhNueva	#12:00:00 AM#	Date
ImporteFac	875	Integer
Precio	50.75	Double
Valor	"mesa"	String

Figura 173. Ventana Locales del depurador.

En el caso de arrays, debemos hacer clic en el signo más (+) que aparece junto al nombre de la variable, para abrir y mostrar los elementos del array. Ver Figura 174.

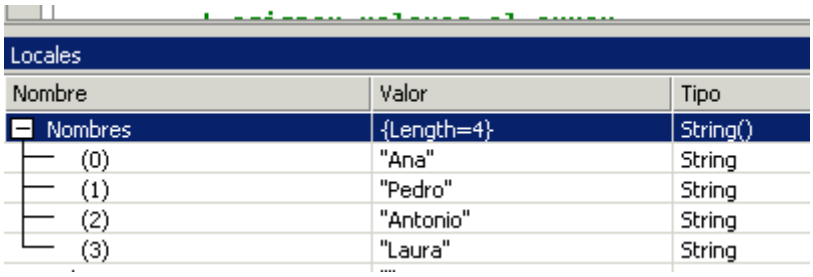


Figura 174. Ventana Locales del depurador, mostrando el contenido de un array.

Si en cualquier momento queremos continuar la ejecución normal del programa sin seguir usando el depurador, pulsaremos [F5].

13

Operadores del lenguaje

Los operadores son aquellos elementos del lenguaje que nos permiten combinar variables, constantes, valores literales, instrucciones, etc., para obtener un valor numérico, lógico, de cadena, etc., como resultado.

La combinación de operadores con variables, instrucciones, etc., se denomina expresión, mientras que a los elementos integrantes de una expresión y que no son operadores, se les denomina operandos.

En función de la complejidad de la operación a realizar, o del tipo de operador utilizado, una expresión puede ser manipulada a su vez como un operando dentro de otra expresión de mayor nivel.

Los operadores se clasifican en las categorías detalladas a continuación, según el tipo de expresión a construir.

Aritméticos

Efectúan el conjunto habitual de operaciones matemáticas.

Potenciación: ^

Eleva un número a determinada potencia. Debemos situar el número base a la izquierda de este operador, mientras que el exponente lo situaremos a la derecha.

Podemos realizar varias potenciaciones al mismo tiempo y utilizar números negativos. El valor devuelto será de tipo Double. Ver Código fuente 71.

```
Dim Resultado As Double

Resultado = 12 ^ 5  ' devuelve: 248832
Resultado = 2 ^ 3 ^ 7   ' devuelve: 2097152
Resultado = (-4) ^ 2    ' devuelve: 16
```

Código fuente 71

Multiplicación: *

Multiplica dos números. En el caso de que alguno de los operandos sea un valor nulo, se usará como cero. Ver Código fuente 72.

```
Dim Resultado As Double
Dim DatoSinValor As Integer
Dim Indefinido As Object

Resultado = 25 * 5  ' devuelve: 125

' la variable DatoSinValor no ha sido
' asignada, por lo que contiene cero
Resultado = 50 * DatoSinValor   ' devuelve: 0

' la variable Indefinido no ha sido
' asignada, por lo que contiene Nothing
Resultado = 25 * Indefinido   ' devuelve: 0

Resultado = 24.8 * 5.98  ' devuelve: 148.304
```

Código fuente 72

División real: /

Divide dos números, devolviendo un resultado con precisión decimal. Ver Código fuente 73.

```
Dim Resultado As Double

Resultado = 50 / 3  ' devuelve: 16.6666666666667
Resultado = 250 / 4 ' devuelve: 62.5
```

Código fuente 73

Por norma general, el valor devuelto será de tipo Double,. No obstante, si uno de los operandos es de tipo Single, el resultado será de tipo Single. De igual manera, si los dos operandos son del tipo de dato Decimal, el resultado también será un valor Decimal.

División entera: \

Divide dos números, devolviendo como resultado un valor numérico entero. Ver Código fuente 74.

```
Dim Resultado As Integer

Resultado = 50 \ 3  ' devuelve: 16
Resultado = 250 \ 4 ' devuelve: 62
```

Código fuente 74

Resto: Mod

Divide dos números y devuelve el módulo o resto de la división. Ver Código fuente 75.

```
Dim Resultado As Double

Resultado = 10 Mod 3    ' devuelve: 1
Resultado = 100 Mod 27  ' devuelve: 19
Resultado = 38 Mod 4    ' devuelve: 2
```

Código fuente 75

Suma: +

En función del tipo de dato de los operandos, este operador realiza una suma de números o una concatenación de cadenas de caracteres. Puede producirse un error dependiendo del tipo de dato del operando y la configuración de Option Strict. El Código fuente 76 muestra algunos ejemplos de suma y concatenación, con la instrucción Option Strict Off.

```
Sub Main()
    Dim Resultado As Double
    Dim Cadena As String
    Dim Valor As Integer
    Dim Nombre As String
    Dim CadenaResulta As String

    ' suma de números
    Resultado = 12 + 7              ' devuelve: 19
    Resultado = 450 + 130           ' devuelve: 580

    ' concatenación de cadenas
    Cadena = "hola " + "amigos"     ' devuelve: "hola amigos"

    ' suma de variables
    Cadena = "15"
    Valor = 20
    CadenaResulta = Cadena + Valor  ' devuelve: "35"

    ' operaciones incorrectas
    Valor = 25
    Nombre = "Alfredo"
```

```
    CadenaResulta = Valor + Nombre ' error
    Resultado = Valor + Nombre  ' error
End Sub
```

Código fuente 76

Si cambiamos a continuación la configuración a Option Strict On, la siguiente operación que antes se ejecutaba, ahora provocará un error. Ver Código fuente 77.

```
' suma de variables
Cadena = "15"
Valor = 20
CadenaResulta = Cadena + Valor   ' error
```

Código fuente 77

Para solucionar el problema debemos convertir explícitamente todos los operandos al mismo tipo de datos. Observe el lector que en esta situación, no se realiza una suma, sino una concatenación. Ver Código fuente 78.

```
' suma de variables
Cadena = "15"
Valor = 20
CadenaResulta = Cadena + CStr(Valor)   ' devuelve: "1520"
```

Código fuente 78

A pesar de que el operador + permite concatenar tipos String, se recomienda el uso del operador específico de concatenación &, que veremos más adelante.

Resta: -

Efectúa una resta entre dos números, o cambia el signo de un número (de positivo a negativo, y viceversa). Ver Código fuente 79.

```
Sub Main()
    Dim Resultado As Integer
    Dim Valor As Integer
    Dim OtroValor As Integer
    ' resta de números
    Resultado = 100 - 75
    ' cambiar a signo negativo un número
    Valor = -50
    ' volver a cambiar el signo de un número,
    ' estaba en negativo, con lo que vuelve
    ' a positivo
    OtroValor = -Valor
End Sub
```

Código fuente 79

Concatenación: &, +

Estos operadores permiten unir dos o más cadenas de caracteres para formar una única cadena. Se recomienda el uso de & para facilitar la legibilidad del código y evitar ambigüedades. El uso de + puede dar lugar a equívoco, ya que en muchas situaciones no sabremos a primera vista si se está realizando una suma o concatenación. Ver Código fuente 80.

```
Sub Main()
    Dim CadResulta As String
    Dim Nombre As String

    CadResulta = "esto es " & "una prueba"
    Console.WriteLine("Variable CadResulta: {0}", CadResulta)

    Nombre = "Juan"
    CadResulta = Nombre & " Almendro"
    Console.WriteLine("Variable CadResulta: {0}", CadResulta)
    Console.ReadLine()
End Sub
```

Código fuente 80

Operadores abreviados de asignación

Estos operadores simplifican la escritura de expresiones, facilitando la creación de nuestro código. El resultado empleado operadores abreviados en una expresión, es el mismo que utilizando la sintaxis normal, pero con un pequeño ahorro en la escritura de código. Cuando pruebe el lector estos ejemplos, ejecute por separado la sintaxis normal, y después la abreviada, para evitar resultados inesperados.

Potencia: ^=

Para elevar un número a una potencia podemos utilizar la sintaxis normal o abreviada. Ver Código fuente 81.

```
Dim Valor As Integer
Dim Resultado As Double

Valor = 3
Resultado = 2

'  sintaxis normal
Resultado = Resultado ^ Valor  '  devuelve: 8

'  sintaxis abreviada
Resultado ^= Valor '  devuelve: 8
```

Código fuente 81

Multiplicación: *=

Para multiplicar dos números podemos utilizar la sintaxis normal o abreviada. Ver Código fuente 82.

```
Dim Valor As Integer
Dim Resultado As Double

Valor = 7
Resultado = 12

' sintaxis normal
Resultado = Resultado * Valor   ' devuelve: 84

' sintaxis abreviada
Resultado *= Valor  ' devuelve: 84
```

Código fuente 82

División real: /=

Para dividir dos números, y obtener un resultado con precisión decimal, podemos utilizar la sintaxis normal o abreviada. Ver Código fuente 83.

```
Dim Valor As Integer
Dim Resultado As Double

Valor = 5
Resultado = 182

' sintaxis normal
Resultado = Resultado / Valor   ' devuelve: 36.4

' sintaxis abreviada
Resultado /= Valor  ' devuelve: 36.4
```

Código fuente 83

División entera: \=

Para dividir dos números, con un resultado entero, podemos utilizar la sintaxis normal o abreviada. Ver Código fuente 84.

```
Dim Valor As Integer
Dim Resultado As Double


Valor = 5
Resultado = 182


' sintaxis normal
Resultado = Resultado \ Valor   ' devuelve: 36
```

```
' sintaxis abreviada
Resultado \= Valor  ' devuelve: 36
```

Código fuente 84

Suma: +=

Podemos sumar números, o concatenar cadenas utilizando la sintaxis normal o abreviada. Ver Código fuente 85.

```
Dim Valor As Integer
Dim Resultado As Double
Dim CadenaA As String
Dim CadenaB As String

' con valores numéricos
Valor = 69
Resultado = 200

' sintaxis normal
Resultado = Resultado + Valor   ' devuelve: 269

' sintaxis abreviada
Resultado += Valor  ' devuelve: 269

' con cadenas de caracteres
CadenaA = " varios numeros"
CadenaB = "589"
CadenaB += CadenaA  ' devuelve: "589 varios numeros"
```

Código fuente 85

Resta: -=

Podemos restar números utilizando la sintaxis normal o abreviada. Ver Código fuente 86.

```
Dim Valor As Integer
Dim Resultado As Double

Valor = 69
Resultado = 200

' sintaxis normal
Resultado = Resultado - Valor   ' devuelve: 131

' sintaxis abreviada
Resultado -= Valor  ' devuelve: 131
```

Código fuente 86

Concatenación: &=

Para concatenar dos cadenas, podemos emplear la sintaxis normal o abreviada. Ver Código fuente 87.

```
Dim PrimeraCad As String
Dim SegundaCad As String

PrimeraCad = "Aquí va "
SegundaCad = "una prueba"

' sintaxis normal
PrimeraCad = PrimeraCad & SegundaCad    ' devuelve: "Aquí va una prueba"

' sintaxis abreviada
PrimeraCad &= SegundaCad    ' devuelve: "Aquí va una prueba"
```

Código fuente 87

Comparación

Estos operadores permiten comprobar el nivel de igualdad o diferencia existente entre los operandos de una expresión. El resultado obtenido será un valor lógico, True (Verdadero) o False (Falso). La Tabla 6 muestra la lista de los operadores disponibles de este tipo.

Operador		El resultado es Verdadero cuando	El resultado es Falso cuando
<	Menor que	ExpresiónA < ExpresiónB	ExpresiónA >= ExpresiónB
<=	Menor o igual que	ExpresiónA <= ExpresiónB	ExpresiónA > ExpresiónB
>	Mayor que	ExpresiónA > ExpresiónB	ExpresiónA <= ExpresiónB
>=	Mayor o igual que	ExpresiónA >= ExpresiónB	ExpresiónA < ExpresiónB
=	Igual a	ExpresiónA = ExpresiónB	ExpresiónA <> ExpresiónB
<>	Distinto de	ExpresiónA <> ExpresiónB	ExpresiónA = ExpresiónB

Tabla 6. Operadores de comparación.

El Código fuente 88 nos muestra algunas expresiones de comparación utilizando números.

```
Dim Resultado As Boolean
Resultado = 10 < 45     ' devuelve: True
Resultado = 7 <= 7      ' devuelve: True
Resultado = 25 > 50     ' devuelve: False
Resultado = 80 >= 100   ' devuelve: False
Resultado = 120 = 220   ' devuelve: False
Resultado = 5 <> 58     ' devuelve: True
```

Código fuente 88

Comparación de cadenas

Podemos utilizar los operadores de comparación antes descritos para comparar también cadenas de caracteres. La instrucción *Option Compare*, junto a sus modificadores Binary/Text, nos permite definir el modo en que se realizarán las comparaciones entre expresiones que contengan cadenas.

- **Option Compare Binary**. Las comparaciones se realizan en base a los valores binarios internos de los caracteres. Esta es la opción por defecto.

- **Option Compare Text**. Las comparaciones se realizan en base a los valores textuales de los caracteres.

Podemos configurar Option Compare a nivel de proyecto y de fichero de código. En el caso de configurar a nivel de proyecto, deberemos abrir la ventana de propiedades del proyecto, y en su apartado Generar, establecer el valor correspondiente en la lista desplegable. Ver Figura 175.

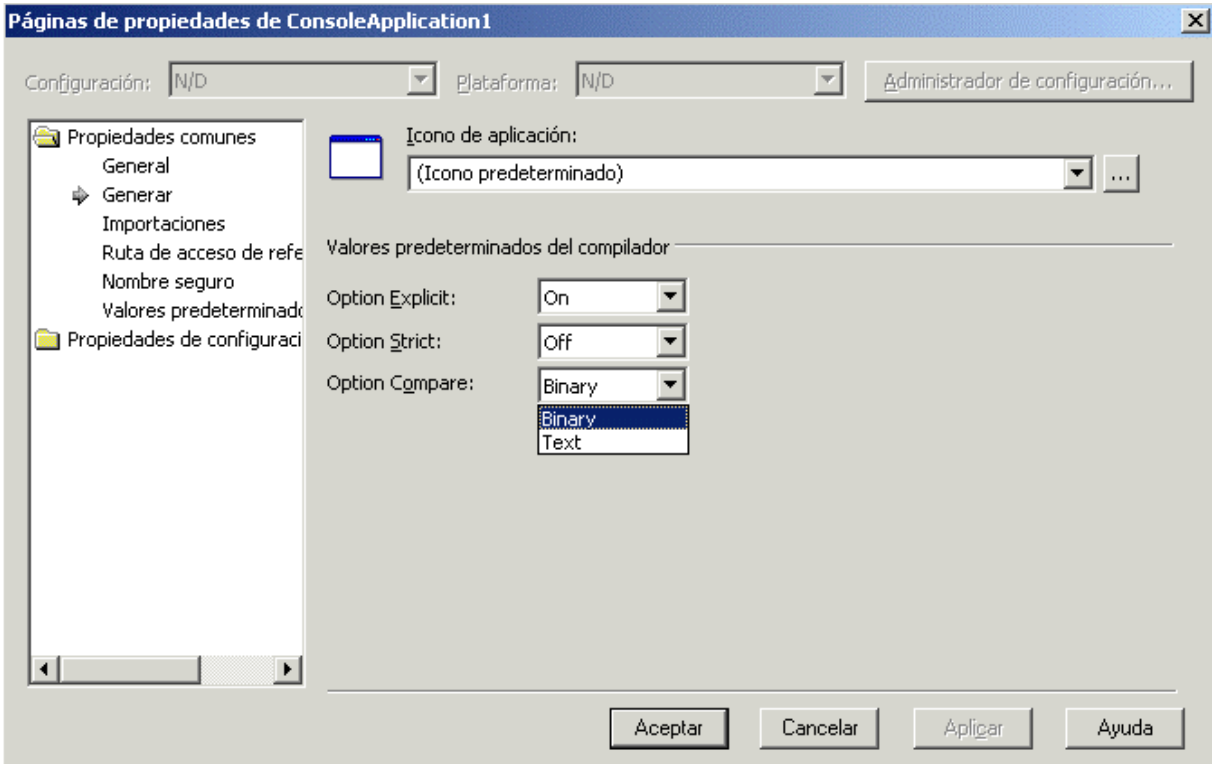


Figura 175. Configuración de Option Compare.

Si configuramos a nivel de fichero de código, escribiremos esta instrucción en la cabecera del fichero con el modificador oportuno. Consulte el lector el apartado sobre declaración obligatoria de variables, para un mayor detalle sobre el acceso a esta ventana de propiedades del proyecto.

En el Código fuente 89 tenemos un ejemplo de comparación de cadenas utilizando Option Compare Binary.

```
Option Compare Binary

Module Module1
```

```
    Sub Main()
        Dim Resultado As Boolean

        Resultado = "A" = "a"   ' devuelve: False
        Resultado = "M" < "Z"   ' devuelve: True
        Resultado = "M" > "m"   ' devuelve: False
        Resultado = "F" <> "f"  ' devuelve: True

    End Sub

End Module
```

Código fuente 89

El motivo de que la comparación “A” con “a” devuelva falso, o de que “M” no sea mayor que “m” se debe a que lo que se comparan son los valores binarios, o códigos que sirven para representar a cada carácter. Por ejemplo, el código de “M” es 77, mientras que el de “m” es 109, por lo que al ser este último mayor, la comparación realizada en el fuente de ejemplo devuelve False.

Si a continuación, cambiamos la configuración de Option Compare a Text y realizamos las mismas comparaciones, en algunos casos obtendremos resultados diferentes. Ver Código fuente 90.

```
Option Compare Text

Module Module1

    Sub Main()
        Dim Resultado As Boolean

        Resultado = "A" = "a"   ' devuelve: True
        Resultado = "M" < "Z"   ' devuelve: True
        Resultado = "M" > "m"   ' devuelve: False
        Resultado = "F" <> "f"  ' devuelve: False
    End Sub

End Module
```

Código fuente 90

En esta ocasión “A” y “a” si son iguales, debido a que se comparan sus valores como texto y no como los códigos internos utilizados para representar los caracteres. De igual forma, se devuelve falso en la expresión que comprueba si “F” y “f” son distintos, ya que bajo esta configuración, ambos caracteres se consideran iguales.

La función Asc( )

Cuando realizamos comparaciones entre cadenas, basadas en los valores binarios de los caracteres, es útil en ocasiones conocer el código de dichos caracteres. Para averiguar cuál es el código correspondiente a un determinado carácter, el lenguaje nos proporciona la función Asc( ).

Esta función recibe como parámetro una cadena, y devuelve un valor numérico de tipo Integer, con el código correspondiente al primer carácter de la cadena. El Código fuente 91 nos muestra algunos ejemplos.

```
Dim CodigoCar As Integer

CodigoCar = Asc("A")     ' devuelve: 65
CodigoCar = Asc("a")     ' devuelve: 97
CodigoCar = Asc("M")     ' devuelve: 77
CodigoCar = Asc("F")     ' devuelve: 70
CodigoCar = Asc("f")     ' devuelve: 102
CodigoCar = Asc("hola") ' devuelve: 104
```

Código fuente 91

La función Chr( )

Si nos encontramos en la situación inversa a la descrita en el apartado anterior, es decir, tenemos el código de un carácter y queremos saber a cuál corresponde, la función Chr( ) recibe un número como parámetro y devuelve el carácter al que pertenece como un dato de tipo Char, aunque también podemos asignar el resultado a una variable String. Veamos unos ejemplos en el Código fuente 92.

```
Dim MiCaracter As Char
Dim MiCadena As String

MiCaracter = Chr(65)     ' devuelve: "A"
MiCaracter = Chr(70)     ' devuelve: "F"
MiCadena = Chr(77)       ' devuelve: "M"
MiCadena = Chr(102)      ' devuelve: "f"
```

Código fuente 92

Comparación de cadenas en base a un patrón. El operador Like

El operador Like permite realizar una comparación entre dos cadenas, en base a un patrón establecido en una de ellas. El formato de uso se muestra en el Código fuente 93.

```
Resultado = Cadena Like Patrón
```

Código fuente 93

- **Resultado**. Valor lógico con el resultado de la comparación. Verdadero indica que hay una coincidencia de Cadena con Patrón. Falso indica que no se ha producido coincidencia de Cadena con Patrón.

- **Cadena**. Cadena de caracteres que se compara con el patrón de coincidencia.

- **Patrón**. Cadena de caracteres en donde se especifican los caracteres especiales que sirven de patrón de coincidencia respecto al valor de Cadena. La Tabla 7 muestra los caracteres y convenciones de uso establecidas por el lenguaje para el uso de patrones de comparación.

Carácter del patrón	*Coincidencia en la cadena a buscar*
?	Cualquier único carácter
*	Varios caracteres o ninguno
#	Cualquier único número
[ListaCaracteres]	Cualquier único carácter que se encuentre dentro de la lista.
[!ListaCaracteres]	Cualquier único carácter que no se encuentre dentro de la lista

Tabla 7. Caracteres patrón del operador Like.

Debemos tener en cuenta que los resultados obtenidos en expresiones que utilicen este operador estarán condicionadas por la configuración establecida mediante Option Compare. Revise el lector el apartado sobre comparación de cadenas en donde se describe esta instrucción.

Cuando utilicemos los corchetes para establecer una lista de caracteres a comparar, debemos emplear el guión (-) como separador de rangos. Si necesitamos que alguno de los caracteres patrón estén entre los que vamos a buscar, debemos encerrarlo entre corchetes. El Código fuente 94 muestra algunos ejemplos de uso de este operador.

```
' ejemplos con el operador Like
Dim Resultado As Boolean
' -------------------------------
' patrón ?

' devuelve True - El patrón coincide con la cadena
' al hacer la sustitución de un carácter
Resultado = "HOLA" Like "HO?A"

' devuelve True - El patrón coincide con la cadena
' al hacer la sustitución de dos caracteres
Resultado = "MONITOR" Like "MO?ITO?"

' devuelve False - El patrón no coincide con la cadena
' al hacer la sustitución de un carácter
Resultado = "ROEDOR" Like "R?DEO"
' -------------------------------
' patrón *

' devuelve True - El patrón coincide con la cadena
' al hacer la sustitución de varios caracteres con
' espacio en blanco a ambos lados
Resultado = "La gran llanura" Like "La * llanura"

' devuelve True - El patrón coincide con la cadena
' al hacer la sustitución de dos grupos de caracteres
Resultado = "La gran llanura" Like "La*llanu*"

' devuelve False - El patrón no coincide con la cadena
' al hacer la sustitución de un grupo de caracteres ,
' puesto que en el patrón falta una palabra que sí
' se halla en la cadena
```

```
Resultado = "La gran llanura" Like "La llanu*"
' --------------------------------
' patrón #

' devuelve True - El patrón coincide con la cadena
' al hacer la sustitución de dos números
Resultado = "Ha ganado 128 millones" Like "Ha ganado ##8 millones"

' devuelve False - El patrón no coincide con la cadena,
' ya que en el patrón se especifican más dígitos de los
' existentes en la cadena
Resultado = "Ha ganado 128 millones" Like "Ha ganado ###8 millones"

' devuelve False - El patrón no coincide con la cadena,
' ya que en el patrón se utilizan caracteres de sustitución
' de dígitos incorrectamente
Resultado = "Ha ganado 128 millones" Like "Ha ganado 128 ##llones"

' --------------------------------
' patrón [Lista]

' devuelve True - El carácter de la cadena se encuentra
' dentro del rango en la lista del patrón
Resultado = "H" Like "[A-M]"

' devuelve False - El carácter de la cadena no se encuentra
' dentro del rango en la lista del patrón
Resultado = "h" Like "[A-M]"

' devuelve True - El carácter de la cadena se encuentra
' dentro del rango en la lista del patrón
Resultado = "h" Like "[a-m]"

' devuelve True - El carácter de la cadena no se encuentra
' dentro del rango en la lista del patrón
Resultado = "D" Like "[!P-W]"

' devuelve False - El carácter de la cadena se encuentra
' dentro del rango en la lista del patrón
Resultado = "R" Like "[!P-W]"

' --------------------------------
' combinación de varios caracteres patrón

' devuelve True - Todas las sustituciones del patrón son correctas
Resultado = "Faltan 48 horas para llegar a destino" Like _
    "Fal* ## * para ll[a-g]gar ? des*"

' devuelve False - Las sustituciones de caracteres númericos son incorrectas
Resultado = "Faltan 48 horas para llegar a destino" Like _
    "Fal## * para ll[a-g]gar ? des*"
' --------------------------------
' comparación utilizando caracteres patrón
' dentro de la expresión

' devuelve True - El carácter de cierre de interrogación
' se sustituye correctamente al encerrarse entre corchetes
Resultado = "¿Ha llegado Ana?, bienvenida" Like "¿Ha*Ana[?], bienvenida"
' --------------------------------
' comparación de dos cadenas vacías

' devuelve True
Resultado = "" Like ""
```

Código fuente 94

Comparación de objetos. El operador Is

El operador Is permite comparar si dos variables que contienen objetos apuntan o no a la misma referencia o instancia del objeto. Para conceptos básicos sobre objetos, consulte el lector el tema dedicado a la programación orientada a objetos en este mismo texto.

El Código fuente 95 muestra el formato de uso para este operador.

```
Resultado = ObjetoA Is ObjetoB
```

Código fuente 95

Para probar este operador podemos crear una aplicación de tipo Windows y añadir un módulo en el que escribiríamos un procedimiento Main( ). Después de configurar el proyecto para que se inicie por este procedimiento, escribiremos las líneas que se muestran en el Código fuente 96.

```
Public Sub Main()
    ' declarar dos variables que
    ' contendran objetos de la clase Form
    Dim VentanaUno As Form
    Dim VentanaDos As Form

    Dim Resultado As Boolean

    ' crear dos instancias de la clase Form
    ' asignando cada uno de los objetos
    ' a las variables
    VentanaUno = New Form()
    VentanaDos = New Form()

    ' la expresión de comparación con Is devuelve
    ' False ya que las variables tienen referencias
    ' a objetos diferentes, aunque sean de la misma clase
    Resultado = VentanaUno Is VentanaDos
End Sub
```

Código fuente 96

Como hemos podido comprobar, al comparar las variables del anterior fuente con Is, el resultado es False, ya que ambos objetos son instancias diferentes, aunque pertenezcan a la misma clase: Form.

Si por el contrario, creamos una única instancia de un objeto y la asignamos a las dos variables, el resultado será muy diferente. En este caso el operador Is devolverá True ya que ambas variables contienen el mismo objeto. Ver Código fuente 97.

```
Public Sub Main()
    ' declarar dos variables que
    ' contendran objetos de la clase Form
    Dim VentanaUno As Form
    Dim VentanaDos As Form

    Dim Resultado As Boolean
```

```
    ' crear una única instancia de la clase Form,
    ' el objeto resultante se asigna a una variable
    VentanaUno = New Form()

    ' después el mismo objeto que ya está
    ' en una variable se asigna a la otra variable
    VentanaDos = VentanaUno

    ' ambas variables contienen una referencia
    ' al mismo objeto, por lo que la expresión
    ' de comparación Is devuelve True
    Resultado = VentanaUno Is VentanaDos
End Sub
```

Código fuente 97

Lógicos y a nivel de bit

Los operadores lógicos devuelven un valor de tipo Boolean (True o False), en base a una condición establecida entre los operandos de la expresión. En expresiones que impliquen el uso de operadores lógicos, es habitual que los operandos sean a su vez expresiones, como veremos en los próximos ejemplos con este tipo de operadores.

El Código fuente 98 muestra el formato de uso para estos operadores.

```
Resultado = ExpresiónA OperadorLogico ExpresiónB
```

Código fuente 98

Cuando los operandos que forman parte de la expresión son numéricos, la evaluación de la expresión se realiza a nivel de bit, es decir, comparando los bits de las posiciones equivalentes de ambos números y obteniendo igualmente, un valor numérico como resultado.

And

A nivel lógico, este operador realiza una conjunción entre dos expresiones. La Tabla 8 muestra los diferentes resultados obtenidos con el uso de este operador en función de los valores que tengan sus expresiones.

Cuando la ExpresiónA devuelve	**Y la ExpresiónB devuelve**	**El resultado es**
True	True	True
True	False	False
False	True	False
False	False	False

Tabla 8. Tabla de valores lógicos del operador And.

El Código fuente 99 muestra algunos ejemplos a nivel lógico con este operador.

```
Dim Resultado As Boolean

Resultado = 58 > 20 And "H" = "H"   ' devuelve: True
Resultado = "H" = "H" And 720 < 150 ' devuelve: False
Resultado = 8 <> 8 And 62 < 115     ' devuelve: False
Resultado = "W" > "b" And "Q" = "R" ' devuelve: False
```

Código fuente 99

A nivel de bit, And realiza las operaciones mostradas en la Tabla 9.

Cuando el bit de ExpresiónA es	**Y el bit de ExpresiónB es**	**El valor del bit resultante es**
0	0	0
0	1	0
1	0	0
1	1	1

Tabla 9. Tabla de valores a nivel de bit del operador And.

El Código fuente 100 muestra algunos ejemplos a nivel de bit con este operador.

```
Dim Resultado As Integer

Resultado = 15 And 8    ' devuelve: 8
Resultado = 6 And 45    ' devuelve: 4
```

Código fuente 100

Uso de paréntesis para mejorar la legibilidad de expresiones

Los ejemplos a nivel lógico del apartado anterior, si bien se ejecutan correctamente, pueden ser un tanto confusos a la hora de leer, ya que al tratarse de una operación lógica, cada operando es a su vez una expresión.

Para facilitar la lectura y compresión en expresiones sobre todo lógicas, podemos encerrar cada operando-expresión entre paréntesis. Ver Código fuente 101.

```
Dim Resultado As Boolean

Resultado = (58 > 20) And ("H" = "H")   ' devuelve: True
Resultado = ("H" = "H") And (720 < 150) ' devuelve: False
```

```
Resultado = (8 <> 8) And (62 < 115)     ' devuelve: False
Resultado = ("W" > "b") And ("Q" = "R") ' devuelve: False
```

Código fuente 101

Como puede comprobar el lector al ejecutar, el resultado es el mismo que si no utilizamos paréntesis, pero la claridad al leer estas líneas de código es mucho mayor.

Not

A nivel lógico, este operador realiza una negación entre dos expresiones. Su formato es ligeramente distinto del resto de operadores lógicos, como vemos en el Código fuente 102.

```
Resultado = Not Expresión
```

Código fuente 102

La Tabla 10 muestra los resultados obtenidos con el uso de este operador en función de su expresión.

Cuando la Expresión devuelve	**El resultado es**
True	False
False	True

Tabla 10. Tabla de valores lógicos del operador Not.

El Código fuente 103 muestra algunos ejemplos a nivel lógico con este operador.

```
Dim Operacion As Boolean
Dim Resultado As Boolean

Operacion = 100 > 60
Resultado = Not Operacion   ' devuelve: False

Resultado = Not (28 > 50)   ' devuelve: True
```

Código fuente 103

A nivel de bit, Not realiza las operaciones mostradas en la Tabla 11.

Cuando el bit de la Expresión devuelve	**El resultado es**
0	1

1	0

Tabla 11. Tabla de valores a nivel de bit del operador Not.

El Código fuente 104 muestra algunos ejemplos a nivel de bit con este operador.

```
Dim Resultado As Integer

Resultado = Not 16  ' devuelve: -17
Resultado = Not 4   ' devuelve: -5
```

Código fuente 104

Or

A nivel lógico, este operador realiza una disyunción entre dos expresiones. La Tabla 12 muestra los diferentes resultados obtenidos con el uso de este operador en función de los valores que tengan sus expresiones.

Cuando la ExpresiónA devuelve	**Y la ExpresiónB devuelve**	**El resultado es**
True	True	True
True	False	True
False	True	True
False	False	False

Tabla 12. Tabla de valores lógicos del operador Or.

El Código fuente 105 muestra algunos ejemplos a nivel lógico con este operador.

```
Dim Resultado As Boolean

Resultado = (58 > 20) Or ("H" = "H")     ' devuelve: True
Resultado = ("H" = "H") Or (720 < 150)   ' devuelve: True
Resultado = (8 <> 8) Or (62 < 115)       ' devuelve: True
Resultado = ("W" > "b") Or ("Q" = "R")   ' devuelve: False
```

Código fuente 105

A nivel de bit, Or realiza las operaciones mostradas en la Tabla 13.

Cuando el bit de ExpresiónA es	Y el bit de ExpresiónB es	El valor del bit resultante es
0	0	0
0	1	1
1	0	1
1	1	1

Tabla 13. Tabla de valores a nivel de bit del operador Or.

El Código fuente 106 muestra algunos ejemplos a nivel de bit con este operador.

```
Dim Resultado As Integer

Resultado = 15 Or 8    ' devuelve: 15
Resultado = 6 Or 45    ' devuelve: 47
```

Código fuente 106

Xor

A nivel lógico, este operador realiza una exclusión entre dos expresiones. La Tabla 14 muestra los diferentes resultados obtenidos con el uso de este operador en función de los valores que tengan sus expresiones.

Cuando la ExpresiónA devuelve	Y la ExpresiónB devuelve	El resultado es
True	True	False
True	False	True
False	True	True
False	False	False

Tabla 14. Tabla de valores lógicos del operador Xor.

El Código fuente 107 muestra algunos ejemplos a nivel lógico con este operador.

```
Dim Resultado As Boolean

Resultado = (58 > 20) Xor ("H" = "H")     ' devuelve: False
Resultado = ("H" = "H") Xor (720 < 150)   ' devuelve: True
Resultado = (8 <> 8) Xor (62 < 115)       ' devuelve: True
```

```
Resultado = ("W" > "b") Xor ("Q" = "R")  ' devuelve: False
```

Código fuente 107

A nivel de bit, Xor realiza las operaciones mostradas en la Tabla 15.

Cuando el bit de ExpresiónA es	Y el bit de ExpresiónB es	El valor del bit resultante es
0	0	0
0	1	1
1	0	1
1	1	0

Tabla 15. Tabla de valores a nivel de bit del operador Or.

El Código fuente 108 muestra algunos ejemplos a nivel de bit con este operador.

```
Dim Resultado As Integer

Resultado = 15 Xor 8      ' devuelve: 7
Resultado = 6 Xor 45    ' devuelve: 43
```

Código fuente 108

AndAlso

Este operador realiza una conjunción lógica de tipo cortocircuito entre dos expresiones. En este tipo de operación, en cuanto la primera expresión devuelva falso como resultado, el resto no será evaluado devolviendo falso como resultado final.

La Tabla 16 muestra los diferentes resultados obtenidos con el uso de este operador en función de los valores que tengan sus expresiones.

Cuando la ExpresiónA devuelve	Y la ExpresiónB devuelve	El resultado es
True	True	True
True	False	False
False	No se evalúa	False

Tabla 16. Tabla de valores lógicos del operador AndAlso.

El Código fuente 109 muestra algunos ejemplos con este operador.

```
Dim Resultado As Boolean

Resultado = (58 > 20) AndAlso ("H" = "H")   ' devuelve: True
Resultado = ("H" = "H") AndAlso (720 < 150) ' devuelve: False
Resultado = (8 <> 8) AndAlso (62 < 115)     ' devuelve: False
```

Código fuente 109

OrElse

Este operador realiza una disyunción lógica de tipo cortocircuito entre dos expresiones. En este tipo de operación, en cuanto la primera expresión devuelva verdadero como resultado, el resto no será evaluado devolviendo verdadero como resultado final.

La muestra los diferentes resultados obtenidos con el uso de este operador en función de los valores que tengan sus expresiones.

Cuando la ExpresiónA devuelve	**Y la ExpresiónB devuelve**	El resultado es
True	No se evalúa	True
False	True	True
False	False	False

Tabla 17. Tabla de valores lógicos del operador OrElse.

El Código fuente 110 muestra algunos ejemplos con este operador.

```
Dim Resultado As Boolean

Resultado = ("H" = "H") OrElse (720 < 150)  ' devuelve: True
Resultado = (8 <> 8) OrElse (62 < 115)      ' devuelve: True
Resultado = ("W" > "b") OrElse ("Q" = "R")  ' devuelve: False
```

Código fuente 110

Prioridad de operadores

Dentro de una línea de código que contenga varias operaciones, estas se resolverán en un orden predeterminado conocido como prioridad de operadores. Dicha prioridad se aplica tanto entre los operadores de un mismo grupo como entre los distintos grupos de operadores.

Prioridad entre operadores del mismo grupo.

Los operadores aritméticos se ajustan a la prioridad indicada en la Tabla 18.

Prioridad de operadores aritméticos
Potenciación (^)
Negación (-)
Multiplicación y división real (* , /)
División entera (\ )
Resto de división (Mod)
Suma y resta (+ , -)

Tabla 18. Prioridad de operadores aritméticos.

El operador de mayor prioridad es el de potenciación, los de menor son la suma y resta. En el caso de operadores con idéntica prioridad como multiplicación y división, se resolverán en el orden de aparición, es decir, de izquierda a derecha. Veamos un ejemplo en el Código fuente 111

```
Dim Resultado As Long

Resultado = 5 + 8 ^ 2 * 4   ' devuelve: 261
```

Código fuente 111

Los operadores de comparación tienen todos la misma prioridad, resolviéndose en el orden de aparición dentro de la expresión.

Los operadores lógicos se ajustan a la prioridad indicada en la Tabla 19.

Prioridad de operadores lógicos
Negación (Not)
Conjunción (And, AndAlso)
Disyunción (Or, OrElse, Xor)

Tabla 19. Prioridad de operadores lógicos.

En el ejemplo del Código fuente 112, el resultado final de la operación es True debido a que el operador Not cambia la segunda expresión a True, resultando las dos expresiones de la operación True.

```
Dim Resultado As Boolean
```

```
Resultado = 10 < 70 And Not 30 = 20 ' devuelve: True
```

Código fuente 112

Prioridad entre operadores de distintos grupos.

Cuando una expresión contenga operadores de distintos grupos, estos se resolverán en el orden marcado por la Tabla 20.

Prioridad entre operadores de distintos grupos
Aritméticos
Concatenación
Comparación
Lógicos

Tabla 20. Prioridad entre grupos de operadores.

El Código fuente 113 muestra un ejemplo de expresión en el que intervienen operadores de diferentes tipos.

```
Dim Resultado As Boolean

Resultado = 30 + 5 * 5 > 100 And 52 > 10    ' devuelve: False
```

Código fuente 113

Uso de paréntesis para alterar la prioridad de operadores

Podemos alterar el orden natural de prioridades entre operadores utilizando los paréntesis, encerrando entre ellos los elementos de una expresión que queramos sean resueltos en primer lugar. De esta forma, se resolverán en primer lugar las operaciones que se encuentren en los paréntesis más interiores, finalizando por las de los paréntesis exteriores. Es importante tener en cuenta, que dentro de los paréntesis se seguirá manteniendo la prioridad explicada anteriormente.

El Código fuente 114 en condiciones normales, devolvería False como resultado. Sin embargo, gracias al uso de paréntesis, cambiamos la prioridad predeterminada, obteniendo finalmente True.

```
Dim Resultado As Boolean

Resultado = ((30 + 5) * 5 > 100) And (52 > 200 / (2 + 5)) ' devuelve: True
```

Código fuente 114.

14

Rutinas de código

División de una línea de código

En el tema dedicado al IDE, el apartado *Ajuste de línea* mostraba como podemos configurar el editor de código para que una línea de código muy larga se divida en varios fragmentos, de modo que esté siempre visible al completo y no quede parte de ella oculta a la derecha de la ventana.

Si no queremos utilizar esta característica del IDE, y preferimos dividir nosotros manualmente una línea lógica de código en varias líneas físicas, lo podemos conseguir situando el carácter de guión bajo (_) en el punto de la línea de código en donde queremos continuar, teniendo en cuenta que siempre debe haber un espacio en blanco antes y después de este carácter, para que la división de la línea sea efectiva.

En el Código fuente 115 podemos ver dos líneas de código exactamente iguales, la primera se muestra en una sola línea física, mientras que la segunda ha sido fraccionada en tres líneas, aunque a efectos de compilación el resultado es el mismo.

```
Dim Resultado As Boolean


' una sola línea lógica y física
Resultado = ((30 + 5) * 5 > 100) And (52 > 200 / (2 + 5))


' varias líneas físicas, aunque internamente
' el compilador reconoce una sola línea lógica
Resultado = ((30 + 5) * 5 > 100) And _
```

```
    (52 > 200 / _
    (2 + 5))
```

Código fuente 115

Escritura de varias sentencias en la misma línea

Aquí tenemos el caso opuesto al anterior apartado. El lenguaje nos permite escribir en una misma línea física, varias sentencias separadas por el carácter de dos puntos (:), que en condiciones normales se escriben en líneas separadas. Ver Código fuente 116.

```
Sub Main()
    Dim Valor As Integer, Nombre As String, Resultado As Boolean

    ' en la siguiente línea hay tres sentencias que
    ' habitualmente se escriben en líneas separadas
    Valor = 50 : Nombre = "Julio" : Resultado = 50 <> 275

    Console.WriteLine("Contenido de variables: {0} - {1} - {2}", _
        Valor, Nombre, Resultado)
    Console.ReadLine()
End Sub
```

Código fuente 116

Si bien en algunas situaciones puede ser útil, esta característica hace que nuestro código se vuelva más complicado de leer, restándole claridad a nuestra aplicación, por lo que recomendamos no utilizarla salvo en casos muy necesarios.

Procedimientos

Todo el código ejecutable de una aplicación se ubica en *rutinas de código* o *procedimientos*. Un procedimiento es un elemento del lenguaje compuesto por un conjunto de líneas de código, a las que se denomina cuerpo del procedimiento. Su comienzo y fin lo establecemos mediante ciertas palabras reservadas del lenguaje, asociándole un identificador, que nos servirá para reconocerlo entre el resto de procedimientos creados en el programa. Podemos enviarle también información adicional en forma de parámetros, con lo que el resultado de la ejecución de un procedimiento variará según los valores que pasemos en cada llamada.

En VB.NET disponemos de los siguientes tipos de procedimientos:

- **Sub**. Procedimiento que realiza un conjunto de operaciones pero no devuelve valor al punto de llamada. A lo largo del texto también nos referiremos a las rutinas de tipo Sub con el nombre genérico de *procedimiento*.

- **Function**. Procedimiento que realiza un conjunto de operaciones, y devuelve un valor denominado valor de retorno al punto de código que realizó la llamada. A lo largo del texto también nos referiremos a las rutinas de tipo Function con el nombre genérico de *función.*

- **Property**. Procedimiento que se utiliza para labores de acceso y asignación de valores a las propiedades de un objeto. Serán tratados con más profundidad en el tema dedicado a la programación orientada a objetos.

Sintaxis de un procedimiento Sub

El formato para la escritura de un procedimiento Sub se muestra en el Código fuente 117.

```
[Ámbito] Sub NombreProcedimiento[(ListaParámetros)]
       [CódigoEjecutable]
       [Exit Sub | Return]
       [CódigoEjecutable]
End Sub
```

Código fuente 117

Los elementos que forman parte de este tipo de rutina son los siguientes:

- **Ámbito**. Define el modo en que vamos a poder acceder o llamar al procedimiento desde otro punto de la aplicación. El ámbito de los elementos del lenguaje será tratado en un apartado posterior.

- **Sub...End Sub**. Palabras clave que indican el comienzo y final del procedimiento respectivamente. Cuando hagamos una llamada al procedimiento, el compilador ejecutará el código comprendido entre estas dos palabras clave.

- **NombreProcedimiento**. Identificador que utilizamos para reconocer y llamar al procedimiento.

- **ListaParámetros**. Lista de variables separadas por comas, y encerradas entre paréntesis, que representan la información que recibe el procedimiento desde el código llamador.

- **Return**. Esta palabra clave permite salir de la ejecución del procedimiento sin haber llegado a su fin. Podemos utilizarla en tantos lugares dentro de un procedimiento como sea necesario. Se recomienda su uso en lugar de Exit Sub, ya que podemos emplear Return para salir de cualquier tipo de procedimiento, con lo cual se unifica la escritura del código.

- **Exit Sub**. Al igual que en el punto anterior, esta palabra clave permite salir de la ejecución del procedimiento sin haber llegado a su fin, pudiendo igualmente, situarla en tantos lugares dentro del procedimiento como sea necesario.

El Código fuente 118 muestra el modo más simple de crear un procedimiento. Escriba el lector este procedimiento en la aplicación de consola sobre la que está realizando las pruebas, a continuación de Main( ).

```
Sub Prueba()
    Console.WriteLine("Estamos en el procedimiento Prueba")
End Sub
```

Código fuente 118

Llamada a un procedimiento Sub

Para realizar una llamada o ejecutar un procedimiento Sub, debemos escribir su nombre en un punto del programa. El Código fuente 119 muestra el código al completo del módulo de nuestra aplicación de consola. La ejecución de este programa comienza como es habitual por Main( ), dentro del cual se realiza una llamada al procedimiento Prueba( ).

```
Module Module1

    Sub Main()
        Console.WriteLine("Estamos en el procedimiento Main")

        ' llamada a un procedimiento
        Prueba()

        Console.ReadLine()
    End Sub

    Sub Prueba()
        Console.WriteLine("Estamos en el procedimiento Prueba")
    End Sub

End Module
```

Código fuente 119

En la llamada a un procedimiento Sub, el uso de los paréntesis es opcional, independientemente de si pasamos o no parámetros. No obstante, es muy recomendable especificar dichos paréntesis, ya que aportan una gran claridad a nuestro código, de forma que al leerlo podemos ver rápidamente los puntos en los que se realiza una llamada a una rutina de código. Debido a esto, el IDE sitúa automáticamente los paréntesis en el caso de que no los especifiquemos de forma explícita.

No es posible situar la llamada a un procedimiento Sub como parte de una expresión, puesto que este tipo de procedimientos, al no devolver un valor, provocaría un error de compilación. Ver Figura 176.

```
Sub Main()
    Dim Dato As String

    Dato = "hola " & Prueba()
                     Esta expresión no genera un valor.
End Sub

Sub Prueba()
    Console.WriteLine("Estamos en el procedimiento Prueba")
End Sub
```

Figura 176. No es posible hacer una llamada a un procedimiento Sub en una expresión.

Sintaxis de un procedimiento Function

El formato para la escritura de un procedimiento Function se muestra en el Código fuente 120.

```
[Ámbito] Function NombreFunción[(ListaParámetros)] As TipoDato
       [CódigoEjecutable]
       [Return Valor]
       [NombreFunción = Valor]
       [Exit Function]
       [CódigoEjecutable]
End Function
```

Código fuente 120

Los elementos que forman parte de este tipo de rutina son los siguientes:

- **Ámbito**. Define el modo en que vamos a poder acceder o llamar al procedimiento desde otro punto de la aplicación. El ámbito de los elementos del lenguaje será tratado en un apartado posterior.

- **Function...End Function**. Palabras clave que indican el comienzo y final de la función respectivamente. Cuando hagamos una llamada a la función, el compilador ejecutará el código comprendido entre estas dos palabras clave.

- **NombreFunción**. Identificador que utilizamos para reconocer y llamar a la función. En este tipo de procedimiento, también utilizamos su nombre para asignar el valor que será devuelto al código llamador en el modo *NombreFunción = Valor*, en esta última situación, podemos situar esta expresión de devolución en tantos lugares como necesitemos dentro de la función.

- **TipoDato**. Tipo de dato del valor devuelto como resultado de la ejecución de la función.

- **ListaParámetros**. Lista de variables separadas por comas, y encerradas entre paréntesis, que representan la información que recibe la función desde el código llamador.

- **Return**. Esta palabra clave permite salir de la ejecución de la función devolviendo al mismo tiempo un valor al código que hizo la llamada. Podemos utilizarla dentro de una función, en tantos lugares como necesitemos.

- **Exit Function**. Esta palabra clave permite salir de la ejecución de la función sin haber llegado a su fin. Podemos utilizarla dentro de una función, en tantos lugares como necesitemos.

El Código fuente 121 muestra un sencillo ejemplo de procedimiento Function, en el cual se pide al usuario que introduzca un número que es devuelto como resultado de la función.

```
Function Calcular() As Integer
    Dim MiValor As Integer
    Console.WriteLine("Introducir un número de 1 a 100")
    MiValor = Console.ReadLine()
    Return MiValor
    ' también podemos utilizar esta
    ' sintaxis para devolver el valor
    ' de retorno de la función:
    'Calcular = MiValor
End Function
```

Código fuente 121

En el caso de devolver el valor de retorno de una función utilizando el propio nombre de la función, nos encontramos con el problema de que si en un momento determinado tenemos que cambiar el nombre de la función, también deberemos cambiar todos aquellos puntos de la rutina en donde devolvemos el valor. Por este motivo es recomendable el uso de Return para el devolver el valor de la función, ya que si tenemos que cambiar el nombre de la función, no será necesario modificar los puntos en los que se devuelve el valor de este tipo de procedimiento.

Llamada a un procedimiento Function

Para realizar una llamada o ejecutar un procedimiento Function debemos escribir su nombre en un punto del programa; en este aspecto ambos tipos de procedimiento son iguales.

Por otro lado, los puntos que marcan las diferencias entre un Function y un Sub son los siguientes:

- Un procedimiento Function devuelve un valor, de modo que si queremos obtenerlo, debemos asignar la llamada a la función a una variable. Los procedimientos Sub no pueden devolver valores.

- Debido precisamente a la capacidad de un procedimiento Function de devolver un valor, podemos situar la llamada a una función dentro de una expresión, y operar con el valor de retorno dentro de la expresión, lo cual dota a nuestro código de una mayor flexibilidad. Los procedimientos Sub no pueden formar parte de expresiones.

En el Código fuente 122 vemos varios ejemplos de llamadas a la función Calcular( ), según el modo en que vamos a manipular su valor de retorno.

```
Module Module1

    Sub Main()
        Dim Resultado As Integer
        Dim NuevoValor As Integer

        ' llamada a una función sin recoger el valor de retorno,
        ' por este motivo, dicho valor se pierde
        Calcular()

        ' llamada a una función obteniendo el valor
        ' de retorno y asignando el valor a una variable
        Resultado = Calcular()
        Console.WriteLine("La variable Resultado contiene: {0}", Resultado)

        ' llamada a una función como parte de una expresión
        NuevoValor = 1500 + Calcular() * 2
        Console.WriteLine("La variable NuevoValor contiene: {0}", NuevoValor)

        Console.ReadLine()
    End Sub

    Function Calcular() As Integer
        Dim MiValor As Integer

        Console.WriteLine("Introducir un número de 1 a 100")
        MiValor = Console.ReadLine()

        Return MiValor

        ' también podemos utilizar esta
```

```
        ' sintaxis para devolver el valor
        ' de retorno de la función:
        'Calcular = MiValor
    End Function

End Module
```

Código fuente 122

Paso de parámetros a procedimientos

Un parámetro consiste en un valor que es pasado a un procedimiento. Dicho valor puede ser una variable, constante o expresión. Los procedimientos pueden recibir parámetros en forma de lista de variables separada por comas, siguiendo la sintaxis que vemos en el Código fuente 123.

```
[Optional] [ByVal | ByRef] [ParamArray] NombreParametro As TipoDato
```

Código fuente 123

Las reglas y cuestiones sobre paso de parámetros descritas en los siguientes apartados son válidas tanto para procedimientos como para funciones, excepto en aquellos lugares donde se indique lo contrario.

Protocolo de llamada o firma de un procedimiento

A la lista de parámetros de un procedimiento se le denomina *protocolo de llamada* o *firma* (signature) del procedimiento. Se trata de un concepto que cubriremos posteriormente en el apartado dedicado a la sobrecarga de procedimientos.

Tipo de dato de un parámetro

Al igual que hacemos cuando declaramos una variable, al declarar un parámetro debemos especificar el tipo de dato que el parámetro va a contener. Esto será o no obligatorio, en función del modificador establecido en la instrucción Option Strict.

En el Código fuente 124 hemos añadido un parámetro de tipo String al procedimiento Prueba( ). Cuando llamemos desde otro procedimiento a Prueba( ), al pasar desde el código llamador una cadena de caracteres, bien de forma literal o en una variable; el contenido de dicha cadena se traspasará a la variable situada en la lista de parámetros del procedimiento, que posteriormente visualizaremos en la consola.

```
Sub Main()
    Dim Nombre As String

    Nombre = "Juan"
    Prueba(Nombre)

    Prueba("Esto es una prueba")
    Console.ReadLine()
```

```
End Sub

Sub Prueba(ByVal ValorMostrar As String)
    Console.WriteLine("El valor del parámetro pasado es {0}", ValorMostrar)
End Sub
```

Código fuente 124

Paso de parámetros por valor y por referencia

Existen dos modos en el lenguaje de pasar parámetros a un procedimiento: *por valor* y *por referencia*. Este aspecto del lenguaje está relacionado con el modo en que el contenido de los parámetros son gestionados internamente en memoria. Ello nos permitirá según el tipo de paso empleado, poder alterar el valor del parámetro en el código que realizó la llamada.

Paso por valor (ByVal)

Cuando pasamos un parámetro por valor a un procedimiento, la variable que contiene el parámetro puede ser modificada dentro del procedimiento, sin que estos cambios afecten al valor original en el código llamador. Debemos situar en este caso en el procedimiento, la palabra clave ByVal antes del nombre del parámetro. Ver Código fuente 125.

```
Sub Main()
    Dim Nombre As String

    Nombre = "Juan"

    ' llamamos a un procedimiento
    ' y le pasamos una variable por valor
    Prueba(Nombre)

    ' la variable que hemos pasado al procedimiento,
    ' al volver aquí no ha sido cambiada, debido a que
    ' ha sido pasada por valor, sigue conteniendo
    ' la cadena "Juan"
    Console.WriteLine("Valor de la variable dentro de Main(): {0}", Nombre)
    Console.ReadLine()
End Sub

Sub Prueba(ByVal ValorMostrar As String)
    ' modificamos el valor del parámetro,
    ' este cambio no afecta a la variable Nombre
    ValorMostrar = "Elena"
    Console.WriteLine("Valor del parámetro dentro de Prueba(): {0}", ValorMostrar)
End Sub
```

Código fuente 125

Lo que ocurre en el fuente anterior a nivel de gestión interna en memoria de los parámetros es lo siguiente: cuando se realiza la llamada al procedimiento y se pasa el parámetro, el entorno detecta que se trata de un parámetro pasado por valor, por lo que crea una nueva variable en memoria que será la que manipulemos dentro del procedimiento. La Figura 177 muestra una representación gráfica de este proceso.

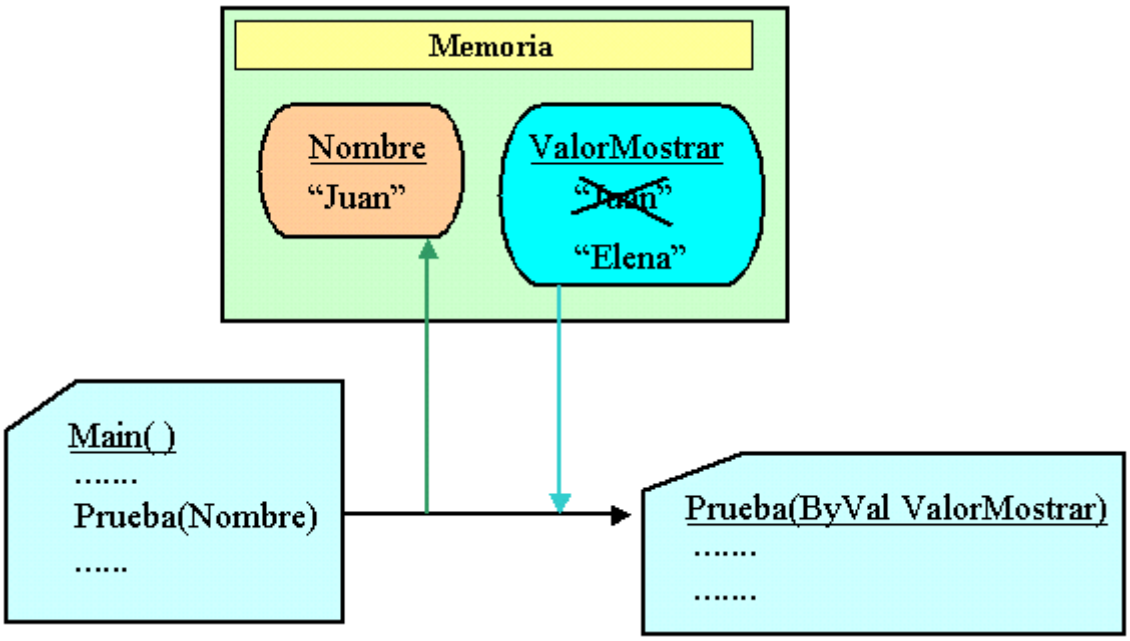


Figura 177. Esquema de gestión interna de variables en el paso por valor.

En el entorno de .NET Framework, por defecto, todos los parámetros son pasados por valor. Esto lo puede comprobar el lector de un modo muy simple: si al declarar un parámetro no especifica el tipo de paso, el IDE automáticamente situará junto a ese parámetro la palabra clave ByVal.

Se recomienda siempre que sea posible el paso de parámetros por valor, ya que ayuda a generar un código más optimizado y contribuye a mejorar la encapsulación.

Paso por referencia (ByRef)

Cuando pasamos un parámetro por referencia a un procedimiento, si modificamos dentro del procedimiento la variable que contiene el parámetro, dichos cambios en este caso sí afectarán al código llamador. Debemos situar en este caso en el procedimiento, la palabra clave ByRef antes del nombre del parámetro. Cambiemos el código del anterior ejemplo, haciendo que en este caso, el parámetro sea pasado por referencia y observemos los resultados. Ver Código fuente 126.

```
Sub Main()
    Dim Nombre As String

    Nombre = "Juan"
    Console.WriteLine("Valor de la variable antes de llamar a Prueba(): {0}",
Nombre)

    ' llamamos a un procedimiento
    ' y le pasamos una variable por referencia
    Prueba(Nombre)

    ' el cambio realizado al parámetro en el procedimiento
    ' ha afectado a la variable Nombre, que aquí contiene
    ' el mismo valor que se asignó en el procedimiento
    Console.WriteLine("Valor de la variable al volver a Main(): {0}", Nombre)
    Console.ReadLine()
End Sub
```

```
Sub Prueba(ByRef ValorMostrar As String)
    ' modificamos el valor del parámetro
    ValorMostrar = "Elena"
    Console.WriteLine("Valor del parámetro dentro de Prueba(): {0}", ValorMostrar)
End Sub
```

Código fuente 126

Lo que ocurre en el fuente anterior a nivel de gestión interna en memoria de los parámetros es lo siguiente: cuando se realiza la llamada al procedimiento y se pasa el parámetro, el entorno detecta que se trata de un parámetro pasado por referencia, y tanto la variable del código llamador como la del procedimiento llamado utilizan la misma dirección de memoria o referencia hacia los datos, por lo que los cambios realizados en un procedimiento afectarán también al otro. La Figura 178 muestra una representación gráfica de este proceso.

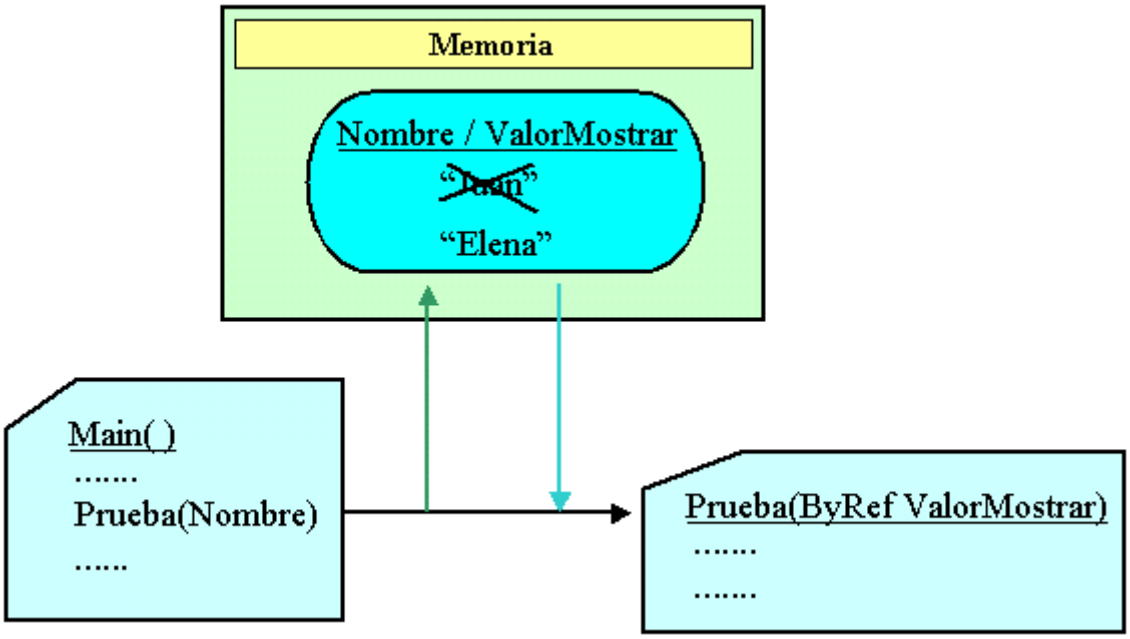


Figura 178. Esquema de gestión interna de variables en el paso por referencia.

Paso de parámetros por posición y por nombre

Cuando al llamar a un procedimiento con parámetros, pasamos estos en el mismo orden en el que están especificados en la declaración del procedimiento, se dice que se ha realizado un *paso de parámetros por posición*.

Existe además, otro modo de paso de parámetros en el cuál no estamos obligados a situarlos en el mismo orden que indica la declaración, es el llamado *paso de parámetros por nombre*. En este tipo de paso, debemos situar en la llamada al procedimiento el nombre del parámetro, seguido de los signos de dos puntos e igual (:=) y finalmente el valor a pasar.

El Código fuente 127 muestra unos ejemplos con ambos tipos de paso de parámetros.

```
Sub Main()
    Dim Localidad As String
    Dim Importe As Integer
    Dim DiaHoy As Date

    ' ---------------------
    Localidad = "Sevilla"
    Importe = 15044
    DiaHoy = #2/10/2002#

    ' paso de parámetros por posición
    Prueba(Localidad, Importe, DiaHoy)

    ' ---------------------
    Localidad = "Madrid"
    Importe = 250
    DiaHoy = #5/8/2002#

    ' paso de parámetros por nombre
    Prueba(Cantidad:=Importe, Fecha:=DiaHoy, Ciudad:=Localidad)

    Console.ReadLine()
End Sub

Sub Prueba(ByVal Ciudad As String, ByVal Cantidad As Integer, ByVal Fecha As Date)
    Console.WriteLine("Valores de los parámetros")
    Console.WriteLine("Ciudad: {0} - Cantidad: {1} - Fecha: {2}", Ciudad, Cantidad,
Fecha)
End Sub
```

Código fuente 127

Podemos mezclar ambos tipos de paso en la llamada a un procedimiento, teniendo en cuenta que los parámetros en los que no utilicemos paso por nombre, deberemos situarlos en su posición correcta. El Código fuente 128 muestra un ejemplo con una variación sobre el ejemplo anterior.

```
Prueba(Localidad, Fecha:=DiaHoy, Cantidad:=Importe)
```

Código fuente 128

Parámetros opcionales

Un parámetro opcional es aquel que no es necesario especificar al hacer la llamada a un procedimiento.

Para indicar en la declaración de un procedimiento que un parámetro es opcional, debemos utilizar la palabra clave Optional seguida de la especificación del parámetro, y finalizar con la asignación de un valor por defecto para el parámetro. Teniendo en cuenta además, que a partir del primer parámetro opcional en la lista de un procedimiento, todos los parámetros sucesivos también deben ser opcionales.

En el Código fuente 129 creamos una función en la que declaramos un parámetro opcional. Después hacemos dos llamadas a dicho procedimiento, pasando y omitiendo el parámetro opcional respectivamente en cada llamada.

```
Sub Main()
    Dim Localidad As String
    Dim Importe As Integer
    Dim Resultado As Integer

    ' ---------------------
    Localidad = "Sevilla"
    Importe = 15044

    ' paso de todos los parámetros al procedimiento
    Resultado = Calcular(Localidad, Importe)
    Console.WriteLine("Primera llamada, valor devuelto: {0}", Resultado)

    ' ---------------------
    Localidad = "Madrid"

    ' paso sólo del primer parámetro al procedimiento,
    ' esto hará que se utilice el valor por defecto
    ' del parámetro opcional
    Resultado = Calcular(Localidad)
    Console.WriteLine("Segunda llamada, valor devuelto: {0}", Resultado)

    Console.ReadLine()
End Sub

Function Calcular(ByVal Ciudad As String, Optional ByVal Cantidad As Integer =
5500) As Integer
    Console.WriteLine("Valores de los parámetros")
    Console.WriteLine("Ciudad: {0} - Cantidad: {1}", Ciudad, Cantidad)

    Return Cantidad + 100

End Function
```

Código fuente 129

Array de parámetros

Cuando en la lista de parámetros de un procedimiento utilizamos la palabra clave ParamArray junto al nombre del último parámetro de la lista, dicho parámetro será considerado un array, por lo que al hacer la llamada al procedimiento podremos pasarle un número variable de valores, que manejaremos a través del array. El Código fuente 130 muestra un ejemplo.

```
Sub Main()
    Dim Valor As Integer
    Dim Ciudad As String
    Dim Nombre As String

    Valor = 7954
    Ciudad = "Valencia"
    Nombre = "Jorge"

    ' en la llamada al procedimiento Prueba()
    ' todos los valores que pasemos a continuación
    ' del primer parámetro, serán depositados
    ' en el array de parámetros
    Prueba(Valor, Ciudad, "mesa", Nombre)

    Console.ReadLine()
End Sub
```

```
' el parámetro MasDatos del procedimiento es un array
' de parámetros variables
Sub Prueba(ByVal Importe As Integer, ByVal ParamArray MasDatos() As String)
    Dim Contador As Integer

    ' mostrar el primer parámetro
    Console.WriteLine("Parámetro Importe: {0}", Importe)
    Console.WriteLine()

    ' el resto de parámetros del procedimiento
    ' están en el array, los obtenemos recorriendolo
    ' con(una) estructura For...Next
    Console.WriteLine("Contenido del array de parámetros MasDatos():")
    For Contador = 0 To UBound(MasDatos)
        Console.WriteLine("Elemento: {0} - Valor: {1}", _
            Contador, MasDatos(Contador))
    Next
End Sub
```

Código fuente 130

Con ParamArray tenemos la ventana de que podemos pasar una cantidad variable de parámetros al procedimiento en cada llamada. La única restricción es que debemos utilizarlo como último parámetro de la lista del procedimiento.

Sobrecarga de procedimientos

Si bien el uso de parámetros opcionales es un medio para ahorrar al programador el paso de los mismos en situaciones en las que no son necesarios, resulta una solución un tanto artificiosa, ya que lo que realmente hace es complicar más que facilitar la escritura de código.

VB.NET aporta al lenguaje una nueva técnica que permite obviar el uso de parámetros opcionales por una solución más elegante y flexible: los procedimientos sobrecargados.

Antes de explicar en qué consiste un procedimiento sobrecargado, situémonos en el siguiente escenario:

Necesitamos mostrar los datos de un empleado de dos formas, en función del modo de consulta. Por un lado visualizaríamos su nombre, domicilio y localidad; y por otra parte su edad, DNI y fecha de alta en la empresa.

Con lo que sabemos hasta el momento, podríamos resolver este problema escribiendo un procedimiento con parámetros opcionales, y según pasáramos un valor u otro, mostrar la información correspondiente.

El Código fuente 131 muestra este modo de resolver el problema. El uso de la estructura If...End If será explicado posteriormente en el apartado dedicado a estructuras de control, por lo que aclararemos brevemente al lector que el uso de esta estructura nos permite ejecutar bloques de código en función de que la expresión utilizada a continuación de If se evalúe o no a Verdadero.

```
Sub Main()
    ' mostrar datos del empleado
    ' en función del nombre
    VerDatosEmpleado("Pedro")
```

```
    ' mostrar datos del empleado
    ' en función de la edad
    VerDatosEmpleado(, 28)

    Console.ReadLine()
End Sub

Sub VerDatosEmpleado(Optional ByVal Nombre As String = "X", Optional ByVal Edad As
Integer = 999)

    If Nombre <> "X" Then
        Console.WriteLine("Nombre del empleado: {0}", Nombre)
        Console.WriteLine("Domicilio: Colina Alta,12")
        Console.WriteLine("Localidad: Salamanca")
    End If

    If Edad <> 999 Then
        Console.WriteLine("Edad del empleado: {0}", Edad)
        Console.WriteLine("DNI:21555666")
        Console.WriteLine("Fecha de alta en la empresa: 10/4/1997")
    End If

    Console.WriteLine()
End Sub
```

Código fuente 131

El uso de parámetros opcionales, como acabamos de constatar, resulta engorroso, ya que nos obliga a comprobar qué valor ha sido pasado y mostrar los datos correspondientes en consecuencia. Tenemos además, un inconveniente añadido, y es que podemos pasar los dos parámetros a la vez, con lo que se mostrarían todos los datos, cuando lo que queremos es visualizar un grupo u otro en cada llamada.

Una aproximación diferente al problema sería escribir dos procedimientos distintos, y llamar a uno u otro según los datos que necesitemos. Ver Código fuente 132.

```
Sub Main()
    ' mostrar datos del empleado según nombre
    VerEmpleNombre("Pedro")


    ' mostrar datos del empleado según edad
    VerEmpleNum(28)


    Console.ReadLine()
End Sub


Public Sub VerEmpleNombre(ByVal Nombre As String)
    Console.WriteLine("Datos empleado por nombre")
    Console.WriteLine("Nombre del empleado: {0}", Nombre)
    Console.WriteLine("Domicilio: Colina Alta,12")
    Console.WriteLine("Localidad: Salamanca")
    Console.WriteLine()
End Sub


Public Sub VerEmpleNum(ByVal Edad As Integer)
    Console.WriteLine("Datos empleado por edad")
    Console.WriteLine("Edad del empleado: {0}", Edad)
```

```
    Console.WriteLine("DNI:21555666")
    Console.WriteLine("Fecha de alta en la empresa: 10/4/1997")
    Console.WriteLine()
End Sub
```

Código fuente 132.

Sin embargo, esta solución nos obliga a tener que saber varios nombres de procedimiento, con lo que tampoco ayuda mucho a simplificar el código.

¿No sería ideal, disponer de un único nombre de procedimiento y que este fuera lo suficientemente inteligente para mostrar los datos adecuados en cada caso?, pues esta característica está implementada en VB.NET a través de la *sobrecarga de procedimientos*.

La sobrecarga de procedimientos es una técnica que consiste en crear varias versiones de un mismo procedimiento, distinguiéndose entre sí por la lista de parámetros o protocolo de llamada del procedimiento.

Para definir un procedimiento como sobrecargado, debemos comenzar su declaración con la palabra clave Overloads. Podemos utilizar procedimientos tanto Sub como Function cuando realizamos sobrecarga., siendo posible que una de las implementaciones no tenga lista de parámetros. El Código fuente 133 muestra un ejemplo de sobrecarga.

```
Overloads Sub Datos()
       ' código del procedimiento
       ' ............
       ' ............
End Sub

Overloads Sub Datos(ListaParametrosA)
       ' código del procedimiento
       ' ............
       ' ............
End Sub

Overloads Function Datos(ListaParametrosB) As TipoDatoRetorno
       ' código del procedimiento
       ' ............
       ' ............
End Function
```

Código fuente 133

En el ejemplo anterior, cuando llamemos al procedimiento Datos( ), el entorno de .NET Framework, en función de si pasamos o no parámetros al procedimiento, y de cómo sean estos parámetros, ejecutará la versión adecuada del procedimiento.

Ya que el protocolo o firma del procedimiento es el elemento que emplea el CLR para diferenciar cada una de sus versiones o implementaciones, las listas de parámetros de cada versión deben ser diferentes al menos en uno de los siguientes aspectos:

- Número de parámetros.
- Orden de los parámetros.

- Tipo de dato de los parámetros.

Por consiguiente, no es posible crear dos procedimientos sobrecargados que sólo se diferencien en los nombres de los parámetros, por los modificadores de ámbito (Public, Private, etc.), o por el tipo de dato de retorno en el caso de un procedimiento Function.

Una vez vistas las normas y restricciones aplicables a los procedimientos sobrecargados, veamos en el Código fuente 134 como solucionaríamos el problema planteado al comienzo de este apartado empleando esta técnica.

```
Sub Main()
    Dim Dias As Integer
    ' mostrar datos del empleado según nombre
    VerEmpleado("Pedro")

    ' mostrar datos del empleado según edad
    Dias = VerEmpleado(28)
    Console.WriteLine("Días libres del empleado: {0}", Dias)
    Console.WriteLine()

    ' mostrar salario pasando las horas trabajadas
    VerEmpleado(25, 80)

    Console.ReadLine()
End Sub

Overloads Sub VerEmpleado(ByVal Nombre As String)
    Console.WriteLine("Datos empleado por nombre")
    Console.WriteLine("Nombre del empleado: {0}", Nombre)
    Console.WriteLine("Domicilio: Colina Alta,12")
    Console.WriteLine("Localidad: Salamanca")
    Console.WriteLine()
End Sub

Overloads Function VerEmpleado(ByVal Edad As Integer) As Integer
    Dim DiasLibres As Integer
    Console.WriteLine("Datos empleado por edad")
    Console.WriteLine("Edad del empleado: {0}", Edad)
    Console.WriteLine("DNI:21555666")
    Console.WriteLine("Fecha de alta en la empresa: 10/4/1997")
    Console.WriteLine()

    DiasLibres = 5
    Return DiasLibres
End Function

Overloads Sub VerEmpleado(ByVal PrecioHora As Integer, ByVal HorasTrabajadas As
Long)
    Dim Salario As Long

    Salario = PrecioHora * HorasTrabajadas
    Console.WriteLine("Salario según horas: {0}", Salario)
    Console.WriteLine()
End Sub
```

Código fuente 134

En este código hemos creado tres versiones sobrecargadas del procedimiento VerEmpleado( ). En una mostramos los datos del empleado según el nombre; en otra también mostramos otro conjunto de datos según la edad y además, al ser una función, devolvemos el número de días libres del empleado;

finalmente en una tercera implementación, calculamos el salario según el precio por hora y las horas trabajadas, que pasamos al protocolo de llamada. Desde Main( ) por lo tanto, siempre llamamos al procedimiento VerEmpleado( ).

Lista desplegable "Nombre de método", en el editor de código

La lista desplegable *Nombre de método*, situada en la parte superior derecha del editor de código, tiene dos finalidades principales que describimos a continuación.

- **Mostrar el nombre del procedimiento sobre el que actualmente trabajamos**. Esta información es útil sobre todo en procedimientos con muchas líneas de código, en las que no tenemos en todo momento visible la declaración del procedimiento.

- **Cambiar a otro procedimiento del módulo de código**. Abriendo la lista desplegable, y haciendo clic en alguno de los nombres de procedimientos que se muestran, no situaremos al comienzo de dicho procedimiento. Este es un medio más rápido para desplazarnos entre los procedimientos que tener que recorrer toda la ventana del editor de código.

En el ejemplo de la Figura 179, estamos situados en el procedimiento Main( ), y al abrir esta lista de procedimientos, podemos cambiar fácilmente a cualquier otro de los que hemos creado.

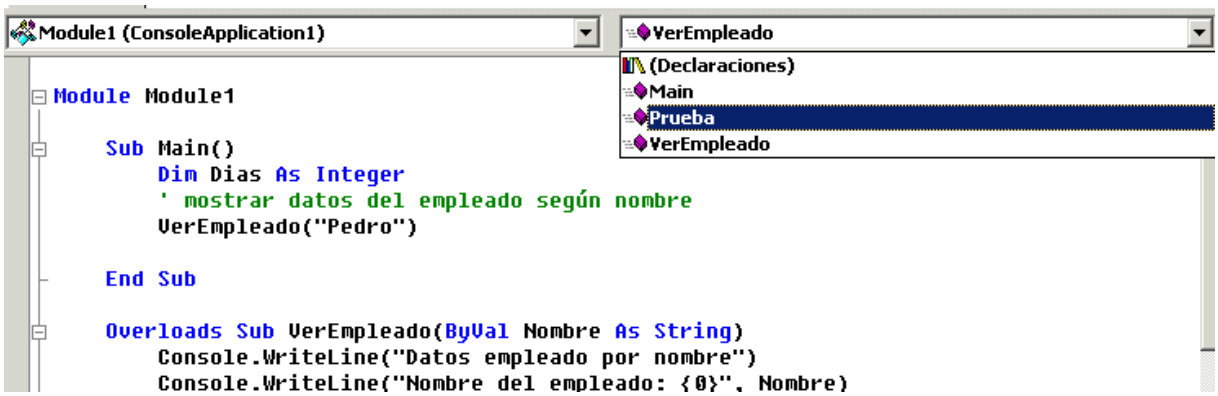


Figura 179. Lista Nombre de método, en el editor de código del IDE.

El motivo de usar el término método en lugar de procedimiento para esta lista, se debe a que como veremos en el tema sobre objetos, todo lo que haremos habitualmente en nuestra labor de programación, será crear clases, objetos, métodos, propiedades, etc. Por ello la terminología empleada en general se aproxima más a las técnicas de programación con objetos que a la programación estructurada.

15

Bifurcación y ámbito del código

Estructuras de control

Las estructuras de control contienen bloques de código que serán ejecutados en función del resultado obtenido al evaluar una expresión asociada a la estructura. A este proceso de redirección del flujo del programa hacia un determinado bloque de código se le denomina bifurcación

Según el modo de ejecución del código que contienen, las estructuras de control se dividen en los siguientes tipos: selección y repetición.

Selección

Las estructuras de selección o decisión permiten ejecutar un bloque de código entre varios disponibles, según el resultado de la evaluación de una expresión situada en la cabecera de la estructura.

If...End If

La sintaxis de esta estructura puede aplicarse de diferentes formas en función del tipo de decisión a resolver.

Decisión simple.

La sintaxis de decisión simple se muestra en el Código fuente 135.

```
If Expresión Then
    ' código
    ' ......
    ' ......
End If
```

Código fuente 135

Si al evaluar Expresión se devuelve como resultado Verdadero, se ejecutarán las líneas o bloque de código comprendido entre If y End If. Si Expresión es Falso, se desviará la ejecución a la primera línea de código que haya después de End If. Veamos un ejemplo en el Código fuente 136.

```
Sub Main()
    Dim Valor As Integer

    Console.WriteLine("Introducir un número")
    Valor = Console.ReadLine()

    If Valor = 5 Then
        Console.WriteLine("Estamos dentro de la estructura If," & _
            " ya que su expresión devuelve Verdadero")
    End If

    Console.ReadLine()
End Sub
```

Código fuente 136

Decisión simple en una línea.

En el caso de que sólo haya que ejecutar una instrucción sencilla cuando se cumple la expresión de la estructura, podemos omitir la palabra clave End If, escribiendo la sentencia a ejecutar en la misma línea de la declaración de la estructura If, justo a continuación de la palabra Then. La sintaxis en este caso, se simplifica, como muestra el Código fuente 137.

```
If Expresión Then Instrucción
```

Código fuente 137

Veamos un ejemplo en el Código fuente 138.

```
Sub Main()
    Dim Valor As Integer
    Dim Resultado As Integer

    Console.WriteLine("Introducir un número")
    Valor = Console.ReadLine()

    If Valor = 5 Then Resultado = Valor + 10

    Console.WriteLine("La variable resultado contiene {0}", Resultado)
    Console.ReadLine()
```

```
End Sub
```

Código fuente 138

Como habrá comprobado el lector, la sentencia que hay a continuación de Then sólo se ejecutará cuando la variable Valor contenga 5.

Decisión doble.

Además de ejecutar un bloque de código cuando la expresión valga Verdadero, podemos también ejecutar código cuando la expresión devuelva Falso. En este caso añadiremos a la estructura la palabra clave Else, como muestra la sintaxis del Código fuente 139.

```
If Expresión Then
       ' código cuando Expresión es Verdadero
       ' ......
       ' ......
Else
       ' código cuando Expresión es Falso
       ' ......
       ' ......
End If
```

Código fuente 139

Veamos un ejemplo en el Código fuente 140.

```
Sub Main()
    Dim Valor As Integer
    Dim Resultado As Integer

    Console.WriteLine("Introducir un número")
    Valor = Console.ReadLine()

    If Valor = 5 Then
        Resultado = Valor + 10
    Else
        Resultado = 777
    End If

    Console.WriteLine("La variable resultado contiene {0}", Resultado)
    Console.ReadLine()
End Sub
```

Código fuente 140

En este ejemplo, cuando Valor contenga 5 se ejecutará el bloque de código que hay a continuación de If, pero cuando Valor contenga un número distinto, se ejecutará el código que hay a continuación de Else. La ejecución en cualquier caso, continuará después a partir de la siguiente línea que haya a partir de la palabra clave End If.

Decisión doble en una línea.

Al igual que ocurre con la decisión simple, si para cada resultado de la expresión, sólo necesitamos ejecutar una instrucción, podemos escribir todo el código en una sola línea. Veamos la sintaxis en el Código fuente 141.

```
If Expresión Then InstrucciónVerdadero Else InstrucciónFalso
```

Código fuente 141

En el Código fuente 142 tenemos un ejemplo de uso.

```
Sub Main()
    Dim Valor As Integer
    Dim Resultado As Integer

    Console.WriteLine("Introducir un número")
    Valor = Console.ReadLine()

    If Valor = 5 Then Resultado = Valor + 10 Else Resultado = 777

    Console.WriteLine("La variable resultado contiene {0}", Resultado)
    Console.ReadLine()
End Sub
```

Código fuente 142

Si bien la ejecución de la estructura If en una línea puede ser útil en ocasiones, tiene como contrapartida el que nuestro código se vuelva más difícil de leer. Por ello es más recomendable el uso de esta estructura de control en su formato If...End If.

Decisión múltiple.

En el caso de que la expresión principal a evaluar devuelva Faso, podemos agregar expresiones adicionales utilizando la palabra clave ElseIf, con su bloque de código respectivo. En el caso de que ninguna de ellas se cumplan, podemos incluir un Else, para ejecutar un bloque de código por defecto. Veamos la sintaxis en el Código fuente 143.

```
If ExpresiónA Then
       ' código cuando ExpresiónA es Verdadero
       ' .......


ElseIf ExpresiónB Then
       ' código cuando ExpresiónB es Verdadero
       ' .......


[ElseIf ExpresiónN Then]
       ' código cuando ExpresiónN es Verdadero
       ' .......


[Else]
       ' código cuando ninguna epxresión devuelve Verdadero
       ' .......
```

```
End If
```

Código fuente 143

A continuación vemos un ejemplo en el Código fuente 144.

```
Sub Main()
    Dim Valor As Integer
    Dim Resultado As Integer

    Console.WriteLine("Introducir un número")
    Valor = Console.ReadLine()

    If Valor = 5 Then
        Resultado = Valor + 10

    ElseIf Valor > 100 Then
        Resultado = Valor + 200

    ElseIf Valor < 1 Then
        Resultado = -8

    Else
        Resultado = 777

    End If

    Console.WriteLine("La variable Resultado contiene {0}", Resultado)
    Console.ReadLine()
End Sub
```

Código fuente 144

En esta situación, si la primera expresión es Verdadero, se ejecutará el código situado a partir de If. Sin embargo, si If devuelve Falso, se comprobarán sucesivamente las expresiones de cada uno de los ElseIf existentes. En el caso de algún ElseIf devuelva Verdadero, se ejecutará el código que haya a partir del mismo. Si ninguna de las anteriores situaciones se cumple, se ejecutará el código que haya a partir de Else en el caso de que este se haya definido.

Select Case...End Select

Se trata de una evolución en la estructura If...End If de decisión múltiple, y su trabajo consiste en evaluar una expresión y comparar el resultado con la lista de expresiones de cada uno de los casos proporcionados. El Código fuente 145 muestra la sintaxis.

```
Select Case Expresión
    Case ListaExpresionesA
             ' código si se cumple ListaExpresionesA
             ' .....

    [Case ListaExpresionesB]
             ' código si se cumple ListaExpresionesB
             ' .....

    [Case Else]
```

```
            ' código si no se cumple ninguna ListaExpresiones
            ' .....
End Select
```

Código fuente 145

La lista de expresiones asociada a cada Case en esta estructura estará separada por comas y podrá tener alguno de los siguientes formatos:

- Expresión.
- ExpresiónMenor To ExpresiónMayor
- Is OperadorComparación Expresión

Tras evaluar la expresión de la estructura, si se encuentra una coincidencia con alguno de los Case, se ejecuta el bloque de código situado entre dicho Case y el siguiente. En caso de que no haya ninguna coincidencia, podemos opcionalmente, ejecutar un bloque por defecto, utilizando la palabra clave Case Else. Finalizada esta estructura, la ejecución continuará a partir de la línea situada después de End Select.

Veamos a continuación, en el Código fuente 146 un ejemplo de uso de esta estructura.

```
Sub Main()
    Dim Valor As Integer

    Console.WriteLine("Introducir un número")
    Valor = Console.ReadLine()

    Select Case Valor
        Case 5
            Console.WriteLine("El valor es 5")

        Case 120, 250
            Console.WriteLine("El valor es 120 ó 250")

        Case 3000 To 4000
            Console.WriteLine("El valor está en el rango de 3000 a 4000")

        Case Is < 10
            Console.WriteLine("El valor es menor de 10")

        Case Else
            Console.WriteLine("El valor es {0}, y no se cumple ningún caso", Valor)

    End Select

    Console.ReadLine()
End Sub
```

Código fuente 146

En el caso de que tras evaluar la expresión, haya más de un Case cuya lista de expresiones se cumpla, se ejecutará el que esté situado en primer lugar. En el ejemplo anterior, cuando la variable Valor contiene 5, se cumplen dos casos. Ver Código fuente 147.

```
        Case 5
            Console.WriteLine("El valor es 5")
      ' .....
      ' .....
        Case Is < 10
            Console.WriteLine("El valor es menor de 10")
```

Código fuente 147

Sin embargo sólo se ejecuta el código del primer Case.

Por otro lado, la lista de expresiones puede ser una combinación de los distintos formatos disponibles. Ver Código fuente 148.

```
Case 12 To 15, 4, 7, Is > 20
```

Código fuente 148

Repetición

Estas estructuras, también denominadas bucles, ejecutan un bloque de código de forma repetitiva mientras se cumpla una condición asociada a la estructura. A cada una de las veces en que se ejecuta el código contenido en estas estructuras se le denomina iteración.

While...End While

Se trata del tipo más sencillo, ejecuta las líneas de código que contiene, mientras que la expresión situada junto a While devuelva Verdadero. Veamos su sintaxis en el Código fuente 149.

```
While Expresión
       ' código
       ' .....
End While
```

Código fuente 149

Y a continuación, un ejemplo en el Código fuente 150.

```
Sub Main()
    Dim Valor As Integer
    Dim Contador As Integer

    Console.WriteLine("Introducir un número")
    Valor = Console.ReadLine()

    Console.WriteLine("Mostrar en consola todos los números desde 1 hasta el
introducido")
```

```
    While Contador < Valor
        Console.Write("-" & Contador)
        Contador += 1
    End While

    Console.ReadLine()
End Sub
```

Código fuente 150

Do...Loop

Esta estructura ejecuta un conjunto de líneas de código, en función del valor devuelto por una expresión, que a modo de condición, podemos situar al comienzo o final de la estructura.

Es posible además, no utilizar la expresión de evaluación al principio o final, debiendo en ese caso, introducir alguna condición en el interior del código de la estructura, para forzar la salida del bucle y evitar caer en un bucle infinito. La instrucción Exit Do nos permite forzar la salida del bucle, pudiendo emplearla tantas veces como sea necesario.

Veamos a continuación, las diferentes variantes disponibles.

Condición al principio.

La sintaxis se muestra en el Código fuente 151.

```
Do While | Until Expresión
       ' código
       ' ......
       [Exit Do]
       ' código
       ' ......
Loop
```

Código fuente 151

La diferencia entre usar While o Until reside en que empleando While, el código del bucle se ejecutará mientras la expresión devuelva Verdadero. En el caso de Until, el código se ejecutará mientras que la expresión devuelva Falso. Veamos los ejemplos del Código fuente 152.

```
Sub Main()
    Dim Valor As Integer
    Dim Palabra As String
    Dim Contador As Integer
    Dim Pruebas As Integer

    ' bucle con While
    Do While Valor <> 200
        Console.WriteLine("Introducir un número")
        Valor = Console.ReadLine()
    Loop

    ' bucle con Until
    Do Until Palabra = "coche"
        Console.WriteLine("Introducir una palabra")
```

```
        Palabra = Console.ReadLine()
    Loop

    ' inicializar contador,
    ' en este caso vamos a pedir también
    ' al usuario que introduzca un número,
    ' pero si después de cinco intentos,
    ' no consigue acertar, forzamos la salida
    ' de la estructura
    Contador = 1
    Do While Pruebas <> 200
        Console.WriteLine("Introducir un número - Intento nro.{0}", Contador)
        Pruebas = Console.ReadLine()

        If Contador = 5 Then
            Exit Do
        Else
            Contador += 1
        End If
    Loop
End Sub
```

Código fuente 152

En el último caso de este ejemplo, podemos observar como empleamos además, la anidación de diferentes estructuras, combinándolas para realizar las comprobaciones oportunas.

Condición al final.

La diferencia en este caso, consiste en que el contenido de la estructura se ejecuta al menos una vez. El Código fuente 153 muestra su sintaxis.

```
Do
       ' código
       ' ......
       [Exit Do]
       ' código
       ' ......
Loop While | Until Expresión
```

Código fuente 153

El Código fuente 154 muestra algunos ejemplos.

```
Sub Main()
    Dim Valor As Integer
    Dim Palabra As String

    ' bucle con While
    Do
        Console.WriteLine("Introducir un número")
        Valor = Console.ReadLine()
    Loop While Valor <> 200

    ' bucle con Until
    Do
        Console.WriteLine("Introducir una palabra")
```

```
        Palabra = Console.ReadLine()
    Loop Until Palabra = "coche"

End Sub
```

Código fuente 154

Sin condición.

Este es el modo más sencillo de la estructura: sin incluir condición al principio o final. También es el modo más peligroso, ya que si no incluimos un control dentro del código, corremos el riesgo de caer en un bucle infinito. En el ejemplo del Código fuente 155, establecemos una condición de salida mediante una estructura If dentro del bucle, que comprueba el contenido de la variable, y fuerza la salida cuando tenga un valor superior a cierto número.

```
Sub Main()
    Dim Valor As Integer

    Do
        Console.WriteLine("Introducir un número")
        Valor = Console.ReadLine()

       ' comprobar y salir del bucle si es necesario
        If Valor > 400 Then
            Exit Do
        End If
    Loop

End Sub
```

Código fuente 155

For...Next

Esta estructura ejecuta un bloque de código un número determinado de veces, establecido por un rango de valores y controlado por un contador. El Código fuente 156 muestra su sintaxis

```
For Contador = Inicio To Fin [Step Incremento]
    ' código
    ' ......
      [Exit For]
    ' código
    ' ......
Next
```

Código fuente 156

El elemento Contador se inicializa con un valor y el código existente entre For y Next es ejecutado una serie de veces, hasta que el valor de Contador se iguala a Fin.

Por defecto, los incrementos de Contador son en uno, pero podemos cambiar este aspecto utilizando el modificador Step, mediante el que podemos establecer el número en el que se van a realizar los incrementos. Step también nos permite realizar decremento utilizando un número negativo.

Si queremos realizar una salida de la ejecución de esta estructura antes de haber completado el número de iteraciones establecidas, podemos utilizar la instrucción Exit For, que provocará dicha salida de igual modo que el explicado anteriormente en la estructura Do...Loop.

El Código fuente 157 muestra diferentes ejemplos de uso de este tipo de bucle.

```
Sub Main()
    Dim Contador As Integer
    Dim Final As Integer

    ' recorrido simple del bucle
    Console.WriteLine("Bucle For normal")
    For Contador = 1 To 10
        Console.WriteLine("Variable Contador: {0}", Contador)
    Next

    Console.WriteLine()

    ' recorrer el bucle especificando un incremento
    Console.WriteLine("Bucle For con incremento")
    Console.WriteLine("Introducir el número de ejecuciones para el bucle")
    Final = Console.ReadLine()
    For Contador = 1 To Final Step 4
        Console.WriteLine("Variable Contador: {0}", Contador)
    Next

    Console.WriteLine()

    ' recorrer el bucle especificando un decremento
    Console.WriteLine("Bucle For con decremento")
    For Contador = 18 To 4 Step -1
        Console.WriteLine("Variable Contador: {0}", Contador)
    Next

    Console.WriteLine()

    ' este bucle no se ejecutará,
    ' al ser mayor el valor de contador
    ' que el valor final, y no haber
    ' establecido un decremento
    For Contador = 18 To 4
        Console.WriteLine("Variable Contador: {0}", Contador)
    Next

    ' recorrer el bucle y salir antes de completar
    ' todas las iteraciones
    Console.WriteLine("Bucle For con salida antes de completar")
    For Contador = 1 To 10
        Console.WriteLine("Variable Contador: {0}", Contador)

        If Contador = 7 Then
            Exit For
        End If
    Next

    Console.ReadLine()
End Sub
```

Código fuente 157

Un truco para optimizar y acelerar la ejecución en un bucle de este tipo, consiste en utilizar como contador una variable de tipo Integer, en vez de una de tipo Short, Long, Decimal, etc. Esto es debido a que los tipos Integer se actualizan más rápidamente que los otros tipos numéricos, aunque la diferencia sólo será apreciable en bucles que ejecuten muchos miles de iteraciones y que contengan muchas instrucciones. Ver Código fuente 158.

```
Dim ContRapido As Integer
Dim ContLento As Decimal

' este bucle se ejecutará más rápido que el siguiente
For ContRapido = 1 To 10000
    ' código
Next

For ContLento = 1 To 10000
    ' código
Next
```

Código fuente 158

For Each...Next

Se trata de una variante de la estructura For...Next, y su misión consiste en ejecutar un bloque de código por cada uno de los elementos existentes en un array o colección. El Código fuente 159 muestra su sintaxis.

```
For Each Elemento In ColecArray
    ' código
    ' ......
       [Exit For]
    ' código
    ' ......
Next
```

Código fuente 159

El Código fuente 160 muestra un ejemplo del uso de esta estructura de control.

```
Sub Main()
       ' crear un array y rellenarlo con valores
       Dim lsColores() As String = {"Azul", "Verde", "Marino", "Violeta"}
       Dim lsColor As String

       ' en cada iteración se obtiene un elemento
       ' del array lsColores, y se guarda en la variable lsColor
       For Each lsColor In lsColores
              Console.WriteLine(lsColor)
       Next
End Sub
```

Código fuente 160

Organización del proyecto en ficheros y módulos de código

Como ya apuntábamos de forma inicial en el tema *Escritura de código*, la plataforma .NET Framework nos permite una gran flexibilidad para ordenar el código de nuestro proyecto, que debemos organizar en contenedores físicos y lógicos de código.

Un contenedor físico no es otra cosa que un fichero con extensión .VB; estos ficheros son los que la plataforma reconoce como ficheros con código VB.NET. Podemos tener uno o varios dentro de un proyecto.

Un contenedor lógico, dicho del modo más simple, es aquel elemento en el entorno de .NET que nos permite escribir en su interior declaraciones y procedimientos, que serán accesibles desde otros elementos dentro del proyecto o ensamblado actual, o bien desde otros ensamblados, en función de su ámbito o accesibilidad.

El CLR dispone de varios tipos de contenedores lógicos, entre los que se encuentran los módulos, clases, interfaces, estructuras, etc. Los espacios de nombres (namespaces) son un tipo de contenedor lógico especial, cuya misión consiste en albergar al resto de contenedores lógicos; una especie de metacontenedor. La estructura básica de un contenedor lógico se muestra en la Figura 180.

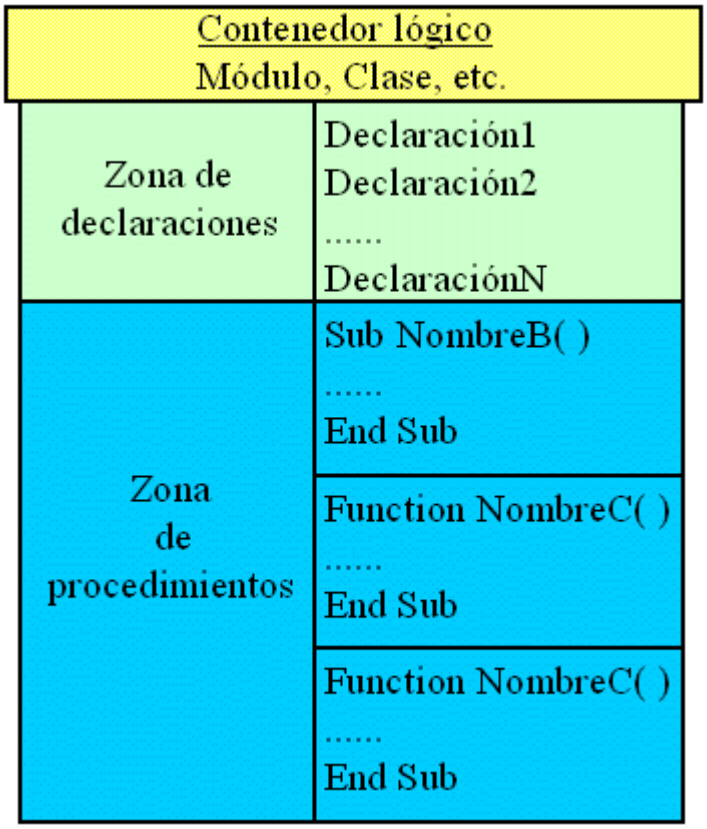


Figura 180. Estructura de un contenedor lógico de código.

La configuración por defecto en este sentido para VS.NET, establece que cada vez que añadimos un nuevo módulo o clase a un proyecto, se crea un nuevo fichero con extensión .VB, que contiene el mencionado módulo o clase. El nombre utilizado es el mismo para el fichero y el módulo o clase. Sin embargo, podemos incluir varios contenedores lógicos, de igual o distinto tipo, dentro del mismo fichero de código. Ver Figura 181.

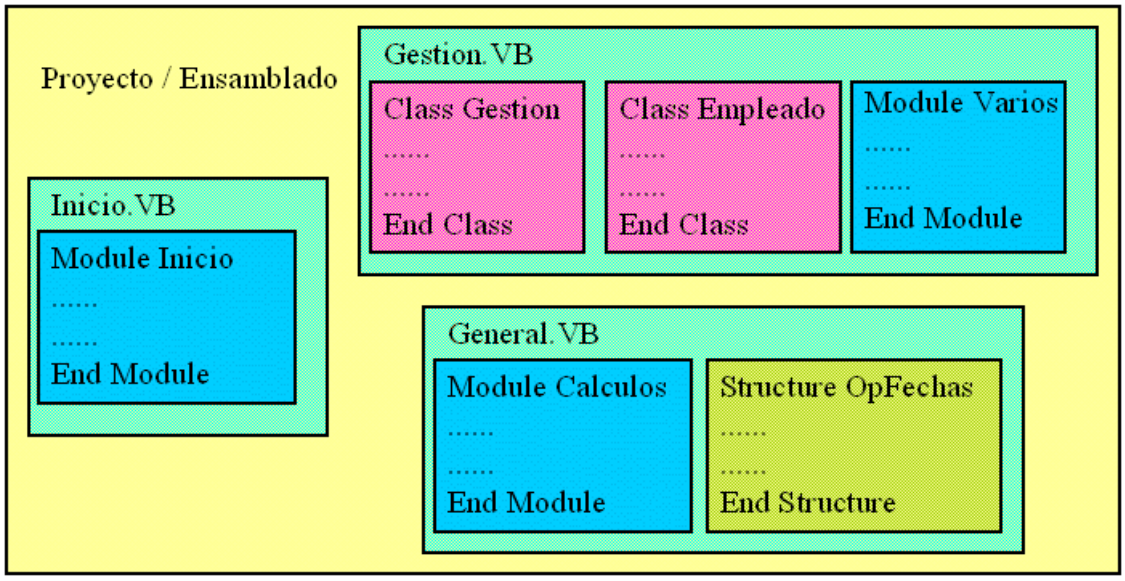


Figura 181. Esquema de organización del código en un proyecto VB.NET.

En este apartado describiremos las operaciones básicas a realizar, para añadir nuevos módulos de código a un proyecto, quitarlos, y agregar otros ya existentes. Lo comentado aquí será igualmente válido a la hora de manejar clases en el tema dedicado a OOP.

Agregar un nuevo módulo (y fichero) de código

Deberemos seleccionar en el IDE de VS.NET, la opción de menú *Proyecto + Agregar módulo*. Esto abrirá la caja de diálogo *Agregar nuevo elemento*, en la que automáticamente aparecerá un nombre para el módulo, asignado por el IDE. Podemos emplear dicho nombre o escribir otro distinto. Ver Figura 182.

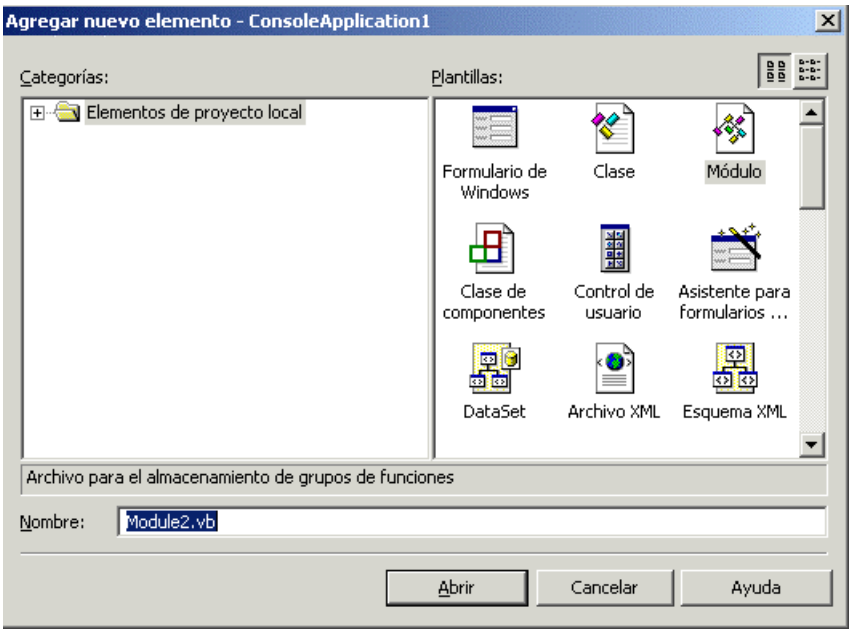


Figura 182. Agregar un nuevo módulo al proyecto.

Al pulsar Abrir, se creará un nuevo fichero con el nombre indicado en la caja de diálogo y la extensión .VB, que contendrá un módulo también del mismo nombre, dentro del cual podemos empezar a escribir código. Ver Figura 183.

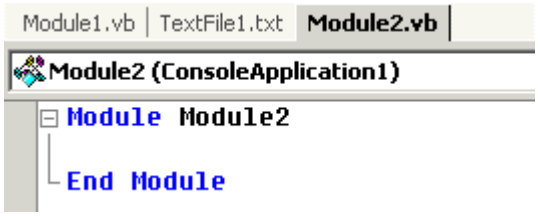


Figura 183. Módulo recién creado.

Crear un nuevo módulo dentro de un fichero existente

Esto es aún más fácil, ya que en esta situación solamente debemos escribir la declaración del módulo dentro del fichero de código, utilizando las palabras clave Module...End Module. Ver su sintaxis en el Código fuente 161.

```
Module NombreModulo
       ' código
       ' ......
End Module
```

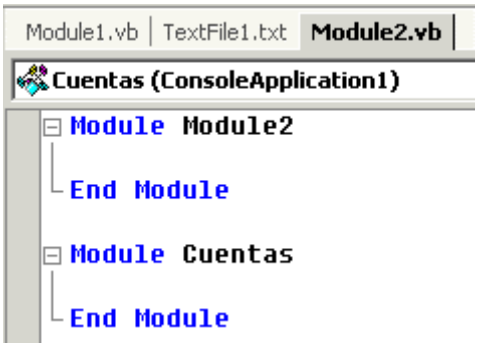
Código fuente 161

Debemos tener en cuenta que no es posible anidar módulos, es decir, no podemos declarar un módulo dentro de la declaración de un módulo ya existente. Ver Código fuente 162.

```
Module NombreModulo
       ' esto no es válido y producirá un error

       Module NombreNuevo

       End Module

End Module
```

Código fuente 162

Veamos a continuación un ejemplo. En el apartado anterior, hemos creado un nuevo módulo con el nombre Module2, creándose al mismo tiempo, un nuevo fichero de código con el nombre Module2.VB. Pues bien, para añadir otro módulo más dentro de este fichero de código, al que daremos el nombre Cuentas, tan sólo hemos de poner la declaración del nuevo módulo antes o después del existente. Ver Código fuente 163.



Código fuente 163

Con este ejemplo intentamos demostrar que los módulos de código son totalmente independientes del fichero físico que los alberga; por tal razón, varios módulos pueden escribirse dentro del mismo fichero.

Cambiar el nombre de un fichero de código

Si no queremos que el nombre de un fichero de código sea igual que alguno de los módulos que contiene, debemos abrir la ventana Explorador de soluciones, hacer clic derecho sobre el nombre del fichero de código, y elegir la opción *Cambiar nombre*. Esto nos permitirá dar un nuevo nombre al fichero .VB que contiene el código. Ver Figura 184.

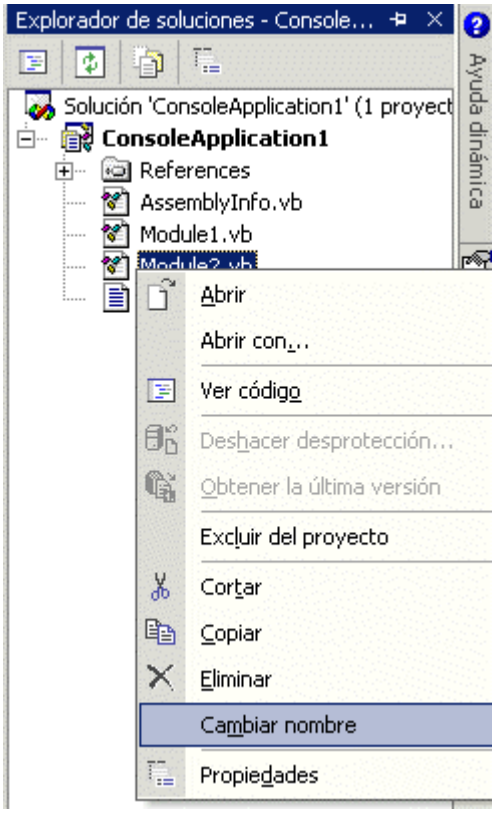


Figura 184. Cambiar el nombre de un fichero de código.

Añadir al proyecto un fichero de código existente

Si tenemos código de otro proyecto, que nos gustaría utilizar en el proyecto actual, o bien hemos escrito código utilizando un editor distinto del proporcionado por VS.NET, podemos añadirlo a nuestro proyecto utilizando la opción de menú *Proyecto + Agregar elemento existente*. Hemos de observar que es necesario que el código escrito esté en un fichero con extensión .VB.

Por ejemplo, he utilizado el Bloc de notas para escribir un módulo que contiene un procedimiento, y lo he guardado en un fichero con el nombre MiCodigo.VB. Al utilizar la opción antes mencionada, se abrirá una caja de diálogo, con la que navegaré por las carpetas del equipo para localizar el fichero; al pulsar el botón Abrir, se añadirá dicho fichero al proyecto. Ver Figura 185.

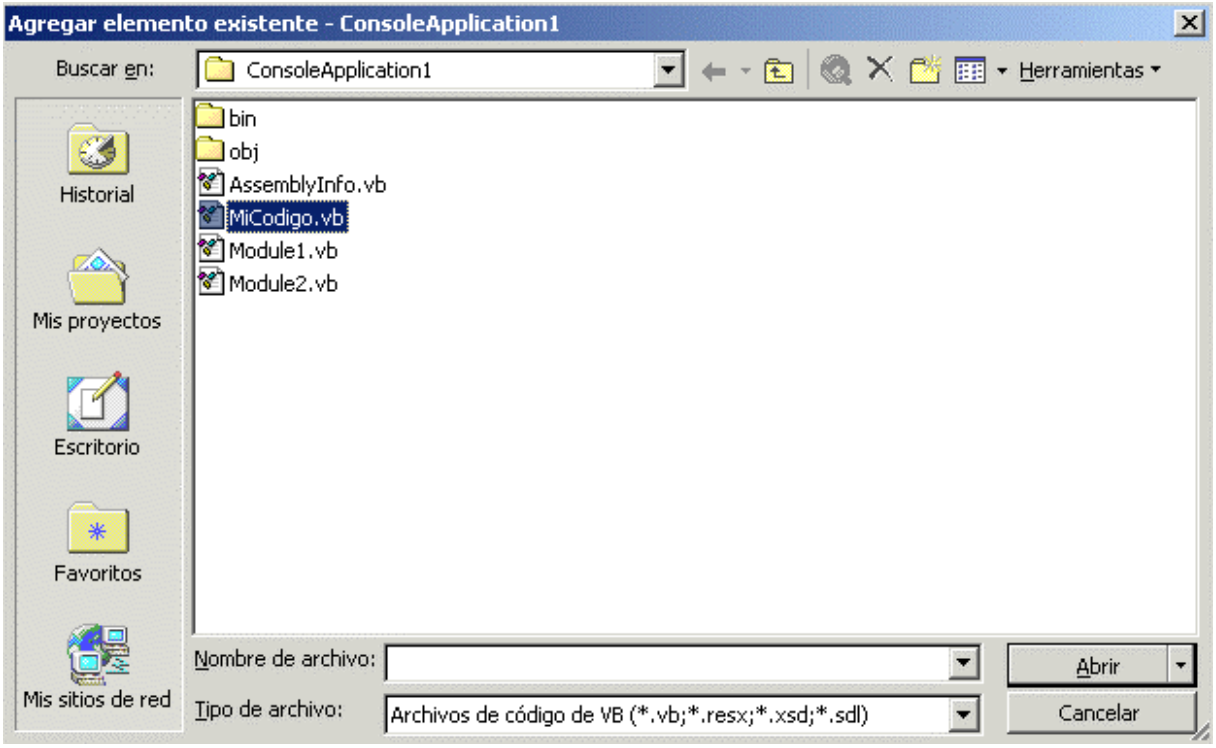


Figura 185. Agregar un fichero de código existente al proyecto.

Lista desplegable "Nombre de clase", en el editor de código

La lista desplegable *Nombre de clase*, situada en la parte superior izquierda del editor de código, tiene dos finalidades principales que describimos a continuación.

- **Mostrar el nombre del módulo sobre el que actualmente trabajamos**. Esta información es útil cuando estamos escribiendo código en un fichero que tiene varios módulos, de forma que siempre podemos saber sobre que módulo estamos posicionados.

- **Cambiar a otro módulo dentro del mismos fichero de código**. Esta operación la realizamos en dos pasos. En primer lugar abriremos esta lista desplegable y seleccionaremos el nombre del módulo al que vamos a cambiar. Por último, abriremos la lista Nombre de método, y elegiremos uno de los procedimientos del módulo sobre el que acabamos de posicionarnos, el editor de código cambiará entonces a dicho procedimiento.

La Figura 186 muestra una imagen de esta lista de módulos.

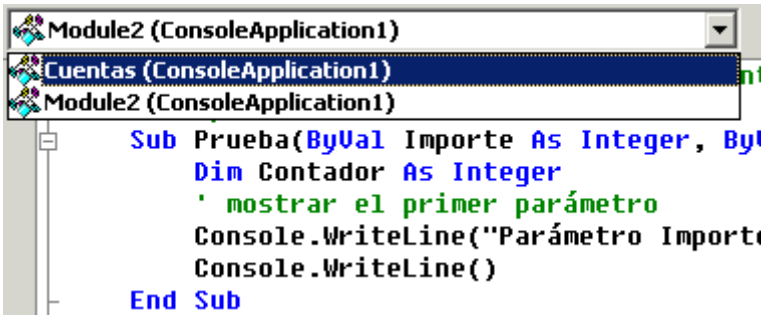


Figura 186. Lista Nombre de clase, en el editor de código del IDE.

El motivo de usar el término clase en lugar de módulo para esta lista, se debe, como ya explicamos anteriormente en el apartado sobre la lista Nombre de método, a que como veremos en el tema sobre objetos, todo lo que haremos habitualmente en nuestra labor de programación, será crear clases, objetos, métodos, propiedades, etc. Por ello la terminología empleada en general se aproxima más a las técnicas de programación con objetos que a la programación estructurada.

Excluir y eliminar ficheros de código del proyecto

Si no queremos que un determinado fichero (con el módulo o módulos que incluye) siga formando parte del proyecto, podemos separarlo del mismo abriendo la ventana Explorador de soluciones, y haciendo clic derecho sobre el nombre del procedimiento, elegir la opción de menú *Excluir del proyecto*. Esta acción quita el fichero del proyecto pero no lo elimina físicamente. Ver Figura 187.

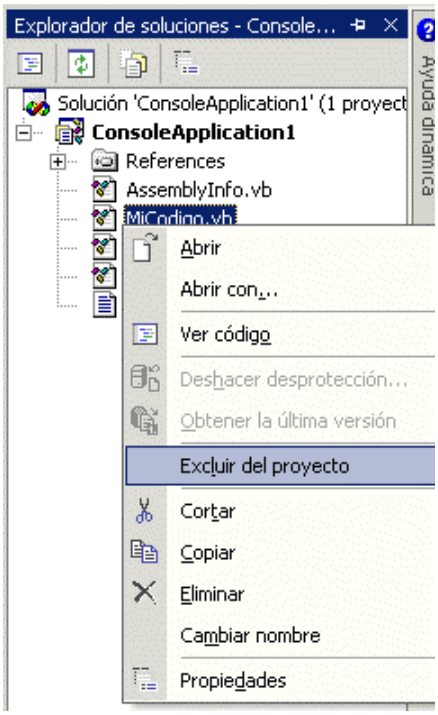


Figura 187. Excluir un fichero del proyecto.

Para eliminar físicamente el fichero de código, debemos realizar la misma operación descrita antes, pero seleccionando en este caso en el menú contextual, la opción Eliminar.

Reglas de ámbito

El ámbito o accesibilidad de un elemento declarado en el código, consiste en la capacidad que tenemos para utilizar dicho elemento desde otro punto cualquiera del código, ya sea desde el interior del propio ensamblado o desde un ensamblado externo.

Las reglas de ámbito se aplican por un lado a procedimientos y variables, visto el lenguaje desde el prisma de la programación estructurada; y por otro lado, estas normas también se aplican a métodos y propiedades, visto el lenguaje desde el punto de vista de la programación orientada a objetos.

En este tema realizaremos una revisión de las cuestiones de ámbito aplicadas a procedimientos y variables, dejando el ámbito de métodos y propiedades para el tema sobre OOP.

Ámbito de procedimientos

El ámbito de procedimientos consiste en la capacidad de poder llamar a un procedimiento desde un punto dado del código, en función del nivel de acceso definido para dicho procedimiento en el momento de su declaración.

Para especificar el ámbito de un procedimiento, lo haremos mediante una palabra clave o modificador de ámbito, anteponiéndolo al tipo de procedimiento (Sub o Function) dentro de la declaración. El Código fuente 164 muestra la sintaxis a utilizar.

```
ModificadorÁmbito Sub | Function NombreProcedimiento([ListaParámetros])
```

Código fuente 164

Veamos a continuación, los ámbitos de procedimiento disponibles.

Público

Un procedimiento con ámbito público puede ser llamado desde cualquier punto del módulo en el que se ha declarado, o desde cualquier otro módulo del proyecto. La palabra clave utilizada como modificador de ámbito en este caso es Public.

En el Código fuente 165 tenemos dos módulos: General y Calculos, que contienen respectivamente los procedimientos Main( ) y Totales( ). Desde Main( ) podemos perfectamente llamar al procedimiento Totales( ), ya que al haber sido declarado como Public, es accesible desde otro módulo.

```
Module General

    Public Sub Main()
        Console.WriteLine("Estamos en el módulo General, procedimiento Main")

       ' llamar al procedimiento Totales() que está en otro módulo
```

```
        Totales(400)
        Console.ReadLine()
    End Sub

End Module

Module Calculos

    Public Sub Totales(ByVal Importe As Integer)
        Dim Resultado As Integer
        Console.WriteLine("Estamos en el módulo Calculos, procedimiento Totales")
        Resultado = Importe + 1000
        Console.WriteLine("El total obtenido es: {0}", Resultado)
    End Sub

End Module
```

Código fuente 165

Public es el modificador de ámbito por defecto para procedimientos, lo que quiere decir que si no lo utilizamos, al crear un procedimiento, su ámbito será público por defecto.

Es posible escribir varios procedimientos con el mismo nombre y ámbito público en distintos módulos. Cuando esto ocurra, al llamar al procedimiento se ejecutará por defecto el que más próximo esté desde el módulo que ha sido llamado. En el caso de que necesitemos ejecutar el procedimiento que se encuentre en otro módulo deberemos hacer la llamada escribiendo el nombre del módulo, un punto, y el nombre del procedimiento. Veamos un ejemplo en el Código fuente 166.

```
Module General

    Public Sub Main()
        ' en esta llamada, se ejecutará el procedimiento Totales()
        ' que está en este mismo módulo
        Totales(400)

        ' en esta llamada, como especificamos el nombre del módulo
        ' en primer lugar, se ejecuta el procedimiento Totales()
        ' que está en el módulo Calculos
        Calculos.Totales(280)

        Console.ReadLine()
    End Sub

    Public Sub Totales(ByVal Importe As Integer)
        ' en esta versión del procedimiento,
        ' multiplicamos el parámetro por un valor
        Dim Resultado As Integer
        Console.WriteLine("Estamos en el módulo General, procedimiento Totales")
        Resultado = Importe * 4
        Console.WriteLine("El total obtenido es: {0}", Resultado)
    End Sub

End Module

Module Calculos

    Public Sub Totales(ByVal Importe As Integer)
        ' en esta versión del procedimiento,
        ' sumamos un valor al parámetro
        Dim Resultado As Integer
        Console.WriteLine("Estamos en el módulo Calculos, procedimiento Totales")
```

```
        Resultado = Importe + 1000
        Console.WriteLine("El total obtenido es: {0}", Resultado)
    End Sub

End Module
```

Código fuente 166

Privado

Un procedimiento con ámbito privado sólo puede ser llamado desde el propio módulo en el que se ha declarado. La palabra clave utilizada como modificador de ámbito en este caso es Private. Veamos un ejemplo en el Código fuente 167.

```
Module General

    Public Sub Main()
        ' podemos ejecutar el procedimiento Totales()
        ' ya que tiene ámbito público
        Totales(400)

        Dim MiNumero As Integer

        ' error, la función ObtenerNumero tiene ámbito privado,
        ' dentro del módulo Calculos,
        ' por lo que no es accesible desde este módulo
        MiNumero = ObtenerNumero()

        Console.ReadLine()
    End Sub


End Module

Module Calculos

    Public Sub Totales(ByVal Importe As Integer)
        Dim Resultado As Integer
        Console.WriteLine("Estamos en el módulo Calculos, procedimiento Totales")

        ' podemos llamar desde aquí a la función ObtenerNumero
        ' ya que estamos en el mismo módulo
        Resultado = Importe + ObtenerNumero()
        Console.WriteLine("El total obtenido es: {0}", Resultado)
    End Sub

    Private Function ObtenerNumero() As Integer
        Console.WriteLine("Estamos en el módulo Calculos," & _
            " procedimiento ObtenerNumero")
        Return 18
    End Function

End Module
```

Código fuente 167

En el anterior fuente, desde Main( ) no podemos llamar a la función ObtenerNumero( ), ya que dicha función tiene ámbito Private y reside en un módulo distinto.

Sin embargo, sí podemos llamarla desde el procedimiento Totales( ), ya que en ese caso, la llamada se realiza dentro del mismo módulo de código.

Ámbito de variables

El ámbito de variables consiste en la capacidad de acceso que tenemos hacia una variable, de forma que podamos obtener su valor, así como asignarlo. Para determinar su nivel de accesibilidad, aquí intervienen, además de los modificadores de ámbito, el lugar o nivel de emplazamiento de la variable dentro del código.

Respecto a los modificadores de ámbito, disponemos de las mismas palabras clave que para los procedimientos: Public y Private, y su sintaxis de uso la vemos en el Código fuente 168.

```
ModificadorÁmbito [Dim] NombreVariable As TipoDato
```

Código fuente 168

En función del punto de código en el que sea declarada una variable, podremos omitir el uso de la palabra clave Dim para realizar dicha declaración.

A continuación se describen los diferentes tipos de ámbito para variables, en función de su lugar de declaración.

Ámbito a nivel de procedimiento

Una variable declarada dentro del cuerpo de un procedimiento se dice que tiene un ámbito local o a nivel de procedimiento, no pudiendo ser accedida por otro código que no sea el de dicho procedimiento. Ver Código fuente 169.

```
Public Sub Main()
    ' declaramos una variable que tiene ámbito
    ' a nivel de este procedimiento
    Dim Nombre As String

    Nombre = "Hola"

    Console.WriteLine("Introducir un valor")
    Nombre &= " " & Console.ReadLine()
End Sub

Public Sub Manipular()
    ' si intentamos desde este procedimiento
    ' acceder a la variable Nombre del
    ' procedimiento Main(), se produce un error
    Nombre = "nuevo valor"
End Sub
```

Código fuente 169

En el ejemplo anterior, la variable Nombre puede ser manipulada dentro del procedimiento Main( ), y cualquier intento de acceder a ella desde otro procedimiento provocará un error.

Ámbito a nivel de bloque

Una variable declarada dentro de una estructura de control se dice que tiene ámbito local a nivel de bloque, siendo accesible sólo dentro del código que está contenido en la estructura. Ver Código fuente 170.

```
Public Sub Main()
    ' variables con ámbito a nivel de procedimiento
    Dim MiNumero As Integer
    Dim Total As Integer

    Console.WriteLine("Introducir un número")
    MiNumero = Console.ReadLine()

    If MiNumero > 0 Then
        ' variable con un ámbito a nivel de bloque
        ' sólo es accesible dentro de esta estructura If
        Dim Calculo As Integer
        Console.WriteLine("Introducir otro número para sumar")
        Calculo = Console.ReadLine()

        MiNumero += Calculo
    End If

    Console.WriteLine("El resultado total es: {0}", MiNumero)

    ' error, la variable Calculo no es accesible desde aquí
    Total = 150 + Calculo

    Console.ReadLine()
End Sub
```

Código fuente 170

En este punto debemos aclarar que el ámbito dentro de un bloque se entiende como la parte de la estructura en la que ha sido declarada la variable. Por ejemplo, en una estructura If...End If con Else, si declaramos una variable a continuación de If, dicha variable no será accesible desde el bloque de código que hay a partir de Else. Ver Código fuente 171.

```
If MiNumero > 0 Then
    ' variable con un ámbito a nivel de bloque
    ' sólo es accesible dentro de esta estructura If
    Dim Calculo As Integer
    ' ......

Else
    ' la variable Calculo no es accesible desde aquí
    ' ......
End If
```

Código fuente 171

Ámbito a nivel de módulo

Una variable declarada en la zona de declaraciones de un módulo, es decir, fuera de cualquier procedimiento, pero dentro de las palabras clave Module...End Module, y utilizando como palabra clave Dim o Private, se dice que tiene ámbito a nivel de módulo.

Aunque tanto Dim como Private son perfectamente válidas para declarar variables a nivel de módulo, se recomienda usar exclusivamente Private; de este modo facilitamos la lectura del código, reservando las declaraciones con Dim para las variables con ámbito de procedimiento, y las declaraciones con Private para el ámbito de módulo.

En el ejemplo del Código fuente 172 declaramos la variable Nombre dentro del módulo, pero fuera de cualquiera de sus procedimientos, esto hace que sea accesible desde cualquiera de dichos procedimientos, pero no desde un procedimiento que se halle en otro módulo.

```
Module General

    'Dim Nombre As String  <--- esta declaración es perfectamente válida...

    Private Nombre As String  ' ...pero se recomienda declarar con Private

    Public Sub Main()

        Console.WriteLine("Procedimiento Main()")
        Console.WriteLine("Asignar valor a la variable")
        Nombre = Console.ReadLine()

        Console.WriteLine("El valor de la variable en Main() es: {0}", Nombre)
        Manipular()
        MostrarValor()

        Console.ReadLine()
    End Sub

    Public Sub Manipular()
        Console.WriteLine("Procedimiento Manipular()")
        Console.WriteLine("Asignar valor a la variable")
        Nombre = Console.ReadLine()

        Console.WriteLine("El valor de la variable en Manipular() es: {0}", Nombre)
    End Sub

End Module

Module Calculos
    Public Sub MostrarValor()
        ' error, no se puede acceder desde este módulo
        ' a la variable Nombre, que está declarada Private
        ' en el módulo General
        Console.WriteLine("Procedimiento MostrarValor()")
        Nombre = "Antonio"
        Console.WriteLine("Valor de la variable Nombre: {0}", Nombre)
    End Sub
End Module
```

Código fuente 172

Para comprobar el valor de estas variables a través del depurador, tenemos que utilizar la ventana Automático, que podemos abrir con el menú *Depurar + Ventanas + Automático*, o las teclas [CTRL + ALT + V, A]. Ver Figura 188.

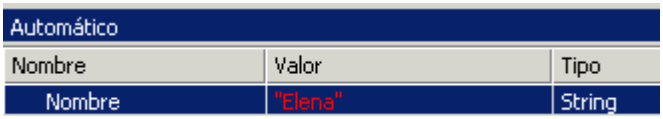


Figura 188. Ventana Automático, del Depurador.

Ámbito a nivel de proyecto

Una variable declarada en la zona de declaraciones de un módulo utilizando la palabra clave Public, se dice que tiene ámbito a nivel del proyecto, es decir, que es accesible por cualquier procedimiento de cualquier módulo que se encuentre dentro del proyecto.

Si tomamos el fuente anterior y declaramos como Public la variable Nombre, ahora sí podremos manipularla desde cualquier punto de la aplicación. Ver Código fuente 173.

```
Module General

    ' esta variable será accesible
    ' desde cualquier lugar del proyecto
    Public Nombre As String

    Public Sub Main()
        Console.WriteLine("Procedimiento Main()")
        Console.WriteLine("Asignar valor a la variable")
        Nombre = Console.ReadLine()

        Console.WriteLine("El valor de la variable en Main() es: {0}", Nombre)
        Manipular()
        MostrarValor()

        Console.ReadLine()
    End Sub

    Public Sub Manipular()
        Console.WriteLine("Procedimiento Manipular()")
        Console.WriteLine("Asignar valor a la variable")
        Nombre = Console.ReadLine()

        Console.WriteLine("El valor de la variable en Manipular() es: {0}", Nombre)
    End Sub

End Module

Module Calculos

    Public Sub MostrarValor()
        ' al haber declarado la variable Nombre
        ' como Public en el módulo General, podemos acceder a ella
        ' desde un módulo distinto al que se ha declarado
        Console.WriteLine("Procedimiento MostrarValor()")
        Nombre = "Antonio"
        Console.WriteLine("Valor de la variable Nombre: {0}", Nombre)
    End Sub
```

```
End Module
```

Código fuente 173

Periodo de vida o duración de las variables

El periodo de vida de una variable es el tiempo durante el cual la variable está activa, ocupando el espacio de memoria que le ha asignado el CLR, y disponible para ser utilizada. El periodo de vida de una variable depende de su ámbito, de forma que podemos clasificarlos como se muestra a continuación.

- **Ámbito de bloque**. El periodo de vida de estas variables se desarrolla desde el momento en que son declaradas dentro del bloque y hasta que dicho bloque finaliza.
- **Ámbito de procedimiento**. Para estas variables, su periodo de vida está comprendido entre el momento en que son declaradas y hasta que la ejecución del procedimiento termina.
- **Ámbito a nivel de módulo y proyecto**. En este caso, el periodo de vida de la variable va desde el comienzo de la ejecución de la aplicación y hasta que esta termina.

Variables Static

Este tipo de variables se caracterizan por el hecho de que retienen su valor al finalizar el procedimiento en el que han sido declaradas. Se deben declarar utilizando la palabra clave Static, pudiendo opcionalmente omitir la palabra clave Dim. El Código fuente 174 muestra su sintaxis.

```
Static [Dim] Importe As Integer
```

Código fuente 174

Cuando declaramos una variable normal dentro de un procedimiento, cada vez que llamamos al procedimiento, dicha variable es inicializada. El ejemplo del Código fuente 175, en cada llamada al procedimiento, se inicializa la variable y le sumamos un número, por lo que la variable siempre muestra el mismo valor por la consola.

```
Public Sub Main()
    Verificar("Primera") ' en esta llamada se muestra 7
    Verificar("Segunda") ' en esta llamada se muestra 7
    Verificar("Tercera") ' en esta llamada se muestra 7
    Console.ReadLine()
End Sub

Public Sub Verificar(ByVal OrdenLlamada As String)
    ' cada vez que se ejecuta este procedimiento
    ' la variable Importe se inicializa a 5
    Dim Importe As Integer = 5

    Importe += 2
    Console.WriteLine("{0} llamada al procedimiento, la variable contiene {1}", _
        OrdenLlamada, Importe)
```

```
End Sub
```

Código fuente 175

Pero cambiemos el modo de declaración de la variable Importe, añadiéndole Static. En este caso, la primera vez que se ejecuta el procedimiento, se inicializa la variable con el valor 5, pero al terminar la ejecución, la variable no se destruye, sino que en la siguiente ejecución conserva el valor, que podemos ir incrementando en cada llamada. Ver Código fuente 176.

```
Public Sub Main()
    Verificar("Primera") ' en esta llamada se muestra 7
    Verificar("Segunda") ' en esta llamada se muestra 9
    Verificar("Tercera") ' en esta llamada se muestra 11
    Console.ReadLine()
End Sub

Public Sub Verificar(ByVal OrdenLlamada As String)
    ' declarar variable con el modificador Static,
    ' en la primera llamada toma el valor inicial de 5,
    ' las sucesivas llamadas no ejecutarán esta línea
    Static Dim Importe As Integer = 5

    Importe += 2
    Console.WriteLine("{0} llamada al procedimiento, la variable contiene {1}", _
        OrdenLlamada, Importe)
End Sub
```

Código fuente 176

Las variables Static por lo tanto, tienen un periodo de vida que abarca todo el tiempo de ejecución del programa, mientras que su ámbito es a nivel de procedimiento o bloque, ya que también pueden crearse dentro de una estructura de control.

16

Funciones complementarias del lenguaje

Convenciones de notación

Las convenciones de notación consisten en una serie de normas no oficiales a la hora de declarar elementos en el código, que facilitan su interpretación y mantenimiento.

Si bien esto no es inicialmente necesario, ni la herramienta de programación obliga a ello, en la práctica se ha demostrado que una serie de normas a la hora de escribir el código redundan en una mayor velocidad de desarrollo y facilidad de mantenimiento de la aplicación. Siendo útil no sólo en grupos de trabajo, sino también para programadores independientes.

Seguidamente describiremos una serie de normas de codificación para variables y constantes, que no son en absoluto obligatorias a la hora de escribir el código del programa, pero si pretenden concienciar al lector de la necesidad de seguir unas pautas comunes a la hora de escribir dicho código, de manera que al compartirlo entre programadores, o cuando tengamos que revisar una aplicación desarrollada tiempo atrás, empleemos el menor tiempo posible en descifrar lo que tal o cual variable significa en el contexto de una rutina o módulo.

- **Variables**. El formato utilizado para la notación de variables se basa en utilizar un carácter para indicar el ámbito de la variable, seguido de uno o dos caracteres para especificar el tipo de dato y el resto del nombre que daremos a la variable o cuerpo. Ver el Código fuente 177.

```
<Ámbito><TipoDato><Cuerpo>
```

Código fuente 177

La Tabla 21 muestra los valores para ámbito.

Carácter	Ámbito que define
l	Local
m	Módulo (privado)
p	Proyecto (público)

Tabla 21. Caracteres para indicar el ámbito en los nombres de variables.

La Tabla 22 muestra los valores para el tipo de dato.

Carácter	Tipo de dato que define
b	Boolean
by	Byte
c	Char
dt	Date
dc	Decimal
db	Double
i	Integer
l	Long
sh	Short
sg	Single
o	Object
s	String

Tabla 22. Caracteres para indicar el tipo de dato en los nombres de las variables.

Para el cuerpo de la variable se utilizará WordMixing, que consiste en una técnica en la cuál empleamos, si es necesario, varias palabras juntas para describir mejor el contenido de la variable. Veamos unos ejemplos en el Código fuente 178.

```
' variable local de tipo integer
liCodAcceso
```

```
' variable a nivel de módulo de tipo string
msNombreUsuario

' variable a nivel de proyecto de tipo fecha
pdtDiaAlta
```

Código fuente 178

En el caso de objetos creados por el programador, utilizaremos como prefijo para el tipo de dato, el carácter "o", o bien tres caracteres indicativos de la clase. Ver el Código fuente 179.

```
' variable local cuyo tipo es un objeto creado por el programador
Dim loEmpleado As Empleado
Dim lempEmpleado As Empleado
```

Código fuente 179

- **Constantes**. En este caso seguiremos el mismo formato de notación que para las variables en lo que respecta al ámbito y tipo de dato. El cuerpo de la constante sin embargo, deberemos escribirlo en mayúsculas, y separar las distintas palabras utilizando el carácter de guión bajo (_) en vez de WordMixing. Código fuente 180.

```
' constante a nivel de proyecto de tipo integer
piTIPO_IVA

' constante a nivel de módulo de tipo string
msCOLOR_INICIAL
```

Código fuente 180

Funciones de comprobación de tipos de datos

Si tenemos desactivada la comprobación de tipos con Option Strict, pero en ciertas situaciones necesitamos comprobar si determinada variable o expresión contienen un valor numérico, fecha, etc., el lenguaje nos proporciona para ello, algunas funciones con las que podremos comprobar el tipo de dato, para evitar posibles errores.

- **IsNumeric( )**. Esta función devuelve un valor lógico indicando si la expresión que pasamos como parámetro contiene un número o una cadena que pueda ser convertida a número. Ver el Código fuente 181.

```
Public Sub Main()
    Dim Valor As Object
    Dim Total As Integer

    Console.WriteLine("Introducir un número")
    Valor = Console.ReadLine()

    If IsNumeric(Valor) Then
        Total = Valor + 100
        Console.WriteLine("Resultado: {0}", Total)
```

```
    Else
        Console.WriteLine("El valor introducido no es numérico")
    End If

    Console.ReadLine()
End Sub
```

Código fuente 181

- **IsDate( )**. Esta función devuelve un valor lógico indicando si la expresión que pasamos como parámetro contiene una fecha o una cadena que pueda ser convertida a fecha. Ver el Código fuente 182.

```
Public Sub Main()
    Dim Valor As Object
    Dim UnaFecha As Date

    Console.WriteLine("Introducir una fecha")
    Valor = Console.ReadLine()

    If IsDate(Valor) Then
        UnaFecha = Valor
        Console.WriteLine("La fecha es: {0}", UnaFecha)
    Else
        Console.WriteLine("El valor introducido no es una fecha")
    End If

    Console.ReadLine()
End Sub
```

Código fuente 182

- **IsArray( )**. Esta función devuelve un valor lógico indicando si la expresión que pasamos como parámetro contiene un array. Ver el Código fuente 183.

```
Public Sub Main()
    Dim Colores() As String = {"Verde", "Azul", "Rojo"}

    Verificar(Colores)
    Verificar("prueba")
    Console.ReadLine()
End Sub

Public Sub Verificar(ByVal ValorPasado As Object)
    ' comprobar si el parámetro contiene un array
    If IsArray(ValorPasado) Then
        Console.WriteLine("El parámetro pasado es un array")
    Else
        Console.WriteLine("El parámetro pasado no es un array")
    End If
End Sub
```

Código fuente 183

Funciones del lenguaje

Cada lenguaje dispone de un grupo de funciones de apoyo, para ayudar al programador en su trabajo cotidiano. Las versiones anteriores de Visual Basic contenían un gran número de funciones para realizar operaciones aritméticas, manipular cadenas, fechas, etc.

VB.NET también tiene funciones para las operaciones antes comentadas. No obstante, debido a la orientación a objetos sobre la que está construida la plataforma .NET, la gran potencia a la hora de resolver cualquier situación la encontraremos en el gran número de clases proporcionadas por el entorno para resolver las más variadas situaciones, lo que veremos en el tema dedicado a OOP.

En este apartado y organizadas por categorías, vemos una pequeña muestra de las funciones disponibles en VB.NET. Consulte el lector la documentación de la plataforma, para obtener información más detallada de todas las funciones disponibles.

Numéricas

- **Int(Número), Fix(Número)**. Estas funciones devuelven la parte entera del parámetro Número. La diferencia entre ambas reside en que cuando el parámetro pasado es negativo, Int( ) devuelve el entero negativo menor o igual que Número, mientras que Fix( ) devuelve el entero negativo mayor o igual que Número. Ver el Código fuente 184.

```
Dim Resultado As Integer

Resultado = Int(66.87)  ' 66
Resultado = Fix(66.87)  ' 66

Resultado = Int(-66.87) ' -67
Resultado = Fix(-66.87) ' -66
```

Código fuente 184

- **Randomize([Número])**. Inicializa el generador de números aleatorios, que utilizaremos posteriormente en la función Rnd( ). Opcionalmente recibe un número como parámetro que sirve al generador como valor inicial o semilla para la creación de estos números.

- **Rnd([Número])**. Devuelve un número aleatorio de tipo Single, que será menor que 1, pero mayor o igual a cero.

Podemos, opcionalmente, variar el modo de generación del número pasando un valor al parámetro de esta función. En función de si el parámetro es mayor, menor de cero, o cero, el comportamiento de Rnd( ) a la hora de generar el número será diferente. Ver el Código fuente 185.

```
Dim Contador As Integer
Dim Aleatorio As Single

Randomize()
For Contador = 1 To 10
    Aleatorio = Rnd()
    Console.WriteLine("Número generado: {0}", Aleatorio)
Next
```

```
Console.ReadLine()
```

Código fuente 185

El anterior código produce una salida similar a la mostrada en la Figura 189.

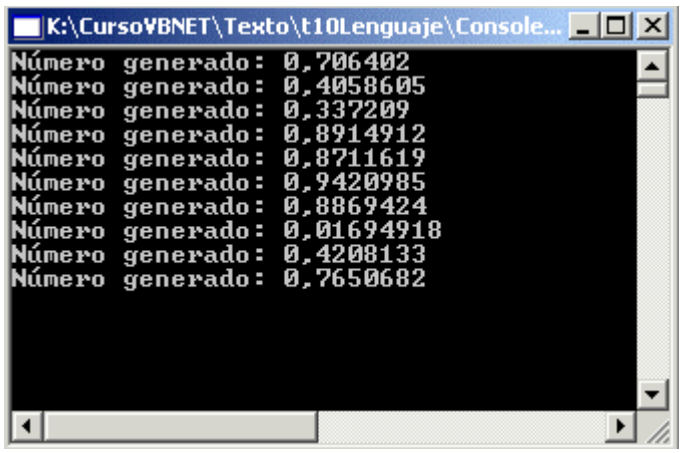


Figura 189. Generación de números aleatorios con Rnd( ).

Si necesitamos que el número aleatorio esté comprendido en un intervalo de números enteros, utilizaremos la fórmula del Código fuente 186 para generarlo.

```
Int((LímiteSuperior - LímiteInferior + 1) * Rnd() + LímiteInferior)
```

Código fuente 186

El ejemplo del Código fuente 187 crea números aleatorios comprendidos entre el intervalo de los números 7 y 12.

```
Dim Contador As Integer
Dim Aleatorio As Single

Randomize()
For Contador = 1 To 10
    Aleatorio = Int((12 - 7 + 1) * Rnd() + 7)
    Console.WriteLine("Número generado: {0}", Aleatorio)
Next
Console.ReadLine()
```

Código fuente 187

Cadena de caracteres

- **Len(Cadena)**. Devuelve un número con la longitud de la cadena pasada como parámetro. Ver el Código fuente 188.

```
Dim Longitud As Integer
Longitud = Len("comprobar cuantos caracteres hay")
Console.WriteLine("La cadena tiene {0} caracteres", Longitud)   ' 32
```

Código fuente 188

- **Space(Número)**. Devuelve una cadena de espacios en blanco, de una longitud igual al número pasado como parámetro. Ver el Código fuente 189.

```
Dim ConEspacios As String
ConEspacios = "Hola" & Space(7) & "a todos"
Console.WriteLine("La cadena con espacios tiene el valor:" & _
    ControlChars.CrLf & ConEspacios)    ' Hola       a todos
```

Código fuente 189

- **InStr([Comienzo,]CadenaBuscar, CadenaBuscada [, TipoComparación])**. Busca dentro de CadenaBuscar la cadena contenida en el parámetro CadenaBuscada. Opcionalmente podemos establecer en Comienzo, la posición en la que comienza la búsqueda y el tipo de comparación (texto, binaria) en el parámetro TipoComparación. Ver el Código fuente 190.

```
Dim CadBuscar As String
Dim CadBuscada As String
Dim PosComienzo As Integer

CadBuscar = "El castillo del bosque"
PosComienzo = InStr(CadBuscar, "tillo")
Console.WriteLine("La posición de comienzo de la cadena encontrada es: {0}",
_
    PosComienzo)    ' 7
```

Código fuente 190

- **Left(Cadena, Longitud)**. Esta función extrae, comenzando por la parte izquierda de Cadena, una subcadena de Longitud de caracteres.

- **Right(Cadena, Longitud)**. Esta función extrae, comenzando por la parte derecha de Cadena, una subcadena de Longitud de caracteres. El Código fuente 191 muestra ejemplos de Left( ) y Right( ).

```
Dim CadIzquierda As String
Dim CadDerecha As String

CadIzquierda = Left("Especial", 3)
Console.WriteLine("Resultado de la función Left(): {0}", CadIzquierda)  ' Esp

CadDerecha = Right("Especial", 3)
Console.WriteLine("Resultado de la función Right(): {0}", CadDerecha)   ' ial
```

Código fuente 191.

- **Mid(Cadena, Inicio [, Longitud])**. Extrae de Cadena, comenzando en la posición Inicio, una subcadena. Opcionalmente podemos utilizar el parámetro Longitud, para indicar el largo de la subcadena. En caso de no utilizar este último parámetro, la subcadena se obtendrá hasta el final. Ver Código fuente 192.

```
Dim MiCadena As String
Dim SubCadena As String

MiCadena = "El bosque encantado"
SubCadena = Mid(MiCadena, 6)
Console.WriteLine("Subcadena hasta el final: {0}", SubCadena)   ' sque
encantado

SubCadena = Mid(MiCadena, 6, 3)
Console.WriteLine("Subcadena de 3 caracteres: {0}", SubCadena)  ' squ
```

Código fuente 192

- **Replace(Cadena,CadOrigen,CadNueva [,Inicio] [,Sustituciones] [,TipoComparación])**. Esta función toma la cadena situada en el primer parámetro y busca la cadena CadOrigen, sustituyendo las ocurrencias encontradas por la cadena CadNueva. Opcionalmente, el parámetro Inicio especifica la posición en la que comenzará la sustitución; el parámetro Sustituciones indica el número de sustituciones a realizar; y TipoComparación indica como se realizarán las comparaciones (texto, binaria). Veamos unos ejemplos en el Código fuente 193.

```
Dim MiCadena As String
Dim CadSustituida As String

MiCadena = "Este coche es especial"

CadSustituida = Replace(MiCadena, "es", "xx")
' resultado: Este coche xx xxpecial
Console.WriteLine("Resultado del reemplazo en la cadena: {0}", CadSustituida)

' en el anterior ejemplo los dos primeros caracteres
' no se sustituyen porque no se ha especificado el tipo
' de comparación, que a continuación sí indicaremos
CadSustituida = Replace(MiCadena, "es", "xx", , , CompareMethod.Text)
' resultado: xxte coche xx xxpecial
' ahora sí se han sustituido todas las ocurrencias de "es"
Console.WriteLine("Resultado del reemplazo en la cadena: {0}", CadSustituida)
```

Código fuente 193

- **LTrim(Cadena), RTrim(Cadena), Trim(Cadena)**. Estas funciones eliminan de una cadena, los espacios en blanco a la izquierda en el caso de LTrim( ); los espacios en blanco a la derecha en el caso de RTrim(); o los espacios en blanco a ambos lados Trim( ). Ver el Código fuente 194.

```
Dim CadEspacios As String
Dim CadResultante As String
CadEspacios = "  Barco    "
CadResultante = LTrim(CadEspacios)  ' "Barco    "
CadResultante = RTrim(CadEspacios)  ' "  Barco"
```

```
CadResultante = Trim(CadEspacios)   ' "Barco"
```

Código fuente 194

- **UCase(Cadena), LCase(Cadena)**. Estas funciones, convierten la cadena pasada como parámetro a mayúsculas y minúsculas respectivamente. Ver el Código fuente 195.

```
Dim Cadena As String
Dim CadMay As String
Dim CadMin As String

Cadena = "Vamos a Convertir En Mayúsculas Y MinúscuLAS"
CadMay = UCase(Cadena)
CadMin = LCase(Cadena)

' "VAMOS A CONVERTIR EN MAYÚSCULAS Y MINÚSCULAS"
Console.WriteLine("Conversión a mayúsculas: {0}", CadMay)

' "vamos a convertir en mayúsculas y minúsculas"
Console.WriteLine("Conversión a minúsculas: {0}", CadMin)
```

Código fuente 195

- **Format(Expresión [,CadenaFormato] [,PrimerDíaSemana] [,PrimeraSemanaAño])**. Formatea la expresión pasada en el primer parámetro, empleando de forma opcional una cadena para especificar el tipo de formateo a realizar. Si el valor a formatear es una fecha, podemos utilizar los dos últimos parámetros para especificar el primer día de la semana y la primera semana del año; estos dos últimos parámetros son enumeraciones, cuyos valores aparecen automáticamente al asignar su valor. Consulte el lector, la documentación de ayuda para más información.

  Como cadena de formato, podemos utilizar los nombres predefinidos de formato, o una serie de caracteres especiales, tanto para formateo de números como de fechas. En lo que respecta a los nombres predefinidos, la Tabla 23 muestra algunos de los utilizados.

Nombre de formato	**Descripción**
General Date	Muestra una fecha con el formato largo del sistema.
Short Date	Muestra una fecha empleando el formato corto del sistema.
Short Time	Muestra un valor horario con el formato corto del sistema.
Standard	Muestra un número utilizando los caracteres de separador de miles y decimales.
Currency	Muestra un número con los caracteres correspondientes a la moneda establecida en la

	configuración regional del sistema.
Percent	Muestra un número multiplicado por 100 y con el carácter de tanto por ciento.

Tabla 23. Nombres de formato para la función Format( ).

El Código fuente 196 muestra algunos ejemplos de formateo con nombre

```
Dim MiFecha As Date
Dim MiNumero As Double
Dim ValorFormato As String

MiFecha = #7/19/2002 6:25:00 PM#
MiNumero = 1804

ValorFormato = Format(MiFecha, "Long Date")      ' "viernes, 19 de julio de
2002"
ValorFormato = Format(MiFecha, "Short Date")     ' "19/07/2002"
ValorFormato = Format(MiFecha, "Short Time")     ' "18:25"

ValorFormato = Format(MiNumero, "Standard")      ' "1.804,00"
ValorFormato = Format(MiNumero, "Currency")      ' "1.804 pta"
ValorFormato = Format(MiNumero, "Percent")       ' "180400,00%"
```

Código fuente 196

Para los caracteres especiales, la Tabla 24 muestra un conjunto de los más habituales.

Carácter de formato	**Descripción**
:	Separador de hora.
/	Separador de fecha.
d	Visualiza el número de día sin cero a la izquierda.
dd	Visualiza el número de día con cero a la izquierda.
ddd	Visualiza el nombre del día abreviado.
dddd	Visualiza el nombre del día completo.
M	Visualiza el número de mes sin cero a la izquierda.
MM	Visualiza el número de mes con cero a la izquierda.
MMM	Visualiza el nombre del mes abreviado.
MMMM	Visualiza el nombre del mes completo.

yy	Visualiza dos dígitos para el año.
yyyy	Visualiza cuatro dígitos para el año.
H	Visualiza la hora sin cero a la izquierda.
HH	Visualiza la hora con cero a la izquierda.
m	Visualiza los minutos cero a la izquierda.
mm	Visualiza los minutos con cero a la izquierda.
s	Visualiza los segundos cero a la izquierda.
ss	Visualiza los segundos con cero a la izquierda.
0	En valores numéricos, muestra un dígito o cero.
#	En valores numéricos, muestra un dígito o nada.
,	Separador de millar.
.	Separador decimal.

Tabla 24. Caracteres de formato para la función Format( ).

El Código fuente 197 muestra algunos ejemplos de formato con caracteres especiales.

```
Dim MiFecha As Date
Dim MiNumero As Double
Dim ValorFormato As String

MiFecha = #7/19/2002 6:25:00 PM#
MiNumero = 16587.097

ValorFormato = Format(MiFecha, "dddd d/MMM/yyyy")   ' "viernes 19/jul/2002"
ValorFormato = Format(MiFecha, "HH:mm")             ' "18:25"
ValorFormato = Format(MiNumero, "#,#.00")           ' "16.587,10"
```

Código fuente 197

- **StrConv(Cadena, TipoConversión [,IDLocal])**. Realiza una conversión de la cadena pasada como parámetro, utilizando algunos de los valores de la enumeración TipoConversión. Opcionalmente podemos pasar también un valor correspondiente al identificador local del sistema. Ver el Código fuente 198.

```
Dim MiCadena As String
Dim Conversion As String
MiCadena = "el tren llegó puntual"
' convertir a mayúscula
Conversion = StrConv(MiCadena, VbStrConv.UpperCase)
' convertir a minúscula
```

```
Conversion = StrConv(MiCadena, VbStrConv.LowerCase)
' convertir a mayúscula la primera letra
' de cada palabra
Conversion = StrConv(MiCadena, VbStrConv.ProperCase)
```

Código fuente 198

Fecha y hora

- **Now( )**. Devuelve un valor de tipo Date con la fecha y hora del sistema.
- **DateAdd(TipoIntervalo, ValorIntervalo, Fecha)**. Suma o resta a una fecha, un intervalo determinado por el parámetro TipoIntervalo. El intervalo a utilizar pueden ser días, semanas, meses, etc. Para determinar si se realiza una suma o resta, ValorIntervalo deberá ser positivo o negativo respectivamente.
- **DateDiff(TipoIntervalo, FechaPrimera, FechaSegunda)**. Calcula la diferencia existente entre dos fechas. En función de TipoIntervalo, la diferencia calculada serán días, horas, meses, años, etc.
- **DatePart(TipoIntervalo, Fecha)**. Extrae la parte de una fecha indicada en TipoIntervalo. Podemos obtener, el día, mes, año, día de la semana, etc.

El Código fuente 199 muestra un conjunto de ejemplos que utilizan las funciones para manipular fechas.

```
Dim MiFecha As Date
Dim FechaPosterior As Date
Dim DiasDiferencia As Long
Dim ParteFecha As Integer

MiFecha = Now()  ' #1/19/2002 12:27:08 PM#
FechaPosterior = DateAdd(DateInterval.Month, 2, MiFecha) ' #3/19/2002 12:27:08 PM#
DiasDiferencia = DateDiff(DateInterval.Day, MiFecha, FechaPosterior)  ' 59
ParteFecha = DatePart(DateInterval.Year, MiFecha)  ' 2002
```

Código fuente 199

Crear múltiples entradas al programa mediante distintos Main( )

Dentro de un proyecto es posible crear varios procedimientos Main( ), eligiendo desde la ventana de propiedades del proyecto, por cuál de ellos vamos a iniciar la ejecución.

En el tema *Escritura de código*, apartado *Configurar el punto de entrada del proyecto*, vimos como establecer el punto de entrada a la aplicación, a través de la ventana de propiedades del proyecto, seleccionando uno de los valores de la lista desplegable *Objeto inicial*. Consulte el lector dicho tema para una información más detallada.

Cuando creamos una aplicación de tipo consola, por defecto se crea un módulo con el nombre Module1, que contiene un Main() vacío, y en las propiedades del proyecto, el objeto inicial es dicho Module1.

Si añadimos nuevos módulos al proyecto, bien en ficheros separados, o agrupando varios módulos en un mismo fichero, podemos escribir un procedimiento Main( ) para cada uno de los módulos de nuestro proyecto. El Código fuente 200 muestra un ejemplo en el que tenemos dos módulos en un proyecto, con un procedimiento Main( ) dentro de cada uno.

```
Module Module1

    Public Sub Main()
        Console.WriteLine("Iniciamos el programa en el modulo General")
        Console.ReadLine()
    End Sub

End Module

Module Calculos

    Public Sub Main()
        Console.WriteLine("Iniciamos el programa en el modulo Calculos")
        Console.ReadLine()
    End Sub

End Module
```

Código fuente 200

Por defecto, y ya que así se establece al crear el proyecto, la ejecución comenzará por el Main( ) del módulo Module1. Pero podemos hacer que el procedimiento de inicio sea el Main( ) que está en el módulo Calculos, abriendo la ventana de propiedades del proyecto y seleccionando como objeto inicial dicho módulo. Ver Figura 190.

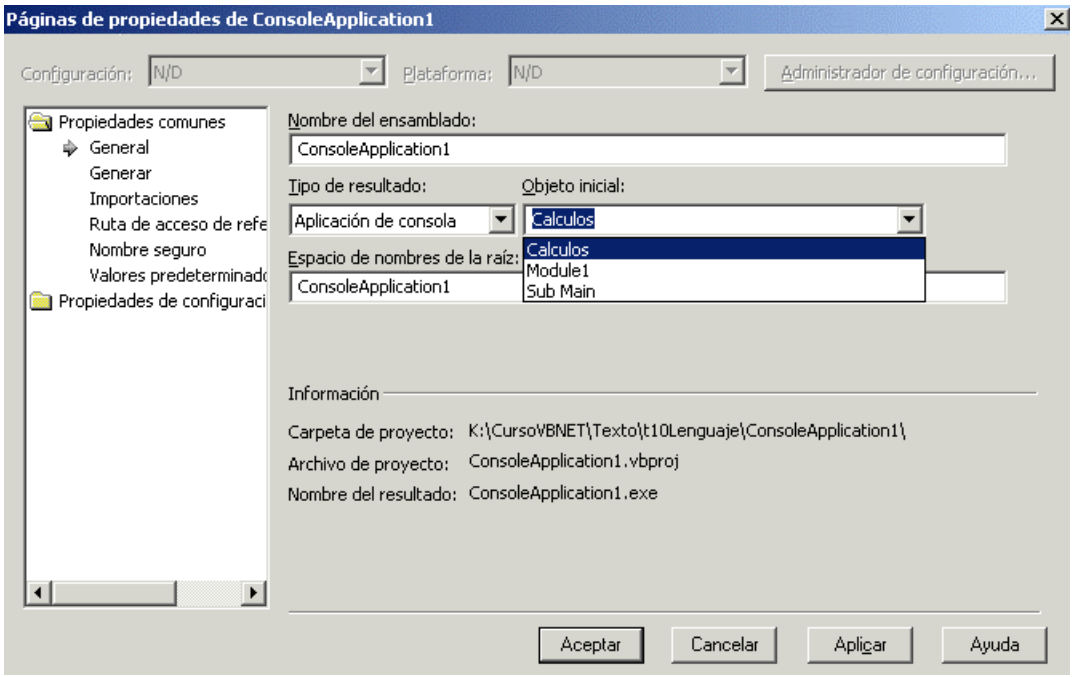


Figura 190. Establecer un módulo como objeto inicial del proyecto.

Con esta técnica, podremos disponer de tantos procedimientos de inicio como módulos contenga nuestro proyecto.

No obstante, si sólo deseamos que exista un único procedimiento Main( ) a lo largo de todo el código de nuestra aplicación, en la lista desplegable *Objeto inicial*, de la ventana de propiedades del proyecto, tendremos que seleccionar la opción Sub Main; esto nos obligará a tener sólo un procedimiento Main() dentro de cualquiera de los módulos, produciéndose un error si al comienzo de la ejecución se detecta más de una versión de Main( ).

17

Programación orientada a objeto (OOP)

Las ventajas de la programación orientada a objeto

La programación orientada a objeto, OOP (Object Oriented Programming) a partir de ahora, se trata de una evolución de la programación procedural basada en funciones, que permite agrupar elementos de código (rutinas y datos) con funcionalidades similares, bajo un sistema unificado de manipulación y acceso a dichos elementos.

Del enfoque procedural al enfoque orientado a objeto

En la programación estructurada procedural, basada en procedimientos y funciones, el crecimiento de una aplicación hace que el mantenimiento de la misma se convierta en una tarea difícil, debido al gran número de procedimientos interrelacionados que podemos llegar a tener. El mero hecho de efectuar una pequeña modificación en un proceso, nos puede suponer el tener que recorrer un gran número de funciones del programa, no relacionadas por un nexo común.

Abordando un problema mediante programación procedural

Tomemos el ejemplo de un programador al que le encargan el desarrollo de una aplicación para la gestión de una empresa. Entre los diferentes cometidos a resolver, se encuentra el control de los empleados en lo que respecta a su alta, pago de sueldos, cálculo de vacaciones, etc.

El programador se pone manos a la obra, desarrollando una aplicación basada en un enfoque procedural. Al llegar a los procesos relativos al empleado, va escribiendo las diferentes rutinas, distribuyéndolas a lo largo de los diferentes módulos que componen el programa. Ver el Código fuente 201.

```
Module General

    Public psNombre As String

    Public Sub Main()
        ' procedimiento de inicio del programa,
        ' aquí mostramos por ejemplo un menú
        ' para seleccionar alguno de los procesos
        ' del programa: altas de empleados,
        ' cálculo de nómina, periodos vacacionales, etc.
        ' ......
        ' ......
        ' ......
    End Sub

    Public Sub CalcularVacaciones(ByVal liIDEmpleado As Integer, _
        ByVal ldtFechaInicio As Date, ByVal liNumDias As Integer)
        ' en este procedimiento calculamos
        ' el periodo de vacaciones del empleado
        ' pasado como parámetro
        Dim ldtFechaFinal As Date
        ' ......
        ' obtener el nombre del empleado en función de su identificador
        psNombre = "Juan"
        psApellidos = "Plaza"
        ' ......
        ' ......
        ' calcular la fecha final y mostrar
        ' el periodo vacacional
        ldtFechaFinal = DateAdd(DateInterval.Day, liNumDias, ldtFechaInicio)
        Console.WriteLine("Empleado: {0} {1}", psNombre, psApellidos)
        Console.WriteLine("Vacaciones desde {0} hasta {1}", _
            Format(ldtFechaInicio, "dd/MMM/yy"), _
            Format(ldtFechaFinal, "d/MMMM/yyyy"))
        Console.ReadLine()
    End Sub

    ' otros procedimientos del módulo
    ' ......
    ' ......
End Module

Module Varios

    Public psApellidos As String

    Public Sub CrearEmpleado(ByVal liIDEmpleado As Integer, _
        ByVal lsNombre As String, ByVal lsApellidos As String, _
        ByVal lsDNI As String, ByVal ldtFechaAlta As Date)
        ' grabamos los datos de un nuevo empleado en la
        ' base de datos que utiliza el programa
        ' ......
        Console.WriteLine("Se ha grabado el empleado: {0} - {1} {2}", _
            liIDEmpleado, lsNombre, lsApellidos)
        Console.ReadLine()
    End Sub

    ' otros procedimientos del módulo
    ' ......
```

```
    ' ......
End Module

Module Pagos

    Public Sub TransfNomina(ByVal liIDEmpleado As Integer, ByVal ldbImporte As
Double)
        ' realizamos la transferencia de nómina
        ' a un empleado, utilizando su identificador
        ' ......
        ' obtenemos los datos del empleado
        psNombre = "Ana"
        psApellidos = "Roca"
        ' ......
        ' visualizamos el resultado
        Console.WriteLine("Pago de nómina")
        Console.WriteLine("Empleado: {0} {1}", psNombre, psApellidos)
        Console.WriteLine("Ingresado: {0}", ldbImporte)
        Console.ReadLine()
    End Sub

    Public Sub MostrarEmpleado(ByVal liIDEmpleado As Integer)
        ' buscar la información del empleado por su identificador
        Dim lsDatosEmpleado As String
        ' ......
        psNombre = "isabel"
        psApellidos = "casillas"
        lsDatosEmpleado = StrConv(psNombre & " " & psApellidos,
VbStrConv.ProperCase)
        Console.WriteLine("El empleado seleccionado es: {0}", lsDatosEmpleado)
        Console.ReadLine()
    End Sub

    ' otros procedimientos del módulo
    ' ......
    ' ......
End Module
```

Código fuente 201

En el ejemplo anterior se declaran variables públicas en diferentes módulos del proyecto, y se crean procedimientos para las tareas relacionadas con el alta, visualización de datos, pagos, etc., del empleado. Todo este código se encuentra disperso a lo largo del programa, por lo que su mantenimiento, según crezca la aplicación, se hará progresivamente más difícil.

Dejemos por el momento, a nuestro atribulado programador, pensando en cómo resolver este problema; posteriormente volveremos a él para aportarle una solución, que vendrá naturalmente, de mano de la OOP.

Los fundamentos de la programación orientada a objeto

La organización de una aplicación en OOP se realiza mediante *estructuras de código*.

Una estructura de código contiene un conjunto de procedimientos e información que ejecutan una serie de procesos destinados a resolver un grupo de tareas con un denominador común. Una aplicación orientada a objetos tendrá tantas estructuras de código como aspectos del programa sea necesario resolver.

Un procedimiento que esté situado dentro de una de estructura de este tipo, no podrá llamar ni ser llamado por otro procedimiento situado en una estructura distinta, si no es bajo una serie de reglas. Lo mismo sucederá con los datos que contenga la estructura, permanecerán aislados del exterior, y sólo serán accesibles siguiendo ciertas normas. Una estructura de código, es lo que en OOP identificamos como objeto.

Al ser las estructuras de código u objetos, entidades que contienen una información precisa y un comportamiento bien definido a través del conjunto de procedimientos que incluyen, pueden ser clasificados en función de las tareas que desempeñan. Precisamente, uno de los fines perseguidos por la OOP es conseguir una mejor catalogación del código, en base a estructuras jerárquicas dependientes, al estilo de un árbol genealógico.

Trasladando las nociones que acabamos de exponer al ejemplo anterior, en el cual se programaban los procesos de gestión de los empleados de una empresa, el resultado obtenido será una estructura de código conteniendo todos los procedimientos, funciones y variables de la aplicación, implicados en las operaciones a realizar con un empleado, o lo que es lo mismo, un objeto Empleado. Entre los elementos de este objeto encontraremos el nombre, apellidos, alta del empleado, pago de nómina, etc.

Todos los elementos que forman parte de un objeto componen la clase del objeto. Una clase consiste en el conjunto de especificaciones que permiten crear los objetos; en el caso expuesto por el ejemplo anterior sería la clase Empleado.

Como acabamos de comprobar, las motivaciones que han llevado al desarrollo de la OOP son facilitar una mejor organización y clasificación del código, que la proporcionada por la programación procedural tradicional; aproximando al mismo tiempo, el modo de programar a la manera en que nuestra mente trabaja para aplicar soluciones a los problemas planteados.

Objetos

Un objeto es una agrupación de código, compuesta de propiedades y métodos, que pueden ser manipulados como una entidad independiente. Las propiedades definen los datos o información del objeto, permitiendo consultar o modificar su estado; mientras que los métodos son las rutinas que definen su comportamiento.

Un objeto es una pieza que se ocupa de desempeñar un trabajo concreto dentro de una estructura organizativa de nivel superior, formada por múltiples objetos, cada uno de los cuales ejerce la tarea particular para la que ha sido diseñado.

Clases

Una clase no es otra cosa que el conjunto de especificaciones o normas que definen cómo va a ser creado un objeto de un tipo determinado; algo parecido a un manual de instrucciones conteniendo las indicaciones para crear el objeto.

Los términos objeto y clase son utilizados en OOP con gran profusión y en contextos muy similares, por lo que para intentar aclarar en lo posible ambos conceptos, diremos que una clase constituye la representación abstracta de algo, mientras que un objeto constituye la representación concreta de lo que una clase define.

La clase determina el conjunto de puntos clave que ha de cumplir un objeto para ser considerado perteneciente a dicha clase o categoría, ya que no es obligatorio que dos objetos creados a partir de la misma clase sean exactamente iguales, basta con que cumplan las especificaciones clave de la clase.

Expongamos ahora las anteriores definiciones mediante un ejemplo preciso: un molde para crear figuras de cerámica y las figuras obtenidas a partir del molde. En este caso, el molde representaría la clase Figura, y cada una de las figuras creadas a partir del molde, sería un objeto Figura. Cada objeto Figura tendrá una serie de propiedades comunes: tamaño y peso iguales; y otras propiedades particulares: un color distinto para cada figura.

Aunque objetos distintos de una misma clase pueden tener ciertas propiedades diferentes, deben tener el mismo comportamiento o métodos. Para explicar mejor esta circunstancia, tomemos el ejemplo de la clase Coche; podemos crear dos coches con diferentes características (color, tamaño, potencia, etc.), pero cuando aplicamos sobre ellos los métodos Arrancar, Acelerar o Frenar, ambos se comportan o responden de la misma manera.

Instancias de una clase

El proceso por el cuál se obtiene un objeto a partir de las especificaciones de una clase se conoce como instanciación de objetos. En la Figura 191 volvemos al ejemplo del molde y las figuras; en dicha imagen vemos un molde para fabricar figuras rectangulares, donde la clase Figura estaría representada por el molde, y cada uno de los objetos Figura (iguales en forma pero con la propiedad Color distinta), representaría una instancia de la clase.

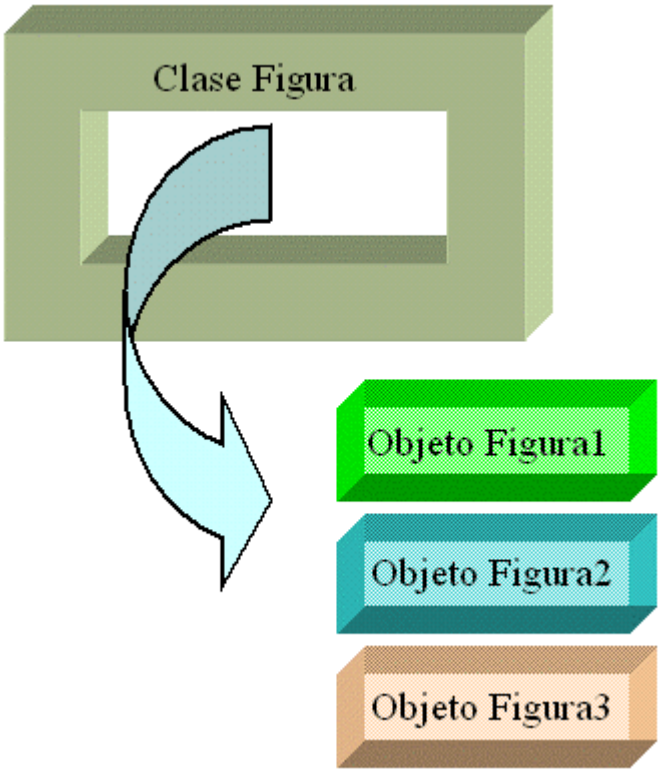


Figura 191. Instanciación de objetos a partir de una clase.

Características básicas de un sistema orientado a objeto

Para que un lenguaje o sistema sea considerado orientado a objeto, debe cumplir las características de los siguientes apartados.

Abstracción

La abstracción es aquella característica que nos permite identificar un objeto a través de sus aspectos conceptuales.

Las propiedades de los objetos de una misma clase, pueden hacerlos tan distintos que sea difícil reconocer que pertenecen a una clase idéntica. No obstante, nosotros reconocemos a qué clase pertenecen, identificando además, si se trata de la misma clase para ambos. Ello es posible gracias a la abstracción.

Tomemos como ejemplo dos objetos coche, uno deportivo y otro familiar; su aspecto exterior es muy diferente, sin embargo, cuando pensamos en cualquiera de ellos, sabemos que ambos pertenecen a la clase Coche, porque realizamos una abstracción o identificación mental de los elementos comunes que ambos tienen (ruedas, volante, motor, puertas, etc.).

Del mismo modo que hacemos al identificar objetos reales, la abstracción nos ayuda a la hora de desarrollar una aplicación, permitiéndonos identificar los objetos que van a formar parte de nuestro programa, sin necesidad de disponer aún de su implementación; nos basta con reconocer los aspectos conceptuales que cada objeto debe resolver.

Por ejemplo, cuando abordamos el desarrollo de un programa de gestión orientado a objetos, realizamos una abstracción de los objetos que necesitaríamos para resolver los procesos del programa: un objeto Empleado, para gestionar al personal de la empresa; un objeto Factura, para gestionar las ventas realizadas de productos; un objeto Usuario, para verificar las personas que utilizan la aplicación, etc.

Encapsulación

La encapsulación establece la separación entre el interfaz del objeto y su implementación, aportándonos dos ventajas fundamentales.

Por una parte proporciona seguridad al código de la clase, evitando accesos y modificaciones no deseadas; una clase bien encapsulada no debe permitir la modificación directa de una variable, ni ejecutar métodos que sean de uso interno para la clase.

Por otro lado la encapsulación simplifica la utilización de los objetos, ya que un programador que use un objeto, si este está bien diseñado y su código correctamente escrito, no necesitará conocer los detalles de su implementación, se limitará a utilizarlo.

Tomando un ejemplo real, cuando nosotros utilizamos un objeto Coche, al presionar el acelerador, no necesitamos conocer la mecánica interna que hace moverse al coche, sabemos que el método Acelerar del coche es lo que tenemos que utilizar para desplazarnos, y simplemente lo usamos.

Pasando a un ejemplo en programación, si estamos creando un programa de gestión y nos proporcionan un objeto Cliente que tiene el método Alta, y sirve para añadir nuevos clientes a la base

de datos, no precisamos conocer el código que contiene dicho método, simplemente lo ejecutamos y damos de alta a los clientes en nuestra aplicación.

Polimorfismo

El polimorfismo determina que el mismo nombre de método, realizará diferentes acciones según el objeto sobre el que sea aplicado. Al igual que sucedía en la encapsulación, el programador que haga uso del objeto, no necesita conocer los detalles de implementación de los métodos, se limita a utilizarlos.

Pasando a un ejemplo real, tomamos dos objetos: Pelota y VasoCristal; si ejecutamos sobre ambos el método Tirar, el resultado en ambos casos será muy diferente; mientras que el objeto Pelota rebotará al llegar al suelo, el objeto VasoCristal se romperá.

En un ejemplo aplicado a la programación, supongamos que disponemos de los objetos Ventana y Fichero; si ejecutamos sobre ambos el método Abrir, el resultado en Ventana será la visualización de una ventana en el monitor del usuario; mientras que en el objeto Fichero, se tomará un fichero en el equipo del usuario y se dejará listo para realizar sobre él operaciones de lectura o escritura.

Herencia

Se trata de la característica más importante de la OOP, y establece que partiendo de una clase a la que denominamos clase base, padre o superclase, creamos una nueva clase denominada clase derivada, hija, o subclase. En esta clase derivada dispondremos de todo el código de la clase base, más el nuevo código propio de la clase hija, que escribamos para extender sus funcionalidades.

A su vez podemos tomar una clase derivada, creando una nueva subclase a partir de ella, y así sucesivamente, componiendo lo que se denomina una jerarquía de clases, que explicaremos seguidamente.

Existen dos tipos de herencia: simple y múltiple. La herencia simple es aquella en la que creamos una clase derivada a partir de una sola clase base, mientras que la herencia múltiple nos permite crear una clase derivada a partir de varias clases base. El entorno de .NET Framework sólo permite utilizar herencia simple.

Como ejemplo real de herencia, podemos usar la clase Coche como clase base; en ella reconocemos una serie de propiedades como Motor, Ruedas, Volante, etc., y unos métodos como Arrancar, Acelerar, Frenar, etc. Como clase derivada creamos CocheDeportivo, en la cuál, además de todas las características mencionadas para la clase Coche, encontramos propiedades y comportamiento específicos como ABS, Turbo, etc.

Un ejemplo basado en programación consistiría en disponer de la ya conocida clase Empleado. Esta clase se ocupa, como ya sabemos, de las operaciones de alta de empleados, pago de nóminas, etc.; pero en un momento dado, surge la necesidad de realizar pagos a empleados que no trabajan en la central de la empresa, ya que se trata de comerciales que pasan la mayor parte del tiempo desplazándose. Para realizar dichos pagos usaremos Internet, necesitando el número de tarjeta de crédito y la dirección email del empleado. Resolveremos esta situación creando la clase derivada CiberEmpleado, que hereda de la clase Empleado, en la que sólo tendríamos que añadir las nuevas propiedades y métodos para las transacciones electrónicas, puesto que las operaciones tradicionales ya las tendríamos disponibles por el mero hecho de haber heredado de Empleado.

Jerarquías de clases

Como decíamos en un apartado anterior, uno de los fines de la OOP consiste en la clasificación del código; para ello se emplean jerarquías o árboles de clases, en los que a base de niveles, se muestra un conjunto de clases conectadas por una relación de herencia. Observemos el esquema de la Figura 192, en el que se muestra un ejemplo de la jerarquía de clases de medios de transporte.

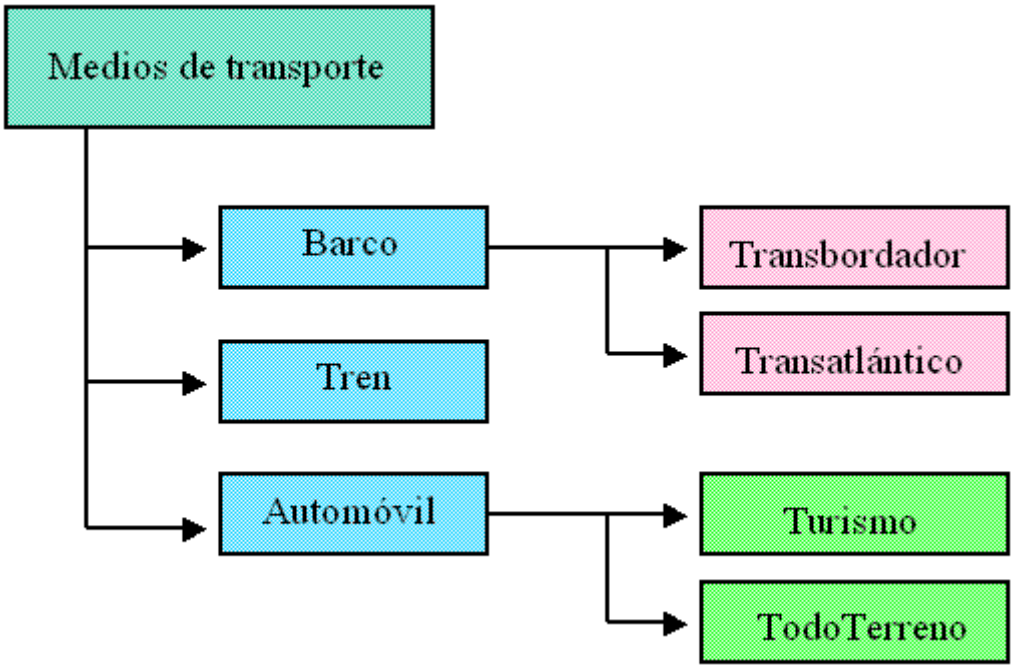


Figura 192. Jerarquía de clases de medios de transporte.

En esta representación de ejemplo, como nivel superior de la jerarquía o clase base estaría Medios de transporte, de la que se derivarían las clases Barco, Tren, Automóvil, y a su vez, de estas últimas, partirían nuevas clases hijas.

Relaciones entre objetos

Los objetos existentes en una aplicación se comunican entre sí mediante una serie de relaciones que describimos a continuación.

Herencia

Como acabamos de describir en el apartado sobre características de la OOP, cuando a partir de una clase existente, creamos una nueva clase derivada, esta nueva clase dispone de todas las propiedades y métodos de la clase base, mas el código propio que implemente.

Para reconocer si existe esta relación entre dos objetos, debemos realizar un análisis sintáctico sobre la misma usando la partícula “es un”.

Tomando como ejemplo los objetos Empleado, CiberEmpleado y Factura, podemos decir que sí hay una relación de herencia entre Empleado y CiberEmpleado, ya que al analizar la frase "Un objeto CiberEmpleado es un Empleado", el resultado es verdadero.

No ocurre lo mismo entre los objetos CiberEmpleado y Factura, ya que el análisis de la frase "Un objeto CiberEmpleado es una Factura", devuelve falso.

Pertenencia

Los objetos pueden estar formados a su vez por otros objetos. Un objeto Factura puede estar compuesto por objetos CabeceraFactura, LineaFactura, etc. Se dice en este caso que hay una relación de pertenencia, puesto que existe un conjunto de objetos que pertenecen a otro objeto o se unen para formar otro objeto. A este tipo de relación se le denomina también Contenedora.

Para reconocer si existe esta relación entre dos objetos, debemos realizar un análisis sintáctico sobre la misma usando la partícula "tiene un". Así, por ejemplo, la frase "Un objeto Factura tiene un objeto LineaFactura" devolvería verdadero.

Utilización

Hay situaciones en que un objeto utiliza a otro para realizar una determinada tarea, sin que ello suponga la existencia de una relación de pertenencia entre dichos objetos.

Por ejemplo, un objeto Ventana puede utilizar un objeto Empleado para mostrar al usuario las propiedades del empleado, sin necesidad de que el objeto Empleado sea propiedad del objeto Ventana.

Nótese la importante diferencia entre esta relación y la anterior, ya que aquí, el objeto Ventana a través de código, creará, o le será pasado como parámetro, un objeto Empleado, para poder mostrarlo en el área de la ventana.

Para reconocer si existe esta relación entre dos objetos, debemos realizar un análisis sintáctico sobre la misma empleando la partícula "usa un". Así, por ejemplo, la frase "Un objeto Ventana usa un objeto Empleado" devolvería verdadero.

Reutilización

Un objeto bien diseñado, puede ser reutilizado en otra aplicación de modo directo o creando una clase derivada a partir de él. Este es uno de los objetivos perseguidos por la OOP, aprovechar en lo posible el código ya escrito, ahorrando un considerable tiempo en el desarrollo de programas.

Análisis y diseño orientado a objetos

Antes de comenzar la escritura del programa, se hace necesario realizar un análisis de los problemas a resolver, que nos permita identificar qué procesos debemos codificar.

Si pretendemos además, abordar la programación utilizando un enfoque orientado a objetos, debemos emplear técnicas adecuadas a este tipo de programación.

Para aunar todas las tendencias de análisis orientadas a objetos existentes, ha aparecido el Lenguaje Unificado de Modelado o UML (Unified Modeling Language), cuyo objetivo es proporcionar un verdadero sistema de análisis y diseño aplicado a objetos.

La descripción de UML es algo que se encuentra fuera del alcance de este texto, por lo que recomendamos al lector consultar la documentación existente al respecto, de manera que pueda familiarizarse con este aspecto de la creación de un programa.

A modo de breve recomendación podemos decir, que cuando se realiza un análisis basado en procedimientos, de los problemas planteados, se identifican los verbos como elementos de los procesos a trasladar a procedimientos y funciones. Sin embargo, cuando se trata de un análisis basado en objetos, se identifican en este caso los nombres existentes en los procesos, como elementos a trasladar a objetos.

Tomemos el siguiente planteamiento: "Crear una aplicación en la que podamos realizar sobre una base de datos, las siguientes operaciones: añadir, borrar y modificar clientes. Por otro lado, será necesario crear facturas, grabando sus datos generales y calcular su importe total".

Analizando la exposición del anterior problema, si necesitáramos resolverlo mediante una aplicación con enfoque procedural, separaríamos los verbos para crear los siguientes procedimientos: AñadirCliente( ), BorrarCliente( ), ModificarCliente( ), GrabarFac( ), CalcularTotalFac( ).

Si por el contrario efectuamos sobre la misma exposición, un análisis orientado a objetos, extraeríamos los siguientes nombres como los objetos a crear: Cliente, Factura.

Para el objeto Cliente, definiríamos entre otras, las propiedades Nombre, Apellidos, Dirección, DNI, etc; creando para su comportamiento, los métodos Añadir( ), Borrar( ), Modificar( ), etc.

Para el objeto Factura, definiríamos entre otras, las propiedades Número, Fecha, Importe, etc; creando para su comportamiento, los métodos Grabar( ), CalcularTotal ( ), etc.

Una vez obtenido el correspondiente análisis, pasaremos a la siguiente fase del desarrollo, la escritura de las diferentes clases que van a componer nuestro programa, y que veremos a continuación.

Creación de clases

Volvamos al ejemplo expuesto al comienzo de este tema, en el cuál, habíamos dejado a nuestro programador desarrollando los procesos de un empleado dentro de una aplicación de gestión empresarial. Recordemos que se planteaba el problema de que ante el crecimiento del programa, el mantenimiento del código, al enfocarse de modo procedural, podía volverse una tarea realmente difícil.

Vamos a replantear este diseño, encauzándolo bajo una perspectiva orientada a objeto, que nos permita un uso más sencillo del código y un mantenimiento también más fácil. Para lo cual desarrollaremos una clase que contenga todas las operaciones a realizar por el empleado; en definitiva, crearemos la clase Empleado, cuyo proceso describiremos a continuación.

Podemos escribir una clase en VB.NET utilizando diferentes tipos de aplicación, en este caso emplearemos una aplicación de consola. Iniciaremos en primer lugar VS.NET, creando un proyecto de tipo consola. A continuación seleccionaremos el menú *Proyecto + Agregar clase*, que nos mostrará la ventana para agregar nuevos elementos al proyecto. El nombre por defecto asignado por el IDE para la clase será Class1; cambiaremos dicho nombre por Empleado, y pulsaremos Abrir. Ver Figura 193.

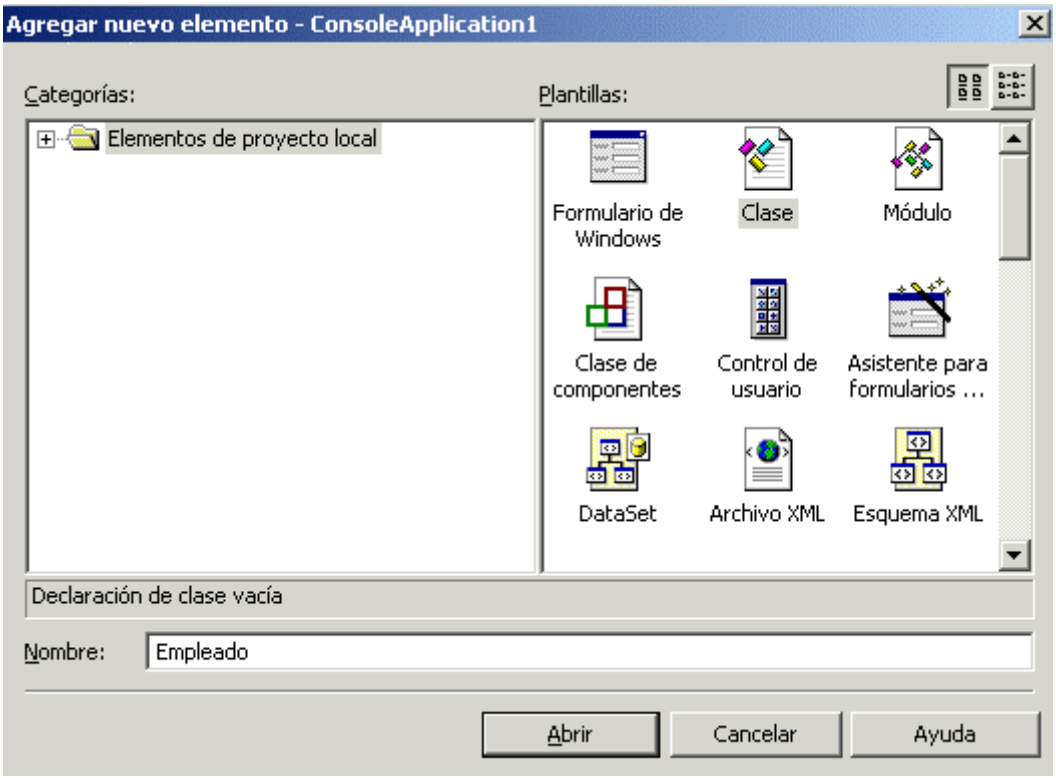


Figura 193. Añadir una clase a un proyecto.

Se creará de esta forma un nuevo fichero de código (EMPLEADO.VB), mostrándose el editor de código con su contenido. Observemos en este sentido, que la definición de una clase se realiza utilizando las palabras clave Class...End Class; entre estas palabras escribiremos el código de la clase. Ver Código fuente 202.

```
Public Class Empleado

End Class
```

Código fuente 202

Organización de clases en uno o varios ficheros de código

Como ya indicábamos en el tema *El lenguaje*, apartado *Organización del proyecto en ficheros y módulos de código*, una clase es uno de los tipos de contenedor lógico disponible dentro del entorno de .NET Framework, y su organización puede realizarse al igual que los módulos (Module) de código estándar del entorno, es decir, cada vez que añadimos una clase al proyecto utilizando el IDE, se crea por defecto, un fichero de código por clase.

Sin embargo, también podemos incluir varias clases dentro del mismo fichero de código, o mezclar clases con módulos y otro tipo de contenedores lógicos, cambiando si es necesario el nombre del fichero, como ya se explicó en el tema sobre el lenguaje. El ejemplo del Código fuente 203 muestra dos clases creadas dentro del mismo fichero.

```
' Fichero MisClases.VB
' ====================
Public Class Empleado
    ' código de la clase
    ' ......
    ' ......
End Class

Public Class Factura
    ' código de la clase
    ' ......
    ' ......
End Class
```

Código fuente 203

Código de clase y código cliente

Antes de comenzar a realizar pruebas con las clases que vayamos escribiendo, debemos explicar dos conceptos referentes a la escritura de código orientado a objeto, ya que en función del lugar desde el que sea manipulado, debemos distinguir entre código de clase y código cliente.

- **Código de clase**. Se trata del código que escribimos para crear nuestra clase, y que se encuentra entre las palabras clave Class...End Class.

- **Código cliente**. Se trata del código que hace uso de la clase mediante la acción de crear o instanciar objetos a partir de la misma. Aquí se englobaría todo el código que se encuentra fuera de la clase.

Reglas de ámbito generales para clases

Las normas de ámbito para variables y procedimientos escritos dentro de una clase son las mismas que las ya explicadas en el tema sobre el lenguaje, pero debemos tener en cuenta que en el caso actual, el ámbito a nivel de módulo debe ser equiparado con el ámbito a nivel de clase.

Por ejemplo, cuando declaremos una variable dentro de las palabras clave Class...End Class, pero fuera de todo procedimiento de la clase, dicha variable tendrá un ámbito a nivel de clase, siendo accesible por los procedimientos de la clase o por todo el código, según la hayamos definido privada o pública.

Existen unas reglas de ámbito específicas para los miembros de una clase que serán comentadas en un apartado posterior.

Instanciación de objetos

En este momento, nuestra clase Empleado cuenta con el código mínimo para poder ser utilizada, para lo que debemos *instanciar* objetos a partir de la misma.

Como ya se explicó en un apartado anterior, el proceso de instanciación consiste en crear un objeto a partir de las especificaciones de la clase. El modo más común de trabajar con una instancia de una clase, o lo que es lo mismo, con un objeto, pasa por asignar dicho objeto a una variable.

Instanciaremos un objeto en el código utilizando la sintaxis de declaración de variables junto a la palabra clave New, empleando como tipo de dato el nombre de la clase. Todo este código lo podemos situar en un módulo dentro del proyecto, bien en un fichero de código aparte o en el mismo fichero en donde estamos escribiendo la clase. El Código fuente 204 muestra las formas disponibles de instanciar un objeto y asignarlo a una variable.

```
Module General

    Sub Main()
       ' declarar primero la variable
       ' y después instanciar el objeto
       Dim loEmpleado1 As Empleado
       loEmpleado1 = New Empleado()

       ' declaración e instanciación simultánea
       Dim loEmpleado2 As New Empleado()

       ' declaración y posterior instanciación en
       ' la misma línea de código
       Dim loEmpleado3 As Empleado = New Empleado()
    End Sub

End Module
```

Código fuente 204

Si bien es cierto que ya es posible crear objetos a partir de nuestra clase, no lo es menos el hecho de que no podemos hacer grandes cosas con ellos, puesto que la clase se encuentra vacía de código. Debemos añadir propiedades y métodos para conseguir que los objetos actúen en nuestra aplicación.

Miembros de la clase

Los elementos de una clase que contienen sus datos y definen su comportamiento, es decir, las propiedades y métodos, reciben además el nombre de *miembros de la clase*, término que también utilizaremos a partir de ahora.

Definir la información de la clase

Existen dos formas de almacenar los datos o información en una clase: a través de campos de clase y de propiedades.

Desde la perspectiva del programador que hace uso de una clase para crear objetos, la diferencia entre un campo o una propiedad resulta imperceptible; sin embargo, desde el punto de vista del programador de la clase existen claras diferencias, concernientes fundamentalmente, a preservar la encapsulación del código de la clase.

El uso de campos o propiedades para una clase es una cuestión de diseño, no pudiendo afirmar categóricamente que un tipo de almacenamiento de datos sea mejor que otro.

Creación de campos para la clase

Un campo de una clase no es otra cosa que una variable, generalmente con ámbito público, accesible desde el exterior de la clase. El Código fuente 205 muestra la creación de un campo para la clase Empleado.

```
Public Class Empleado

    ' declaramos un campo en la clase
    ' para guardar el identificador
    ' del empleado
    Public piIdentificador As Integer

End Class
```

Código fuente 205

Para manipular un campo desde código cliente, debemos instanciar un objeto, a continuación de la variable que lo contiene situar un punto (.), y finalmente el nombre del campo a manipular. Este modo de operación es común para todos los miembros de clases, tanto creadas por el programador, como pertenecientes a la propia plataforma .NET Framework. Ver el Código fuente 206.

```
Module General

    Sub Main()
        Dim loEmpleado As Empleado

        ' instanciar el objeto
        loEmpleado = New Empleado()

        ' asignar un valor al campo del objeto
        loEmpleado.piIdentificador = 75

        ' mostrar el valor de un campo del objeto
        Console.WriteLine("El valor del campo es: {0}", loEmpleado.piIdentificador)
        Console.ReadLine()
    End Sub

End Module
```

Código fuente 206

Como habrá observado el lector, al escribir el nombre del objeto y el punto, aparece una lista con los miembros de la clase accesibles desde el código cliente. De momento sólo disponemos del campo y el método GetType( ), que devuelve un objeto de la clase Type, conteniendo información sobre el tipo del objeto. Esta lista irá aumentando progresivamente según añadimos más propiedades y métodos a la clase, constituyendo una inestimable ayuda para el programador, que le evita el tener que recordar los nombres de todos los elementos de la clase, o consultar continuamente su documentación.

Creación de propiedades para la clase

Una propiedad en la clase se define, por norma general, mediante dos elementos: una variable de propiedad y un procedimiento de propiedad.

La variable de propiedad, tal y como su nombre indica, es una variable con ámbito privado a nivel de la clase, que se encarga de guardar el valor de la propiedad. Por su parte el procedimiento de propiedad o Property, es el encargado de actuar de puente entre el código cliente y la variable de propiedad, realizando las operaciones de acceso y asignación de valores a dicha variable.

Por lo tanto, para crear una propiedad en nuestra clase, declararemos en primer lugar una variable Private, y en segundo lugar un procedimiento de tipo Property, que consta de dos bloques: Get, para devolver el valor de la variable de propiedad; y Set, para asignárselo. La sintaxis a emplear se muestra en el Código fuente 207.

```
Public Class Empleado

    ' declarar una variable de propiedad
    ' para la propiedad Nombre
    Private msNombre As String

    ' declarar el procedimiento Property
    ' para la propiedad Nombre
    Public Property Nombre() As String
        ' bloque Get para devolver
        ' el valor de la propiedad
        Get
            Return msNombre
        End Get

        ' bloque Set para asignar
        ' valor a la propiedad
        Set(ByVal Value As String)
            msNombre = Value
        End Set
    End Property

End Class
```

Código fuente 207

Cuando declaramos un procedimiento Property, debemos, al igual que en una función, tipificarlo, ya que una de sus labores consiste en la devolución de un valor.

Para devolver el valor, en el bloque Get podemos utilizar la palabra clave Return, seguida del valor de retorno, o bien la sintaxis clásica de asignar el valor al nombre de la función. Nuestra recomendación es el uso de Return por las ventajas explicadas en el tema del lenguaje.

En cuanto a la asignación de valor, el bloque Set utiliza un parámetro con el nombre Value, que contiene el valor para asignar a la propiedad.

Observe el lector que al declarar un procedimiento de este tipo, el IDE de VS.NET crea automáticamente los correspondientes bloques Get y Set, ahorrando ese trabajo al programador.

A la hora de manipular una propiedad desde el código cliente, y como ya habíamos apuntado anteriormente, la diferencia no será notoria, como muestra el Código fuente 208. La única forma de

hacer más patente el uso del procedimiento Property, consiste en ejecutar el programa utilizando el depurador; de esta manera comprobaremos como el flujo de la ejecución salta a los bloques Get y Set al manejar la variable del objeto.

```
Sub Main()
    Dim loEmpleado As New Empleado()

    ' asignar valor a una propiedad
    loEmpleado.Nombre = "Guillermo"

    ' mostrar el valor de una propiedad del objeto
    Console.WriteLine("El valor de la propiedad Nombre es: {0}", _
        loEmpleado.Nombre)
    Console.ReadLine()
End Sub
```

Código fuente 208

Dado que los procedimientos Property no son otra cosa que rutinas de código, también se les denomina *métodos de acceso y asignación* en el argot OOP.

Ventajas en el uso de propiedades

Comprobada la facilidad de los campos de clase, el lector se estará preguntando en estos momentos por qué debe utilizar propiedades, si en definitiva, su finalidad es la misma que los campos: guardar un valor en el objeto.

Existen varias y poderosas razones, por las cuales nos debemos decantar en muchas ocasiones, hacia el uso de propiedades. En los siguientes apartados haremos una descripción de ellas.

Encapsulación a través de propiedades

Una de las características de la OOP, la encapsulación, establece que el código de una clase debe permanecer, siempre que sea posible, protegido de modificaciones no controladas del exterior (código cliente). Nuestra clase debe actuar como una especie de caja negra, que expone un interfaz para su uso, pero que no debe permitir el acceso a la implementación de dicho interfaz.

Supongamos que en nuestra clase Empleado necesitamos crear un elemento para guardar el sueldo pagado, pero el importe del sueldo deberá estar entre un rango de valores en función de la categoría del empleado. Si la categoría es 1, el sueldo estará entre 1 y 200, mientras que si la categoría es 2, el sueldo podrá llegar hasta 300. Si abordamos este problema utilizando campos de clase, puede ocurrir lo que mostramos en el Código fuente 209.

```
Module General

    Sub Main()
        Dim loEmpleado As Empleado
        loEmpleado = New Empleado()
        loEmpleado.psNombre = "Juan"
        loEmpleado.piCategoria = 1
```

```
        ' atención, el sueldo para este empleado
        ' debería estar entre 1 a 200, debido a su categoría
        loEmpleado.pdbSueldo = 250
    End Sub

End Module

Public Class Empleado
    Public msNombre As String
    Public miCategoria As Integer
    Public mdbSueldo As Double
End Class
```

Código fuente 209

¿Que está sucediendo aquí?. Hemos creado un objeto empleado al que le hemos dado categoría 1, sin embargo le estamos asignando un sueldo que no corresponde a su categoría, pero se nos permite hacerlo sin ningún problema, ya que no existe un medio de control que nos lo impida.

Afrontando el problema mediante el uso de propiedades, contamos con la ventaja de escribir código de validación en los correspondientes procedimientos Property; con ello encapsulamos el código de la clase, manteniéndolo a salvo de asignaciones incoherentes. Veamos esta solución en el Código fuente 210.

```
Module General

    Sub Main()
        Dim loEmpleado As Empleado
        loEmpleado = New Empleado()

        loEmpleado.psNombre = "Pedro"
        loEmpleado.Categoria = 1

        loEmpleado.Sueldo = 250
        Console.WriteLine("Asignación incorrecta")
        Console.WriteLine("Empleado {0} - Categoria {1} - Sueldo {2}", _
            loEmpleado.psNombre, loEmpleado.Categoria, loEmpleado.Sueldo)

        loEmpleado.Sueldo = 175
        Console.WriteLine("Asignación correcta")
        Console.WriteLine("Empleado {0} - Categoria {1} - Sueldo {2}", _
            loEmpleado.psNombre, loEmpleado.Categoria, loEmpleado.Sueldo)

        Console.ReadLine()
    End Sub

End Module

Public Class Empleado
    Public psNombre As String

    ' variables de propiedad
    Private miCategoria As Integer
    Private mdbSueldo As Double

    ' procedimientos de propiedad
    Public Property Categoria() As Integer
        Get
            Return miCategoria
        End Get
```

```
        Set(ByVal Value As Integer)
            miCategoria = Value
        End Set
    End Property

    Public Property Sueldo() As Double
        Get
            Return mdbSueldo
        End Get

        ' cuando asignamos el valor a esta propiedad,
        ' ejecutamos código de validación en el bloque Set
        Set(ByVal Value As Double)
            ' si la categoría del empleado es 1...
            If miCategoria = 1 Then
                ' ...pero el sueldo supera 200
                If Value > 200 Then
                    ' mostrar un mensaje y asignar un cero
                    Console.WriteLine("La categoría no corresponde con el sueldo")
                    mdbSueldo = 0
                Else
                    ' si todo va bien, asignar el sueldo
                    mdbSueldo = Value
                End If
            End If

        End Set
    End Property

End Class
```

Código fuente 210

Propiedades de sólo lectura o sólo escritura

Se nos plantea ahora un nuevo caso para nuestra clase Empleado: debemos guardar el valor del código de cuenta bancaria del empleado en el objeto, pero sin permitir que dicha información sea accesible desde el código cliente.

Igualmente y en función de los primeros dígitos de la cuenta bancaria, necesitamos mostrar el nombre de la entidad, pero sin permitir al código cliente su modificación, ya que esta va a ser siempre una operación que debe calcular el código de la clase.

Utilizando campos de clase no es posible resolver esta situación, ya que al ser de ámbito público, permiten tanto la escritura como lectura de sus valores.

Pero si empleamos propiedades, estas nos permiten la creación de las denominadas propiedades de sólo lectura o sólo escritura, en las que utilizando las palabras clave ReadOnly y WriteOnly, conseguimos que a una determinada propiedad, sólo podamos asignarle o recuperar su valor.

Debido a esto, en una propiedad ReadOnly no podremos escribir el bloque Set, ya que no tendría sentido, puesto que no se va a utilizar. Lo mismo podemos aplicar para una propiedad WriteOnly, sólo que en esta, el bloque que no podremos codificar será Get.

Igualmente obtendremos un error del compilador, si en el código cliente intentamos asignar un valor a una propiedad ReadOnly, u obtener un valor de una propiedad WriteOnly.

Veamos a continuación, en el Código fuente 211, un ejemplo de cómo resolver el problema comentado al comienzo de este apartado.

```
Module General

    Sub Main()
        Dim loEmpleado As Empleado
        loEmpleado = New Empleado()

        loEmpleado.psNombre = "Pedro"

        ' a esta propiedad sólo podemos asignarle
        ' valor, si intentamos obtenerlo, se producirá
        ' un error
        loEmpleado.CuentaBancaria = "2222-56-7779995555"

        ' en esta línea, la propiedad EntidadBancaria sólo
        ' nos permite obtener valor, si intentamos asignarlo
        ' se producirá un error
        Console.WriteLine("La entidad del empleado {0} es {1}", _
            loEmpleado.psNombre, loEmpleado.EntidadBancaria)
        Console.ReadLine()
    End Sub

End Module

Public Class Empleado
    ' campo de clase
    Public psNombre As String

    ' variables de propiedad
    Private msCtaBancaria As String
    Private msEntidad As String

    ' variables diversas
    Private msCodigoEntidad As String

    ' esta propiedad sólo permite asignar valores,
    ' por lo que no dispone de bloque Get
    Public WriteOnly Property CuentaBancaria() As String
        Set(ByVal Value As String)
            Select Case Left(Value, 4)
                Case "1111"
                    msEntidad = "Banco Universal"
                Case "2222"
                    msEntidad = "Banco General"
                Case "3333"
                    msEntidad = "Caja Metropolitana"
                Case Else
                    msEntidad = "entidad sin catalogar"
            End Select
        End Set
    End Property

    ' esta propiedad sólo permite obtener valores,
    ' por lo que no dispone de bloque Set
    Public ReadOnly Property EntidadBancaria() As String
        Get
            Return msEntidad
        End Get
    End Property
End Class
```

Código fuente 211

Propiedades virtuales

Otra de las ventajas del uso de propiedades reside en la posibilidad de definir *propiedades virtuales*; es decir, una propiedad que no tenga una correspondencia directa con una variable de propiedad, ya que podemos crear un procedimiento Property que no esté obligatoriamente asociado con una variable.

Siguiendo con la clase Empleado, en esta ocasión creamos una propiedad para almacenar la fecha en la que el empleado ha sido incorporado a la empresa; esto no entraña ninguna novedad. Sin embargo, seguidamente necesitamos disponer de una propiedad que nos permita mostrar el nombre del mes en el que se ha dado de alta al empleado.

Podemos resolver esta cuestión creando una variable de propiedad, guardando en ella una cadena con el nombre del mes; pero si disponemos de la fecha de alta, que ya contiene el mes, nos ahorraremos ese trabajo extra creando una propiedad, en este caso de sólo lectura, en la que extraigamos el nombre del mes de la fecha de alta y lo devolvamos como resultado. Veamos como hacerlo en el Código fuente 212.

```
Module General
    Sub Main()
        Dim loEmpleado As Empleado
        loEmpleado = New Empleado()

        loEmpleado.psNombre = "Antonio"
        loEmpleado.FechaAlta = "12/6/2002"

        ' mostramos el mes de alta, que corresponde
        ' a una propiedad virtual del objeto
        Console.WriteLine("El empleado {0} se ha dado de alta en el mes de {1}", _
            loEmpleado.psNombre, loEmpleado.MesAlta)

        Console.ReadLine()
    End Sub
End Module

Public Class Empleado
    ' campo de clase
    Public psNombre As String

    ' variables de propiedad
    Private mdtFechaAlta As Date

    ' propiedad para manejar la fecha
    ' de alta del empleado
    Public Property FechaAlta() As Date
        Get
            Return mdtFechaAlta
        End Get

        Set(ByVal Value As Date)
            mdtFechaAlta = Value
        End Set
    End Property

    ' propiedad virtual
    ' en ella devolvemos el nombre del mes en el que se ha dado
    ' de alta al empleado, utilizando la variable de otra propiedad
    Public ReadOnly Property MesAlta() As String
        Get
            Return Format(mdtFechaAlta, "MMMM")
        End Get
```

```
    End Property
End Class
```

Código fuente 212

Nombres de propiedad más naturales

Cuando desde código cliente trabajamos con objetos, estos ofrecen habitualmente nombres de propiedades claros y sin notaciones.

En el caso de la clase Empleado tenemos un inconveniente a este respecto con el campo de clase correspondiente al nombre del empleado, ya que en él utilizamos convenciones de notación para facilitar el mantenimiento del código, pero por otra parte, estamos contribuyendo a dificultar la legibilidad de los miembros de la clase desde el código cliente.

Es cierto que podemos obviar las convenciones de notación en el código, pero esto, como ya comentamos en el tema sobre el lenguaje, puede hacer que la lectura del programa sea más complicada.

Como hemos comprobado también en los pasados ejemplos, si utilizamos propiedades, podemos mantener nuestras normas de notación en cuanto a las variables de la clase, sea cual sea su tipo, y ofrecer al código cliente, nombres más naturales a través de los procedimientos Property.

Por lo tanto, si en lugar de utilizar un campo de clase para el nombre del empleado, la convertimos en una propiedad, habremos ganado en claridad de cara al programador usuario de nuestra clase. Veámoslo en el Código fuente 213.

```
Module General
    Sub Main()
        Dim loEmpleado As New Empleado()

        ' al utilizar un objeto desde código cliente
        ' siempre es más sencillo manipular la
        ' propiedad Nombre, que msNombre, en cuanto
        ' a claridad del código se refiere
        loEmpleado.Nombre = "Juan"
    End Sub
End Module

Public Class Empleado
    ' antes usábamos un campo de clase...
    'Public psNombre As String   <---

    ' ...pero lo convertimos en una variable de propiedad...
    Private msNombre As String
    ' ...creando su procedimiento de propiedad correspondiente
    Public Property Nombre() As String
        Get
            Return msNombre
        End Get
        Set(ByVal Value As String)
            msNombre = Value
        End Set
    End Property
End Class
```

Código fuente 213

Propiedades predeterminadas

Una propiedad predeterminada o por defecto, es aquella que nos permite su manipulación omitiendo el nombre.

Para establecer una propiedad como predeterminada en una clase, la variable de propiedad asociada deberá ser un array, pudiendo crear sólo una propiedad predeterminada por clase.

Al declarar una propiedad por defecto, deberemos utilizar al comienzo de la sentencia de declaración, la palabra clave Default.

Para asignar y obtener valores de este tipo de propiedad, tendremos que utilizar el índice del array que internamente gestiona la propiedad. Pongamos como ejemplo el hecho de que el trabajo desempeñado por el empleado le supone realizar viajes a diferentes ciudades; para llevar un control de los viajes realizados, crearemos una nueva propiedad, que además será predeterminada. Veamos este ejemplo en el Código fuente 214.

```
Module General
    Sub Main()
        Dim loEmpleado As New Empleado()
        Dim liContador As Integer

        ' primero manejamos la propiedad predeterminada
        ' igual que una normal
        loEmpleado.Viajes(0) = "Valencia"

        ' aquí manipulamos la propiedad predeterminada
        ' sin indicar su nombre
        loEmpleado(1) = "Toledo"

        For liContador = 0 To 1
            Console.WriteLine("Visita: {0} - Ciudad: {1}", _
                liContador, loEmpleado(liContador))
        Next

        Console.ReadLine()
    End Sub
End Module

Public Class Empleado
    ' este es el array asociado a
    ' la propiedad predeterminada
    Private msViajes() As String

    ' declaración de la propiedad predeterminada
    Default Public Property Viajes(ByVal Indice As Integer) As String
        Get
            ' para devolver un valor, empleamos
            ' el número de índice pasado
            ' como parámetro
            Return msViajes(Indice)
        End Get

        Set(ByVal Value As String)
            ' para asignar un valor a la propiedad,
            ' comprobamos primero si el array está vacío

            ' comprobar si el array está vacío,
            ' al ser el array también un objeto,
            ' utilizamos el operador Is
```

```
            If msViajes Is Nothing Then
                ReDim msViajes(0)
            Else
                ' si el array ya contenía valores,
                ' añadir un nuevo elemento
                ReDim Preserve msViajes(UBound(msViajes) + 1)
            End If

            ' asignar el valor al array
            msViajes(Indice) = Value
        End Set
    End Property

End Class
```

Código fuente 214

El uso de propiedades predeterminadas proporciona una cierta comodidad a la hora de escribir el código, sin embargo, si nos acostumbramos a especificar en todo momento las propiedades en el código, ganaremos en legibilidad.

Eliminación de la palabra clave Set para asignar objetos

Al lector procedente de versiones anteriores de VB le resultará, cuanto menos sorprendente, el hecho de que ahora no se deba utilizar la palabra clave Set, para asignar un objeto a una variable. Este aspecto del lenguaje, viene motivado por las diferencias que en el tratamiento de las propiedades predeterminadas hace VB.NET respecto de las otras versiones.

En VB6, como no era obligatorio que la propiedad predeterminada de un objeto fuera un array, la única forma que tenía el lenguaje de saber si a una variable de objeto le estaban asignando una referencia de un objeto, o un valor a su propiedad predeterminada, era a través del uso de Set. Veamos el Código fuente 215.

```
' Esto es código VB6
' ==================
' La variable txtCaja contiene un objeto TextBox,
' este objeto tiene la propiedad Text como predeterminada

' estas dos líneas son equivalentes,
' asignan valores a la propiedad Text
txtCaja.Text = "coche"
txtCaja = "tren"

' Si no dispusiéramos en VB6 de Set, en la
' siguiente línea no sabríamos si se está
' intentando asignar un valor a la propiedad Text
' o un nuevo objeto a la variable
txtCaja = txtNuevaCaja   ' esto produce error en VB6

' esto es lo correcto y nos indica que
' asignamos un nuevo objeto
' a la variable
Set txtCaja = txtNuevaCaja
```

Código fuente 215

Debido a la restricción impuesta por VB.NET, que nos obliga a que las propiedades predeterminadas sean un array, el uso de Set deja de tener sentido, ya que siempre que hagamos referencia a una propiedad predeterminada tendremos que usar el índice de su array, generándose un código más claro y eliminando la incomodidad de usar Set para cada asignación de objetos a variables. Ver el Código fuente 216.

```
' esto es código VB.NET
' ======================
Dim loEmpleado As Empleado

' no utilizamos Set para asignar
' un objeto a una variable
loEmpleado = New Empleado()

' como la propiedad predeterminada es un array,
' siempre debemos trabajar con sus índices
loEmpleado.Viajes(0) = "Valencia"
' ....
' ....
' asignamos otro objeto Empleado a la variable
' sabemos claramente que es una asignación
' porque no estamos manejando el índice de la
' propiedad predeterminada del objeto
loEmpleado = loOtroEmp
```

Código fuente 216.

18

Métodos y espacios de nombre

Creación de métodos para la clase

Para crear un método en una clase debemos escribir un procedimiento de tipo Sub o Function, en función de si necesitamos devolver o no, un valor desde el método. Por este motivo, podemos deducir que un método es lo mismo que un procedimiento, siendo las diferencias existentes entre ambos tan sólo a nivel conceptual: mientras que a una rutina de código dentro de un módulo se le denomina procedimiento, si la escribimos dentro de una clase se le denomina método.

Los métodos, tal y como explicamos en los primeros apartados teóricos sobre OOP de este tema, son aquellos miembros de una clase que definen el comportamiento de los objetos, como consecuencia de las acciones que llevan a cabo al ser ejecutados. Veamos a continuación, un ejemplo concreto de creación de método.

En la clase Empleado necesitamos realizar un cálculo del día en que va a finalizar un empleado sus vacaciones; para ello precisamos conocer la fecha de comienzo y la cantidad de días que va a estar de vacaciones, por lo que escribiremos un método en nuestra clase al que llamaremos CalcularVacaciones( ); a este método le pasaremos los parámetros de la fecha de inicio y el número de días, devolviendo, al ser de tipo Function, la fecha de finalización del periodo vacacional.

```
Module General
    Sub Main()
        ' instanciar objeto Empleado
        Dim loEmpleado As Empleado
        loEmpleado = New Empleado()
```

```
        ' asignar valores a propiedades
        loEmpleado.Identificador = 78
        loEmpleado.Nombre = "Antonio"
        loEmpleado.Apellidos = "Iglesias"

        ' llamar a método
        loEmpleado.CalcularVacaciones("20/07/2002", 15)
    End Sub
End Module

Public Class Empleado
    ' variables de propiedad
    Private miID As Integer
    Private msNombre As String
    Private msApellidos As String

    ' procedimientos de propiedad
    Public Property Identificador() As Integer
        ' ......
    End Property

    Public Property Nombre() As String
        ' ......
    End Property

    Public Property Apellidos() As String
        ' ......
    End Property

    ' métodos
    Public Sub CalcularVacaciones(ByVal ldtInicio As Date, _
        ByVal liDias As Integer)
        ' en este método calculamos el periodo
        ' de vacaciones del empleado,
        ' mostrando los resultados en consola
        Dim ldtFinal As Date
        ldtFinal = DateAdd(DateInterval.Day, liDias, ldtInicio)
        Console.WriteLine("Empleado {0} - {1} {2}", _
            Identificador, Nombre, Apellidos)
        Console.WriteLine("Vacaciones desde {0} hasta {1}", _
            Format(ldtInicio, "dd/MMM/yy"), _
            Format(ldtFinal, "d/MMMM/yyyy"))
        Console.ReadLine()
    End Sub

End Class
```

Código fuente 217

Llegados a este punto, hemos completado todos los pasos elementales en cuanto a la creación de una clase. Retomemos pues, el caso del ejemplo expuesto al comienzo del tema, de manera que si sustituimos el enfoque procedural de los procesos del empleado, por uno orientado a objeto, la clase Empleado resultante podría ser algo similar a la mostrada en el Código fuente 218.

```
Public Class Empleado
    ' variables de propiedad
    Private miID As Integer
    Private msNombre As String
    Private msApellidos As String
    Private msDNI As String
    Private mdtFechaAlta As Date
    Private mdbSueldo As Double
```

```
    Private mdtInicioVacaciones As Date
    Private miDiasVacaciones As Integer

    ' procedimientos de propiedad
    Public Property Identificador() As Integer
        Get
            Return miID
        End Get
        Set(ByVal Value As Integer)
            miID = Value
        End Set
    End Property

    Public Property Nombre() As String
        Get
            Return msNombre
        End Get
        Set(ByVal Value As String)
            msNombre = Value
        End Set
    End Property

    Public Property Apellidos() As String
        Get
            Return msApellidos
        End Get
        Set(ByVal Value As String)
            msApellidos = Value
        End Set
    End Property

    Public Property DNI() As String
        Get
            Return msDNI
        End Get
        Set(ByVal Value As String)
            msDNI = Value
        End Set
    End Property

    Public Property FechaAlta() As Date
        Get
            Return mdtFechaAlta
        End Get
        Set(ByVal Value As Date)
            mdtFechaAlta = Value
        End Set
    End Property

    Public Property Sueldo() As Double
        Get
            Return mdbSueldo
        End Get
        Set(ByVal Value As Double)
            mdbSueldo = Value
        End Set
    End Property

    Public Property InicioVacaciones() As Date
        Get
            Return mdtInicioVacaciones
        End Get
        Set(ByVal Value As Date)
            mdtInicioVacaciones = Value
        End Set
    End Property
```

```
    Public Property DiasVacaciones() As Integer
        Get
            Return miDiasVacaciones
        End Get
        Set(ByVal Value As Integer)
            miDiasVacaciones = Value
        End Set
    End Property

    Public Sub CalcularVacaciones()
        ' en este método calculamos el periodo
        ' de vacaciones del empleado,
        ' mostrando los resultados en consola
        Dim ldtFinal As Date
        ldtFinal = DateAdd(DateInterval.Day, miDiasVacaciones, mdtInicioVacaciones)
        Console.WriteLine("Empleado {0} - {1} {2}", _
            miID, msNombre, msApellidos)
        Console.WriteLine("Vacaciones desde {0} hasta {1}", _
            Format(mdtInicioVacaciones, "dd/MMM/yy"), _
            Format(ldtFinal, "d/MMMM/yyyy"))
        Console.ReadLine()
    End Sub

    Public Sub CrearEmpleado()
        ' crear un nuevo registro en la base de datos,
        ' grabar los valores que debe haber
        ' en las propiedades
        ' .......
        Console.WriteLine("Se ha grabado el empleado: {0} - {1} {2}", _
            miID, msNombre, msApellidos)
        Console.ReadLine()
    End Sub

    Public Sub TransfNomina()
        ' realizamos la transferencia de nómina
        ' a un empleado, utilizando su identificador
        ' ......
        ' obtener los datos del empleado de la base de datos
        ' y traspasarlos a las propiedades
        ' ......
        ' visualizamos el resultado
        Console.WriteLine("Pago de nómina")
        Console.WriteLine("Empleado: {0} {1}", msNombre, msApellidos)
        Console.WriteLine("Ingresado: {0}", mdbSueldo)
        Console.ReadLine()
    End Sub

    Public Sub MostrarEmpleado()
        ' buscar la información del empleado en la base de datos
        ' usando el valor de la propiedad identificador
        Dim lsDatosEmpleado As String
        ' ......
        Console.WriteLine("El empleado seleccionado es: {0}", msNombre,
msApellidos)
        Console.ReadLine()
    End Sub
End Class
```

Código fuente 218

Gracias a que la codificación de todos los procesos reside ahora en la clase, el código cliente que tenga que tratar ahora con el empleado, quedaría simplificado y reducido a lo que se muestra en el Código fuente 219.

```
Module General
    Sub Main()
        ' instanciar objeto
        Dim loEmpleado As New Empleado()
        loEmpleado.Identificador = 850
        loEmpleado.Nombre = "Juan"
        loEmpleado.Apellidos = "García"
        ' asignar resto de propiedades
        ' .....
        ' .....
        ' llamar a sus métodos
        loEmpleado.MostrarEmpleado()
        loEmpleado.TransfNomina()
        ' .....
        ' .....
    End Sub
End Module
```

Código fuente 219

Hemos podido comprobar lo sencillo e intuitivo que resulta trabajar con determinados procesos a través de técnicas OOP, ya que una vez codificada la clase, tan sólo hemos de hacer uso de ella instanciando el correspondiente objeto; con la ventaja añadida de que podemos tener varios objetos de la misma clase funcionando al mismo tiempo.

¿Cuándo crear una propiedad y cuándo un método?

Debido a que una propiedad, a través de su procedimiento Property asociado, puede ejecutar código, la decisión de escribir cierta operación en una clase empleando una propiedad o un método, es en algunas ocasiones difícil, ya que existen procesos que pueden ser resueltos utilizando ambos modos.

Sin ir más lejos, el método CalcularVacaciones( ), visto en el ejemplo del apartado anterior, bien podría haberse resuelto a través de una propiedad, como muestra el Código fuente 220. En él hemos incluido sólo las partes modificadas de la clase Empleado para solucionar este problema.

```
Module General
    Sub Main()
        ' crear objeto Empleado
        Dim loEmpleado As Empleado
        loEmpleado = New Empleado()

        ' asignar valores a propiedades
        loEmpleado.Identificador = 78
        loEmpleado.Nombre = "Antonio"
        loEmpleado.Apellidos = "Iglesias"

        ' esta sería la parte nueva en el código cliente:
        ' asignar la fecha de inicio y número de días
        ' de vacaciones, y obtener de la propiedad FinVacaciones
        ' el día en que termina las vacaciones, aplicando
        ' en este caso, un formato a la fecha obtenida
        loEmpleado.InicioVacaciones = "20/07/2002"
        loEmpleado.DiasVacaciones = 15
        Console.WriteLine("El empleado {0} - {1} {2}" & ControlChars.CrLf & _
            "finaliza sus vacaciones el día {3}", _
            loEmpleado.Identificador, loEmpleado.Nombre, _
            loEmpleado.Apellidos, _
```

```
            Format(loEmpleado.FinVacaciones, "d-MMMM-yy"))

        Console.ReadLine()
    End Sub
End Module

Public Class Empleado
    ' en esta clase creamos 3 propiedades nuevas,
    ' para guardar la fecha de inicio de vacaciones,
    ' los días y la fecha de fin

    ' variables de propiedad
    ' .....
    ' .....
    Private mdtInicioVacaciones As Date
    Private mdtFinVacaciones As Date
    Private miDiasVacaciones As Integer

    ' procedimientos de propiedad
    ' .....
    ' .....
    Public Property InicioVacaciones() As Date
        Get
            Return mdtInicioVacaciones
        End Get
        Set(ByVal Value As Date)
            mdtInicioVacaciones = Value
        End Set
    End Property

    Public Property DiasVacaciones() As Integer
        Get
            Return miDiasVacaciones
        End Get
        Set(ByVal Value As Integer)
            miDiasVacaciones = Value
        End Set
    End Property

    ' en este procedimiento de propiedad
    ' realizamos el cálculo para obtener
    ' la fecha de fin de vacaciones y
    ' devolvemos dicha fecha al código cliente
    Public ReadOnly Property FinVacaciones() As Date
        Get
            ' calcular la fecha de fin de vacaciones
            Return DateAdd(DateInterval.Day, _
                DiasVacaciones, InicioVacaciones)
        End Get
    End Property
    ' .....
    ' .....
End Class
```

Código fuente 220

Queda por lo tanto, en manos del programador, determinar el criterio por el cuál un proceso se resolverá mediante una propiedad o un método, debiendo ser una decisión flexible y no basarse en unas normas rígidas.

La estructura With...End With

Este elemento del lenguaje nos facilita la escritura de código cuando hacemos referencia a los miembros de un objeto, ya que nos ahorra tener que escribir el nombre del objeto, siendo preciso indicar sólo sus miembros. La sintaxis de esta estructura se muestra en el Código fuente 221.

```
With Objeto
       .Campo
       .Propiedad
       .Método()
End UIT
```

Código fuente 221

Pongamos como ejemplo, que hemos creado una clase con el nombre Empleado que tiene las propiedades Nombre, Apellidos, y el método MostrarDatos( ), para manipular un objeto de esta clase mediante With, lo haríamos como muestra el Código fuente 222.

```
Dim loEmp As Empleado = New Empleado()
With loEmp
    .Nombre = "Ana"
    .Apellidos = "Naranjo"
    .MostrarDatos()
End UIT
```

Código fuente 222

Podemos también anidar esta estructura, con el fin de manipular más de un objeto, veamos el Código fuente 223.

```
Dim loEmp As Empleado = New Empleado()
Dim loUsu As New Usuario()

With loEmp
    .Nombre = "Ana"
    .Apellidos = "Naranjo"
    .MostrarDatos()

    With loUsu
        .AsignarNombre("Jacinto")
    End With
End With
```

Código fuente 223

Resultados distintos en objetos de la misma clase

La capacidad de instanciar al mismo tiempo varios objetos de la misma clase nos lleva a una interesante cuestión: la obtención de resultados distintos a partir de objetos del mismo tipo, cuando

dichos objetos tienen datos diferentes en sus propiedades, ya que aunque el código ejecutado es el mismo, los valores de sus propiedades difieren entre sí.

Un ejemplo ilustrativo de esta situación sería la creación de dos objetos de la clase Empleado, en los que cada uno tuviera fechas de comienzo y días de vacaciones distintos. En este caso, aunque los objetos son del mismo tipo, la finalización de sus vacaciones sería distinta. Ver el Código fuente 224.

```
Dim loEmpleado1 As Empleado
Dim loEmpleado2 As Empleado

loEmpleado1 = New Empleado()
loEmpleado2 = New Empleado()

loEmpleado1.InicioVacaciones = "25/07/2002"
loEmpleado1.DiasVacaciones = 20

loEmpleado2.InicioVacaciones = "25/07/2002"
loEmpleado2.DiasVacaciones = 30

' los dos objetos son de la clase Empleado,
' pero el resultado en este caso al usar la
' propiedad FinVacaciones no será igual
' para estos objetos, dados los diferentes
' valores de algunas de sus propiedades
```

Código fuente 224

Uso de Me y MyClass para llamar a los miembros de la propia clase

Cuando desde el código de una clase queramos hacer referencia a un miembro de la propia clase: campo, propiedad o método, podemos utilizar las palabras clave Me y MyClass para manipular dicho elemento. Veamos el Código fuente 225.

```
Module Module1
    Sub Main()
        Dim loEmp As New Empleado()
        loEmp.piID = 980
        loEmp.Nombre = "Almudena Bosque"
        loEmp.VerDatos()

        Console.ReadLine()
    End Sub

End Module

Public Class Empleado
    Public piID As Integer
    Private msNombre As String

    Public Property Nombre() As String
        Get
            Return msNombre
        End Get
        Set(ByVal Value As String)
            msNombre = Value
```

```
        End Set
    End Property

    Public Sub VerDatos()
        ' utilizamos Me y MyClass en este método para tomar
        ' el valor de la variable piID que está en esta
        ' misma clase, y para llamar al método NombreMay()
        ' que también está en la clase
        Console.WriteLine("Código del empleado: {0}", Me.piID)
        Console.WriteLine("Nombre del empleado: {0}", MyClass.NombreMay())
    End Sub

    Public Function NombreMay() As String
        Return UCase(msNombre)
    End Function
End Class
```

Código fuente 225

Como acabamos de ver, desde el código de la propia clase Empleado llamamos a una variable y a un método situados también en la clase, anteponiendo la palabra clave Me y MyClass respectivamente.

Aunque el uso de estas palabras clave no es obligatorio, ya que el compilador reconoce el miembro que queremos ejecutar, sí es recomendable ya que facilita la lectura de nuestro código.

Sobrecarga de métodos o polimorfismo, en una misma clase

La *sobrecarga de métodos*, tal y como ya vimos en el tema sobre el lenguaje, apartado *Sobrecarga de procedimientos*, es una técnica que consiste en crear varios métodos con idéntico nombre dentro de la misma clase, distinguiéndose entre sí por su lista de parámetros.

Para declarar un método como sobrecargado, debemos utilizar la palabra clave Overloads después del modificador de ámbito. Podemos sobrecargar métodos de tipo Sub y Function.

Una situación que podría requerir la sobrecarga de métodos sería la siguiente: la clase Empleado necesita manejar en diferentes formas, la información que referente al sueldo existe sobre el empleado. Por tal motivo, vamos a crear tres métodos con el nombre Sueldo( ), que variarán en su firma, o protocolo de llamada, y realizarán diferentes tareas, pero todas ellas relacionadas con el sueldo del empleado. Veamos el Código fuente 226.

```
Module General
    Sub Main()
        Dim loEmpleado As New Empleado()
        Dim ldbResultadoIncent As Double
        loEmpleado.Salario = 1020.82

        'llamada al primer método sobrecargado
        loEmpleado.Sueldo()

        'llamada al segundo método sobrecargado
        Console.WriteLine("El sueldo se transferirá el día {0}", _
            loEmpleado.Sueldo(29))

        'llamada al tercer método sobrecargado
        ldbResultadoIncent = loEmpleado.Sueldo(50.75, "Extras")
```

```
        Console.WriteLine("El incentivo a pagar será {0}", ldbResultadoIncent)

        Console.ReadLine()
    End Sub
End Module

Public Class Empleado
    Private mdbSalario As Double

    Public Property Salario() As Double
        Get
            Return mdbSalario
        End Get
        Set(ByVal Value As Double)
            mdbSalario = Value
        End Set
    End Property

    ' métodos sobrecargados
    Public Overloads Sub Sueldo()
        ' aquí mostramos en consola el importe del sueldo formateado
        Console.WriteLine("El sueldo es {0}", Format(Me.Salario, "#,#.##"))
        Console.ReadLine()
    End Sub

    Public Overloads Function Sueldo(ByVal liDia As Integer) As String
        ' aquí mostramos la fecha del mes actual
        ' en la que se realizará la transferencia
        ' del sueldo al banco del empleado
        Dim ldtFechaActual As Date
        Dim lsFechaCobro As String
        ldtFechaActual = Now()

        lsFechaCobro = CStr(liDia) & "/" & _
            CStr(Month(ldtFechaActual)) & "/" & _
            CStr(Year(ldtFechaActual))

        Return lsFechaCobro
    End Function

    Public Overloads Function Sueldo(ByVal ldbImporteIncentivo As Double, _
        ByVal lsTipoIncentivo As String) As Double
        ' aquí calculamos la cantidad de incentivo
        ' que se añadirá al sueldo del empleado,
        ' en función del tipo de incentivo
        Dim ldbIncentivo As Double

        ' según el tipo de incentivo,
        ' se descuenta un importe
        ' de la cantidad del incentivo
        Select Case lsTipoIncentivo
            Case "Viajes"
                ldbIncentivo = ldbImporteIncentivo - 30
            Case "Extras"
                ldbIncentivo = ldbImporteIncentivo - 15
        End Select

        Return ldbIncentivo

    End Function

End Class
```

Código fuente 226

Vemos pues, cómo a través de la sobrecarga conseguimos también polimorfismo para una clase, ya que el mismo nombre de método, en función de los parámetros pasados, actuará de diferente forma.

A pesar de haber indicado que la palabra clave Overloads nos permite sobrecargar los métodos con nombres iguales en la clase, realmente no sería necesario su uso, ya que el compilador detecta la diferencia entre dichos métodos a través de su lista de parámetros. Sin embargo se recomienda el uso de esta palabra clave por motivos de legibilidad del código, de forma que nos ayude a reconocer más rápidamente los métodos sobrecargados.

Cuando realmente necesitaremos emplear Overloads será al sobrecargar un método en una clase derivada, aspecto este que se explicará en un próximo apartado.

Enlace (binding) de variables a referencias de objetos

El enlace, también denominado binding, es un proceso que determina cuándo va a ser efectuada la localización de los miembros de un objeto, por parte de la variable que va a manipular dicho objeto. Existen dos tipos de enlace, los cuales describimos a continuación.

Enlace temprano

También conocido como *early binding* o *static binding*, este enlace establece que las referencias entre la variable y el objeto que contiene van a ser resueltas en tiempo de compilación.

El enlace temprano se realiza en el momento de declarar la variable, asignándole a esta el tipo de objeto con el que va a trabajar. Con ello conseguimos un mejor rendimiento del programa, puesto que el código generado, al conocer de forma precisa qué propiedades y métodos debe usar, se ejecutará de modo más veloz. En nuestros anteriores ejemplos con la clase Empleado, al declarar una variable de dicha clase, tenemos desde ese momento, acceso directo a todos sus miembros. Ver Figura 194.

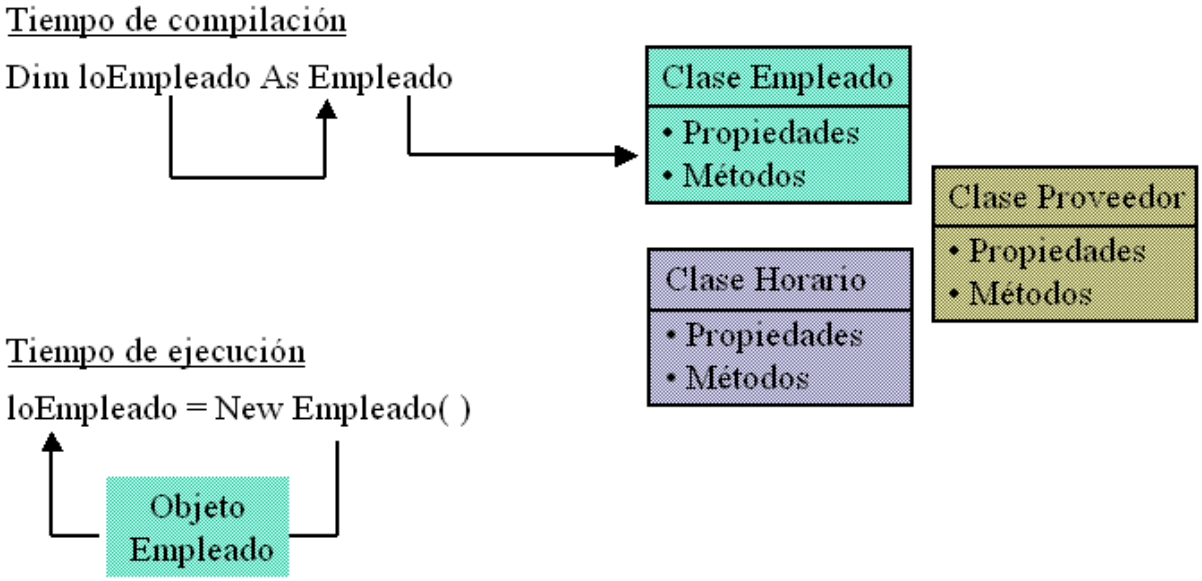


Figura 194. Esquema de funcionamiento del enlace temprano de objetos.

Además, y como habrá podido comprobar hasta ahora el lector, la escritura de código mediante enlace temprano también se facilita, ya que en todo momento, los asistentes del IDE muestran las listas de miembros disponibles para el objeto que estemos codificando. Ver Figura 195

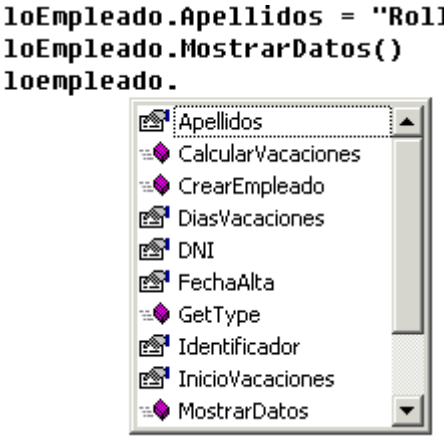


Figura 195. Lista de miembros de un objeto en el editor de código.

El enlace temprano, debido a su mejor rendimiento, es el tipo de enlace utilizado por defecto dentro del CLR.

Enlace tardío

También conocido como *late binding* o *dynamic binding*, este enlace establece que las referencias entre la variable y el objeto que contiene van a ser resueltas en tiempo de ejecución.

El principal inconveniente en este tipo de enlace radica en que el código generado será más lento, ya que desconoce con qué miembros de objeto tendrá que trabajar, debiendo averiguar esta información durante la ejecución del programa. Adicionalmente, el trabajo del programador será también mayor, ya que tendrá que conocer con antelación, la lista de miembros o interfaz que implementa el objeto.

Como ventaja nos aporta una mayor flexibilidad, ya que con la misma variable podemos manipular objetos de distinto tipo. Para ello, tendremos que tipificar la variable como Object. Ver Figura 196.

Por ejemplo, si aparte de nuestra conocida clase Empleado, escribimos otra nueva llamada Proveedor, con algunos aspectos similares, como las propiedades Nombre, Apellidos, el método MostrarDatos( ), etc., podremos utilizar la misma variable para manipular cada uno de los objetos que instanciemos de estas clases; evidentemente, tendremos que asignar el objeto pertinente a la variable antes de poder manejarlo.

Vamos incluso a crear otra clase más, llamada Horario, con un método que devuelva la hora actual del sistema, y ejecutaremos dicho método asignando un objeto de esta clase a la misma variable utilizada para manejar los objetos Empleado y Proveedor. Veamos todo en el Código fuente 227.

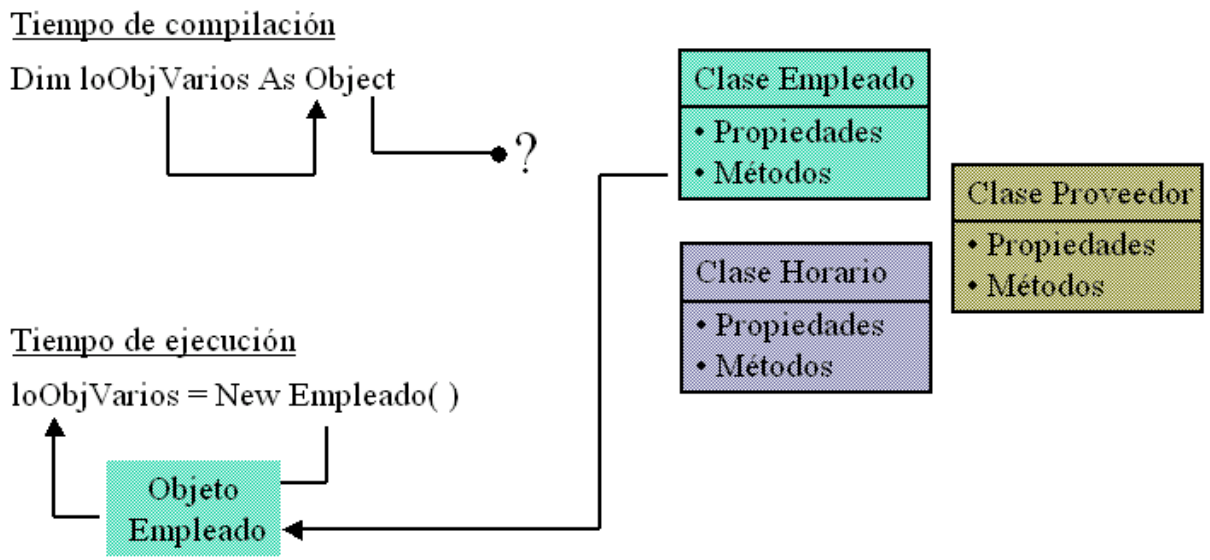


Figura 196. Esquema de funcionamiento del enlace tardío de objetos.

```
Module General
    Sub Main()
        ' tipificamos como Object,
        ' por lo que obtendremos enlace tardío
        Dim loVariosObj As Object

        ' instanciamos un objeto de Empleado
        loVariosObj = New Empleado()
        loVariosObj.Nombre = "Juan"
        loVariosObj.Apellidos = "Rollo"
        loVariosObj.MostrarDatos()

        ' instanciamos un objeto de Proveedor
        loVariosObj = New Proveedor()
        loVariosObj.Nombre = "Alicia"
        loVariosObj.Apellidos = "Cañaveral"
        loVariosObj.MostrarDatos()

        ' instanciamos un objeto de Horario
        loVariosObj = New Horario()
        loVariosObj.HoraActual()
    End Sub
End Module

Public Class Empleado
    Private msNombre As String
    Private msApellidos As String

    Public Property Nombre() As String
        Get
            Return msNombre
        End Get
        Set(ByVal Value As String)
            msNombre = Value
        End Set
    End Property

    Public Property Apellidos() As String
        Get
            Return msApellidos
        End Get
        Set(ByVal Value As String)
```

```
            msApellidos = Value
        End Set
    End Property

    Public Sub MostrarDatos()
        Console.WriteLine("El empleado seleccionado es: {0} {1}", _
            msNombre, msApellidos)
        Console.ReadLine()
    End Sub
End Class

Public Class Proveedor
    Private msNombre As String
    Private msApellidos As String

    Public Property Nombre() As String
        Get
            Return msNombre
        End Get
        Set(ByVal Value As String)
            msNombre = Value
        End Set
    End Property

    Public Property Apellidos() As String
        Get
            Return msApellidos
        End Get
        Set(ByVal Value As String)
            msApellidos = Value
        End Set
    End Property

    Public Sub MostrarDatos()
        Console.WriteLine("El proveedor actual es: {0} {1}", _
            msNombre, msApellidos)
        Console.ReadLine()
    End Sub
End Class

Public Class Horario
    Public Sub HoraActual()
        Console.WriteLine("Hora del sistema: {0}", Format(Now(), "HH:mm"))
        Console.ReadLine()
    End Sub
End Class
```

Código fuente 227

Otro de los factores indicativos de que se está produciendo enlace tardío reside en que si depuramos este código línea a línea, el depurador no saltará al código de los miembros de los objetos, ya que durante la compilación no ha podido localizar dónde se encuentra su implementación.

De cara a próximos apartados referentes a la herencia, tengamos en cuenta la siguiente regla respecto a los tipos de enlace.

El enlace temprano se basa en el tipo de la referencia o clase establecida al declarar la variable, mientras que el enlace tardío se basa en el tipo del propio objeto asignado a la variable, sin tener en cuenta la clase con que haya sido declarada la variable.

Espacios de nombres (namespaces)

Un espacio de nombres es un contenedor lógico de código, que nos permite organizar de un modo más óptimo, las clases dentro de un proyecto o ensamblado. Para una mayor información a nivel conceptual sobre este elemento de la plataforma .NET, consulte el lector el tema sobre el lenguaje, apartado *Organización del proyecto en ficheros y módulos de código*; y también el tema sobre .NET Framework, apartado Namespaces.

En el presente apartado, dedicaremos nuestros esfuerzos al trabajo con los espacios de nombres desde su vertiente práctica, es decir, veremos cómo se utilizan los espacios de nombres en un proyecto para agrupar funcionalmente las clases que contiene.

Para ello se acompaña el proyecto de ejemplo NamespacePruebas, al que podemos acceder haciendo clic aquí.

Cada vez que creamos un nuevo proyecto en VB.NET, se crea un espacio de nombres a nivel del ensamblado, con su mismo nombre, y que engloba a todos los tipos o clases que vayamos creando. Este espacio de nombres recibe la denominación de *espacio de nombres raíz*, y podemos verlo abriendo la ventana de propiedades del proyecto. Ver Figura 197.

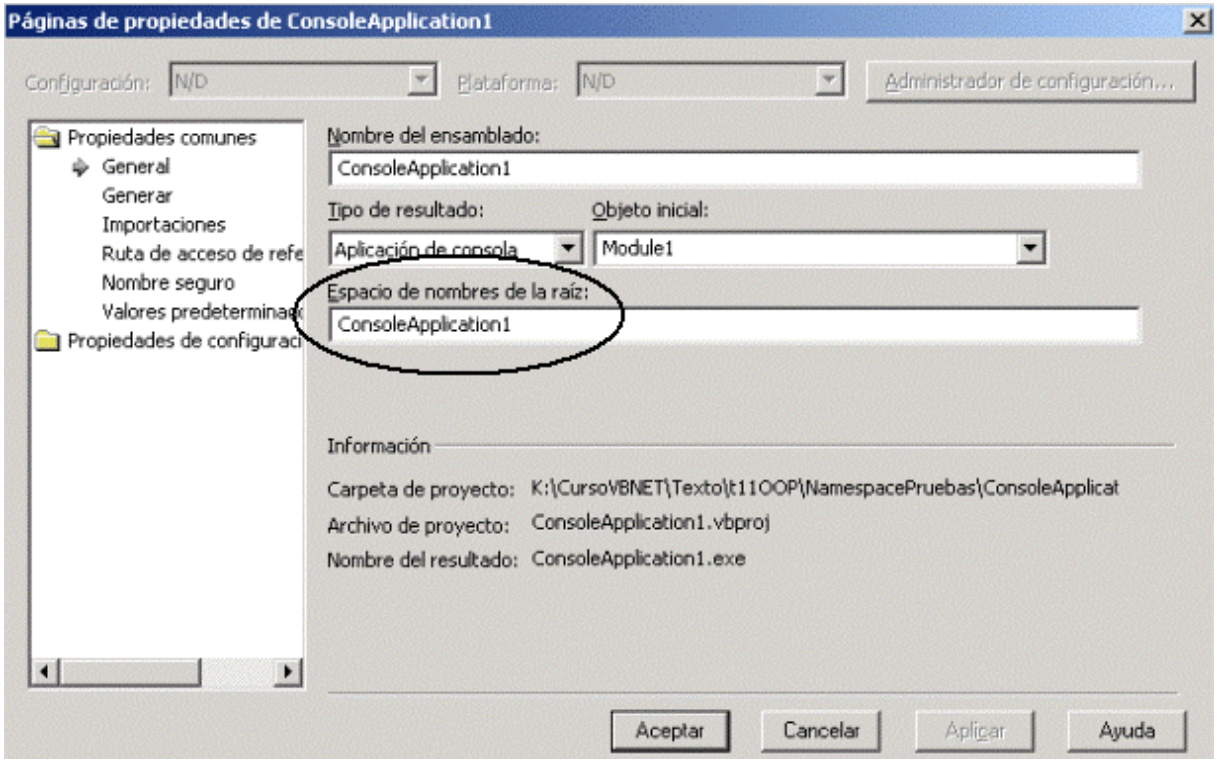


Figura 197. Nombre del espacio de nombres raíz en las propiedades del proyecto / ensamblado.

Como muestra la imagen, tanto el ensamblado como su espacio de nombres tienen como nombre ConsoleApplication1, por lo que todas las clases que escribamos dentro de este proyecto estarán dentro de dicho espacio de nombres.

Vamos a ir construyendo progresivamente un ejemplo, para ver las variantes de uso de clases en función del espacio de nombres en el que estén contenidas. Crearemos para ello una nueva aplicación

de consola, y en el fichero de código que incluye por defecto, además del módulo Module1 ya incluido al crearse el proyecto, escribiremos la clase Factura, ver Código fuente 228.

```
Module Module1
    Sub Main()
        ' como la clase Factura se encuentra
        ' en el espacio de nombres raíz,
        ' instanciamos normalmente
        Dim loFac As New Factura()
        loFac.piID = 5
        loFac.piImporte = 200
        loFac.Datos()
        Console.ReadLine()
    End Sub
End Module

' clase Factura
' esta clase se encuentra dentro
' del espacio de nombres raíz del ensamblado
Public Class Factura
    Public piID As Integer
    Public piImporte As Integer

    Public Sub Datos()
        Console.WriteLine("La factura {0}, tiene un importe de {1}", _
            Me.piID, Me.piImporte)
    End Sub
End Class
```

Código fuente 228

Seguidamente, y en el mismo fichero de código, creamos la clase Empleado, pero la incluimos en el espacio de nombres Personal. Para crear un espacio de nombres en el código de la aplicación debemos utilizar las palabras clave Namespace...End Namespace. Ver Código fuente 229.

```
' clase Empleado
' esta clase se encuentra dentro
' del espacio de nombres raíz del ensamblado,
' y a su vez, dentro del espacio de
' nombres Personal
Namespace Personal
    Public Class Empleado
        Public psID As Integer

        Public Sub MostrarDatos()
            Console.WriteLine("Identificador del empleado: {0}", Me.psID)
            Console.ReadLine()
        End Sub

    End Class
End Namespace
```

Código fuente 229

Debido a que hemos creado una clase dentro de un nuevo espacio de nombres definido en el código, dicho espacio de nombres queda anidado dentro del espacio de nombres raíz del ensamblado. Para instanciar objetos de una clase escrita en un espacio de nombres de esta forma, en primer lugar,

debemos importar dicho espacio de nombres en la cabecera del fichero de código, utilizando la palabra clave Imports, como se muestra en el Código fuente 230.

```
' debemos importar el espacio de nombres
' o no podremos instanciar objetos de las
' clases que contiene
Imports ConsoleApplication1.Personal

Module Module1
    Sub Main()
        ' como hemos importado el espacio de nombres Personal
        ' podemos instanciar un objeto de su clase Empleado
        Dim loEmp As Empleado
        loEmp = New Empleado()
        loEmp.piID = 5
        loEmp.MostrarDatos()

        Console.ReadLine()
    End Sub
End Module
```

Código fuente 230

Si no utilizamos Imports, también podemos instanciar objetos de clases halladas en espacios de nombres distintos, utilizando en este caso la sintaxis calificada, es decir, escribimos en primer lugar el espacio de nombres, un punto y la clase. El inconveniente de esta forma de codificación, reside en que cada vez que declaremos e instanciemos un objeto tenemos que emplear esta sintaxis calificada, por lo cuál, es mucho más cómodo importar el espacio de nombres al comienzo del fichero. Ver Código fuente 231.

```
Dim loEmp As Personal.Empleado
loEmp = New Personal.Empleado()
```

Código fuente 231

Finalmente, vamos a agregar una nueva clase al proyecto, a la que daremos el nombre GESTION.VB. Sin embargo no utilizaremos la clase que crea por defecto, borraremos todo el código de ese fichero y escribiremos dos nuevas clases en él: Cuenta y Balance, que además, estarán contenidas en el espacio de nombres Contabilidad. De esta forma queda demostrado como podemos organizar nuestro código, además de en clases, en espacios de nombre que contengan clases con funcionalidades similares. Ver Código fuente 232.

```
Namespace Contabilidad
    Public Class Cuenta
        Public piCodigo As Integer

        Public Function Obtener() As Integer
            Return Me.piCodigo
        End Function

    End Class

    Public Class Balance
        Public psDescripcion As String
```

```
        Public Sub MostrarDescrip()
            Console.WriteLine("La descripción del balance es: {0}",
Me.psDescripcion)
            Console.ReadLine()
        End Sub
    End Class
End Namespace
```

Código fuente 232

El modo de instanciar, desde Main( ), objetos de las clases del espacio de nombres Contabilidad, es exactamente el mismo que hemos descrito para el espacio de nombres Personal: bien importamos el espacio de nombres, o empleamos los nombres calificados. Veamos el Código fuente 233.

```
Imports ConsoleApplication1.Contabilidad

Module Module1
    Sub Main()
        ' instanciamos con sintaxis calificada
        Dim loCuen As New Contabilidad.Cuenta()
        Dim liDatoCuenta As Integer
        loCuen.piCodigo = 158
        liDatoCuenta = loCuen.Obtener()

        ' al haber importado el espacio de nombres
        ' podemos instanciar usando el nombre
        ' de la clase directamente
        Dim loBal As Balance
        loBal = New Balance()
        loBal.psDescripcion = "Resultado trimestral"
        loBal.MostrarDescrip()

        Console.ReadLine()
    End Sub
End Module
```

Código fuente 233

Una cualidad muy de agradecer cuando escribimos clases dentro de espacios de nombre, reside en que podemos tener las clases de un mismo espacio de nombres diseminadas por todo el código de la aplicación. Por ejemplo, en el módulo Module1 hemos definido el espacio de nombres Personal, y creado en su interior la clase Empleado; pues bien, si añadimos otra clase al proyecto, podemos incluir también esta clase en el espacio de nombres Personal, a pesar de que dicho código se encuentre en otro fichero distinto. Ver Código fuente 234.

```
Namespace Personal
    Public Class Proveedor
        Public psID As Integer
        Public psDescrip As String


        Public Sub MuestraProv()
            Console.WriteLine("El proveedor tiene el código" & _
                " {0} y la descripción {1}", Me.psID, Me.psDescrip)
        End Sub
    End Class
```

```
End Namespace
```

Código fuente 234

Cuando importemos el espacio de nombres Personal, todas las clases que contiene pasarán a estar disponibles, con independencia del fichero de código que las contenga.

Acceso a espacios de nombre de otros ensamblados

Al igual que accedemos a las clases a través de los espacios de nombre del ensamblado en el proyecto actual, podemos acceder a las clases residentes en otros ensamblados, y en sus correspondientes espacios de nombre. Los pasos necesarios se describen a continuación.

El ejemplo de este apartado, NamespOtrosEnsam está disponible haciendo clic aquí.

En primer lugar crearemos un proyecto de tipo consola al que daremos el nombre AppVariosProy; en él, ya disponemos del correspondiente Main( ), que codificaremos más tarde.

A continuación debemos crear otro proyecto, pero agregándolo a la solución existente, es decir, seleccionaremos el menú del IDE *Archivo + Agregar proyecto + Nuevo proyecto*. Este nuevo proyecto debe ser del tipo *Biblioteca de clases*, cuyo resultado es un fichero con extensión .DLL; le asignaremos el nombre Adicional. Ver Figura 198.

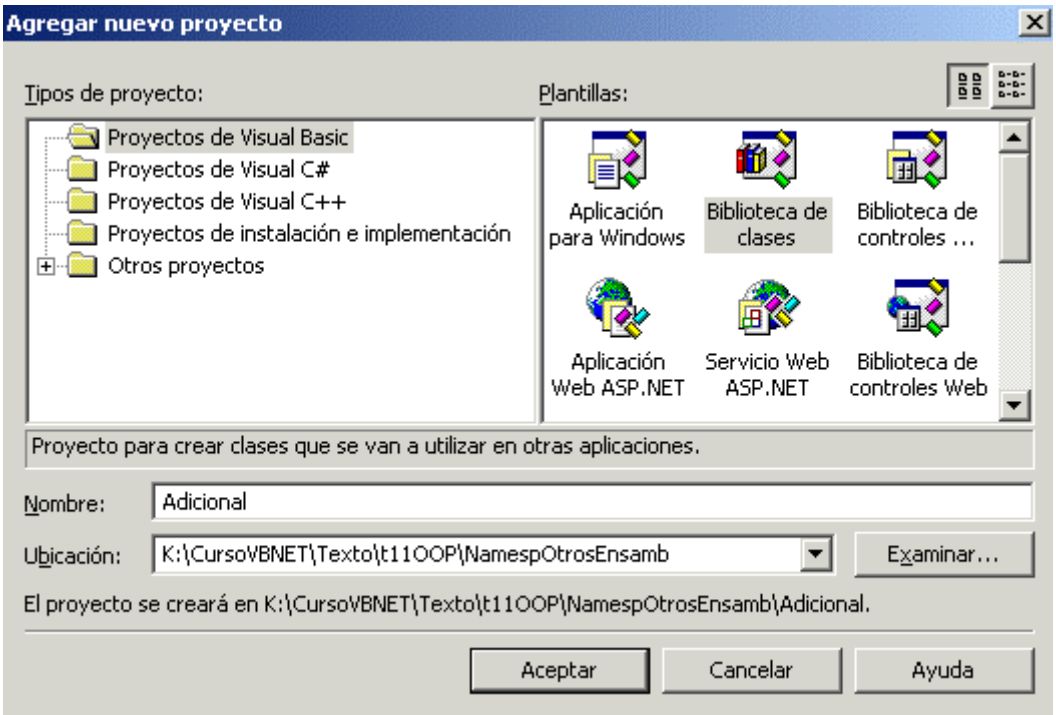


Figura 198. Agregar a la solución un proyecto de biblioteca de clases.

Al agregar un proyecto de este tipo a la solución, la ventana Explorador de soluciones muestra ambos proyectos, remarcando en negrita el nombre del proyecto AppVariosProy como el proyecto de inicio, puesto que el nuevo que acabamos de agregar, sólo contendrá clases que serán utilizadas por otros ensamblados.

Un proyecto de biblioteca de clases añade por defecto una clase con el nombre Class1, cuyo código eliminaremos y lo sustituiremos por el mostrado en el Código fuente 235, que como podrá comprobar el lector, se trata de dos clases, con la particularidad de que una de ellas está a su vez, contenida en un espacio de nombres.

```
Public Class Factura
    Public piID As Integer
    Public piImporte As Integer
    Public Sub Datos()
        Console.WriteLine("La factura {0}, tiene un importe de {1}", _
            Me.piID, Me.piImporte)
    End Sub
End Class
Namespace Correo
    Public Class Mensaje
        Public psTexto As String
        Public pdtFecha As Date
        Public Sub Visualizar()
            Console.WriteLine("Atención, mensaje: {0}, de fecha: {1}", _
                Me.psTexto, Me.pdtFecha)
        End Sub
    End Class
End Namespace
```

Código fuente 235

Sin embargo, todavía no podemos hacer uso de estas clases desde nuestra aplicación de consola, para lograr que las clases del proyecto Adicional sean visibles desde el proyecto AppVariosProy, debemos hacer clic sobre este último en el Explorador de soluciones, y a continuación seleccionar el menú *Proyecto + Agregar referencia*; en el cuadro de diálogo que aparece seguidamente, haremos clic en la pestaña Proyectos, seleccionando el único proyecto que muestra la lista, y que se trata de la biblioteca de clases que hemos añadido a la solución. Ver Figura 199.

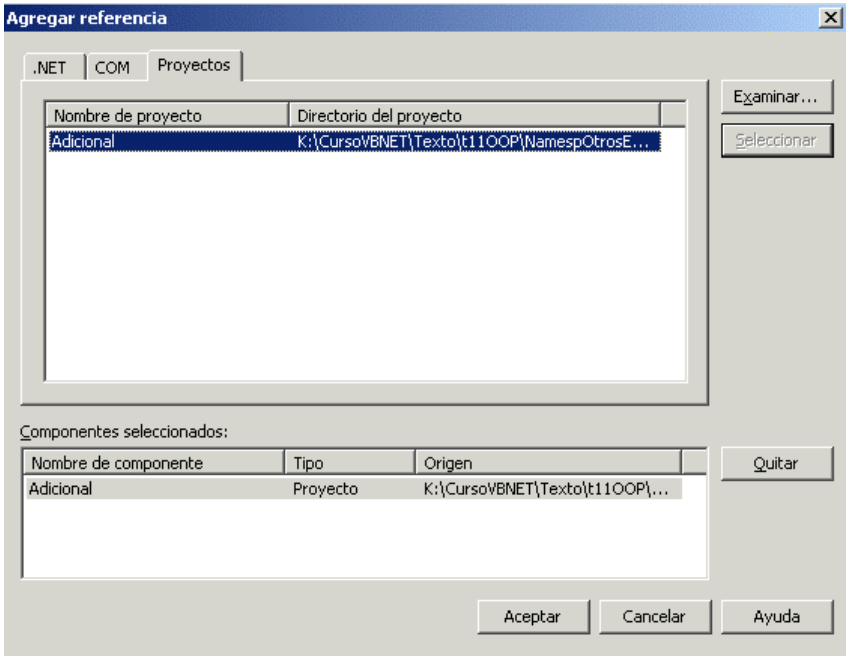


Figura 199. Agregar referencia de una biblioteca de clases a un proyecto.

Completadas todas estas operaciones, pasaremos al fichero de código del proyecto de consola, y dado que vamos a utilizar las clases contenidas en un ensamblado distinto del que estamos posicionados, debemos importar los espacio de nombres del ensamblado; tanto su espacio raíz, como el que hemos creado manualmente. De forma adicional, hemos añadido una clase a continuación de Main( ) para demostrar como para instanciar dicha clase, al estar en el espacio de nombres raíz del proyecto de consola, no es necesario realizar ninguna importación, veamos el Código fuente 236.

```
' importamos el namespace Adicional,
' este namespace es el raíz del proyecto
' de biblioteca de clases y
' nos servirá para acceder a la clase
' Factura del proyecto Adicional
Imports Adicional

' por otro lado importamos el namespace
' Adicional.Correo que nos permitirá
' acceder a la clase Mensaje, que también
' está en la biblioteca de clases
Imports Adicional.Correo

Module Module1
    Sub Main()
        Dim loEmp As New Empleado()
        loEmp.psID = 254
        loEmp.MostrarDatos()

        Dim loFac As New Factura()
        loFac.piID = 785
        loFac.piImporte = 1200
        loFac.Datos()

        Dim loMsg As New Mensaje()
        loMsg.psTexto = "Hola mundo"
        loMsg.pdtFecha = Today
        loMsg.Visualizar()

        Console.ReadLine()
    End Sub
End Module

Public Class Empleado
    Public psID As Integer

    Public Sub MostrarDatos()
        Console.WriteLine("Identificador del empleado: {0}", Me.psID)
        Console.ReadLine()
    End Sub
End Class
```

Código fuente 236

Cuando ejecutemos el programa depurando línea a línea, comprobaremos como el flujo de la aplicación pasa al código de la biblioteca de clases al instanciar sus objetos.

19

Constructores y herencia

Métodos constructores

El primer método que es ejecutado al instanciar un objeto de la clase se denomina constructor. Este tipo de método resulta útil para tareas de configuración iniciales sobre el objeto.

No es necesario escribir un método constructor en la clase, ya que en el caso de que no exista, el compilador se encarga de crearlo implícitamente.

Para escribir nuestros propios constructores de clase, crearemos un método con el nombre New( ), como vemos en el Código fuente 237. En dicho ejemplo, al instanciarse un objeto de la clase Empleado, se asignará a una de sus propiedades la fecha actual.

```
Module General
    Sub Main()
        Dim loEmp As Empleado
        loEmp = New Empleado()
        Console.WriteLine("El objeto se ha creado el día {0}", loEmp.FechaCrea)
        Console.ReadLine()
    End Sub
End Module

Public Class Empleado
    Private mdtFechaCrea As Date

    Public Property FechaCrea() As Date
        Get
            Return mdtFechaCrea
```

```
        End Get
        Set(ByVal Value As Date)
            mdtFechaCrea = Value
        End Set
    End Property

    ' método constructor
    Public Sub New()
        ' asignamos un valor inicial
        ' a una variable de propiedad
        Me.FechaCrea = Now
    End Sub
End Class
```

Código fuente 237

Al igual que ocurre en un método normal, New( ) admite parámetros; esto nos sirve para asignar valores de inicio al objeto en el momento de su instanciación. La denominación para este tipo de métodos es *constructor parametrizado*. El Código fuente 238 nos muestra una variación del fuente anterior, utilizando un constructor de este tipo.

```
Module General
    Sub Main()
        Dim loEmp As Empleado
        loEmp = New Empleado("5/7/2002")
        Console.WriteLine("El objeto se ha creado el día {0}", loEmp.FechaCrea)
        Console.ReadLine()

        ' este es otro modo de instanciar
        ' un objeto con un constructor parametrizado
        Dim loEmp2 As New Empleado("08/4/2002")

    End Sub
End Module

Public Class Empleado
    Private mdtFechaCrea

    Public Property FechaCrea() As Date
        Get
            Return mdtFechaCrea
        End Get
        Set(ByVal Value As Date)
            mdtFechaCrea = Value
        End Set
    End Property

    ' método constructor con parámetro
    Public Sub New(ByVal ldtFecha As Date)
        ' asignamos el valor del parámetro
        ' a una variable de propiedad
        Me.FechaCrea = ldtFecha
    End Sub
End Class
```

Código fuente 238

Combinando las características de métodos constructores junto a las de sobrecarga, podemos crear un conjunto de constructores sobrecargados para la clase. Ver el Código fuente 239.

```
Public Class Empleado
    Public psNombre
    Public psApellidos
    Public psCiudad
    Private mdtFechaCrea

    ' en este constructor sin parámetros,
    ' asignamos la fecha actual
    Public Sub New()
        mdtFechaCrea = Now()
    End Sub

    ' en este constructor, asignamos valores
    ' a todos los campos de la clase
    Public Sub New(ByVal lsNombre As String, _
        ByVal lsApellidos As String, ByVal lsCiudad As String)

        psNombre = lsNombre
        psApellidos = lsApellidos
        psCiudad = lsCiudad

    End Sub
End Class
```

Código fuente 239

Herencia

Este es uno de los aspectos más importantes, sino el más importante, en un lenguaje orientado a objetos. VB.NET es la primera versión de este lenguaje que incorpora herencia, una característica demandada por un amplio colectivo de los programadores de esta herramienta; y es que, a pesar de haber sido incorporados progresivamente aspectos de orientación a objetos al lenguaje, este era el elemento fundamental que faltaba para ser plenamente OOP.

Empleando la herencia podemos crear una clase base o padre, con especificaciones generales, y a partir de ella, crear nuevas clases derivadas o hijas.

En el momento de declarar una clase derivada, y sin haber escrito más código, ya tenemos acceso a todos los miembros de la clase base; posteriormente, podemos escribir código adicional en la clase derivada para ampliar sus funcionalidades.

Una clase hija puede servir a su vez como clase base para la creación de otra clase derivada, y así sucesivamente, con lo que podemos componer nuestra propia jerarquía con la estructura de clases que necesitemos.

Para crear una clase derivada, debemos declarar una nueva clase, especificando cuál es su clase base mediante la palabra clave Inherits. En el Código fuente 240 se muestran los dos modos disponibles de crear una clase heredada.

```
' crear clase derivada en dos líneas
Public Class Administrativo
    Inherits Empleado
' crear clase derivada en la misma línea
Public Class Administrativo : Inherits Empleado
```

Código fuente 240

Una vez que creemos la clase derivada, tendremos a nuestra disposición todos los elementos de la clase base, tanto desde la propia clase, como desde el código cliente.

Por ejemplo, supongamos que se nos plantea la necesidad de crear un tipo de empleado, con características más especializadas de las que ahora tiene nuestra clase Empleado. Podemos optar por modificar toda la implementación ya existente para la clase Empleado, lo que afectaría al código cliente que ya utiliza dicha clase, y seguramente de forma negativa; o bien, podemos crear una nueva clase, que herede de Empleado, y en la que definiríamos el comportamiento de este nuevo tipo de empleado. A esta nueva clase le daremos el nombre Administrativo, y su código podemos verlo en el Código fuente 241.

```
Public Class Administrativo
    Inherits Empleado

    Public Sub EnviarCorreo(ByVal lsMensaje As String)
        Console.WriteLine("Remitente del mensaje: {0} {1}", _
            Me.Nombre, Me.Apellidos)
        Console.WriteLine("Texto del mensaje: {0}", lsMensaje)
        Console.ReadLine()
    End Sub

End Class
```

Código fuente 241

Observemos en el único método de esta clase, que estamos haciendo referencia a las propiedades Nombre y Apellidos; al heredar de Empleado, no ha sido necesario crear de nuevo dichas propiedades, estamos reutilizando las existentes en la clase base.

De igual modo sucede al instanciar un objeto de la clase, como vemos en el Código fuente 242, hemos creado un objeto Administrativo, y estamos haciendo uso tanto del nuevo método que tiene dicho objeto, como de todos los pertenecientes a su clase padre.

```
Module General
    Sub Main()
        ' instanciamos un objeto de la clase derivada
        Dim loAdmin As New Administrativo()
        ' accedemos a los miembros de la clase padre
        loAdmin.Nombre = "Almudena"
        loAdmin.Apellidos = "Cerro"
        loAdmin.MostrarDatos()
        ' ahora accedemos a los miembros de esta propia clase
        loAdmin.EnviarCorreo("Acabo de llegar")
    End Sub
End Module
```

Código fuente 242

Todas las clases necesitan una clase base

El diseño del CLR dicta como norma que toda clase creada necesita heredar de una clase base. Esto puede resultar un tanto confuso al principio, ya que en los ejemplos con la clase Empleado, no hemos heredado, al menos aparentemente, de ninguna clase.

Cuando creamos una nueva clase, si en ella no establecemos una relación explícita de herencia con otra clase, el CLR internamente la creará haciendo que herede de la clase Object, que se encuentra en el espacio de nombres System. Esto es debido a que el tipo de herencia en .NET Framework es simple, y en la jerarquía de clases de la plataforma, Object es la clase base, a partir de la cuál, se deriva el resto de clases.

Por este motivo, las declaraciones mostradas en el Código fuente 243 serían equivalentes.

```
' declaración normal (se hereda implícitamente de Object)
Public Class Empleado

' declaración heredando explícitamente de Object
Public Class Empleado
    Inherits System.Object
```

Código fuente 243

Reglas de ámbito específicas para clases

Las normas de ámbito que ya conocemos, establecen que cuando declaramos un miembro de clase con el modificador de ámbito Public, dicho elemento será accesible por todo el código de la clase, clases heredadas y código cliente; mientras que si lo declaramos con Private, ese miembro sólo será accesible por el código de la propia clase. Veamos el Código fuente 244.

```
Module General
    Sub Main()
        Dim loUsu As Usuario
        loUsu = New Usuario()
        ' accedemos al método público del objeto
        loUsu.AsignarNombre("Daniel")
    End Sub

End Module

Public Class Usuario
    ' esta variable sólo es accesible
    ' por el código de la propia clase
    Private msNombre As String

    ' este método es accesible desde cualquier punto
    Public Sub AsignarNombre(ByVal lsValor As String)
        msNombre = lsValor
    End Sub
End Class

Public Class Operador
    Inherits Usuario

    Public Sub New()
        ' accedemos a un método público
        ' de la clase base
        Me.AsignarNombre("Alfredo")
    End Sub
End Class
```

Código fuente 244

En el anterior fuente, la variable msNombre de la clase Usuario, declarada privada a nivel de clase, sólo es manipulada por los métodos de la propia clase. Por otro lado, el método AsignarNombre( ), al declararse público, es utilizado desde clases heredades y código cliente.

Además de estas normas, ya conocidas, disponemos de los modificadores descritos en los siguientes apartados, diseñados para resolver problemas concretos de ámbito entre clases.

Protected

Un miembro de clase declarado con este modificador, será accesible desde el código de su propia clase y desde cualquier clase heredada. El Código fuente 245 muestra un ejemplo del uso de Protected

```
Module Module1
    Sub Main()
        ' con una instancia del objeto Empleado o Administrativo
        ' no podemos acceder al método VerFecha()
        ' ya que es Protected
        Dim loEmp As Empleado = New Empleado()
        loEmp.psNombre = "Pedro Peral"

        Dim loAdmin As New Administrativo()
        loAdmin.piID = 567
        loAdmin.psNombre = "Juan Iglesias"
        loAdmin.pdtFecha = "5/9/2002"
        loAdmin.AsignarDNI("11223344")
        loAdmin.DatosAdmin()

        Console.Read()
    End Sub
End Module

Public Class Empleado
    Public psNombre As String
    Public pdtFecha As Date

    ' los dos siguientes miembros sólo serán visibles
    ' dentro de esta clase o en sus clases derivadas
    Protected psDNI As String

    Protected Function VerFecha()
        Return pdtFecha
    End Function

    Public Sub AsignarDNI(ByVal lsDNI As String)
        ' desde aquí sí tenemos acceso a la variable
        ' Protected declarada en la clase
        Me.psDNI = lsDNI
    End Sub
End Class

Public Class Administrativo
    Inherits Empleado

    Public piID As Integer

    Public Sub DatosAdmin()
        Console.WriteLine("Datos del administrativo")
        Console.WriteLine("Identificador: {0}", Me.piID)
        Console.WriteLine("Nombre: {0}", Me.psNombre)
        ' desde esta clase derivada sí tenemos acceso
        ' a lo miembtos Protected de la clase padre
```

```
        Console.WriteLine("Fecha: {0}", Me.VerFecha())
        Console.WriteLine("DNI: {0}", Me.psDNI)
    End Sub
End Class
```

Código fuente 245

Friend

Un miembro de clase declarado con este modificador, será accesible por todo el código de su proyecto o ensamblado.

Para poder comprobar el comportamiento utilizando el ámbito friend, debemos crear una solución formada por un proyecto de tipo consola y uno de biblioteca de clases; en este último escribimos una clase definiendo alguno de sus miembros con este modificador, como vemos en el Código fuente 246.

```
Public Class Empleado
    Public piID As Integer
    Private msNombre As String

    ' esta variable sólo puede ser
    ' accesible por tipos que estén
    ' dentro de este ensamblado
    Friend mdbSueldo As Double

    Public Property Nombre() As String
        Get
            Return msNombre
        End Get
        Set(ByVal Value As String)
            msNombre = Value
        End Set
    End Property

    Public Sub VerDatos()
        Console.WriteLine("Datos del empleado")
        Console.WriteLine("Código: {0}", Me.piID)
        Console.WriteLine("Nombre: {0}", Me.msNombre)
        Console.WriteLine("Sueldo: {0}", Me.mdbSueldo)
    End Sub

End Class

Public Class Plantilla
    Public Sub Analizar()
        Dim loEmp As Empleado = New Empleado()
        loEmp.piID = 50
        loEmp.Nombre = "Francisco Perea"

        ' desde esta clase sí podemos acceder
        ' al miembro mdbSueldo del objeto
        ' Empleado, ya que estamos en el mismo ensamblado
        loEmp.mdbSueldo = 450
        loEmp.VerDatos()
    End Sub
End Class
```

Código fuente 246

A continuación, agregamos una referencia desde el proyecto de consola hacia el proyecto de biblioteca de clases, y en el módulo de código, importamos el espacio de nombres del ensamblado correspondiente a la biblioteca de clases, escribiendo después en Main( ), código que interactúe con un objeto de la clase Empleado, comprobaremos cómo no es posible manipular los miembros friend del objeto Empleado. Ver Código fuente 247.

```
Imports ClassLibrary1

Module Module1
    Sub Main()
        Dim loEmplea As Empleado = New Empleado()

        ' al acceder a las propiedades del objeto
        ' desde este proyecto, no está disponible
        ' el miembro mdbSueldo ya que está declarado
        ' como Friend en la clase Empleado
        loEmplea.piID = 70
        loEmplea.Nombre = "Alicia Mar"
        loEmplea.VerDatos()
        Console.Read()
    End Sub
End Module
```

Código fuente 247

Aunque hemos descrito su modo de manejo a través de clases, la palabra clave Friend también puede ser utilizada como modificador de ámbito para variables y procedimientos situados en módulos de código.

Protected Friend

Los miembros de clase declarados al mismo tiempo con Protected y Friend, obtendrán una combinación de ambos modificadores; por lo tanto, serán accesibles desde el código de su clase, clases derivadas, y por todo el código que se encuentre dentro de su ensamblado.

Herencia y sobrecarga de métodos

Podemos sobrecargar métodos existentes en una clase base dentro de una clase derivada, para ello simplemente escribimos la implementación del método sobrecargado utilizando la palabra clave Overloads, tal y como se ha explicado en anteriores apartados.

Tomemos como ejemplo una clase base Empleado y su clase derivada Administrativo. Cuando calculamos los incentivos para un empleado, lo hacemos basándonos en una operación sobre el salario; sin embargo, los incentivos para el administrativo se calculan en base a un número de horas, por lo que escribimos dos implementaciones del mismo método en cada clase, sobrecargando el método en la clase Administrativo, como muestra el Código fuente 248.

```
Module Module1
    Sub Main()
        Dim loEmp As Empleado = New Empleado()
        loEmp.psNombre = "Ana Gómez"
```

```
        loEmp.piSalario = 2000
        loEmp.CalcularIncentivos()
        loEmp.VerIncentivos()

        Dim loAdmin As New Administrativo()
        loAdmin.psNombre = "Jorge Peral"
        loAdmin.piSalario = 1600
        loAdmin.CalcularIncentivos(10)
        loAdmin.VerIncentivos()

        Console.ReadLine()
    End Sub
End Module

Public Class Empleado
    Public piID As Integer
    Public psNombre As String
    Public piSalario As Integer
    Public piIncentivos As Integer

    ' calcular los incentivos en base
    ' al salario
    Public Sub CalcularIncentivos()
        Me.piIncentivos = Me.piSalario / 10
    End Sub

    Public Sub VerIncentivos()
        Console.WriteLine("Los incentivos de {0} son {1}", _
            Me.psNombre, Me.piIncentivos)
    End Sub
End Class

Public Class Administrativo
    Inherits Empleado

    ' calcular los incentivos en base a horas
    Public Overloads Sub CalcularIncentivos(ByVal liHoras As Integer)
        Me.piIncentivos = liHoras * 15
    End Sub
End Class
```

Código fuente 248

Hemos de aclarar que si no utilizáramos la palabra clave Overloads en la subclase, el programa también se ejecutaría, pero obtendríamos un aviso del compilador advirtiéndonos de la situación. Este mensaje lo podemos ver utilizando la ventana Lista de tareas, que emplea el IDE para mostrar los errores y avisos del compilador. Ver Figura 200.

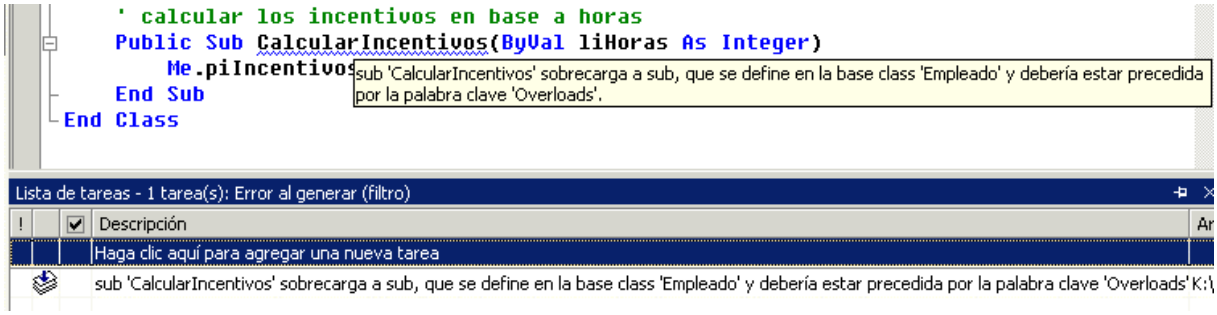


Figura 200. Lista de tareas y editor de código mostrando aviso del compilador.

MyBase, acceso a los métodos de la clase base

Esta palabra clave proporciona acceso a los miembros de una clase base desde su correspondiente subclase.

Siguiendo con el ejemplo de la sobrecarga descrito en el apartado anterior, supongamos que para calcular los incentivos de un administrativo, queremos en primer lugar, realizar la misma operación que hacemos con los empleados base, y después, un cálculo específico para el administrativo. En tal caso, modificaremos el método CalcularIncentivos( ) en la clase Administrativo, añadiéndole una llamada al mismo método de la clase padre. Veamos el Código fuente 249.

```
Public Class Administrativo
    Inherits Empleado

    Public Overloads Sub CalcularIncentivos(ByVal liHoras As Integer)
        ' llamamos a la clase base con MyBase para hacer
        ' en primer lugar los mismos cálculos de incentivos
        ' de la clase Empleado
        MyBase.CalcularIncentivos()

        ' después se hacen los cálculos propios de
        ' esta clase
        Me.piIncentivos += liHoras * 15
    End Sub
End Class
```

Código fuente 249

Al utilizar MyBase, no es obligatorio llamar desde el método en la clase hija, a su misma versión en la clase padre; podríamos perfectamente en el ejemplo anterior, haber llamado desde el método CalcularIncentivos( ) de la clase Administrativo, al método VerIncentivos( ) de la clase Empleado, todo depende de los requerimientos del diseño de la clase. Ver Código fuente 250.

```
MyBase.VerIncentivos()
```

Código fuente 250

Herencia y sobre-escritura de métodos

Esta técnica consiste en la capacidad de crear, en una clase derivada, un método que altere parcial o totalmente, la implementación ya existente de dicho método en la clase base. Una de las diferencias existentes con la sobrecarga de métodos, estriba en que al sobrescribir, el método en la subclase puede tener el mismo nombre y lista de parámetros que el ya existente en la clase padre. Podemos sobrescribir tanto métodos como propiedades.

Para indicar en la clase base que un método podrá ser sobrescrito en una subclase, debemos declarar dicho método utilizando la palabra clave Overridable. Posteriormente, cuando en una clase derivada queramos rescribir el método de la clase base, lo declararemos empleando la palabra clave Overrides. Podemos deducir por lo tanto, que la reescritura de métodos es un proceso que se debe realizar con el consentimiento previo de la clase base.

El Código fuente 251 muestra un ejemplo del uso de este tipo de métodos. En él creamos las ya conocidas clase base Empleado y subclase Administrativo, y en ambas escribimos el método VerDatos( ), con la particularidad de que en la clase hija, cambiamos totalmente su implementación.

```
Module Module1
    Sub Main()
        Dim loEmp As New Empleado()
        loEmp.piID = 50
        loEmp.Nombre = "juan casas"
        loEmp.VerDatos()

        Console.WriteLine()

        Dim loAdmin As New Administrativo()
        loAdmin.piID = 129
        loAdmin.Nombre = "elena redondo"
        loAdmin.VerDatos()

        Console.ReadLine()
    End Sub
End Module

Public Class Empleado
    Public piID As Integer
    Private msNombre As String

    Public Property Nombre() As String
        Get
            Return msNombre
        End Get
        Set(ByVal Value As String)
            msNombre = Value
        End Set
    End Property

    ' marcamos el método como rescribible con Overridable
    Public Overridable Sub VerDatos()
        Console.WriteLine("Datos del empleado: {0}-{1}", _
            Me.piID, Me.Nombre)
    End Sub
End Class

Public Class Administrativo : Inherits Empleado
    ' rescribimos este método totalmente usando Overrides
    Public Overrides Sub VerDatos()
        Console.WriteLine("Datos del empleado")
        Console.WriteLine("==================")
        Console.WriteLine("Código: {0}", Me.piID)
        Console.WriteLine("Nombre: {0}", UCase(Me.Nombre))
    End Sub
End Class
```

Código fuente 251

Pero, ¿qué sucede si queremos utilizar la implementación del método base en la clase derivada?, pues sólo necesitamos llamar al método de la clase padre usando la palabra clave MyBase.

Para ilustrar esta situación, añadiremos a la clase Empleado la propiedad Salario, y un método para calcularlo, de modo que todos los empleados tengan inicialmente el mismo salario, sin embargo, los administrativos necesitan un pequeño incremento. Para no tener que volver a realizar el cálculo en la

clase Administrativo, vamos a aprovechar el cálculo que ya se realiza en la clase padre, añadiendo sólo las operaciones particulares que necesitemos. Veámoslo en el Código fuente 252.

```
Module Module1
    Sub Main()
        Dim loEmp As New Empleado()
        loEmp.piID = 50
        loEmp.Nombre = "juan casas"
        loEmp.VerDatos()
        loEmp.CalcularSalario()
        Console.WriteLine("Salario {0}", loEmp.Salario)

        Console.WriteLine()

        Dim loAdmin As New Administrativo()
        loAdmin.piID = 129
        loAdmin.Nombre = "elena redondo"
        loAdmin.VerDatos()
        loAdmin.CalcularSalario()
        Console.WriteLine("Salario {0}", loAdmin.Salario)

        Console.ReadLine()
    End Sub
End Module

Public Class Empleado
    '......
    '......
    Public miSalario As Integer

    Public Property Salario() As Integer
        Get
            Return miSalario
        End Get
        Set(ByVal Value As Integer)
            miSalario = Value
        End Set
    End Property

    Public Overridable Sub CalcularSalario()
        Me.Salario = 800
    End Sub
    '......
    '......
End Class

Public Class Administrativo : Inherits Empleado
    '......
    '......
    Public Overrides Sub CalcularSalario()
        ' utilizamos el método de la clase base
        MyBase.CalcularSalario()
        Me.Salario += 50
    End Sub
End Class
```

Código fuente 252

Debido a cuestiones de diseño, en algunas ocasiones precisaremos que al mismo tiempo que sobrescribimos un miembro dentro de una clase heredada, dicho miembro no pueda ser sobrescrito por las clases que hereden de esta. En estas situaciones, al declarar el miembro, usaremos la palabra clave NotOverridable.

Volvamos pues, al ejemplo de la clase Administrativo, en el que sobrescribíamos el método VerDatos(). Si cambiamos la declaración de dicho método por la mostrada en el Código fuente 253, una tercera clase Directivo, que heredase de Administrativo, no podría sobrescribir el mencionado método.

```
Public Class Administrativo : Inherits Empleado
    ' rescribimos este método totalmente usando Overrides
    ' e impedimos que pueda ser rescrito por clases
    ' derivadas de esta
    Public NotOverridable Overrides Sub VerDatos()
        Console.WriteLine("Datos del empleado")
        Console.WriteLine("==================")
        Console.WriteLine("Código: {0}", Me.piID)
        Console.WriteLine("Nombre: {0}", UCase(Me.Nombre))
    End Sub
End Class

Public Class Directivo : Inherits Administrativo
    ' se produce un error, no se puede sobrescribir este método
    ' ya que la clase Administrativo lo impide con NotOverridable
    Public Overrides Sub VerDatos()
        '.....
        '.....
    End Sub
End Class
```

Código fuente 253

No podemos utilizar NotOverridable en métodos de una clase base, ya que la misión de este modificador es impedir la sobre-escritura de miembros en clases derivadas, pero desde una clase que a su vez también ha sido derivada desde la clase base. Si no queremos, en una clase base, que un método pueda ser sobrescrito, simplemente no utilizamos en su declaración la palabra clave Overridable.

Diferencias entre sobrecarga y sobre-escritura en base al tipo de enlace

La otra diferencia entre sobrecarga y sobre-escritura consiste en el tipo de enlace que utilizan. Mientras que la sobrecarga se basa en enlace temprano, la sobre-escritura emplea enlace tardío.

El mejor modo de comprobar este punto, consiste en declarar una variable con un tipo perteneciente a una clase base, pero asignándole un objeto correspondiente a una clase heredada.

Por ejemplo, en el caso de la sobrecarga, creamos las ya conocidas clases Empleado y Administrativo, escribiendo el método VerAlta( ), sobrecargado en cada una de ellas.

A continuación declaramos una variable de tipo Empleado, pero instanciamos un objeto de la clase Administrativo y lo asignamos a la variable. Debido a que el enlace temprano se basa en el tipo de la variable y no en el objeto que contiene, el método VerAlta( ) al que podremos acceder será el que se encuentra en la clase Empleado. Veamos el Código fuente 254.

```
Module Module1
    Sub Main()
```

```
        Dim loPersona As Empleado
        loPersona = New Administrativo()
        loPersona.psNombre = "Juan García"
        loPersona.pdtFHAlta = "15/1/2002"
        ' como la sobrecarga utiliza enlace temprano,
        ' se basa en el tipo de la variable y no
        ' en el objeto que se asigna a esa variable,
        ' por ello sólo es visible la implementación
        ' del método que hay en la clase Empleado
        loPersona.VerAlta()
        ' si intentamos ejecutar el método VerAlta()
        ' que recibe una cadena, se producirá el siguiente error:
        ' "Demasiados argumentos para 'Public Sub VerAlta()'."
        loPersona.VerAlta("Mes")  ' <-- error
        Console.ReadLine()
    End Sub
End Module

Public Class Empleado
    Public psNombre As String
    Public pdtFHAlta As Date

    ' mostrar la fecha de alta al completo
    Public Sub VerAlta()
        Console.WriteLine("El empleado {0} se incorporó el {1}", _
            Me.psNombre, Me.pdtFHAlta)
    End Sub
End Class

Public Class Administrativo : Inherits Empleado
    ' mostrar sólo una parte de la fecha de alta
    ' según el parámetro pasado
    Public Overloads Sub VerAlta(ByVal lsParteFecha As String)
        Dim lsValorFecha As String
        Select Case lsParteFecha
            Case "Mes"
                lsValorFecha = Format(Me.pdtFHAlta, "MMMM")
            Case "DiaSemana"
                lsValorFecha = Format(Me.pdtFHAlta, "dddd")
        End Select
        Console.WriteLine("Empleado {0}", Me.psNombre)
        Console.WriteLine("Incorporado {0}", lsValorFecha)
    End Sub
End Class
```

Código fuente 254

Pasando seguidamente a la sobre-escritura, escribiremos en la clase padre el método MostrarNombre( ), y lo sobrescribiremos en la clase hija. Instanciaremos después un objeto Administrativo y lo asignaremos a la variable tipificada como Empleado. Debido a que el enlace tardío se basa en el tipo del objeto que contiene la variable, y no en el tipo de la variable, al llamar al método MostrarNombre( ), se ejecutará la versión de la clase Administrativo. Veamos el Código fuente 255.

```
Module Module1
    Sub Main()
        Dim loPersona As Empleado
        loPersona = New Administrativo()
        loPersona.psNombre = "Juan García"
        ' como la sobre-escritura utiliza enlace tardío,
        ' se basa en el objeto que contiene la variable y
        ' no en el tipo de dato de la variable, se ejecuta
        ' la versión del método MostrarNombre() que está
```

```
        ' en la clase Administrativo, ya que el objeto
        ' que contiene la variable loPersona es una
        ' instancia de Administrativo
        loPersona.MostrarNombre()
        Console.ReadLine()
    End Sub
End Module

Public Class Empleado
    Public psNombre As String
    Public pdtFHAlta As Date

    Public Overridable Sub MostrarNombre()
        Console.WriteLine("El nombre del empleado es {0}", _
            Me.psNombre)
    End Sub
End Class

Public Class Administrativo : Inherits Empleado
    Public Overrides Sub MostrarNombre()
        Console.WriteLine("Nombre del empleado")
        Console.WriteLine("===================")
        Console.WriteLine(UCase(Me.psNombre))
    End Sub
End Class
```

Código fuente 255

La Tabla 25 muestra, al utilizar sobre-escritura, la clase de la cuál será ejecutado el método, en función de la referencia de la variable y el tipo de objeto.

Si la referencia de la variable es de la clase...	**...y el tipo de objeto es de la clase...**	**...el método ejecutado será de la clase**
Base	Base	Base
Base	Derivada	Derivada
Derivada	Derivada	Derivada

Tabla 25. Método ejecutado mediante enlace tardío bajo sobre-escritura.

Debido al hecho de que los miembros sobrescritos emplean enlace tardío, otra de las denominaciones que se utiliza para ellos es la de *método virtual*.

Ocultamiento de miembros de una clase

Esta técnica consiste en crear dentro de una clase derivada, miembros con el mismo nombre (y firma, en el caso de métodos) que los existentes en la clase base, pero ocultando el acceso a los miembros de la clase base para los objetos instanciados de la subclase. Utilizaremos la palabra clave Shadows, en la declaración de aquellos miembros a esconder.

Cuando aplicamos el ocultamiento sobre una subclase que tiene métodos sobrecargados en la clase base, dichos métodos sobrecargados quedarán inaccesibles para la clase derivada. Como ejemplo, en el Código fuente 256, la clase Empleado implementa dos versiones sobrecargadas del método Sueldo( ),

mientras que la clase hija Administrativo también tiene el método Sueldo( ), pero al declararse con Shadows, impide que los objetos de tipo Administrativo ejecuten los métodos Sueldo( ) de la clase Empleado.

```
Module Module1
    Sub Main()
        Dim loAdmin As New Administrativo()
        Dim ldbImporte As Double
        Dim lsFecha As String
        loAdmin.Salario = 925.86
        ldbImporte = loAdmin.Sueldo(80, "Viajes")

        ' los siguientes métodos están ocultos
        ' desde este objeto y se produce un error al llamarlos
        loAdmin.Sueldo()
        lsFecha = loAdmin.Sueldo(5)
    End Sub

End Module

Public Class Empleado
    Private mdbSalario As Double

    Public Property Salario() As Double
        Get
            Return mdbSalario
        End Get
        Set(ByVal Value As Double)
            mdbSalario = Value
        End Set
    End Property

    ' métodos sobrecargados
    Public Overloads Sub Sueldo()
        ' aquí mostramos en consola el importe del sueldo formateado
        Console.WriteLine("El sueldo es {0}", Format(Me.Salario, "#,#.##"))
        Console.ReadLine()
    End Sub

    Public Overloads Function Sueldo(ByVal liDia As Integer) As String
        ' aquí mostramos la fecha del mes actual
        ' en la que se realizará la transferencia
        ' del sueldo al banco del empleado
        Dim ldtFechaActual As Date
        Dim lsFechaCobro As String
        ldtFechaActual = Now()

        lsFechaCobro = CStr(liDia) & "/" & _
            CStr(Month(ldtFechaActual)) & "/" & _
            CStr(Year(ldtFechaActual))

        Return lsFechaCobro
    End Function
End Class

Public Class Administrativo
    Inherits Empleado

    ' este método ensombrece/oculta a los sobrecargados
    ' de la clase base Empleado
    Public Shadows Function Sueldo(ByVal ldbImporteIncentivo As Double, _
      ByVal lsTipoIncentivo As String) As Double
        ' aquí calculamos la cantidad de incentivo
        ' que se añadirá al sueldo del empleado,
```

```
        ' en función del tipo de incentivo
        Dim ldbIncentivo As Double

        ' según el tipo de incentivo,
        ' se descuenta un importe
        ' de la cantidad del incentivo
        Select Case lsTipoIncentivo
            Case "Viajes"
                ldbIncentivo = ldbImporteIncentivo - 30
            Case "Extras"
                ldbIncentivo = ldbImporteIncentivo - 15
        End Select

        Return ldbIncentivo
    End Function
End Class
```

Código fuente 256

Cuando en una clase hija creamos un método con el mismo nombre y parámetros que en la clase padre, el compilador realiza un ocultamiento implícito, aunque genera un aviso, recomendando que declaremos el método de la clase hija con Shadows. Veamos el Código fuente 257.

```
Public Class Empleado
    '....
    Public Sub Sueldo()
        ' aquí mostramos en consola el importe del sueldo formateado
        Console.WriteLine("El sueldo es {0}", Format(Me.Salario, "#,#.##"))
        Console.ReadLine()
    End Sub
    '....
End Class

Public Class Administrativo
    '....
    ' si aquí no utilizáramos Shadows, el entorno
    ' marcaría este método con un aviso
    Public Shadows Sub Sueldo()
        ' aquí incrementamos el valor actual de la propiedad Salario
        Me.Salario += 250
    End Sub
    '....
End Class
```

Código fuente 257

Por otra parte, si aplicamos el ocultamiento en la sobre-escritura, el comportamiento del objeto se verá profundamente afectado. La mejor situación para comprobar este particular consiste en declarar una variable de la clase base y asignarle un objeto de una clase heredada.

A pesar de que, como hemos comentado anteriormente, la sobre-escritura se basa en el enlace tardío, si ocultamos un miembro de la clase derivada, forzaremos al objeto a dirigirse a la versión de dicho miembro existente en la clase padre.

El ejemplo del Código fuente 258 muestra este caso. En él creamos nuestras dos clases habituales, Empleado y Administrativo, relacionadas mediante herencia, con un método sobrescrito en ambas, que tiene la particularidad de que la versión existente en la clase derivada está oculto con Shadows. Esto hará que al instanciar un objeto de la clase hija, y pasárselo a una variable referenciada hacia la clase

padre, la llamada al método sea desviada hacia la implementación existente en la clase padre, en lugar de a la clase derivada como sería su comportamiento habitual.

```
Module Module1
    Sub Main()
        Dim loPersona As Empleado
        loPersona = New Administrativo()
        loPersona.psNombre = "Juan"
        ' estamos utilizando sobre-escritura,
        ' por lo que el enlace tardío emplea el objeto
        ' que hay dentro de la variable y no la
        ' referencia de la variable;
        ' al estar oculta con Shadows la implementación
        ' del método MostrarNombre() en la clase Administrativo
        ' se ejecuta dicho método pero de la clase Empleado
        loPersona.MostrarNombre()

        Console.ReadLine()
    End Sub
End Module

Public Class Empleado
    Public psNombre As String
    Public pdtFHAlta As Date

    Public Overridable Sub MostrarNombre()
        Console.WriteLine("El nombre del empleado es {0}", _
            Me.psNombre)
    End Sub
End Class

Public Class Administrativo : Inherits Empleado
    ' ocultamos este método
    Public Shadows Sub MostrarNombre()
        Console.WriteLine("Nombre del empleado")
        Console.WriteLine("===================")
        Console.WriteLine(UCase(Me.psNombre))
    End Sub
End Class
```

Código fuente 258

La Tabla 26 muestra, al utilizar sobre-escritura y ocultamiento, la clase de la cuál será ejecutado el método, en función de la referencia de la variable y el tipo de objeto.

Si la referencia de la variable es de la clase...	**...y el tipo de objeto es de la clase...**	**...el método ejecutado será de la clase**
Base	Base	Base
Base	Derivada	Base
Derivada	Derivada	Derivada

Tabla 26. Método ejecutado mediante enlace tardío bajo sobre-escritura, aplicando ocultamiento.

Comportamiento de las palabras clave Me, MyClass y MyBase ante la sobre-escritura de métodos

En anteriores apartados, explicamos que las palabras clave Me y MyClass, sirven para llamar a los miembros de una clase desde el código de la propia clase; mientras que con la palabra clave MyBase podíamos llamar a los miembros de la clase base desde una subclase.

Sin embargo, al efectuar sobre-escritura de miembros, los resultados utilizando estas palabras clave pueden no ser los inicialmente esperados, debido al tipo de enlace que emplean. MyBase y MyClass utilizan enlace temprano para realizar la llamada, mientras que Me usa enlace tardío.

Tomando como ejemplo la clase base Empleado y sus clases derivadas Administrativo y Directivo, creamos el método sobrescrito VerDatos( ) en cada una de estas clases; cada versión tendrá una implementación diferente que nos permita diferenciar la clase que lo ejecuta.

A continuación, en la clase Administrativo escribimos el método DameInformacion( ), desde el que hacemos llamadas a VerDatos( ) utilizando MyBase, MyClass, Me, y una última llamada especificando solamente el nombre del método.

Si instanciamos un objeto de la clase Administrativo, al ejecutar DameInformacion( ) el resultado será el esperado: MyBase llamará a la versión de VerDatos( ) que hay en la clase Empleado, mientras que el resto de llamadas se harán a la implementación de dicho método en la clase Administrativo.

Sin embargo, los efectos no serán tan obvios al instanciar un objeto de la clase Directivo: al ejecutar DameInformacion( ), MyBase llamará a VerDatos( ) de Empleado, MyClass llamará a VerDatos( ) de Administrativo, pero Me realiza una llamada al método que se encuentra en la clase más alejada en la jerarquía, respecto a la clase base, es decir, la clase Directivo; el efecto será el mismo si ponemos el nombre del método sin utilizar Me. Veamos este ejemplo en el Código fuente 259.

```
Module Module1
    Public Sub Main()
        Dim loDirec As Directivo = New Directivo()
        loDirec.piID = 980
        loDirec.psNombre = "Gema Peral"
        loDirec.pdtFHAlta = "20/5/2002"
        loDirec.DameInformacion()
        Console.Read()
    End Sub
End Module

Public Class Empleado
    Public piID As Integer
    Public psNombre As String
    Public pdtFHAlta As Date

    Public Overridable Sub VerDatos()
        Console.WriteLine("Información sobre el empleado." & _
            " ID:{0} - Nombre:{1} - Fecha de alta:{2}", _
            Me.piID, Me.psNombre, Me.pdtFHAlta)
        Console.WriteLine()
    End Sub
End Class

Public Class Administrativo : Inherits Empleado
    Public Overrides Sub VerDatos()
        Console.WriteLine("Datos del empleado")
        Console.WriteLine("==================")
```

```
        Console.WriteLine("Identificador: {0}", Me.piID)
        Console.WriteLine("Nombre: {0}", StrConv(Me.psNombre,
VbStrConv.ProperCase))
        Console.WriteLine("Fecha de alta: {0}", Format(Me.pdtFHAlta, "dd/MMM/yy"))
        Console.WriteLine()
    End Sub

    Public Sub DameInformacion()
        ' cuando el objeto en ejecución
        ' es de la clase derivada Directivo
        ' los resultados de las llamadas desde
        ' este método al método VerDatos() son
        ' los siguientes:
        MyBase.VerDatos()   ' llama a Empleado.VerDatos() - enlace temprano
        MyClass.VerDatos()  ' llama a Administrativo.VerDatos() - enlace temprano
        Me.VerDatos()       ' llama a Directivo.VerDatos() - enlace tardío
        VerDatos()          ' llama a Directivo.VerDatos() - enlace tardío
    End Sub
End Class

Public Class Directivo : Inherits Administrativo
    Public Overrides Sub VerDatos()
        Console.WriteLine("El empleado {0} comenzó a trabajar en el mes de:", _
            Me.psNombre)
        Console.WriteLine(Format(Me.pdtFHAlta, "MMMM"))
        Console.WriteLine()
    End Sub
End Class
```

Código fuente 259

Herencia y métodos constructores

Podemos crear una clase base que implemente un constructor y una subclase sin él. En esta situación, cuando instanciemos un objeto de la subclase, se llamará implícitamente al constructor de la clase base para ejecutar el código de inicialización. También es posible crear el constructor sólo en la clase derivada.

Si ambas clases disponen de un constructor, en primer lugar se ejecutará el constructor de la clase base y después el de la clase derivada. Realmente, el primer constructor ejecutado corresponde a la clase Object, y sucesivamente, se irán ejecutando todos los constructores de la jerarquía de clases hasta llegar a la clase que originó la llamada.

El problema sobreviene cuando en la clase base creamos un constructor parametrizado, ya que ello obliga a sus clases derivadas a crear también un método constructor dentro del cuál se haga una llamada al constructor de la clase base. Para llamar explícitamente a un método de la clase base desde una subclase utilizaremos la palabra clave MyBase, que contiene una referencia hacia la clase padre.

Veamos un ejemplo, en el Código fuente 260 se crea una clase padre Empleado y la subclase Administrativo. Puesto que Empleado dispone de un constructor parametrizado, en Administrativo debemos crear también un constructor, y dentro de él llamar en primer lugar al constructor base.

```
Public Class Empleado
    Public piID As Integer
    Public psNombre As String
    Public piSalario As Integer

    ' constructor parametrizado
```

```
    Public Sub New(ByVal lsNombre As String)
        Me.psNombre = lsNombre
    End Sub
End Class

Public Class Administrativo
    Inherits Empleado

    ' constructor normal
    Public Sub New()
        ' llamada al constructor
        ' de la clase base
        MyBase.New("Juan")
        Me.piSalario = 100
    End Sub
End Class
```

Código fuente 260

Podemos no obstante, evitar la obligación de escribir un constructor en la clase derivada, si en la clase padre creamos un constructor sin parámetros. Como ya sabemos, la sobrecarga de métodos nos permite escribir varios métodos con el mismo nombre y diferente lista de parámetros. Modifiquemos para ello la clase Empleado como muestra el Código fuente 261.

```
Public Class Empleado
    Public piID As Integer
    Public psNombre As String
    Public piSalario As Integer

    ' constructor parametrizado
    Public Sub New(ByVal lsNombre As String)
        Me.psNombre = lsNombre
    End Sub

    ' constructor sin parámetros
    Public Sub New()
        psNombre = "hola"
    End Sub
End Class

Public Class Administrativo
    Inherits Empleado

    ' al disponer en la clase base de
    ' un constructor normal, ya no hay
    ' necesidad de crear un constructor
    ' en esta clase derivada
End Class
```

Código fuente 261

Finalmente, debemos apuntar dos reglas que debe cumplir todo método constructor de una subclase que llame al constructor de su clase base: en primer lugar, el constructor base debe ser llamado en la primera línea del constructor derivado; en segundo lugar, el constructor base sólo puede ser llamado una vez desde el constructor derivado.

Clases selladas o no heredables

Toda clase que declaremos en nuestro código es heredable por defecto; esto supone un elevado grado de responsabilidad, en el caso de que diseñemos una clase pensando en que pueda ser utilizada por otros programadores que hereden de ella. Si en un determinado momento, necesitamos hacer cambios en nuestra clase, dichos cambios afectarán a las clases derivadas que hayan sido creadas.

Por dicho motivo, si no queremos que nuestra clase pueda ser heredada por otras, debemos declararla de forma que no permita herencia; a este tipo de clase se le denomina clase no heredable o sellada (sealed).

Para definir una clase no heredable, debemos utilizar la palabra clave NotInheritable en el momento de su declaración.

En la Figura 201 hemos creado la clase Fichero como no NotInheritable, por ello, cuando a continuación declaramos la clase FiTexto e intentamos que herede de Fichero, se mostrará un mensaje de error en el editor de código, indicándonos que no es posible establecer esta relación de herencia puesto que Fichero es una clase sellada.

```
Public NotInheritable Class Fichero
    Public psRuta As String

End Class

Public Class FiTexto
    Inherits Fichero
                    'FiTexto' no puede heredar de 'class Fichero' porque 'Fichero' aparece marcado como 'NotInheritable'.
End Class
```

Figura 201. No es posible heredar de una clase NotInheritable.

Clases abstractas o no instanciables

Una clase abstracta es aquella que no permite la instanciación directa de objetos a partir de ella, siendo necesario una clase derivada para acceder a sus miembros. Una clase concreta es el tipo de clase que venimos utilizando hasta el momento, desde la cuál, podemos instanciar objetos.

Aunque en una clase abstracta podemos escribir un método constructor, sólo será accesible desde el constructor de la subclase.

Para definir una clase abstracta utilizaremos la palabra clave MustInherit en el momento de su declaración, como muestra el Código fuente 262.

```
Public MustInherit Class Empleado
```

Código fuente 262

Dentro de una clase abstracta podemos implementar propiedades y métodos, en la forma que hemos visto hasta el momento. Adicionalmente, podemos obligar a que determinados miembros sean sobrescritos por la clase heredada; son los denominados *miembros abstractos*, y se declaran usando la palabra clave MustOverride, como vemos en el Código fuente 263.

```
Module Module1
    Public Sub Main()
        Dim loAdmin As New Administrativo()
        loAdmin.piID = 789
        loAdmin.psNombre = "Pedro Pinares"
        Console.WriteLine("Nombre en mayúsculas del administrativo: {0}", _
            loAdmin.NombreMay)
        loAdmin.pdtFHAlta = "8/10/01"
        loAdmin.MesesActivo()
        Console.Read()
    End Sub
End Module

' clase abstracta,
' no podemos crear objetos Empleado
Public MustInherit Class Empleado
    Public piID As Integer
    Public psNombre As String
    Public pdtFHAlta As Date

    Public ReadOnly Property NombreMay() As String
        Get
            Return UCase(Me.psNombre)
        End Get
    End Property

    ' método abstracto;
    ' este método debe ser sobrescrito
    ' por la clase derivada
    Public MustOverride Sub MesesActivo()

    Public Sub VerDatos()
        Console.WriteLine("Información sobre el empleado." & _
            " ID:{0} - Nombre:{1} - Fecha de alta:{2}", _
            Me.piID, Me.psNombre, Me.pdtFHAlta)
        Console.WriteLine()
    End Sub
End Class

' desde esta clase tendremos acceso
' a los miembros de la clase Empleado
Public Class Administrativo
    Inherits Empleado

    ' en esta clase sobrescribimos este método
    ' declarado como abstracto en la clase abstracta
    Public Overrides Sub MesesActivo()
        Console.WriteLine("Entró en el mes de {0}", _
            Format(Me.pdtFHAlta, "MMMM"))
        Console.WriteLine("El número de meses que lleva es: {0}", _
            DateDiff(DateInterval.Month, Me.pdtFHAlta, Now))
    End Sub

    Public Overrides Sub VerDatos()
        '....
        '....
    End Sub
End Class
```

Código fuente 263

Debemos tener en cuenta que los miembros abstractos sólo tienen sentido si son declarados en clases abstractas. Por tal motivo, sólo podremos crear métodos con MustOverride en clases que hayan sido definidas como MustInherit.

En lo que respecta al polimorfismo conseguido a través de clases abstractas, podemos crear un procedimiento que reciba como parámetro tipificado como clase abstracta, de forma que en función del objeto pasado, y que debe ser de un tipo derivado de la clase abstracta, el comportamiento será diferente en cada caso. Veámoslo en el Código fuente 264.

```
Module Module1
    Public Sub Main()
        '....
    End Sub

    ' el objeto que reciba este procedimiento será
    ' de cualquiera de las clases que hereden de
    ' Empleado y ejecutará la implementación del método
    ' VerDatos() que tenga dicho objeto
    Public Sub MostrarInformacion(ByVal loEmple As Empleado)
        loEmple.VerDatos()
    End Sub
End Module
```

Código fuente 264.

20

Elementos compartidos e interfaces

Comprobación del tipo de un objeto y moldeado (casting)

En algunas circunstancias, puede ser útil codificar un procedimiento que reciba como parámetro un objeto genérico, y dependiendo del tipo del objeto, ejecutar ciertos métodos.

Si tipificamos un parámetro como un objeto genérico, es decir, de tipo Object, necesitamos implementar dentro del procedimiento un mecanismo que, en primer lugar, compruebe a qué tipo pertenece el objeto, y por otra parte, acceda adecuadamente a su lista de miembros.

Como ejemplo consideremos el siguiente caso: creamos una clase Empleado y otra Factura, las cuales vemos en el Código fuente 265.

```
Public Class Empleado
    Public piID As String
    Public psNombre As String

    Public Overridable Sub VerDatos()
        Console.WriteLine("Código de empleado: {0}, nombre: {1}", _
            Me.piID, Me.psNombre)
    End Sub
End Class

Public Class Factura
    Public pdtFecha As Date
    Public piImporte As Integer

    Public Sub Emitir()
```

```
        Console.WriteLine("Se procede a generar la siguiente factura:")
        Console.WriteLine("Fecha de factura: {0}", Me.pdtFecha)
        Console.WriteLine("Importe de la factura: {0}", Me.piImporte)
    End Sub
End Class
```

Código fuente 265

A continuación, necesitamos un procedimiento al que podamos pasar indistintamente un objeto Empleado o Factura. Dentro de dicho procedimiento, al que denominaremos ManipularVarios( ), y que llamaremos desde Main( ), comprobaremos el tipo de objeto llamando al método GetType( ), que implementan todos los objetos del entorno, debido a que es un método de la clase Object, y como ya sabemos, todas las clases heredan implícitamente de Object.

GetType( ) devuelve un objeto de la clase Type. Esta clase es de una enorme utilidad, ya que nos proporciona toda la información relativa al tipo del objeto. Para nuestro problema particular, interrogaremos a la propiedad Name del objeto, que nos devolverá una cadena con el nombre de la clase a que pertenece el objeto.

Conociendo ya el tipo de objeto con el que tratamos, utilizaremos la función CType( ), que realiza un moldeado de la variable que contiene el objeto hacia un tipo determinado, y nos permite acceder a los elementos del objeto.

El lector puede argumentar que no sería necesario el uso de CType(), y en efecto es así; podríamos haber utilizado directamente la variable, situando a continuación el método a ejecutar. Esta técnica, no obstante, tiene el inconveniente de que utiliza enlace tardío para acceder al objeto.

CType( ) sin embargo, tiene la ventaja de que opera bajo enlace temprano, con lo cuál, el rendimiento en ejecución es mayor. Veamos el código cliente que accedería a estas clases en el Código fuente 266.

```
Module Module1
    Public Sub Main()
        Dim loEmple As New Empleado()
        loEmple.piID = 58
        loEmple.psNombre = "Elena Peral"
        ManipularVarios(loEmple)

        Dim loFac As New Factura()
        loFac.pdtFecha = "25/2/2002"
        loFac.piImporte = 475
        ManipularVarios(loFac)

        Console.Read()
    End Sub

    Public Sub ManipularVarios(ByVal loUnObjeto As Object)
        ' obtenemos información sobre el tipo del objeto
        Dim loTipoObj As Type
        loTipoObj = loUnObjeto.GetType()

        ' comprobamos qué tipo de objeto es,
        ' y en función de eso, ejecutamos el
        ' método adecuado
        Select Case loTipoObj.Name
            Case "Empleado"
                CType(loUnObjeto, Empleado).VerDatos()
            Case "Factura"
                CType(loUnObjeto, Factura).Emitir()
```

```
        End Select
    End Sub
End Module
```

Código fuente 266

En el caso de clases heredadas, y con métodos sobrescritos, CType( ) discierne la implementación de clase a la que debe dirigirse.

Si añadimos la clase Administrativo como hija de Empleado, y sobrescribimos el método VerDatos( ), cuando pasemos al procedimiento ManipularVarios( ) un objeto Administrativo, dado que esta clase hereda de Empleado, con un sencillo cambio en la estructura Select...Case que comprueba el tipo de objeto, ejecutaremos la implementación sobrescrita del método VerDatos( ) en la clase Administrativo. Veamos el Código fuente 267.

```
Module Module1
    Public Sub Main()
        Dim loEmple As New Empleado()
        loEmple.piID = 58
        loEmple.psNombre = "Elena Peral"
        ManipularVarios(loEmple)

        Dim loAdmin As New Administrativo()
        loAdmin.piID = 80
        loAdmin.psNombre = "Alfredo Iglesias"
        ManipularVarios(loAdmin)

        Dim loFac As New Factura()
        loFac.pdtFecha = "25/2/2002"
        loFac.piImporte = 475
        ManipularVarios(loFac)

        Console.Read()
    End Sub

    Public Sub ManipularVarios(ByVal loUnObjeto As Object)
        ' obtenemos información sobre el tipo del objeto
        Dim loTipoObj As Type
        loTipoObj = loUnObjeto.GetType()

        ' comprobamos qué tipo de objeto es,
        ' y en función de eso, ejecutamos el
        ' método adecuado
        Select Case loTipoObj.Name
            Case "Empleado", "Administrativo"
                CType(loUnObjeto, Empleado).VerDatos()
            Case "Factura"
                CType(loUnObjeto, Factura).Emitir()
        End Select
    End Sub
End Module

Public Class Administrativo
    Inherits Empleado

    Public pdtFHAlta As Date

    Public Overrides Sub VerDatos()
        Console.WriteLine("DATOS DEL ADMINISTRATIVO")
        Console.WriteLine("==================")
        Console.WriteLine("Código: {0}", Me.piID)
```

```
        Console.WriteLine("Nombre: {0}", Me.psNombre)
    End Sub

    Public Function MesAlta()
        Return Format(Me.pdtFHAlta, "MMMM")
    End Function
End Class
```

Código fuente 267

Pero ¿cómo podríamos ejecutar el método particular MesAlta( ) de Administrativo, que no se encuentra en Empleado?, pues creando en la estructura Select...Case, un caso particular que compruebe cuándo estamos tratando con un objeto Administrativo. Veámoslo en el Código fuente 268.

```
Public Sub ManipularVarios(ByVal loUnObjeto As Object)
    ' obtenemos información sobre el tipo del objeto
    Dim loTipoObj As Type
    loTipoObj = loUnObjeto.GetType()

    ' comprobamos qué tipo de objeto es,
    ' y en función de eso, ejecutamos el
    ' método adecuado
    Select Case loTipoObj.Name
        Case "Empleado"
            CType(loUnObjeto, Empleado).VerDatos()

        Case "Administrativo" ' <-- añadimos este caso a la estructura
            CType(loUnObjeto, Administrativo).VerDatos()
            Console.WriteLine("El administrativo comenzó en {0}", _
                CType(loUnObjeto, Administrativo).MesAlta())

        Case "Factura"
            CType(loUnObjeto, Factura).Emitir()
    End Select
End Sub
```

Código fuente 268

Miembros compartidos (shared) de una clase

Los miembros compartidos o shared son aquellos que no precisan de una instancia previa de un objeto de la clase para poder ser utilizados, aunque pueden también ser usados por una instancia de la clase.

Dentro de este contexto, podemos pues clasificar los miembros de una clase en dos categorías:

- **Miembros de instancia (instance members)**. Son aquellos a los que accedemos a través de un objeto instanciado previamente de la clase.

- **Miembros compartidos (shared members)**. Son aquellos a los que podemos acceder sin necesidad de que exista un objeto creado de la clase.

Podemos declarar como compartidos los métodos, propiedades y campos de una clase. Para ello deberemos emplear la palabra clave Shared en la declaración.

Para utilizar desde el código cliente un miembro compartido, tan sólo debemos poner el nombre de la clase a la que pertenece, el punto y el nombre del miembro a utilizar.

El ejemplo del Código fuente 269 demuestra como podemos ejecutar un método compartido sin haber instanciado antes un objeto de la clase a la que pertenece dicho método.

```
Module General
    Sub Main()
        Dim lsValor As String
        ' aunque no hemos instanciado objetos
        ' de la clase Empleado, podemos llamar
        ' a este método compartido
        Console.WriteLine("Nombre del mes: {0}", Empleado.VerNombreMes())

        ' ahora creamos una instancia de la clase
        Dim loEmpleado1 As New Empleado()
        lsValor = loEmpleado1.VerNombreDia()
        Console.WriteLine("Nombre del día: {0}", lsValor)

        Console.ReadLine()
    End Sub
End Module

Public Class Empleado
    Public Shared Function VerNombreMes() As String
        ' este método puede ser llamado
        ' directamente empleando el nombre
        ' de la clase como calificador
        Return Format(Now(), "MMMM")
    End Function

    Public Function VerNombreDia() As String
        ' este método precisa de una instancia
        ' para ser llamado
        Return Format(Now(), "dddd")
    End Function
End Class
```

Código fuente 269

En el caso de variables de clase declaradas como miembros compartidos, este tipo de variable sólo es creado una vez, manteniendo su valor para todas las instancias de la clase. Esto contrasta con los miembros de instancia, de los que se crea una copia particular para cada objeto.

El efecto de miembro compartido se hace más patente cuando se aplica sobre variables, por ello, en el ejemplo del Código fuente 270, creamos dos campos compartidos para la clase Empleado; uno de ellos actuará como contador de los objetos creados de la clase, usando el método constructor para ser incrementado. El otro nos servirá para comprobar que siendo compartido no se inicializa, y mantiene el valor asignado previamente.

```
Module General
    Sub Main()
        ' accedemos a la variable compartida
        ' y le asignamos valor
        Empleado.psApellidos = "Naranjo"

        ' instanciamos un primer objeto Empleado
        Dim loEmp1 As New Empleado()
```

```
        ' asignamos valor a su variable de instancia
        loEmp1.psNombre = "Luis"
        ' mostramos las dos variables del objeto
        Console.WriteLine("Objeto loEmp1 - valores de sus variables")
        Console.WriteLine("psNombre: {0} - psApellidos: {1}", _
            loEmp1.psNombre, loEmp1.psApellidos)

        Console.WriteLine()

        ' instanciamos un segundo objeto Empleado
        Dim loEmp2 As New Empleado()
        ' asignamos valor a su variable de instancia
        loEmp2.psNombre = "Juan"
        ' mostramos las dos variables del objeto
        Console.WriteLine("Objeto loEmp2 - valores de sus variables")
        Console.WriteLine("psNombre: {0} - psApellidos: {1}", _
            loEmp2.psNombre, loEmp2.psApellidos)

        Console.WriteLine()
        ' ahora mostramos el valor de
        ' la variable compartida miContar
        Console.WriteLine("Se han instanciado {0} objetos de la clase Empleado", _
            Empleado.piContar)
        Console.ReadLine()
    End Sub
End Module

Public Class Empleado
    Public psNombre As String   ' miembro de instancia
    Public Shared psApellidos As String     ' miembro compartido
    Public Shared piContar As Integer   ' miembro compartido

    Public Sub New()
        ' por cada instancia de la clase creada,
        ' incrementar este campo compartido
        Me.piContar += 1
    End Sub
End Class
```

Código fuente 270

Definir una clase como punto de entrada de la aplicación

Podemos crear una clase en el proyecto, que contenga un método Main( ) declarado como Shared, de forma que dicho método constituya el punto del programa. Ver Código fuente 271.

```
Public Class Comienzo
    Public Shared Sub Main()
        Console.WriteLine("Está comenzando la aplicación")
        Console.ReadLine()
    End Sub
End Class
```

Código fuente 271

Además de crear la clase con este método, deberemos modificar las propiedades del proyecto, definiendo como objeto inicial el nombre de la clase o directamente Sub Main. Como habrá podido

adivinar el lector, ello hace innecesario el uso de módulos de código dentro del proyecto, pudiendo de esta manera, codificar la aplicación completamente basada en clases.

Como detalle interesante, debemos destacar el hecho de que al utilizar el modo tradicional de inicio de una aplicación, es decir, a través de un procedimiento Main( ) en un módulo, el compilador convierte internamente dicho módulo en una clase y al procedimiento en un método compartido.

Destrucción de objetos y recolección de basura

La destrucción de objetos y liberación de recursos en VB6 está basada en una técnica que mantiene un contador interno de las referencias que en el código cliente hay establecidas hacia un objeto; cuando la última referencia es eliminada, se ejecuta su evento Terminate( ), y el objeto se destruye. Este sistema de destrucción se denomina *finalización determinista*, ya que nos permite saber el momento preciso en el que un objeto es destruido.

A pesar de ser efectiva, la finalización determinista de VB6 tiene como problema principal el hecho de que si existen dos objetos que mantienen una referencia recíproca entre ellos, lo que se denomina una referencia circular, dichos objetos pueden quedar en memoria permanentemente durante toda la ejecución del programa, a pesar de no ser ya utilizados; este contratiempo se produce básicamente en la programación de componentes, cuando los objetos provienen de componentes distintos.

El esquema de destrucción de objetos en la plataforma .NET, para evitar las referencias circulares entre objetos, no utiliza conteo de referencias, sino que en su lugar, implementa un sistema de búsqueda y eliminación de objetos que ya no están siendo utilizados, denominado recolección de basura (garbage collection).

Cada ciertos periodos de tiempo dictados por el entorno, el recolector de basura se activa y realiza una exploración entre los objetos existentes, destruyendo aquellos que no están siendo utilizados. Este es un proceso controlado automáticamente por el entorno de .NET.

Dado que mediante la recolección de basura, no podemos predecir el momento exacto en el que el objeto es destruido, y la memoria que utiliza es liberada, a este modelo de gestión de recursos se le denomina *finalización no determinista*.

Además de evitar el problema de las referencias circulares, el recolector de basura está diseñado para realizar su trabajo en los momentos de menor actividad del sistema, para que el impacto en el rendimiento general sea el menor posible.

Como ya hemos mencionado, el recolector de basura funciona de modo automático; no obstante, si en un determinado momento queremos forzar su uso, la jerarquía de .NET nos provee de la clase GC (Garbage Collector), que representa al objeto recolector, pudiendo manipularlo a través de sus miembros compartidos.

Por ejemplo, si necesitamos que se efectúe la recolección de objetos que no estén siendo utilizados ejecutaríamos el método Collect( ) de este objeto como vemos en el Código fuente 272.

```
GC.Collect()
```

Código fuente 272

A pesar de tener acceso al recolector de basura del sistema, se recomienda no abusar de su utilización en nuestro código, ya que consume una importante cantidad de recursos cuando está en ejecución. Lo más adecuado es dejar esta labor en manos de la propia plataforma, que activará el proceso de recolección en los momentos más apropiados.

Cuando en VB6 asignamos Nothing a un objeto durante la ejecución, provocamos explícitamente la destrucción inmediata del objeto. Durante ese proceso se ejecuta su evento Terminate( ), en el que podemos escribir código para eliminar elementos que pudiera estar utilizando a su vez el objeto: cierre de conexiones a bases de datos, ficheros, etc. A este tipo de evento, que se ejecuta justo antes de la destrucción de un objeto se le denomina finalizador.

En .NET Framework los objetos también disponen de métodos finalizadores. Para implementar este tipo de miembro en nuestras clases, escribiremos un método con el nombre Finalize( ), de ámbito Protected, que sobrescriba y llame al finalizador de su clase padre. Podemos abrir la lista de métodos en el editor de código del IDE, en donde encontraremos este método preconstruido, que tendremos que completar con el código necesario. Ver el Código fuente 273.

```
Protected Overrides Sub Finalize()
    MyBase.Finalize()
    ' operaciones de finalización del objeto
    ' ....
End Sub
```

Código fuente 273

Sin embargo, debido al sistema de gestión de recursos que implementa la finalización no determinista, no podemos saber con tanta precisión el momento en el que un objeto es destruido.

Cuando asignamos Nothing a un objeto, o la variable que lo contiene pierde su ámbito y es eliminada, transcurre un tiempo que no es posible determinar hasta que el objeto es definitivamente destruido. Esta circunstancia hace que la eliminación de los elementos usados por el propio objeto pueda demorarse hasta varios minutos, con lo que una conexión a un origen de datos que realizara un excesivo consumo de recursos podría estar abierta un tiempo innecesario. El uso de un evento finalizador en esta situación no sería muy efectivo.

La solución a este problema pasa por crear un método, en el cuál realizaríamos las tareas de finalización que no fuera conveniente dejar en el método Finalize( ) de la clase. La denominación para este método puede elegirla el lector libremente, pero por convención, se recomienda asignarle el nombre Dispose( ), ya que este es el nombre que utilizan el resto de clases de la plataforma. Ver el Código fuente 274.

```
Module Module1
    Sub Main()
        Dim loEmple As Empleado
        loEmple = New Empleado()
        ' ....
        ' ....
        loEmple.Dispose()
        loEmple = Nothing
        ' a partir de aquí, en cualquier momento el entorno
        ' activará el recolector de basura, que ejecutará
        ' el evento Finalize() y destruirá el objeto
        ' ....
        ' ....
```

```
    End Sub
End Module

Public Class Empleado
    ' ....
    ' ....
    Public Sub Dispose()
        ' en este método escribiremos las tareas
        ' de finalización más urgentes:
        ' cierre de conexiones, manipuladores de ficheros, etc
        ' que no sea conveniente dejar en Finalize()
        ' ....
    End Sub
End Class
```

Código fuente 274

Interfaces

Un interfaz proporciona, a modo de declaración, una lista de propiedades y métodos, que posteriormente serán codificados en una o varias clases.

Debido a su naturaleza declarativa, un interfaz no contiene el código de los miembros que expresa; dicho código será escrito en las clases que implementen el interfaz.

El concepto de interfaz es análogo al de contrato, las partes integrantes son el propio interfaz y la clase que lo implementa. Mientras que el interfaz no puede ser cambiado desde el momento en que sea implementado, la clase que lo implementa se compromete a crear la lista de miembros en la misma forma que indica el interfaz

Los interfaces nos permiten definir conjuntos reducidos de funcionalidades, constituyendo una útil herramienta de cara al polimorfismo. El mismo interfaz, implementado en distintas clases, podrá tener a su vez código distinto, con lo que los objetos de diferentes clases que implementen un interfaz común, pueden tener un comportamiento diferente.

Supongamos por ejemplo, que necesitamos que algunas clases dispongan de ciertos miembros para la manipulación de cadenas, pero no queremos que estas características sean algo rígido, es decir, cada clase debe de cumplir con un conjunto de nombres y parámetros para los miembros que se encargarán de manipular cadenas, pero la implementación del código que haga cada clase para gestionarlas es libre.

Ante esta situación podemos definir un interfaz e implementarlo en cada clase. El ejemplo desarrollado al completo para este caso se encuentra en el proyecto InterfacesPrueba (hacer clic aquí para acceder al ejemplo).

En primer lugar crearemos un nuevo proyecto de tipo aplicación de consola. A continuación, para crear un interfaz, utilizaremos la palabra clave Interface junto con el nombre que asignemos al interfaz. Para indicar el final del interfaz usaremos la palabra clave End Interface, situando dentro del interfaz las declaraciones de miembros que necesitemos. En nuestro ejemplo vamos a crear el interfaz ICadena que declara la propiedad Longitud y el método ObtenerValor. Aunque no es en absoluto necesario, se recomienda que utilicemos la letra I como prefijo para los nombres de interfaces, de cara a facilitar la lectura del código, como vemos en el Código fuente 275.

```
' las clases que implementen este interfaz
```

```
' deberán crear la propiedad Longitud y el
' método ObtenerValor(); la codificación de
' dichos miembros será particular a cada clase
Public Interface ICadena
    ReadOnly Property Longitud() As Integer
    Function ObtenerValor() As String
End Interface
```

Código fuente 275

Seguidamente crearemos la clase Empleado. Para que dicha clase pueda implementar (utilizar) las definiciones de un interfaz, emplearemos después de la declaración de la clase, la palabra clave Implements junto al nombre del interfaz que deseamos que implemente la clase. Veamos el Código fuente 276.

```
Public Class Empleado
    Implements Cadena

       '....
       '....
End Class
```

Código fuente 276

Esta acción obligará a la clase a crear, como mínimo, tantos miembros como indica el interfaz que implementa, es decir, debemos escribir una propiedad Longitud y un método ObtenerValor( ), o en caso contrario, se producirá un error al intentar ejecutar el programa. Observe el lector, que el editor de código sitúa una marca sobre el nombre del interfaz en la clase mientras que no se hayan implementado todos sus miembros. Ver Figura 202.

```
Public Class Empleado
    Implements Cadena
```

El tipo 'InterfacesPrueba.Empleado' debe implementar 'Function ObtenerValor() As String' para la interfaz 'InterfacesPrueba.Cadena'.

```
    Private msNomb
```

Figura 202. Mensaje del interfaz que indica que no se han implementado todos sus miembros.

Para implementar un miembro de un interfaz, en el momento de escribir su declaración, utilizaremos la palabra clave Implements, seguida del interfaz y miembro que implementa. Veamos en el Código fuente 277, la implementación del método ObtenerValor( ) en la clase Empleado, con su código correspodiente.

```
Public Function ObtenerValor() As String Implements ICadena.ObtenerValor
    Return UCase(Me.Nombre)
End Function
```

Código fuente 277

La necesidad de especificar el interfaz y miembro del mismo que implementa, tiene como ventaja, el que para el nombre del método podemos utilizar uno distinto del que indica el interfaz. Por ejemplo, el anterior método podríamos haberlo escrito como muestra el Código fuente 278.

```
Public Function DameDatos() As String Implements ICadena.ObtenerValor
    Return UCase(Me.Nombre)
End Function
```

Código fuente 278

Un medio muy sencillo de crear un método vacío que implemente un interfaz, consiste en abrir la lista Nombre de clase del editor de código y allí, seleccionar en la clase, el interfaz que implementa. Ver Figura 203.

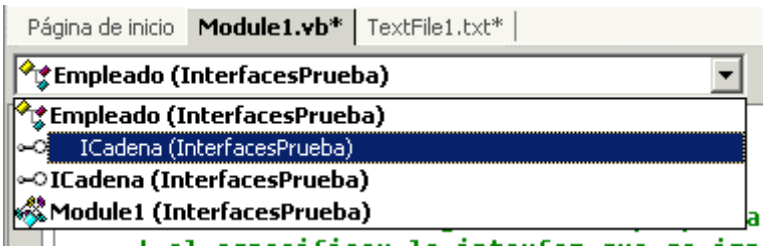


Figura 203. Selección del interfaz implementado por una clase en el editor de código.

Después pasaremos a la lista Nombre de método y elegiremos el miembro a implementar. Ver Figura 204.

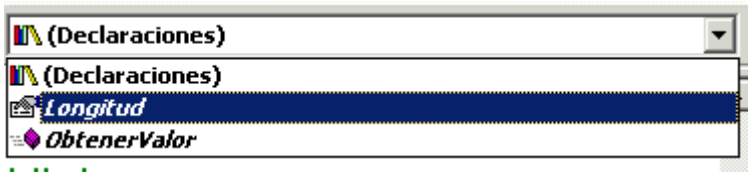


Figura 204. Selección del miembro del interfaz a implementar.

Estas acciones crearán el método vacío con la implementación del interfaz. Como podemos observar en el Código fuente 279, en la declaración del método se incluye el nombre calificado al completo.

```
Public ReadOnly Property Longitud() As Integer _
    Implements InterfacesPrueba.ICadena.Longitud
    Get

    End Get
End Property
```

Código fuente 279

En la implementación del interfaz ICadena para la clase Empleado, devolvemos el nombre en mayúsculas del empleado y la longitud de dicho nombre en los dos miembros correspondientes a dicho interfaz.

Naturalmente, aparte de los miembros del interfaz, una clase puede tener todos los demás que necesite.

Posteriormente creamos una segunda clase en nuestro proyecto con el nombre Cuenta, en la que también implementamos el interfaz ICadena, pero en los miembros implementados sobre esta clase las operaciones realizadas no serán exactamente iguales, ya que como hemos indicado, la implementación que hagamos de los miembros de un interfaz en una clase es totalmente libre para el programador.

El Código fuente 280 muestra el código al completo de este ejemplo, incluyendo el interfaz, las clases que lo implementan y el procedimiento Main( ) que instancia objetos de las clases.

```
Module Module1
    Sub Main()
        Dim loEmple As Empleado = New Empleado()
        loEmple.Nombre = "Raquel Rodrigo"
        Console.WriteLine("El nombre del empleado tiene {0} caracteres", _
            loEmple.Longitud)
        Console.WriteLine("Valor del empleado: {0}", loEmple.ObtenerValor())

        Dim loCuenta As Cuenta = New Cuenta()
        loCuenta.Codigo = 700256
        Console.WriteLine("El código de cuenta {0} tiene una longitud de {1}", _
            loCuenta.Codigo, loCuenta.Longitud)
        Console.WriteLine("Información de la cuenta: {0}", loCuenta.ObtenerValor())

        Console.Read()
    End Sub
End Module

' las clases que implementen este interfaz
' deberán crear la propiedad Longitud y el
' método ObtenerValor(); la codificación de
' dichos miembros será particular a cada clase
Public Interface ICadena
    ReadOnly Property Longitud() As Integer
    Function ObtenerValor() As String
End Interface

Public Class Empleado
    Implements ICadena

    Private msNombre As String

    Public Property Nombre() As String
        Get
            Return msNombre
        End Get
        Set(ByVal Value As String)
            msNombre = Value
        End Set
    End Property

    ' devolvemos la longitud de la propiedad Nombre,
    ' al especificar el interfaz que se implementa,
    ' se puede poner todo el espacio de nombres
    Public ReadOnly Property Longitud() As Integer _
        Implements InterfacesPrueba.ICadena.Longitud
        Get
            Return Len(Me.Nombre)
        End Get
    End Property

    ' devolvemos el nombre en mayúscula;
    ' no es necesario poner todo el espacio
    ' de nombres calificados, basta con el
    ' nombre de interfaz y el miembro a implementar
```

```
    Public Function ObtenerValor() As String Implements ICadena.ObtenerValor
        Return UCase(Me.Nombre)
    End Function
End Class

Public Class Cuenta
    Implements ICadena

    Private miCodigo As Integer

    Public Property Codigo() As Integer
        Get
            Return miCodigo
        End Get
        Set(ByVal Value As Integer)
            miCodigo = Value
        End Set
    End Property

    ' en esta implementación del miembro, devolvemos
    ' el número de caracteres que tiene el campo
    ' miCodigo de la clase
    Public ReadOnly Property Longitud() As Integer _
        Implements InterfacesPrueba.ICadena.Longitud
        Get
            Return Len(CStr(miCodigo))
        End Get
    End Property

    ' en este método del interfaz, devolvemos valores
    ' diferentes en función del contenido de la
    ' propiedad Codigo
    Public Function ObtenerValor() As String _
        Implements InterfacesPrueba.ICadena.ObtenerValor

        Select Case Me.Codigo
            Case 0 To 1000
                Return "Código en intervalo hasta 1000"
            Case 1001 To 2000
                Return "El código es: " & Me.Codigo
            Case Else
                Return "El código no está en el intervalo"
        End Select
    End Function
End Class
```

Código fuente 280

Los interfaces pueden heredar de otros interfaces que hayamos creado. De esta manera, si creamos el interfaz IGestion, podemos hacer que herede las características de ICadena y agregue más declaraciones de miembros. Ver Código fuente 281.

```
Public Interface IGestion
    Inherits ICadena

    Sub Calcular()
End Interface
```

Código fuente 281

También es posible en un interfaz heredar de múltiples interfaces. Ver el Código fuente 282.

```
Public Interface IGestion
    Inherits ICadena, ICalculo

End Interface
```

Código fuente 282

Estructuras

Una estructura consiste en un conjunto de datos que se unen para formar un tipo de dato compuesto. Este elemento del lenguaje se conocía en versiones anteriores de VB como tipo definido por el usuario (UDT o User Defined Type), y nos permite agrupar bajo un único identificador, una serie de datos relacionados.

Como novedad en VB.NET, los miembros de una estructura pueden ser, además de los propios campos que almacenan los valores, métodos que ejecuten operaciones, por lo cual, su aspecto y modo de manejo es muy parecido al de una clase.

Por ejemplo, si disponemos de la información bancaria de una persona, como pueda ser su código de cuenta, titular, saldo, etc., podemos manejar dichos datos mediante variables aisladas, o podemos crear una estructura que contenga toda esa información, simplificando la forma en que accedemos a tales datos.

Creación y manipulación de estructuras

Para crear una estructura, tenemos que utilizar la palabra clave Structure junto al nombre de la estructura, situando a continuación los miembros de la estructura, y finalizándola con la palabra clave End Structure, como muestra el Código fuente 283.

```
Public Structure DatosBanco
    Public IDCuenta As Integer
    Public Titular As String
    Public Saldo As Integer
End Structure
```

Código fuente 283

El modo de utilizar una estructura desde el código cliente, consiste en declarar una variable del tipo correspondiente a la estructura, y manipular sus miembros de forma similar a un objeto. En el Código fuente 284, manejamos de esta forma, una variable de la estructura DatosBanco.

```
Sub Main()
    Dim lDBanco As DatosBanco
    lDBanco.IDCuenta = 958
    lDBanco.Titular = "Carlos Perea"
    lDBanco.Saldo = 900
```

```
End Sub
```

Código fuente 284

Como hemos comentado antes, una estructura admite también métodos y propiedades, de instancia y compartidos, al estilo de una clase. Por lo que podemos añadir este tipo de elementos a nuestra estructura, para dotarla de mayor funcionalidad. El Código fuente 285 muestra un ejemplo más completo de la estructura DatosBanco.

```
Module Module1
    Sub Main()
        ' declarar una variable de la estructura
        ' y manipularla directamente
        Dim lDBanco1 As DatosBanco
        lDBanco1.IDCuenta = 958
        lDBanco1.Titular = "Carlos Perea"
        lDBanco1.DNI = "112233"
        lDBanco1.Saldo = 900
        lDBanco1.Informacion()

        Console.WriteLine()

        ' declaramos una variable de la estructura
        ' pero aquí la instanciamos antes de
        ' comenzar a trabajar con ella, para que
        ' se ejecute su método constructor
        Dim lDBanco2 As DatosBanco
        lDBanco2 = New DatosBanco(450)
        lDBanco2.IDCuenta = 580
        lDBanco2.Titular = "Alfredo Peral"
        lDBanco2.DNI = "556677"
        lDBanco2.Informacion()

        Console.WriteLine()

        ' en esta ocasión trabajamos con los
        ' miembros compartidos de la estructura,
        ' por lo que no es necesario una variable
        ' con el tipo de la estructura, sino que
        ' usamos directamente los miembros compartidos
        Console.WriteLine("La fecha de apertura de la cuenta es {0}", _
            DatosBanco.FHApertura)
        DatosBanco.SaldoMin()

        Console.ReadLine()
    End Sub
End Module

Public Structure DatosBanco
    Public IDCuenta As Integer
    Public Titular As String
    Public Saldo As Integer
    Private mDNI As String
    Shared FHApertura As Date
    Shared SaldoMinimo As Integer

    ' el constructor de una estructura debe definirse con
    ' parámetros; si no ponemos parámetros hay que declararlo
    ' como compartido
    Public Sub New(ByVal liSaldo As Integer)
        Saldo = liSaldo
    End Sub
```

```
    Public Property DNI() As String
        Get
            Return mDNI
        End Get
        Set(ByVal Value As String)
            mDNI = Value
        End Set
    End Property

    Public Sub Informacion()
        Console.WriteLine("Información sobre la cuenta")
        Console.WriteLine("Código: {0} - Titular: {1} - DNI: {2} - Saldo: {3}", _
            IDCuenta, Titular, Me.DNI, Saldo)
    End Sub

    ' miembros compartidos de la estructura
    ' ====================================
    ' si el constructor no tiene parámetros
    ' debe ser compartido (shared), y además,
    ' los miembros que maneje deben ser también compartidos
    Shared Sub New()
        FHApertura = Now
        SaldoMinimo = 200
    End Sub

    ' método compartido que devuelve el saldo mínimo
    Shared Sub SaldoMin()
        Console.WriteLine("El saldo mínimo necesario debe ser de {0}", _
            SaldoMinimo)
    End Sub
End Structure
```

Código fuente 285

Estructuras o clases, ¿cuál debemos utilizar?

Llegados a este punto, el lector puede estar preguntándose cuál sería la mejor manera de abordar la aplicación: con clases o estructuras. Bien, no podemos decantarnos totalmente por un modo u otro de trabajo. Aún guardando muchas similitudes con las clases, las estructuras mantienen ciertas diferencias respecto a las primeras, que harán conveniente su uso en determinados momentos, siendo menos adecuadas en otras situaciones.

Las estructuras no soportan herencia, por lo que el medio más parecido que tenemos de extender sus funcionalidades es a través de interfaces.

Para crear un manejador de evento dentro de una estructura, el procedimiento que actúe como manejador deberá ser un miembro compartido, no podrá ser un miembro de instancia. El manejo de eventos será tratado posteriormente.

Una estructura es un tipo por valor, lo que quiere decir que los datos que contiene se manejan en la pila (stack) de la memoria. Si asignamos una variable que contiene una estructura a otra variable, se realizará una copia de la estructura, y obtendremos dos estructuras cuyos datos serán totalmente independientes.

Esto último contrasta claramente con las clases, que son tipos por referencia, y sus datos se manejan en el montón (heap) de la memoria. Lo que realmente contiene una variable de objeto no es el objeto en sí, sino un puntero de cuatro bytes, con la referencia hacia la zona de memoria en la que reside el objeto. Por lo tanto, si asignamos una variable de objeto a otra variable, se realiza lo que se denomina

una copia superficial (shallow copy) de una variable a otra. Esto quiere decir que sólo se copia el puntero de cuatro bytes que contiene la referencia hacia el objeto. El efecto conseguido son dos variables que apuntan al mismo objeto y *no* dos variables con copias independientes del objeto.

Observemos el Código fuente 286, en el que se crean dos variables de estructura y una se asigna a otra. Si hacemos un cambio en la segunda, la primera estructura permanecerá inalterada. Sin embargo, a continuación creamos dos variables de objeto, y asignamos la primera a la segunda. Cuando hagamos un cambio en la segunda, se reflejará en la primera; esto es debido a que son dos variables que apuntan al mismo objeto.

```
Sub Main()
    Dim lDBanco1 As DatosBanco
    Dim lDBanco2 As DatosBanco

    lDBanco1.IDCuenta = 55
    lDBanco2 = lDBanco1
    lDBanco2.IDCuenta = 188

    Console.WriteLine("lDBanco1.IDCuenta --> {0}", lDBanco1.IDCuenta)   ' 55
    Console.WriteLine("lDBanco2.IDCuenta --> {0}", lDBanco2.IDCuenta)   ' 188

    Console.WriteLine()

    Dim loEmpleado1 As Empleado
    Dim loEmpleado2 As Empleado

    loEmpleado1 = New Empleado()
    loEmpleado1.piIdentificador = 55
    loEmpleado2 = loEmpleado1
    loEmpleado2.piIdentificador = 188

    Console.WriteLine("loEmpleado1.piIdentificador --> {0}", _
        loEmpleado1.piIdentificador)    ' 188
    Console.WriteLine("loEmpleado2.piIdentificador --> {0}", _
        loEmpleado2.piIdentificador)    ' 188

    Console.ReadLine()
End Sub
```

Código fuente 286

Siguiendo con este mismo aspecto del manejo en memoria de las estructuras , queremos hacer notar al lector que al tratarse de tipos por valor, podemos realizar sobre ellas operaciones de embalaje y desembalaje de tipos (boxing - unboxing). Consulte el lector en el tema dedicado a .NET Framework, el apartado dedicado a este tipo de operaciones.

Si embalamos una estructura, asignándola a una variable tipificada como Object, o pasándola a un procedimiento que tiene el parámetro declarado como Object, y posteriormente la desembalamos, volviéndola a asignar a una variable del tipo de la estructura, incurriremos en una penalización sobre el rendimiento, debido a que el CLR debe manipular la variable en el montón cuando es tratada como un objeto y después devolverla a la pila cuando se vuelve a tratar como una estructura. Por consiguiente, si vamos a tratar la estructura mayormente como un objeto, debemos considerar la posibilidad de crear mejor una clase.

La estructura del sistema DateTime

El entorno de .NET Framework proporciona, al igual que ocurre con las clases, una serie de estructuras del sistema, con funcionalidades diseñadas para ayudar al programador en las más variadas situaciones.

Como ejemplo de este tipo de estructura encontramos a DateTime, en la que a través de sus miembros compartidos y de instancia, nos provee de diversas operaciones para el manejo de fechas.

El Código fuente 287 muestra algunos ejemplos de uso de esta estructura. Consulte el lector la documentación de la plataforma para una descripción detallada sobre cada uno de sus miembros.

```
Module Module1
    Sub Main()
        ' ejemplos con la estructura DateTime
        ' ====================================
        ' miembros compartidos
        Dim ldtFechaActual As Date
        Dim ldtFechaA, ldtFechaB As Date

        ' la propiedad Today devuelve la fecha actual
        ldtFechaActual = DateTime.Today
        Console.WriteLine("La fecha de hoy es {0}", ldtFechaActual)

        ' el método DaysInMonth() devuelve el número
        ' de días que tiene un mes
        Console.WriteLine("El mes de Febrero de 2002 tiene {0} días", _
            DateTime.DaysInMonth(2002, 2))

        ' el método Compare() compara dos fechas
        Console.WriteLine("Introducir primera fecha")
        ldtFechaA = Console.ReadLine()
        Console.WriteLine("Introducir segunda fecha")
        ldtFechaB = Console.ReadLine()

        Select Case DateTime.Compare(ldtFechaA, ldtFechaB)
            Case -1
                Console.WriteLine("La primera fecha es menor")
            Case 0
                Console.WriteLine("Las fechas son iguales")
            Case 1
                Console.WriteLine("La primera fecha es mayor")
        End Select

        ' miembros de instancia
        Dim loMiFecha As DateTime
        Dim ldtFDias As Date
        Dim ldtFMeses As Date
        Dim lsFHFormateada As String

        ' usar el constructor de la estructura
        ' para crear una fecha
        loMiFecha = New DateTime(2002, 5, 12)
        ' agregar días a la fecha
        ldtFDias = loMiFecha.AddDays(36)
        ' restar meses a la fecha
        ldtFMeses = loMiFecha.AddMonths(-7)
        ' formatear la fecha
        lsFHFormateada = loMiFecha.ToLongDateString()

        Console.WriteLine("Uso de métodos de instancia de DateTime")
        Console.WriteLine("Fecha creada: {0} - Agregar días: {1}" & _
```

```
            " - Restar meses: {2} - Fecha formateada: {3}", _
            loMiFecha, ldtFDias, ldtFMeses, lsFHFormateada)
        Console.ReadLine()
    End Sub
End Module
```

Código fuente 287

Enumeraciones

Una enumeración consiste en un conjunto de constantes relacionadas. A cada constante se le asigna un nombre, mientras que la agrupación de tales constantes, es decir, la propia enumeración recibe también un nombre identificativo.

Supongamos por ejemplo, que en un programa debemos realizar clasificaciones por estilos musicales: Rock, Blues, New Age, Funky, etc. El modo más sencillo de manipular estos valores en el código es identificarlos mediante números, de forma que cuando en un procedimiento tengamos que saber la selección del estilo que ha realizado un usuario, empleemos el número identificativo en lugar de cadenas de caracteres. Veamos el Código fuente 288.

```
Public Sub SelecEstilo(ByVal liEstiloMusical As Integer)
    Select Case liEstiloMusical
        Case 1
            ' se la seleccionado Rock
            ' ....
        Case 2
            ' se la seleccionado Blues
            ' ....
        Case 3
            ' se la seleccionado New Age
            ' ....
        Case 4
            ' se la seleccionado Funky
            ' ....
    End Select
End Sub
```

Código fuente 288

Sin embargo, la sencillez que supone el uso de números para identificar determinadas características en nuestra aplicación, tiene el efecto adverso de dificultar la lectura del código, ya que, en todos los procedimientos donde debamos trabajar con estilos musicales, deberemos de añadir un comentario de código junto al número de estilo, para saber de cuál se trata.

Podemos solucionar parcialmente el problema utilizando constantes, de modo que por cada estilo musical, crearemos una constante a la que asignaremos el número de un estilo. Veamos el Código fuente 289.

```
Public Const ROCK As Integer = 1
Public Const BLUES As Integer = 2
Public Const NEWAGE As Integer = 3
Public Const FUNKY As Integer = 4

Public Sub SelecEstilo(ByVal liEstiloMusical As Integer)
```

```
    Select Case liEstiloMusical
        Case ROCK
            ' ....
        Case BLUES
            ' ....
        Case NEWAGE
            ' ....
        Case FUNKY
            ' ....
    End Select
End Sub
```

Código fuente 289

Si bien el uso de constantes mejora la situación, su proliferación provocará la aparición de un nuevo problema: la organización y clasificación de todas las constantes del programa.

Aquí es donde entran en escena las enumeraciones, ya que con ellas, podemos crear conjuntos de constantes relacionadas por una cualidad común, agrupando cada conjunto bajo un identificador genérico.

Para crear una enumeración debemos utilizar las palabras clave Enum...End Enum, situando junto a Enum el nombre que vamos a dar a la enumeración, y a continuación, la lista de constantes que agrupará. Por lo tanto, si queremos reunir bajo una enumeración, las constantes de los estilos musicales, lo haremos del modo mostrado en el Código fuente 290.

```
Public Enum Musicas
    Rock
    Blues
    NewAge
    Funky
End Enum
```

Código fuente 290

Una enumeración debe tener un tipo de dato. Los tipos que podemos asignar a una enumeración deben ser los numéricos enteros soportados por el lenguaje que estemos utilizando. En el caso de VB.NET, los tipos de datos admisibles son Byte, Integer, Long y Short. En el caso de que no especifiquemos el tipo, tomará Integer por defecto.

El hecho de tipificar una enumeración está relacionado con los valores que podemos asignar a cada una de las constantes que contiene. De ello se deduce, que sólo vamos a poder asignar valores numéricos a estas constantes.

Cuando creamos una enumeración, si no asignamos valores a sus constantes, el entorno asigna automáticamente los valores, comenzando por cero y en incrementos de uno. Podemos en cualquier momento, asignar manualmente valores, no siendo obligatorio tener que asignar a todas las constantes. Cuando dejemos de asignar valores, el entorno seguirá asignando los valores utilizando como valor de continuación, el de la última constante asignada. Veamos unos ejemplos en el Código fuente 291.

```
Public Enum Musicas As Integer
    Rock    ' 0
    Blues   ' 1
```

```
    NewAge   ' 2
    Funky    ' 3
End Enum

Public Enum DiasSemana As Integer
    Lunes              ' 0
    Martes             ' 1
    Miercoles = 278
    Jueves             ' 279
    Viernes            ' 280
    Sabado             ' 281
    Domingo            ' 282
End Enum
```

Código fuente 291

Para utilizar una enumeración definida en nuestra aplicación, debemos declarar una variable, a la que daremos como tipo de dato el mismo de la enumeración. Una vez creada, la forma de asignar un valor es muy sencilla, ya que en cuanto escribamos el operador de asignación, el editor de código nos abrirá una lista con los posibles valores que admite la variable, que corresponderán, evidentemente, sólo a los de la enumeración. De esta forma, facilitamos enormemente la escritura de código, ya que se reducen las posibilidades de error en la asignación de valores a la variable enumerada. Ver Figura 205.

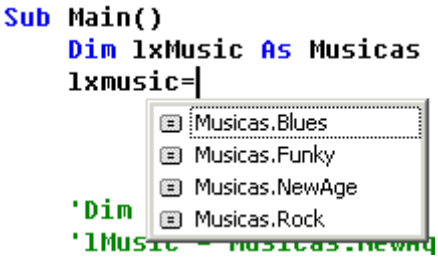


Figura 205. Asignación de valor a una variable de tipo enumerado.

El valor almacenado en una variable de enumeración corresponderá al número de la constante que hayamos seleccionado. Al declarar la variable, su valor inicial será cero.

No obstante, la manipulación de una enumeración va mucho más allá de la asignación y recuperación simple de las constantes que componen la enumeración. Cuando declaramos una variable de una enumeración, el contenido real de dicha variable es un objeto de la clase Enum; por lo tanto, podemos utilizar los métodos de dicho objeto, para realizar diversas operaciones. Para tareas genéricas, la clase Enum también dispone de métodos compartidos que podremos ejecutar directamente, sin necesidad de crear una enumeración previa. El Código fuente 292 muestra algunos ejemplos.

```
Module Module1
    Public Enum Musicas As Integer
        Rock
        Blues
        NewAge
        Funky
    End Enum

    Sub Main()
        ' creamos una variable de enumeración
        ' y le asignamos valor
        Dim lxMusic As Musicas
```

```
        lxMusic = Musicas.NewAge

        ' el contenido de la variable es el número asignado
        ' a la constante
        Console.WriteLine(lxMusic)

        ' el método ToString() permite mostrar el nombre
        ' de la constante elegida de la lista que tiene
        ' la enumeración
        Console.WriteLine(lxMusic.ToString("G"))

        ' obtener los valores de las constantes de la enumeración
        ' con GetValues(), y los nombres con GetNames(); estos métodos
        ' son compartidos de la clase Enum, y reciben como parámetro una
        ' variable de enumeración de la que debe obtenerse el tipo ejecutando
        ' su método GetType(); devuelven un array con los datos pedidos
        Dim liValores() As Integer
        Dim lsNombres() As String
        Dim liContador As Integer

        liValores = System.Enum.GetValues(lxMusic.GetType())
        lsNombres = System.Enum.GetNames(lxMusic.GetType())
        Console.WriteLine()
        Console.WriteLine("Valor - Nombre")

        ' recorrer los arrays y mostrar su contenido
        For liContador = 0 To UBound(liValores)
            Console.WriteLine(liValores(liContador) & Space(7) & _
                lsNombres(liContador))
        Next

        ' comprobar si un nombre introducido por el
        ' usuario está entre los nombres de las
        ' constantes en la enumeración
        Dim lsNombreMusica As String
        Console.WriteLine("Introducir nombre de constante")
        lsNombreMusica = Console.ReadLine()

        If System.Enum.IsDefined(lxMusic.GetType(), lsNombreMusica) Then
            Console.WriteLine("El tipo de música sí está en la enumeración")
        Else
            Console.WriteLine("El tipo de música no está en la enumeración")
        End If

        Console.ReadLine()
    End Sub
End Module
```

Código fuente 292.

21

Aplicando un enfoque enteramente OOP en el código

Los tipos de datos también son objetos

En anteriores versiones de Visual Basic, cuando necesitábamos desarrollar un proceso que implicara la manipulación de cadenas de caracteres, fechas, cálculos aritméticos, etc., debíamos recurrir a un conjunto de funciones de soporte existentes en el lenguaje, para obtener entre otras, la longitud de una cadena, subcadenas, conversiones, formatos, fecha actual, partes de una fecha, etc. El Código fuente 293, muestra cómo obtenemos la longitud de una cadena con la función Len( ).

```
Dim lsCadena As String
Dim liLongitud As Integer
lsCadena = "esto es una prueba"
liLongitud = Len(lsCadena)  ' 18
```

Código fuente 293

La mayoría de estas funciones del lenguaje siguen estando disponibles (el fuente anterior funciona perfectamente en VB.NET), pero a lo largo de los siguientes apartados, comprobaremos que su uso será cada vez menos necesario.

Como ya apuntábamos en el tema dedicado a la plataforma .NET Framework, todos los elementos del lenguaje se consideran tipos: los propios tipos de datos, clases, estructuras, enumeraciones, etc.,

componen lo que se denomina el CTS, o sistema común de tipos; una enorme jerarquía que parte del tipo base Object, y del que heredan el resto de tipos de la plataforma.

Al ser los tipos de datos, uno de los muchos tipos existentes dentro del esquema del CTS, podemos manipularlos de la forma tradicional o como si fueran objetos; aspecto este, que trataremos en el siguiente apartado.

Manipulación de cadenas con la clase String

La clase String nos provee de un amplio abanico de métodos, para realizar todas las operaciones que en anteriores versiones del lenguaje debíamos codificar a través de funciones.

Tomemos el que podría ser el ejemplo más significativo: el tipo String. Observemos el Código fuente 294, y comparemos con el fuente del ejemplo anterior.

```
Dim lsCadena As String
Dim liLongitud As Integer
Dim lsCambiada As String
lsCadena = "esto es una prueba"
liLongitud = lsCadena.Length    ' 18
lsCambiada = lsCadena.ToUpper()   ' ESTO ES UNA PRUEBA
```

Código fuente 294

Al ser una cadena, tanto un tipo de dato como un objeto de la clase String, podemos manipularlo como cualquier otro objeto de la jerarquía de la plataforma. En esta ocasión, hemos recuperado la longitud de la cadena mediante su propiedad Length, y la hemos convertido a mayúsculas ejecutando su método ToUpper( ), asignado el resultado a otra variable.

Para comprobar la versatilidad que ahora nos proporcionan los tipos de datos, cuando declaramos una variable String, podemos hacerlo como en versiones anteriores del lenguaje, o al estilo OOP. Si consultamos la ayuda de .NET Framework, encontraremos una entrada con el título String Class, que describe este tipo como una clase más del sistema. Veamos el Código fuente 295.

```
Sub Main()
    ' modo tradicional
    Dim lsCad1 As String
    lsCad1 = "mesa"

    ' instanciar un objeto String y asignar valor
    Dim loCad2 As New String("silla")

    ' declarar variable e instanciar un objeto
    ' String en la misma línea
    Dim loCad3 As String = New String("coche")

    ' declarar variable e instanciar un objeto
    ' String en la misma línea; el constructor
    ' utilizado en este caso requiere un array
    ' de objetos Char; observe el lector la forma
    ' de crear un array, asignando valores al
    ' mismo tiempo
    Dim loCad4 As String = New String(New Char() {"t", "r", "e", "n"})
```

```
    Console.WriteLine("lsCad1 --> {0}", lsCad1)
    Console.WriteLine("loCad2 --> {0}", loCad2)
    Console.WriteLine("loCad3 --> {0}", loCad3)
    Console.WriteLine("loCad4 --> {0}", loCad4)

    Console.ReadLine()
End Sub
```

Código fuente 295

Una vez visto el fuente anterior, debemos realizar algunas aclaraciones.

Como ya adelantábamos en el tema sobre .NET Framework, y podemos comprobar utilizando el constructor de la clase String que recibe como parámetro un array Char; el tipo String no pertenece puramente al conjunto de tipos primitivos de la plataforma, ya que internamente, el entorno manipula una cadena como un array de tipos Char; aunque para nuestra comodidad, este es un proceso transparente, que gestiona la plataforma por nosotros.

En segundo lugar, y este también es un trabajo realizado transparentemente por el entorno, cada vez que creamos o instanciamos un tipo String, obtenemos lo que se denomina una cadena inalterable. Internamente, cuando realizamos una operación sobre la cadena: convertirla a mayúsculas, extraer una subcadena, etc., el CLR crea una nueva instancia de String, asignándola a la misma variable. En apariencia, realizamos modificaciones sobre la misma cadena, pero en realidad, cada operación genera nuevos objetos String.

Por último, no es ahora posible crear cadenas de longitud fija, como ocurría en versiones anteriores de VB.

En este apartado realizaremos una revisión de los métodos de esta clase, a través de un conjunto de ejemplos, que a modo ilustrativo, nos permitan familiarizarnos con el modo en que se manejan cadenas en VB.NET.

Debido al elevado número de miembros que contienen la mayoría de los tipos de la plataforma .NET, tanto clases, como estructuras, tipos de datos, etc,; y a que muchos de ellos disponen de versiones sobrecargadas; en la descripción de cada tipo haremos un repaso de sus miembros principales, remitiendo al lector, a la documentación de referencia que sobre los tipos existe en la ayuda de la plataforma .NET, en donde encontrará toda la información detallada.

Antes de comenzar a describir los métodos de esta clase, y puesto que una cadena es un array de tipos Char, es importante tener en cuenta que la primera posición corresponde al cero. Esta aclaración la realizamos fundamentalmente, de cara a los métodos que requieran el manejo de posiciones concretas de la cadena.

- **Trim( ), TrimStart( ), TrimEnd( )**. Eliminan los espacios a ambos lados de una cadena, al comienzo o al final. Ver el Código fuente 296.

```
Dim lsCadena As String
lsCadena = "    Hola .NET  "

Dim lsQuitar As String
lsQuitar = lsCadena.TrimEnd()    ' "    Hola .NET"
lsQuitar = lsCadena.TrimStart() ' "Hola .NET  "
lsQuitar = lsCadena.Trim()      ' "Hola .NET"
```

Código fuente 296

- **PadLeft( ), PadRight( )**. Rellenan una cadena por la izquierda o derecha utilizando un determinado carácter de relleno. Debemos especificar la longitud total de la cadena resultante. Como el carácter de relleno es un tipo Char, podemos especificar que se trata de este tipo, situando junto al carácter de relleno, la letra c. Ver el Código fuente 297.

```
Dim lsCadena As String
Dim lsRellena As String
lsCadena = "Hola"
lsRellena = lsCadena.PadLeft(10)        ' "      Hola"
lsRellena = lsCadena.PadRight(10, "W"c) ' "HolaWWWWWW"
```

Código fuente 297

- **Insert( )**. Inserta en la cadena, una subcadena a partir de una posición determinada. Ver el Código fuente 298.

```
Dim lsCadena As String
Dim lsAgregar As String
lsCadena = "Estamos programando"
lsAgregar = lsCadena.Insert(2, "HOLA")  ' "EsHOLAtamos programando"
```

Código fuente 298

- **Remove( )**. Elimina de la cadena un número determinado de caracteres, comenzando por una posición específica. Ver el Código fuente 299.

```
Dim lsCadena As String
Dim lsQuitar As String
lsCadena = "Estamos programando"
lsQuitar = lsCadena.Remove(5, 3)    ' "Estamprogramando"
```

Código fuente 299

- **Replace( )**. Cambia dentro de la cadena, todas las ocurrencias de una subcadena por otra. Ver el Código fuente 300.

```
Dim lsCadCompleta As String
lsCadCompleta = "En el bosque se alza el castillo negro"
Console.WriteLine("Replace --> {0}", lsCadCompleta.Replace("el", "la"))
```

Código fuente 300

- **StartsWith( ), EndsWith( )**. Comprueban que en la cadena exista una subcadena al principio o final respectivamente. Ver el Código fuente 301.

```
Dim lsCadena As String
lsCadena = "veinte"
Console.WriteLine(lsCadena.StartsWith("ve"))     ' True
Console.WriteLine(lsCadena.EndsWith("TE"))       ' False
```

Código fuente 301

- **SubString( )**. Obtiene una subcadena comenzando por una posición de la cadena, y extrayendo un número de caracteres.

- **IndexOf( ), LastIndexOf( )**. Buscan una subcadena pasada como parámetro, comenzando por el principio y el fin respectivamente; y devuelven la posición de comienzo de dicha subcadena. Ver el Código fuente 302.

```
Dim lsCadCompleta As String
lsCadCompleta = "En el bosque se alza el castillo negro"
Console.WriteLine("Substring --> {0}", lsCadCompleta.Substring(6, 5)) '
"bosqu"
Console.WriteLine("IndexOf --> {0}", lsCadCompleta.IndexOf("el"))   ' 3
Console.WriteLine("LastIndexOf --> {0}", lsCadCompleta.LastIndexOf("el"))   '
21
```

Código fuente 302

- **ToUpper( ), ToLower( )**. Cambian la cadena a mayúsculas y minúsculas respectivamente. Ver el Código fuente 303.

```
Dim lsCadMayMin As String
lsCadMayMin = "CambIaNDO A mayúsCUlAs Y MINúscULAS"
Console.WriteLine("Pasar a may. --> {0}", lsCadMayMin.ToUpper())
Console.WriteLine("Pasar a min. --> {0}", lsCadMayMin.ToLower())
```

Código fuente 303

- **Concat( )**. Concatena dos cadenas pasadas como parámetro. Este es un método compartido de la clase String, por lo que no se requiere una instancia previa de la clase. El modo, sin embargo más rápido y sencillo para concatenar, sigue siendo el operador específico de concatenación: &. Ver el Código fuente 304.

```
Dim lsConcatenar As String
lsConcatenar = String.Concat("Hola ", "a todos")
lsConcatenar = "ahora usamos" & " el operador para concatenar"
```

Código fuente 304

- **Copy( )**. Crea un nuevo objeto String, aunque el medio más sencillo consiste en asignar una cadena a la variable. Ver el Código fuente 305.

```
Dim lsCadA As String
Dim lsCadB As String
lsCadA = "uno"
lsCadB = String.Copy("OTRO")
Console.WriteLine("CadenaA --> {0}", lsCadA)
Console.WriteLine("CadenaB --> {0}", lsCadB)
```

Código fuente 305

- **Compare( )**. Este método compartido compara dos cadenas, y devuelve un valor menor de cero, si la primera cadena es menor que la segunda; cero si ambas cadenas son iguales; y mayor de cero, si la primera cadena es mayor. Ver el Código fuente 306.

```
Dim lsCompara1 As String
Dim lsCompara2 As String
Dim liResultaComp As Integer
Console.WriteLine("Introducir primera cadena a comparar")
lsCompara1 = Console.ReadLine()
Console.WriteLine("Introducir segunda cadena a comparar")
lsCompara2 = Console.ReadLine()

liResultaComp = String.Compare(lsCompara1, lsCompara2)
Select Case liResultaComp
    Case Is < 0
        Console.WriteLine("Primera cadena es menor")
    Case 0
        Console.WriteLine("Las cadenas son iguales")
    Case Is > 0
        Console.WriteLine("Primera cadena es mayor")
End Select
```

Código fuente 306

- **Equals( )**. Compara el objeto con una cadena pasada como parámetro, y devuelve un valor lógico, que indica si las cadenas son o no iguales. Ver el Código fuente 307.

```
Dim lsCadInicial As String
Dim lsCadComparar As String
lsCadInicial = "Prueba"
Console.WriteLine("Introducir una cadena a comparar con la cadena inicial")
lsCadComparar = Console.ReadLine()
If lsCadInicial.Equals(lsCadComparar) Then
    Console.WriteLine("Las cadenas son iguales")
Else
    Console.WriteLine("Las cadenas son diferentes")
End If
```

Código fuente 307

Optimizando la manipulación de cadenas con la clase StringBuilder

Como ya hemos comentado en el apartado anterior, cualquier operación que tengamos que hacer sobre una cadena de caracteres, genera un nuevo objeto String, dado que este es un objeto inalterable una vez instanciado.

Este motivo hace que el rendimiento de la aplicación se vea afectado negativamente, en procesos que requieran operaciones intensivas con cadenas de caracteres, debido a la penalización que impone la generación constante de nuevos objetos String para cualquier mínimo cambio realizado.

Para solventar este problema, el entorno nos provee de la clase StringBuilder, que como su propio nombre indica, nos permite la construcción de cadenas, utilizando un mismo objeto para una serie de operaciones sobre una cadena de caracteres. Una vez terminadas todas las tareas sobre la cadena, podremos asignar el resultado a un objeto String.

Para poder utilizar objetos de esta clase, debemos importar en nuestro código el espacio de nombres System.Text.

Un objeto StringBuilder optimiza su manipulación de la siguiente manera:

Al instanciar el objeto, se reserva memoria para una cantidad predeterminada de caracteres. Este valor viene indicado en la propiedad Capacity, pudiendo ser asignado y consultado.

En sucesivas operaciones, si se añaden al objeto más caracteres de los que su capacidad actual está preparada para contener, se aumenta automáticamente la cantidad de memoria necesaria, de forma que el objeto se adapta a la nueva situación. El incremento de la propiedad Capacity se realiza en valores prefijados, por lo que si queremos saber realmente la longitud de la cadena que contiene el objeto, deberemos consultar la propiedad Length.

El Código fuente 308 contiene un ejemplo de manejo de un objeto StringBuilder, sobre el que realizamos varias operaciones antes de pasarlo definitivamente a un String. A lo largo de este fuente se encuentran comentarios descriptivos de los diferentes miembros utilizados del objeto StringBuilder.

```
Imports System.Text

Module Module1
    Sub Main()
        ' instanciar un objeto
        Dim sbCaracteres As StringBuilder = New StringBuilder()

        ' con el método Append() añadimos caracteres al objeto
        sbCaracteres.Append("hola ")
        sbCaracteres.Append("vamos a crear ")
        sbCaracteres.Append("caracteres con StringBuilder")

        ' la propiedad Length devuelve la cantidad real
        ' de caracteres que contiene el objeto
        Console.WriteLine("Longitud de la cadena del objeto StringBuilder: {0}", _
            sbCaracteres.Length)

        ' la propiedad Capacity devuelve la cantidad de caracteres
        ' que el objeto puede contener
        Console.WriteLine("Capacidad del objeto StringBuilder: {0}", _
            sbCaracteres.Capacity)
```

```
        ' el método Insert() permite incluir una cadena
        ' dentro del objeto, a partir de una posición determinada
        sbCaracteres.Insert(6, "SORPRESA")
        Console.WriteLine("Inserción de cadena")
        Console.WriteLine("Cadena: {0}", sbCaracteres.ToString())

        ' con el método Remove(), borramos a partir de una
        ' posición del objeto, un número de caracteres
        sbCaracteres.Remove(45, 3)
        Console.WriteLine("Eliminación de caracteres")
        Console.WriteLine("Cadena: {0}", sbCaracteres.ToString())

        ' con el método Replace(), sustituimos una cadena
        ' por otra dentro del objeto
        sbCaracteres.Replace("crear", "pintar")
        Console.WriteLine("Reemplazo de caracteres")
        Console.WriteLine("Cadena: {0}", sbCaracteres.ToString())

        ' la siguiente línea ajusta la capacidad
        ' del objeto a la cantidad real de caracteres que tiene
        sbCaracteres.Capacity = sbCaracteres.Length

        Console.WriteLine()

        ' volcamos el contenido del objeto a una cadena,
        ' el método ToString() devuelve un tipo String,
        ' que pasamos a una variable de dicho tipo
        Dim sCadena As String
        sCadena = sbCaracteres.ToString()
        Console.WriteLine("La variable sCadena contiene: {0}", sCadena)

        Console.ReadLine()
    End Sub
End Module
```

Código fuente 308

Conversión de tipos con la clase Convert

Si queremos realizar conversiones entre los tipos base del sistema, podemos utilizar las funciones de conversión de tipos del lenguaje, o lo que es más recomendable, usar los métodos compartidos de la clase Convert, con la ventaja de conseguir un código más orientado a objetos en todos los sentidos.

El Código fuente 309 convierte un número a cadena, y después esa cadena a un número utilizando los métodos de esta clase.

```
Dim lsCadena As String
lsCadena = Convert.ToString(150)    ' "150"

Dim liNum As Integer
liNum = Convert.ToInt32(lsCadena)   ' 150
```

Código fuente 309

La estructura Char

Cuando necesitemos manipular caracteres independientes, utilizaremos los métodos compartidos de esta estructura, que nos informarán del tipo de carácter que estamos manejando, además de poder realizar determinadas operaciones sobre el carácter.

El Código fuente 310 muestra un ejemplo de uso de la estructura Char. Cada uno de los miembros de Char empleados se encuentra con un pequeño comentario aclaratorio de su funcionalidad.

```
Public Sub Main()
    Dim lcCaracter As Char
    Dim lsResultado As String
    Dim lcConvertido As Char

    Do
        Console.WriteLine("Introducir un carácter o cero para salir")
        lcCaracter = Convert.ToChar(Console.ReadLine())
        lsResultado = ""
        lcConvertido = Nothing

        ' IsDigit() indica si el carácter es un dígito decimal
        If Char.IsDigit(lcCaracter) Then
            lsResultado = "dígito"
        End If

        ' IsLetter() indica si el carácter es una letra
        If Char.IsLetter(lcCaracter) Then
            lsResultado = "letra"
        End If

        ' IsWhiteSpace() indica si el carácter es un espacio en blanco
        If Char.IsWhiteSpace(lcCaracter) Then
            lsResultado = "espacio"
        End If

        ' IsPunctuation() indica si el carácter es un signo de puntuación
        If Char.IsPunctuation(lcCaracter) Then
            lsResultado &= "puntuación"
        End If

        ' IsUpper() comprueba si el carácter está en mayúscula
        If Char.IsUpper(lcCaracter) Then
            lsResultado &= " mayúscula"
            ' ToLower() convierte el carácter a minúscula
            lcConvertido = Char.ToLower(lcCaracter)
        End If

        ' IsLower() comprueba si el carácter está en minúscula
        If Char.IsLower(lcCaracter) Then
            lsResultado &= " minúscula"
            ' ToUpper() convierte el carácter a mayúscula
            lcConvertido = Char.ToUpper(lcCaracter)
        End If

        ' mostramos una cadena con el tipo de carácter
        Console.WriteLine("El carácter es: {0}", lsResultado)

        ' si hemos convertido el caracter a mayúscula/minúscula,
        ' lo mostramos
        If Char.IsLetter(lcConvertido) Then
            Console.WriteLine("El carácter se ha convertido: {0}", lcConvertido)
        End If
```

```
        Console.WriteLine()

        ' no salimos hasta que no se introduzca un 0
    Loop Until lcCaracter = "0"c
End Sub
```

Código fuente 310

Para asignar un valor de manera explícita a una variable, parámetro, etc., de tipo Char, es recomendable situar el carácter c junto a dicho valor. Veamos el Código fuente 311.

```
Dim lcCaracter As Char
' ambas asignaciones son equivalentes, pero se recomienda la primera
lcCaracter = "H"c
lcCaracter = "H"
```

Código fuente 311

Sin embargo, si queremos asignar un valor Char a una variable tipificada como Object, debemos utilizar irremisiblemente el indicador c junto al valor, o de otro modo, el subtipo almacenado en la variable Object lo tomará como String en lugar de Char. El mejor modo de comprobarlo, es abriendo la ventana Locales en modo de depuración. Veamos un ejemplo en el Código fuente 312.

```
Dim loValor As Object
loValor = "H"   ' objeto de subtipo String
loValor = "H"c  ' objeto de subtipo Char
```

Código fuente 312

El tipo Date (fecha)

Este tipo de dato, que utilizamos para trabajar con fechas, hace uso de la estructura DateTime, por lo que cuando tipificamos una variable como Date, los miembros que realmente manipulamos son los de un tipo DateTime. Consulte el lector, el apartado dedicado a la estructura DateTime para una mayor información.

Operaciones aritméticas, la clase Math

La clase Math, canaliza en VB.NET el conjunto de operaciones aritméticas más habituales, que en versiones anteriores del lenguaje se encontraban de forma aislada en funciones.

Gracias a que sus miembros son compartidos, es muy fácil su uso, ya que sólo debemos especificar el nombre de la clase, seguido del método a ejecutar.

El Código fuente 313 muestra algunos ejemplos utilizando métodos de la clase Math. Consulte el lector la documentación de .NET Framework para una explicación detallada sobre todos los miembros de esta clase.

```
Sub Main()
    Dim liSigno As Integer
    Dim ldbRedondear As Double

    ' Abs(): devuelve el valor absoluto del número
    ' pasado como parámetro
    Console.WriteLine("Abs --> {0}", Math.Abs(-1867.79))

    ' Ceiling(): devuelve el número sin precisión decimal,
    ' más grande o igual que el pasado como parámetro
    Console.WriteLine("Ceiling --> {0}", Math.Ceiling(256.7235))

    ' Floor(): devuelve el número sin precisión decimal,
    ' más pequeño o igual que el pasado como parámetro
    Console.WriteLine("Floor --> {0}", Math.Floor(256.7235))

    ' Sign(): devuelve un valor informando del signo del número
    ' pasado como parámetro
    Console.WriteLine("Introducir número para averiguar su signo")
    liSigno = Console.ReadLine()
    Select Case Math.Sign(liSigno)
        Case -1
            Console.WriteLine("El número es negativo")
        Case 0
            Console.WriteLine("El número es cero")
        Case 1
            Console.WriteLine("El número es positivo")
    End Select

    ' Round(): redondea el número pasado como parámetro
    ldbRedondear = Math.Round(28.3215)
    Console.WriteLine("Redondear 28.3215 --> {0}", ldbRedondear)

    ldbRedondear = Math.Round(28.63215)
    Console.WriteLine("Redondear 28.63215 --> {0}", ldbRedondear)

    Console.ReadLine()
End Sub
```

Código fuente 313

Formateo de valores

La utilización de un formato sobre un tipo de dato, nos permite mostrar su valor de un modo distinto a como se encuentra almacenado en la aplicación. Por ejemplo, el valor puro de una fecha no muestra el nombre del mes; sin embargo, si aplicamos un formato a una fecha, podemos hacer que se muestre la fecha en un modo extendido, con el nombre del mes, día de la semana, etc.

Podemos aplicar los formatos de varias maneras. A continuación, se muestran algunas de las técnicas a emplear.

- Mediante caracteres que representan formatos definidos internamente por la plataforma.
- A través de patrones de formato, que consisten en un conjunto de caracteres especiales, cuya combinación nos permite crear formatos personalizados.
- Utilizando alguna de las clases del sistema que implementan el interfaz IFormatProvider. Al instanciar un objeto de una clase de este tipo, podemos alterar el formato por defecto que utilizará el tipo de dato.

Todos los tipos de datos del entorno que pueden mostrar información formateada, disponen del método ToString( ), al cuál podemos pasarle una cadena, con los especificadores de formato (caracteres, patrones, objetos de formato) que necesitemos.

Fechas

El tipo Date, aparte del método ToString( ), tiene algunos miembros que devuelven un tipo de formato fijo. Veamos el Código fuente 314.

```
Sub Main()
      Dim ldtFecha As Date
      ldtFecha = Date.Now()
      Console.WriteLine("ToLongDateString: {0}", ldtFecha.ToLongDateString())
      Console.WriteLine("ToUniversalTime: {0}", ldtFecha.ToUniversalTime())
End Sub
```

Código fuente 314

Empleando alguna de las sobrecargas del método ToString( ), podemos formatear en los modos mostrados seguidamente.

La Tabla 27 muestra algunos caracteres asociados a los formatos predefinidos.

Carácter de formato	Tipo de formato
d	Fecha corta
D	Fecha larga
G	General (fecha corta, hora larga)
g	General (fecha y hora cortas)
t	Hora corta
T	Hora larga
m, M	Mes y día
y, Y	Año y día

Tabla 27. Algunos caracteres de formatos predefinidos.

En el Código fuente 315 podemos ver un formateo de fechas con caracteres de formato.

```
Sub Main()
    Dim ldtFecha As Date
    Dim lsListaFormatos() As String = {"d", "D", "g", "G", "t", "T", "m", "y"}
    Dim lsFormato As String
```

```
    ldtFecha = Date.Now()
    For Each lsFormato In lsListaFormatos
        Console.WriteLine("Formato: {0}, resultado: {1}", _
            lsFormato, ldtFecha.ToString(lsFormato))
    Next
End Sub
```

Código fuente 315

La Tabla 28 por otra parte, muestra algunos caracteres utilizados para crear patrones de formato personalizados, los cuales, se deben combinar entre sí, para componer el formato que necesitemos.

Carácter para patrón de formato	**Resultado**
D	Día del mes sin cero a la izquierda
Dd	Día del mes con cero a la izquierda
Ddd	Nombre del día abreviado
Dddd	Nombre del día completo
M	Número de mes sin cero a la izquierda
MM	Número de mes con cero a la izquierda
MMM	Nombre del mes abreviado
MMMM	Nombre del mes completo
Yy	Año en dos dígitos
Yyyy	Año en cuatro dígitos
H	Hora en formato 12 horas
H	Hora en formato 24 horas
M	Minutos sin cero a la izquierda
Mm	Minutos con cero a la izquierda
S	Segundos sin cero a la izquierda
Ss	Segundos con cero a la izquierda

\literal	Si queremos que un carácter que forma parte de los caracteres especiales de formato, se muestre de forma literal, debemos anteponerle este marcador

Tabla 28. Caracteres para patrones de formato.

El Código fuente 316 muestra algunos formatos personalizados, construidos a base de patrones de formato.

```
Sub Main()
    Dim ldtFecha As Date
    ldtFecha = Date.Now()
    Console.WriteLine(ldtFecha.ToString("ddd, dd-MMM/yyyy"))
    Console.WriteLine(ldtFecha.ToString("dddd, a dd \de MMMM ,en el año yyyy"))
    Console.WriteLine(ldtFecha.ToString("H:mm:s"))
End Sub
```

Código fuente 316

Si queremos obtener un array con todos los posibles formatos de una fecha, usaremos el método GetDateTimeFormats( ). Ver el Código fuente 317.

```
Sub Main()
    Dim ldtFecha As Date
    ldtFecha = Date.Now()

    ' array para obtener todos los formatos de fecha del sistema
    Dim lsListaFormatos() As String
    lsListaFormatos = ldtFecha.GetDateTimeFormats()
    Dim lsFormato As String
    For Each lsFormato In lsListaFormatos
        Console.WriteLine(lsFormato)
    Next

    Console.ReadLine()
End Sub
```

Código fuente 317

Modificando el formato estándar para las fechas

El método ToString( ) también puede recibir un objeto de la clase DateTimeFormatInfo. Esta clase implementa el interfaz IFormatProvider, y como hemos comentado antes, nos va a permitir alterar el formato estándar utilizado para mostrar las fechas, por uno personalizado.

Mediante los miembros de un objeto DateTimeFormatInfo, podemos asignar valores al formato que necesitamos crear. En primer lugar, para poder usar objetos de este tipo, precisamos importar esta clase en nuestro código, desde la ruta de espacios de nombre System.Globalization. En el ejemplo del Código fuente 318, utilizamos un objeto de DateTimeFormatInfo para mostrar el mes y día de una fecha.

```
' debemos importar este espacio de nombres
Imports System.Globalization

Module Module1
    Sub Main()
        Dim ldtFecha As Date
        ldtFecha = Date.Now()

        ' creamos un objeto específico de formato,
        ' y aplicamos dicho formato
        Dim loDTFormato As New DateTimeFormatInfo()
        ' ahora mostramos con el objeto de formato
        ' sólo el mes y día
        Console.WriteLine(ldtFecha.ToString(loDTFormato.MonthDayPattern))
      End Sub
End Module1
```

Código fuente 318

El ejemplo del Código fuente 319 va un poco más allá. En él, después de instanciar un objeto DateTimeFormatInfo, altera algunas de sus propiedades, para que al formatear una fecha, los formatos estándar se alteren en función de las modificaciones efectuadas sobre el objeto de formato.

```
Imports System.Globalization

Module Module1
    Sub Main()
        ' creamos una fecha
        Dim ldtFecha As Date
        ldtFecha = #8/6/2002 6:35:02 PM#

        ' creamos un objeto de formato para fechas
        Dim loDTFormato As New DateTimeFormatInfo()

        ' al mostrar la hora, podemos hacer que se
        ' visualice indicando si es AM o PM
        ' con un literal personalizado
        loDTFormato.PMDesignator = "Desp.Mediodía"
        loDTFormato.AMDesignator = "Antes del mediodía"

        ' podemos asignar una cadena con los nombres
        ' de los días de la semana en un estilo propio
        Dim lsDias() As String = {"1LUNES", "2MARTES", _
            "3MIÉRCOLES", "4JUEVES", "5VIERNES", "6SÁBADO", "7DOMINGO"}
        loDTFormato.DayNames = lsDias

        ' aquí, asignamos un formato para la fecha larga,
        ' este es el formato que se muestra al pasar "D" a ToString()
        loDTFormato.LongDatePattern = "Diario e\s\telar: ddd, dd \de MMMM \de yyyy"

        ' una vez configurado el objeto de formato
        ' lo utilizamos en la versión sobrecargada
        ' de ToString() correspondiente
        Dim lsFHFormateada As String
        lsFHFormateada = ldtFecha.ToString("D", loDTFormato)
        Console.WriteLine("Formato largo: {0}", lsFHFormateada)
        lsFHFormateada = ldtFecha.ToString("dddd MMM yy", loDTFormato)
        Console.WriteLine("Mostrar nombre de día personalizado: {0}",
lsFHFormateada)
        lsFHFormateada = ldtFecha.ToString("H:m:s tt", loDTFormato)
        Console.WriteLine("Mostrar hora con AM/PM propio: {0}", lsFHFormateada)
```

```
        Console.ReadLine()
    End Sub
End Module
```

Código fuente 319

Números

Podemos formatear un número mediante los caracteres de formato predefinidos. La Tabla 29 muestra los existentes.

Carácter de formato	Tipo de formato
c,C	Monetario
d,D	Decimal
e,E	Científico
f,F	Punto fijo
g,G	General
n,N	Numérico
r,R	Redondeo en ambas direcciones. Asegura que las conversiones de número a cadena y viceversa, no alteren el valor de los números
x,X	Hexadecimal

Tabla 29. Caracteres para formatos predefinidos.

El Código fuente 320 muestra algunos formatos aplicados sobre un tipo numérico.

```
Sub Main()
    Dim ldcMiNum As Decimal
    ' crear un array con caracteres de formato
    Dim lsFormatos() As String = {"c", "e", "f", "g", "n"}
    Dim lsNumFormateado As String

    ldcMiNum = 850.678  ' asignar valor al número

    ' recorrer el array de formatos y aplicar cada
    ' uno de los formatos al número
    For Each lsNumFormateado In lsFormatos
        Console.WriteLine(ldcMiNum.ToString(lsNumFormateado))
    Next
```

```
    Console.ReadLine()
End Sub
```

Código fuente 320

Formateando directamente en la consola

Mediante el método WriteLine( ) del objeto Console, podemos establecer los especificadores de formato en los parámetros sustituibles de dicho método, empleando el siguiente esquema: {NumParámetro:EspecificadorFormato}. El Código fuente 321 muestra algunos ejemplos.

```
Dim ldbImporte As Double
Dim ldtFecha As Date

ldbImporte = 58.367
ldtFecha = Date.Now()

Console.WriteLine("El precio total {0:C} se paga en efectivo", ldbImporte)
Console.WriteLine("El dato {0:E} está en notación científica", ldbImporte)
Console.WriteLine("Hoy es {0:D} en modo largo", ldtFecha)
Console.WriteLine("La hora del sistema es {0:t} ", ldtFecha)
Console.WriteLine("El día y mes actuales: {0:m} ", ldtFecha)
```

Código fuente 321

Usando la clase String para formatear

El método Format( ) de esta clase, nos permite efectuar operaciones de formateo utilizando la misma técnica que acabamos de describir en el apartado anterior. En el primer parámetro pasaremos la cadena con los especificadores de formato, mientras que en lo siguientes parámetros pasaremos los valores a sustituir. Veamos unos ejemplos en el Código fuente 322.

```
Dim ldbImporte As Double
Dim ldtFecha As Date
Dim lsCadFormateada As String

ldbImporte = 58.367
ldtFecha = Date.Now()
lsCadFormateada = String.Format("El valor de total de la compra es {0:C}", _
    ldbImporte)
lsCadFormateada = String.Format("La hora actual es {0:T}", ldtFecha)
lsCadFormateada = String.Format("Hoy es {0:dddd}, y el mes es {0:MMM}", _
    ldtFecha)
```

Código fuente 322

Usando una clase para crear formatos personalizados

Supongamos que hay una operación de formato que necesitamos aplicar, pero que no existe por defecto. Por ejemplo, convertir una cadena que contiene un nombre a mayúscula cada primera letra, y a minúscula el resto.

Si no disponemos de ningún tipo en el sistema que realice tal conversión, podemos crear una clase que implemente el interfaz IFormattable, y por ende, el método ToString( ) sobrecargado. Este método debe recibir una cadena con el tipo de formato que debemos aplicar, y un objeto que implemente el interfaz IFormatProvider, que se encargue de operaciones de formato. Veamos como aplicar esta solución en el Código fuente 323.

```
Module Module1
    Public Sub Main()
        Dim loNombre As CadenaFmt
        loNombre = New CadenaFmt("anToNIo mEsa pErAl")
        Console.WriteLine(loNombre.ToString("NOMPROPIO", _
            New System.Globalization.CultureInfo("es")))
        ' resultado: Antonio Mesa Peral
        Console.ReadLine()
    End Sub
End Module

' clase para formatear nombres propios
Public Class CadenaFmt
    Implements IFormattable

    Private msCadValor As String

    Public Sub New(ByVal lsCadValor As String)
        msCadValor = lsCadValor
    End Sub

    ' debemos escribir el método ToString() en esta clase,
    ' ya que forma parte de la implementación del interfaz IFormattable
    Public Overloads Function ToString(ByVal format As String, _
        ByVal formatProvider As System.IFormatProvider) _
        As String Implements System.IFormattable.ToString

        If format = "NOMPROPIO" Then
            ' cuando el nombre del formato a aplicar
            ' sea NOMPROPIO, llamar a un método
            ' que convierte a mayúsculas la primera
            ' letra de cada palabra
            Return Me.ConvertirProperCase()
        Else
            ' si no queremos utilizar nuestro formato
            ' personalizado, aplicar el que indique el
            ' parámetro formarProvider de este método
            Return String.Format(format, formatProvider)
        End If
    End Function

    Private Function ConvertirProperCase() As String
        Dim lcCaracteres() As Char
        Dim lcUnCaracter As Char
        Dim lbPasarMay As Boolean
        Dim liContador As Integer

        ' pasamos la cadena a convertir a un array Char
        lcCaracteres = msCadValor.ToCharArray()

        ' recorrer el array y hacer las oportunas conversiones
        lbPasarMay = True
        For liContador = 0 To UBound(lcCaracteres)
            If lbPasarMay Then
                lbPasarMay = False
                ' convertir a mayúscula cada primera letra del nombre
                lcCaracteres(liContador) = Char.ToUpper(lcCaracteres(liContador))
            Else
```

```
                ' convertir a minúscula el resto de letras
                lcCaracteres(liContador) = Char.ToLower(lcCaracteres(liContador))
            End If

            If Char.IsWhiteSpace(lcCaracteres(liContador)) Then
                lbPasarMay = True
            End If
        Next

        ' devolvemos el nombre convertido,
        ' aunque previamente pasamos el array Char
        ' a un tipo String
        Return New String(lcCaracteres)
    End Function
End Class
```

Código fuente 323

Como puede observar el lector, la no existencia de un tipo determinado de formato no es un problema, ya que con este sistema, podemos crear nuestro propios formatos.

22

Delegación de código y eventos

Delegados (delegates)

Un delegado o delegate, es un objeto al que otros objetos ceden (delegan) la ejecución de su código. También se conocen como *punteros a función con seguridad de tipos*.

Al instanciar un delegado, se asocia con un método de instancia o compartido de un objeto, y posteriormente, durante la ejecución, será el delegado el que se encargue de ejecutar dicho método y no el propio objeto. También se pueden asociar los delegados con procedimientos Sub o Function de módulos.

Declaración de delegados

Para declarar un delegado, debemos utilizar la palabra clave Delegate, seguida del tipo de método (Sub o Function) al que posteriormente deberemos asociar el delegado; y finalmente, el nombre del delegado con la lista de parámetros y valor de retorno si es necesario. El lugar de declaración debe ser la zona de declaraciones de la clase o módulo. Veamos unos ejemplos en el Código fuente 324.

```
' delegado sin parámetro
Public Delegate Sub VerMensaje()

' delegado con parametros
Public Delegate Sub Aviso(ByVal lsTexto As String)

' delegado de tipo function
```

```
Public Delegate Function Obtener(ByVal ldtFecha As Date) As String
```

Código fuente 324

Para que el lector pueda reproducir los ejemplos mostrados en este tema, abra un nuevo proyecto de tipo aplicación de consola en VS.NET.

Creación de delegados

Seguidamente, y ya en un procedimiento, declaramos una variable correspondiente al tipo del delegado. A continuación, conectamos el delegado con el procedimiento que posteriormente deberá ejecutar, empleando la palabra clave AddressOf, seguida del nombre del procedimiento.

AddressOf devuelve el puntero o dirección de entrada al procedimiento, que será lo que utilice el delegado para saber la ubicación del procedimiento que debe ejecutar. Por último, para ejecutar el procedimiento al que apunta el delegado, llamaremos a su método Invoke( ). En el Código fuente 325, se muestran dos técnicas para crear un delegado; la segunda es mucho más simple, pero en ambas, el resultado es el mismo: la ejecución indirecta del procedimiento MostrarTexto( ), a través del delegado.

```
Module Module1
    Public Delegate Sub VerMensaje()

    ' Modo 1 ----------------------------
    Sub Main()
       ' declarar un delegado
       Dim loDelegTexto As VerMensaje

       ' obtener la dirección del procedimiento a ejecutar
       ' y asignarla al delegado
       loDelegTexto = AddressOf MostrarTexto

       ' ejecutar el procedimiento a través del delegado
       loDelegTexto.Invoke()
    End Sub

    ' Modo 2 ----------------------------
    Sub Main()
       ' declarar el delegado y asociar con una dirección
       ' o puntero a un procedimiento
       Dim loDelegTexto As New VerMensaje(AddressOf MostrarTexto)
       loDelegTexto.Invoke()
    End Sub

    ' ***********************************************
    ' este será el procedimiento invocado (ejecutado)
    ' por el delegado
    Public Sub MostrarTexto()
       Console.WriteLine("Hola, esto es una prueba con delegados")
    End Sub
End Module
```

Código fuente 325

Una de las ventajas de este tipo de entidades de la plataforma, consiste en que un mismo delegado puede llamar a métodos diferentes de objetos distintos. En el caso del Código fuente 326, un delegado invoca a dos procedimientos diferentes.

```
Module Module1
    Public Delegate Sub VerMensaje()

    Sub Main()
        Dim loDelegMensa As VerMensaje

        loDelegMensa = AddressOf MostrarTexto
        loDelegMensa.Invoke()

        loDelegMensa = AddressOf VisualizarFecha
        loDelegMensa.Invoke()
    End Sub

       ' procedimientos ejecutados por los delegados
    Public Sub MostrarTexto()
        Console.WriteLine("Hola, esto es una prueba con delegados")
    End Sub

    Public Sub VisualizarFecha()
        Dim ldtFecha As Date
        ldtFecha = Date.Today
        Console.WriteLine("La fecha actual es {0:G}", ldtFecha)
    End Sub
End Module
```

Código fuente 326

Si delegamos un procedimiento que admite parámetros, a la hora de invocarlo con el delegado, debemos pasar al método Invoke( ) los valores de los parámetros, en el mismo orden que especifica el procedimiento. Veamos el ejemplo del Código fuente 327.

```
Module Module1
    Public Delegate Sub Aviso(ByVal lsTexto As String)

    Sub Main()
        Dim loGestionarAviso As Aviso

        loGestionarAviso = New Aviso(AddressOf Normal)
        loGestionarAviso.Invoke("Recuerda apagar el servidor")

        loGestionarAviso = New Aviso(AddressOf Urgente)
        loGestionarAviso.Invoke("Realizar la copia de seguridad")

        Console.ReadLine()
    End Sub

    Public Sub Normal(ByVal lsTexto As String)
        Console.WriteLine("* {0} *", lsTexto)
    End Sub
    Public Sub Urgente(ByVal lsTexto As String)
        Console.WriteLine("¡¡¡ {0} !!!", lsTexto.ToUpper)
    End Sub
End Module
```

Código fuente 327

En el caso de delegación hacia funciones, cuando invoquemos el código con el delegado, deberemos obtener el valor de retorno de la función. Veamos el ejemplo del Código fuente 328.

```
Module Module1
    Public Delegate Function Obtener(ByVal ldtFecha As Date) As String

    Sub Main()
        ' obtener una fecha
        Dim ldtFecha As Date
        Console.WriteLine("Introducir una fecha")
        ldtFecha = Console.ReadLine()

        ' crear un delegado, y según el mes de la fecha obtenida,
        ' el delegado ejecutará una función determinada
        Dim loManipFecha As Obtener
        Dim lsResultado As String
        If ldtFecha.Month < 6 Then
            loManipFecha = AddressOf RecuperaMes
        Else
            loManipFecha = AddressOf DameDiaSemana
        End If

        ' como el delegado ejecuta funciones, recuperamos el valor
        ' de retorno al invocar la función correspondiente
        lsResultado = loManipFecha.Invoke(ldtFecha)
        Console.WriteLine("El resultado obtenido es: {0}", lsResultado)

        Console.ReadLine()
    End Sub

    Public Function RecuperaMes(ByVal ldtFecha As Date) As String
        Return ldtFecha.ToString("MMMM")
    End Function

    Public Function DameDiaSemana(ByVal ldtFecha As Date) As String
        Return ldtFecha.ToString("dddd")
    End Function
End Module
```

Código fuente 328

Aunque en los anteriores ejemplos, hemos invocado los delegados desde el mismo procedimiento en que han sido creados, podemos, naturalmente, pasar un delegado como parámetro a un procedimiento, y que sea dicho procedimiento el encargado de ejecutar el código que guarda el delegado. De esta forma, si el anterior ejemplo lo variamos ligeramente, y añadimos un procedimiento que reciba el delegado y el parámetro, obtendríamos el Código fuente 329.

```
Sub Main()
    ' ......
    ' ......
    ' llamar a un procedimiento que ejecuta el delegado
    Gestionar(loManipFecha, ldtFecha)
    ' ......
    ' ......
End Sub

' en este procedimiento uno de los parámetros es un delegado
' y el otro el valor que utilizará el delegado como parámetro
Public Sub Gestionar(ByVal loDelObtener As Obtener, _
```

```
    ByVal ldtUnaFecha As Date)

    Dim lsResultado As String
    lsResultado = loDelObtener.Invoke(ldtUnaFecha)
    Console.WriteLine("El resultado obtenido es: {0}", lsResultado)
End Sub
```

Código fuente 329

Extender las funcionalidades de una clase a través de delegados

Supongamos que hemos creado una clase, con el nombre ManipFecha, que tiene una propiedad a la que asignamos una fecha, y un método que devuelve la fecha formateada. Ver el Código fuente 330.

```
Module Module1
    Sub Main()
        Dim loManipFecha As ManipFecha
        loManipFecha = New ManipFecha()
        loManipFecha.Fecha = #6/27/2002 7:40:00 PM#
        Console.WriteLine("Fecha larga: {0}", loManipFecha.DevFechaLarga())

        Console.ReadLine()
    End Sub
End Module

Public Class ManipFecha
    Private mdtFecha As Date

    Public Sub New()
        mdtFecha = Date.Now
    End Sub

    Public Property Fecha() As Date
        Get
            Return mdtFecha
        End Get

        Set(ByVal Value As Date)
            mdtFecha = Value
        End Set
    End Property

    Public Function DevFechaLarga() As String
        Return mdtFecha.ToLongDateString()
    End Function
End Class
```

Código fuente 330

Después de finalizar el desarrollo de la clase y distribuirla, se presenta el siguiente problema: los programadores que la utilizan, disponen de rutinas de formateo propias que necesitarían implementar en nuestra clase.

Este punto, evidentemente, lo podríamos resolver utilizando herencia; no obstante, vamos a solucionar el problema mediante delegados. Por lo tanto, añadiremos un delegado a la clase ManipFecha, y un

método que lo invoque; al ejecutar este método, le pasaremos la dirección del procedimiento que contiene la rutina personalizada de formateo. Ver Código fuente 331.

```
Module Module1
    Sub Main()
        ' crear un objeto de la clase
        Dim loManipFecha As ManipFecha
        loManipFecha = New ManipFecha()
        loManipFecha.Fecha = #6/27/2002 7:40:00 PM#

        ' llamar ahora al método que ejecuta un delegate;
        ' le pasamos a este método la dirección del
        ' procedimiento que el delegate debe invocar
        Dim lsFormatoResultado As String
        lsFormatoResultado = loManipFecha.FormateoExterno(AddressOf ObtenerHora)
        Console.WriteLine("Formateo externo resultante: {0}", lsFormatoResultado)
        Console.ReadLine()
    End Sub

    ' este es un procedimiento que contiene una rutina
    ' personalizada de formateo, y que será invocado
    ' desde el delegate del objeto
    Public Function ObtenerHora(ByVal ldtFecha As Date) As String
        Return ldtFecha.ToString("H:m")
    End Function
End Module

Public Class ManipFecha
    ' ....
    ' ....
    ' declarar un delegado, a través del cual,
    ' ejecutaremos rutinas de código externas a la clase
    Public Delegate Function RealizaFormato(ByVal ldtFecha As Date) As String
    ' ....
    ' ....
    ' este método utiliza un delegado pasado como
    ' parámetro para invocar a un procedimiento externo
    ' a la clase
    Public Function FormateoExterno(ByVal loDelegFormat As RealizaFormato) As
String
        Return loDelegFormat.Invoke(mdtFecha)
    End Function
End Class
```

Código fuente 331

No obstante, si disponemos de un conjunto más extenso de rutinas de formateo, lo mejor es aglutinarlas todas en métodos de una clase, invocando a estos métodos desde la clase que contiene el delegado.

El Código fuente 332 muestra la clase Formatos, en la que hemos creado dos métodos que realizan operaciones de formato.

```
Public Class Formatos
    ' los métodos de esta clase, reciben una fecha a la que
    ' aplican un formato y devuelven una cadena con el
    ' formato resultante
    Public Function Marinero(ByVal ldtFecha As Date) As String
        Return ldtFecha.ToString("Cua\derno \de bi\tácora: " & _
```

```
            "en el año yyyy, dd \de MMMM")
    End Function

    Shared Function Espacial(ByVal ldtFecha As Date) As String
        Return ldtFecha.ToString("Diario e\s\telar: " & _
            "MMM-d-yyyy / Sec\tor Ga\m\ma")
    End Function
End Class
```

Código fuente 332

Para conseguir, ya en Main( ), que el delegado de la clase ManipFecha ejecute el código de la clase Formatos, podemos utilizar dos técnicas.

Por un lado, instanciamos un objeto de Formatos, y pasamos al método FormateoExterno( ), del objeto ManipFecha, la dirección del método Marinero( ) del objeto Formatos.

Por otra parte, en lo referente al método Espacial( ) de la clase Formatos, no es necesario crear un objeto. Debido a que dicho método es compartido, podemos pasar su dirección al método FormateoExterno( ), del objeto ManipFecha.

```
Module Module1
    Sub Main()
        ' crear un objeto de la clase
        Dim loManipFecha As ManipFecha
        loManipFecha = New ManipFecha()
        loManipFecha.Fecha = #6/27/2002 7:40:00 PM#

        Dim lsFormatoResultado As String

        ' instanciar un objeto de la clase en la que tenemos
        ' los métodos personalizados de formateo
        Dim oFormato As New Formatos()

        ' ahora pasamos al método FormateoExterno() del objeto
        ' ManipFecha, la dirección de un método del objeto
        ' Formatos, en donde tenemos una rutina personalizada
        ' de formateo de fecha
        lsFormatoResultado = loManipFecha.FormateoExterno(AddressOf
oFormato.Marinero)
        Console.WriteLine("Formato estilo marinero: {0}", lsFormatoResultado)

        ' aquí efectuamos la misma operación que antes,
        ' pero pasamos como parámetro un método shared
        ' de la clase Formatos, es decir, no es necesario
        ' instanciar un objeto de Formatos
        lsFormatoResultado = loManipFecha.FormateoExterno(AddressOf
Formatos.Espacial)
        Console.WriteLine("Formato estilo espacial: {0}", lsFormatoResultado)

        Console.ReadLine()
    End Sub
End Module
```

Código fuente 333

Eventos. ¿Qué es un evento?

Un evento es un suceso o situación, que acontece en una ubicación de espacio y tiempo no predecible.

Cuando una máquina deja de funcionar por una avería, o cuando una persona resbala y cae, estamos en ambos casos, ante ejemplos de eventos, ya que ocurren en momentos inesperados.

Para que se desencadene un evento, se deben dar determinadas circunstancias, las cuales favorecen el que dicho evento se produzca.

Eventos en .NET

Ciñéndonos al ámbito de la programación, un evento es, dentro de una aplicación, una notificación lanzada por un objeto, que podrá ser respondida por aquellos otros objetos interesados en darle soporte.

Programación estrictamente procedural

Antes de la llegada de los sistemas y lenguajes orientados a eventos, las aplicaciones ejecutaban su código en un orden fijo, ya que estaban basadas en un modelo construido exclusivamente a base de procedimientos: se realizaban llamadas a las rutinas de código en un orden predeterminado, y una vez terminada la ejecución de tales rutinas, finalizaba la aplicación.

Un escenario de trabajo sin eventos

Supongamos que nos encargan desarrollar una clase llamada Empleado, entre cuyos miembros tenemos la propiedad Sueldo. Uno de los requerimientos respecto a esta propiedad es que su valor no debe ser superior a 1000; por ello, en su procedimiento Property, realizamos una validación a tal efecto, emitiendo un mensaje cuando el sueldo que asignemos sea superior. Ver el Código fuente 334.

```
Public Class Empleado
    ' variables de propiedad
    Private msNombre As String
    Private mdbSueldo As Double

    ' propiedad Nombre
    Public Property Nombre() As String
        Get
            Return msNombre
        End Get
        Set(ByVal Value As String)
            msNombre = Value
        End Set
    End Property

    ' propiedad Sueldo
    Public Property Sueldo() As Double
        Get
            Return mdbSueldo
        End Get

        ' al asignar un valor a la propiedad,
```

```
        ' si el valor es superior a 1000
        ' mostrar un mensaje y no permitir la
        ' asignación del sueldo
        Set(ByVal Value As Double)
            If Value > 1000 Then
                Console.WriteLine("Asignación de sueldo incorrecta")
                Console.ReadLine()
            Else
                mdbSueldo = Value
            End If
        End Set
    End Property
End Class
```

Código fuente 334

Una vez finalizado el desarrollo de la clase, la distribuimos a nuestro cliente. Posteriormente, un nuevo cliente nos requiere la clase, pero en esta ocasión, aunque necesita la validación sobre la propiedad Sueldo, no quiere que se muestre el mensaje al sobrepasar el sueldo asignado.

Se nos plantea en este caso un problema, ya que si escribimos una nueva versión de la clase Empleado, tendremos el trabajo extra de mantener ambas. Para solucionarlo mediante una única versión de la clase recurriremos a los eventos.

Programación basada en eventos

La aparición de Windows trajo consigo un nuevo esquema en el desarrollo de aplicaciones. En un programa que se ejecute dentro de un sistema basado en eventos como pueda ser Windows, se están produciendo constantemente eventos (sucesos), provocados por las acciones del usuario o por el propio sistema. Tan elevado es el número de eventos que se producen, que dar respuesta a todos, es decir, codificar todas aquellas situaciones que acontecen a lo largo de la ejecución de un programa, es algo impensable.

Por tal motivo, la técnica seguida al escribir código orientado a eventos, se basa en codificar sólo los eventos que nos interese tratar, ya que para el resto, será el propio sistema quien proporcione el comportamiento por defecto.

En una aplicación Windows típica, todos los elementos que forman parte de la misma, es decir, el propio formulario y los controles contenidos en él, lanzan eventos en respuesta a las acciones del usuario. El ejemplo más habitual: al pulsar un botón, se produce su evento clic; si queremos que el programa realice alguna acción al pulsar dicho botón, deberemos escribir código en el procedimiento de evento asociado, para dar respuesta a tal suceso.

Esquema básico de un sistema orientado a eventos

Un sistema conducido por eventos basa su funcionamiento en dos pilares fundamentales: un emisor y un receptor de eventos.

El primero genera y lanza el evento al sistema, mientras que el segundo, si está interesado en tratar el evento lanzado, lo captura y le da respuesta. Si un objeto receptor no necesita gestionar eventos, simplemente no lo obtiene. Ver Figura 206.

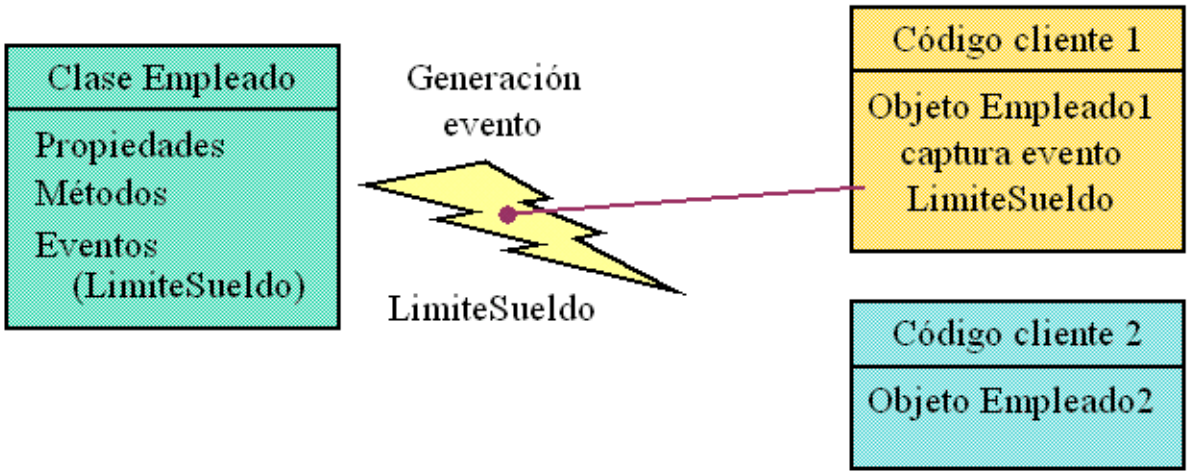


Figura 206. Esquema de generación y captura de eventos.

Dejemos ya atrás los aspectos conceptuales sobre eventos que estamos discutiendo, y veamos en los siguientes apartados, cada uno de los integrantes de la gestión de eventos en detalle.

El emisor de eventos

Un emisor de eventos, también denominado origen de eventos (*event source* o *event sender*), es un objeto capacitado para generar y lanzar eventos al sistema, que puedan ser recuperados por otros objetos preparados para realizar su tratamiento.

Para que un objeto pueda desencadenar eventos, en su clase debemos realizar dos tareas:

- Declarar el propio evento usando la palabra clave Event, especificando si es necesario una lista de parámetros que acompañan al evento.

- Lanzar el evento mediante la palabra clave RaiseEvent, seguida del nombre del evento a disparar. Si hemos declarado el evento con parámetros, deberemos añadir los valores para cada uno de los parámetros en el mismo orden en el que los hemos declarado.

Situándonos pues ante el problema planteado por la clase Empleado en un apartado anterior, la solución que proponemos consistirá en generar desde la clase Empleado un evento cuando se produzca un fallo en la validación del sueldo. De esta manera, el código cliente que lo necesite, responderá al evento; y el que no lo precise, hará caso omiso del evento lanzado.

En primer lugar, declaramos en la zona de declaraciones de la clase el evento LimiteSueldo, que irá acompañado de un parámetro que nos informará del importe erróneo que se intentaba asignar a la propiedad.

A continuación, en la propiedad Sueldo, cuando detectemos que el sueldo sobrepasa el valor permitido, en lugar de lanzar allí el mensaje a la consola, generaremos el evento LimiteSueldo, que podrá ser recuperado por el código cliente que haga uso de la clase, actuando como necesite en cada ocasión. Observe el lector, que al mismo tiempo que lanzamos el evento, le pasamos el importe del sueldo que se intentaba asignar. Veamos el Código fuente 335.

```
Public Class Empleado
```

```
    ' declaramos el evento
    Public Event LimiteSueldo(ByVal ldbImporte As Double)

    Private msNombre As String
    Private mdbSueldo As Double

    Public Property Nombre() As String
        Get
            Return msNombre
        End Get
        Set(ByVal Value As String)
            msNombre = Value
        End Set
    End Property

    Public Property Sueldo() As Double
        Get
            Return mdbSueldo
        End Get
        Set(ByVal Value As Double)
            ' si el valor que intentamos asignar
            ' al sueldo supera el permitido...
            If Value > 1000 Then
                ' ...lanzamos el evento, y le pasamos
                ' como parámetro informativo el valor
                ' incorrecto que intentábamos asignar
                RaiseEvent LimiteSueldo(Value)
            Else
                mdbSueldo = Value
            End If
        End Set
    End Property
End Class
```

Código fuente 335

Con estas modificaciones sobre la clase Empleado, ya tenemos listo nuestro emisor de eventos. Queda ahora por completar la parte que captura los eventos lanzados por el emisor.

El receptor de eventos

Un receptor de eventos, también denominado manipulador de eventos (*event receiver* o *event handler*), es aquella parte del código cliente, que configuramos para que sea capaz de recibir los eventos generados por un objeto emisor. Para que ambos elementos, en este canal de comunicación que es la transmisión de eventos puedan operar, es necesario conectarlos.

Conexión de un emisor de eventos con un manipulador de eventos

Existen dos medios para comunicar un evento con un manipulador de eventos:

- En tiempo de compilación, realizando un enlace estático entre la clase y el manipulador mediante las palabras clave WithEvents y Handles. Esta técnica tiene la ventaja de que permite escribir un código mucho más legible, en cuanto a la manipulación de eventos se refiere.

- En tiempo de ejecución, realizando un enlace dinámico entre la clase y el manipulador mediante la palabra clave AddHandler. La ventaja en este caso, es que podemos asociar procedimientos manipuladores de evento dinámicamente durante el transcurso de la ejecución del programa.

Enlace estático de eventos

Este es el modo mas sencillo para implementar la conexión entre un evento y un procedimiento manipulador de evento.

En primer lugar, declaramos una variable del tipo de objeto cuyos eventos queremos capturar, en la zona de declaraciones del módulo, clase, etc., utilizando la palabra clave WithEvents. Veamos el Código fuente 336.

```
Module Module1

    Private WithEvents moEmple As Empleado
'......
'......
```

Código fuente 336

A continuación, tenemos que escribir el procedimiento manipulador, que será invocado cada vez que se dispare el evento. Dicho procedimiento debe ser de tipo Sub, ya que un evento no puede devolver valores, por lo que no podremos utilizar un Function; también debemos finalizar su declaración con la palabra clave Handles, seguida del nombre de la variable del objeto que hemos declarado en la zona de declaraciones, y el nombre del evento que el procedimiento va a tratar. En el Código fuente 337, el procedimiento moEmple_LimiteSueldo( ), será llamado cada vez que se produzca el evento LimiteSueldo en el objeto Empleado.

```
Public Sub moEmple_LimiteSueldo(ByVal ldbImporte As Double) _
    Handles moEmple.LimiteSueldo

    Console.WriteLine("Se ha sobrepasado para {0} el límite" & _
        " establecido de sueldo", _
        moEmple.Nombre)
    Console.WriteLine("El importe {0} no es válido", ldbImporte)
    Console.ReadLine()
End Sub
```

Código fuente 337

El nombre utilizado para el procedimiento puede ser cualquiera, aunque en este caso hemos empleado la convención NombreObjeto_NombreEvento simplemente para facilitar la lectura del código, pero podríamos haber empleado, por ejemplo, el que se muestra en el Código fuente 338.

```
Public Sub Sobrepasado(ByVal ldbImporte As Double) _
      Handles moEmple.LimiteSueldo
      ' ....
```

```
        ' ....
End Sub
```

Código fuente 338

Un pequeño truco que tenemos en el editor de código de VS.NET, para facilitar la creación de los procedimientos manipuladores de evento, consiste en abrir la lista Nombre de clase y seleccionar el nombre de la variable que hemos declarado WithEvents. Ver Figura 207.

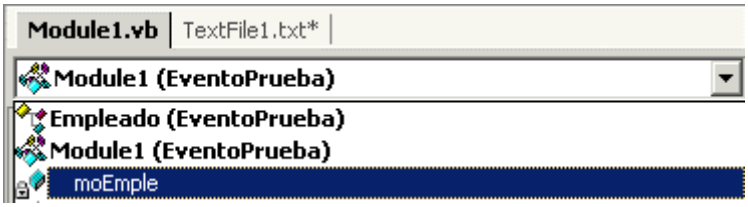


Figura 207. Seleccionar objeto declarado WithEvents.

Seguidamente pasamos a la lista Nombre de método, y allí elegimos el nombre del evento que vamos a codificar. Ver Figura 208.

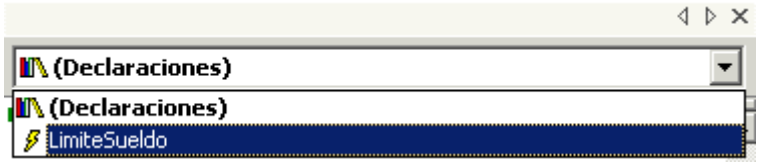


Figura 208. Seleccionar el evento a codificar.

Esto nos crea el procedimiento manipulador de evento vacío, en base a una convención de nombres predefinida en el IDE. Ver Código fuente 339.

```
Public Sub moEmple_LimiteSueldo(ByVal ldbImporte As Double) Handles
moEmple.LimiteSueldo

End Sub
```

Código fuente 339

Como hemos escrito el manipulador de evento para el objeto Empleado en un módulo, vamos ahora a escribir un procedimiento Main(), instanciando en el mismo, un objeto de esta clase. Asignaremos en primer lugar, un valor correcto a la propiedad Sueldo, y a continuación un valor que provocará el evento en la clase. Recomendamos al lector que ejecute el código línea a línea con el depurador, para observar el efecto cuando se produzca el evento.

```
Sub Main()
    moEmple = New Empleado()
    moEmple.Nombre = "Juan"
    moEmple.Sueldo = 500     ' esta asignación no provoca el evento
    moEmple.Sueldo = 8000    ' esta sí provoca el evento
```

```
End Sub
```

Código fuente 340

Enlace dinámico de eventos

Siendo un poco más complejo a nivel sintáctico que el enlace estático, el enlace dinámico de eventos a sus correspondientes manipuladores, tiene la ventaja de que nos permite asociar el mismo evento a diferentes procedimientos manipuladores de dicho evento, durante el transcurso de la ejecución del programa.

Por lo tanto, en el módulo de código donde tenemos a Main( ), vamos a escribir dos procedimientos que asociaremos dinámicamente al evento que hemos creado en la clase Empleado. Ver Código fuente 341.

```
Module Module1
'....
'....

' manipuladores de evento que conectaremos en tiempo de ejecución
Public Sub SobreAsignacionSueldo(ByVal ldbImporte As Double)
    Console.WriteLine("Se intentó asignar a un empleado el sueldo {0}" & _
        ControlChars.CrLf & "¡ESTO ES INCORRECTO!", ldbImporte)
End Sub

Public Sub SalarioIncorrecto(ByVal ldbImporte As Double)
    Console.WriteLine("INFORME DE INCIDENCIAS")
    Console.WriteLine("======================")
    Console.WriteLine("Error al intentar asignar el salario {0} a un empleado", _
        ldbImporte)
End Sub

'....
'....
End Module
```

Código fuente 341

Como ventaja adicional, el objeto sobre el que vamos a manipular sus eventos podemos declararlo tanto a nivel local como en la zona de declaraciones, a diferencia del enlace estático, que nos obligaba a declarar el objeto en la zona de declaraciones del módulo en el que fuéramos a utilizarlo.

Para establecer un enlace dinámico entre un evento y un manipulador, utilizaremos la instrucción AddHandler. Esta instrucción, recibe como primer parámetro el evento a conectar en el formato NombreObjeto.NombreEvento. Como segundo parámetro, pasaremos la dirección de entrada al procedimiento que deberá ejecutar el evento, y que obtenemos a través de la instrucción AddressOf. El Código fuente 342, muestra el procedimiento Main( ), en el que pedimos al usuario que introduzca un número, y según el valor obtenido, conectamos el evento con uno de los dos procedimientos manipuladores antes descritos.

```
Module Module1
'....
'....
```

```
Sub Main()
    ' pedir un número al usuario para conectar a uno de los
    ' dos procedimientos manipuladores de evento que hemos escrito
    Dim liTipoManip As Integer
    Console.WriteLine("Introduzca el número 1 ó 2," & _
        " para seleccionar el manipulador de evento a utilizar")
    liTipoManip = Console.ReadLine()

    ' instanciar un objeto Empleado
    Dim loMiEmpleado As New Empleado()

    ' asignar un manejador de evento en tiempo de ejecución
    ' en función del número que el usuario ha introducido
    Select Case liTipoManip
        Case 1
            AddHandler loMiEmpleado.LimiteSueldo, AddressOf SobreAsignacionSueldo

        Case 2
            AddHandler loMiEmpleado.LimiteSueldo, AddressOf SalarioIncorrecto
    End Select

    loMiEmpleado.Nombre = "ANTONIO"

    ' esta asignación provoca el evento,
    ' ello ejecutará uno de los manipuladores
    ' de evento que hemos conectado
    loMiEmpleado.Sueldo = 2500
    Console.ReadLine()
End Sub
'....
'....
End Module
```

Código fuente 342

Un evento es un delegado

El sistema interno que utiliza .NET Framework para la creación, conexión y ejecución de eventos, está basado en delegados.

Cuando declaramos en una clase, un evento con la instrucción Event, se crea de modo transparente para el programador, un nuevo delegado con el nombre del evento más la palabra EventHandler. Ver Código fuente 343.

```
Public Class Empleado
    ' al declarar el evento...
    Public Event LimiteSueldo(ByVal ldbImporte As Double)

    ' ...internamente se crea el siguiente delegado
    Public Delegate Sub LimiteSueldoEventHandler(ByVal ldbImporte As Double)
'....
'....
End Class
```

Código fuente 343

Al conectar el evento de un objeto con un procedimiento manipulador, en alguno de los modos descritos en los anteriores apartados, se crea, también de un modo transparente, una nueva instancia

del delegado. No obstante, en este caso, el programador sí puede de forma explícita, realizar la creación del delegado. El Código fuente 344, muestra como en el segundo parámetro de AddHandler creamos manualmente el delegado, que internamente ejecuta el procedimiento manipulador de evento.

```
AddHandler loMiEmpleado.LimiteSueldo, _
      New Empleado.LimiteSueldoEventHandler(AddressOf SobreAsignacionSueldo)
```

Código fuente 344

Finalmente, cuando desde la clase se lanza el evento con RaiseEvent, internamente se ejecuta el método Invoke( ) del delegado, lo cual producirá la ejecución del procedimiento asociado al delegado. Ver Código fuente 345.

```
' la instrucción RaiseEvent...
RaiseEvent LimiteSueldo(Value)

' ...realmente ejecuta el método Invoke() del delegado
' LimiteSueldoEventHandler.Invoke(Value)
```

Código fuente 345

La clase EventArgs, o cómo obtener información del objeto emisor del evento

En los ejemplos sobre captura de eventos realizados hasta ahora con la clase Empleado, dentro de los procedimientos manipuladores de evento, no disponemos de acceso a los miembros del objeto que ha originado el evento.

El único modo de los vistos hasta ahora de conseguir tal acceso, es declarar el objeto en la zona de declaraciones del módulo y, en ese caso, al tener visibilidad sobre la variable del objeto en todos los procedimientos del módulo, sí podríamos manejar el objeto.

Sin embargo, ¿qué ocurre cuando instanciamos un objeto Empleado con ámbito local en Main( ), y asociamos sus manipuladores de evento con AddHandler?. Simplemente, que desde dichos procedimientos manipuladores de evento, no podemos obtener información del objeto Empleado para, por ejemplo, recuperar el valor de la propiedad Nombre.

Una solución simple, pero no eficaz, consistiría en pasar la/s propiedad/es como parámetro cuando lanzamos el evento, es decir, al llamar a RaiseEvent( ) en la clase Empleado. Ver Código fuente 346.

```
Public Class Empleado
'....
      RaiseEvent LimiteSueldo(Value,Me.Nombre)
'....
End Class
```

Código fuente 346

Pero seguiríamos limitados, en el caso de que necesitáramos pasar cualquier otro tipo de información que no estuviera directamente relacionada con el objeto.

Para solucionar este problema, podemos utilizar la técnica empleada por la propia plataforma .NET en la retransmisión de eventos, y que explicamos a continuación.

La jerarquía de clases de .NET dispone de la clase EventArgs, diseñada para guardar la información adicional que pasamos a un procedimiento manipulador de evento.

Podemos crear una clase que herede de EventArgs, y adaptarla, en este caso, para que contenga la información adicional sobre un evento que se ha producido en la clase Empleado, de modo que cuando se ejecute su manipulador asociado, pasemos a dicho manipulador, como primer parámetro, el propio objeto Empleado, y como segundo, un objeto de nuestra clase EventArgs personalizada, con datos adicionales sobre el evento generado. Este es el esquema general de trabajo con los eventos en .NET.

Escribiremos por lo tanto la clase EmpleadoEventArgs, que hereda de EventArgs, y que servirá para que cuando a un objeto Empleado se le intente asignar un sueldo incorrecto, se almacene en ella dicho valor erróneo. Ver Código fuente 347.

```
' clase para guardar información sobre
' los eventos lanzados por la clase Empleado;
' esta clase en concreto, guardará el valor del sueldo
' erróneo que se ha intentado asignar a un empleado
Public Class EmpleadoEventArgs
    Inherits EventArgs

    Private mdbSueldoIntentadoAsig As Double

    Public Property SueldoIntentadoAsig() As Double
        Get
            SueldoIntentadoAsig = mdbSueldoIntentadoAsig
        End Get
        Set(ByVal Value As Double)
            mdbSueldoIntentadoAsig = Value
        End Set
    End Property
End Class
```

Código fuente 347

Seguidamente retocaremos el código de la clase Empleado, cambiando la declaración del evento LimiteSueldo, y la sección Set de su procedimiento Property Sueldo, tal y como se muestra en el Código fuente 348.

```
Public Class Empleado
    ' declaramos el evento LimiteSueldo,
    ' el primer parámetro será la instancia y objeto
    ' Empleado actualmente en ejecución;
    ' el segundo parámetro será un objeto EmpleadoEventArgs,
    ' con la información adicional del evento producido
    Public Event LimiteSueldo(ByVal sender As Empleado, _
        ByVal e As EmpleadoEventArgs)
       '....
       '....
    Public Property Sueldo() As Double
        Get
```

```
            Return mdbSueldo
        End Get
        Set(ByVal Value As Double)
            ' si el valor que intentamos asignar
            ' al sueldo supera el permitido...
            If Value > 1000 Then
                ' ...creamos un objeto EmpleadoEventArgs
                ' y le pasamos a sus propiedades la
                ' información sobre el evento; en este
                ' caso sólo pasamos el valor incorrecto
                ' que se intentó asignar a la propiedad Sueldo
                Dim loEvArgs As New EmpleadoEventArgs()
                loEvArgs.SueldoIntentadoAsig = Value

                ' después lanzamos el evento y le pasamos
                ' como parámetro el propio objeto Empleado actual
                ' y el objeto con la información del evento
                RaiseEvent LimiteSueldo(Me, loEvArgs)

            Else
                ' si el sueldo es correcto, se asigna
                mdbSueldo = Value
            End If
        End Set
    End Property
       '....
       '....
End Class
```

Código fuente 348

Los nombres empleados en la declaración del evento de esta clase: *sender*, para designar al emisor del evento; y *e*, para designar los argumentos del evento, no son en absoluto obligatorios, pudiendo el lector utilizar los nombres que estime oportunos. El haber utilizado estas denominaciones se debe a seguir la misma convención que utiliza la plataforma. En los temas dedicados a formularios y controles Windows, el lector podrá comprobar que los procedimientos manipuladores de evento, usan estos mismos nombres.

Para terminar, escribimos en el módulo un procedimiento manipulador para el evento LimiteSueldo, y en Main( ) instanciamos un objeto Empleado, asociando el evento del objeto al manipulador de evento que acabamos de escribir. Ver el Código fuente 349.

```
Module Module1
    Sub Main()
        ' declarar e instanciar un objeto Empleado
        Dim loEmpleado As Empleado
        loEmpleado = New Empleado()

        ' añadir un manipulador de evento para el evento LimiteSueldo
        AddHandler loEmpleado.LimiteSueldo, AddressOf SobreAsignacionSueldo

        loEmpleado.Nombre = "ANA"
        loEmpleado.Sueldo = 5000    ' esto provoca el evento

        Console.ReadLine()
    End Sub

    ' procedimiento manipulador del evento LimiteSueldo;
    ' del parámetro sender obtenemos el nombre del Empleado,
    ' del parámetro e obtenermos el importe incorrecto
    ' que intentábamos asignar al sueldo del empleado
```

```
    Public Sub SobreAsignacionSueldo(ByVal sender As Empleado, _
        ByVal e As EmpleadoEventArgs)

        Console.WriteLine("Se intentó asignar al empleado {0}, el sueldo {1}" & _
            ControlChars.CrLf & "¡ESTO ES INCORRECTO!", _
            sender.Nombre, _
            e.SueldoIntentadoAsig)

    End Sub

End Module
```

Código fuente 349

Aunque este modo de trabajo suponga un esfuerzo adicional por nuestra parte en cuanto a que tengamos que escribir algo más de código, los eventos de nuestras clases tendrán una estructura de llamada más acorde con el resto de eventos de las clases pertenecientes a la plataforma.

23

Arrays

Aspectos básicos

También conocido con las denominaciones de matriz y vector, un array es aquel elemento del lenguaje que nos permite agrupar un conjunto de valores del mismo tipo, y acceder a ellos a través de una misma variable o identificador, especificando la posición o índice en donde se encuentra el dato a recuperar. El Código fuente 350, muestra las operaciones esenciales que podemos realizar con un array.

```
Sub Main()
    ' declarar un array de tipo String,
    ' el número de elementos es el indicado
    ' en la declaración más uno, porque la primera
    ' posición de un array es cero
    Dim sNombres(3) As String

    ' asignar valores al array
    sNombres(0) = "Ana"
    sNombres(1) = "Pedro"
    sNombres(2) = "Antonio"
    sNombres(3) = "Laura"

    ' pasar un valor del array a una variable
    Dim sValor As String
    sValor = sNombres(2)

    ' mostrar en la consola el valor pasado a una variable
    ' y un valor directamente desde el array
    Console.WriteLine("Valor de la variable sValor: {0}", sValor)
```

```
    Console.WriteLine("Valor del array, posición 1: {0}", sNombres(1))
    Console.ReadLine()
End Sub
```

Código fuente 350

A lo largo de este texto, emplearemos de forma genérica el término array, para referirnos a este elemento del lenguaje. Por otra parte, recomendamos al lector la creación de un nuevo proyecto en el IDE de tipo consola, para realizar las pruebas mostradas a lo largo del tema.

La clase Array

Esta clase, perteneciente a la jerarquía de clases del sistema, es decir, incluida en el espacio de nombres System, proporciona a través de sus miembros, acceso orientado a objeto para los arrays que manipulemos en nuestras aplicaciones. Esto quiere decir que los arrays, como sucede con otros elementos del lenguaje, son también objetos.

Al igual que el resto de elementos del entorno, los arrays son tipos pertenecientes al sistema común de tipos de la plataforma o CTS, y se encuentran clasificados como tipos por referencia; esto quiere decir, que durante la ejecución, un array será gestionado en la zona de memoria conocida como montón o *heap*.

Aunque podemos trabajar con los arrays como objetos, no será necesario instanciar un objeto de esta clase para poder disponer de un array. Al declarar una variable como array, implícitamente se instancia un objeto de la clase. En sucesivos apartados de este tema, haremos una descripción de los miembros de instancia y compartidos más importantes de la clase Array.

Adecuación de los arrays en VB con los arrays de la plataforma .NET

Los arrays son uno de los elementos de VB que menos han evolucionado a lo largo de las sucesivas versiones aparecidas de este lenguaje. Todo esto, sin embargo, ha cambiado con la llegada de la plataforma .NET.

La especificación CLS del entorno .NET, dicta que todos los lenguajes que cumplan con la misma, podrán ser utilizados bajo .NET Framework. Esto quiere decir además, que dos ensamblados escritos en distintos lenguajes de la plataforma, podrán compartir código entre ellos. En el caso que nos ocupa, una aplicación VB.NET podrá llamar a un método de un objeto escrito en C# que devuelva un array, y dicho array, será manejado desde VB.NET.

Los diseñadores de .NET han realizado un gran esfuerzo en proporcionar la máxima optimización y versatilidad a los arrays, siempre y cuando, el lenguaje del entorno que los utilice, cumpla con unos mínimos requerimientos. En este aspecto, VB.NET como lenguaje, ha obtenido toda la potencia de base inherente en el sistema para la creación y manipulación de arrays; mientras que como contrapartida, ciertas características exclusivas en VB para el manejo de arrays han necesitado ser readaptadas. Algunas de estas características se describen a continuación.

El primer índice de un array debe ser siempre cero

VB.NET no soporta la instrucción Option Base, que nos permitía indicar que el primer índice de un array podía ser cero o uno. Por lo tanto, al declarar ahora un array en VB.NET, su número de elementos será el indicado en la declaración más uno. Veamos las diferencias en el Código fuente 351.

```
Código VB6
==========
Option Base 1

Public Sub Main()
       Dim sNombres(2) As String
       sNombres(1) = "Pedro"
       sNombres(2) = "Ana"
End Sub


Código VB.NET
=============
Public Sub Main()
       ' array de 3 elementos
       Dim sNombres(2) As String
       sNombres(0) = "Pedro"
       sNombres(1) = "Ana"
       sNombres(2) = "Jaime"
End Sub
```

Código fuente 351

No es posible crear arrays con rangos de índices

La característica que en VB6, nos permitía declarar un array en el que sus índices estuvieran en un intervalo, tampoco se admite ya en VB.NET. Al tener los arrays que comenzar por cero en su primer índice, los rangos de índices han dejado de tener sentido. Ver Código fuente 352.

```
Código VB6, no soportado en VB.NET
==================================
Public Sub Main()
    ' array de 4 elementos, entre los
    ' índices 5 y 8
    Dim Nombres(5 To 8) As String
    Nombres(5) = "Pedro"
    Nombres(6) = "Ana"
    Nombres(7) = "Jaime"
    Nombres(8) = "Elena"
End Sub
```

Código fuente 352

Todos los arrays son dinámicos

En VB6, dependiendo del modo de creación de un array, este podía ser estático, es decir, con un número fijo e invariable de elementos; o bien dinámico, es decir, su número de elementos podía ser modificado en tiempo de ejecución.

En VB.NET sin embargo, todos los arrays son de tamaño variable, tanto si se especifica como no un tamaño al ser declarados. En este punto debemos matizar un aspecto: cada vez que en VB.NET se cambia el tamaño de un array, el entorno internamente destruye el array actual, y crea un nuevo objeto de la clase Array, con el nuevo tamaño especificado, usando el mismo nombre de la variable correspondiente al array eliminado en primer lugar.

Declaración

Declararemos un array de igual forma que hacemos con una variable normal, con la excepción de que junto al nombre de la variable, situaremos unos paréntesis. Esto indica que dicha variable contiene un array. Opcionalmente, podemos especificar entre los paréntesis las dimensiones del array, o número de elementos que va a contener. Es posible también, realizar una asignación de valores al array en el mismo momento de su declaración. El Código fuente 353, muestra algunos ejemplos.

```
Sub Main()
    ' formas de declaración de arrays
    ' ==============================
    ' 1)
    ' estableciendo el número de elementos
    Dim sNombres(2) As String
    ' 2)
    ' asignando valores al array al mismo tiempo que se declara,
    ' la lista de valores debe ir encerrada entre llaves
    Dim sEstaciones() As String = {"Ana", "Pedro", "Luis"}
    ' 3)
    ' indicando el tipo de dato pero no el número de elementos,
    ' de este modo la variable todavía no es considerada un array
    ' ya que contiene una referencia a Nothing
    Dim iValores() As Integer
    ' 4)
    ' indicando el tipo de dato y estableciendo una
    ' lista vacía de elementos,
    ' a diferencia del caso anterior, la variable ahora sí
    ' es considerada un array aunque de longitud cero
    Dim iDatos() As Integer = {}
    ' 5)
    ' instanciando el tipo de dato, estableciendo el número
    ' de elementos al instanciar, e indicando que se trata de un array
    ' al situar las llaves
    Dim iCantidades() As Integer = New Integer(20) {}
    ' 6)
    ' declarar primero la variable que contendrá el array,
    ' asignar valores al array al mismo tiempo que se instancia
    ' la lista de valores debe ir encerrada entre llaves
    Dim iNumeros() As Integer
    iNumeros = New Integer() {10, 20, 30, 10, 50, 60, 10, 70, 80}
End Sub
```

Código fuente 353

Recomendamos al lector, que en estos ejemplos con arrays, utilice el depurador para ejecutar línea a línea el código, y abra la ventana Locales del depurador para ver en cada caso, el contenido de los elementos del array.

Asignación y obtención de valores

Para asignar u obtener valores de los elementos de un array, emplearemos la variable que contiene el array haciendo referencia al índice o posición a manipular. O bien, puesto que un array es un objeto, utilizaremos los métodos SetValue( ) y GetValue( ) que asignan y obtienen respectivamente los valores del array. Veamos un ejemplo en el Código fuente 354.

```
Sub Main()
    ' asignación de valores a los elementos de un array
    ' ==================================================
    Dim sNombres(4) As String
    ' directamente sobre la variable,
    ' haciendo referencia al índice
    sNombres(0) = "Juan"
    sNombres(1) = "Ana"
    sNombres(2) = "Luis"
    ' o con el método SetValue(), asignando el
    ' valor en el primer parámetro y especificando
    ' la posición en el segundo
    sNombres.SetValue("Elena", 3)
    sNombres.SetValue("Miguel", 4)

    ' obtención de valores de un array
    ' ================================
    Dim sValorA As String
    Dim sValorB As String
    sValorA = sNombres(2)  ' directamente de la variable
    sValorB = sNombres.GetValue(3)  ' usando el meth GetValue
    Console.WriteLine("Contenido de las variables")
    Console.WriteLine("==========================")
    Console.WriteLine("ValorA: {0} -- ValorB: {1}", sValorA, sValorB)
    Console.ReadLine()
End Sub
```

Código fuente 354

Recorrer el contenido

Para realizar un recorrido por los elementos de un array, disponemos de las funciones LBound( ) y UBound( ), que devuelven el número de índice inferior y superior respectivamente del array que pasemos como parámetro. No obstante, la orientación a objetos proporcionada por el entorno, pone a nuestra disposición el nuevo conjunto de características que comentamos seguidamente.

- **Length**. Esta propiedad de un objeto array devuelve el número de elementos que contiene.

- **GetLowerBound( ), GetUpperBound( )**. Estos métodos de un objeto array, devuelven respectivamente, el número de índice inferior y superior de una dimensión del array. El resultado es el mismo que usando LBound( ) y UBound( ), pero desde una perspectiva orientada a objetos.

- **Enumeradores**. Un objeto enumerador pertenece al interfaz IEnumerator, diseñado para realizar un recorrido o iteración a través de uno de los diferentes tipos de colección (arrays incluidos) existentes en .NET Framework. Mediante el método GetEnumerator( ) de un objeto array, obtenemos un objeto que implementa el interfaz Ienumerator, que sólo puede realizar labores de lectura sobre el array, en ningún caso de modificación.

La estructura de control utilizada para recorrer el array, puede ser indistintamente un bucle For...Next, For Each...Next, o la novedosa técnica de los objetos enumeradores proporcionados por el objeto array.

Como muestra de estas funcionalidades, el Código fuente 355 que vemos a continuación, contiene algunos ejemplos de cómo realizar una iteración sobre los elementos de un array.

```
Sub Main()
    ' recorrer un array
    ' =================
    Dim sNombres() As String = {"Ana", "Luis", "Pablo"}
    Dim iContador As Integer
    Dim sUnNombre As String

    ' modo tradicional
    Console.WriteLine("Recorrido del array con LBound() y UBound()")
    For iContador = LBound(sNombres) To UBound(sNombres)
        Console.WriteLine("Posicion: {0} - Valor: {1}", _
            iContador, sNombres(iContador))
    Next
    Console.WriteLine()

    ' con bucle For Each
    Console.WriteLine("Recorrido del array con bucle For Each")
    For Each sUnNombre In sNombres
        Console.WriteLine("Nombre actual: {0}", sUnNombre)
    Next
    Console.WriteLine()

    ' usando la propiedad Length
    Console.WriteLine("Recorrido del array con propiedad Length")
    For iContador = 0 To (sNombres.Length - 1)
        Console.WriteLine("Posicion: {0} - Valor: {1}", _
            iContador, sNombres(iContador))
    Next
    Console.WriteLine()

    ' usando los métodos GetLowerBound() y GetUpperBound()
    Console.WriteLine("Recorrido del array con métodos GetLowerBound() y
GetUpperBound()")
    For iContador = sNombres.GetLowerBound(0) To sNombres.GetUpperBound(0)
        Console.WriteLine("Posicion: {0} - Valor: {1}", _
            iContador, sNombres(iContador))
    Next
    Console.WriteLine()

    ' recorrer con un enumerador
    Console.WriteLine("Recorrido del array con un enumerador")
    Dim sLetras() As String = {"a", "b", "c", "d"}
    Dim oEnumerador As System.Collections.IEnumerator
    ' obtener el enumerador del array
    oEnumerador = sLetras.GetEnumerator()
    ' con un enumerador no es necesario posicionarse
    ' en el primer elemento ni calcular la cantidad
    ' de elementos del array, sólo hemos de avanzar
    ' posiciones con MoveNext() y obtener el valor
    ' actual con Current
    While oEnumerador.MoveNext()
        Console.WriteLine("Valor actual: {0}", oEnumerador.Current)
    End While
    Console.ReadLine()
End Sub
```

Código fuente 355.

Modificación de tamaño

Para aumentar o disminuir el número de elementos de un array disponemos de la palabra clave ReDim. Esta instrucción crea internamente un nuevo array, por lo que los valores del array original se pierden.

Evitaremos este problema utilizando junto a ReDim la palabra clave Preserve, que copia en el nuevo array, los valores del array previo. Veamos unos ejemplos en el Código fuente 356.

```
Sub Main()
    ' modificar el tamaño de un array
    ' ================================

    Dim sNombres(2) As String
    sNombres(0) = "Juan"
    sNombres(1) = "Pedro"
    sNombres(2) = "Elena"
    Console.WriteLine("Array sNombres original")
    MostrarArray(sNombres)

    ' ampliamos el número de elementos
    ' pero perdemos el contenido previo
    ReDim sNombres(4)
    sNombres(3) = "Isabel"
    sNombres(4) = "Raquel"
    Console.WriteLine("Array sNombres con tamaño ampliado")
    MostrarArray(sNombres)

    ' creamos otro array
    Dim sMasNombres(2) As String
    sMasNombres(0) = "Juan"
    sMasNombres(1) = "Pedro"
    sMasNombres(2) = "Miguel"
    Console.WriteLine("Array sMasNombres original")
    MostrarArray(sMasNombres)

    ' ampliamos el array sin perder elementos
    ReDim Preserve sMasNombres(4)
    sMasNombres(3) = "Antonio"
    sMasNombres(4) = "Paco"
    Console.WriteLine("Array sMasNombres ampliado sin perder valores")
    MostrarArray(sMasNombres)

    ' reducimos el array, pero sin perder los
    ' primeros elementos
    ReDim Preserve sMasNombres(1)
    Console.WriteLine("Array sMasNombres reducido")
    MostrarArray(sMasNombres)

    Console.ReadLine()
End Sub

Private Sub MostrarArray(ByVal sMiLista() As String)
    ' este es un procedimiento de apoyo que
    ' muestra el array pasado como parámetro
    Dim iContador As Integer
    For iContador = 0 To sMiLista.Length - 1
        Console.WriteLine("Elemento: {0} - Valor: {1}", _
            iContador, sMiLista(iContador))
    Next
    Console.WriteLine()
End Sub
```

Código fuente 356

Uso del método CreateInstance( ) para establecer el número de elementos en un array

Ya hemos comprobado que al crear un array en VB.NET, el primer índice es siempre cero, y además, el número de elementos del array es el indicado en la declaración más uno.

Sin embargo, la clase Array dispone del método compartido CreateInstance( ), que como su nombre indica, permite crear una nueva instancia de la clase, es decir un objeto array, con la particularidad de que en este caso, el número de elementos del array será realmente el que establezcamos al llamar a este método.

El Código fuente 357, muestra la diferencia entre crear un array del modo habitual, y empleando CreateInstance( ).

```
Sub Main()
    ' declarar un array del modo habitual:
    ' este array tiene cuatro elementos,
    ' desde el índice 0 al 3
    Dim sEstaciones(3) As String
    sEstaciones(0) = "Primavera"
    sEstaciones(1) = "Verano"
    sEstaciones(2) = "Otoño"
    sEstaciones(3) = "Invierno"
    Console.WriteLine("Array sEstaciones")
    MostrarArray(sEstaciones)

    ' crear un array instanciándolo
    ' con el método CreateInstance()
    ' de la clase Array
    Dim sColores As Array
    ' este array tendrá tres elementos reales
    ' que van desde el índice 0 hasta el 2
    sColores = Array.CreateInstance(GetType(String), 3)
    sColores(0) = "Azul"
    sColores(1) = "Rojo"
    sColores(2) = "Verde"
    Console.WriteLine("Array sColores")
    MostrarArray(sColores)

    Console.ReadLine()
End Sub

Private Sub MostrarArray(ByVal sMiLista() As String)
    ' muestra el array pasado como parámetro
    Dim iContador As Integer
    For iContador = 0 To sMiLista.Length - 1
        Console.WriteLine("Elemento: {0} - Valor: {1}", _
            iContador, sMiLista(iContador))
    Next
    Console.WriteLine()
End Sub
```

Código fuente 357

Paso de arrays como parámetros, y devolución desde funciones

Podemos pasar un array como parámetro a una rutina de código, teniendo en cuenta que los cambios que realicemos sobre el array en el procedimiento llamado, se mantendrán al volver el flujo de la ejecución al procedimiento llamador.

Ello es debido a que los arrays son tipos por referencia del entorno, y por lo tanto, las variables del array que manejamos tanto desde el procedimiento llamador, como desde el procedimiento llamado, son en realidad punteros hacia una misma zona de memoria o referencia, la que contiene el array.

En el ejemplo del Código fuente 358, comprobaremos que al pasar un array por valor, los cambios que realicemos sobre sus elementos se mantendrán al volver al procedimiento que hizo la llamada.

```
Sub Main()
    Dim iValores() As Integer = {10, 20, 30}
    ' en ambos casos, se pasa una referencia del array
    ManipArrayVal(iValores)
    ManipArrayRef(iValores)
    ' al volver de las llamadas a los procedimientos,
    ' el array ha sido modificado en ambas llamadas,
    ' independientemente de que haya sido pasado por
    ' valor o referencia
    MostrarArray(iValores)
    Console.ReadLine()
End Sub

' a este procedimiento le pasamos un array por valor
Private Sub ManipArrayVal(ByVal iListaPorValor As Integer())
    ' cambiar elemento del array
    iListaPorValor(0) = 888
End Sub

' a este procedimiento le pasamos un array por referencia
Private Sub ManipArrayRef(ByRef iListaPorReferencia As Integer())
    ' cambiar elemento del array
    iListaPorReferencia(2) = 457
End Sub

Private Sub MostrarArray(ByVal sMiLista() As Integer)
    ' muestra el array pasado como parámetro
    Dim iContador As Integer
    For iContador = 0 To sMiLista.Length - 1
        Console.WriteLine("Elemento: {0} - Valor: {1}", _
            iContador, sMiLista(iContador))
    Next
    Console.WriteLine()
End Sub
```

Código fuente 358

Clonación

Para evitar el problema planteado en el apartado anterior, si necesitamos disponer de un array con las mismas características que uno ya existente, y que sea totalmente independiente del primero, utilizaremos el método Clone( ).

Con esto solucionaremos el problema de que al pasar un array como parámetro, las modificaciones que precisemos realizar, afecten al array original. Veamos un ejemplo en el Código fuente 359.

```
Sub Main()
    ' crear un array
    Dim iValores() As Integer = {10, 20, 30}
    CambiaArray(iValores)

    ' mostrar el array original,
    ' en este no se habrán producido cambios
    Console.WriteLine("Array original")
    MostrarArray(iValores)

    Console.ReadLine()
End Sub

Private Sub CambiaArray(ByVal iListaDatos As Integer())
    ' crear un array clónico,
    ' cambiarle valores y mostrarlo
    Dim iListaClonada As Array
    iListaClonada = iListaDatos.Clone()
    iListaClonada(0) = 621
    iListaClonada(1) = 900
    Console.WriteLine("Array clónico")
    MostrarArray(iListaClonada)
End Sub

Private Sub MostrarArray(ByVal sMiLista() As Integer)
    Dim iContador As Integer
    For iContador = 0 To sMiLista.Length - 1
        Console.WriteLine("Elemento: {0} - Valor: {1}", _
            iContador, sMiLista(iContador))
    Next
    Console.WriteLine()
End Sub
```

Código fuente 359

Copia

Si intentamos copiar un array asignando la variable que contiene un array a otra, el resultado real serán dos variables que apuntan a la misma lista de valores, por lo que en definitiva sólo tendremos un array, al cual podremos acceder usando dos variables. Ello es debido a que como explicamos en un apartado anterior, los arrays son tipos por referencia que apuntan al mismo conjunto de valores.

Podemos clonar el array, como se ha descrito en el apartado anterior, con lo que obtendremos un nuevo array, que será idéntico al original.

O bien, podemos copiar el array utilizando los métodos CopyTo( ) y Copy( ) de la clase array. La diferencia con respecto a la clonación, consiste en que al copiar un array, el array destino ya debe estar creado con el número suficiente de elementos, puesto que los métodos de copia de la clase Array, lo que hacen es traspasar valores de los elementos del array origen al array destino, en función de los parámetros utilizados, copiaremos todos los elementos o un subconjunto. Veamos unos ejemplos en el Código fuente 360.

```
Sub Main()
```

```
    Dim sColores(3) As String
    sColores(0) = "Azul"
    sColores(1) = "Verde"
    sColores(2) = "Rosa"
    sColores(3) = "Blanco"
    MostrarArray(sColores)

    ' copiar usando el método CopyTo(),
    ' copiamos en el array sColorDestino,
    ' y comenzando por su posición 2, los
    ' valores del array sColores
    Dim sColorDestino(6) As String
    sColores.CopyTo(sColorDestino, 2)
    Console.WriteLine("Array sColorDestino")
    MostrarArray(sColorDestino)

    ' copiar usando el método Copy(),
    ' copiamos en el array sListaColores,
    ' a partir de su posición 2,
    ' 2 elementos del array sColores, comenzando
    ' desde la posición 1 de sColores
    Dim sListaColores(5) As String
    Array.Copy(sColores, 1, sListaColores, 2, 2)
    Console.WriteLine("Array sListaColores")
    MostrarArray(sListaColores)

    Console.ReadLine()
End Sub

Private Sub MostrarArray(ByVal sMiLista() As String)
    Dim iContador As Integer
    For iContador = 0 To sMiLista.Length - 1
        Console.WriteLine("Elemento: {0} - Valor: {1}", _
            iContador, sMiLista(iContador))
    Next
    Console.WriteLine()
End Sub
```

Código fuente 360

Inicialización de valores

Para inicializar o eliminar los valores de los elementos de un array, utilizaremos el método Clear, al que pasaremos el array a inicializar, el índice a partir del que comenzaremos, y el número de elementos.

Los valores serán inicializados en función del tipo de dato del array; cadena vacía en arrays String; cero en arrays numéricos, etc Veamos el Código fuente 361.

```
Sub Main()
    ' array String, asignar valores e inicializar
    Dim sLetras(2) As String
    sLetras(0) = "a"
    sLetras(1) = "b"
    sLetras(2) = "c"
    ' limpiar elementos en un array de tipo String,
    ' los elementos limpiados quedan como cadena vacía
    Array.Clear(sLetras, 0, 1)
    Console.WriteLine("Array sLetras")
    MostrarArray(sLetras)
```

```
    ' array Integer, asignar valores e inicializar
    Dim iNumeros() As Integer = {100, 200, 300, 400, 500, 600}
    ' limpiar elementos en un array de tipo Integer,
    ' los elementos limpiados se ponen a 0
    Array.Clear(iNumeros, 1, 2)
    Console.WriteLine("Array iNumeros")
    MostrarArrayNum(iNumeros)

    ' array Object, asignar valores e inicializar
    Dim oVarios(6) As Object
    oVarios(0) = "Hola"
    oVarios(1) = 456
    oVarios(2) = 1200
    oVarios(3) = #12/25/2001#
    oVarios(4) = 900
    oVarios(5) = True
    oVarios(6) = "adelante"
    ' al ser este un array de tipo Object
    ' los elementos limpiados se establecen a Nothing
    Array.Clear(oVarios, 3, 2)
    Console.WriteLine("Array oVarios")
    MostrarArrayObj(oVarios)

    Console.ReadLine()
End Sub

' recorrer un array de cadenas
Private Sub MostrarArray(ByVal sMiLista() As String)
    Dim iContador As Integer
    For iContador = 0 To sMiLista.Length - 1
        Console.WriteLine("Elemento: {0} - Valor: {1}", _
            iContador, sMiLista(iContador))
    Next
    Console.WriteLine()
End Sub

' recorrer un array de números
Private Sub MostrarArrayNum(ByVal iMiLista() As Integer)
    Dim iContador As Integer
    For iContador = 0 To iMiLista.Length - 1
        Console.WriteLine("Elemento: {0} - Valor: {1}", _
            iContador, iMiLista(iContador))
    Next
    Console.WriteLine()
End Sub

' recorrer un array de objetos
Private Sub MostrarArrayObj(ByVal oMiLista() As Object)
    Dim iContador As Integer
    For iContador = 0 To oMiLista.Length - 1
        Console.WriteLine("Elemento: {0} - Valor: {1}", _
            iContador, oMiLista(iContador))
    Next
    Console.WriteLine()
End Sub
```

Código fuente 361

Ordenación

Para ordenar un array disponemos del método Sort( ), que al estar sobrecargado, tiene varias implementaciones; la más básica de ellas es la que ordena la totalidad del array. También podemos ordenar una parte del array, indicando la posición inicial y cantidad de elementos a ordenar, etc.

El método Reverse( ), invierte la posición de todos o parte de los elementos de un array. En este punto, debemos matizar que no se realiza un orden inverso de los elementos, sino que se cambian las posiciones de los mismos. Ver Código fuente 362.

```
Sub Main()
    ' ordenar todo el array
    Dim sLetras1() As String = {"z", "a", "g", "m", "w", "i", "c", "b"}
    Array.Sort(sLetras1)
    Console.WriteLine("Ordenar todos el array")
    MostrarArray(sLetras1)

    ' ordenar parte del array
    Dim sLetras2() As String = {"z", "a", "g", "m", "w", "i", "c", "b"}
    Array.Sort(sLetras2, 4, 3)
    Console.WriteLine("Ordenar parte del array")
    MostrarArray(sLetras2)

    ' invertir valores dentro del array
    Dim sLetras3() As String = {"z", "a", "g", "m", "w", "i", "c", "b"}
    Array.Reverse(sLetras3, 2, 4)
    Console.WriteLine("Invertir valores del array")
    MostrarArray(sLetras3)

    Console.ReadLine()
End Sub

Private Sub MostrarArray(ByVal sMiLista() As String)
    Dim iContador As Integer
    For iContador = 0 To sMiLista.Length - 1
        Console.WriteLine("Elemento: {0} - Valor: {1}", _
            iContador, sMiLista(iContador))
    Next
    Console.WriteLine()
End Sub
```

Código fuente 362

Búsqueda

Los métodos IndexOf( ) y LastIndexOf( ) de la clase Array, nos permiten buscar un elemento en un array comenzando la búsqueda desde el principio o final respectivamente.

Ya que ambos disponen de diferentes implementaciones al estar sobrecargados, consulte el lector la documentación de la plataforma. El Código fuente 363 muestra algunos ejemplos de uso.

```
Sub Main()
    Dim sNombres() As String = {"Alberto", "Juan", "Ana", "Paco", "Miguel", "Ana"}

    ' buscar una cadena a partir del índice 0 del array
    Console.WriteLine("Paco está en la posición {0}", _
```

```
        Array.IndexOf(sNombres, "Paco"))

    ' buscar una cadena a partir del índice 3 del array
    Console.WriteLine("Ana está en la posición {0}," & _
        " comenzando a buscar desde índice 3", _
        Array.IndexOf(sNombres, "Ana", 3))

    ' introducir un valor a buscar en el array,
    ' si no existe se devuelve -1
    Dim iPosicionBuscar As Integer
    Console.WriteLine("Introducir nombre a buscar")
    iPosicionBuscar = Array.IndexOf(sNombres, _
        Console.ReadLine())

    If iPosicionBuscar = -1 Then
        Console.WriteLine("El nombre no está en el array")
    Else
        Console.WriteLine("El nombre está en la posición {0} del array", _
            iPosicionBuscar)
    End If

    ' buscar comenzando por la última posición
    Dim iNumeros() As Integer
    Dim iUltPosicionBuscar As Integer
    iNumeros = New Integer() {10, 20, 30, 10, 50, 60, 10, 70, 80}
    Console.WriteLine("El 10 está en la posición {0} comenzando por el final", _
        Array.LastIndexOf(iNumeros, 10))

    Console.ReadLine()
End Sub
```

Código fuente 363

Arrays multidimensionales

Todos los arrays vistos hasta el momento han sido de tipo unidimensional, es decir, estaban compuestos de una lista de valores única.

.NET Framework nos provee también de la capacidad de crear arrays formados por más de una lista de valores, o lo que es igual, arrays multidimensionales. Un array de este tipo, se caracteriza por estar compuesto de varias dimensiones o listas anidadas al estilo de filas y columnas.

Si declaramos un array del modo que muestra el Código fuente 364.

```
Dim iDatos(2, 4) As Integer
```

Código fuente 364

Crearíamos un array multidimensional formado por tres filas y cinco columnas. En este caso, el número correspondiente a la primera dimensión denota las filas, mientras que el número de la segunda dimensión hace lo propio para las columnas. La Figura 209 muestra un diagrama con la estructura de este array.

Dim iDatos(2, 4) As Integer

(0,0)	(0,1)	(0,2)	(0,3)	(0,4)
(1,0)	(1,1)	(1,2)	(1,3)	(1,4)
(2,0)	(2,1)	(2,2)	(2,3)	(2,4)

Figura 209. Estructura de un array multidimensional.

En este tipo de arrays, para acceder a los valores, debemos especificar la dimensión y la posición a la que vamos a asignar o recuperar un dato. Ver Código fuente 365.

```
Sub Main()
    ' crear array multidimensional y rellenar de valores
    Dim iDatos(2, 4) As Integer
    iDatos(0, 0) = 1000
    iDatos(0, 1) = 2000
    iDatos(0, 2) = 3000
    iDatos(0, 3) = 4000
    iDatos(0, 4) = 5000

    iDatos(1, 0) = 25
    iDatos(1, 1) = 35
    iDatos(1, 2) = 45
    iDatos(1, 3) = 55
    iDatos(1, 4) = 65

    iDatos(2, 0) = 111
    iDatos(2, 1) = 222
    iDatos(2, 2) = 333
    iDatos(2, 3) = 444
    iDatos(2, 4) = 555
End Sub
```

Código fuente 365

Para recorrer arrays multidimensionales, la clase Array dispone de varios miembros, algunos de los cuales, describimos seguidamente.

- **Rank**. Devuelve el número de dimensiones del array.
- **GetLength(Dimension)**. Devuelve el número de elementos de la dimensión de array pasada como parámetro.
- **GetLowerBound(Dimension)**. Devuelve el número de índice inferior de la dimensión pasada como parámetro.
- **GetUpperBound(Dimension)**. Devuelve el número de índice superior de la dimensión pasada como parámetro.

Vamos a completar el ejemplo anterior con las líneas del Código fuente 366, necesarias para recorrer el array multidimensional mostrado.

```
Sub Main()
    '....
    '....
    Dim iContadorDimUno As Integer
    Dim iContadorDimDos As Integer
    Dim sTextoFila As String

    ' poner títulos de la fila y columnas del array a mostrar
    Console.WriteLine("Fila" & ControlChars.Tab & _
        "Col 0" & ControlChars.Tab & "Col 1" & ControlChars.Tab & _
        "Col 2" & ControlChars.Tab & "Col 3" & ControlChars.Tab & "Col 4")

    ' el bucle externo recorre la primera dimensión
    For iContadorDimUno = iDatos.GetLowerBound(0) To iDatos.GetUpperBound(0)
        ' aquí obtenemos el número de fila
        ' que se está procesando
        sTextoFila = iContadorDimUno & ControlChars.Tab
        ' este bucle recorre la segunda dimensión
        For iContadorDimDos = iDatos.GetLowerBound(1) To iDatos.GetUpperBound(1)
            sTextoFila = sTextoFila & iDatos(iContadorDimUno, iContadorDimDos) & _
                ControlChars.Tab
        Next
        ' mostrar en la consola el contenido
        Console.WriteLine(sTextoFila)
        sTextoFila = ""
    Next
    Console.WriteLine()
    Console.WriteLine("El número de dimensiones es: {0}", iDatos.Rank)
    Console.WriteLine("El número total de elementos es: {0}", iDatos.Length)
    Console.ReadLine()
End Sub
```

Código fuente 366

24

Colecciones

Colecciones, la especialización de los arrays

En el tema sobre arrays hemos comprobado cómo a través de la clase Array, podemos manipular estos elementos del lenguaje con una mayor potencia y flexibilidad que en pasadas versiones del lenguaje.

No obstante, en muchas ocasiones nos encontraremos con situaciones en las que sería muy de agradecer que los arrays dispusieran de algunas características adicionales, dependiendo del problema que tengamos que resolver en ese preciso momento.

Por ejemplo, sería una gran idea poder manejar un array que creciera dinámicamente, sin tener que preocuparnos por aumentar o disminuir su tamaño; o también, disponer de un array a cuyos valores pudiéramos acceder, a través de identificadores claves, y no por el número de índice, que en algunas situaciones es más incómodo de manejar.

No se preocupe el lector, ya que no va a necesitar escribir complicados algoritmos para implementar estas características en sus arrays. Todas las funcionalidades mencionadas, y algunas más, se encuentran disponibles en un tipo especial de array denominado colección (collection).

Una colección es un objeto que internamente gestiona un array, pero que está preparado, dependiendo del tipo de colección, para manejar el array que contiene de una manera especial; podríamos definirlo como un array optimizado o especializado en ciertas tareas.

El espacio de nombres System.Collections

Este espacio de nombres del entorno de .NET Framework, es el encargado de agrupar el conjunto de clases e interfaces que nos permiten la creación de los distintos tipos de objetos collection. Por lo que si necesitamos un array con alguna característica especial, sólo hemos de instanciar un objeto de alguna de las clases de este espacio de nombres para disponer de un array con esa funcionalidad. Entre las clases más significativas de System.Collections, podemos destacar las siguientes.

- **ArrayList**. Proporciona una colección, cuyo array es redimensionado dinámicamente.
- **Hashtable**. Las colecciones de este tipo, contienen un array cuyos elementos se basan en una combinación de clave y valor, de manera que el acceso a los valores se facilita, al realizarse mediante la clave.
- **SortedList**. Consiste en una colección ordenada de claves y valores.
- **Queue**. Representa una lista de valores, en el que el primer valor que entra, es el primero que sale.
- **Stack**. Representa una lista de valores, en el que el último valor que entra, es el primero que sale.

Para hacer uso de colecciones en una aplicación VB.NET creada desde VS.NET, no es necesario importar este espacio de nombres, ya que como habrá observado el lector en ejemplos anteriores, el propio IDE incluye por defecto la importación del espacio System al proyecto.

La clave se halla en los interfaces

Las clases integrantes de System.Collections implementan en mayor o menor grado, un conjunto común de interfaces, que proporcionan la funcionalidad para el trabajo con arrays especializados o colecciones. Entre alguno de los interfaces de Collections, podemos mencionar los siguientes.

- **IEnumerable**. Proporciona el soporte para recorrer colecciones de valores.
- **ICollection**. Proporciona las características para manipular el tamaño, gestionar enumeradores, etc., de listas de valores.
- **IList**. Referencia a una lista de valores que puede ordenarse.
- **ICloneable**. Permite la creación de copias exactas e independientes de objetos.

Todo ello significa, que además de las clases con las funcionalidades especiales de Collections, podemos crear nuestras propias clases, para aquellos casos en los que necesitemos disponer de un array con funcionalidades especiales, no contempladas por los arrays base, y que además tampoco exista como colección. La manera de crear nuestro propio tipo de colección sería heredando de una clase collection existente y/o la implementación de alguno de los interfaces de Collections.

Seguidamente realizaremos una descripción general y pruebas con algunas de las colecciones existentes en el entorno, remitiendo al lector a la documentación de la plataforma accesible desde Visual Studio .NET para los detalles más específicos.

La clase ArrayList

Los objetos de tipo colección creados con esta clase, implementan un array cuyo número de elementos puede modificarse dinámicamente.

Instanciación de objetos ArrayList

Podemos instanciar una colección ArrayList en alguno los modos mostrados en el Código fuente 367. Observe el lector, cómo en el último ejemplo de este fuente, el constructor de ArrayList recibe como parámetro una *colección dinámica*.

```
Sub Main()
    ' crear una lista sin elementos
    Dim alEstaciones As New ArrayList()
    ' crear una lista indicando el número de elementos
    ' pero sin darles valor
    Dim alDatos As New ArrayList(3)
    ' crear una lista utilizando una colección dinámica
    Dim alLetras As New ArrayList(New String() {"a", "b", "c"})
End Sub
```

Código fuente 367

Una colección dinámica se crea de forma muy similar a un array, con la diferencia de que no es necesario usar una variable a la que asignar la colección, ya que en su lugar, se pasa como parámetro al constructor de ArrayList. El modo de creación de una colección dinámica consiste en utilizar la palabra clave New, seguida del tipo de dato de la colección, los paréntesis indicativos de array, y por último, encerrados entre llaves, los valores de la colección.

Agregar valores a un ArrayList

Una vez creado un ArrayList, podemos utilizar algunos de los métodos indicados a continuación para añadir valores a la colección.

- **Add(Valor)**. Añade el valor representado por Valor.
- **AddRange(Colección)**. Añade un conjunto de valores mediante un objeto del interfaz ICollection, es decir, una colección dinámica creada en tiempo de ejecución.
- **Insert(Posición, Valor)**. Inserta el valor Valor en la posición Posición del array, desplazando el resto de valores una posición adelante.
- **InsertRange(Posición, Colección)**. Inserta un conjunto de valores mediante una colección dinámica, a partir de una posición determinada dentro del array.
- **SetRange(Posición, Colección)**. Sobrescribe elementos en un array con los valores de la colección Colección, comenzando en la posición Posición.

El Código fuente 368 muestra algunos ejemplos de asignación de nuevos valores a un ArrayList.

```
Sub Main()
    Dim alDatos As New ArrayList(10)

    alDatos.Add("a")
    alDatos.AddRange(New String() {"b", "c", "d"})
    Console.WriteLine("ArrayList después de usar Add() y AddRange()")
    RecorrerAList(alDatos)

    alDatos.Insert(2, "hola")
    Console.WriteLine("ArrayList después de usar Insert()")
    RecorrerAList(alDatos)

    alDatos.InsertRange(1, New Integer() {55, 77, 88})
    Console.WriteLine("ArrayList después de usar InsertRange()")
    RecorrerAList(alDatos)

    alDatos.SetRange(3, New String() {"zzz", "yyy"})
    Console.WriteLine("ArrayList después de usar SetRange()")
    RecorrerAList(alDatos)

    Console.ReadLine()
End Sub

Private Sub RecorrerAList(ByVal alValores As ArrayList)
    Dim oEnumerador As IEnumerator = alValores.GetEnumerator()
    While oEnumerador.MoveNext()
        Console.WriteLine("Valor: {0}", oEnumerador.Current)
    End While
    Console.WriteLine()
End Sub
```

Código fuente 368

Los valores que espera recibir una colección son del tipo genérico Object, por lo que podemos insertar valores de diferentes tipos de dato.

Recorrer y obtener valores de un ArrayList

Para recorrer un array podemos emplear la técnica habitual del bucle For...Next y la propiedad Count del objeto ArrayList, que devuelve el número de elementos que tiene el objeto; o bien podemos usar un objeto del interfaz IEnumerator, proporcionado por el método GetEnumerator( ), mucho más simple de recorrer. Ver el Código fuente 369.

```
Sub Main()
    ' crear ArrayList y añadir valores
    Dim alLetras As New ArrayList(6)
    alLetras.Add("a")
    alLetras.AddRange(New String() {"b", "c", "d"})

    ' recorrer con un bucle For y usando la propiedad Count,
    ' tener en cuenta que al ser cero el primer índice del array,
    ' tenemos que restar uno a la propiedad Count
    Console.WriteLine("Recorrer objeto ArrayList con bucle For")
    Dim iContador As Integer
    For iContador = 0 To (alLetras.Count - 1)
        Console.WriteLine("Elemento actual {0}, valor: {1}", _
            iContador, alLetras(iContador))
    Next
```

```
    Console.WriteLine()

    ' recorrer el array con un enumerador
    Console.WriteLine("Recorrer objeto ArrayList con un enumerador")
    Dim oEnumerador As IEnumerator
    oEnumerador = alLetras.GetEnumerator()
    While oEnumerador.MoveNext()
        Console.WriteLine("Elemento de la lista: {0}", oEnumerador.Current())
    End While

    Console.ReadLine()
End Sub
```

Código fuente 369

Capacidad y valores en una colección ArrayList

Cuando manipulamos un objeto ArrayList debemos distinguir entre los conceptos capacidad y valores asignados.

La capacidad de un ArrayList hace referencia al número de elementos del array subyacente que contiene este objeto, mientras que los valores asignados se refieren a aquellos elementos del array a los que se ha asignado valor mediante métodos como Add( ) o AddRange( ). Podemos obtener esta información a través de las propiedades Capacity y Count del objeto colección. Ver Código fuente 370.

```
Console.WriteLine("Valores asignados al array: {0}", alLetras.Count)
Console.WriteLine("Capacidad del array: {0}", alLetras.Capacity)
```

Código fuente 370

La capacidad es un aspecto de la clase ArrayList que mejora el rendimiento a la hora de añadir o eliminar elementos del array. Analicemos esta característica con más detenimiento.

En primer lugar, todo objeto ArrayList dispone de una propiedad oculta llamada _items, conteniendo el array que internamente gestiona los valores asignados. Esta es una propiedad que no puede manipular el programador, pero que puede visualizar a través del depurador, abriendo la ventana Locales y expandiendo el contenido de un objeto ArrayList. Ver Figura 210.

Locales

Nombre	Valor	Tipo
alLetras	{System.Collections.ArrayList}	System.Collections.ArrayList
Object	{System.Collections.ArrayList}	Object
_defaultCapacity	16	Integer
_items	{Length=6}	Object()
(0)	"a"	String
(1)	"b"	String
(2)	"c"	String
(3)	"d"	String
(4)	Nothing	Object
(5)	Nothing	Object
_size	4	Integer
_version	2	Integer
Capacity	6	Integer
Count	4	Integer

Figura 210. Propiedad _items de un objeto ArrayList.

Cuando creamos un objeto ArrayList con un tamaño como el del último ejemplo, la acción de añadir un valor a la colección no redimensiona su array subyacente, puesto que ya está creado con un tamaño determinado, sino que asigna un valor al siguiente elemento libre que no hubiera sido previamente asignado. Veámoslo en el esquema de la Figura 211.

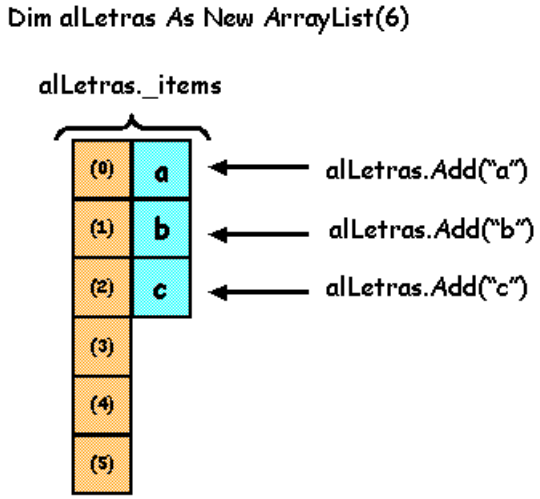


Figura 211. Asignación de valores al array subyacente de una colección ArrayList.

Este comportamiento del objeto tiene la ventaja de que mejora el rendimiento y optimiza recursos, puesto que cada vez que añadimos o eliminamos valores, el array _items no siempre tiene que ser redimensionado.

¿Qué sucede, sin embargo, cuando se han añadido valores y el array está completo?, pues que el objeto ArrayList detecta esta situación y en la siguiente ocasión en que se añade un nuevo valor, automáticamente redimensiona el array _items, duplicando el número de elementos inicial que contenía. La Figura 212 muestra un esquema con los pasos de este proceso.

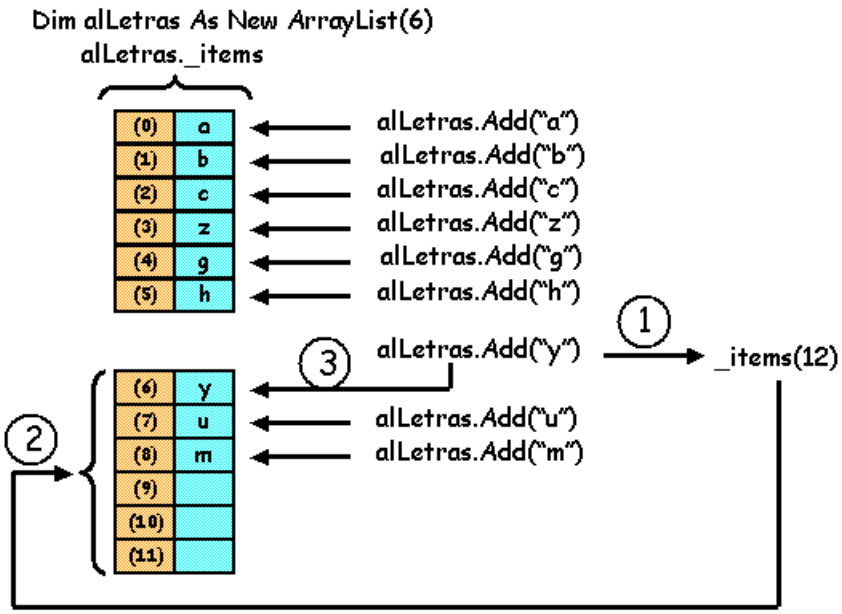


Figura 212. Redimensión automática del array _items de un objeto ArrayList.

En el caso que muestra la anterior figura, después de añadir la letra m al objeto, la propiedad Capacity devolvería 12 y la propiedad Count devolvería 9.

Un detalle muy importante que debe tener en cuenta el lector, es que al crear un objeto ArrayList, si no especificamos el tamaño, la propiedad _items tiene una capacidad por defecto de 16 elementos.

Obtención de subarrays a partir de un objeto ArrayList

La clase ArrayList nos proporciona métodos tanto para obtener un fragmento o rango (subarray) de un objeto ArrayList, como para crear nuevos objetos mediante métodos shared o compartidos. Entre este tipo de métodos, se encuentran los siguientes.

- **GetRange(Posición, Elementos)**. Obtiene un subarray comenzando en el índice Posición, y tomando el número que indica Elementos.
- **FixedSize(ArrayList)**. Método compartido que crea un array de tamaño fijo a partir de un objeto ArrayList pasado como parámetro. Sobre el nuevo array obtenido, podemos modificar los elementos existentes, pero no añadir nuevos.
- **Repeat(Valor, Cantidad)**. Método compartido que crea un ArrayList de valores repetidos, tomando como valor a repetir el parámetro Valor, y creando tantos elementos como se especifica en el parámetro Cantidad.
- **ToArray()**. Copia los elementos del ArrayList en un objeto Array, al ser ambos arrays independientes, el objeto sobre el que se han copiado los elementos puede modificarse sin que afecte al ArrayList.
- **ReadOnly( )**. Método compartido que crea un objeto ArrayList de sólo lectura a partir de un array existente.

Veamos algunos ejemplos de uso de estos métodos en el Código fuente 371.

```
Sub Main()
    Dim alLetras As New ArrayList(10)
    alLetras.AddRange(New String() {"a", "b", "c", "d", "e", "f", "g"})

    Console.WriteLine("Array alLetras")
    RecorrerAList(alLetras)

    ' obtener un subarray con un rango determinado
    Dim alRangoLetras As ArrayList
    alRangoLetras = alLetras.GetRange(4, 2)
    Console.WriteLine("Array alRangoLetras")
    RecorrerAList(alRangoLetras)

    ' obtener un subarray de tamaño fijo,
    ' se pueden modificar sus elementos,
    ' no se pueden añadir valores
    Dim alLetrasFijo As ArrayList = ArrayList.FixedSize(alLetras)
    'alLetrasFijo.Add("m")  <-- esto provocaría error
    alLetrasFijo(2) = "vvv"
    Console.WriteLine("Array alLetrasFijo")
    RecorrerAList(alLetrasFijo)
```

```
    ' ArrayList de valores repetidos
    Dim alRepetidos As ArrayList
    alRepetidos = ArrayList.Repeat("hola", 3)
    alRepetidos.Add("otro valor")
    Console.WriteLine("Array alRepetidos")
    RecorrerAList(alRepetidos)

    ' copiar ArrayList sobre un array normal
    Dim aNuevo As Array
    aNuevo = alLetras.ToArray()
    aNuevo(2) = "zzz"
    Console.WriteLine("Array aNuevo")
    RecorrerArray(aNuevo)

    ' crear ArrayList de sólo lectura
    ' a partir de un array existente
    Dim alLetrasSoloLeer As ArrayList = ArrayList.ReadOnly(alLetras)
    ' solamente podemos recorrerlo
    Console.WriteLine("ArrayList de sólo lectura")
    RecorrerAList(alLetrasSoloLeer)

    ' las dos líneas siguientes provocarían un error
    'alLetrasSoloLeer.Add("yyy")
    'alLetrasSoloLeer(2) = "wwer"

    Console.ReadLine()
End Sub

Private Sub RecorrerAList(ByVal alValores As ArrayList)
    Dim oEnumerador As IEnumerator = alValores.GetEnumerator()
    While oEnumerador.MoveNext()
        Console.WriteLine("Valor: {0}", oEnumerador.Current)
    End While
    Console.WriteLine()
End Sub

Private Sub RecorrerArray(ByVal aValores As Array)
    Dim oEnumerador As IEnumerator = aValores.GetEnumerator()
    While oEnumerador.MoveNext
        Console.WriteLine("Valor: {0}", oEnumerador.Current)
    End While
    Console.WriteLine()
End Sub
```

Código fuente 371

Respecto a los ArrayList de tamaño fijo, tipo FixedSize, queremos advertir al lector que a la hora de ver su contenido en el depurador, no debe consultar la propiedad _items mencionada anteriormente, ya que esta contendrá un array de valores vacíos.

Este punto puede crear confusión, ya que el lector al ejecutar la aplicación sí obtendrá los valores de la colección FixedSize, por lo que se preguntará dónde han ido a parar esos valores.

El secreto reside en el siguiente lugar: al ejecutar con el depurador, debemos expandir la propiedad [System.Collections.ArrayList.FixedSizeArrayList] del ArrayList con tamaño fijo. Dentro de esta propiedad, que es realmente una variante del objeto ArrayList, abriremos la propiedad _list, y de nuevo dentro de esta propiedad encontraremos otra con el nombre _items, la cual será la que contiene realmente los valores del array de tamaño fijo. La Figura 213 muestra la ventana Locales del depurador ejecutando el ejemplo.

Locales

Nombre	Valor	Tipo
alLetrasFijo	{System.Collections.ArrayList.FixedSizeArrayL	System.Collections.ArrayList
[System.Collections.ArrayList.FixedSizeArrayList]	{System.Collections.ArrayList.FixedSizeArrayL	System.Collections.ArrayList.FixedSizeArrayList
System.Collections.ArrayList	{System.Collections.ArrayList.FixedSizeArrayL	System.Collections.ArrayList
_list	{System.Collections.ArrayList}	System.Collections.ArrayList
Object	{System.Collections.ArrayList}	Object
_defaultCapacity	16	Integer
_items	{Length=10}	Object()
(0)	"a"	String
(1)	"b"	String
(2)	"yyy"	String
(3)	"d"	String
(4)	"e"	String
(5)	"f"	String
(6)	"g"	String
(7)	Nothing	Object
(8)	Nothing	Object
(9)	Nothing	Object

Figura 213. Acceso a la lista de valores de un ArrayList de tipo FixedSize.

Búsquedas en colecciones ArrayList

Además de los métodos IndexOf( ) y LastIndexOf( ), disponibles en los arrays estándar, un ArrayList aporta el método Contains( ), que nos devuelve un valor lógico indicando si el valor pasado como parámetro existe en el array. Veamos unos ejemplos en el Código fuente 372.

```
Sub Main()
    Dim alLetras As New ArrayList(10)
    alLetras.AddRange(New String() {"jj", "oo", "aa", _
        "jj", "ee", "tt", "mm", "xx"})

    Dim iPosicIndexOf As Integer
    ' buscar un elemento de la colección desde el principio
    iPosicIndexOf = alLetras.IndexOf("aa")
    Console.WriteLine("Posición al buscar con IndexOf: {0}", iPosicIndexOf)

    Dim iPosicLastIndexOf As Integer
    ' buscar un elemento de la colección desde el final
    iPosicLastIndexOf = alLetras.LastIndexOf("jj")
    Console.WriteLine("Posición al buscar con LastIndexOf: {0}", _
        iPosicLastIndexOf)

    Dim bEncontrado As Boolean
    ' comprobar si existe un valor en la colección
    bEncontrado = alLetras.Contains("oo")
    Console.WriteLine("Resultado de la búsqueda con Contains: {0}", bEncontrado)

    Console.ReadLine()
End Sub
```

Código fuente 372

Borrado de elementos en una colección ArrayList

Para realizar un borrado de valores, la clase ArrayList proporciona los métodos descritos a continuación.

- **Remove(Valor)**. Elimina el elemento del array que corresponde a Valor.
- **RemoveAt(Posicion)**. Elimina el elemento del array situado en el índice Posición.
- **RemoveRange(Posición, Elementos)**. Elimina el conjunto de elementos indicados en el parámetro Elementos, comenzando por el índice Posición.
- **Clear( )**. Elimina todos los elementos del objeto.

Debido a las causas de optimización en cuanto a ejecución comentadas en un apartado anterior, al borrar elementos de un ArrayList, el valor de la propiedad Capacity permanece inalterable. Eso quiere decir que si añadimos muchos valores a un ArrayList para después eliminar gran parte de ellos, el array subyacente de la propiedad _items no disminuirá su tamaño para ahorrar recursos. Esto no es algo preocupante sin embargo, puesto que si utilizamos el método TrimToSize( ) de esta clase, conseguiremos que el tamaño o capacidad del ArrayList se ajuste al número real de elementos que contiene. El Código fuente 373 muestra algunos ejemplos de borrados sobre este tipo de objeto.

```
Sub Main()
    ' borra todos los elementos
    Dim alLetras1 As New ArrayList(10)
    alLetras1.AddRange(New String() {"a", "b", "c", "d", "e", "f", "g"})
    alLetras1.Clear()
    Estado("alLetras1", alLetras1)

    ' borra un elemento por el valor
    Dim alLetras2 As New ArrayList(10)
    alLetras2.AddRange(New String() {"a", "b", "c", "d", "e", "f", "g"})
    alLetras2.Remove("c")
    Estado("alLetras2", alLetras2)

    ' borra un elemento por posición
    Dim alLetras3 As New ArrayList(10)
    alLetras3.AddRange(New String() {"a", "b", "c", "d", "e", "f", "g"})
    alLetras3.RemoveAt(5)
    Estado("alLetras3", alLetras3)

    ' borra un rango de elementos por posición
    Dim alLetras4 As New ArrayList(10)
    alLetras4.AddRange(New String() {"a", "b", "c", "d", "e", "f", "g"})
    alLetras4.RemoveRange(2, 3)
    Console.WriteLine("Array alLetras4: estado antes de reajustar tamaño")
    Estado("alLetras4", alLetras4)

    ' reajustar capacidad del array
    alLetras4.TrimToSize()
    Console.WriteLine("Array alLetras4: estado después de reajustar tamaño")
    Estado("alLetras4", alLetras4)

    Console.ReadLine()
End Sub

Public Sub Estado(ByVal sNombre As String, ByVal alValores As ArrayList)
    Console.WriteLine("Array: {0} / Capacidad: {1} / Número de elementos: {2}", _
        sNombre, alValores.Capacity, alValores.Count)
    Console.WriteLine()
End Sub
```

Código fuente 373

Ordenar elementos en un objeto ArrayList

Al igual que en la clase base Array, los objetos ArrayList disponen del método Sort( ), que ordena los elementos del array; y del método Reverse( ), que invierte las posiciones de un número determinado de elementos del array. El Código fuente 374 muestra un ejemplo de cada uno.

```
Sub Main()
    Dim alLetras As New ArrayList(10)
    alLetras.AddRange(New String() {"z", "t", "c", "a", "k", "f", "m"})

    ' ordenar
    alLetras.Sort()
    Console.WriteLine("ArrayList después de ordenar")
    RecorrerAList(alLetras)

    ' invertir posiciones de elementos
    alLetras.Reverse(3, 3)
    Console.WriteLine("ArrayList después de invertir elementos")
    RecorrerAList(alLetras)

    Console.ReadLine()
End Sub

Private Sub RecorrerAList(ByVal alValores As ArrayList)
    Dim oEnumerador As IEnumerator = alValores.GetEnumerator()
    While oEnumerador.MoveNext()
        Console.WriteLine("Valor: {0}", oEnumerador.Current)
    End While
    Console.WriteLine()
End Sub
```

Código fuente 374

La clase Hashtable

Esta clase tiene la particularidad de que el acceso a los valores del array que gestiona internamente se realiza a través de una clave asociada a cada elemento, al estilo de los objetos Dictionary de versiones anteriores de VB. Como dato significativo, esta clase implementa el interfaz IDictionary, por lo que si hemos utilizado anteriormente objetos Dictionary, ya conocemos gran parte de su filosofía de trabajo.

En este tipo de colección no es necesario preocuparse por la posición o índice de los elementos, ya que accedemos a ellos a través de literales, lo cual en algunas circunstancias es mucho más cómodo de manejar.

Manejo básico de colecciones Hashtable

Supongamos como ejemplo, que un proceso en una aplicación necesita guardar de forma temporal, datos de un cliente en un array. Podemos naturalmente utilizar un array estándar para tal fin, como muestra el Código fuente 375.

```
Dim aCliente = Array.CreateInstance(GetType(Object), 6)
aCliente(0) = 22
aCliente(1) = "Pedro"
```

```
aCliente(2) = "Naranjo"
aCliente(3) = "C/Rio Bravo, 25"
aCliente(4) = 35
aCliente(5) = 250
```

Código fuente 375

En un planteamiento como el anterior, debemos acordarnos, a la hora de obtener los datos del array, que la primera posición corresponde al código de cliente, la siguiente al nombre, etc.,. Bien es cierto que podemos utilizar constantes numéricas para cada posición, pero sigue siendo una solución poco flexible.

Utilizando un objeto Hashtable sin embargo, tenemos la ventaja de que no necesitamos saber la posición en que se encuentra cada valor, ya que precisamente a cada posición le damos un nombre clave que es mediante el que accedemos posteriormente cuando queremos obtener los valores. El Código fuente 376, mejora sustancialmente el fuente del caso anterior, al estar basado en una colección Hashtable.

```
Sub Main()
    ' declarar colección Hashtable
    Dim htCliente As Hashtable
    htCliente = New Hashtable()

    ' añadir valores
    htCliente.Add("ID", 22)
    htCliente.Add("Nombre", "Pedro")
    htCliente.Add("Apellidos", "Naranjo")
    htCliente.Add("Domicilio", "C/Rio Bravo, 25")
    htCliente.Add("Edad", 35)
    htCliente.Add("Credito", 250)

    ' mostrar el número de elementos que contiene
    Console.WriteLine("El objeto Hashtable tiene {0} elementos", _
        htCliente.Count)
    Console.WriteLine()

    ' obtener algunos valores
    Console.WriteLine("Obtener algunos valores del objeto Hashtable")
    Console.WriteLine("Domicilio: {0} ", htCliente.Item("Domicilio"))
    Console.WriteLine("Nombre: {0} ", htCliente("Nombre"))
    Console.WriteLine("Credito: {0} ", htCliente("Credito"))
    Console.WriteLine()

    ' recorrer el array al completo
    Console.WriteLine("Recorrer objeto Hashtable con un enumerador")
    Dim oEnumerador As IDictionaryEnumerator
    oEnumerador = htCliente.GetEnumerator()
    While oEnumerador.MoveNext()
        Console.WriteLine("Clave: {0} / Valor: {1}", _
            oEnumerador.Key, oEnumerador.Value)
    End While

    Console.ReadLine()
End Sub
```

Código fuente 376

Para crear un nuevo objeto Hashtable hemos utilizado el constructor sin parámetros, que es el más básico disponible. Puesto que se trata de un constructor sobrecargado, sugerimos al lector que consulte la documentación de .NET Framework para ver una lista completa de todos los constructores disponibles.

Respecto a la asignación de valores a la colección, esta clase utiliza el método Add( ), cuyo primer parámetro corresponde a la clave del elemento y el segundo corresponde al valor que vamos a asignar a la posición de su array.

Para obtener un valor del array utilizamos el método Item( ), pasando como parámetro el nombre de la clave de la que vamos a obtener el valor correspondiente. Al tratarse del método por defecto de esta clase no es necesario especificarlo. Como hemos podido comprobar, el resultado es el mismo tanto si especificamos como si no el nombre del método.

Si queremos recorrer el array al completo contenido en el objeto, podemos utilizar el método GetEnumerator( ), que nos devuelve un enumerador para poder obtener todos los valores del objeto en un bucle. La diferencia con los otros algoritmos de recorrido que hemos visto anteriormente, es que al estar la colección Hashtable basada en una combinación de clave/valor, necesitamos un enumerador basado en el interfaz IDictionaryEnumerator, especialmente adaptado para manipular arrays de este tipo.

Como habrá comprobado el lector, un enumerador IDictionaryEnumerator proporciona la información de la colección mediante las propiedades Key y Value, a diferencia de un enumerador simple, basado en IEnumerator, en el que usamos la propiedad Current, para extraer los datos.

La clase Hashtable no sitúa los valores que se añaden al array en posiciones consecutivas, por lo que al obtener los valores mediante un enumerador posiblemente no aparecerán en el mismo orden en el que los añadimos inicialmente. Dada la filosofía de funcionamiento de este tipo de objetos, el orden en el que se graban los valores dentro del array no debería ser un problema, ya que nosotros accedemos al array utilizando claves y no índices, como sucede en un array estándar.

Operaciones varias con colecciones Hashtable

Antes de obtener valores de un objeto Hashtable, podemos comprobar que la clave o valor que necesitamos están realmente en la colección, empleando respectivamente los métodos ContainsKey( ) y ContainsValue( ). Al primero le pasamos como parámetro la clave a buscar, mientras que al segundo le pasamos el valor. Ambos métodos devuelven un resultado de tipo Boolean, que indica si la comprobación tuvo o no éxito.

Por otra parte, el método Remove( ), elimina un elemento del objeto, pasándole como parámetro la clave a borrar, mientras que el método Clear( ), elimina el contenido completo de la colección. El Código fuente 377 muestra un ejemplo.

```
Sub Main()
    ' crear colección Hashtable y añadir valores
    Dim htCliente As New Hashtable()
    htCliente.Add("ID", 22)
    htCliente.Add("Nombre", "Pedro")
    htCliente.Add("Apellidos", "Naranjo")
    htCliente.Add("Domicilio", "C/Rio Bravo, 25")
    htCliente.Add("Edad", 35)
    htCliente.Add("Credito", 250)

    ' comprobar la existencia de elementos por la clave
```

```
    Dim sClave As String
    Console.WriteLine("Introducir la clave a buscar")
    sClave = Console.ReadLine()
    If htCliente.ContainsKey(sClave) Then
        Console.WriteLine("La clave {0} existe, su valor es: {1}", _
            sClave, htCliente(sClave))
    Else
        Console.WriteLine("La clave {0} no existe", sClave)
    End If
    Console.WriteLine()

    ' comprobar la existencia de elementos por el valor
    Dim oValor As Object
    Console.WriteLine("Introducir el valor a buscar")
    oValor = Console.ReadLine()
    ' antes de comprobar si existe el valor
    ' debemos hacer una conversión al tipo
    ' específico de dato del contenido de
    ' la variable oValor
    If IsNumeric(oValor) Then
        oValor = CType(oValor, Integer)
    Else
        oValor = CType(oValor, String)
    End If
    ' ahora comprobamos
    If htCliente.ContainsValue(oValor) Then
        Console.WriteLine("El valor {0} existe", oValor)
    Else
        Console.WriteLine("El valor {0} no existe", oValor)
    End If

    ' para borrar un elemento se usa la clave
    htCliente.Remove("Nombre")
    Console.WriteLine()
    RecorrerHT(htCliente)

    ' ahora borramos el contenido al completo del objeto
    htCliente.Clear()

    Console.ReadLine()
End Sub

Public Sub RecorrerHT(ByVal htTabla As Hashtable)
    Dim oEnumerador As IDictionaryEnumerator
    oEnumerador = htTabla.GetEnumerator()
    While oEnumerador.MoveNext()
        Console.WriteLine("Clave: {0} / Valor: {1}", _
            oEnumerador.Key, oEnumerador.Value)
    End While
End Sub
```

Código fuente 377

Las propiedades Keys y Values de la clase Hashtable, devuelven un array con los nombres de las claves y los valores de un objeto Hashtable respectivamente.

Realmente devuelven un objeto del interfaz ICollection, pero ya que un array implementa este interfaz, dicho objeto podemos manipularlo como un array.

Seguidamente mostramos un ejemplo del uso de estas propiedades en el Código fuente 378. Observe el lector el diferente modo de declarar y obtener los objetos ICollection e IEnumerator, que en definitiva, nos llevan al mismo resultado, demostrando así, la gran flexibilidad sintáctica que la especificación CLS proporciona al lenguaje.

```
Sub Main()
    ' crear y llenar la colección
    Dim htCliente As New Hashtable()
    htCliente.Add("ID", 22)
    htCliente.Add("Nombre", "Pedro")
    htCliente.Add("Apellidos", "Naranjo")
    htCliente.Add("Domicilio", "C/Rio Bravo, 25")
    htCliente.Add("Edad", 35)
    htCliente.Add("Credito", 250)

    ' obtener un array con las claves,
    ' y recorrer con un enumerador
    Dim aClaves As ICollection
    aClaves = htCliente.Keys
    Dim oEnumerador As IEnumerator
    oEnumerador = aClaves.GetEnumerator()
    Console.WriteLine("Claves del objeto Hashtable")
    RecorrerEnumerador(oEnumerador)

    ' obtener un array con los valores,
    ' y recorrer con un enumerador
    Dim aValores As ICollection = htCliente.Values
    Dim oEnumVal As IEnumerator = aValores.GetEnumerator()
    Console.WriteLine("Valores del objeto Hashtable")
    RecorrerEnumerador(oEnumVal)

    Console.ReadLine()
End Sub

Public Sub RecorrerEnumerador(ByVal oEnumerador As IEnumerator)
    While oEnumerador.MoveNext
        Console.WriteLine(oEnumerador.Current)
    End While
    Console.WriteLine()
End Sub
```

Código fuente 378

Traspaso de elementos desde una colección Hashtable a un array básico

El método CopyTo( ) de la clase Hashtable, se encarga de traspasar a un array estándar un determinado número de elementos desde un objeto Hashtable. Como ya sablemos, cada elemento de una colección de este tipo está formado realmente por dos componentes, clave y valor; por ello, cabe preguntarse cómo se las ingenia esta clase para poder traspasarlos a un array normal.

La respuesta es la siguiente: lo que realmente se traspasa al array normal no son valores independientes, sino una estructura de tipo DictionaryEntry, cuyas propiedades son precisamente Key y Value, es decir, la clave y valor del elemento de la colección.

El Código fuente 379 muestra un ejemplo, en el cual, después de crear un objeto Hashtable con sus correspondientes valores, creamos un array normal de tipo Object al que traspasamos los elementos desde el Hashtable.

```
Sub Main()
    ' crear un array Hashtable
    Dim htCliente As New Hashtable()
```

```
    htCliente.Add("ID", 22)
    htCliente.Add("Nombre", "Pedro")
    htCliente.Add("Apellidos", "Naranjo")
    htCliente.Add("Domicilio", "C/Rio Bravo, 25")
    htCliente.Add("Edad", 35)
    htCliente.Add("Credito", 250)

    ' crear un array estándar
    Dim aDeposito(10) As Object
    aDeposito(0) = "aaa"
    aDeposito(1) = "bbb"
    aDeposito(2) = "ccc"
    aDeposito(3) = "ddd"

    ' volcar elementos del Hashtable al array estándar,
    ' comenzando a copiar desde una posición determinada
    ' del array estándar
    htCliente.CopyTo(aDeposito, 2)

    ' obtener algunos valores del objeto Hashtable
    ' que se han pasado al array normal
    Dim oEntradaDicc As DictionaryEntry
    Dim iContador As Integer
    For iContador = 3 To 6
        oEntradaDicc = aDeposito(iContador)
        Console.WriteLine("Clave: {0} / Valor: {1}", _
            oEntradaDicc.Key, oEntradaDicc.Value)
    Next

    Console.ReadLine()
End Sub
```

Código fuente 379

Como el array normal tenía valores asignados previamente, algunos de ellos se pierden, puesto que el método CopyTo( ) no redimensiona el array normal. Tenga este hecho en cuenta el lector, ya que el array destino deberá tener el suficiente tamaño cuando traspasamos valores desde una colección de este tipo.

La clase SortedList

Esta clase es una variación de Hashtable, ya que nos permite crear colecciones basadas en pares de claves y valor, pero con la diferencia de que en una colección SortedList, los elementos se ordenan por la clave según van siendo agregados. El funcionamiento general, es básicamente igual que para los objetos Hashtable. Veamos un ejemplo en el Código fuente 380.

```
Sub Main()
    ' crear una colección
    Dim slListaOrden As New SortedList()

    ' según añadimos elementos, se van
    ' ordenando dinámicamente
    slListaOrden.Add("H", 111)
    slListaOrden.Add("A", 222)
    slListaOrden.Add("Z", 333)
    slListaOrden.Add("M", 444)

    ' recorrer la colección SortedList
    Dim oEnumerador As IDictionaryEnumerator
```

```
    oEnumerador = slListaOrden.GetEnumerator()
    While oEnumerador.MoveNext()
        Console.WriteLine("Clave: {0} / Valor: {1}", _
            oEnumerador.Key, oEnumerador.Value)
    End While

    Console.ReadLine()
End Sub
```

Código fuente 380

La clase Queue

Esta clase implementa toda la operativa de una lista de tipo FIFO (first in, first out), primero en entrar/primero en salir; también denominada *cola de valores*. A pesar de lo básico de su funcionalidad, es un aspecto que el programador agradece en muchas situaciones. La Figura 214 muestra un esquema del funcionamiento de este tipo de objeto.

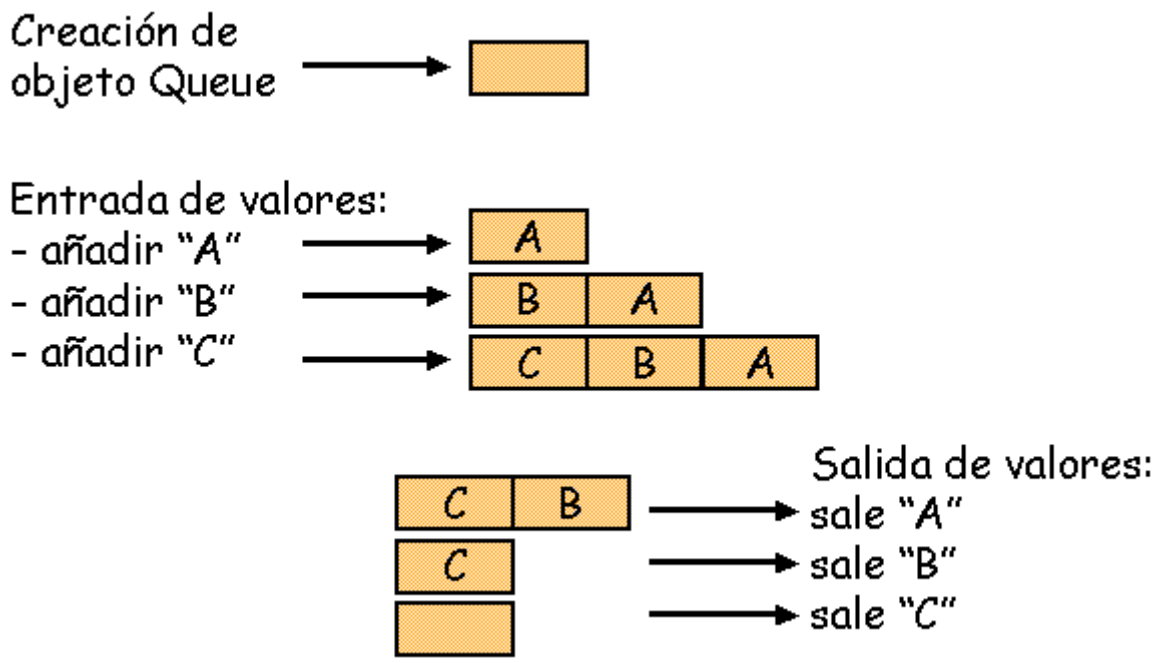


Figura 214. Representación gráfica del funcionamiento de una colección Queue.

Como el resto de clases en el espacio de nombres System.Collections, los objetos de tipo Queue contienen un array gestionado por la propia clase e inaccesible para el programador, que almacena los valores que vamos *encolando*. Este array parte con un tamaño inicial y es redimensionado por el propio objeto cuando todos sus elementos han sido asignados y se intenta añadir un nuevo valor.

Manipulación de valores en una colección Queue

El método Enqueue( ) añade nuevos valores, que quedan situados al final de la lista, del modo ilustrado en la figura anterior.

Para desarrollar un ejemplo de uso de la clase Queue, vamos a plantear la siguiente situación: necesitamos implementar un sistema de recepción y envío de mensajes. El primer mensaje entrante se situará en la lista de modo que será el primero en enviarse, el segundo mensaje recibido será el siguiente enviado y así sucesivamente.

El Código fuente 381 mostrado a continuación, ilustra el modo de introducción de valores, en un estilo diferente al utilizado en los anteriores ejemplos. En este caso, en lugar de introducir directamente por código los valores en la lista, utilizaremos el método ReadLine( ) del objeto Console, de manera que el usuario introduzca los valores que precise. Cuando pulse [INTRO] sin haber escrito valor alguno, se considerará que ha terminado la introducción de datos.

```
Sub Main()
    ' crear objeto Queue, cola de valores
    Dim aqListaMensa As New Queue()

    Console.WriteLine("Introducir mensajes")
    Dim sMensaje As String
    ' bucle de recepción de mensajes
    Do
        sMensaje = Console.ReadLine()
        ' si hemos escrito algo...
        If sMensaje.Length > 0 Then
            ' añadimos a la cola
            aqListaMensa.Enqueue(sMensaje)
        Else
            ' salimos
            Exit Do
        End If
    Loop

    ' la propiedad Count nos indica la cantidad de
    ' elementos en la lista
    Console.WriteLine("Hay {0} mensajes para procesar", aqListaMensa.Count)

    ' con un enumerador recorremos la lista
    Dim oEnumerador = aqListaMensa.GetEnumerator()
    Console.WriteLine("Contenido del objeto Queue")
    RecorrerEnumerador(oEnumerador)

    Console.ReadLine()
End Sub

Public Sub RecorrerEnumerador(ByVal oEnumerador As IEnumerator)
    While oEnumerador.MoveNext
        Console.WriteLine(oEnumerador.Current)
    End While
    Console.WriteLine()
End Sub
```

Código fuente 381

Normalmente, en un sistema de gestión de mensajes, una vez que solicitamos un mensaje, este es enviado y borrado de su contenedor. Pues algo similar es lo que haremos seguidamente, ya que después de que el usuario haya introducido todos los mensajes, utilizaremos el método Dequeue( ), que irá extrayendo o *desencolando* cada uno de los valores de la lista, comenzando por el primero que fue introducido. Este método, además de devolver el valor, lo elimina de la lista. Ver el Código fuente 382.

```
Sub Main()
    Dim aqListaMensa As New Queue()
    Dim sMensaje As String
    Dim iContador As Integer

    ' bucle de recepción de mensajes
    Do
        iContador += 1
        Console.WriteLine("Mensaje nro. {0}. " & _
            "Pulse [INTRO] para finalizar captura", _
            iContador)
        sMensaje = Console.ReadLine()
        ' si hemos escrito algo...
        If sMensaje.Length > 0 Then
            ' añadimos a la lista
            aqListaMensa.Enqueue(sMensaje)
        Else
            ' salimos
            Exit Do
        End If
    Loop
    Console.WriteLine()

    ' la propiedad Count nos indica la cantidad de
    ' elementos en la lista
    Console.WriteLine("Hay {0} mensajes para procesar", aqListaMensa.Count)
    Console.WriteLine()

    ' procesar los mensajes de la lista
    iContador = 0
    Console.WriteLine("Procesar lista de mensajes")
    While aqListaMensa.Count > 0
        iContador += 1
        Console.WriteLine("Mensaje nro. {0} - texto: {1}", _
            iContador, aqListaMensa.Dequeue())
        Console.WriteLine("Quedan {0} mensajes por procesar", aqListaMensa.Count)
        Console.WriteLine()
    End While

    Console.ReadLine()
End Sub
```

Código fuente 382

Pero supongamos que queremos comprobar el contenido del próximo mensaje a procesar antes de sacarlo de la lista. En ese caso, el método Peek( ) de la clase Queue es el indicado, ya que precisamente devuelve el siguiente valor de la lista sin eliminarlo, como ilustra el Código fuente 383.

```
Console.WriteLine("Obtenemos el primer valor sin eliminar: {0}",
aqListaMensa.Peek())
```

Código fuente 383

Puede ocurrir también, que necesitemos obtener los valores de la lista pero sin eliminarlos, por ejemplo, para poder repetir el proceso posteriormente. ¿Tenemos que volver a pedirlos al usuario?, no en absoluto, ya que mediante el método Clone( ) de la clase Queue, podemos obtener un nuevo objeto independiente, pero con los mismo elementos. A partir de ese momento, aunque procesemos la lista original, mantendremos los datos en una copia del objeto.

Otra técnica consiste en utilizar el método ToArray( ), que copia los valores de la lista a un array estándar. Dicho array deberá haber sido previamente creado con el tamaño adecuado, ya que si tiene menos elementos que el objeto Queue, cuando copiemos los valores al array se producirá un error. Para saber el tamaño del array que tenemos que crear, podemos ayudarnos de la propiedad Count del objeto Queue. El Código fuente 384 muestra unos ejemplos.

```
Public Sub Main()
    ' crear objeto Queue e introducir valores
    Dim aqListaMensa As New Queue()
    '....
    '....
    ' crear copia de seguridad de la lista
    Dim aqCopiaMensajes As Queue = aqListaMensa.Clone()

    ' o también, podemos crear un array normal con los
    ' elementos del objeto Queue
    Dim aListaDatos As Array = Array.CreateInstance(GetType(Object), _
        aqListaMensa.Count)
    aListaDatos = aqListaMensa.ToArray()

    ' procesar los mensajes de la lista original,
    ' mantendremos los valores en el objeto clonado
    ' o en un array
    '....
    '....
End Sub
```

Código fuente 384

Copiando los valores a un array, tenemos la ventaja de poder ordenarlos, o hacer cualquier operación que nos permita la clase Array.

Y ya para terminar con esta clase, el método Clear( ) elimina todos los valores del array interno que mantiene un objeto Queue. Podemos utilizar este método en el caso de que después de haber obtenido algunos valores de la lista, no queramos seguir extrayendo información, pero necesitemos el objeto vacío para poder repetir el proceso de captura de datos.

La clase Stack

Al igual que en el caso anterior, esta clase nos permite gestionar colas de valores, pero los objetos de este tipo, crean listas de valores al estilo de pilas LIFO (last in, first out), último en entrar/primero en salir.

Su modo de manejo es similar a las colecciones Queue, con la característica de que en una pila, los valores que extraemos son los últimos que han entrado. Veamos un ejemplo en el Código fuente 385.

```
Public Sub Main()
    ' creamos una colección de tipo pila
    Dim oPila As Stack
    oPila = New Stack()

    ' para añadir valores, usamos el método Push()
    oPila.Push("A") ' este será el último en salir
    oPila.Push("B")
```

```
    oPila.Push("C")
    oPila.Push("D")
    oPila.Push("E") ' este será el primero en salir

    ' para extraer un valor de la pila, usamos el método Pop(),
    ' dicho valor se eliminará de la lista
    While oPila.Count > 0
        Console.WriteLine("El valor obtenido de la lista es: {0}", oPila.Pop)
    End While

    Console.ReadLine()
End Sub
```

Código fuente 385

25

Colecciones personalizadas

Cuando el tipo de array que necesitamos no existe

Los arrays básicos proporcionados por el entorno de .NET, a través de la clase Array, proveen al programador, a través del conjunto de miembros que implementan, de un excelente conjunto de funcionalidades para realizar la manipulación de listas de valores.

Por si ello no fuera suficiente, disponemos además del espacio de nombres System.Collection, que aportan un conjunto de clases, que nos facilitan objetos para manejar arrays con características avanzadas.

Pese a todo, habrá ocasiones en que necesitemos trabajar con un array de un modo especial, y no dispongamos entre los miembros de la clase Array, ni entre las clases de Collection, de las funcionalidades requeridas. Sin embargo, gracias a la arquitectura orientada a objeto de la plataforma, podemos crear nuestras propias clases para la manipulación de arrays, heredando de las colecciones existentes o implementando los interfaces necesarios.

Utilizando la herencia para crear una nueva colección

Situémonos en primer lugar en un escenario de ejemplo: estamos desarrollando una aplicación en la que utilizamos colecciones de tipo ArrayList. Sin embargo, necesitamos que dichas colecciones admitan sólo números, y que además, el array de la colección esté ordenado.

Dado que estas características no están implementadas en los objetos ArrayList, pero este tipo de colección se adapta en gran medida a nuestras necesidades, crearemos una nueva clase con el nombre

NumerosOrden, que herede de ArrayList, y en la que codificaremos el comportamiento personalizado que necesitamos. Ver el Código fuente 386.

```
' creamos una clase que herede de ArrayList,
' que ordene los elementos del array,
' y que sólo admita números
Class NumerosOrden
    Inherits ArrayList

    ' sobrescribimos el método Add() de la clase base;
    ' sólo vamos a permitir añadir números a la colección
    Public Overrides Function Add(ByVal value As Object) As Integer
        ' si es un número, añadimos
        If IsNumeric(value) Then
            MyBase.Add(value)   ' llamamos al método base
            Me.Sort()           ' ordenamos el array
            Return Me.Count     ' devolvermos el número de elementos que hay
        Else
            ' si no es un número...
            Return -1
        End If
    End Function

End Class
```

Código fuente 386.

Para usar esta clase desde el código cliente, instanciaremos un objeto y trabajaremos con él como si se tratara de una colección normal. Ver el Código fuente 387.

```
Sub Main()
    ' instanciar un objeto de nuestra
    ' colección personalizada que ordena números
    Dim oNumOr As New NumerosOrden()
    oNumOr.Add(980)
    oNumOr.Add(500)
    oNumOr.Add(25)
    oNumOr.Add(700)

    ' recorrer la colección
    Dim oEnumera As IEnumerator = oNumOr.GetEnumerator()
    Console.WriteLine("Colección ordenada de números")
    While oEnumera.MoveNext
        Console.WriteLine("valor: {0}", oEnumera.Current)
    End While

    Console.ReadLine()
End Sub
```

Código fuente 387

En este ejemplo, el lector ha podido comprobar cómo con una mínima cantidad de código en una clase derivada, conseguimos toda la potencia de una colección base, y le proporcionamos un comportamiento del que originalmente carecía.

Implementando un interfaz para crear una nueva colección

Como acabamos de comprobar en el ejemplo anterior, la creación de nuevas clases de tipo collection mediante herencia es una estupenda solución. Debido a que el código principal ya existe en la clase base, simplemente tenemos que añadir el código para el nuevo comportamiento en la clase derivada, mediante la creación de nuevos miembros, a través de sobrecarga, o sobre-escritura. Sin embargo, esta técnica tiene el inconveniente de que no podemos modificar o eliminar ciertos comportamientos de la clase base.

Para salvar este obstáculo, en lugar de heredar de una colección, podemos emplear otra técnica, que consiste en crear una clase, e implementar en ella uno o varios interfaces de los incluidos en el espacio de nombres System.Collections, los cuales definen las características que deben tener los arrays y colecciones del entorno.

La ventaja de implementar interfaces de colección en una clase, reside en que tenemos control absoluto en cuanto al comportamiento que tendrán los objetos collection que instanciemos de dicha clase. Aunque como inconveniente, tenemos el hecho de que la implementación de un interfaz, obliga a codificar todos los miembros que vienen en la declaración de dicho interfaz; respecto a esto último, podemos escribir sólo la declaración de algunos miembros y no su código; de esta manera, para aquellos aspectos del interfaz en los que no estemos interesados, no será necesario codificar.

Supongamos como ejemplo, que necesitamos en nuestro programa una colección que convierta a mayúsculas los valores que añadimos o modificamos. Para ello, vamos a crear una clase con el nombre ListaMay, que implemente el interfaz IList. Este interfaz está compuesto por un numeroso conjunto de miembros, pero nosotros sólo vamos a codificar los siguientes.

- **Add( )**. Añadir un nuevo elemento.
- **Item( )**. Asigna u obtiene un valor.
- **Contains( )**. Comprueba si existe un valor entre los elementos de la colección, devolviendo un valor lógico Verdadero si existe, o Falso en el caso de que no exista.
- **Count**. Devuelve el número de elementos que hay en la colección.
- **GetEnumerator( )**. Devuelve un enumerador para recorrer el array de la colección.
- **Clear( )**. Elimina los valores de los elementos del array que tiene la colección.

Para el resto de miembros del interfaz, aunque no vayamos a utilizarlos, debemos crear su declaración vacía, ya que en caso contrario, se producirá un error al intentar ejecutar el programa, porque el entorno detectará la falta de la creación de los miembros de dicho interfaz en nuestra clase.

No codificamos todos los miembros del interfaz para simplificar el presente ejemplo, y al mismo tiempo, demostramos que sólo hemos de escribir código para aquellos aspectos del interfaz que necesitemos, aunque lo recomendable sería, evidentemente, implementar correctamente todos los métodos y propiedades que indica el interfaz. Veámoslo en el Código fuente 388.

```
' esta clase implementa la funcionalidad de una colección;
' admite como elementos cadenas de caracteres,
' y los convierte a mayúsculas
```

```
Class ListaMay
    ' para conseguir el comportamiento de una colección
    ' en esta clase implementamos el siguiente interfaz
    Implements IList

    ' declaramos un array que contendrá
    ' los valores de la colección
    Protected aValores() As String

    ' a partir de aquí, debemos declarar todos los miembros
    ' del interfaz, y codificar aquellos que necesitemos
    ' -----------------------------------------------------
    Public ReadOnly Property Count() As Integer _
        Implements System.Collections.IList.Count

        Get
            If Not (aValores Is Nothing) Then
                Count = aValores.Length
            End If
        End Get

    End Property

    Public ReadOnly Property IsSynchronized() As Boolean _
        Implements System.Collections.IList.IsSynchronized

        Get
        End Get

    End Property

    Public ReadOnly Property SyncRoot() As Object _
        Implements System.Collections.IList.SyncRoot

        Get
        End Get

    End Property

    Public Function GetEnumerator() As System.Collections.IEnumerator _
        Implements System.Collections.IList.GetEnumerator

        Return aValores.GetEnumerator()

    End Function

    Public Function Add(ByVal value As Object) As Integer _
        Implements System.Collections.IList.Add

        If aValores Is Nothing Then
            ReDim Preserve aValores(0)
            aValores(0) = CStr(value).ToUpper()
        Else
            Dim iIndiceMayor As Integer = aValores.GetUpperBound(0)
            ReDim Preserve aValores(iIndiceMayor + 1)
            aValores(iIndiceMayor + 1) = CStr(value).ToUpper()
        End If

        Return Me.Count
    End Function

    Public Sub Clear() _
        Implements System.Collections.IList.Clear

        Array.Clear(aValores, 0, aValores.Length)

    End Sub
```

```
Public Function Contains(ByVal value As Object) As Boolean _
    Implements System.Collections.IList.Contains

    Dim oEnumera As IEnumerator
    oEnumera = aValores.GetEnumerator()
    While oEnumera.MoveNext
        If (oEnumera.Current = CType(value, String).ToUpper()) Then
            Return True
        End If
    End While

    Return False

End Function

Public Function IndexOf(ByVal value As Object) As Integer _
    Implements System.Collections.IList.IndexOf

End Function

Public ReadOnly Property IsFixedSize() As Boolean _
    Implements System.Collections.IList.IsFixedSize

    Get
    End Get

End Property

Public ReadOnly Property IsReadOnly() As Boolean _
    Implements System.Collections.IList.IsReadOnly

    Get
    End Get

End Property

Default Public Property Item(ByVal index As Integer) As Object _
    Implements System.Collections.IList.Item

    Get
        Item = aValores.GetValue(index)
    End Get
    Set(ByVal Value As Object)
        Dim oTipo As Type
        oTipo = Value.GetType()

        If oTipo.Name = "String" Then
            aValores.SetValue(CType(Value, String).ToUpper(), index)
        Else
            aValores.SetValue(Value, index)
        End If
    End Set

End Property

Public Sub Remove(ByVal value As Object) _
    Implements System.Collections.IList.Remove

End Sub

Public Sub RemoveAt(ByVal index As Integer) _
    Implements System.Collections.IList.RemoveAt

End Sub

Public Sub Insert(ByVal index As Integer, ByVal value As Object) _
```

```
        Implements System.Collections.IList.Insert

    End Sub

    Public Sub CopyTo(ByVal array As System.Array, ByVal index As Integer) _
        Implements System.Collections.IList.CopyTo

    End Sub

End Class
```

Código fuente 388

Desde código cliente, el modo de utilizar nuestra nueva clase es igual que para una de las colecciones normales, aunque debemos tener en cuenta los métodos no codificados, para no utilizarlos. Veamos el Código fuente 389.

```
Public Sub Main()
    ' instanciar un objeto de la colección personalizada
    Dim oLis As ListaMay = New ListaMay()

    Console.WriteLine("Número de elementos {0}", oLis.Count)
    Console.WriteLine()

    ' añadir valores
    oLis.Add("Esta es")
    oLis.Add("mi clase")
    oLis.Add("de arrays personalizados")
    RecorrerMiLista(oLis)

    Console.WriteLine("Número de elementos {0}", oLis.Count)
    Console.WriteLine()

    ' comprobar si existe un valor
    Dim sValorBuscar As String
    Console.WriteLine("Introducir un valor a buscar en la colección")
    sValorBuscar = Console.ReadLine()
    If oLis.Contains(sValorBuscar) Then
        Console.WriteLine("El valor introducido sí existe")
    Else
        Console.WriteLine("El valor introducido no existe")
    End If
    Console.WriteLine()

    ' eliminar valores de la colección (no elimina elementos)
    oLis.Clear()

    ' asignar valores a los índices existentes de la colección
    oLis(2) = "mesa"
    oLis.Item(0) = "coche"

    RecorrerMiLista(oLis)

    Console.ReadLine()
End Sub

Public Sub RecorrerMiLista(ByVal oListaValores As ListaMay)
    Dim oEnumera As IEnumerator
    oEnumera = oListaValores.GetEnumerator()
    While oEnumera.MoveNext
        Console.WriteLine("valor {0}", oEnumera.Current)
    End While
```

```
    Console.WriteLine()
End Sub
```

Código fuente 389.

26

Manipulación de errores

Errores, ese mal común

Difícil es, por no decir imposible, encontrar al programador que no tenga errores en su código.

Por mucho cuidado que pongamos al codificar nuestras aplicaciones, los errores de ejecución serán ese incómodo, pero inevitable compañero de viaje que seguirá a nuestros programas allá donde estos vayan.

En primer lugar, antes de abordar el tratamiento de errores en nuestras aplicaciones, y los elementos que nos proporciona el entorno para manipularlos, podemos clasificar los tipos de errores en una serie de categorías genéricas.

Errores de escritura

Son los de localización más inmediata, ya que se producen por un error sintáctico al escribir nuestro código, y gracias al IDE de Visual Studio .NET, podemos detectarlos rápidamente.

Cuando escribimos una sentencia incorrectamente, dejamos algún paréntesis sin cerrar, etc., el IDE subraya la parte de código errónea, y nos muestra un mensaje informativo del error al situar el cursor del ratón sobre el mismo. En el ejemplo de la Figura 215, hemos declarado una estructura While que no hemos cerrado con la correspondiente instrucción End While; por lo tanto, el IDE nos lo indica.

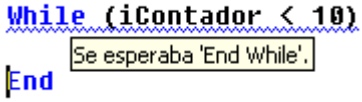


Figura 215. Error de escritura de código.

Errores de ejecución

Este tipo de errores son los que provocan un fallo en la ejecución del programa y su interrupción. No obstante, si utilizamos los gestores de error que proporciona la herramienta de desarrollo correspondiente, podremos en algunos casos, evitar la cancelación de la ejecución, recuperando su control. El ejemplo del Código fuente 390 provoca un error, ya que se intenta asignar un valor que no corresponde al tipo de dato de una variable.

```
Dim dtFecha As Date
dtFecha = "prueba"
```

Código fuente 390

Los errores de ejecución son el objetivo del presente tema; sobre su captura y manipulación nos centraremos a lo largo de los próximos apartados.

Errores lógicos

Estos errores son los de más difícil captura, ya que el código se encuentra correctamente escrito, produciéndose el problema por un fallo de planteamiento en el código, motivo por el cual, por ejemplo, el control del programa no entra en un bucle porque una variable no ha tomado determinado valor; el flujo del programa sale antes de lo previsto de un procedimiento, al evaluar una expresión que esperábamos que tuviera un resultado diferente, etc.

Errores y excepciones

Dentro del esquema de gestión de errores del entorno .NET Framework, encontramos las figuras del error y la excepción. Estos elementos son utilizados indistintamente en muchas ocasiones para hacer referencia genérica a los errores producidos; sin embargo, aunque complementarios, cada uno tiene su propia funcionalidad dentro del proceso de tratamiento de un error.

- **Error**. Un error es un evento que se produce durante el funcionamiento de un programa, provocando una interrupción en su flujo de ejecución. Al producirse esta situación, el error genera un objeto excepción.

- **Excepción**. Una excepción es un objeto generado por un error, que contiene información sobre las características del error que se ha producido.

Manipuladores de excepciones

Un manipulador de excepción es un bloque de código que proporciona una respuesta al error que se ha producido, y que se incluye en una estructura proporcionada por el lenguaje a tal efecto, es decir, para la captura de excepciones.

Tipos de tratamiento de error en VB.NET

VB.NET proporciona dos tipos de tratamiento de error: estructurado y no estructurado.

El primero se basa en los esquemas de captura de errores de lenguajes como C# y C++; gestionando los errores a través de excepciones, y una estructura de control que se encarga de atrapar aquellas excepciones que se produzcan.

El segundo es un sistema heredado de versiones anteriores de VB, y está basado en la detección y captura de errores a través de etiquetas de código, mediante saltos no estructurados en el flujo de la ejecución.

Manipulación estructurada de errores

En este tipo de tratamiento, cada vez que se produce un error, se genera un objeto de la clase Exception o alguna de sus derivadas, conteniendo la información del error ocurrido. La manera de capturar este tipo de objetos pasa por utilizar una estructura de control del lenguaje, proporcionada para esta finalidad.

La estructura Try...End Try

Esta estructura de control del lenguaje, proporciona el medio para definir un bloque de código sensible a errores, y los correspondientes manipuladores de excepciones, en función del tipo de error producido. El Código fuente 391 muestra su sintaxis.

```
Try
       ' código que puede provocar errores
       ' ....
       ' ....

[Catch [Excepcion [As TipoExcepcionA]] [When Expresión]
       ' respuesta a error de tipo A
       ' ....
       ' ....
       [Exit Try]
]

[Catch [Excepcion [As TipoExcepcionN]] [When Expresión]
       ' respuesta a error de tipo N
       ' ....
       ' ....
       [Exit Try]
]

[Finally
       ' código posterior al control de errores
```

```
        ' ....
        ' ....
]

End Try
```

Código fuente 391

Analicemos con detalle los principales elementos de esta estructura.

En primer lugar nos encontramos con su declaración mediante la palabra clave Try. Todo el código que escribimos a partir de dicha palabra clave, y hasta la primera sentencia Catch, es el código que definimos como sensible a errores, o dicho de otro modo, el bloque de instrucciones sobre las que deseamos que se active el control de errores cuando se produzca algún fallo en su ejecución.

A continuación, establecemos cada uno de los manipuladores de excepción mediante la palabra clave Catch. Junto a esta palabra clave, situaremos de forma opcional, un identificador que contendrá el objeto con la excepción generada. Finalmente, y también de modo opcional, con la palabra clave When, especificaremos una condición para la captura del objeto de excepción. Podemos escribir uno o varios manipuladores Catch dentro de una estructura de control Try...End Try.

Cada vez que se produzca un error, el flujo de la ejecución saltará a la sentencia Catch más acorde con el tipo de excepción generada por el error, siempre y cuando hayamos situado varios manipuladores de excepciones en el controlador de errores.

Tal y como acaba de ver el lector en la sintaxis de la estructura Try...End Try, es posible utilizar Catch de un modo genérico, es decir, sin establecer qué tipo de excepción se ha producido. Este es el tipo de control de errores más sencillo que podemos implementar, aunque también el más limitado, ya que sólo podemos tener un manipulador de excepciones. Veamos un ejemplo en el Código fuente 392.

```
Public Sub Main()
    Dim sValor As String
    Dim iNumero As Integer

    Try
        ' comienza el control de errores
        Console.WriteLine("Introducir un número")
        sValor = Console.ReadLine()
        ' si no hemos introducido un número...
        iNumero = sValor    ' ...aquí se producirá un error...

        ' ...y no llegaremos a esta parte del código
        iNumero = iNumero + 1000

    Catch
        ' si se produce un error, se genera una excepción
        ' que capturamos en este bloque de código
        ' manipulador de excepción, definido por Catch
        Console.WriteLine("Error al introducir el número" & _
            ControlChars.CrLf & _
            "El valor {0} es incorrecto", _
            sValor)

    End Try

    ' resto del código del procedimiento
    ' ....
    Console.ReadLine()
```

```
End Sub
```

Código fuente 392

Tanto si se produce un error como si no, la sentencia Finally de la estructura Try...End Try, nos permite escribir un bloque de código que será ejecutado al darse una condición de error, o bajo ejecución normal del procedimiento.

El Código fuente 393 muestra el mismo ejemplo anterior, pero introduciendo un bloque Finally. Pruebe el lector alternativamente, a forzar un error, y a ejecutar sin errores este fuente; en ambos casos verá que el bloque Finally es ejecutado. Para completar el ejemplo, tras la estructura Try...End Try se han escrito varias líneas de código potencialmente problemáticas; en el caso de que se produzca un error, la ejecución será cancelada, al no estar dichas líneas situadas en un controlador de errores.

```
Public Sub Main()
    Dim sValor As String
    Dim iNumero As Integer

    Try
        ' comienza el control de errores
        Console.WriteLine("Introducir un número")
        sValor = Console.ReadLine()
        ' si no hemos introducido un número...
        iNumero = sValor    ' ...aquí se producirá un error...

        ' ...y no llegaremos a esta parte del código
        iNumero = iNumero + 1000

    Catch
        ' si se produce un error, se genera una excepción
        ' que capturamos en este bloque de código
        ' manipulador de excepción, definido por Catch
        Console.WriteLine("Error al introducir el número" & _
            ControlChars.CrLf & _
            "El valor {0} es incorrecto", _
            sValor)

    Finally
        ' si se produce un error, después de Catch se ejecuta este bloque;
        ' si no se produce error, después de Try también se ejecuta
        Console.WriteLine("El controlador de errores ha finalizado")
    End Try

    ' resto del código del procedimiento
    Dim dtFecha As Date
    Console.WriteLine("Introducir una fecha")
    ' si ahora se produce un error,
    ' al no disponer de una estructura para controlarlo
    ' se cancelará la ejecución
    dtFecha = Console.ReadLine()
    Console.WriteLine("La fecha es {0}", dtFecha)
    Console.ReadLine()
End Sub
```

Código fuente 393

La clase Exception

Como hemos explicado en anteriores apartados, cada vez que se produce un error, el entorno de ejecución genera una excepción con la información del error acaecido.

Para facilitarnos la tarea de manipulación de la excepción producida, en un controlador de excepciones obtenemos un objeto de la clase Exception, o de alguna de sus derivadas, de forma que, a través de sus miembros, podemos saber qué ha pasado. Entre las propiedades y métodos que podemos utilizar, se encuentran las siguientes.

- **Message**. Descripción del error.
- **Source**. Nombre del objeto o aplicación que provocó el error.
- **StackTrace**. Ruta o traza del código en la que se produjo el error.
- **ToString( )**. Devuelve una cadena con información detallada del error. En esta cadena podemos encontrar también, los valores obtenidos de las propiedades anteriores; por lo que el uso de este método, en muchas ocasiones será el modo más recomendable para obtener los datos de la excepción.

Podemos obtener el objeto de excepción creado a partir de un error, utilizando la sentencia Catch de la estructura Try. Para ello, a continuación de Catch, escribimos el nombre de un identificador, definiéndolo como tipo Exception o alguno de los tipos de su jerarquía.

El Código fuente 394 muestra la captura de la excepción en el ejemplo anterior, dentro de la sentencia Catch, pero en este caso utilizando un objeto Exception. El resto del código es igual que el anterior ejemplo.

```
' ....
Try
    ' ....
    ' ....
Catch oExcep As Exception
    ' si se produce un error, se crea un objeto excepción
    ' que capturamos volcándolo a un identificador
    ' de tipo Exception
    Console.WriteLine("Se produjo un error. Información de la excepción")
    Console.WriteLine("==============================================")
    Console.WriteLine("Message: {0}", oExcep.Message)
    Console.WriteLine()
    Console.WriteLine("Source: {0}", oExcep.Source)
    Console.WriteLine()
    Console.WriteLine("StackTrace: {0}", oExcep.StackTrace)
    Console.WriteLine()
    Console.WriteLine(oExcep.ToString())

Finally
    ' ....
    ' ....
End Try
' ....
```

Código fuente 394

El Código fuente 395 contiene una pequeña muestra de los valores obtenidos a partir de las propiedades Message, Source y StackTrace, tras la ejecución del fuente anterior.

```
Message: Cast from String ('hola') to Integer is not valid.

Source: Microsoft.VisualBasic

StackTrace:    at Microsoft.VisualBasic.Helpers.IntegerType.FromString(String
Value)
   at ErroresPru.Module1.Main() in
K:\CursoVBNET\Texto\t16Errores\ErroresPru\Module1.vb:line 12
```

Código fuente 395

Exception representa la clase base en la jerarquía de tipos de excepción que se pueden producir dentro del entorno de ejecución.

Partiendo de Exception, disponemos de un conjunto de clases derivadas, que nos permiten un tratamiento más particular del error producido, como ApplicationException, IOException, SystemException, etc. Cada una de ellas puede tener, además de los miembros generales de Exception, una serie de métodos y propiedades particulares de su tipo de excepción; por ello, lo más conveniente, será utilizar estas clases, a través de las que podremos averiguar más detalles sobre el problema producido.

Captura de excepciones de diferente tipo en el mismo controlador de errores

Cuando en el código de un controlador de errores puedan producirse errores de distintos tipos de excepción, debemos situar tantas sentencias Catch como excepciones queramos controlar.

En el Código fuente 396, hasta el momento, hemos controlado los errores por conversión de tipos. Ahora vamos a añadir varias líneas más, que obtienen un valor, y lo asignan a un índice de un array. Dado que el índice a manipular lo pedimos al usuario, y es posible que dicho elemento no exista en el array, añadiremos un nuevo manipulador para este tipo de excepción, mediante la sentencia Catch correspondiente.

```
Public Sub Main()
    Dim sValor As String
    Dim iNumero As Integer
    Dim sLetras() As String = {"a", "b", "c", "d"}

    Try
        ' comienza el control de errores
        Console.WriteLine("Introducir un número")
        sValor = Console.ReadLine()
        ' si no hemos introducido un número...
        iNumero = sValor    ' ...aquí se producirá un error...

        ' ...y no llegaremos a esta parte del código
        iNumero = iNumero + 1000

        ' introducir una letra y asignarla a una
        ' posición del array
```

```
        Dim sNuevaLetra As String
        Dim iPosicion As Integer
        Console.WriteLine("Introducir una letra")
        sNuevaLetra = Console.ReadLine()
        Console.WriteLine("Introducir posición del array para la letra")
        iPosicion = Console.ReadLine()
        ' si al asignar la letra al array no existe
        ' el índice, se producirá un error
        sLetras(iPosicion) = sNuevaLetra

    Catch oExcep As System.InvalidCastException
        ' excepción producida por un error al intentar
        '  realizar una conversión de tipos
        Console.WriteLine(oExcep.ToString())

    Catch oExcep As System.IndexOutOfRangeException
        ' excepción producida por un error
        ' al intentar usar un índice inexistente
        ' de array, o índice fuera de rango
        Console.WriteLine(oExcep.ToString())

    Finally
        ' si se produce un error, después de Catch se ejecuta este bloque;
        ' si no se produce error, después de Try también se ejecuta
        Console.WriteLine("El controlador de errores ha finalizado")
    End Try

    Console.ReadLine()
End Sub
```

Código fuente 396

Establecer una condición para un manipulador de excepciones

Mediante la cláusula When, de la sentencia Catch, podemos situar una expresión que sirva como filtro o condición, para que dicho manipulador de excepciones se ejecute, en función de que el resultado de la expresión devuelva Verdadero o Falso.

En el siguiente ejemplo, definimos un manipulador de excepciones, para cuando se produzcan errores de desbordamiento al asignar un valor a una variable de tipo Byte. Sin embargo, mediante When, establecemos que dicho manipulador sólo se ejecute cuando sea un determinado mes; lo que provoca, que en el caso de que no se cumpla tal condición, saltará el mensaje de excepción predeterminado del IDE. Veamos el Código fuente 397.

```
Public Sub Main()
    Dim byMiNum As Byte
    Dim dtFHActual As Date

    ' obtener la fecha actual
    dtFHActual = System.DateTime.Today()

    Try
        ' comienza el control de errores
        Console.WriteLine("Introducir un número")
        ' si introducimos un número no incluido
        ' en el rango de Byte...
        byMiNum = Console.ReadLine()

    Catch oExcep As OverflowException When (dtFHActual.Month = 3)
        ' ...saltará este manipulador de excepciones, pero sólo
```

```
        ' cuando las excepciones de desbordamiento
        ' se produzcan en el mes de Marzo
        Console.WriteLine("El número introducido " & _
            "no se encuentra en el rango adecuado")

    Finally
        Console.WriteLine("El controlador de errores ha finalizado")
    End Try

    Console.ReadLine()
End Sub
```

Código fuente 397

Si queremos capturar también el resto de excepciones de desbordamiento, u otro tipo de excepciones, tenemos varias alternativas que describimos seguidamente.

- **Quitar la condición de filtro al manipulador de excepciones actual**. De este modo, capturaremos todas las excepciones de desbordamiento, pero no podremos filtrar por un mes determinado.

- **Añadir un nuevo manipulador a la estructura de control, para todas las excepciones de desbordamiento**. En esta situación, si se produce un error de desbordamiento, y no estamos en el mes definido por el anterior manipulador, se ejecutará este nuevo manipulador. Ver el Código fuente 398.

```
' ....
Catch oExcep As OverflowException When (dtFHActual.Month = 3)
    ' ...saltará este manipulador de excepciones, pero sólo
    ' cuando las excepciones de desbordamiento
    ' se produzcan en el mes de Marzo
    Console.WriteLine("El número introducido " & _
        "no se encuentra en el rango adecuado")

Catch oExcep As OverflowException
    Console.WriteLine("Error de desbordamiento")
' ....
```

Código fuente 398

- **Añadir un manipulador de excepciones genérico**. Con esto evitaremos el mensaje de error no controlado, generado por el IDE. Si por ejemplo, además de las operaciones con el tipo Byte, nos encontramos manipulando fechas, podremos capturar todas las excepciones producidas. Veamos este caso en el Código fuente 399.

```
Public Sub Main()
    Dim byMiNum As Byte
    Dim dtFecha As Date
    Dim dtFHActual As Date

    ' obtener la fecha actual
    dtFHActual = System.DateTime.Today()

    Try
        ' comienza el control de errores
```

```
        Console.WriteLine("Introducir un número")
        ' si introducimos un número no incluido
        ' en el rango de Byte, según el mes actual iremos
        ' a uno de los manipuladores de excepción existentes
        byMiNum = Console.ReadLine()

        ' si introducimos un valor incorrecto para la fecha,
        ' iremos al controlador de errores genérico
        Console.WriteLine("Introducir una fecha")
        dtFecha = Console.ReadLine()

    Catch oExcep As OverflowException When (dtFHActual.Month = 3)
        ' manipulador de excepciones sólo
        ' cuando las excepciones de desbordamiento
        ' se produzcan en el mes de Marzo
        Console.WriteLine("El número introducido " & _
            "no se encuentra en el rango adecuado")

    Catch oExcep As Exception
        ' manipulador genérico de excepciones
        Console.WriteLine("Se ha producido un error")

    Finally
        Console.WriteLine("El controlador de errores ha finalizado")
    End Try

    Console.ReadLine()
End Sub
```

Código fuente 399

El manipulador genérico de excepciones de este último ejemplo tiene un problema, ya que aunque las captura correctamente, no proporciona suficiente información, por lo que no podremos saber si el error se produjo por asignar un valor incorrecto a la variable Byte o a la fecha.

Este problema tiene una fácil solución: al ser una excepción un objeto, y por lo tanto, un tipo del sistema, mediante su método GetType( ) obtendremos el tipo de excepción producida, mostrándola en el mensaje del manipulador de excepciones. Ver el Código fuente 400.

```
' ....
' ....
Catch oExcep As Exception
    ' manipulador genérico de excepciones
    Dim oTipo As Type
    oTipo = oExcep.GetType()
    Console.WriteLine("Se ha producido un error de tipo {0}", oTipo.Name)
' ....
' ....
```

Código fuente 400

La influencia del orden de los manipuladores de excepciones

El orden en el que situemos las sentencias Catch dentro de una estructura Try...End Try, es determinante, a la hora de que ciertas excepciones puedan o no, ser capturadas. Por este motivo, al escribir un controlador de errores, se recomienda situar en primer lugar, los manipuladores más específicos, y dejar para el final, los más genéricos.

En el ejemplo que muestra el Código fuente 401 se pueden producir dos tipos de excepción: por desbordamiento, y por acceso a índice no existente en un array. El problema que tenemos en dicha construcción de código, reside en que el manipulador de excepciones de desbordamiento nunca se ejecutará, ya que en primer lugar hemos situado uno más genérico que captura todo tipo de excepciones, incluidas las que se produzcan por desbordamiento.

```
Public Sub Main()
    Dim byMiNum As Byte
    Dim aValores() As String = {"a", "b", "c"}

    Try
        ' comienza el control de errores
        Console.WriteLine("Introducir un número")
        ' si no es un número Byte se produce error
        byMiNum = Console.ReadLine()

        ' esta línea produce error siempre, ya
        ' que no existe el índice 5 en el array
        aValores(5) = "d"

    Catch oExcep As Exception
        ' manipulador genérico de excepciones,
        ' captura todo tipo de excepciones, por lo que si
        ' también se produce una excepción OverflowException,
        ' se ejecutará este manipulador, por estar situado primero
        Console.WriteLine("Se ha producido un error")

    Catch oExcep As OverflowException
        ' captura de errores de desbordamiento,
        ' este manipulador nunca se ejecutará, por estar
        ' uno más genérico antes
        Console.WriteLine("El número introducido " & _
            "no se encuentra en el rango adecuado")

    Finally
        Console.WriteLine("El controlador de errores ha finalizado")
    End Try

    Console.ReadLine()
End Sub
```

Código fuente 401

En este caso que acabamos de ver, situaremos en primer lugar el manejador de excepciones de desbordamiento, y por último, el genérico.

Forzar la salida de un controlador de errores mediante Exit Try

A través de esta sentencia de la estructura Try...End Try, obligamos al flujo del programa a salir de la estructura de control de errores, desde cualquier punto de la misma en que nos encontremos.

En el Código fuente 402, y retomando parte del código del anterior ejemplo, vemos como en el bloque de código del controlador de errores, forzamos la salida de la estructura sin haber finalizado de ejecutar todo el código propenso a errores.

```
' ....
Try
    ' comienza el control de errores
    Console.WriteLine("Introducir un número")
    ' si no es un número Byte se produce error
    byMiNum = Console.ReadLine()

    ' salimos de controlador de errores
    ' sin finalizarlo
    Exit Try

    ' esta línea produce error siempre, ya
    ' que no existe el índice 5 en el array
    aValores(5) = "d"

Catch oExcep As OverflowException
    ' ....

Catch oExcep As Exception
    ' ....
    ' ....
End Try
' ....
```

Código fuente 402

Creación de excepciones personalizadas

En algunos escenarios de trabajo, puede suceder que la información proporcionada por alguno de los objetos Exception no se ajuste a lo que realmente necesitamos. Es por tanto, hora de crear una clase, que heredando de la clase base Exception, contenga la información a la medida en que la precisamos.

Supongamos, tal como vemos en el Código fuente 403, que escribimos la clase CtaCredito, para llevar el control de cuentas bancarias. En ella introduciremos el titular de la cuenta y un importe para el crédito que necesitemos asignar. No obstante, dicho crédito no podrá sobrepasar el valor de 2500; así que, para controlar tal circunstancia, creamos adicionalmente, la clase CreditoException, que heredando de Exception, contendrá información en uno de sus miembros, sobre la excepción producida en cuanto al importe que se intentó asignar a la cuenta.

```
' clase que contiene información sobre una cuenta bancaria
Public Class CtaCredito
    Private msTitular As String
    Private mdbDisponible As Double

    Public Property Titular() As String
        Get
            Return msTitular
        End Get
        Set(ByVal Value As String)
            msTitular = Value
        End Set
    End Property

    Public ReadOnly Property Credito() As Double
        Get
            Return mdbDisponible
        End Get
    End Property
```

```
    ' en este método, si se intenta asignar un importe
    ' superior al permitido, se lanza una excepción,
    ' utilizando un objeto de la clase CreditoException,
    ' heredado de Exception
    Public Sub AsignarCredito(ByVal ldbCredito As Double)
        If ldbCredito > 2500 Then
            Throw New CreditoException("Límite disponible: 2500 " & _
                "- Se intentó asignar " & CType(ldbCredito, String))
        Else
            mdbDisponible = ldbCredito
        End If
    End Sub
End Class

' ------------------------------------------------
' esta clase contiene la información sobre un error
' producido en un objeto CtaCredito
Public Class CreditoException
    Inherits Exception

    Private msDescripcion As String

    Public Sub New(ByVal lsDescripcion As String)
        msDescripcion = lsDescripcion
    End Sub

    Public ReadOnly Property Descripcion() As String
        Get
            Return msDescripcion
        End Get
    End Property
End Class
```

Código fuente 403

El esquema del proceso es el siguiente: cuando al método AsignarCredito( ), de un objeto CtaCredito, se intente asignar un valor no permitido, se genera un nuevo objeto CreditoException y se lanza a través de la palabra clave Throw, que es la encargada de emitir las excepciones en el entorno de ejecución.

Desde código cliente, el uso de estas clases sería el que muestra el Código fuente 404.

```
Module Module1
    Public Sub Main()
        ' crear un objeto de la nueva clase
        Dim oCredito As New CtaCredito()

        Try
            ' asignar valores a propiedades
            oCredito.Titular = "Jaime Peral"
            oCredito.AsignarCredito(1000)   ' esto no produce error
            Console.WriteLine("El credito actual de {0} es {1:C}", _
                oCredito.Titular, oCredito.Credito)

            ' esta instrucción sí produce error
            oCredito.AsignarCredito(5000)

        Catch oExcep As CreditoException
            ' manipulador para las excepciones producidas
            ' sobre un objeto CtaCredito
            Console.WriteLine("Error al asignar crédito: {0}", oExcep.Descripcion)
```

```
        Finally
            Console.WriteLine("El controlador de errores ha finalizado")
        End Try

        Console.ReadLine()
    End Sub
End Module
```

Código fuente 404

Manipulación no estructurada de errores

En este tipo de gestión de errores, cada vez que se produce un error, consultaremos el objeto del sistema Err. Este objeto contiene, a través de sus propiedades, la información sobre el error producido.

Para poder capturar los errores mediante este sistema, utilizaremos la instrucción On Error, que nos permitirá seleccionar el controlador de errores a ejecutar.

El objeto Err

Este objeto se crea automáticamente al iniciarse la aplicación, y proporciona al usuario información sobre los errores producidos en el transcurso de la aplicación. Tiene ámbito público, por lo que podremos usarlo desde cualquier punto del programa.

Cuando se produzca un error, la propiedad Number de este objeto tendrá un valor mayor de cero, mientras que la propiedad Description, nos dará una información textual del error producido.

On Error

Esta instrucción activa o desactiva una rutina de manejo de errores. Tiene diversos modos de empleo, que describimos en los siguientes apartados.

On Error Goto Etiqueta

Activa una etiqueta de control de errores, que consiste en un bloque de código, al que se desviará el flujo del programa cuando se produzca un error. El Código fuente 405 muestra un ejemplo.

```
Public Sub Main()
    On Error Goto ControlErrores

    Dim dtFecha As Date
    dtFecha = "valor incorrecto"


    Exit Sub


    ' ------------
    ' etiqueta de control de errores
ControlErrores:
    Console.WriteLine("Error: {0} - {1}", Err.Number, Err.Description)
```

```
    Console.ReadLine()
End Sub
```

Código fuente 405

Si queremos reintentar la ejecución de la línea que produjo el error, debemos utilizar en la etiqueta de control de errores la instrucción Resume, como muestra el Código fuente 406.

```
Public Sub Main()
    On Error Goto ControlErrores

    Dim dtFecha As Date
    Console.WriteLine("Introducir una fecha")
    dtFecha = Console.ReadLine()

    Console.WriteLine("Esta línea se ejecuta después del error")
    Console.ReadLine()

    Exit Sub

    ' ------------
    ' etiqueta de control de errores
ControlErrores:
    Console.WriteLine("Error: {0} - {1}", Err.Number, Err.Description)
    Console.ReadLine()
    Resume
End Sub
```

Código fuente 406

De esta forma, en el ejemplo anterior, damos una nueva oportunidad al usuario, en el caso de que haya introducido una fecha incorrecta.

Si no queremos volver a reintentar la línea del error, usaremos la instrucción Resume Next, que después de ejecutar la etiqueta de control de errores, seguirá la ejecución en la siguiente línea a la que provocó el error. También podemos utilizar el formato Resume Etiqueta, en donde Etiqueta representa a otra etiqueta de control, a la que saltará el código después de finalizar la ejecución de la actual.

On Error Resume Next

Esta variante de la instrucción On Error, hace que al producirse un error, la ejecución continúe con la siguiente línea de código, por lo que no utiliza etiquetas de control para desviar la ejecución.

Debido a sus características, en este tipo de captura de errores, tras cada línea susceptible de provocar un error, debemos consultar los valores del objeto Err, para comprobar si existe un error, y actuar en consecuencia.

En este tipo de situaciones, después de comprobar un error, debemos inicializar el objeto Err, llamando a su método Clear( ).

Veamos pues, un ejemplo de este tipo de gestión de errores en el Código fuente 407.

```
Public Sub Main()
    On Error Resume Next

    Dim dtFecha As Date
    Console.WriteLine("Introducir una fecha")
    dtFecha = Console.ReadLine()

    If Err.Number > 0 Then
        Console.WriteLine("Error: {0} - {1}", Err.Number, Err.Description)
        Console.ReadLine()
        Err.Clear()
    End If

    Console.WriteLine("Esta línea se ejecuta después de un posible error")
    Console.ReadLine()
End Sub
```

Código fuente 407

Creación de errores con el objeto Err

El método Raise( ), del objeto Err, nos permite generar nuestros propios errores, que se comportarán igual que los errores del sistema. Veamos un ejemplo en el Código fuente 408.

```
Public Sub Main()
    On Error Goto ControlErrores

    Dim iValor As Integer
    Console.WriteLine("Introducir un número")
    iValor = Console.ReadLine()

    If iValor > 500 Then
        Err.Raise(5100, , "El número debe ser menor de 500")
    End If

    Console.WriteLine("Esta línea se ejecuta después de un posible error")
    Console.ReadLine()
    Exit Sub

    ' ------------
    ' etiqueta de control de errores
ControlErrores:
    Console.WriteLine("Error: {0} - {1}", Err.Number, Err.Description)
    Console.ReadLine()
    Resume Next
End Sub
```

Código fuente 408

On Error Goto 0

Este uso de la instrucción On Error, desactiva el manejador de errores que hubiera activado hasta el momento; de modo, que a no ser que activemos otro manejador, los errores que se produzcan a partir de esa línea, provocarán un error fatal, cancelando la ejecución del programa. Ver el Código fuente 409.

```
Public Sub Main()
    On Error Goto ControlErrores

    Dim iValor As Integer
    Console.WriteLine("Introducir un número")
    iValor = Console.ReadLine()

    On Error Goto 0
    Console.WriteLine("Introducir otro número")
    iValor = Console.ReadLine()

    Console.ReadLine()
    Exit Sub

    ' ------------
    ' etiqueta de control de errores
ControlErrores:
    Console.WriteLine("Error: {0} - {1}", Err.Number, Err.Description)
    Console.ReadLine()
    Resume Next
End Sub
```

Código fuente 409.

27

Operaciones de entrada y salida (I/O). Gestión del sistema de archivos

La remodelación del viejo esquema de entrada y salida

Desde las primeras versiones del lenguaje, el programador de Visual Basic ha dispuesto de un conjunto de instrucciones y funciones para el manejo de las operaciones de lectura/escritura con archivos, y la gestión de los mismos dentro del sistema operativo, en cuanto a su creación, borrado copia, etc., entre directorios y unidades.

Si bien este modo de trabajo ha sido válido durante mucho tiempo, la incorporación de las técnicas OOP a los lenguajes de programación, hacían necesario una renovación en este aspecto de VB.

Las instrucciones Open, Input, Write, Put, etc., a pesar de resolver su cometido, no proporcionan un entorno de trabajo cómodo, en un mundo en el que cada vez prima más el trabajo con objetos.

La jerarquía de objetos FileSystemObject, introducida recientemente, vino a paliar en parte esta carencia, aportando un conjunto de clases que ya nos permitían, desde un prisma orienta a objeto, trabajar con todos los aspectos del sistema de archivos, en cuanto a su lectura, escritura, manejo de directorios, unidades, etc. La evolución de este conjunto de objetos se halla en la plataforma .NET.

System.IO, el punto de partida

Con la llegada de la tecnología .NET, el acceso al sistema de archivos, es un aspecto que ya no forma parte de un lenguaje determinado, como ocurría en las anteriores versiones de VB, sino que ha sido

integrado dentro de la jerarquía de clases de la plataforma, en el espacio de nombres IO de System. Con ello, todos los lenguajes compatibles con .NET podrán utilizar este conjunto de objetos.

Las clases incluidas en System.IO, nos van a permitir realizar labores de lectura y escritura en archivos de texto, binarios, etc., así como la creación y manipulación de los archivos y directorios que contienen la información.

A lo largo de este tema realizaremos una descripción, y ejemplos de uso, de algunas de las clases contenidas en IO. Por lo que, en todos los ejemplos utilizados aquí, tendremos que importar este espacio de nombres.

Objetos Stream

Un objeto Stream representa un flujo o corriente de datos, es decir, un conjunto de información guardada en formato de texto o binario, que podremos leer y escribir sobre un soporte físico, también denominado en la plataforma .NET, almacén de respaldo (backing store).

Algunos tipos de Stream, para optimizar el flujo de transferencia de datos entre el objeto y su medio físico de almacenamiento, disponen de una característica denominada almacenamiento intermedio (buffering), que consiste en mantener un búfer intermedio con los datos. En el caso, por ejemplo, de tareas de escritura, todas las operaciones se realizarían en el búfer, mientras este dispusiera de capacidad. Una vez terminado el proceso de escritura, o cuando el búfer estuviera lleno, su contenido pasaría al archivo físico. Podemos también, alterar el comportamiento por defecto del búfer a través de diversas propiedades y métodos del objeto Stream correspondiente.

Las clases TextReader y TextWriter

Estas clases contienen los miembros genéricos para realizar lectura y escritura de caracteres.

Se trata de clases abstractas, por lo que deberemos utilizar las clases derivadas StreamReader, StreamWriter, StringReader y StringWriter, comentadas a continuación.

La clase StreamWriter

Un objeto StreamWriter realiza operaciones de escritura de texto sobre un archivo.

El proceso típico de escritura de datos mediante un StreamWriter, comprende los siguientes pasos:

- **Instanciar un objeto de esta clase mediante alguno de los constructores disponibles**. Aquí creamos un nuevo archivo para escribir datos sobre él, o abrimos uno existente.

- **Escritura de texto mediante los métodos WriteLine( ) y Write( )**. El primero escribe el texto pasado como parámetro, y añade los caracteres especiales de retorno de carro y nueva línea. El segundo escribe el texto pasado y deja el puntero de escritura a partir del último carácter escrito, con lo que no produce un cambio automático de línea. Deberemos utilizar la propiedad NewLine para introducir manualmente un salto de línea.

- **Cierre del Stream con el método Close( )**. Esta acción vuelca el contenido del búfer del objeto en el archivo.

El Código fuente 410 muestra el proceso básico que acabamos de describir.

```
Imports System.IO

Module Module1
    Sub Main()
        Dim swEscritor As StreamWriter
        ' creamos un stream de escritura, y al mismo tiempo un
        ' nuevo archivo para escribir texto sobre él
        swEscritor = New StreamWriter("\pruebas\NOTAS.txt")

        ' escribir líneas
        swEscritor.WriteLine("esta es la primera línea")
        swEscritor.WriteLine("segunda línea de texto")

        ' ahora escribimos texto pero sin provocar un salto de línea
        swEscritor.Write("Juan y Luna ")
        swEscritor.Write("van de paseo")
        swEscritor.Write(swEscritor.NewLine)    ' esto introduce el salto de línea

        swEscritor.WriteLine("con esta línea cerramos")

        ' cerrar el stream y el archivo asociado
        swEscritor.Close()
    End Sub
End Module
```

Código fuente 410

Algunas de las clases de tipo Stream de escritura disponen del campo compartido Null, que permite realizar una operación de escritura que no será volcada en el medio físico de almacenamiento, con lo que se perderán los datos escritos. Ver el Código fuente 411.

```
' escribir a un medio inexistente (nulo)
swEscritor.Null.WriteLine("este texto no llegará al archivo")
```

Código fuente 411

En el caso de que el archivo sobre el que vamos a escribir ya exista, podemos utilizar un constructor de StreamWriter que nos permite especificar si vamos a añadir texto al archivo o vamos a sobrescribir, perdiendo el texto que hubiera. Veamos un ejemplo en el Código fuente 412.

```
' abre el archivo y se sitúa al final del texto para añadir
swEscritor = New StreamWriter("\pruebas\NOTAS.txt", True)

' se elimina el contenido previo del archivo
swEscritor = New StreamWriter("\pruebas\NOTAS.txt", False)
```

Código fuente 412

Después de crear un objeto de este tipo, y escribir algunas líneas de texto sin cerrar el Stream, si abrimos su archivo de texto correspondiente, nos encontraremos con que no hay texto dentro del archivo. Ello es debido a que todavía no se ha volcado el contenido del búfer del objeto sobre el archivo. Para forzar dicho volcado, deberemos llamar al método Flush( ), que se encarga de traspasar el búfer al archivo asociado al Stream. Veamos el Código fuente 413.

```
Dim swEscritor As StreamWriter
' creamos un stream de escritura
swEscritor = New StreamWriter("\pruebas\NOTAS.txt", False)

' escribir líneas
swEscritor.WriteLine("la primera línea")
swEscritor.WriteLine("un poco más de texto")

' si abrimos el archivo antes de la siguiente, estará vacío
swEscritor.Flush()
' ahora el archivo ya contendrá texto

' cerrar el stream y el archivo asociado
swEscritor.Close()
```

Código fuente 413

La clase StreamReader

Un objeto StreamReader realiza operaciones de lectura de texto sobre un archivo.

El proceso que debemos llevar a cabo para leer el contenido de un Stream de lectura es muy similar al de escritura: instanciar el objeto con uno de sus constructores, abriendo un archivo para leer; ejecutar alguno de los métodos de lectura del StreamReader, y cerrar el objeto con Close( ).

Entre los métodos de lectura de este objeto, tenemos ReadLine( ), que devuelve una línea del archivo; y ReadToEnd( ), que devuelve el resto del contenido del archivo, desde el punto en el que se encontrara el Stream al realizar la última lectura. Veamos unos ejemplos en el Código fuente 414.

```
Dim srLector As StreamReader = New StreamReader("\pruebas\NOTAS.txt")

Console.WriteLine("**Leer una primera línea**")
Dim Linea As String
Linea = srLector.ReadLine()
Console.WriteLine("La línea contiene --> {0}", Linea)

Console.WriteLine()
Console.WriteLine("**Ahora leemos el resto del archivo**")
Dim Texto As String
Texto = srLector.ReadToEnd()
Console.WriteLine("El texto restante contiene --> {0}", Texto)

srLector.Close()

'*********************************************
' leer línea a línea mediante un bucle
Dim srLector As StreamReader = New StreamReader("\pruebas\Datos.txt")
Dim Linea As String
Dim ContadorLin As Integer = 1
Linea = srLector.ReadLine()
Do While Not (Linea Is Nothing)
    Console.WriteLine("Línea: {0} - Contenido: {1}", ContadorLin, Linea)
    ContadorLin += 1
    Linea = srLector.ReadLine()
Loop
```

Código fuente 414

Otro de los métodos de lectura es ReadBlock( ), que recibe como parámetro un array de tipo Char, sobre el que se depositarán una cierta cantidad de caracteres leídos del archivo. En el segundo parámetro de este método indicamos la posición del array desde la que se comenzarán a guardar los caracteres. En el tercer parámetro, el número de caracteres a leer.

El método Read( ), también permite realizar una lectura igual que ReadBlock( ), pero en el caso de no utilizar parámetros, devuelve un valor numérico, correspondiente al código del carácter que acaba de leer. Cuando llega al final del Stream, devuelve –1.

Para convertir de nuevo a carácter los valores que devuelve Read( ), debemos pasar estos valores a un array de tipo Byte, y después, utilizando un objeto ASCIIEncoding, mediante su método GetString( ), pasaríamos el array a una cadena. Veamos unos ejemplos de estos métodos en el Código fuente 415.

```
Imports System.IO
Imports System.Text

Module Module1
    Sub Main()
        ' crear un Stream de lectura
        Dim srLector As StreamReader = New StreamReader("\pruebas\NOTAS.txt")

        ' obtener valores del Stream con el método ReadBlock()
        ' ----------------------------------------------------
        ' crear un array Char que contendrá los caracteres leídos
        Dim Caracteres(15) As Char
        ' leemos 16 caracteres del archivo y los pasamos al array
        ' comenzando a grabarlos a partir de su posición 0
        srLector.ReadBlock(Caracteres, 0, 16)
        ' pasamos el array de valores Char a String mediante
        ' el constructor de la clase String que recibe como
        ' parámetro un array Char
        Dim Parte1 As String = New String(Caracteres)
        Console.WriteLine("Resultado de la lectura con ReadBlock()")
        Console.WriteLine(Parte1)

        Console.WriteLine()

        ' obtener valores del stream con el método Read()
        ' ----------------------------------------------
        Dim Valor As Integer
        Dim Codigos() As Byte

        ' vamos a ir volcando en un bucle los códigos de carácter
        ' leidos desde el archivo a un array Byte
        Valor = srLector.Read()
        While (Valor <> -1)     ' cuando lleguemos al final, obtendremos -1
            If Codigos Is Nothing Then
                ReDim Codigos(0)
            Else
                ReDim Preserve Codigos(Codigos.GetUpperBound(0) + 1)
            End If

            Codigos(Codigos.GetUpperBound(0)) = Valor
            Valor = srLector.Read()
        End While

        Dim Codificador As New ASCIIEncoding()
        Dim Parte2 As String
        ' con el objeto ASCIIEncoding, método GetString(),
        ' obtenemos una cadena, pasando como parámetro un array
        ' de tipos Byte
```

```
        Parte2 = Codificador.GetString(Codigos)
        Console.WriteLine("Resultado de la lectura con ReadBlock()")
        Console.WriteLine(Parte2)

        Console.ReadLine()
    End Sub
End Module
```

Código fuente 415

Finalmente, el método Peek( ), al igual que Read( ), devuelve el siguiente valor disponible del Stream, pero sin extraerlo del búfer, con lo que deberemos utilizar alguno de los métodos anteriormente descritos para realizar una lectura real.

Las clases StringWriter y StringReader

Estas clases proporcionan la misma funcionalidad que StreamWriter y StreamReader, con la diferencia de que StringWriter trabaja con un objeto StringBuilder como almacén de datos, mientras que StringReader utiliza un String para leer su contenido.

La clase Stream (flujo de datos)

La clase Stream representa un flujo o corriente de datos, es decir, un conjunto secuencial de bytes, como puede ser un archivo, un dispositivo de entrada/salida, memoria, un conector TCP/IP, etc.

Se trata de una clase abstracta, por lo que si queremos hacer uso de un stream concreto, tenemos que acudir a alguna de sus clases derivadas como son FileStream, MemoryStream, BufferedStream, etc.

La clase FileStream

Realiza escritura y lectura de bytes sobre un archivo; en el caso de que el archivo no exista, lo crearíamos al mismo tiempo que instanciamos este objeto.

Uno de los constructores de esta clase, nos permite especificar una cadena con la ruta del archivo a utilizar, mientras que en el segundo parámetro utilizaremos un valor de la enumeración FileMode, mediante la que indicamos el modo de trabajo sobre el archivo: añadir, abrir, crear, etc.

Las propiedades CanRead, CanWrite y CanSeek, devuelven un valor lógico que nos informa de si en el objeto podemos realizar operaciones de lectura, escritura y desplazamiento por los bytes que contiene.

Para escribir datos, disponemos del método WriteByte( ), que escribe un byte en el archivo; y también tenemos el método Write( ), que escribe de un array de bytes pasado como parámetro, una cantidad de elementos determinada empezando por una de las posiciones de dicho array. Veamos un ejemplo de escritura en el Código fuente 416.

```
' escrituras con Filestream
Dim oFileStream As FileStream
oFileStream = New FileStream("\pruebas\apuntes.dtt", FileMode.CreateNew)
```

```
oFileStream.Write(New Byte() {15, 160, 88, 40, 67, 24, 37, 50, 21}, 0, 6)
oFileStream.WriteByte(75)
Console.WriteLine("Opciones en el FileStream")
Console.WriteLine("Podemos leer: {0}", IIf(oFileStream.CanRead, "SI", "NO"))
Console.WriteLine("Podemos escribir: {0}", IIf(oFileStream.CanWrite, "SI", "NO"))
Console.WriteLine("Podemos movernos: {0}", IIf(oFileStream.CanSeek, "SI", "NO"))

oFileStream.Close()
oFileStream = Nothing
```

Código fuente 416

Para las operaciones de lectura, tenemos ReadByte( ), que devuelve el valor sobre el que esté posicionado el objeto en ese momento. También disponemos del método Read( ), que traspasa valores un array de bytes.

Si queremos desplazarnos por los elementos del Stream, podemos utilizar el método Seek( ), pasando la cantidad de posiciones a movernos, y el punto desde el que queremos realizar dicho desplazamiento, mediante los valores de la enumeración SeekOrigin.

Para averiguar el elemento del Stream en el que estamos situados, disponemos de la propiedad Position.

Veamos algunos ejemplos de lectura sobre este tipo de objetos, en el Código fuente 417.

```
' lectura con FileStream
Dim oFileStream As FileStream
oFileStream = New FileStream("\pruebas\apuntes.dtt", FileMode.Open)
Dim Valor As Byte
Valor = oFileStream.ReadByte()  ' obtener un valor
Console.WriteLine("Se ha leido el valor: {0}", Valor)

Console.WriteLine("Nos desplazamos dos bytes en el stream")
oFileStream.Seek(2, SeekOrigin.Begin)

Valor = oFileStream.ReadByte()
Console.WriteLine("Se ha leido el valor: {0}", Valor)

Console.WriteLine("La posición actual del stream es: {0}", _
    oFileStream.Position)

' leer varios valores, pasándolos a un array
' previamente dimensionado
Dim VariosValores(3) As Byte
oFileStream.Read(VariosValores, 0, 4)

Console.WriteLine("Leer bloque de valores del stream")
Dim Enumerador As IEnumerator
Enumerador = VariosValores.GetEnumerator()
While Enumerador.MoveNext
    Console.WriteLine("Valor: {0}", Enumerador.Current)
End While

Console.ReadLine()
```

Código fuente 417

Las clases BufferedStream y MemoryStream, que también heredan de Stream, disponen de los mismos miembros que FileStream, teniendo como principal diferencia el que utilizan la memoria de la máquina como almacén de respaldo.

Manejo de datos binarios

Las clases BinaryWriter y BinaryReader, tal y como podemos anticipar por sus nombres, nos permiten escribir y leer respectivamente, datos binarios en archivos, utilizando los métodos ya vistos en clases anteriores. Veamos un ejemplo en el Código fuente 418.

```
' escritura binaria
Dim oBin As New BinaryWriter(New FileStream("\pruebas\info.bin",
FileMode.CreateNew))
oBin.Write("H"c)
oBin.Write("D"c)
oBin.Write("U"c)
oBin.Close()
oBin = Nothing

' lectura binaria
Dim oBinLector As BinaryReader
oBinLector = New BinaryReader(New FileStream("\pruebas\info.bin", FileMode.Open))
Console.WriteLine("Valor 1 del lector: {0}", oBinLector.ReadChar())
Console.WriteLine("Valor 2 del lector: {0}", oBinLector.ReadChar())
Console.WriteLine("Valor 3 del lector: {0}", oBinLector.ReadChar())
oBinLector.Close()

Console.ReadLine()
```

Código fuente 418

Manipulación de archivos mediante File y FileInfo

Las clases File y FileInfo, proporcionan a través de sus miembros, el conjunto de operaciones comunes que podemos realizar con archivos en cuanto a su creación, copia, borrado, etc.

La diferencia principal entre ambas radica en que los miembros de File son todos compartidos, con lo cual se facilita en gran medida su uso, al no tener que crear una instancia previa de la clase; mientras que en FileInfo deberemos crear un objeto para poder utilizarla, ya que sus miembros son de instancia. FileInfo dispone de algunos métodos adicionales que no se encuentran en File.

Comenzando por la clase File, los métodos CreateText( ) y OpenText( ), devuelven respectivamente un objeto StreamWriter y StreamReader, que utilizaremos para escribir y leer en el archivo pasado como parámetro a estos métodos. Con el método Exists( ), comprobamos si existe un determinado archivo. Veamos un ejemplo en el Código fuente 419.

```
Dim sNombreFich As String
Dim srLector As StreamReader
Dim swEscritor As StreamWriter

Console.WriteLine("Introducir ruta y archivo")
```

```
sNombreFich = Console.ReadLine()

If File.Exists(sNombreFich) Then
    srLector = File.OpenText(sNombreFich)
    Console.WriteLine("El archivo contiene:{0}{1}", _
        ControlChars.CrLf, srLector.ReadToEnd())
    srLector.Close()
Else
    swEscritor = File.CreateText(sNombreFich)
    swEscritor.WriteLine("este es")
    swEscritor.WriteLine("un nuevo archivo")
    swEscritor.Close()
End If

Console.WriteLine("Proceso finalizado")
Console.ReadLine()
```

Código fuente 419

Para obtener los atributos de un archivo, disponemos del método GetAttributes( ), al que pasamos la ruta de un archivo, y devuelve un valor de la enumeración FileAttributes con la información sobre los atributos. En el caso de que al intentar acceder a un archivo, este no exista, se producirá una excepción de tipo FileNotFoundException, que podemos tratar en una estructura de manejo de excepciones. Ver el Código fuente 420.

```
Dim sNombreFich As String
Dim oAtributos As FileAttributes

Try
    Console.WriteLine("Introducir ruta y archivo")
    sNombreFich = Console.ReadLine()
    oAtributos = File.GetAttributes(sNombreFich)
    Console.WriteLine("Atributos del archivo: {0}", oAtributos.ToString())

Catch oExcep As FileNotFoundException
    Console.WriteLine("Se ha producido un error {0}{1}", _
        ControlChars.CrLf, oExcep.Message)
Finally
    Console.WriteLine("Proceso finalizado")
    Console.ReadLine()
End Try
```

Código fuente 420

Además de esta excepción, el espacio de nombres IO proporciona algunas clases de excepción adicionales para tratar otras diversas circunstancias de error. Consulte el lector la documentación de la plataforma referente a IO.

Los métodos Copy( ), Move( ) y Delete( ), nos permiten copiar, mover y borrar respectivamente el nombre de archivo que pasemos como parámetro. El método GetCreationTime( ) nos devuelve un tipo Date con la fecha de creación del archivo.

Por otro lado, si queremos obtener información adicional sobre un archivo, como su nombre, extensión, ruta, etc., instanciaremos un objeto FileInfo( ), pasando al constructor una cadena con el nombre del archivo, y utilizaremos algunas de sus propiedades como Name, Extensión, DirectoryName. Veamos una muestra de todo esto en el Código fuente 421.

```
Dim sNombreFich As String
Dim iOperacion As Integer
Dim oFInfo As FileInfo

Console.WriteLine("Introducir ruta y archivo")
sNombreFich = Console.ReadLine()

Console.WriteLine("Fecha creación archivo: {0}", _
    File.GetCreationTime(sNombreFich))

oFInfo = New FileInfo(sNombreFich)

Console.WriteLine("Introducir el número de operación a realizar:")
Console.WriteLine("1 - Copiar")
Console.WriteLine("2 - Mover")
Console.WriteLine("3 - Borrar")
iOperacion = Console.ReadLine()

Select Case iOperacion
    Case 1
        File.Copy(sNombreFich, "\pruebas\distinto" & oFInfo.Extension)

    Case 2
        Console.WriteLine("Vamos a mover el archivo {0}", oFInfo.Name)
        Console.WriteLine("que está en la ruta {0}", oFInfo.DirectoryName)
        File.Move(sNombreFich, "\pruebas\" & oFInfo.Name)
        Console.WriteLine("Completado")
        Console.ReadLine()

    Case 3
        File.Delete(sNombreFich)

End Select
```

Código fuente 421

Manipulación de archivos mediante Directory y DirectoryInfo

Las clases Directory y DirectoryInfo contienen métodos y propiedades para crear, borrar, copiar y mover directorios, así como otra serie de tareas para su manejo y obtención de información.

Al igual que sucedía con las clases del anterior apartado, los miembros de Directory son compartidos, mientras que los de DirectoryInfo son de instancia; esta es su principal diferencia.

En el ejemplo del Código fuente 422, el método Exists( ) comprueba la existencia de un directorio, y en caso afirmativo, obtenemos su última fecha de uso mediante GetLastAccessTime( ). Seguidamente obtenemos un array String con su lista de archivos mediante GetFiles( ), y creamos un subdirectorio de respaldo con CreateSubdirectory( ). En caso de que el directorio no exista, lo creamos con CreateDirectory( ).

```
Dim sNombreDir As String
Dim Archivos() As String
Dim Archivo As String
Dim oDirInfo As DirectoryInfo
```

```
Console.WriteLine("Introducir un nombre de directorio")
sNombreDir = Console.ReadLine()

If Directory.Exists(sNombreDir) Then
    Console.WriteLine("Fecha último acceso: {0}", _
        Directory.GetLastAccessTime(sNombreDir))

    Console.WriteLine("Archivos del directorio {0}", sNombreDir)
    Archivos = Directory.GetFiles(sNombreDir)
    For Each Archivo In Archivos
        Console.WriteLine(Archivo)
    Next

    oDirInfo = New DirectoryInfo(sNombreDir)
    oDirInfo.CreateSubdirectory("bak")
Else
    Directory.CreateDirectory(sNombreDir)
    Console.WriteLine("No existía el directorio, se acaba de crear")
End If
```

Código fuente 422

Para obtener el directorio actual de ejecución, disponemos del método GetCurrentDirectory( ), mientras que si queremos subir al directorio de nivel superior, tenemos el método GetParent( ), que devuelve un tipo DirectoryInfo, con el que podemos, por ejemplo, mostrar su nombre completo mediante la propiedad FullName, y fecha de creación con CreationTime. Veamos el Código fuente 423.

```
Dim sNombreDir As String
Dim oDirInfo As DirectoryInfo

' obtenemos el directorio actual de ejecución
sNombreDir = Directory.GetCurrentDirectory()
Console.WriteLine("Directorio actual: {0}", sNombreDir)

' obtenemos el directorio padre del actual,
' y mostramos información de dicha directorio
oDirInfo = Directory.GetParent(sNombreDir)
Console.WriteLine("Directorio padre y fecha de creación {0}{1}{2}{3}", _
    ControlChars.CrLf, oDirInfo.FullName, _
    ControlChars.CrLf, oDirInfo.CreationTime)
```

Código fuente 423

En el siguiente ejemplo, el método GetDirectories( ) devuelve un array de cadenas, con los nombres de los subdirectorios que se encuentran dentro del directorio pasado como parámetro a este método. A continuación, mediante el método Move( ), cambiamos de lugar un directorio; con Delete( ) borramos otro de los directorios. Observe el lector, cómo utilizando de forma combinada CType( ), Directory.GetFiles( ), y un elemento del array que contiene la lista de directorios, creamos una expresión que nos permite averiguar, si en un determinado directorio hay o no archivos. Ver el Código fuente 424.

```
Dim sNombreDir As String
Dim oDirInfo As DirectoryInfo
Dim sDirectorios() As String
```

```
Dim sDirectorio As String

Console.WriteLine("Introducir un nombre de directorio")
sNombreDir = Console.ReadLine()

' obtener directorios del directorio especificado
sDirectorios = Directory.GetDirectories(sNombreDir)

' comprobar que el directorio contiene a su vez
' varios directorios; en caso negativo, finalizar
If Not (sDirectorios.Length > 1) Then
    Console.WriteLine("El directorio especificado debe contener al menos dos
subdirectorios")
    Console.ReadLine()
    Exit Sub
End If

' mostrar nombres de directorios
For Each sDirectorio In sDirectorios
    Console.WriteLine(sDirectorio)
Next

' mover uno de los directorios a otra ubicación del disco actual
Directory.Move(sDirectorios(0), "\temp\BIS")

' borrar otro de los directorios;
' el directorio a borrar debe estar vacío;
' comprobar con la siguiente expresión si dicho
' directorio contiene o no archivos
If (CType(Directory.GetFiles(sDirectorios(1)), String()).Length() > 0) Then
    Console.WriteLine("No se puede borrar el directorio: {0} - " & _
        "contiene archivos", sDirectorios(1))
Else
    Directory.Delete(sDirectorios(1))
End If

Console.WriteLine("Completado")
Console.ReadLine()
```

Código fuente 424

La clase Path

Esta clase nos proporciona un conjunto de campos y métodos compartidos, para la obtención de información y manipulación de rutas de archivos. El Código fuente 425 muestra un ejemplo en el que, una vez introducido un directorio, se muestra la información de cada uno de sus archivos, en lo que respecta a los métodos de esta clase.

```
Console.WriteLine("Introducir nombre de directorio")
Dim sDirectorio As String
sDirectorio = Console.ReadLine()

Dim sArchivos() As String
sArchivos = Directory.GetFiles(sDirectorio)

Console.WriteLine("Datos sobre archivos obtenidos del objeto Path")
Console.WriteLine("=============================================")
Dim sArchivo As String
For Each sArchivo In sArchivos
    Console.WriteLine("GetDirectoryName() {0}", Path.GetDirectoryName(sArchivo))
    Console.WriteLine("GetExtension() {0}", Path.GetExtension(sArchivo))
```

```
    Console.WriteLine("GetFileName() {0}", Path.GetFileName(sArchivo))
    Console.WriteLine("GetFileNameWithoutExtension() {0}",
Path.GetFileNameWithoutExtension(sArchivo))
    Console.WriteLine("GetFullPath() {0}", Path.GetFullPath(sArchivo))
    Console.WriteLine()
Next

Console.ReadLine()
```

Código fuente 425

Monitorización del sistema de archivos con FileSystemWatcher

Esta clase contiene los mecanismos necesarios, que nos van a permitir la creación de objetos que actúen como observadores de los sucesos que ocurran en el sistema de archivos de un equipo local o remoto en cuanto a la creación, borrado, modificación, etc., de archivos y directorios.

La creación de este proceso de vigilancia podemos dividirla en dos partes: instanciación y configuración del propio objeto FileSystemWatcher; y la escritura de los procedimientos manipuladores de los diversos eventos que pueden ocurrir sobre los archivos y directorios.

Para facilitar la escritura de los manipuladores de evento, podemos declarar una variable de esta clase a nivel de módulo, con la palabra clave WithEvents. Ver Código fuente 426.

```
Module Module1

    Private WithEvents oFSW As FileSystemWatcher

    Sub Main()
        '....
        '....
```

Código fuente 426

Declarado el objeto FileSystemWatcher, lo instanciaremos y asignaremos valor a las propiedades mencionadas a continuación, que nos permitirán configurar el modo de observación que realizará este objeto sobre los archivos.

- **Path**. Tipo String. Contiene la ruta de la unidad de disco sobre la que se efectuará la monitorización.

- **Filter**. Tipo String. Contiene el tipo de fichero que se va a observar, admitiendo los caracteres comodín; por ejemplo: “*.*”, “*.txt”.

- **IncludeSubdirectories**. Tipo Boolean. Establece si se van a monitorizar los subdirectorios de la ruta especificada en la propiedad Path. El valor True incluye los subdirectorio, mientras que False no los incluye.

- **EnableRaisingEvents**. Tipo Boolean. Activa el proceso de observación sobre el sistema de archivos, teniendo en cuenta la configuración establecida en las demás propiedades

mencionadas arriba. El valor True pone en marcha el mecanismo de observación, mientras que el valor False lo detiene.

Veamos un ejemplo de estas propiedades en el Código fuente 427.

```
Sub Main()
    ' instanciar objeto FileSystemWatcher
    oFSW = New FileSystemWatcher()
    ' configurar objeto
    oFSW.Path = "C:\pruebas"
    oFSW.Filter = "*.txt"
    oFSW.IncludeSubdirectories = True

    ' activar
    oFSW.EnableRaisingEvents = True

    ' mientras que no pulsemos S, el objeto inspeccionará
    ' el sistema de archivos del equipo
    While (Console.ReadLine() <> "S")
    End While
End Sub
```

Código fuente 427

Para completar este proceso que estamos describiendo, sólo nos queda escribir los procedimientos que van a ejecutarse cuando se realice la creación, borrado, modificación, etc., de un archivo.

Puesto que hemos declarado la variable FileSystemWatcher a nivel del módulo de código, seleccionaremos dicha variable en la lista desplegable *Nombre de clase*, del editor de código. Seguidamente, abriremos la lista *Nombre de método*, también del editor; seleccionando el evento a codificar.

Las anteriores acciones, crearán el procedimiento de evento correspondiente, pero vacío, por lo que tendremos que escribir el código que queramos ejecutar en respuesta a tal evento. La lista de parámetros de este procedimiento consiste en un tipo Object, que contiene la referencia al objeto FileSystemWatcher que originó el evento; y un tipo FileSystemEventArgs, que contiene información adicional sobre el evento ocurrido, como el nombre y ruta del archivo.

El Código fuente 428 muestra los procedimientos de evento que se ejecutarán cuando se cree o borre un archivo.

```
' al crear un fichero se ejecutará este procedimiento de evento
Public Sub oFSW_Created(ByVal sender As Object, ByVal e As
System.IO.FileSystemEventArgs) Handles oFSW.Created
    Console.WriteLine("Se ha creado un archivo : {0}", e.FullPath)
End Sub

' al borrar un fichero se ejecutará este procedimiento de evento
Public Sub oFSW_Deleted(ByVal sender As Object, ByVal e As
System.IO.FileSystemEventArgs) Handles oFSW.Deleted
    Console.WriteLine("Se ha producido el borrado: {0}", e.FullPath)
End Sub
```

Código fuente 428

Ajuste preciso de filtros para el monitor de archivos

Si queremos realizar un filtro más puntual, por ejemplo, cuando hagamos cambios sobre los archivos a monitorizar, la clase FileSystemWatcher dispone de la propiedad NotifyFilter, que contiene una enumeración de tipo NotifyFilters, cuyos valores podemos combinar para que sólo se detecten los cambios al modificar el tamaño y/o la última escritura sobre un archivo. La Figura 216 muestra un ejemplo del uso de esta propiedad.

```
  ' para concatenar valores de la enumeración,
  ' utilizamos el operador Or
  oFSW.NotifyFilter = NotifyFilters.Size Or NotifyFilters.LastWrite Or |

  ' activar
  oFSW.EnableRaisingEvents = True

  ' mientras que no pulsemos S, el objeto inspeccionará
  ' el sistema de archivos del equipo
  While (Console.ReadLine() <> "S")
  End While
nd Sub
```

NotifyFilters.Attributes
NotifyFilters.CreationTime
NotifyFilters.DirectoryName
NotifyFilters.FileName
NotifyFilters.LastAccess
NotifyFilters.LastWrite
NotifyFilters.Security
NotifyFilters.Size

Figura 216. Uso de filtros de notificación para un objeto FileSystemWatcher.

Establecer el procedimiento de evento con AddHandler

Además de crear los procedimientos de evento de la forma descrita en apartados anteriores, podemos emplear una técnica más flexible, que nos permite conectar los eventos del objeto con sus manipuladores, utilizando la palabra clave AddHandler. El Código fuente 429 muestra un ejemplo de esta situación.

```
Sub Main()
    ' instanciar objeto FileSystemWatcher
    Dim oFSW As New FileSystemWatcher()

    ' configurar objeto
    oFSW.Path = "C:\pruebas"
    oFSW.Filter = "*.txt"
    oFSW.IncludeSubdirectories = True

    ' conectamos manualmente los eventos del objeto
    ' con los procedimientos manipuladores de esos eventos
    AddHandler oFSW.Created, AddressOf oFSW_Created
    AddHandler oFSW.Changed, AddressOf CambioProducido

    ' activar
    oFSW.EnableRaisingEvents = True

    ' mientras que no pulsemos S, el objeto inspeccionará
    ' el sistema de archivos del equipo
    While (Console.ReadLine() <> "S")
    End While
End Sub

Public Sub oFSW_Created(ByVal sender As Object, ByVal e As
System.IO.FileSystemEventArgs)
    Console.WriteLine("Se ha creado un archivo: {0}", e.FullPath)
End Sub
```

```
Public Sub CambioProducido(ByVal emisor As Object, ByVal argumentos As
FileSystemEventArgs)
    Console.WriteLine("Se ha cambiado el archivo: {0}", argumentos.FullPath)
End Sub
```

Código fuente 429

Observe el lector, que para el nombre del procedimiento manipulador de evento, podemos emplear tanto el formato que utiliza el editor de código, como otro nombre cualquiera.

Para el evento de creación de archivo hemos utilizado el formato que usa también el editor, consistente en poner el nombre de objeto, guión bajo, y nombre de evento: oFSW_Created( ).

Sin embargo para el evento de modificación de archivo hemos utilizado un nombre que no se ajusta en absoluto al formato del editor: CambioProducido( ).

Consideraciones sobre la ruta de archivos

El modo en que asignemos a la propiedad Path del objeto FileSystemWatcher, la cadena con la ruta a inspeccionar, influirá en el modo en que recuperemos la información del evento en el procedimiento manipulador correspondiente.

Si asignamos a Path la ruta, sin especificar la unidad de disco, al intentar utilizar la propiedad FullName del objeto FileSystemEventArgs, en el procedimiento de evento, se producirá un error. Ver el Código fuente 430.

```
Sub Main()
    '....
    oFSW.Path = "\pruebas"
    '....
End Sub

Public Sub oFSW_Created(ByVal sender As Object, ByVal e As
System.IO.FileSystemEventArgs)
    ' al intentar utilizar la propiedad FullPath ocurrirá un error
    Console.WriteLine("Se ha creado un archivo: {0}", e.FullPath)
End Sub
```

Código fuente 430

Para que en el anterior ejemplo no se produzca un error, debemos indicar también la letra de unidad correspondiente al asignar la ruta a Path. Ver Código fuente 431.

```
oFSW.Path = "C:\pruebas"
```

Código fuente 431

Detección con espera, de eventos producidos sobre archivos

El método WaitForChanged( ) de la clase FileSystemWatcher, devuelve un objeto de tipo WaitForChangedResult, el cual efectúa una parada en la ejecución, quedando a la espera de que ocurra un determinado evento sobre el sistema de archivos. Una vez que dicho evento se produzca, se continuará la ejecución del programa.

El tipo de evento que ponemos a la espera, lo definimos pasando como parámetro al método WaitForChanged( ), un valor de la enumeración WatcherChangeTypes. Veamos un ejemplo en el Código fuente 432.

```
Dim oFSW As New FileSystemWatcher()
'....
' crear un objeto de espera para un evento
Dim oWFCR As WaitForChangedResult
oWFCR = oFSW.WaitForChanged(WatcherChangeTypes.Created)
Console.WriteLine("Se ha creado el archivo: {0}", oWFCR.Name)
'....
```

Código fuente 432

Manipulación de archivos mediante funciones específicas de Visual Basic

Como comentábamos al comienzo de este tema, en anteriores versiones de VB, el programador tenía a su disposición un grupo de instrucciones como Open, Input, Write, etc., para la lectura y escritura de información en archivos.

Por cuestiones de compatibilidad y migración de aplicaciones existentes, estas instrucciones han sido transformadas en funciones, para facilitar su manejo.

Funciones como FileOpen( ), para abrir un archivo; FileClose( ), para cerrarlo; LineInput( ), para leer una línea de texto de un archivo, etc, son las que permiten en la actual versión del lenguaje, realizar las operaciones que anteriormente efectuábamos mediante sus correspondientes instrucciones.

El Código fuente 433 muestra un pequeño ejemplo, en el que se abre un fichero de texto y se lee su contenido utilizando algunas de estas funciones. Consulte el lector, la documentación de la plataforma, para una mayor información.

```
Dim iNumArchivo As Integer
' obtener número de manipulador de archivo libre
iNumArchivo = FreeFile()

' abrir archivo para lectura
FileOpen(iNumArchivo, "\cubo\notas.txt", OpenMode.Input)

Dim sLinea As String
' recorrer archivo hasta el final
While Not EOF(iNumArchivo)
    ' leer una línea del archivo
```

```
    sLinea = LineInput(iNumArchivo)
    Console.WriteLine(sLinea)
End While

' cerrar el archivo
FileClose(iNumArchivo)

Console.ReadLine()
```

Código fuente 433

A pesar de que estas funciones nos permiten la manipulación de ficheros, debemos tener muy presente que se trata de elementos fundamentalmente proporcionados para compatibilidad con versiones anteriores, por lo que se recomienda que cuando tengamos que hacer cualquier tipo de operación con archivos en cuanto a su lectura, escritura, manipulación, etc., utilicemos las clases del espacio de nombres IO.

28

Formularios Windows

Interfaces de ventana. Formularios y controles

Es un hecho palpable el que la programación para Internet, ha ganado en los últimos tiempos una importante cuota de desarrollo, en detrimento de las aplicaciones basadas en Windows. Sin embargo, todavía existe un importante conjunto de programas que deberán seguir funcionando en Windows y que tendrán que migrarse a la plataforma .NET.

Para este sector del desarrollo, .NET Framework proporciona una arquitectura renovada, en lo que a la programación de aplicaciones Windows se refiere: los nuevos formularios y controles Windows, que describiremos seguidamente.

Un formulario Windows representa la conocida ventana, que se utiliza en las aplicaciones ejecutadas bajo alguno de los sistemas operativos de la familia Windows: Windows95/98, NT, ME, 2000, XP, etc.

Un control, por otra parte, es aquel elemento situado dentro de una ventana o formulario, y que permite al usuario de la aplicación Windows, interactuar con la misma, para introducir datos o recuperar información.

Dentro de .NET, las ventanas clásicas Windows, reciben la denominación de Windows Forms, o WinForms, para diferenciarlas de los formularios Web o WebForms, que son los que se ejecutan en páginas ASP.NET.

System.Windows.Forms

Este espacio de nombres contiene todos los tipos del entorno, a través de los cuales podremos desarrollar aplicaciones compuestas por formularios Windows, junto a los correspondientes controles que permiten al usuario la interacción con el programa.

El conjunto de clases, estructuras, enumeraciones, etc., de System.Windows.Forms, permiten la creación de aplicaciones Windows, basadas en el nuevo motor de generación de formularios (Form Engine), más potente y versátil que el disponible en anteriores versiones de VB.

La clase Form

Esta clase contiene todos los miembros para la creación y manipulación de formularios.

Tras instanciar un objeto de Form, mediante la configuración de las adecuadas propiedades, podemos crear formularios estándar, de diálogo, de interfaz múltiple o MDI, con diferentes bordes, etc.

Creación de un formulario básico

A pesar de que en los temas iniciales de este texto, se describió el modo de creación de una aplicación basada en formularios Windows, en el presente tema abordaremos la creación y diseño de formularios, desde un mayor número de perspectivas, relacionadas con los estilos de formularios, controles, creación desde código, herencia, etc.

Comenzaremos creando una aplicación basada sólo en un formulario, para repasar el diseñador de formularios del IDE, y algunas de sus propiedades.

Iniciaremos por lo tanto Visual Studio .NET, y crearemos un nuevo proyecto VB.NET de tipo aplicación Windows, proceso este, que ya describimos en los mencionados temas iniciales. En este ejemplo dejaremos el nombre de proyecto que nos propone el propio IDE, que será WindowsApplication1, o bien otro número, si ya hemos creado algún proyecto con este nombre.

En lo que respecta al diseñador del formulario, podemos modificar su tamaño haciendo clic sobre las guías de redimensión que tiene en los bordes de la plantilla de diseño, y arrastrando hasta dar el tamaño deseado. Las guías de color blanco son las que permiten modificar el tamaño, mientras que las de color gris, son fijas. Por ejemplo, si vamos a incluir muchos controles, o un título largo, y el tamaño que tiene por defecto no es lo bastante grande, lo ampliaremos hasta quedar como muestra la Figura 217.

También podemos conseguir el mismo efecto de cambiar el tamaño del formulario desde la ventana de propiedades, asignando valores a la propiedad Size. Para ello, haremos clic en el icono de expansión de esta propiedad y daremos valores a sus elementos X e Y.

Igualmente haremos con la propiedad Location, de modo que cambiaremos las coordenadas iniciales en las que el formulario será visualizado. Para que el formulario se visualice en estas coordenadas que establecemos manualmente, debemos también asignar a la propiedad StartPosition el valor Manual.

Para asignar una imagen de fondo al formulario, recurriremos a la propiedad BackgroundImage, que nos mostrará una caja de diálogo, mediante la que seleccionaremos un archivo con formato gráfico que será mostrado en la superficie de la ventana. Si por algún motivo, necesitamos eliminar dicha imagen de fondo para el formulario, haremos clic derecho sobre el pequeño rectángulo situado al lado del

nombre de la propiedad, y seleccionaremos la opción Restablecer del menú contextual. Ver Figura 218.

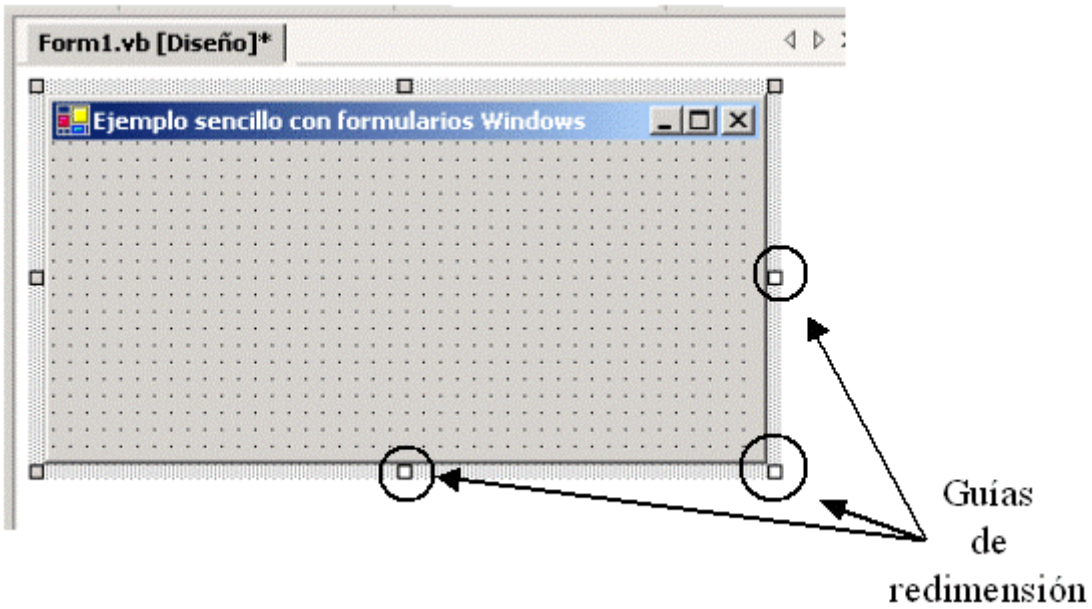


Figura 217. Guías de redimensión del diseñador de formularios.

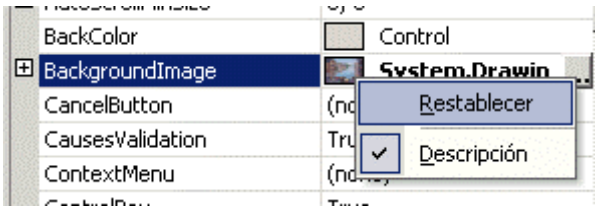


Figura 218. Eliminar imagen de la propiedad BackgroundImage de un formulario.

El icono por defecto lo cambiaremos con la propiedad Icon, seleccionándolo de igual forma que para la imagen de fondo, y asignando un archivo con extensión .ICO.

Finalmente, asignaremos el valor False a la propiedad MaximizeBox, con lo que se deshabilitará el botón del formulario que permite maximizarlo. La Figura 219 muestra el formulario de este ejemplo en ejecución.

Figura 219. Formulario de ejemplo resultante.

El código del formulario

Como ya describimos en los temas iniciales del texto, cuando creamos un formulario desde Visual Studio .NET del modo en que acabamos de mostrar, el diseñador del formulario genera por nosotros el código del formulario, que consiste en una clase que hereda de la clase base Form. El nombre de la clase es el mismo que hemos asignado a la propiedad Name en la ventana de propiedades del diseñador, en este caso Form1. El código es grabado en un archivo con la extensión .VB, que tiene el nombre del formulario: FORM1.VB, en este ejemplo.

Para ver dicho código, tan sólo tenemos que hacer clic derecho sobre el formulario, y en el menú contextual seleccionar *Ver código*, lo que abrirá la ventana del editor de código del IDE, mostrando el código de nuestro formulario.

Parte del código estará oculto por un elemento Region con el nombre *Windows Form Designer generated code*; para verlo al completo debemos hacer clic en el icono de expansión de esta región.

Es posible modificar este código generado por el diseñador, para completar aquellos aspectos que necesitemos del formulario. Sin embargo, no debemos modificar el método InitializeComponent( ), ya que se trata de un método directamente relacionado con el aspecto visual del formulario, y su edición podría dejar el formulario inservible.

El Código fuente 434 muestra el código de la clase Form1, correspondiente al formulario de nuestro proyecto, que ha generado el diseñador.

```
Public Class Form1
    Inherits System.Windows.Forms.Form

#Region " Windows Form Designer generated code "

    Public Sub New()
        MyBase.New()

        'This call is required by the Windows Form Designer.
        InitializeComponent()

        'Add any initialization after the InitializeComponent() call

    End Sub

    'Form overrides dispose to clean up the component list.
    Protected Overloads Overrides Sub Dispose(ByVal disposing As Boolean)
        If disposing Then
            If Not (components Is Nothing) Then
                components.Dispose()
            End If
        End If
        MyBase.Dispose(disposing)
    End Sub

    'Required by the Windows Form Designer
    Private components As System.ComponentModel.Container

    'NOTE: The following procedure is required by the Windows Form Designer
    'It can be modified using the Windows Form Designer.
    'Do not modify it using the code editor.
    <System.Diagnostics.DebuggerStepThrough()> Private Sub InitializeComponent()
        Dim resources As System.Resources.ResourceManager = New
System.Resources.ResourceManager(GetType(frmPrueba))
        '
```

```
        'frmPrueba
        '
        Me.AutoScaleBaseSize = New System.Drawing.Size(5, 13)
        Me.BackgroundImage = CType(resources.GetObject("$this.BackgroundImage"),
System.Drawing.Bitmap)
        Me.ClientSize = New System.Drawing.Size(336, 101)
        Me.Icon = CType(resources.GetObject("$this.Icon"), System.Drawing.Icon)
        Me.Location = New System.Drawing.Point(3200, 6000)
        Me.MaximizeBox = False
        Me.Name = "frmPrueba"
        Me.Text = "Ejemplo sencillo con formularios Windows"

    End Sub

#End Region

End Class
```

Código fuente 434.

Entre los diferentes miembros de esta clase, podemos identificar el método constructor New( ); el método Dispose( ), que podemos utilizar para destruir explícitamente el objeto formulario; y el método InitializeComponent( ), que sirve para inicializar los valores tanto del propio formulario, como de los controles que pudiera contener.

Cambiando el nombre del formulario

Cambiar el nombre de un formulario es algo tan sencillo como acceder a la ventana de propiedades del diseñador del formulario, y asignar un nuevo nombre en la propiedad Name. Por ejemplo, asignemos frmPrueba como nuevo nombre al formulario de nuestro ejemplo. Ver Figura 220.

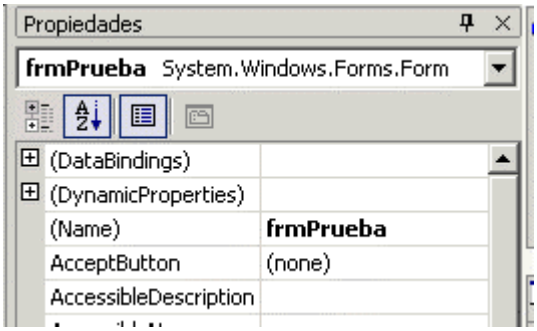


Figura 220. Cambio del nombre del formulario.

Sin embargo, esta acción tiene más implicaciones de las que en un principio pudiera parecer, ya que si intentamos ahora ejecutar el programa, se producirá un error.

Esto es debido a que al crear el proyecto, el objeto inicial del mismo era el formulario, pero tenía como nombre Form1, al cambiar el nombre a frmPrueba, el IDE no puede encontrarlo y genera el error.

Para solucionarlo, debemos abrir la ventana del *Explorador de soluciones*; hacer clic en el nombre del proyecto, y después clic en el último botón de esta ventana, que abrirá la ventana correspondiente a las propiedades del proyecto. En dicha ventana, abriremos la lista desplegable *Objeto inicial*, y seleccionaremos el nombre del nuevo formulario. Ver Figura 221.

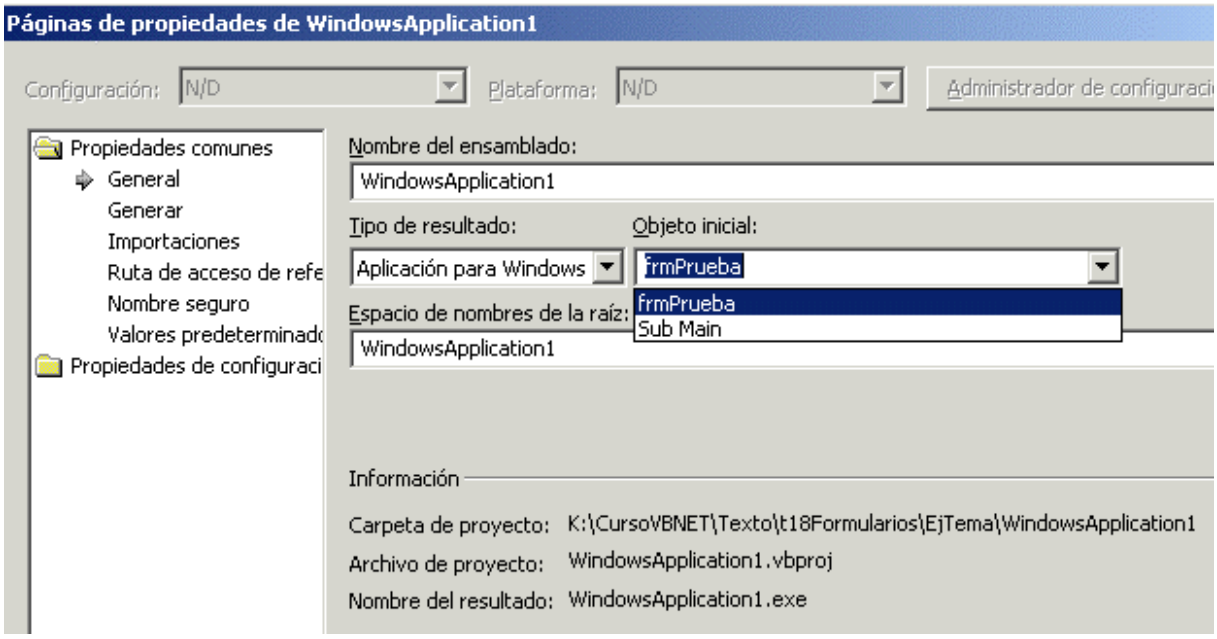


Figura 221. Cambio del objeto inicial del proyecto.

Al volver a ejecutar, el programa funcionará correctamente mostrando el formulario.

Un detalle a destacar consiste en que cuando cambiamos el nombre del formulario, el archivo que contiene el código fuente del mismo no cambia, ya que como sabemos, un archivo de código puede ahora albergar más de una clase o cualquier otro tipo de elemento de la aplicación: enumeración, módulo, estructura, etc. Por ese motivo, el archivo que contiene el formulario seguirá con el nombre FORM1.VB, independientemente del nombre que le hayamos dado al formulario.

Creación de formularios desde código

En el ejemplo anterior hemos visto que a raíz de los valores que asignábamos a las propiedades del formulario, se generaba el código correspondiente, que es el que mostrará el formulario al ser ejecutado.

Aunque en la mayor parte de las ocasiones, sino en todas, utilizaremos el diseñador de formularios para crear el interfaz de usuario, podemos perfectamente prescindir de este diseñador, y construir nuestro formulario escribiendo todo su código. A continuación, mostramos el modo de hacerlo.

Después de haber creado un proyecto de tipo aplicación Windows, eliminaremos el formulario que por defecto nos proporciona el IDE, abriendo la ventana *Explorador de soluciones*, haciendo clic en dicho formulario, y pulsando la tecla [SUPR].

Seguidamente, añadiremos una clase al proyecto utilizando el menú del IDE, *Proyecto + Agregar clase*, y daremos el nombre frmManual a dicha clase, escribiendo en ella el código mostrado en el Código fuente 435.

```
Public Class frmManual
    Inherits System.Windows.Forms.Form

    Public Sub New()
```

```
        Me.Name = "frmManual"
        Me.Text = "formulario creado desde código"
        Me.StartPosition = FormStartPosition.CenterScreen
        Me.ClientSize = New System.Drawing.Size(300, 50)
    End Sub

End Class
```

Código fuente 435.

Como puede comprobar el lector, lo que hacemos en esta clase es heredar de Form, y mediante un método constructor, asignamos valores a las propiedades del formulario que será creado cuando se instancie un objeto de nuestra clase frmManual.

Antes de poder ejecutar este proyecto, debemos, al igual que en el ejemplo anterior, abrir la ventana de propiedades de proyecto, y establecer como objeto inicial esta clase. Nuestro formulario en ejecución tendrá el aspecto mostrado en la Figura 222.

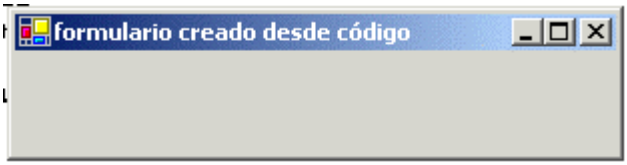


Figura 222. Formulario creado sin utilizar el diseñador de formularios.

La posibilidad de manipular el formulario mediante código de esta manera, abre la puerta a un elevado número de posibilidades, que hasta la fecha, estaban vetadas a los programadores de VB. De esta forma, podemos construir la base principal del formulario con el diseñador, y dinámicamente, en ejecución, modificar sus propiedades, añadir y quitar controles, etc.

Iniciar el formulario desde Main( )

En todos los ejemplos con formularios Windows realizados hasta el momento, la aplicación comienza su ejecución directamente por el formulario, lo cual resulta una comodidad, ya que no tenemos que preocuparnos de configurar el arranque del programa, a no ser que cambiemos el nombre del formulario, como hemos visto en los últimos apartados.

A pesar de todo, este es un escenario, que en muchas ocasiones no será válido, puesto que necesitaremos realizar alguna tarea antes de la visualización del formulario, como cambiar ciertas propiedades del mismo.

Podemos crear un procedimiento Main( ), bien en un módulo o en una clase, y configurarlo como punto de entrada de la aplicación, codificando en dicho procedimiento la instanciación del formulario a mostrar. A continuación mostramos los pasos a dar para conseguirlo.

Creamos un nuevo proyecto Windows, y abrimos la ventana de propiedades del proyecto; en la lista *Objeto inicial*, elegimos esta vez el elemento *Sub Main*. Ver Figura 223.

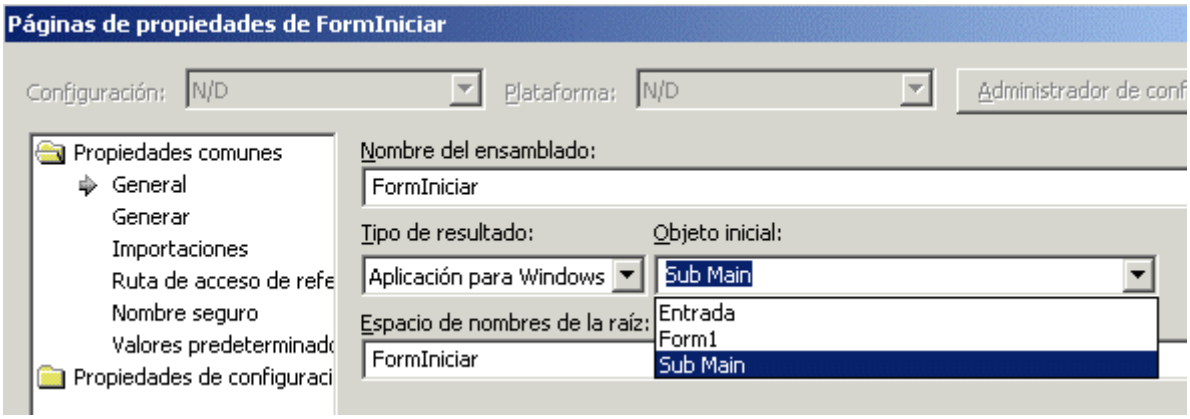


Figura 223. Configurar un proyecto Windows para comenzar por un procedimiento Main( ).

Después añadimos un módulo al proyecto, empleando la opción de menú *Proyecto + Agregar módulo*, de VS.NET, y en dicho módulo codificamos un procedimiento Main( ), que se encargue de instanciar un objeto del formulario. Si escribimos algo parecido a lo que muestra el Código fuente 436, el programa, en efecto, se iniciará, creará el formulario, pero inmediatamente lo cerrará.

```
Module Entrada
    Public Sub Main()
        ' instanciar un objeto de la clase del formulario
        Dim frmVentana As New Form1()
        frmVentana.Text = "probando desde código"
        frmVentana.Show()
    End Sub
End Module
```

Código fuente 436.

El código anterior, aunque válido, tiene un problema: un formulario, al tratarse de una ventana Windows, necesita lo que se denomina un bucle de mensajes, que le permita detectar los mensajes que le envía el sistema operativo, y actuar en consecuencia.

En .NET, para conseguir que un formulario disponga de un bucle de mensajes, debemos utilizar la clase Application, entre cuyos miembros compartidos, se encuentra el método Run( ). Cuando a dicho método, le pasemos un objeto formulario como parámetro, creará un bucle de mensajes para dicho formulario y lo mantendrá en ejecución hasta que el usuario de la aplicación lo cierre.

Modificando pues, el código anterior, por el mostrado en el Código fuente 437, conseguiremos que el formulario permanezca en ejecución una vez creado. Como detalle adicional, y a efectos meramente estéticos, asignamos un color de fondo a la ventana, de modo que el lector compruebe lo sencillo que resulta mediante el uso de la propiedad BackColor, y la estructura Color.

```
Module Entrada
    Public Sub Main ()
        ' instanciar un objeto de la clase del formulario
        Dim frmVentana As New Form1()
        frmVentana.Text = "probando desde código"

        ' asignamos el color de fondo al formulario
        ' utilizando uno de los miembros de la
```

```
        ' estructura Color de la plataforma
        frmVentana.BackColor = Color.Aquamarine

        ' utilizamos el objeto Application y su
        ' método Run() para crear un bucle de
        ' mensajes para el formulario y
        ' ponerlo en ejecución
        Application.Run(frmVentana)
    End Sub
End Module
```

Código fuente 437

La Figura 224 muestra el formulario resultante al ejecutar el proyecto.

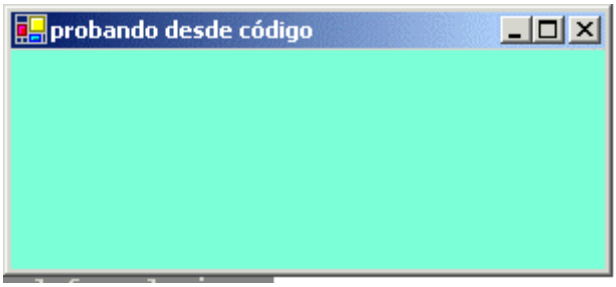


Figura 224. Formulario puesto en ejecución mediante el objeto Application.

Trabajo con controles

Los controles proporcionan al usuario el medio para comunicarse con nuestro formulario, y en definitiva, con la aplicación. Por ello, en los siguientes apartados, mostraremos los principales aspectos que debemos tener en cuenta a la hora de su creación, manipulación y codificación.

También realizaremos una revisión de los principales controles, mostrando algunas de sus características más destacadas.

El Cuadro de herramientas

Una vez creado un proyecto, o después de añadir un nuevo formulario, para utilizar controles en el mismo, tendremos que tomarlos de la ventana *Cuadro de herramientas* disponible en el IDE de VS.NET, y añadirlos al formulario. Ver Figura 225.

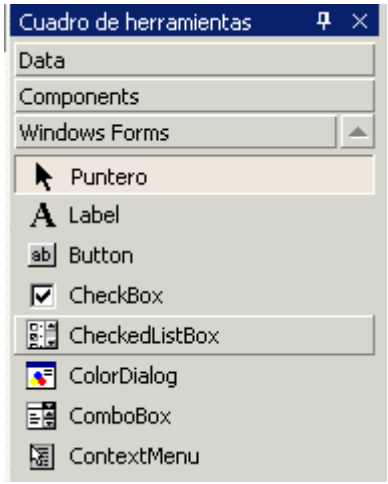


Figura 225. Cuadro de herramientas de Visual Studio .NET.

Insertar un control en el formulario

Para añadir un control en el formulario, proceso que también se conoce como *dibujar un control*, debemos seleccionar primeramente el control a utilizar de la lista que aparece en el cuadro de herramientas.

Una vez localizado el control, haremos doble clic sobre él, o pulsaremos [INTRO], lo que añadirá una copia del mismo en el formulario, que después, mediante el ratón o teclado, situaremos en la posición adecuada.

Otra técnica, esta más habitual, consiste en hacer clic sobre el control, situar el cursor del ratón en la superficie del formulario y hacer clic en él, arrastrando hasta dar la forma deseada; de esta manera, proporcionamos al control en un solo paso la ubicación y tamaño iniciales.

Dibujemos, a modo de práctica, un control Button sobre el formulario, con un aspecto similar al mostrado en la Figura 226.

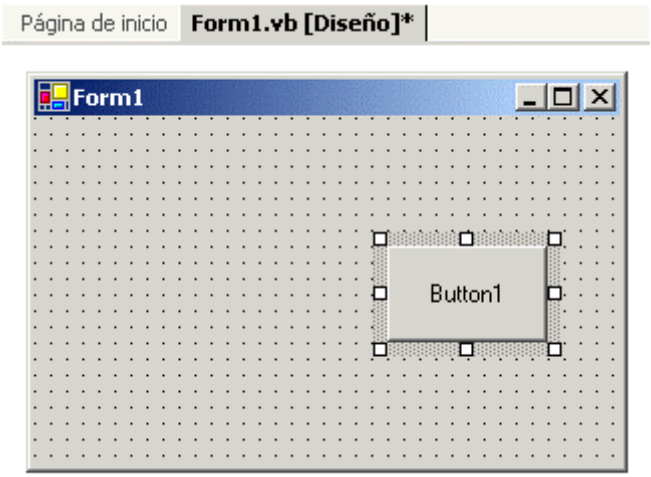


Figura 226. Control Button dibujado sobre el formulario.

Un control, al igual que un formulario, dispone a su alrededor de un conjunto de guías de redimensión, de modo que si después de situarlo en el formulario, queremos modificar su tamaño, sólo tenemos que hacer clic sobre alguna de estas guías, y arrastrar modificando las dimensiones del control.

Además de utilizando el ratón, podemos desplazar un control, manteniendo pulsada la tecla [CONTROL], y pulsando además, algunas de las teclas de dirección.

Ajuste de la cuadrícula de diseño del formulario

La cuadrícula de diseño del formulario, consiste en el conjunto de líneas de puntos que surcan la superficie del formulario, y nos sirven como ayuda, a la hora de realizar un ajuste preciso de un control en una posición determinada.

Si el lector ya ha realizado algunas prácticas situando controles en el formulario, se percatará de que cuando movemos un control con el ratón, dicho control se ajusta a la cuadrícula obligatoriamente, por lo que no podemos ubicarlo entre dos líneas de puntos de la cuadrícula.

Para solventar este problema tenemos algunas soluciones que proponemos a continuación:

La más simple y directa, consiste en acceder a la ventana de propiedades del formulario, y en la propiedad GridSize, cambiar los valores de espaciado de puntos que tiene la rejilla. Cuanto menor sea ese valor, más junta estará la trama de puntos de la rejilla, con lo que podremos ajustar de forma más exacta el control. Este ajuste es válido sólo para el formulario sobre el que lo aplicamos. Probemos por ejemplo, a introducir 5 en cada valor, como muestra la Figura 227.

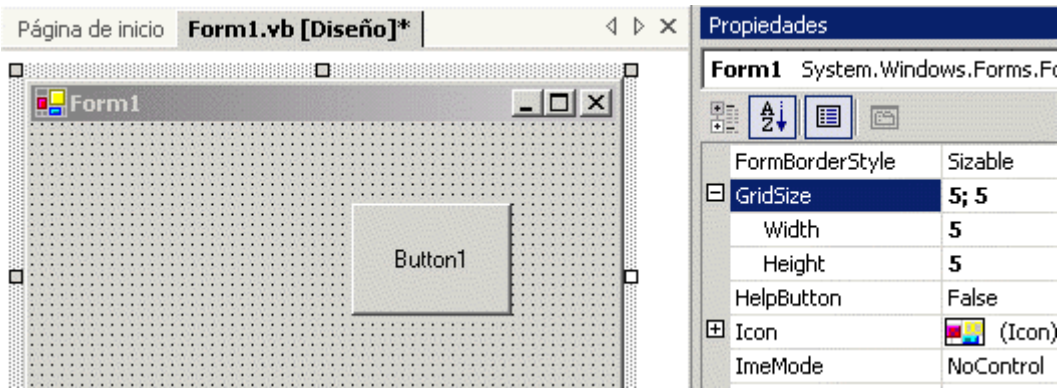


Figura 227. Cambiando el tamaño de la cuadrícula de diseño del formulario.

El otro modo consiste en asignar en el formulario, a la propiedad SnapToGrid, el valor False; esto deshabilita el ajuste a la cuadrícula automático de los controles, con lo que perdemos en precisión de ajuste, pero ganamos en libertad de ubicación para el control. Si no queremos que la cuadrícula se visualice, asignaremos False a la propiedad DrawGrid del formulario.

Los anteriores ajustes los podemos realizar también de modo genérico para todos los formularios. Seleccionaremos para ello, la opción de menú del IDE *Herramientas + Opciones*, y en la ventana Opciones, haremos clic sobre el elemento *Diseñador de Windows Forms*. En el panel derecho de esta ventana, podremos configurar estas propiedades de modo general para todo el IDE. Ver Figura 228.

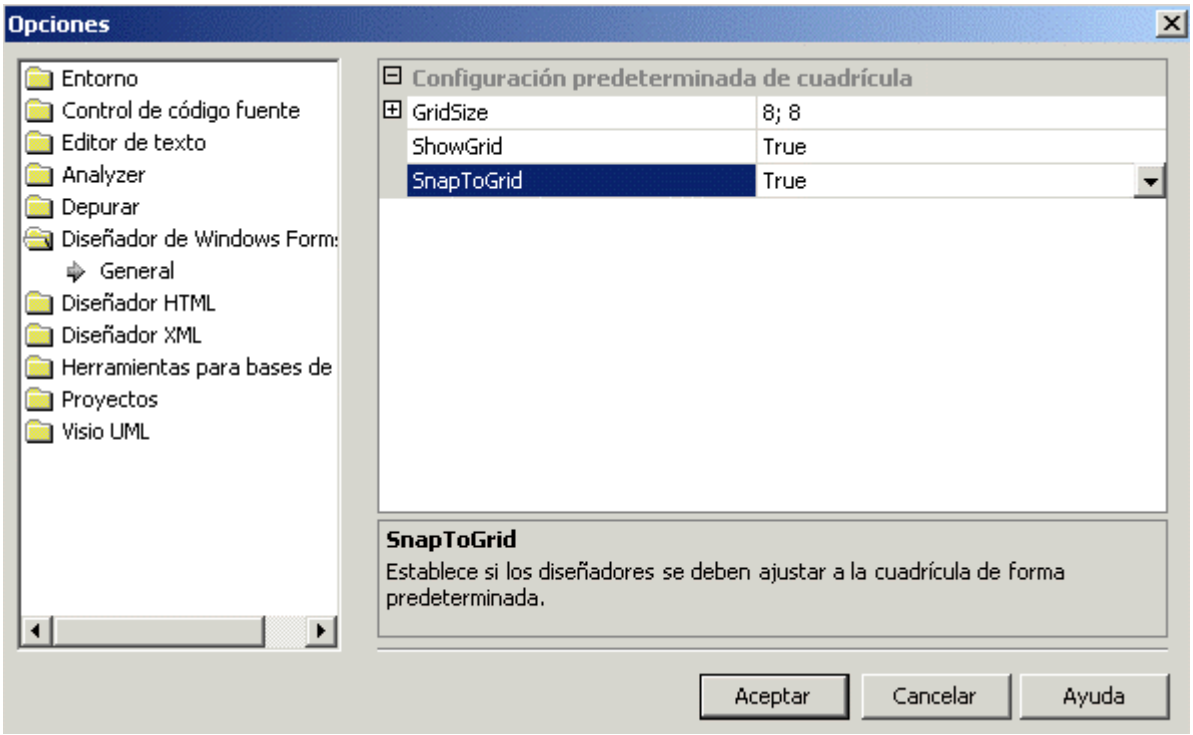


Figura 228. Ajuste general de las propiedades para la cuadrícula de diseño de formularios.

Organización-formato múltiple de controles

Cuando tenemos un grupo numeroso de controles en el formulario, que necesitamos mover de posición, o cambiar su tamaño, para redistribuir el espacio; podemos optar por cambiar uno a uno los controles, tarea pesada y nada aconsejable; o bien, podemos seleccionar todos los controles a modificar, y realizar esta tarea en un único paso, mediante las opciones del menú Formato del IDE.

Supongamos que en el formulario tenemos dos controles Button y un ListBox distribuidos como muestra la Figura 229.

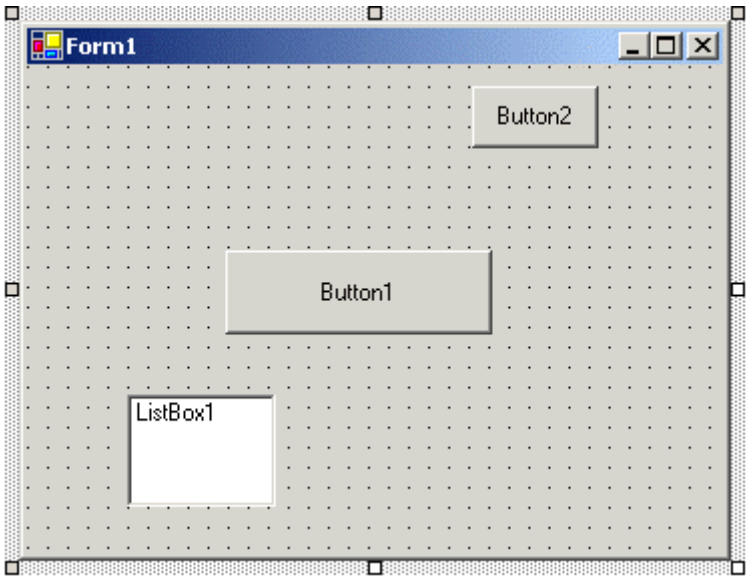


Figura 229. Controles para redistribuir dentro del formulario.

En primer lugar, para seleccionarlos todos, debemos hacer clic sobre el formulario y arrastrar, de modo que el rectángulo de selección que aparece, abarque a los controles, que quedarán con sus correspondientes marcas de redimensión visibles, señal de que están seleccionados.

En este punto, podemos hacer clic en uno de los controles y desplazarlos todos conjuntamente por el formulario, o bien, hacer clic en una de las guías de redimensión y cambiar su tamaño, lo que afectará a todos los controles seleccionados. Si necesitamos de alguna acción especial, utilizaremos las opciones del menú Formato del IDE.

Por ejemplo, podemos ejecutar la opción *Formato + Alinear + Lados izquierdos*, de modo que todos los controles se alinearán por la izquierda, tomando como referencia el control que tiene las marcas de redimensión negras. Ver Figura 230.

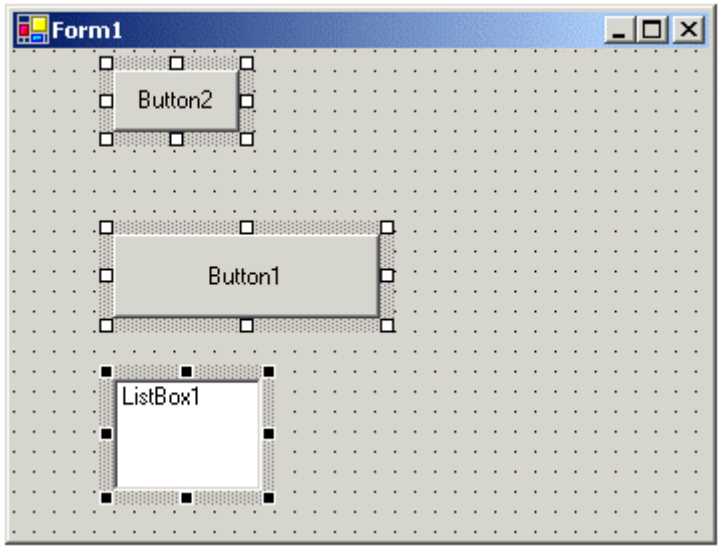


Figura 230. Alineación de controles por su lado izquierdo.

Después ejecutaremos la opción de menú *Formato + Igualar tamaño + Ambos*, que ajustará tanto el ancho como el alto de todos los controles seleccionados. Ver Figura 231.

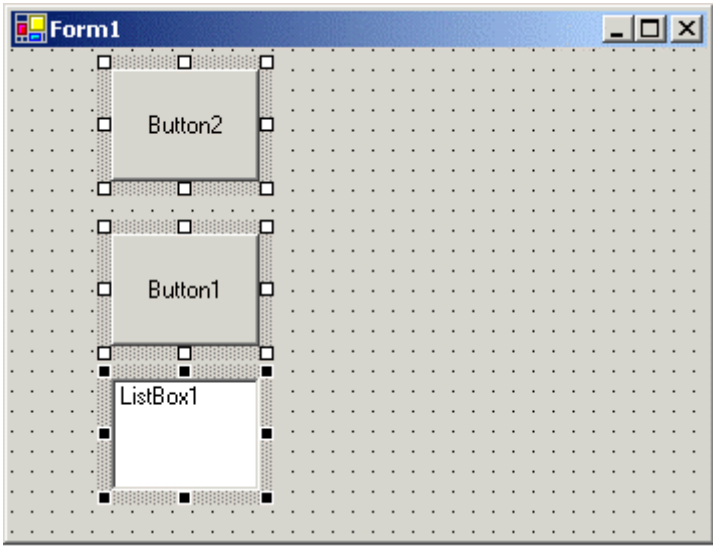


Figura 231. Igualando tamaño de controles.

Para evitar que, una vez completado el diseño y ajuste de todos lo controles, accidentalmente podamos modificar alguno, seleccionaremos la opción de menú *Formato + Bloquear controles*, que bloqueará los controles seleccionados, impidiendo que puedan ser movidos o modificado su tamaño. Para desbloquear los controles del formulario, debemos seleccionar al menos uno y volver a utilizar esta opción de menú, que desbloqueará todos los controles.

Una característica interesante del bloqueo de controles, consiste en que una vez que tengamos bloqueados los controles del formulario, si añadimos un nuevo control, este no estará inicialmente bloqueado, lo que facilita su diseño. Una vez que hayamos finalizado de diseñar el último control, lo seleccionaremos en el formulario y seleccionaremos la opción de bloqueo de controles, de modo que ya estarán bloqueados todos de nuevo.

El menú Formato de VS.NET consta de un numeroso conjunto de opciones. Acabamos de ver una muestra de sus posibilidades, por lo que recomendamos al lector, que realice pruebas con el resto de opciones, para ver todas las posibilidades en cuanto a la disposición de los controles dentro del formulario.

Anclaje de controles

La propiedad Anchor, existente en un gran número de controles, nos permite *anclar* dicho control a uno o varios bordes del formulario.

Cuando un control es anclado a un borde, la distancia entre el control y dicho borde será siempre la misma, aunque redimensionemos el formulario.

Para establecer esta propiedad, debemos pasar a la ventana de propiedades del control, y en Anchor, pulsar el botón que dispone, y que nos mostrará una representación de los bordes para anclar. Ver Figura 232.

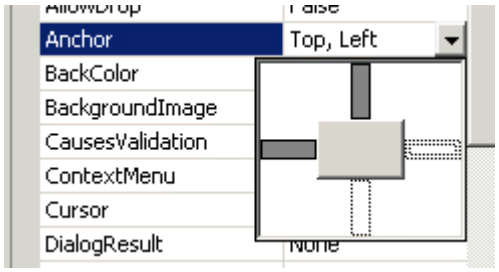


Figura 232. Propiedad Anchor de un control.

Las zonas de color gris oscuro representan los bordes del control que ya están anclados a los bordes del formulario. Debemos marcar y desmarcar respectivamente estos elementos según los bordes que necesitemos anclar. Por defecto, los controles se encuentran inicialmente anclados a los bordes superior e izquierdo (Top, Left), como hemos comprobado en la anterior figura.

La Figura 233 muestra un ejemplo en el que vemos dos controles que tienen distintos tipos de anclaje. Button1 tiene el anclaje normal: Top-Left, mientras que Button2 tiene sólo Right, por ello, su borde derecho siempre mantendrá la misma distancia con ese borde del formulario.

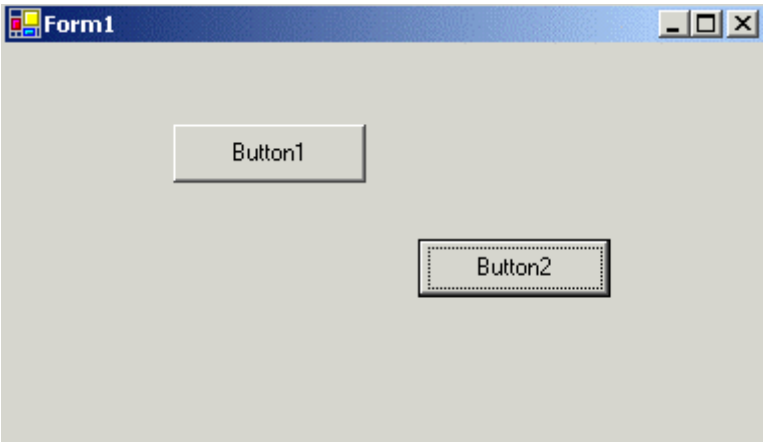


Figura 233. Controles con diferentes valores en la propiedad Anchor.

Acople de controles

A través de la propiedad Dock de los controles, podremos *acoplar* un control a uno de los bordes de un formulario, consiguiendo que dicho control permanezca pegado a ese borde del formulario en todo momento.

Para seleccionar el tipo de acople, haremos clic en el botón que tiene la propiedad Dock en la ventana de propiedades, y que nos mostrará un guía de los tipos de acople disponibles. Ver Figura 234.

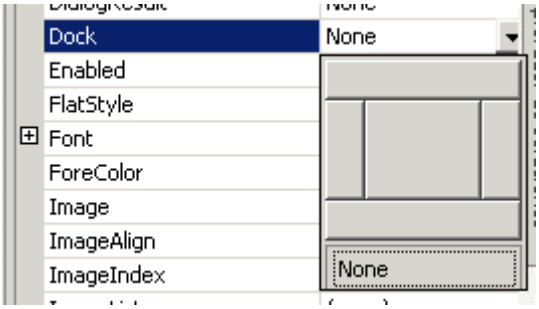


Figura 234. Propiedad Dock, tipos de acople disponibles.

Por defecto, los controles no se encuentran acoplados al insertarse en el formulario, y sólo es posible establecer un tipo de acople en cada ocasión. La Figura 235 muestra un control Button acoplado a la izquierda del formulario.

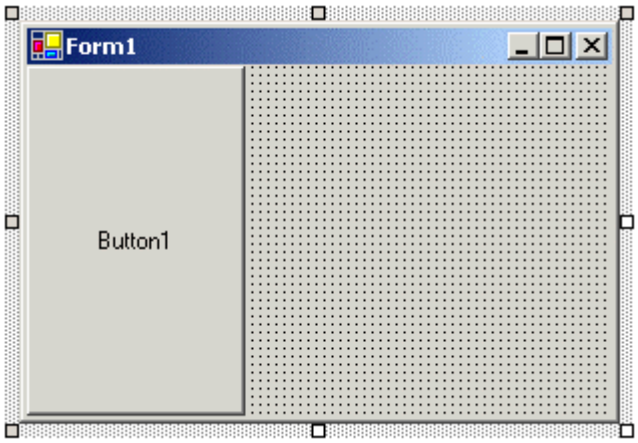


Figura 235. Control acoplado a la izquierda del formulario.

Si pulsamos en la propiedad Dock el botón central de los indicadores de acoplamiento, la propiedad tomará el valor Fill, es decir, el control llenará la superficie del formulario. Veamos en la Figura 236, el mismo control con este valor de acople.

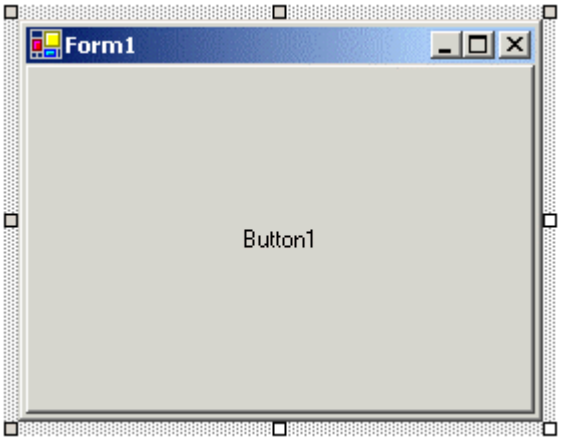


Figura 236. Control con el valor Fill en la propiedad Dock.

29

Controles Windows

Controles más habituales

Como habrá comprobado el lector, el número de controles del cuadro de herramientas es muy numeroso, por lo que en los próximos apartados, vamos a trabajar con los que se consideran controles básicos o estándar, dada su gran frecuencia de uso.

La Tabla 30 relaciona este conjunto de controles básico, junto a una breve descripción.

Control	Descripción
Button	Botón de pulsación
Label	Etiqueta de literal
TextBox	Cuadro de texto
ListBox	Lista de valores
ComboBox	Lista de valores desplegable, y cuadro de texto
CheckBox	Casilla de verificación

RadioButton	Botón autoexcluyente
GroupBox	Caja de agrupación de controles

Tabla 30. Controles básicos de formularios Windows.

Button

Este control representa un botón de pulsación, conocido en versiones anteriores de VB como CommandButton. Entre el nutrido conjunto de propiedades de este control, destacaremos las siguientes.

- **Text**. Cadena con el título del botón.
- **TextAlign**. Alineación o disposición del título dentro del área del botón; por defecto aparece centrado.
- **BackColor**. Color de fondo para el botón.
- **Cursor**. Permite modificar el cursor del ratón que por defecto tiene el botón.
- **Image**. Imagen que podemos mostrar en el botón como complemento a su título, o bien, en el caso de que no asignemos un texto al botón, nos permitirá describir su funcionalidad.
- **ImageAlign**. Al igual que para el texto, esta propiedad nos permite situar la imagen en una zona del botón distinta de la central, que es en la que se ubica por defecto.
- **BackgroundImage**. Imagen de fondo para el botón.
- **FlatStyle**. Tipo de resaltado para el botón. Por defecto, el botón aparece con un cierto relieve, que al ser pulsado, proporciona el efecto de hundirse y recuperar nuevamente su estado, pero podemos, mediante esta propiedad, hacer que el botón se muestre en modo plano, con un ligero remarcado al pulsarse, etc.
- **Font**. Cambia el tipo de letra y todas las características del tipo elegido, para el texto del botón.

La Figura 237 muestra un ejemplo de control Button, sobre el que se han modificado algunos valores por defecto de sus propiedades.

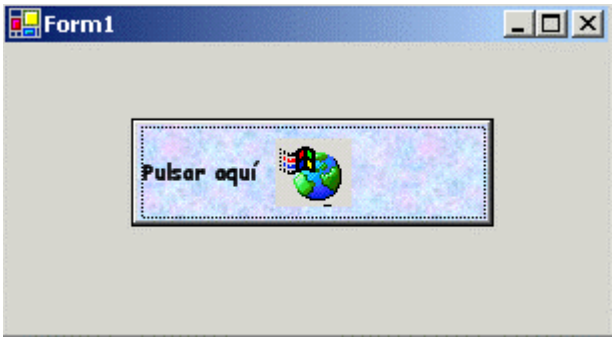


Figura 237. Control Button.

Codificación de los eventos de controles

Windows es un sistema operativo orientado a eventos, por lo que cualquier mínima interacción que realicemos sobre un formulario o control, generará el oportuno evento, para el que si estamos interesados, deberemos responder.

Prosiguiendo con el control Button, cuando pulsamos sobre el mismo, se origina el evento Click de dicho control. Si dibujamos un Button en un formulario y pulsamos en él, no ocurrirá nada, ya que aunque el evento se produce, no existe código que le proporcione respuesta.

Para dar oportuna respuesta a un evento emitido por un control, debemos escribir un procedimiento manipulador del correspondiente evento. La creación de manipuladores de evento es un aspecto que ya hemos visto en detalle en el tema *Delegación de código y eventos*. De igual modo, los aspectos básicos de la escritura de código para un evento se comentaron en el tema *Una aplicación con funcionalidad básica*; sugerimos por lo tanto al lector, la consulta de estos temas para cualquier referencia básica que necesite al respecto.

No obstante, en el presente apartado, y para reforzar conceptos, realizaremos un repaso del proceso de creación del manipulador de evento para un control.

Como ejemplo, insertaremos en un formulario un control Button, al que daremos el nombre btnMensaje, y en su propiedad Text asignaremos la cadena *Mostrar mensaje.*

Seguidamente haremos doble clic sobre el Button; esta acción abrirá la ventana del editor de código, creando al mismo tiempo, la declaración o esqueleto del procedimiento manipulador de evento Click del botón, listo para ser codificado.

Ya que necesitamos que se muestre un mensaje al ser pulsado este control, utilizaremos el objeto MessageBox de la plataforma, llamando a su método compartido Show( ), para visualizar el mensaje. En definitiva, el manipulador de este evento quedaría como muestra el Código fuente 438.

```
Private Sub btnMensaje_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnMensaje.Click

    MessageBox.Show("Se acaba de pulsar el botón del formulario")

End Sub
```

Código fuente 438.

El resultado en ejecución, sería el que muestra la Figura 238.

Observando con más detalle el procedimiento del evento, vemos que al final de su declaración, aparece la palabra clave Handles, que como vimos en el tema sobre eventos, nos sirve para asociar esta rutina de código con un evento de un objeto. En el ejemplo que nos ocupa, asociamos el procedimiento btnMensaje_Click( ), con el evento Click del objeto btnMensaje, perteneciente a la clase Button.

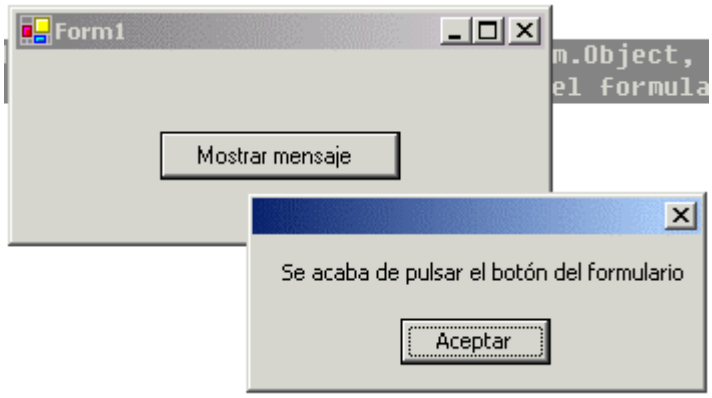


Figura 238. Resultado de la ejecución del evento Click de un control Button, al ser pulsado.

Como ya sabemos, el enlace procedimiento-evento de objeto mediante la palabra Handles, se produce de modo estático. Esto requiere que en el código, el identificador que contenga el objeto del control, deba ser declarado con ámbito a nivel de clase, y utilizando además la palabra clave WithEvents. Dicha tarea es realizada automáticamente por el diseñador del formulario cuando genera el código del mismo. Veamos en el Código fuente 439, el fragmento de código generado por el diseñador que realiza esta labor.

```
' esta declaración es situada a nivel del código
' de la clase del formulario, es decir,
' fuera de cualquier método
Friend WithEvents btnMensaje As System.Windows.Forms.Button
```

Código fuente 439.

Codificando otros eventos de un control

En un control Button, el evento por defecto es Click; esto supone, como acabamos de ver, que al hacer doble clic sobre el control en el formulario, el procedimiento de evento sobre el que nos situará el editor será precisamente este. Sin embargo, un control Button, al igual que el resto de controles de los formularios Windows, disponen de un gran número de eventos que podemos codificar para adaptar a nuestras necesidades.

Por ejemplo, el evento MouseEnter, se produce cuando se detecta que el ratón entra en el área de un control, en este caso Button. Como este no es el evento por defecto, debemos buscar su declaración vacía manualmente en el editor. Para ello, abriremos la lista desplegable *Nombre de clase*, situada en la parte superior izquierda del editor de código, y seleccionaremos el nombre de nuestro control: btnMensaje. Ver Figura 239.

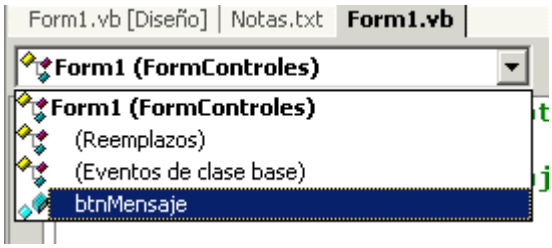


Figura 239. Lista de clases del editor de código.

A continuación, abriremos la otra lista desplegable del editor de código: *Nombre de método*, situada en la parte superior derecha del editor. En ella aparecerán los nombres de todos los eventos de que dispone el control. Localizaremos el evento MouseEnter, y lo seleccionaremos. Ver Figura 240.

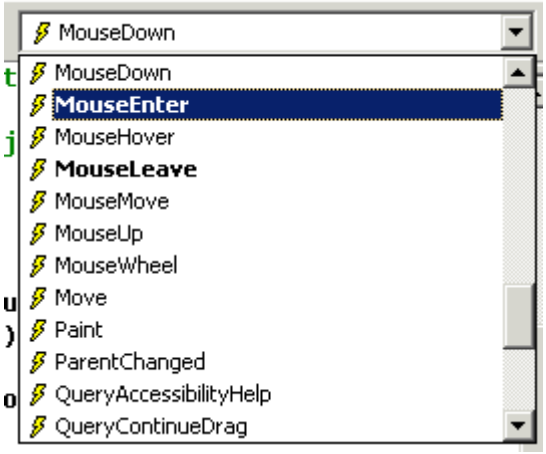


Figura 240. Lista de eventos de una clase-control en el editor de código.

De igual modo que sucedió con el evento Click en el apartado anterior, el editor de código creará el procedimiento manipulador de evento vacío, para el evento que acabamos de seleccionar. Lo que vamos a hacer a continuación, es escribir el código que permita cambiar el color del botón cuando el ratón entre al mismo. Veamos el Código fuente 440.

```
Private Sub btnMensaje_MouseEnter(ByVal sender As Object, ByVal e As
System.EventArgs) Handles btnMensaje.MouseEnter

    Me.btnMensaje.BackColor = Color.Cyan

End Sub
```

Código fuente 440.

Cuando al ejecutar, situemos el ratón en el botón, este cambiará su color, mostrando el aspecto de la Figura 241.

Figura 241. Resultado de la ejecución del evento MouseEnter sobre un Button.

Escritura del manipulador de evento sin usar el nombre proporcionado por el editor

El nombre del procedimiento manipulador de evento, que crea automáticamente el editor de código, por ejemplo: btnMensaje_MouseEnter( ), se basa en una convención establecida por el editor con el siguiente formato: nombre del objeto-guión bajo-nombre del evento.

Sin embargo, esta forma de crear la rutina manipuladora de evento no es obligatoria, ya que podemos crear un procedimiento de evento con el nombre que queramos, siempre y cuando, asociemos el procedimiento con el evento necesario utilizando la palabra clave Handles.

Siguiendo con el ejemplo que estamos desarrollando, una vez que el ratón entra en el área del botón, este cambia de color, pero al salir, no se devuelve el botón a su color inicial; esto es lo que vamos a hacer a continuación.

El evento que se produce cuando el ratón abandona el área de un control es MouseLeave. Conociendo este dato, vamos a escribir un procedimiento con el nombre Salimos( ), que conectaremos con el mencionado evento utilizando Handles. Dentro de esta rutina, haremos una llamada al método ResetBackColor( ) del control, que devuelve el color del botón a su estado original.

Como requerimiento adicional, y ya que los manipuladores de evento de control, internamente están construidos mediante un delegado de tipo EventHandler, debemos incluir dos parámetros en la lista de nuestro procedimiento: uno de tipo Object, y otro de tipo SystemEventArgs. Para seguir el mismo esquema que el resto de eventos, llamaremos a estos parámetros sender y e respectivamente, aunque ello no sería necesario. Al producirse el evento, el control será el encargado de depositar en estos parámetros los objetos correspondientes, de modo transparente para el programador.

Veamos el código de este manipulador de evento en el Código fuente 441.

```
Private Sub Salimos(ByVal sender As Object, ByVal e As System.EventArgs) Handles
btnMensaje.MouseLeave

    Me.btnMensaje.ResetBackColor()

End Sub
```

Código fuente 441

A partir de ahora, cuando ejecutemos el programa, al quitar el ratón de la superficie del botón, el control volverá a tomar su color original.

Respondiendo a los eventos de un formulario

El modo de escribir los manipuladores de eventos para un formulario es igual que para un control. El proyecto de ejemplo EventosForm ilustra este aspecto. En el Código fuente 442 vemos los manipuladores de dos eventos de formulario.

```
Private Sub Form1_MouseMove(ByVal sender As Object, ByVal e As
System.Windows.Forms.MouseEventArgs) Handles MyBase.MouseMove
```

```
    Me.Text = "Coordenadas ratón: X:" & e.X & " Y:" & e.Y

End Sub

Private Sub Form1_Closing(ByVal sender As Object, ByVal e As
System.ComponentModel.CancelEventArgs) Handles MyBase.Closing

    If MessageBox.Show("¿Cerrar la ventana?", _
        "Atención", MessageBoxButtons.YesNo, _
        MessageBoxIcon.Hand) = DialogResult.No Then

        e.Cancel = True

    End If

End Sub
```

Código fuente 442

El evento MouseMove se produce al mover el ratón por el formulario, mientras que Closing se produce cuando el formulario está en proceso de cierre. Este último evento tiene la característica de que nos permite cancelar el proceso de cierre del formulario, mediante la manipulación del parámetro que contiene los argumentos de evento, en concreto se trata de los argumentos de cancelación. La Figura 242 muestra este evento en ejecución.

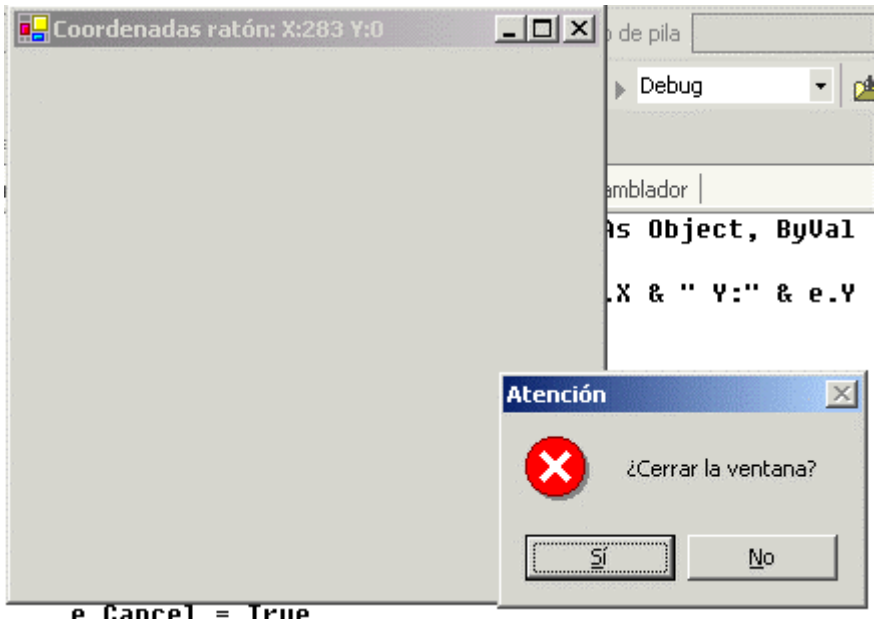


Figura 242. Ejecución del evento Closing de un formulario.

Label

El control Label o Etiqueta, muestra un texto informativo al usuario. Podemos utilizar este control como complemento a otro control, por ejemplo, situándolo junto a un TextBox, de modo que indiquemos al usuario el tipo de dato que esperamos que introduzca en la caja de texto.

Se trata de un *control estático*; esto quiere decir que el usuario no puede interaccionar con él, a diferencia, por ejemplo, de un control Button, sobre el que sí podemos actuar pulsándolo; o de un TextBox, en el que podemos escribir texto.

Una de sus propiedades es BorderStyle, que permite definir un borde o recuadro alrededor del control, o que dicho borde tenga un efecto 3D; por defecto se muestra sin borde. Veamos unos ejemplos en la Figura 243.

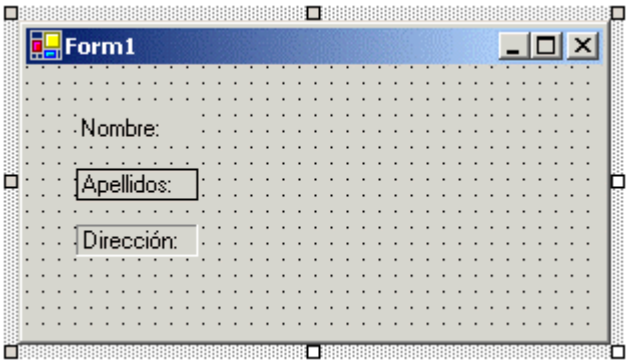


Figura 243. Controles Label.

Foco de entrada

Para que las pulsaciones de teclado puedan ser recibidas por un determinado control, dicho control debe tener lo que se denomina el *foco de entrada*.

El modo de dar a un control el foco de entrada, consiste en hacer clic sobre él, o bien, pulsar la tecla [TAB], pasando el foco hasta el control deseado. Cuando un control recibe el foco, el sistema operativo lo remarca visualmente o en el caso de controles de escritura, muestra el cursor de escritura en su interior.

TextBox

Un control TextBox muestra un recuadro en el que podemos introducir texto. Para poder escribir texto en un control de este tipo, debemos darle primeramente el foco, lo que detectaremos cuando el control muestre el cursor de escritura en su interior.

Entre las propiedades disponibles por este control, destacaremos las siguientes.

- **Text**. Cadena con el texto del control.
- **Multiline**. Permite establecer si podemos escribir una o varias líneas. Por defecto contiene False, por lo que sólo podemos escribir el texto en una línea.
- **WordWrap**. En controles multilínea, cuando su valor es True, al llegar al final del control cuando estamos escribiendo, realiza un desplazamiento automático del cursor de escritura a la siguiente línea de texto.
- **Enabled**. Contiene un valor lógico mediante el que indicamos si el control está o no habilitado para poder escribir texto sobre él.

- **ReadOnly**. Permite indicar si el contenido del control será de sólo lectura o bien, podremos editarlo.

- **CharacterCasing**. Esta propiedad, permite que el control convierta automáticamente el texto a mayúsculas o minúsculas según lo estamos escribiendo.

- **MaxLength**. Valor numérico que establece el número máximo de caracteres que podremos escribir en el control.

- **PasswordChar**. Carácter de tipo máscara, que será visualizado por cada carácter que escriba el usuario en el control. De esta forma, podemos dar a un cuadro de texto el estilo de un campo de introducción de contraseña.

- **AutoSize**. Cuando esta propiedad tenga el valor True, al modificar el tamaño del tipo de letra del control, dicho control se redimensionará automáticamente, ajustando su tamaño al del tipo de letra establecido.

La Figura 244 muestra un formulario con varios controles TextBox, a los cuales se han aplicado diferentes efectos mediante sus propiedades.

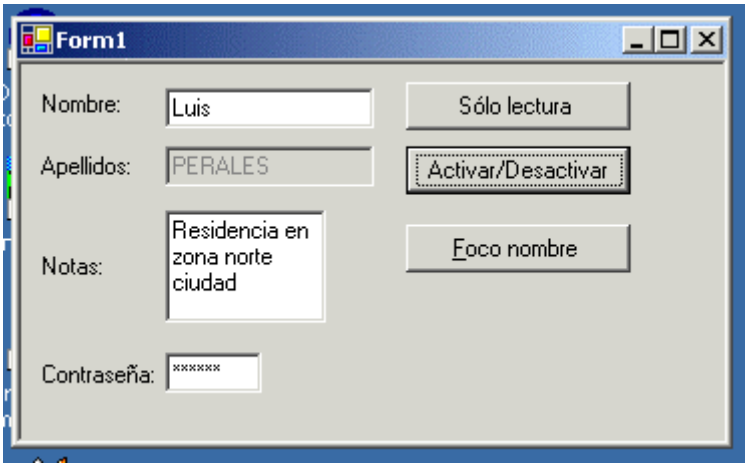


Figura 244. Pruebas con controles TextBox.

Al comenzar a ejecutar el programa, observaremos que el foco de entrada no está situado en el primer TextBox del formulario. Para asignar por código el foco a un determinado control, disponemos del método Focus( ). En este caso, al pulsar el botón *Foco nombre*, desviamos el foco al primer TextBox del formulario. Ver Código fuente 443.

```
Private Sub btnFoco_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnFoco.Click

    Me.txtNombre.Focus()

End Sub
```

Código fuente 443.

Observe el lector, que en el botón *Foco nombre*, que acabamos de mencionar, la letra F se encuentra subrayada, actuando de acelerador o hotkey. De este modo, no es necesario pulsar con el ratón sobre ese botón para ejecutarlo, basta con pulsar la tecla [CONTROL] junto a la letra subrayada para conseguir el mismo efecto.

Para definir una tecla aceleradora en un control, debemos anteponer el carácter & a la letra que vamos a definir como acelerador, en este ejemplo se ha logrado con *&Foco nombre*.

Por otro lado, mediante el botón btnSoloLectura conseguimos activar/desactivar la propiedad ReadOnly del TextBox txtNombre, cambiando el estado de dicha propiedad en cada pulsación del botón. Ver Código fuente 444.

```
Private Sub btnSoloLectura_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnSoloLectura.Click

    If (Me.txtNombre.ReadOnly) Then
        Me.txtNombre.ReadOnly = False
    Else
        Me.txtNombre.ReadOnly = True
    End If

End Sub
```

Código fuente 444

Sin embargo, hay otro modo mucho más eficiente de cambiar el estado de una propiedad que contiene un tipo Boolean: utilizando el operador Not.

Con el botón btnActivar, cambiamos el valor de la propiedad Enabled del cuadro de texto que contiene los apellidos. Para ello, aplicamos el operador Not a dicha propiedad, y el resultado lo asignamos a esa misma propiedad. Ver Código fuente 445.

```
Private Sub btnActivar_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnActivar.Click

    ' utilizando operador Not simplificamos
    Me.txtApellidos.Enabled = Not (Me.txtApellidos.Enabled)

End Sub
```

Código fuente 445

Finalizando con este ejemplo, y aunque no tiene relación directa con el control TextBox, el formulario se muestra con un tipo de borde especial que no permite su redimensión. La propiedad del formulario con la que podemos establecer el tipo de borde es FormBorderStyle, y en este caso, su valor es Fixed3D. Alterando los valores de esta propiedad, conseguiremos distintos bordes y tipos de redimensión para el formulario.

Orden de tabulación de controles

Los controles disponen de la propiedad TabIndex, que contiene un número que será utilizado para pasar el foco entre ellos al pulsar la tecla [TAB] durante la ejecución del programa.

Según vamos añadiendo nuevos controles, el IDE va asignando automáticamente nuevos números a esta propiedad; de forma que, cuando comencemos la ejecución, el primer control que tomará el foco será el que tiene el menor número en su TabIndex.

En el ejemplo anterior, el primer control que tomaba el foco era el TextBox de la contraseña, lo cual no era nada lógico, ya que dicho control era el último en el formulario para el que debíamos introducir datos.

Para solucionar este problema, simplemente tenemos que cambiar los valores de la propiedad TabIndex de los controles, de modo que el orden de tabulación sea el que mejor se adapte a nuestras necesidades.

Podemos obtener un mapa del orden de tabulación de los controles del formulario seleccionando el menú del IDE *Ver + Orden de tabulación*; esto mostrará los controles con el número de TabIndex que les hemos asignado. Como ventaja adicional, en esa situación, podemos hacer clic en los controles y cambiar también el número de tabulación. Ver Figura 245.

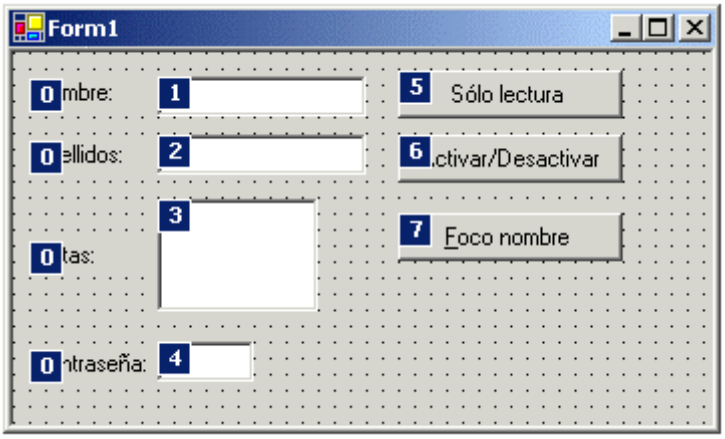


Figura 245. Diseñador del formulario mostrando el orden de tabulación de controles.

Si por el contrario, no queremos dar el foco a un control pulsando [TAB], debemos asignar a la propiedad TabStop de dicho control el valor False. Por defecto, TabStop vale True, permitiendo de esta el paso de foco entre controles mediante la tecla [TAB].

Selección de texto en un TextBox

La selección de texto en un control TextBox es un proceso que funciona de modo transparente al programador, en el sentido en que este no necesita añadir código adicional para las operaciones de selección, cortar, copiar, etc., al ser tareas integradas en el sistema operativo.

Sin embargo, podemos necesitar en un determinado momento, tener información acerca de las operaciones de selección que está realizando el usuario en nuestros controles de texto. Para ello, el control TextBox dispone de las siguientes propiedades.

- **SelectionStart**. Posición del texto del control, en la que comienza la selección que hemos realizado.
- **SelectionLength**. Número de caracteres seleccionados en el control.
- **SelectedText**. Cadena con el texto que hemos seleccionado en el control.

Mediante estas propiedades, no sólo averiguamos la selección que pueda tener un control TextBox, sino que también podemos utilizarlas para establecer por código una selección; teniendo el mismo efecto que si la hubiera efectuado el usuario con el ratón o teclado.

Para comprobar el funcionamiento de las propiedades de selección del TextBox, crearemos un proyecto Windows, y en su formulario añadiremos varios controles para manipular la selección de texto que hagamos en un TextBox. La Figura 246 muestra el formulario del ejemplo.

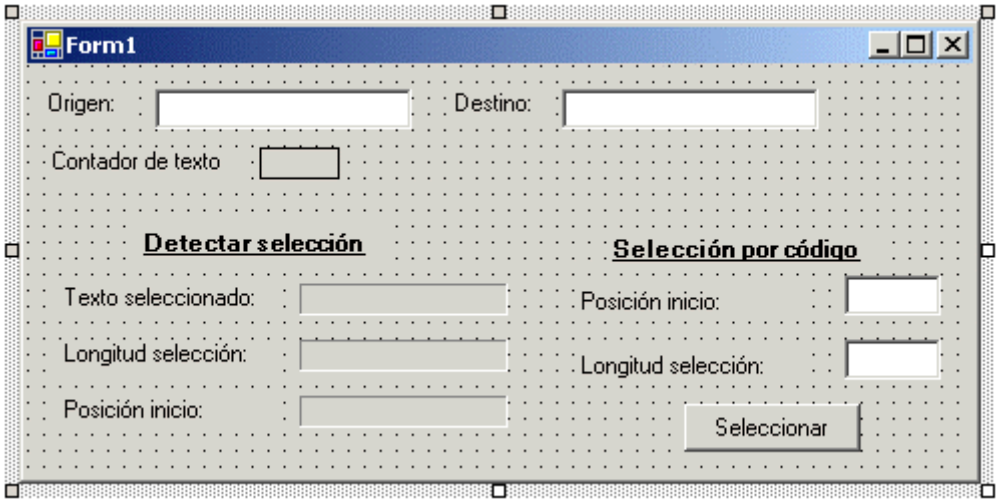


Figura 246. Formulario para realizar selección en el control TextBox

El control de este formulario, que vamos a emplear para las operaciones de selección es txtOrigen. En primer lugar, y aunque no se trata de una selección de texto, veremos su evento TextChanged, el cual se produce cada vez que cambia el contenido del cuadro de texto; lo usaremos por tanto, para contar la cantidad de caracteres escritos y mostrarlos en un Label. Ver Código fuente 446.

```
' al cambiar el texto del control se produce
' este evento
Private Sub txtOrigen_TextChanged(ByVal sender As Object, ByVal e As
System.EventArgs) Handles txtOrigen.TextChanged

    ' calculamos la longitud del texto escrito
    Me.lblContador.Text = Me.txtOrigen.TextLength

End Sub
```

Código fuente 446

Los eventos MouseMove y KeyDown del TextBox, se producen respectivamente, cuando movemos el ratón sobre el control, o cada vez que pulsamos una tecla para escribir texto. Detectaremos en este

caso, si existen teclas o botones especiales presionados, que nos indiquen que se está realizando una selección de texto, y mostraremos en el formulario el texto seleccionado, el número de caracteres y la posición del carácter de inicio de la selección. Veamos los procedimientos manipuladores de estos eventos en el Código fuente 447.

```
' al mover el ratón por el TextBox se produce
' este evento
Private Sub txtOrigen_MouseMove(ByVal sender As Object, ByVal e As
System.Windows.Forms.MouseEventArgs) Handles txtOrigen.MouseMove

    ' comprobamos si al mover el ratón
    ' está pulsado su botón izquierdo

    ' en caso afirmativo es que se está
    ' seleccionando texto, por lo que obtenemos
    ' la información de selección con las
    ' propiedades de selección del TextBox
    If e.Button.Left Then
        Me.lblTextoSelec.Text = Me.txtOrigen.SelectedText
        Me.lblLongitud.Text = Me.txtOrigen.SelectionLength
        Me.lblPosicion.Text = Me.txtOrigen.SelectionStart
    End If

End Sub

' este evento se produce cuando se pulsa
' una tecla en el TextBox
Private Sub txtOrigen_KeyDown(ByVal sender As Object, ByVal e As
System.Windows.Forms.KeyEventArgs) Handles txtOrigen.KeyDown

    ' comprobamos las teclas pulsadas

    ' si está pulsada la tecla mayúsculas,
    ' y además se está pulsando la tecla
    ' flecha derecha, quiere decir que se
    ' está seleccionando texto;
    ' obtener la información de las propiedades
    ' de selección del control TextBox
    If e.Shift Then
        If e.KeyCode.Right Then
            Me.lblTextoSelec.Text = Me.txtOrigen.SelectedText
            Me.lblLongitud.Text = Me.txtOrigen.SelectionLength
            Me.lblPosicion.Text = Me.txtOrigen.SelectionStart
        End If
    End If

End Sub
```

Código fuente 447

Finalmente, tras introducir un valor en los controles txtPosicion y txtLongitud, pulsaremos el botón btnSeleccionar. Con ello conseguiremos realizar una selección de texto en el TextBox txtOrigen, y pasar el texto seleccionado al control txtDestino. El efecto será el mismo que si lo hubiera realizado el usuario, pero en este caso sin su intervención. Veamos en el Código fuente 448, el evento Click del botón btnSeleccionar.

```
' al pulsar este botón, seleccionar por código texto
' del control txtOrigen y pasarlo al control txtDestino
```

```
Private Sub btnSeleccionar_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnSeleccionar.Click

    Me.txtOrigen.SelectionStart = Me.txtPosicion.Text
    Me.txtOrigen.SelectionLength = Me.txtLongitud.Text
    Me.txtDestino.Text = Me.txtOrigen.SelectedText

End Sub
```

Código fuente 448

La Figura 247 muestra este ejemplo en ejecución.

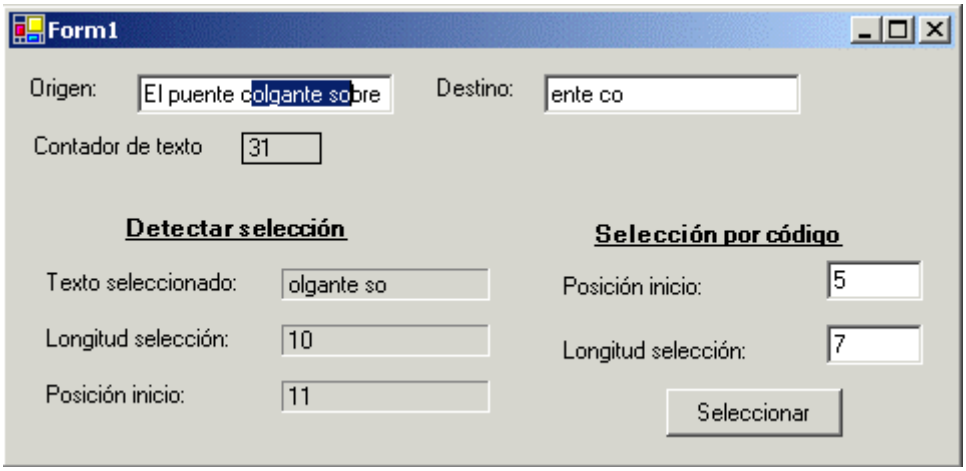


Figura 247. Pruebas de selección de texto con el control TextBox

CheckBox

Este control muestra una casilla de verificación, que podemos marcar para establecer un estado.

Generalmente el estado de un CheckBox es marcado (verdadero) o desmarcado (falso), sin embargo, podemos configurar el control para que sea detectado un tercer estado, que se denomina indeterminado, en el cual, el control se muestra con la marca en la casilla pero en un color de tono gris.

Las propiedades remarcables de este control son las siguientes.

- **Checked**. Valor lógico que devuelve True cuando la casilla está marcada, y False cuando está desmarcada.
- **CheckState**. Valor del tipo enumerado CheckState, que indica el estado del control. Checked, marcado; Unchecked, desmarcado; e Indeterminate, indeterminado.
- **ThreeState**. Por defecto, un control de este tipo sólo tiene dos estados, pero asignando True a esta propiedad, conseguimos que sea un control de tres estados.
- **CheckAlign**. Permite establecer de modo visual la ubicación de la casilla de verificación dentro del área del control.

Como detalle destacable de las propiedades Checked y CheckState, si modificamos desde código sus valores, conseguiremos alterar el estado de la casilla del control.

El ejemplo CheckBoxPru, muestra un formulario con dos controles CheckBox. El control chkPonColor asigna un color de fondo al formulario o restablece el color original. Esto lo conseguimos codificando el evento CheckedChanged del control. Ver Código fuente 449.

```
' este evento se produce cuando se hace clic
' en el CheckBox y cambia el contenido de la casilla
Private Sub chkPonColor_CheckedChanged(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles chkPonColor.CheckedChanged

    If Me.chkPonColor.CheckState = CheckState.Checked Then
        Me.BackColor = Color.LightBlue
    Else
        Me.ResetBackColor()
    End If
End Sub
```

Código fuente 449

Por su parte, el control chkMostrar, definido con tres estados, muestra, al estar marcado, una cadena en un control Label; elimina la cadena al desmarcarlo; y muestra la mitad al entrar en el estado indeterminado. El evento CheckStateChanged es el que debemos de utilizar para detectar el estado del CheckBox en cada ocasión. Para mantener el valor de la cadena a mostrar, utilizamos una variable a nivel de la clase que inicializamos en el constructor del formulario. Ver Código fuente 450.

```
Public Class Form1
    Inherits System.Windows.Forms.Form

    Private sCadenaOriginal As String

    Public Sub New()
        '....
        ' inicializar la variable que contiene la cadena
        ' a mostrar en el label y asignarla
        sCadenaOriginal = "Estamos visualizando una cadena"
        Me.lblMuestra.Text = sCadenaOriginal
    End Sub
'....

' este evento se produce cuando cambia el estado
' de la casilla
Private Sub chkMostrar_CheckStateChanged(ByVal sender As Object, ByVal e As
System.EventArgs) Handles chkMostrar.CheckStateChanged

    Select Case Me.chkMostrar.CheckState
        Case CheckState.Checked
            Me.lblMuestra.Text = sCadenaOriginal

        Case CheckState.Unchecked
            Me.lblMuestra.Text = ""

        Case CheckState.Indeterminate
            Me.lblMuestra.Text = sCadenaOriginal.Substring(0,
(sCadenaOriginal.Length / 2))

    End Select
```

```
End Sub
```

Código fuente 450

La Figura 248 muestra este ejemplo en ejecución.

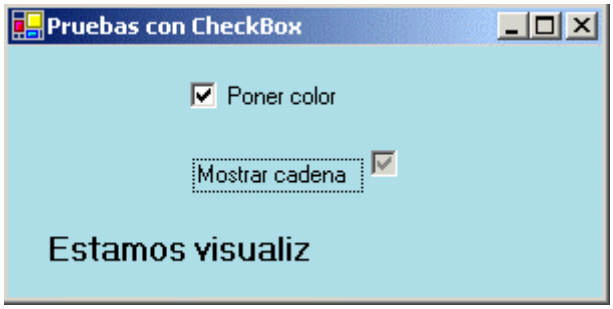


Figura 248. Controles CheckBox en ejecución.

RadioButton y GroupBox

Los controles RadioButton nos permiten definir conjuntos de opciones autoexcluyentes, de modo que situando varios controles de este tipo en un formulario, sólo podremos tener seleccionado uno en cada ocasión.

Vamos a crear un proyecto de ejemplo con el nombre RadioButtonPru, en el que situaremos dentro de un formulario, una serie de controles RadioButton y un TextBox, de modo que mediante los RadioButton cambiaremos el tipo de fuente y color del cuadro de texto. La Figura 249 muestra un diseño inicial del formulario.

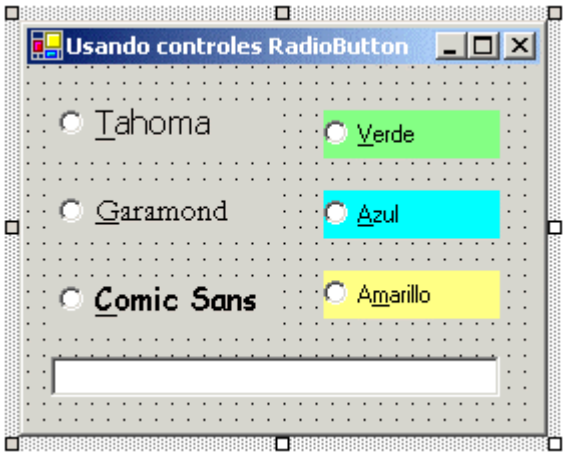


Figura 249. Pruebas con el control RadioButton.

Al ejecutar el proyecto, sin embargo, no podemos conseguir establecer simultáneamente un tipo de letra y color, puesto que al pulsar cualquiera de los botones de radio, se quita el que hubiera seleccionado previamente.

Para solucionar este problema, disponemos del control GroupBox, que nos permite, como indica su nombre, agrupar controles en su interior, tanto RadioButton como de otro tipo, ya que se trata de un control contenedor.

Una vez dibujado un GroupBox sobre un formulario, podemos arrastrar y soltar sobre él, controles ya existentes en el formulario, o crear nuevos controles dentro de dicho control. De esta forma, podremos ya, en este ejemplo, seleccionar más de un RadioButton del formulario, como vemos en la Figura 250.

Figura 250. Selección de varios RadioButton en un formulario.

El evento CheckedChanged, al igual que ocurría con los controles CheckBox, será el que tendremos que escribir para ejecutar el código en respuesta a la pulsación sobre un control RadioButton. El Código fuente 451 muestra los eventos correspondientes a los controles de radio de este ejemplo. Para cambiar el tipo de fuente, instanciamos un objeto Font y lo asignamos a la propiedad Font del TextBox; mientras que para cambiar el color, utilizamos la estructura Color y la propiedad BackColor, también del TextBox.

```
Private Sub rbtTahoma_CheckedChanged(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles rbtTahoma.CheckedChanged

    Me.txtNombre.Font = New Font("Tahoma", 12)

End Sub

Private Sub rbtGaramond_CheckedChanged(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles rbtGaramond.CheckedChanged

    Me.txtNombre.Font = New Font("Garamond", 8)

End Sub

Private Sub rbtComic_CheckedChanged(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles rbtComic.CheckedChanged

    Me.txtNombre.Font = New Font("Comic Sans MS", 15)

End Sub

Private Sub rbtVerde_CheckedChanged(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles rbtVerde.CheckedChanged

    Me.txtNombre.BackColor = Color.Green
```

```
End Sub

Private Sub rbtAzul_CheckedChanged(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles rbtAzul.CheckedChanged

    Me.txtNombre.BackColor = Color.Blue

End Sub

Private Sub rbtAmarillo_CheckedChanged(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles rbtAmarillo.CheckedChanged

    Me.txtNombre.BackColor = Color.Yellow

End Sub
```

Código fuente 451

ListBox

Un control ListBox contiene una lista de valores, de los cuales, el usuario puede seleccionar uno o varios simultáneamente. Entre las principales propiedades de este control, podemos resaltar las siguientes.

- **Items**. Contiene la lista de valores que visualiza el control. Se trata de un tipo ListBox.ObjectCollection, de manera que el contenido de la lista puede ser tanto tipos carácter, como numéricos y objetos de distintas clases. Al seleccionar esta propiedad en la ventana de propiedades del control, y pulsar el botón que contiene, podemos introducir en una ventana elementos para el control. Ver Figura 251.

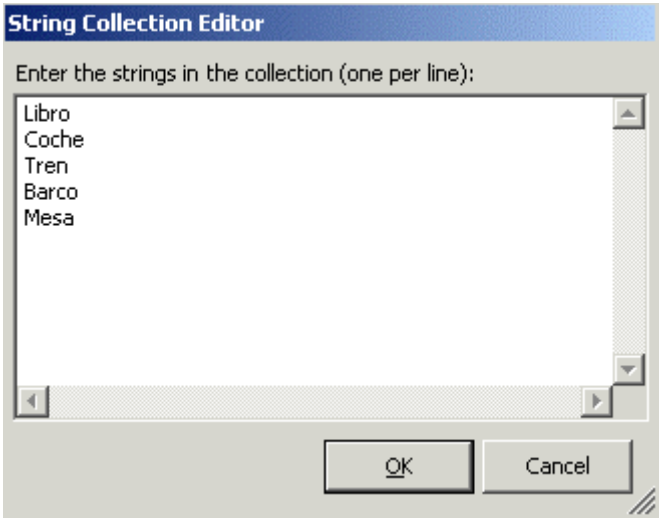


Figura 251. Introducción de valores para un ListBox en tiempo de diseño.

El control quedaría por lo tanto con valores asignados en la etapa de diseño, como muestra la Figura 252.

Figura 252. ListBox en diseño con valores en su lista.

- **Sorted**. Cuando esta propiedad contiene el valor True, ordena el contenido de la lista. Cuando contiene False, los elementos que hubiera previamente ordenados, permanecen con dicho orden, mientras que los nuevos no serán ordenados.

- **IntegralHeight**. Los valores de la lista son mostrados al completo cuando esta propiedad contiene True. Sin embargo, al asignar el valor False, según el tamaño del control, puede que el último valor de la lista se visualiza sólo en parte. La Figura 253 muestra un ListBox con esta propiedad a False.

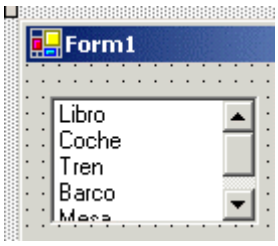


Figura 253. ListBox mostrando parte del último elemento debido a la propiedad IntegralHeight.

- **MultiColumn**. Visualiza el contenido de la lista en una o varias columnas en función de si asignamos False o True respectivamente a esta propiedad.

- **SelectionMode**. Establece el modo en el que vamos a poder seleccionar los elementos de la lista. Si esta propiedad contiene None, no se realizará selección; One, permite seleccionar los valores uno a uno; MultiSimple permite seleccionar múltiples valores de la lista pero debemos seleccionarlos independientemente; por último, MultiExtended nos posibilita la selección múltiple, con la ventaja de que podemos hacer clic en un valor, y arrastrar, seleccionando en la misma operación varios elementos de la lista.

- **SelectedItem**. Devuelve el elemento de la lista actualmente seleccionado.

- **Selecteditems**. Devuelve una colección ListBox.SelectedObjectCollection, que contiene los elementos de la lista que han sido seleccionados.

- **SelectedIndex**. Informa del elemento de la lista seleccionado, a través del índice de la colección que contiene los elementos del ListBox.

Para mostrar algunas de las funcionalidades de este control, utilizaremos el proyecto de ejemplo ListBoxPru. La Figura 254 muestra esta aplicación en ejecución.

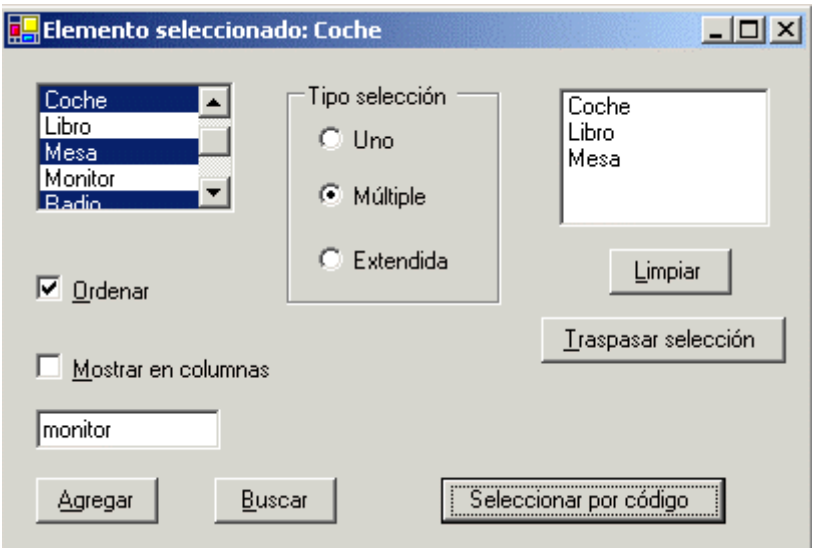


Figura 254. Ejemplo de uso del control ListBox.

El ejemplo, como puede comprobar el lector, consiste en un formulario que contiene un ListBox principal, con el nombre lstValores, que dispone de una serie de valores asignados en tiempo de diseño.

Cada vez que hacemos clic en alguno de los valores, se produce el evento SelectedIndexChanged, que utilizamos para mostrar en este caso, el nombre del elemento en el título del formulario, como muestra el Código fuente 452, de la clase frmListas, correspondiente al formulario.

```
' declaramos esta constante a nivel de clase, para poner como título
' del formulario junto al elemento seleccionado de la lista
Public Const TITULO As String = "Elemento seleccionado: "

' este evento se produce cada vez que se cambia el
' índice seleccionado del ListBox
Private Sub lstValores_SelectedIndexChanged(ByVal sender As System.Object, ByVal e
As System.EventArgs) Handles lstValores.SelectedIndexChanged

    ' mostrar en el título del formulario el valor
    ' actualmente seleccionado de la lista
    Me.Text = TITULO & Me.lstValores.SelectedItem

End Sub
```

Código fuente 452

A través de los RadioButton, cambiamos el tipo de selección que podemos efectuar en el control lstValores. Ver Código fuente 453.

```
Private Sub rbtUno_CheckedChanged(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles rbtUno.CheckedChanged

    ' establecer tipo de selección en el ListBox a un elemento
    Me.lstValores.SelectionMode = SelectionMode.One

End Sub
```

```
Private Sub rbtMultiple_CheckedChanged(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles rbtMultiple.CheckedChanged

    ' establecer tipo de selección en el ListBox a un múltiples
    ' elementos
    Me.lstValores.SelectionMode = SelectionMode.MultiSimple

End Sub

Private Sub rbtExtendida_CheckedChanged(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles rbtExtendida.CheckedChanged

    ' establecer tipo de selección en el ListBox a múltiples
    ' elementos de modo extendido
    Me.lstValores.SelectionMode = SelectionMode.MultiExtended

End Sub
```

Código fuente 453

Mediante los controles chkOrdenar y chkColumnas, ordenaremos y mostraremos en columnas respectivamente el ListBox. Ver Código fuente 454.

```
Private Sub chkOrdenar_CheckedChanged(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles chkOrdenar.CheckedChanged

    ' según el valor del CheckBox, ordenamos o quitamos
    ' la opción de ordenar del ListBox
    Me.lstValores.Sorted = Me.chkOrdenar.Checked

End Sub

Private Sub chkColumnas_CheckedChanged(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles chkColumnas.CheckedChanged

    ' según el valor del CheckBox, mostramos el ListBox
    ' en varias columnas o en una
    Me.lstValores.MultiColumn = Me.chkColumnas.Checked

End Sub
```

Código fuente 454

El TextBox de este formulario lo usaremos para añadir nuevos elementos al ListBox lstValores, y buscar también elementos existentes, pulsando los botones btnAgregar y btnBuscar en cada caso. Observemos el miembro NoMatches del ListBox, mediante el que averiguamos si la búsqueda tuvo éxito. Ver el Código fuente 455.

```
Private Sub btnAgregar_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnAgregar.Click

    ' añadimos el contenido del TextBox como
    ' un elemento a la lista
    Me.lstValores.Items.Add(Me.txtValor.Text)

End Sub
```

```
Private Sub btnBuscar_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnBuscar.Click

    Dim iPosicion As Integer
    ' el método FindString() de la lista busca un valor
    iPosicion = Me.lstValores.FindString(Me.txtValor.Text)

    ' el campo NoMatches indica si no existe el valor buscado
    If iPosicion = Me.lstValores.NoMatches Then
        MessageBox.Show("No existe el valor")
    Else
        ' si encontramos el valor en la lista,
        ' lo seleccionamos por código
        Me.lstValores.SelectedIndex = iPosicion
    End If

End Sub
```

Código fuente 455

La selección de los elementos de un ListBox no es competencia exclusiva del usuario. El programador puede también, si lo necesita, seleccionar valores de la lista mediante el código del programa. Al pulsar el botón btnSelecCod, utilizaremos el método SetSelected( ) del ListBox para realizar esta tarea. En este método pasamos como parámetro el índice de la lista con el que vamos a operar, y el valor True para seleccionarlo, o False para quitarle la selección. Ver el Código fuente 456.

```
Private Sub btnSelecCod_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnSelecCod.Click

    ' para seleccionar elementos de un ListBox por código
    ' podemos utilizar el método SetSelected()
    Me.rbtMultiple.Checked = True
    Me.lstValores.SetSelected(1, True)
    Me.lstValores.SetSelected(3, True)
    Me.lstValores.SetSelected(5, True)

End Sub
```

Código fuente 456

El botón btnTraspasarSelec lo usaremos para tomar los elementos seleccionados de lstValores, y pasarlos al otro ListBox del formulario. La propiedad SelectedItems del control lstValores, devuelve una colección con sus elementos seleccionados. Por otra parte, podemos eliminar los elementos de un ListBox llamando al método Clear( ) de la colección de valores del control, cosa que hacemos pulsando el botón btnLimpiar. Ver Código fuente 457.

```
Private Sub btnTrapasarSelec_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnTrapasarSelec.Click

    Dim oSeleccion As ListBox.SelectedObjectCollection
    ' obtenemos con SelectedItems los elementos seleccionados
    ' de un ListBox
    oSeleccion = Me.lstValores.SelectedItems

    ' si existen elementos seleccionados,
    ' los traspasamos a otro ListBox del formulario
    If oSeleccion.Count > 0 Then
```

```
        Dim oEnumerador As IEnumerator
        oEnumerador = oSeleccion.GetEnumerator()
        While oEnumerador.MoveNext()
            Me.lstTraspaso.Items.Add(oEnumerador.Current)
        End While
    End If
End Sub

Private Sub btnLimpiar_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnLimpiar.Click

    ' con el método Clear() de la colección de elementos
    ' de un ListBox, borramos los elementos del controls
    Me.lstTraspaso.Items.Clear()

End Sub
```

Código fuente 457

ComboBox

El ComboBox es un control basado en la combinación (de ahí su nombre) de dos controles que ya hemos tratado: TextBox y ListBox.

Un control ComboBox dispone de una zona de edición de texto y una lista de valores, que podemos desplegar desde el cuadro de edición.

El estilo de visualización por defecto de este control, muestra el cuadro de texto y la lista oculta, aunque mediante la propiedad DropDownStyle podemos cambiar dicho estilo. La Figura 255 muestra un formulario con diversos ComboBox, cada uno con diferente estilo.

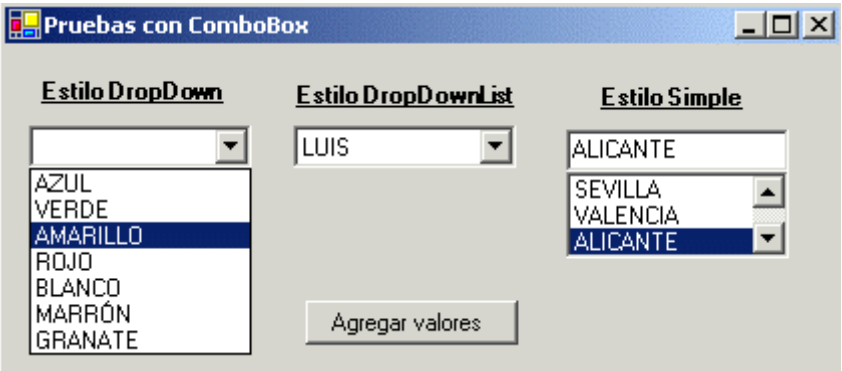


Figura 255. Controles ComboBox de distintos estilos.

La propiedad DropDownStyle también influye en una diferencia importante de comportamiento entre el estilo DropDownList y los demás, dado que cuando creamos un ComboBox con el mencionado estilo, el cuadro de texto sólo podrá mostrar información, no permitiendo que esta sea modificada.

En el caso de que la lista desplegable sea muy grande, mediante la propiedad MaxDropDownItems, asignaremos el número de elementos máximo que mostrará la lista del control.

El resto de propiedades y métodos son comunes con los controles TextBox y ListBox. En el Código fuente 458 se muestra el código del botón btnAgregar, mediante el que llenamos de valores los controles de este ejemplo.

```
Private Sub btnAgregar_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnAgregar.Click

    Me.cboColores.Items.AddRange(New String() {"AZUL", "VERDE", "AMARILLO", "ROJO",
"BLANCO", "MARRÓN", "GRANATE"})

    Me.cboNombres.Items.AddRange(New String() {"ELENA", "JOSE", "ANA", "ALFREDO",
"LUIS", "ANGEL", "RAQUEL"})


    Me.cboCiudades.Items.AddRange(New String() {"SEVILLA", "VALENCIA", "ALICANTE",
"TOLEDO", "SEGOVIA"})

End Sub
```

Código fuente 458.

30

Codificación avanzada de controles y herencia visual

Compartiendo código entre controles

En versiones anteriores de Visual Basic, podíamos crear un array de controles estableciendo en la propiedad Index del control, el número correspondiente a la posición de dicho control dentro del array. Como restricción, el array debía estar compuesto por controles del mismo tipo.

Una característica de los arrays de controles era que el código de los eventos era compartido entre todos los controles. El procedimiento de evento correspondiente, recibía un número que identificaba el control del array que había provocado dicho evento. De esta forma, podíamos tener código común para ejecutar sobre cualquiera de los controles del array, y código particular, sólo para ciertos controles.

El Código fuente 459 muestra el evento Click de un array de controles CommandButton en VB6.

```
' código VB6
' ==========
Private Sub cmdPulsar_Click(Index As Integer)

' utilizamos una estructura Select Case para comprobar qué
' control ha provocado el evento

Select Case Index
    Case 0
```

```
        ' código que se ejecutará cuando
        ' pulsemos el control de la posición 0
        ' del array
        '....

    Case 1
        ' código que se ejecutará cuando
        ' pulsemos el control de la posición 1
        ' del array
        '....

    '....
End Select

' código general que se ejecutará sobre cualquier control del array
'....
'....

End Sub
```

Código fuente 459

Los arrays de controles no están soportados por VB.NET, ya que existe un medio mucho más potente y flexible de escribir código común para un conjunto de controles: la creación de manejadores de evento comunes.

En versiones anteriores de VB, el nombre del procedimiento que manipulaba un determinado evento de un control era algo riguroso que no podía ser cambiado.

Pero como hemos visto anteriormente, VB.NET supera esta limitación, permitiéndonos dar el nombre que queramos a un procedimiento manipulador del evento de un control, y asociando dicho procedimiento al evento mediante la palabra clave Handles.

Handles encierra una potencia mayor de la que en un principio pudiera parecer, ya que si disponemos de un formulario con varios controles, y escribimos un procedimiento manipulador de evento, podemos asociar dicho procedimiento a los eventos de más de un control del formulario al mismo tiempo, basta con escribir a continuación de Handles, los nombres de objeto-evento separados por comas. El resultado será que cada vez que se produzca en esos controles el evento en cuestión, se llamará al mismo procedimiento de evento; y dentro de ese código, deberemos discernir cuál control originó el evento.

Para demostrar cómo enfocar desde VB.NET, el escenario descrito al comienzo de este apartado, que estaba escrito en VB6, crearemos un proyecto con el nombre CompartirEventos (hacer clic aquí para acceder a este ejemplo), y en su formulario, insertaremos tres Button. Ver Figura 256.

El trabajo a desempeñar consiste en que al pulsar cualquiera de los botones, la cadena de caracteres de su propiedad Text sea convertida a mayúscula.

Podríamos hacer doble clic en cada uno de estos controles y realizar dicha operación. Sin embargo, en esta ocasión, escribiremos un único procedimiento manipulador para el evento Click de estos controles, ya que como la acción a realizar es idéntica para todos los controles, ahorraremos una importante cantidad de tiempo y código. Ver el Código fuente 460.

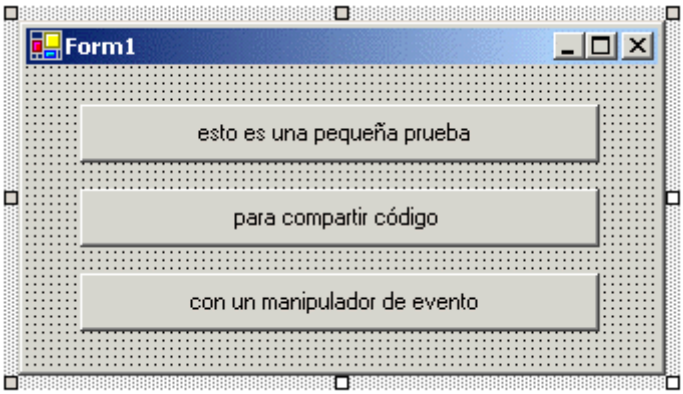


Figura 256. Formulario con controles que tendrán un manipulador de evento común.

```
Private Sub Pulsar(ByVal sender As Object, ByVal e As EventArgs) Handles
btnUno.Click, btnDos.Click, btnTres.Click

    ' antes de convertir a mayúsculas, debemos realizar
    ' un moldeado de tipo con CType() del parámetro que contiene
    ' el objeto que provocó el evento
    CType(sender, Button).Text = CType(sender, Button).Text.ToUpper()

End Sub
```

Código fuente 460

Complicando un poco más la situación, puede ocurrir que para este evento, tengamos que realizar tareas comunes y otras particulares para cada control; por ejemplo, poner a cada botón un color de fondo diferente. Pues no existe problema en ese sentido, ya que el parámetro sender del manipulador de evento, nos va a informar de cuál de los controles ha sido pulsado. El Código fuente 461 muestra, en ese sentido, una ampliación del código del evento.

```
Private Sub Pulsar(ByVal sender As Object, ByVal e As EventArgs) Handles
btnUno.Click, btnDos.Click, btnTres.Click
    ' antes de convertir a mayúsculas, debemos realizar
    ' un moldeado de tipo con CType() del parámetro que contiene
    ' el objeto que provocó el evento
    CType(sender, Button).Text = CType(sender, Button).Text.ToUpper()
    ' comprobar cuál botón ha sido pulsado,
    ' y en función de esto, dar un color
    ' distinto a cada control
    If sender Is Me.btnUno Then
        Me.btnUno.BackColor = Color.BurlyWood
    End If
    If sender Is Me.btnDos Then
        Me.btnDos.BackColor = Color.Cornsilk
    End If
    If sender Is Me.btnTres Then
        Me.btnTres.BackColor = Color.HotPink
    End If
End Sub
```

Código fuente 461

La Figura 257 muestra la ejecución del formulario tras estos cambios.

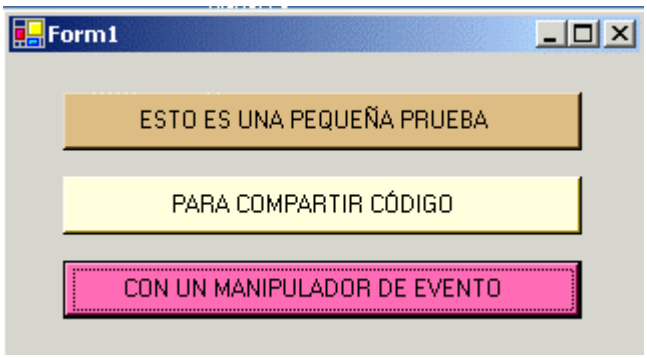


Figura 257. Controles con manipulador de evento Clic único.

No sólo es posible escribir un manipulador de evento para varios controles del mismo tipo, sino que también podemos establecer esta asociación entre controles de distinto tipo, naturalmente, siempre y cuando todos esos controles dispongan de dicho evento común.

En el siguiente ejemplo, EventoVariosCtl (hacer clic aquí para acceder a este ejemplo), creamos un formulario con tres controles de diferente tipo. Seguidamente escribimos en el código de la clase del formulario, un método con el nombre ControlPulsado( ), que asociamos con Handles, al evento Click de cada uno de estos controles, tal y como muestra el Código fuente 462.

```
Public Class Form1
    Inherits System.Windows.Forms.Form
    '....
    '....

    ' este procedimiento de evento lo asignamos al evento
    ' click de distintos tipos de control en el formulario
    ' mediante la palabra clave Handles
    Private Sub ControlPulsado(ByVal sender As Object, ByVal e As EventArgs)
Handles btnPulsar.Click, txtNombre.Click, lblNombre.Click
        ' comprobar sobre cuál control se ha hecho click

        ' si es el Label, cambiar estilo borde
        If sender Is Me.lblNombre Then
            Me.lblNombre.BorderStyle = System.Windows.Forms.BorderStyle.Fixed3D
        End If

        ' si es el Button, cerrar el formulario
        If sender Is Me.btnPulsar Then
            Me.Close()
        End If

        ' si es el TextBox, cambiar su color
        If sender Is Me.txtNombre Then
            Me.txtNombre.BackColor = Color.LightSeaGreen
        End If
    End Sub

End Class
```

Código fuente 462

La Figura 258 muestra la aplicación en funcionamiento

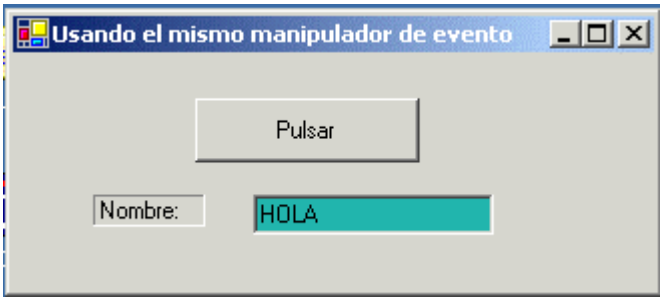


Figura 258. Controles diferentes utilizando el mismo manipulador para el evento Click.

En cualquier caso, si el programador necesita arrays de controles en sus programas, puede utilizar cualquiera de los tipos de la plataforma para esta finalidad, desde un array simple, hasta alguno de los diferentes tipos de colección que nos proporciona el entorno de ejecución de .NET Framework.

Creación de controles desde código

Al igual que hemos visto la posibilidad de crear un formulario sólo con código, sin utilizar su diseñador, es también posible la creación de los controles que componen el formulario, así como la definición de los manipuladores de evento, tanto para controles como para formulario.

Como ejemplo, crearemos un proyecto con el nombre ControlCodigo (hacer clic aquí para acceder al ejemplo), y en él escribiremos el código para crear, tanto el interfaz de usuario como los eventos que necesitemos detectar, para formulario y controles.

Código para el interfaz de usuario

En esta aplicación mostraremos un TextBox, del que contaremos la cantidad de texto que escribe el usuario, visualizándola en un control Label. Mediante dos RadioButton seleccionaremos el tipo de información a mostrar: fecha u hora del sistema, que visualizaremos al pulsar un Button. Finalmente, otro Button cerrará el formulario.

Una vez creado el proyecto, eliminaremos el formulario que incluye por defecto, y añadiremos una nueva clase al proyecto, a la que daremos como nombre frmDatos. En esta clase declararemos las diferentes variables que van a contener los controles, mientras que en el método constructor, instanciamos los controles, configurándolos para mostrarlos en el formulario. Ver Código fuente 463.

```
Public Class frmDatos
    Inherits Windows.Forms.Form

    Private WithEvents txtInfo As TextBox
    Private lblContador As Label
    Private btnMostrar As Button
    Private btnCerrar As Button
    Private rbtFecha As RadioButton
    Private rbtHora As RadioButton

    Public Sub New()
        MyBase.New()
```

```
        ' TextBox
        Me.txtInfo = New TextBox()
        Me.txtInfo.Location = New Point(20, 24)
        Me.txtInfo.Name = "txtInfo"
        Me.txtInfo.Size = New Size(132, 20)
        Me.txtInfo.TabIndex = 0
        Me.txtInfo.Text = ""

        ' Label
        Me.lblContador = New Label()
        Me.lblContador.Location = New Point(175, 28)
        Me.lblContador.Name = "lblContador"
        Me.lblContador.Size = New Size(55, 16)
        Me.lblContador.TabIndex = 1

        ' Button: Mostrar
        Me.btnMostrar = New Button()
        Me.btnMostrar.Location = New Point(20, 66)
        Me.btnMostrar.Name = "btnMostrar"
        Me.btnMostrar.Size = New Size(103, 23)
        Me.btnMostrar.TabIndex = 2
        Me.btnMostrar.Text = "&Mostrar"

        ' Button: Cerrar
        Me.btnCerrar = New Button()
        Me.btnCerrar.Location = New Point(20, 100)
        Me.btnCerrar.Name = "btnCerrar"
        Me.btnCerrar.Size = New Size(103, 23)
        Me.btnCerrar.TabIndex = 3
        Me.btnCerrar.Text = "&Cerrar"

        ' RadioButton: Fecha
        Me.rbtFecha = New RadioButton()
        Me.rbtFecha.Location = New Point(160, 66)
        Me.rbtFecha.Name = "rbtFecha"
        Me.rbtFecha.Size = New Size(100, 23)
        Me.rbtFecha.TabIndex = 4
        Me.rbtFecha.Text = "&Fecha"

        ' RadioButton: Hora
        Me.rbtHora = New RadioButton()
        Me.rbtHora.Location = New Point(160, 95)
        Me.rbtHora.Name = "rbtHora"
        Me.rbtHora.Size = New Size(100, 23)
        Me.rbtHora.TabIndex = 5
        Me.rbtHora.Text = "&Hora"

        ' Form
        Me.Controls.AddRange(New Control() {Me.txtInfo, Me.lblContador,
Me.btnMostrar, Me.btnCerrar, Me.rbtFecha, Me.rbtHora})
        Me.ClientSize = New Size(292, 140)
        Me.Name = "frmDatos"
        Me.Text = "Creación de controles desde código"
        Me.FormBorderStyle = FormBorderStyle.Fixed3D

    End Sub
End Class
```

Código fuente 463

A continuación, añadimos una nueva clase al proyecto con el nombre Inicio, que utilizaremos para iniciar la aplicación. Al tratarse de una clase, necesitamos escribir un método Main( ) compartido, que instancie un objeto formulario, como muestra el Código fuente 464.

```
Public Class Inicio
    Public Shared Sub Main()
        Application.Run(New frmDatos())
    End Sub
End Class
```

Código fuente 464

La Figura 259 muestra en ejecución nuestro formulario y controles creados exclusivamente con código.

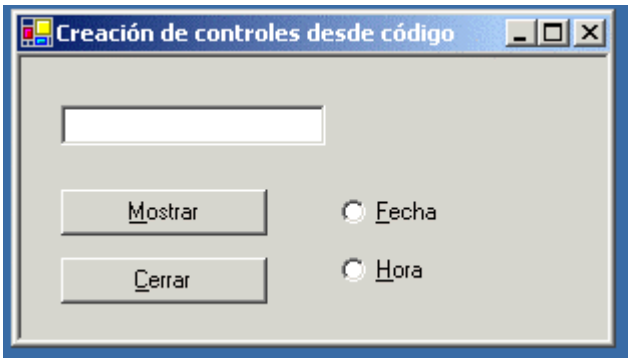


Figura 259. Formulario y controles creados con código en tiempo de ejecución.

Código para eventos del formulario, conectando con Handles

Para detectar uno de los eventos del formulario, por ejemplo MouseMove, que se produce en cada movimiento del ratón por la superficie del formulario, escribiremos el Código fuente 465, que como ya hemos visto en otras ocasiones, se trata de un procedimiento al que, mediante la palabra clave Handles, asociamos con un determinado evento.

```
' este manipulador de evento se produce cuando se mueve
' el ratón por el formulario
Private Sub PosicionRaton(ByVal sender As Object, ByVal e As MouseEventArgs)
Handles MyBase.MouseMove

    ' mostramos las coordenadas del ratón
    ' utilizando el parámetro con los argumentos
    ' adicionales del evento
    Me.Text = "Coordenadas ratón: X:" & e.X & " - Y:" & e.Y

End Sub
```

Código fuente 465

Código para eventos de controles, conectando con Handles

De la misma manera que con el formulario, podemos asociar mediante Handles, un procedimiento con un evento.

En este caso, y ya que hemos declarado el control txtInfo usando la palabra clave WithEvents, escribiremos el Código fuente 466, en el que cada vez que cambie el contenido de txtInfo, se mostrará la longitud de su texto en el control Label del formulario.

```
' este manipulador de evento se produce cada vez que cambie
' el contenido de este TextBox
Private Sub TextoCambiado(ByVal sender As Object, ByVal e As EventArgs) Handles
txtInfo.TextChanged

    ' mostramos la longitud de caracteres en el Label del formulario
    Me.lblContador.Text = Me.txtInfo.Text.Length

End Sub
```

Código fuente 466

Código para eventos de controles, conectando con AddHandler

En el tema dedicado a los eventos en .NET describimos el modo de asociar en tiempo de ejecución un procedimiento manipulador de evento con un determinado evento. Pues bien, esto naturalmente, podemos hacerlo también en el contexto de aplicaciones con interfaz Windows.

Como caso ilustrativo, el control Button btnCerrar no ha sido declarado con WithEvents, por lo tanto, no podemos asociar un manipulador de evento a dicho control mediante Handles como ocurre con el TextBox de nuestro formulario.

Sin embargo, vamos a conectar el evento Click de ese control Button con un procedimiento empleando la palabra clave AddHandler, pasando la dirección del procedimiento a asociar con AddressOf; como muestra el Código fuente 467.

```
Public Class frmDatos
    '....
    Public Sub New()
        '....
        ' Button: Cerrar
        Me.btnCerrar = New Button()
        '....
        ' asociar el botón Cerrar a un procedimiento
        ' de evento con AddHandler
        AddHandler btnCerrar.Click, AddressOf CerrarVentana
        '....
    End Sub

    ' este manipulador de evento se producirá cuando
    ' se pulse el botón btnCerrar
    Private Sub CerrarVentana(ByVal sender As Object, ByVal e As EventArgs)
        Me.Close()
    End Sub
    '....
End Class
```

Código fuente 467

Código para eventos de controles, asociando y separando dinámicamente con AddHandler y RemoveHandler

Debido a que AddHandler emplea enlace tardío para conectar un procedimiento con el evento de un objeto, podemos conseguir que un mismo evento de un control, en determinados momentos ejecute un procedimiento manipulador, y en otras ocasiones otro distinto.

Esto es lo que vamos a conseguir con los RadioButton del formulario. Cuando pulsemos el control rbtFecha, el evento Click del control btnMostrar ejecutará un manipulador de evento, y cuando pulsamos el control rbtHora, el botón btnMostrar ejecutará un manipulador distinto.

En primer lugar, debemos asociar los controles RadioButton con su propio evento Click. Esto lo conseguimos mediante AddHandler, en el momento de su creación.

Una vez que pulsemos alguno de los RadioButton, se ejecutará su correspondiente método Click, y en el procedimiento manipulador de ese evento del control de radio, asociaremos al botón btnMostrar un procedimiento para su evento de pulsación. Veamos el Código fuente 468.

```
Public Class frmDatos
    '....
    Public Sub New()
        '....
        ' RadioButton: Fecha
        Me.rbtFecha = New RadioButton()
        '....
        ' asociar este RadioButton a un procedimiento
        ' de evento con AddHandler
        AddHandler Me.rbtFecha.Click, AddressOf PulsaFecha

        ' RadioButton: Hora
        Me.rbtHora = New RadioButton()
        '....
        ' asociar este RadioButton a un procedimiento
        ' de evento con AddHandler
        AddHandler Me.rbtHora.Click, AddressOf PulsaHora
        '....
    End Sub
    '....
    '....
    ' este manipulador de evento se producirá cuando
    ' se pulse el RadioButton rbtFecha
    Private Sub PulsaFecha(ByVal sender As Object, ByVal e As EventArgs)
        ' asociamos un nuevo manipulador para el botón btnMostrar
        AddHandler btnMostrar.Click, AddressOf MuestraFecha
    End Sub

    ' este manipulador de evento se producirá cuando
    ' se pulse el RadioButton rbtHora
    Private Sub PulsaHora(ByVal sender As Object, ByVal e As EventArgs)
        ' asociamos un nuevo manipulador para el botón btnMostrar
        AddHandler btnMostrar.Click, AddressOf MuestraHora
    End Sub

    ' este manipulador de evento se asociará al control btnMostrar
    ' cuando se pulse el RadioButton rbtFecha
    Private Sub MuestraFecha(ByVal sender As Object, ByVal e As EventArgs)
        Dim dtFecha As Date
        dtFecha = DateTime.Today
        MessageBox.Show("Fecha actual: " & dtFecha.ToString("D"))
    End Sub
```

```
    ' este manipulador de evento se asociará al control btnMostrar
    ' cuando se pulse el RadioButton rbtHora
    Private Sub MuestraHora(ByVal sender As Object, ByVal e As EventArgs)
        Dim dtFecha As Date
        dtFecha = DateTime.Now
        MessageBox.Show("Hora actual: " & dtFecha.ToString("T"))
    End Sub

End Class
```

Código fuente 468

Sin embargo, este modo de asignación del procedimiento manipulador, al evento Click del botón btnMostrar, tiene la siguiente pega: cada vez que pulsamos uno de los RadioButton, el manipulador del evento Click antiguo no se elimina, sino que se va apilando a los ya existentes.

Como consecuencia, cuando hayamos pulsado repetidas veces los controles de radio del formulario, se ejecutarán también repetidamente los manipuladores del evento Click de btnMostrar.

El motivo de este comportamiento se debe a que el delegado en el que está basado el evento, contiene lo que se denomina una *lista de invocación*, y cada vez que usamos AddHandler, se añade el nombre del procedimiento de evento a dicha lista. Si no quitamos de la lista de manipuladores de evento la referencia a un procedimiento, cada vez que se produzca el evento se ejecutarán todos los procedimientos de su lista de invocación. Para quitar un procedimiento de la lista de un evento, emplearemos la instrucción RemoveHandler.

Para que todo funcione ya correctamente, cuando pulsemos los RadioButton, en el código de los eventos de estos controles haremos un pequeño añadido, consistente en quitar al botón btnMostrar, el manipulador de evento que tenía hasta ese momento. Veámoslo en el Código fuente 469.

```
' este manipulador de evento se producirá cuando
' se pulse el RadioButton rbtFecha
Private Sub PulsaFecha(ByVal sender As Object, ByVal e As EventArgs)

    ' quitamos el manipulador que hubiera para btnMostrar...
    RemoveHandler btnMostrar.Click, AddressOf MuestraHora

    ' ...y asociamos un nuevo manipulador para el botón btnMostrar
    AddHandler btnMostrar.Click, AddressOf MuestraFecha

End Sub

' este manipulador de evento se producirá cuando
' se pulse el RadioButton rbtHora
Private Sub PulsaHora(ByVal sender As Object, ByVal e As EventArgs)

    ' quitamos el manipulador que hubiera para btnMostrar...
    RemoveHandler btnMostrar.Click, AddressOf MuestraFecha

    ' ...y asociamos un nuevo manipulador para el botón btnMostrar
    AddHandler btnMostrar.Click, AddressOf MuestraHora

End Sub
```

Código fuente 469

Al ejecutar ahora, cuando pulsemos el control btnMostrar, sólo se ejecutará un único procedimiento manipulador de su evento Click. Esto nos proporciona de nuevo una idea de la potencia que encierra el lenguaje en esta versión de VB.

Recorriendo los controles de un formulario

La propiedad Controls de la clase Form, devuelve un tipo ControlCollection, que como indica su nombre, consiste en una colección que contiene los controles del formulario. El modo de uso de esta colección es igual que el ya explicado en los temas del texto que trataban sobre colecciones.

El ejemplo ColecControles (hacer clic aquí para obtener el ejemplo), consiste en un proyecto con el formulario de la Figura 260.

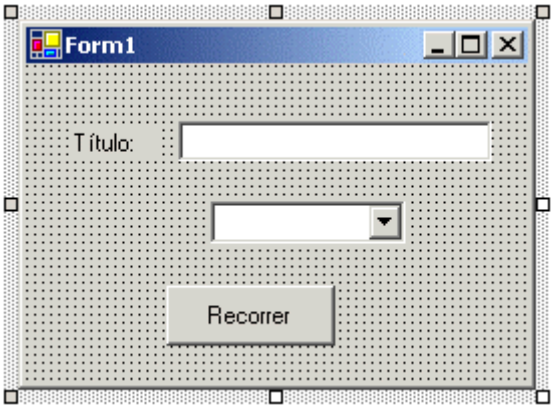


Figura 260. Formulario con los controles a recorrer.

Al pulsar el botón Recorrer de este formulario, ejecutaremos el Código fuente 470, en el que podemos ver cómo recorrer su colección de controles, realizando cambios en alguno de ellos.

```
Private Sub btnRecorrer_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnRecorrer.Click

    Dim oListaControles As ControlCollection
    ' obtener una colección con los
    ' controles del formulario
    oListaControles = Me.Controls

    Dim oUnControl As Control
    ' recorrer la colección de controles
    For Each oUnControl In oListaControles
        MessageBox.Show("El control actual es: " & oUnControl.Name)

        If oUnControl Is Me.txtDatos Then
            ' actuar directamente sobre el
            ' control; añadir texto
            Me.txtDatos.Text = "PROBANDO EL CONTROL"
        End If

        If oUnControl Is Me.cboNombres Then
            ' hacer un moldeado de tipo de la
            ' variable que usamos para recorrer la
            ' colección, conviertiendo al tipo de
            ' control adecuado; añadir elementos a la lista
```

```
            CType(oUnControl, ComboBox).Items.AddRange(New String() {"JUAN",
"LUIS", "MARTA", "ANA"})
        End If
    Next

End Sub
```

Código fuente 470

Temporizadores

En VB.NET disponemos al igual que en anteriores versiones, del control Timer, que nos permite la ejecución de código a intervalos de tiempo predeterminados.

Este control ha sufrido una importante reestructuración, ya que internamente hace uso de la clase Timer, perteneciente al conjunto de clases del sistema. El hecho de poder acceder a esta clase, nos proporciona una gran flexibilidad en nuestros desarrollos, ya que, a partir de ahora también crearemos temporizadores por código, sin necesidad de utilizar el control Timer.

En el ejemplo TimerPru que comentamos a continuación, vamos a construir un formulario en el que utilizaremos ambos tipos de temporizadores, el propio control Timer y un objeto de la clase (hacer clic aquí para acceder a este ejemplo).

El primer proceso a codificar, consistirá en traspasar a intervalos de tiempo, el contenido de un TextBox del formulario, a otro control de este mismo tipo. El formulario del proyecto se muestra en la Figura 261.

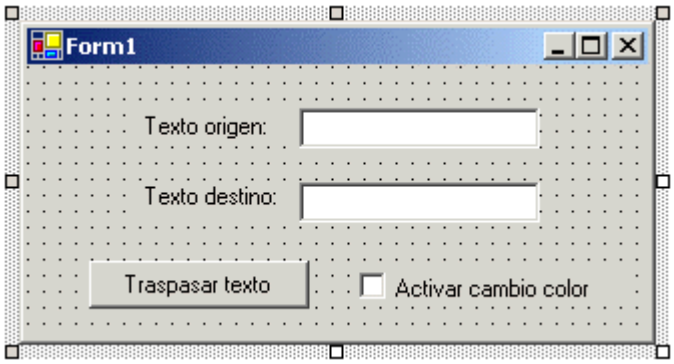


Figura 261. Formulario para ejemplo con temporizadores.

Tras incluir los controles de usuario en el formulario, añadiremos un control Timer, al que daremos el nombre tmrTemporizador. Esta acción abrirá, bajo el diseñador del formulario, un panel para controles especiales, como es el caso de Timer, en el que se mostrará dicho control. Ver Figura 262.

En este panel se depositan los controles del formulario que no tienen una interacción directa con el usuario, o cuyo diseño es diferente al de los controles habituales.

Para especificar el espacio de tiempo en el que este control será ejecutado cuando lo activemos, utilizaremos la propiedad Interval, a la que tenemos que asignar un valor numérico, que establece dicho tiempo en milisegundos. En nuestro caso, asignaremos 500, con lo que el control se ejecutará cada medio segundo.

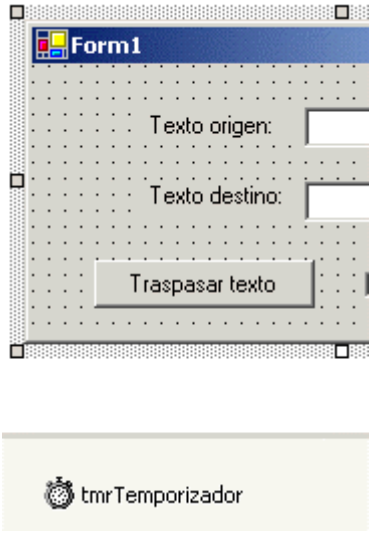


Figura 262. Panel para controles especiales del diseñador de formularios.

El control Timer lo activaremos llamando a su método Start( ), cosa que hacemos al pulsar el botón btnTraspasar. Ver Código fuente 471.

```
Private Sub btnTraspasar_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnTraspasar.Click

    ' iniciar el temporizador
    Me.tmrTemporizador.Start()

End Sub
```

Código fuente 471

Una vez activado un temporizador, cada vez que transcurre el tiempo indicado en Interval, genera un evento Tick. Es precisamente en este evento en el que debemos escribir el código que necesitamos que se ejecute a intervalos regulares de tiempo. Haremos, por consiguiente, doble clic en el control Timer del diseñador, para acceder al procedimiento manipulador de este evento, cuyo contenido lo podemos ver en el Código fuente 472.

```
' este evento se produce en el intervalo especificado
' en el control Timer
Private Sub tmrTemporizador_Tick(ByVal sender As Object, ByVal e As
System.EventArgs) Handles tmrTemporizador.Tick

    ' quitamos una letra del TextBox de origen...
    Dim sLetra As String
    sLetra = Me.txtOrigen.Text.Substring(0, 1)
    Me.txtOrigen.Text = Me.txtOrigen.Text.Remove(0, 1)

    ' ...y lo pasamos al TextBox de destino
    Me.txtDestino.Text &= sLetra

    ' cuando se haya traspaso todo el texto
    ' detener el temporizador
    If Me.txtOrigen.Text.Length = 0 Then
        Me.tmrTemporizador.Stop()
```

```
        MessageBox.Show("Traspaso finalizado")
    End If

End Sub
```

Código fuente 472

En cuanto a los temporizadores por código, vamos a crear un proceso en el que intercambiaremos los colores de fondo de los TextBox del formulario cada segundo.

En primer lugar, vamos a declarar una variable de tipo Timer en la clase del formulario, y otra variable Boolean. Ver Código fuente 473.

```
Public Class Form1
    Inherits System.Windows.Forms.Form

    ' temporizador por código
    Private oTiempo As Timer
    ' esta variable la utilizaremos para intercambiar
    ' los colores de fondo de los TextBox
    Private bIntercambio As Boolean
    '....
    '....
End Class
```

Código fuente 473

Al marcar el CheckBox del formulario, instanciaremos un objeto Timer. Asignaremos valores a sus propiedades, lo asociaremos a un procedimiento que manipule su evento Tick, y lo pondremos en marcha con Start( ). Como puede ver el lector, los métodos y propiedades son los mismos que para el control Timer. En el Código fuente 474 vemos el código del CheckBox y del manipulador del evento Tick.

```
Private Sub chkActivar_CheckedChanged(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles chkActivar.CheckedChanged

    ' cuando marquemos el CheckBox...
    If Me.chkActivar.Checked Then
        ' creamos el temporizador
        oTiempo = New Timer()
        oTiempo.Interval = 1000     ' se ejecutará cada segundo
        ' le asignamos un manipulador para el evento Tick
        AddHandler oTiempo.Tick, AddressOf CambioDeColor
        oTiempo.Start()     ' arrancamos el temporizador
    Else
        ' cuando desmarquemos el CheckBox paramos el temporizador
        oTiempo.Stop()
        oTiempo = Nothing
    End If

End Sub

' método manipulador del evento Tick del temporizador
' creado por código
Private Sub CambioDeColor(ByVal sender As Object, ByVal e As EventArgs)
    ' según el estado de la variable bIntercambio
    ' cada vez que se ejecute este método, se
```

```
    ' intercambiarán los colores de fondo de los TextBox
    bIntercambio = Not bIntercambio

    If bIntercambio Then
        Me.txtOrigen.BackColor = Color.Aquamarine
        Me.txtDestino.BackColor = Color.LightSteelBlue
    Else
        Me.txtOrigen.BackColor = Color.LightSteelBlue
        Me.txtDestino.BackColor = Color.Aquamarine
    End If

End Sub
```

Código fuente 474

Al ejecutar el ejemplo, podemos poner en marcha ambos temporizadores, comprobando así, como son ejecutados simultáneamente sin interferencia.

Crear una clase derivada de un control

De igual modo que creamos los formularios heredando de la clase Form, podemos crear clases que hereden de las clases pertenecientes a los controles Windows, adaptando el comportamiento del control a nuestras necesidades particulares, a través de la sobrecarga y sobre-escritura de los miembros existentes, o bien por la creación de nuevos métodos y propiedades.

Supongamos como ejemplo que necesitamos un control TextBox, enfocado fundamentalmente a mostrar números. A pesar de ello, nos debe permitir la escritura en él de otro tipo de caracteres, pero en ese caso, necesitamos que el control nos avise cuando su contenido no sea numérico. Por otro lado, no debe permitir la asignación de caracteres mediante su propiedad Text.

Crearemos pues un proyecto con el nombre ClaseControl (hacer clic aquí para acceder a este ejemplo). Además del formulario por defecto, agregaremos al proyecto una nueva clase, a la que daremos el nombre TextNumeros, y en ella codificaremos el comportamiento de nuestra propia versión del TextBox. El Código fuente 475 muestra esta clase.

```
Public Class TextNumeros
    Inherits TextBox

    ' declaramos un evento para poder generarlo
    ' cuando se asigne un valor al control que
    ' no sea numérico
    Public Event AsignacionNoNum(ByVal sender As system.Object, ByVal e As
EventArgs)

    Public Sub New()
        ' en el constructor dar un color al control
        Me.BackColor = Color.PaleGreen
    End Sub

    ' implementamos nuestra propia versión
    ' de la propiedad Text
    Public Overloads Overrides Property Text() As String
        Get
            Return MyBase.Text
        End Get
        Set(ByVal Value As String)
            ' eludimos la asignación por código
```

```
            ' de texto al control en el bloque Set del Property
        End Set
    End Property

    ' en este procedimiento de evento detectamos que
    ' cuando el contenido del control cambie...
    Private Sub TextNumeros_TextChanged(ByVal sender As Object, ByVal e As
System.EventArgs) Handles MyBase.TextChanged

        ' ...si el valor actual del control no es numérico,
        ' lanzar el evento
        If Not IsNumeric(Me.Text) Then
            RaiseEvent AsignacionNoNum(Me, New EventArgs())
        End If

    End Sub
End Class
```

Código fuente 475

A continuación pasaremos al formulario del proyecto, y en el código del mismo, declararemos a nivel de clase una variable del tipo correspondiente a nuestro control. Seguidamente, insertaremos un control Button para crear en tiempo de ejecución una instancia de nuestro control y mostrarlo en el formulario; al crear nuestro control, conectaremos con AddHandler, su evento AsignacionNoNum con un procedimiento del formulario que actúe como manipulador.

Finalmente, y para demostrar cómo no podemos asignar por código, valores a la propiedad Text del control, añadiremos otro botón adicional. Veamos todo ello en el Código fuente 476.

```
Public Class Form1
    Inherits System.Windows.Forms.Form

    Private txtNum As TextNumeros
    '....
    '....
    Private Sub btnCrear_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnCrear.Click
        ' creamos un objeto de nuestra clase control
        Me.txtNum = New TextNumeros()
        Me.txtNum.Location = New Point(60, 50)
        Me.txtNum.Name = "txtNum"
        Me.txtNum.Size = New Size(100, 40)
        Me.txtNum.TabIndex = 1

        ' asociamos el evento que hemos creado en el control
        ' con un procedimiento manejador de evento escrito
        ' en esta clase
        AddHandler txtNum.AsignacionNoNum, AddressOf ValorIncorrecto

        ' añadimos el control a la colección
        ' de controles del formulario
        Me.Controls.Add(Me.txtNum)

    End Sub

    ' procedimiento manipulador del evento AsignacionNoNum de nuestro
    ' control; aquí mostramos un mensaje informativo cuando
    ' escribamos un valor que no sea numérico
    Private Sub ValorIncorrecto(ByVal sender As System.Object, ByVal e As
System.EventArgs)
```

```
        MessageBox.Show("En este control sólo se deben introducir números",
"Atención")

    End Sub

    ' al pulsar este botón intentamos asignar una cadena en la
    ' propiedad Text de nuestro control, pero debido al comportamiento
    ' programado en el control, no podremos realizar esta asignación
    Private Sub btnAsignar_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnAsignar.Click

        Me.txtNum.Text = "PRUEBA"

    End Sub
End Class
```

Código fuente 476

El formulario en ejecución, con nuestro control propio ya creado, lo muestra la Figura 263.

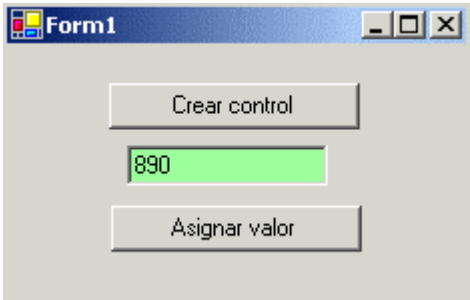


Figura 263. Formulario con un control heredado de TextBox.

Herencia visual

Además de la herencia habitual por código que hemos utilizado en los ejemplos de escritura de clases, los formularios Windows disponen de los mecanismos necesarios para crear un formulario base, a partir del cual, posteriormente podremos heredar en formularios derivados; todo ello de modo visual.

Vamos a desarrollar por lo tanto un ejemplo, en el que mostraremos los pasos necesarios a dar, tanto en la creación del formulario base como del heredado (hacer clic aquí, para obtener la solución de proyectos HerenciaVisual con el ejemplo).

La situación planteada en este ejemplo es la siguiente: necesitamos crear un formulario para identificar y validar a un usuario antes de permitirle el acceso a una aplicación. Como norma general, los datos mínimos que todo usuario debe teclear son su nombre (login) y contraseña (password); pero en algunos casos, dependiendo del programa a escribir, esta ventana de identificación puede requerir la introducción de datos adicionales, como un código adicional, una fecha, etc.

Dado que a priori, desconocemos los datos adicionales que podrán ser necesarios para este formulario, crearemos el formulario base incluyendo la introducción del login, password, y un botón para validar dichos datos, y posteriormente, en otro proyecto, heredaremos este formulario en uno derivado, al que añadiremos nuevos controles.

El formulario base

Comencemos por tanto, abriendo Visual Studio .NET, y creando un nuevo proyecto VB.NET, de tipo Windows, al que daremos el nombre HerenciaVisual.

A continuación, abriremos la ventana del explorador de soluciones, seleccionaremos el formulario por defecto de este proyecto y lo eliminaremos pulsando la tecla [SUPR].

Siguiendo en el explorador de soluciones, en esta ocasión haremos clic sobre el nombre del proyecto, y pulsaremos el botón de propiedades de esta ventana. En la ventana de propiedades, abriremos la lista desplegable *Tipo de resultado*, y seleccionaremos la opción *Biblioteca de clases*. Para poder utilizar un formulario como clase base de otro formulario, el proyecto que contiene el formulario base debe ser de tipo librería, para que al compilarlo, genere un fichero .DLL con dicho formato de biblioteca. Ver Figura 264.

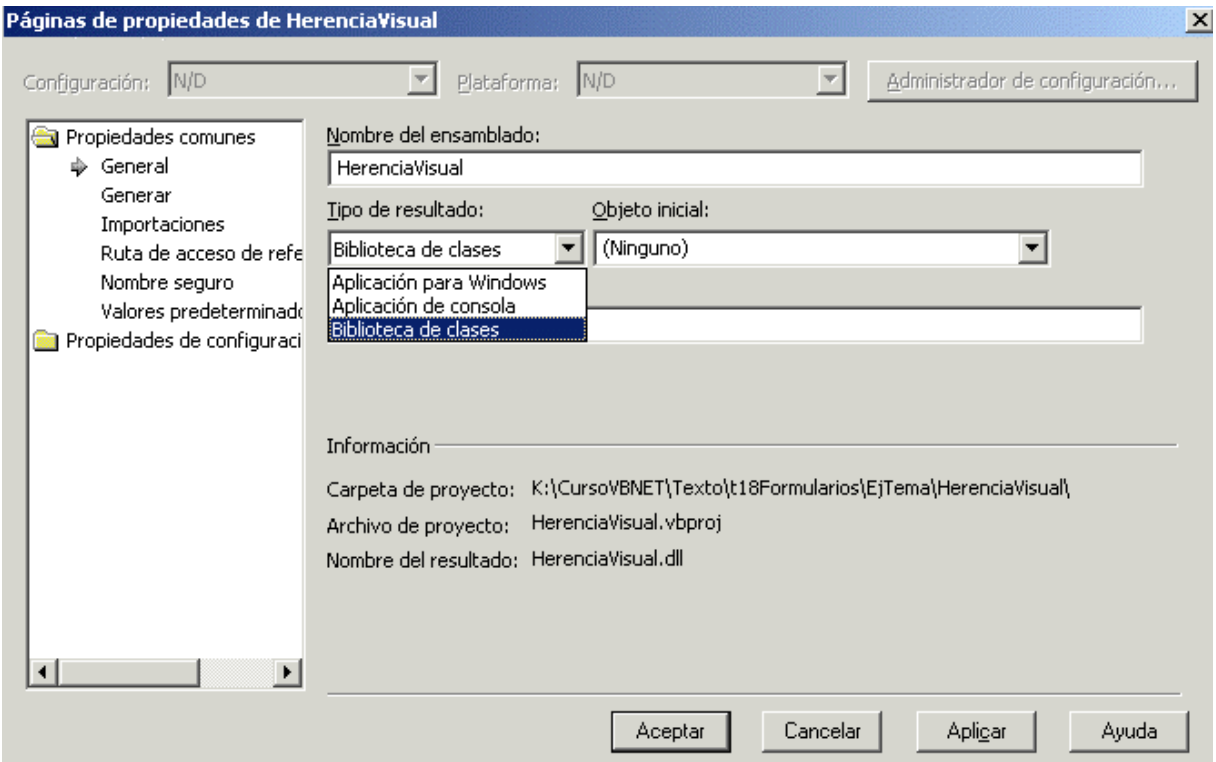


Figura 264. Estableciendo el tipo de proyecto a biblioteca de clases.

El siguiente paso consiste en añadir a este proyecto, el formulario que va a actuar como base. Para ello, seleccionaremos del IDE, la opción de menú *Proyecto + Agregar formulario de Windows*, y daremos el nombre frmValidar al nuevo formulario.

En el formulario frmValidar, insertaremos los controles necesarios para introducir los datos de login de usuario, su password, y un botón de validación de dichos datos. La Figura 265 muestra el aspecto de este formulario una vez concluido su diseño.

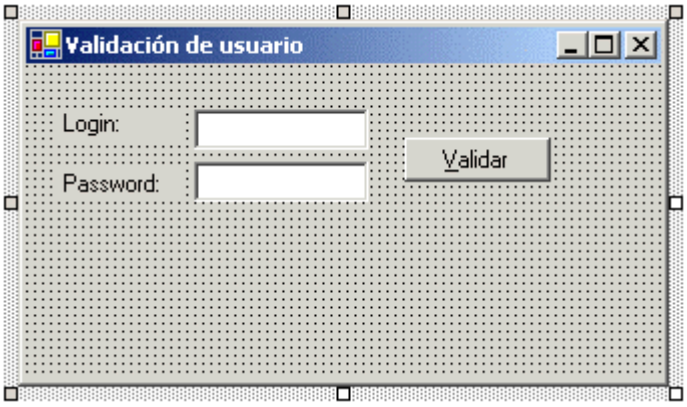


Figura 265. Formulario base para heredar.

Respecto al código de este formulario, escribiremos el correspondiente a la pulsación del Button que contiene. Ver Código fuente 477.

```
Private Sub btnValidar_Click(ByVal sender As Object, ByVal e As System.EventArgs)
Handles btnValidar.Click

    ' si el nombre de usuario es superior a 5 caracteres
    ' permitir acceso
    If Me.txtLogin.Text.Length > 5 Then
        MessageBox.Show("Bienvenido al sistema")
    End If

End Sub
```

Código fuente 477

Para finalizar con el formulario base, seleccionaremos en el IDE la opción de menú *Generar + Generar*, que creará la biblioteca de clases, es decir, el fichero HERENCIAVISUAL.DLL.

A partir de este punto, deberemos crear el proyecto que contenga un formulario que herede el formulario base que acabamos de crear. Esto lo podemos conseguir de dos modos: agregando un nuevo proyecto a la solución, o bien, creando un nuevo proyecto aparte. En ambos casos, tendremos que establecer la oportuna referencia, bien hacia el proyecto del formulario base en el primer caso, o hacia el archivo que contiene la librería de clases. Veamos ambos modos.

Agregar un proyecto con un formulario derivado

Mediante el menú del IDE *Archivo + Agregar proyecto + Nuevo proyecto*, añadiremos a la solución actual un nuevo proyecto de tipo aplicación Windows, con el nombre FormuDerivado1. Ello hará que nuestra solución de proyectos HerenciaVisual, quede como muestra la Figura 266, con el proyecto del formulario base y el nuevo.

Para poder realizar las oportunas pruebas, haremos clic derecho en el nuevo proyecto, FormuDerivado1, y seleccionaremos del menú contextual la opción *Establecer como proyecto de inicio*, ya que como una biblioteca de clases no podemos ejecutarla visualmente, necesitamos un proyecto que sea de tipo aplicación Windows, que sí disponen de medio visual de representación a través de formularios.

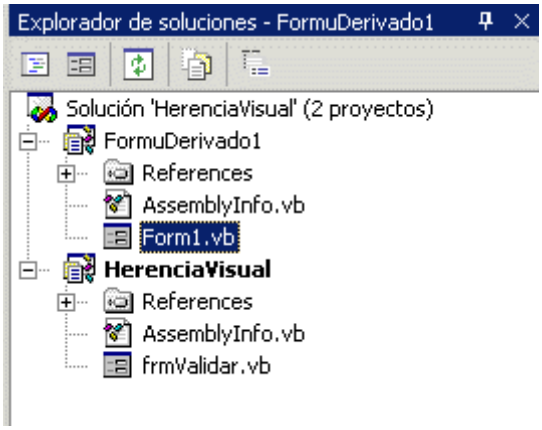


Figura 266. Solución con proyecto de formulario base y proyecto de formulario derivado.

El siguiente paso será eliminar el formulario que contiene el proyecto FormuDerivado1, de igual modo que el mostrado anteriormente.

Permaneciendo posicionados en el proyecto FormuDerivado1, seleccionaremos el menú *Proyecto + Agregar formulario heredado*, al que daremos el nombre frmAcceso. Ver Figura 267.

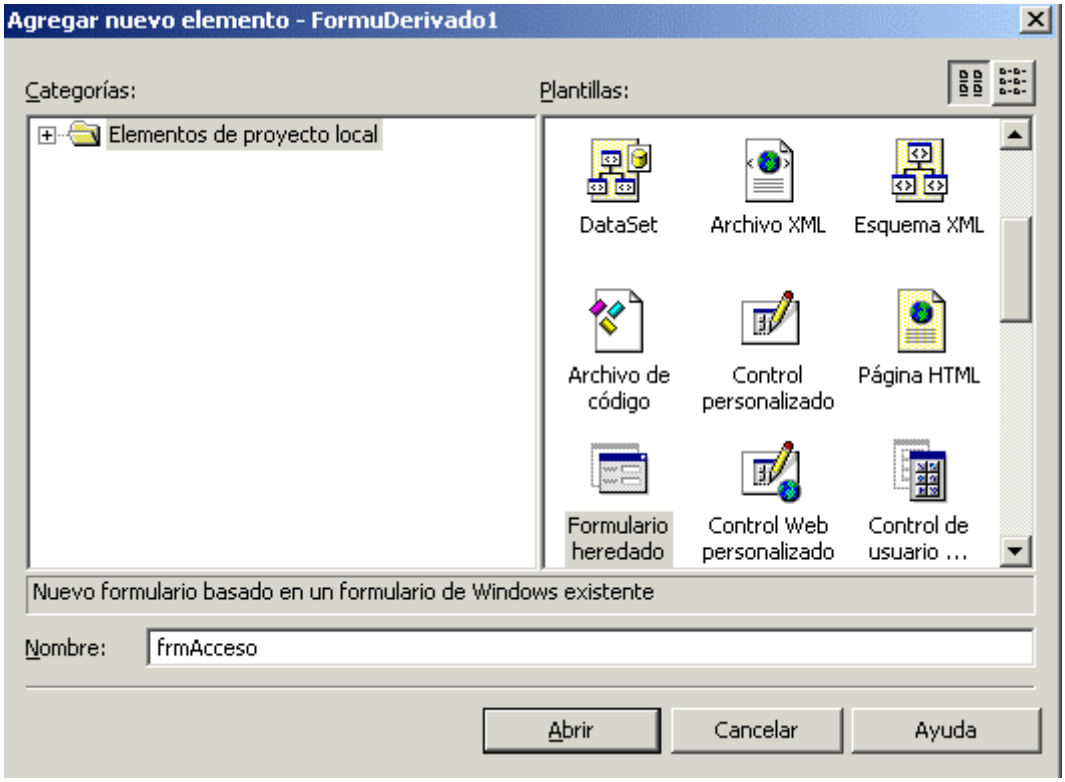


Figura 267. Agregar formulario heredado.

Al aceptar la ventana de creación del formulario, el entorno de desarrollo buscará la biblioteca de clases con el ámbito más próximo, y que disponga de formularios heredables, mostrando el resultado en la ventana *Selector de herencia*. En nuestro caso, naturalmente, aparecerá el formulario base frmValidar, contenido en la DLL que hemos generado anteriormente. Ver Figura 268.

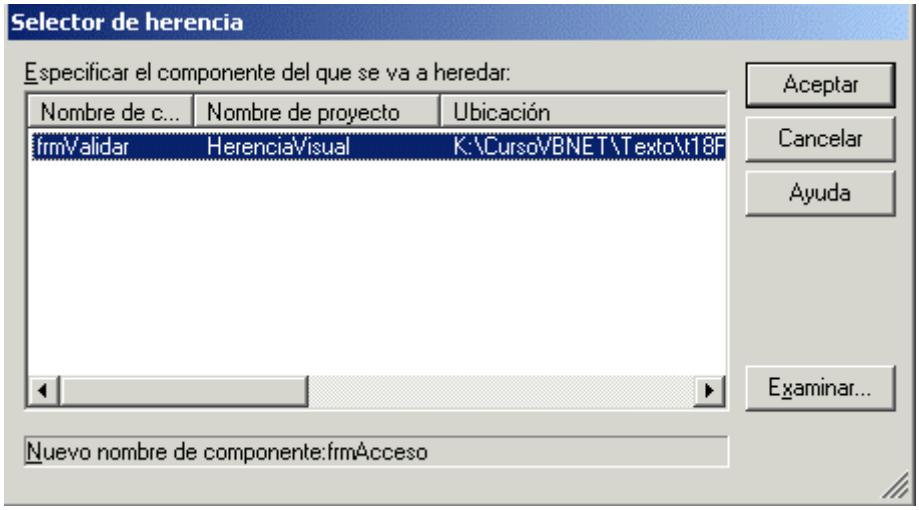


Figura 268. Seleccionar formulario base a partir del que se realizará la herencia.

Al pulsar Aceptar en esta ventana, será creado el nuevo formulario, en base al formulario base especificado, y establecida una referencia en el proyecto FormuDerivado1 hacia el proyecto base HerenciaVisual. Ver Figura 269.

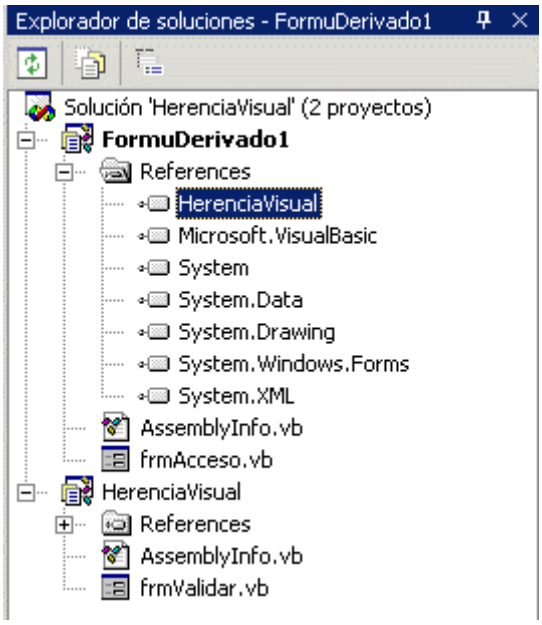


Figura 269. Proyecto con formulario heredado y con referencia hacia proyecto base.

Al abrir el formulario frmAcceso en el diseñador de formularios, se mostrará con los controles pertenecientes al formulario base bloqueados; ello es debido a que tales controles pertenecen a la clase base del formulario, y sólo pueden ser manipulados desde el proyecto del formulario base. En este formulario derivado añadiremos algunos controles más, quedando con el aspecto que muestra la Figura 270.

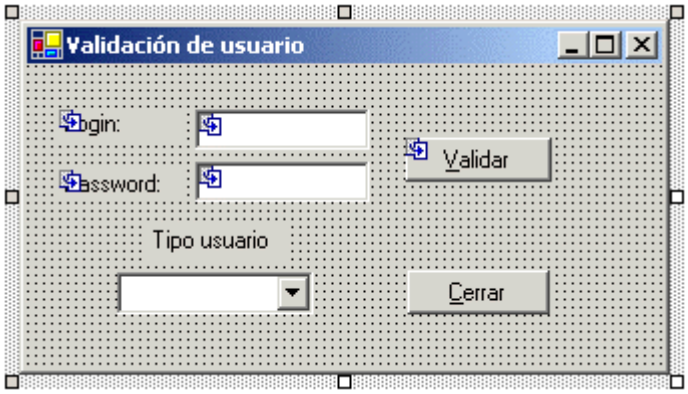


Figura 270. Formulario heredado mostrando controles base y nuevos controles.

En lo que respecta al código de este formulario, sólo podemos escribir los eventos de los nuevos controles, ya que el código de los controles heredados se encuentra protegido, siendo sólo modificable desde el proyecto del formulario base. El único evento, por lo tanto que vamos a escribir aquí, será el del botón btnCerrar, que llamará al método de cierre del formulario, como vemos en el Código fuente 478. Observemos también la declaración de clase, y como se establece la herencia con el formulario base.

```
Public Class frmAcceso
    Inherits HerenciaVisual.frmValidar
    '....
    '....
    Private Sub btnCerrar_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnCerrar.Click

        ' cerramos el formulario
        Me.Close()

    End Sub

End Class
```

Código fuente 478

Para poder ya finalmente, ejecutar este formulario derivado, debemos establecerlo como objeto inicial en las propiedades de su proyecto, ya que al crear su proyecto estaba configurado para que arrancar por Form1.

Crear un formulario heredado desde un proyecto independiente

La diferencia principal en este caso consiste en que dentro de la solución de proyectos, no está el proyecto que contiene el formulario base, por lo que tendremos que buscar el archivo que contiene la librería de clases manualmente.

Vamos a crear un nuevo proyecto de aplicación Windows, al que daremos el nombre FormuDerivado2 (hacer clic aquí para acceder a este ejemplo), eliminando el formulario por defecto que contiene. Después, seleccionaremos el nombre del proyecto en el explorador de soluciones.

A continuación, elegiremos el menú de VS.NET *Proyecto + Agregar formulario heredado*, asignando al formulario el nombre frmEntrada. Ver Figura 271.

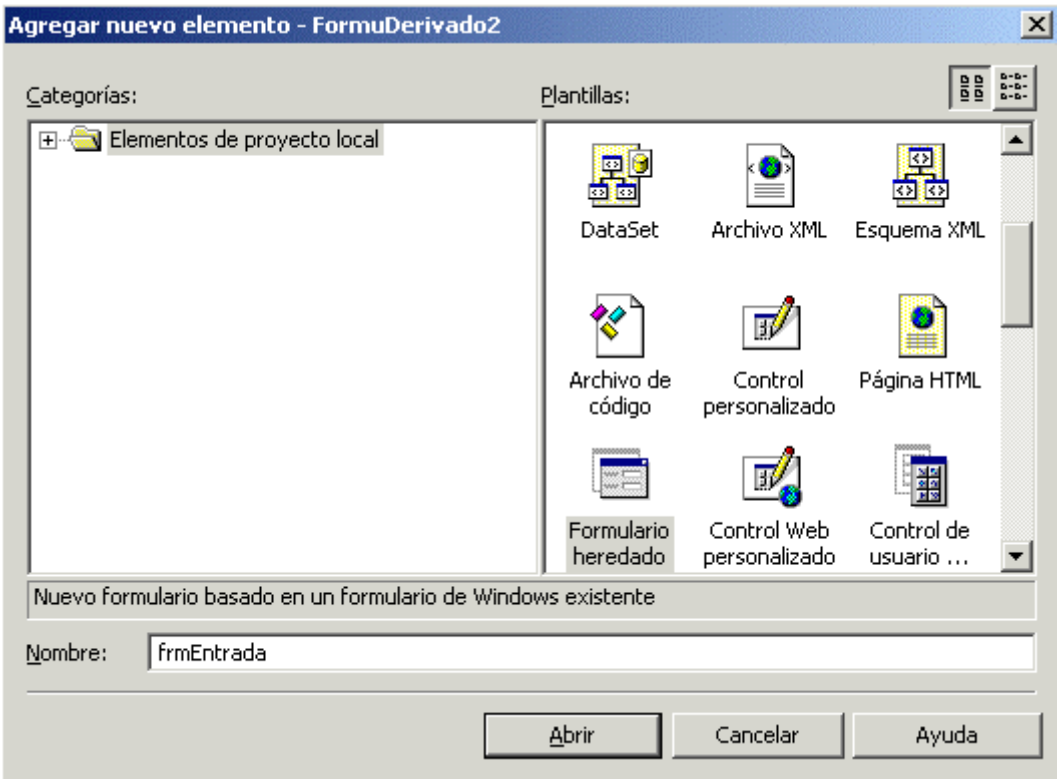


Figura 271. Crear formulario heredado en proyecto independiente.

Al pulsar Abrir, el selector de herencia nos avisa de que no puede localizar en el ámbito de que dispone, un ensamblado (biblioteca de clases en este caso) que contenga formularios de los que heredar. Ver Figura 272.

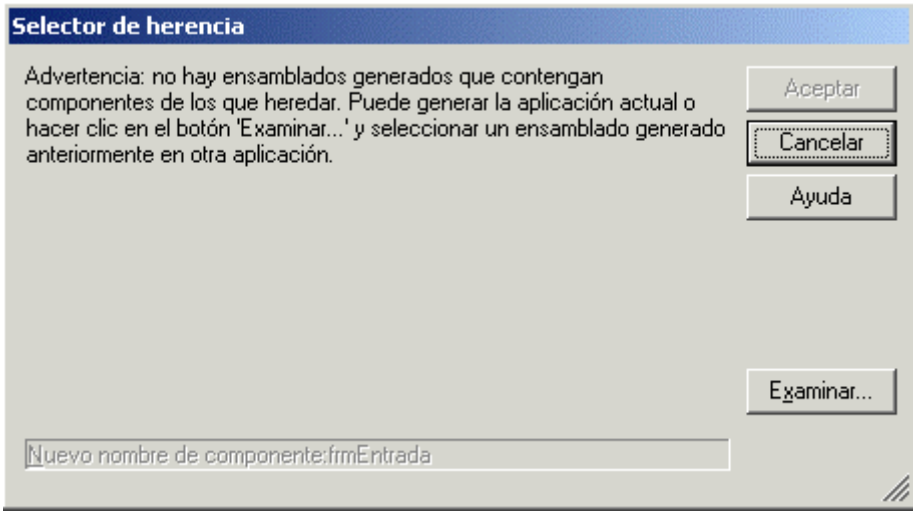


Figura 272. El IDE no puede localizar referencias hacia formularios base.

Tenemos por lo tanto, que pulsar el botón Examinar y buscar el archivo HERENCIAVISUAL.DLL, que creamos en el ejemplo anterior y se encuentra en la ruta de dicho proyecto, en su directorio bin: LetraUnidad:\HerenciaVisual\bin. Una vez localizado, aceptaremos esta ventana. Ver Figura 273.

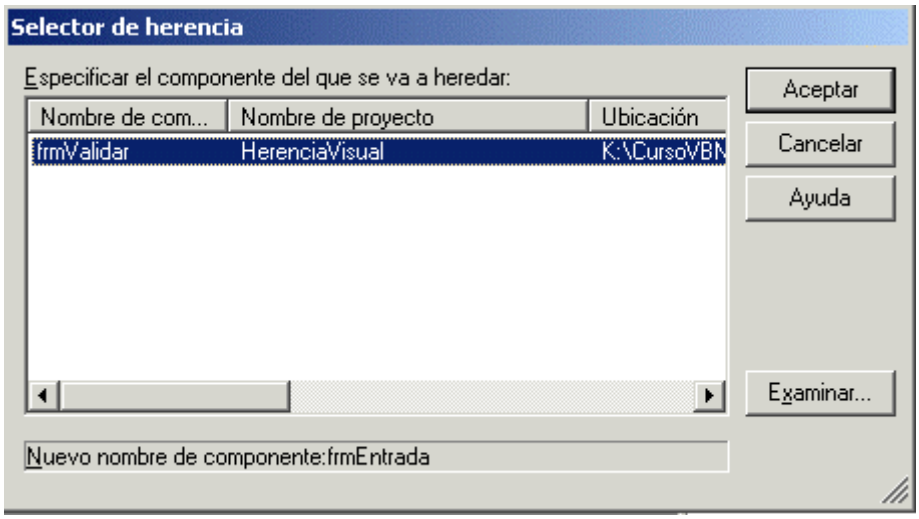


Figura 273. Seleccionar formulario base que se encuentra en otro ensamblado.

El IDE se encargará a continuación, de establecer la referencia entre nuestro proyecto y el ensamblado o archivo DLL que contiene el formulario base, mostrando finalmente el nuevo formulario heredado del mismo modo en que vimos en el ejemplo anterior. Seguidamente, añadiremos algunos controles para adaptarlo a las necesidades particulares de este proyecto. Ver Figura 274.

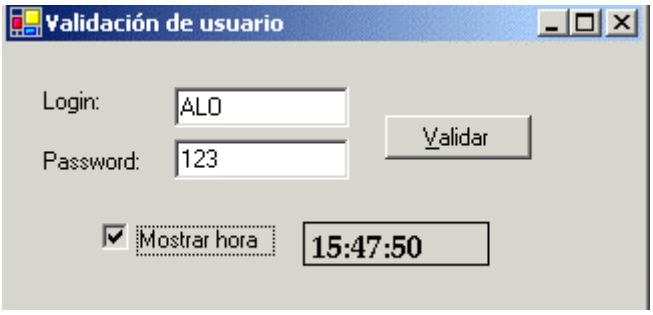


Figura 274. Formulario derivado de formulario base a través de herencia visual.

Como podemos comprobar, hemos agregado un control Timer, un CheckBox y un Label. Cuando hagamos clic sobre el CheckBox, se visualizará el Label mostrando la hora actual, que será actualizada cada segundo en el evento Tick del control Timer. Observe el lector los métodos Show( ) y Hide( ), que nos permiten mostrar y ocultar respectivamente un control. El Código fuente 479 muestra el código necesario para los eventos de estos controles.

```
Public Class frmEntrada
    Inherits HerenciaVisual.frmValidar
    '....
    '....
    ' al hacer clic en el CheckBox, mostrar u ocultar
    ' la hora según el valor de la casilla
```

```
    Private Sub chkMostrarHora_CheckedChanged(ByVal sender As System.Object, ByVal
e As System.EventArgs) Handles chkMostrarHora.CheckedChanged

        If Me.chkMostrarHora.Checked Then
            Me.lblHora.Show()
            Me.tmrTemporizador.Start()
        Else
            Me.lblHora.Hide()
            Me.tmrTemporizador.Stop()
        End If

    End Sub

    ' cuando el temporizador esté activo, al lanzarse este
    ' evento mostrar la hora en el Label
    Private Sub tmrTemporizador_Tick(ByVal sender As Object, ByVal e As
System.EventArgs) Handles tmrTemporizador.Tick

        Me.lblHora.Text = DateTime.Now.ToString("H:mm:ss")

    End Sub
End Class
```

Código fuente 479.

31

Menús

Controles de tipo menú

El menú es uno de los tipos de control más frecuentemente utilizados en los formularios Windows. Consiste en un conjunto de opciones, a través de las cuales, el usuario ejecutará algunos procesos de la aplicación. Disponemos de tres tipos de control menú: MainMenu, ContextMenu y MenuItem.

MainMenu y ContextMenu actúan como contenedores de grupos de controles MenuItem, representando este último control, la opción de menú sobre la que el usuario pulsa o hace clic.

El proyecto MenuPru que se acompaña como ejemplo, muestra los diferentes tipos de menú (hacer clic aquí para acceder al ejemplo). A continuación, describiremos los principales pasos a dar en el proceso de su construcción.

Menú Principal. MainMenu

Un control MainMenu, consiste en un conjunto de opciones que se sitúan horizontalmente debajo del título del formulario. A partir de cada opción, podemos asociar a su vez, grupos de opciones que se mostraran verticalmente al hacer clic en la opción principal o situada en la barra horizontal.

Para crear un menú principal, seleccionaremos del cuadro de herramientas el control MainMenu, y tras dibujarlo en el formulario, se añadirá una referencia del control al panel de controles especiales situado bajo el diseñador. La Figura 275 muestra un control de este tipo al que le hemos dado el nombre mnuPrincipal.

Figura 275. Menú de tipo MainMenu añadido al diseñador del formulario.

La creación de las diferentes opciones que compondrán el menú es un proceso que se ha mejorado y simplificado al máximo respecto a versiones anteriores de VB. El proceso de edición del menú se realiza directamente en el formulario, en el mismo lugar en el que el menú aparecerá en tiempo de ejecución.

Al hacer clic en la primera opción del menú, podemos dar nombre y propiedades a esa opción. Al mismo tiempo, de un modo muy intuitivo, veremos las próximas opciones disponibles, tanto las desplegables a partir de dicho menú, como las de la barra principal. Sólo hemos de movernos en la dirección que necesitemos y dar nombre a las opciones, y valores a sus propiedades. Ver Figura 276.

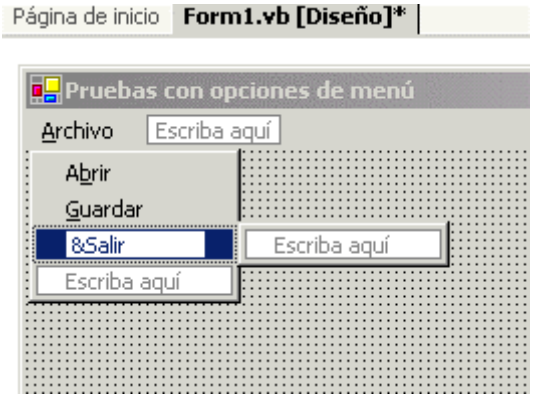


Figura 276. Creación de las opciones de un menú principal de formulario.

Cada una de las opciones que componen el menú es a su vez un control MenuItem. Si durante su creación sólo proporcionamos el nombre, el IDE va asignando a dicho control valores por defecto en sus propiedades.

Para modificar las propiedades de una opción de menú, sólo hemos de seleccionarlo en la estructura de menú que estamos creando en el diseñador del formulario, y pasar a la ventana de propiedades. Entre las propiedades disponibles para un MenuItem, podemos destacar las siguientes.

- **Text**. Contiene una cadena con el literal o texto descriptivo de la opción de menú.

- **Enabled**. Permite habilitar/deshabilitar la opción de menú. Cuando se encuentra deshabilitada, se muestra su nombre en un tono gris, indicando que no puede ser seleccionada por el usuario.

- **DefaultItem**. Permite establecer opciones por defecto. En una opción de menú por defecto, su texto se resalta en negrita.

- **Checked**. Marca/desmarca la opción. Cuando una opción está marcada, muestra junto a su nombre un pequeño símbolo de verificación o punteo.

- **RadioCheck**. En el caso de que la opción de menú se encuentre marcada, si asignamos True a esta propiedad, en lugar de mostrar el símbolo de verificación estándar, se muestra uno con forma de punto.

- **ShortCut**. Se trata de un atajo de teclado, o combinación de teclas que nos van a permitir ejecutar la opción de menú sin tener que desplegarlo. Al elegir esta propiedad, aparecerá una lista con todos los atajos disponibles para asignar.

- **ShowShortCut**. Permite mostrar u ocultar la combinación de teclas del atajo de teclado que tenga asignado una opción de menú.

- **Visible**. Muestra u oculta la opción de menú.

- **MdiList**. Esta propiedad se utiliza habitualmente en opciones situadas en la barra de menú, y permite establecer que dicha opción al desplegarse, muestre, además de las opciones de menú que le hayamos asignado, la lista de ventanas secundarias MDI, en el caso de que el menú principal esté contenido en un formulario de tipo MDI. Los formularios MDI serán tratados posteriormente.

Podemos adicionalmente, asignar una tecla de acceso rápido o *hotkey* a una opción de menú, anteponiendo el carácter & a la letra que deseemos, de las que se encuentran en la propiedad Text del control MenuItem. Al igual que sucede con los demás tipos de controles, en el texto de la opción de menú, aparecerá subrayada la mencionada letra. De este modo, cuando despleguemos un menú, no será necesario posicionarnos en una de ellas para ejecutarla, sino que simplemente pulsando la tecla rápida, se ejecutará el código de dicha opción.

También podemos establecer separadores entre las opciones de menú simplemente creando una opción y asignando a su propiedad Text el carácter de guión (-).

En nuestro formulario de ejemplo, vamos pues a diseñar un menú con la estructura del esquema mostrado en la Figura 277.

Para todas las opciones se ha asignado una tecla de acceso rápido, y adicionalmente, para las opciones que se indican a continuación, se han modificado algunas propiedades por defecto.

- **Guardar**. Deshabilitada.

- **Salir**. Atajo de teclado en Ctrl. + S.

- **Copiar**. Opción por defecto.

- **Pegar**. Marcada con símbolo normal.

- **Cortar**. Marcada con símbolo de círculo.

- **Elipse**. Opción no visible.

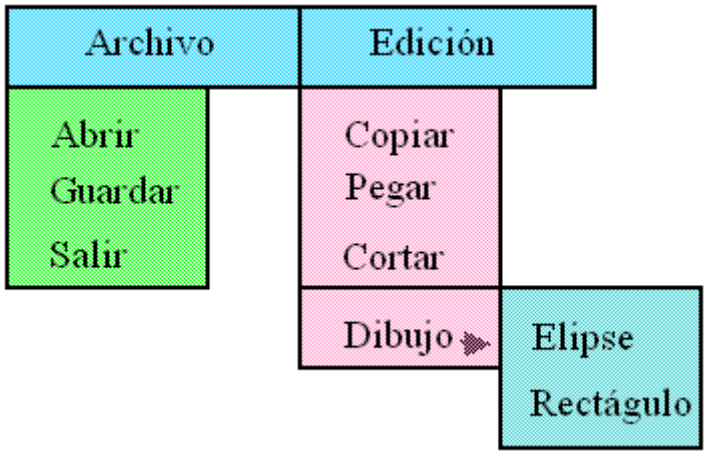


Figura 277. Esquema del menú de ejemplo.

La Figura 278 muestra el formulario en ejecución con una parte del menú abierta.

Figura 278. Menú desplegado a varios niveles, mostrando opciones.

Una vez finalizada la fase de diseño del menú, debemos proceder a escribir el código para sus opciones. El evento Click es el que permite a un control MenuItem ejecutar código cuando la opción de menú es seleccionada. Abriendo por tanto, el menú desde el diseñador del formulario, y haciendo doble clic en la opción correspondiente, nos situaremos en el editor de código, dentro del procedimiento manipulador del evento Click para esa opción. El Código fuente 480 muestra el código que se ejecutará cuando seleccionemos las opciones de menú Abrir y Salir de nuestro ejemplo.

```
Private Sub mnuAbrir_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles mnuAbrir.Click

    MessageBox.Show("Opción Abrir del menú")

End Sub

Private Sub mnuSalir_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles mnuSalir.Click

    Me.Close()
```

```
End Sub
```

Código fuente 480

Puesto que muchas de las propiedades de un control MenuItem son manipulables en tiempo de ejecución, añadiremos al formulario varios botones, mediante los cuales realizaremos operaciones sobre las opciones del menú tales como habilitar y deshabilitar, mostrar y ocultar, cambiar el nombre, etc. La Figura 279 muestra el formulario con estos nuevos botones.

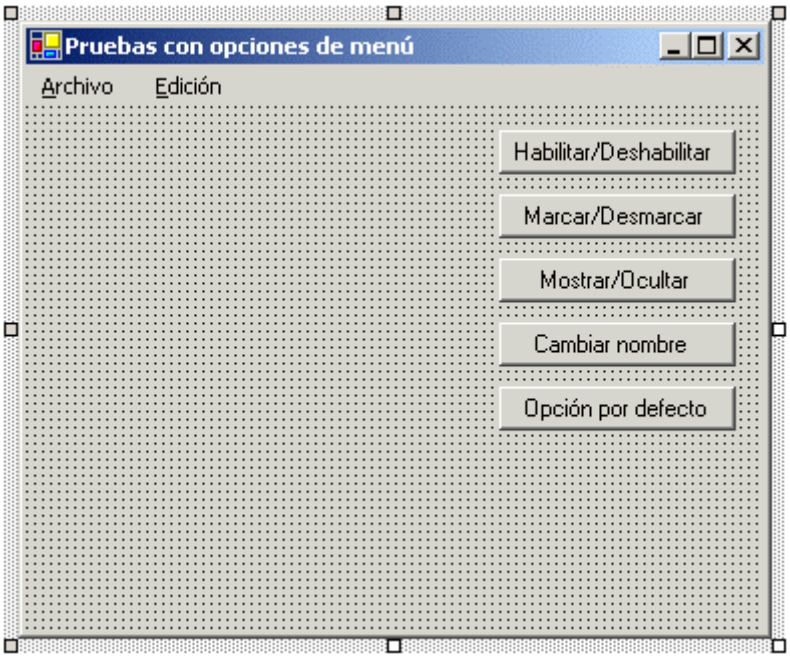


Figura 279. Controles Button para manipular por código las propiedades de las opciones del menú.

En el Código fuente 481 podemos ver los manipuladores de evento de estos botones.

```
Private Sub btnHabilitar_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnHabilitar.Click

    Me.mnuGuardar.Enabled = Not Me.mnuGuardar.Enabled

End Sub

Private Sub btnMarcar_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnMarcar.Click

    Me.mnuPegar.Checked = Not Me.mnuPegar.Checked

End Sub

Private Sub btnMostrar_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnMostrar.Click

    Me.mnuElipse.Visible = Not Me.mnuElipse.Visible

End Sub
```

```
Private Sub btnNombre_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnNombre.Click

    If Me.mnuAbrir.Text = "A&brir" Then
        Me.mnuAbrir.Text = "HO&LA"
    Else
        Me.mnuAbrir.Text = "A&brir"
    End If

End Sub

Private Sub btnDefecto_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnDefecto.Click

    Me.mnuCopiar.DefaultItem = Not Me.mnuCopiar.DefaultItem

End Sub
```

Código fuente 481

Menú Contextual. ContextMenu

El control ContextMenu representa un menú contextual o flotante. Este tipo de menú se asocia al formulario o a uno de sus controles, de modo que al hacer clic derecho, se mostrará sobre el elemento al que se haya asociado.

El modo de añadir un control ContextMenu y sus correspondientes opciones al formulario, es el mismo que para un MainMenu; situándose también una referencia del menú contextual en el panel de controles especiales del diseñador. Antes de poder diseñar las opciones de un ContextMenu, debemos pulsar la referencia de dicho menú que existe en el panel de controles especiales, ya que por defecto, el formulario muestra el menú principal en caso de que tenga uno definido.

La Figura 280 muestra el menú contextual mnuContexto, que hemos añadido al formulario. Para asociar este menú con un control o formulario, utilizaremos la propiedad ContextMenu de que disponen la mayoría de los controles Windows. En este ejemplo, insertaremos el control txtValor, de tipo TextBox, y le asociaremos el menú de contexto que acabamos de crear.

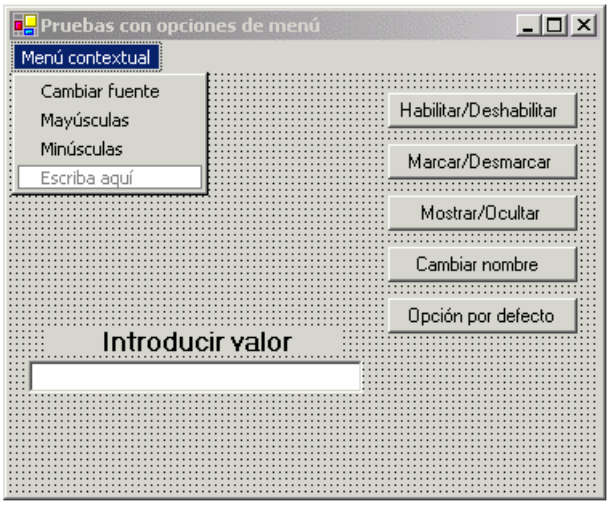


Figura 280. Diseño de un menú contextual.

Como resultado, cuando ejecutemos la aplicación, al hacer clic derecho sobre el TextBox, aparecerá el menú contextual que hemos asignado, mediante el que cambiaremos el tipo de fuente de la caja de texto, transformaremos el texto a mayúsculas y minúsculas. El Código fuente 482 muestra el código de los eventos Click correspondiente a las opciones del menú contextual.

```
Private Sub mnuFuente_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles mnuFuente.Click

    Dim oFuente As New Font("Comic", 15)
    Me.txtValor.Font = oFuente

End Sub

Private Sub mnuMayusculas_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles mnuMayusculas.Click

    Me.txtValor.Text = Me.txtValor.Text.ToUpper()

End Sub

Private Sub mnuMinusculas_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles mnuMinusculas.Click

    Me.txtValor.Text = Me.txtValor.Text.ToLower()

End Sub
```

Código fuente 482

La Figura 281 muestra el aspecto del menú contextual, cuando es utilizado desde el control TextBox.

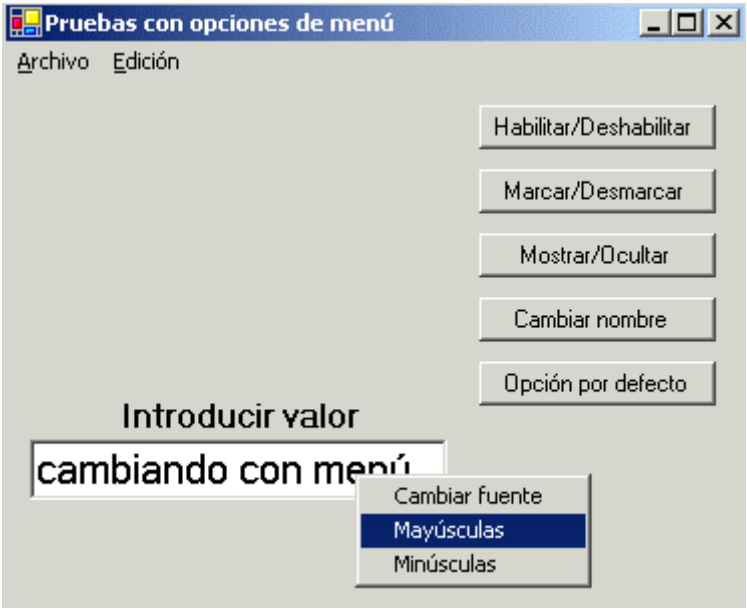


Figura 281. Control ContextMenu asociado a un TextBox.

Creación de menús desde código

Si analizamos el código generado por el diseñador del formulario, observaremos que es relativamente fácil la construcción por código de la estructura de menús de un formulario, o la inserción de nuevas opciones de menú al menú principal ya existente.

Centrándonos en esta última situación, añadiremos dos nuevos controles Button al formulario, btnCrear y btnEliminar. En el primero escribiremos el código para crear una nueva opción en la barra de menús, con algunas opciones desplegables que partan de él; asociando también a estas opciones, los manipuladores de evento necesarios.

En el último botón eliminaremos este menú creado en tiempo de ejecución. El Código fuente 483 muestra las instrucciones necesarias para ejecutar estos procesos.

```
' al comienzo de la clase
' declaramos las variables que contendrán
' los tipos MenuItem creados en ejecución
Public Class Form1
    Inherits System.Windows.Forms.Form

    ' declaración de las variables que contendrán
    ' los tipos MenuItem creados en ejecución
    Private mnuContabilidad As MenuItem
    Private WithEvents mnuApuntes As MenuItem
    Private mnuBalances As MenuItem
    '....
    '....
End Class

'***************
Private Sub btnCrear_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnCrear.Click
    ' crear opciones de menú en tiempo de ejecución
    ' y añadirlas al menú del formulario

    ' opción de barra de menús
    Me.mnuContabilidad = New MenuItem()
    Me.mnuContabilidad.Text = "&Contabilidad"

    ' opciones a desplegar del menú Contabilidad
    Me.mnuApuntes = New MenuItem()
    Me.mnuApuntes.Text = "&Apuntes"
    Me.mnuApuntes.Checked = True
    Me.mnuApuntes.RadioCheck = True

    Me.mnuBalances = New MenuItem()
    Me.mnuBalances.Text = "&Balances"
    Me.mnuBalances.DefaultItem = True
    AddHandler mnuBalances.Click, AddressOf OpcionBalances

    ' añadir al menú Contabilidad sus opciones
    Me.mnuContabilidad.MenuItems.AddRange(New MenuItem() {Me.mnuApuntes,
Me.mnuBalances})

    ' añadir al menú del formulario la opción de la barra
    Me.Menu.MenuItems.AddRange(New MenuItem() {Me.mnuContabilidad})
End Sub

'***************
' este procedimiento se lanzará al seleccionar
' la opción de menú Apuntes
```

```
Private Sub OpcionApuntes(ByVal sender As Object, ByVal e As EventArgs) Handles
mnuApuntes.Click

    MessageBox.Show("Opción Apuntes del menú")

End Sub

'***************
' este procedimiento se lanzará al seleccionar
' la opción de menú Balances
Private Sub OpcionBalances(ByVal sender As Object, ByVal e As EventArgs)

    MessageBox.Show("Se ha seleccionado la opción Balances")

End Sub

'***************
' al pulsar este botón, eliminamos el menú Contabilidad
' del menú principal del formulario
Private Sub btnEliminar_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnEliminar.Click
    Dim oMenuItem As MenuItem

    ' recorrer la colección de menús del formulario
    For Each oMenuItem In Me.Menu.MenuItems
        ' buscar por el nombre de menú
        If oMenuItem.Text.IndexOf("Contabilidad") > 0 Then
            ' si lo encontramos, quitarlo
            Me.Menu.MenuItems.Remove(oMenuItem)
        End If
    Next

End Sub
```

Código fuente 483

En la Figura 282 se visualiza este formulario al ser ejecutado, incluyendo el nuevo menú creado por código.

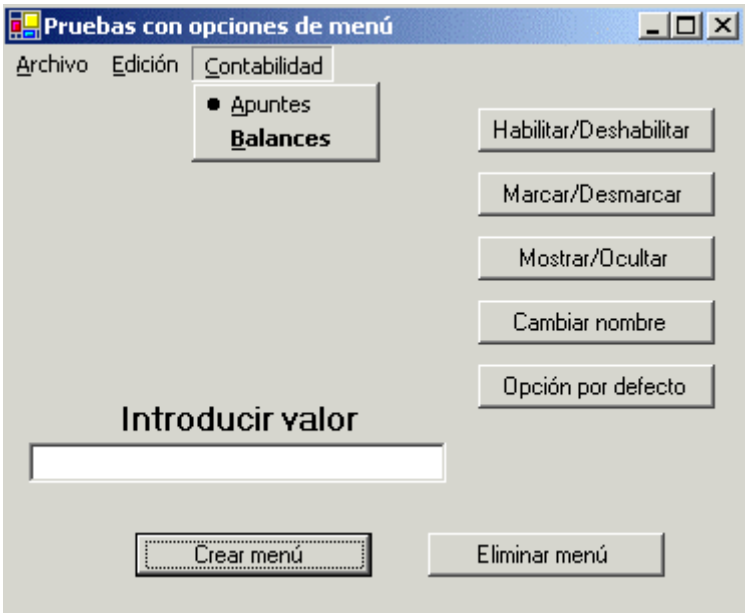


Figura 282. Formulario con menú creado desde código.

Aunque en este ejemplo hemos añadido opciones en tiempo de ejecución a un menú de formulario ya existente, podemos partir de un formulario sin menú, y crear todas las opciones partiendo de cero, asignando el menú construido por código a la propiedad Menu de nuestro objeto formulario.

32

Programación con hebras

Manipulación de hebras de ejecución

La plataforma .NET Framework proporciona, a través de la jerarquía de clases del sistema, la creación y manipulación de hebras independientes de ejecución para nuestras aplicaciones. Esto quiere decir que las hebras creadas en el programa pueden ejecutar procesos en paralelo compartiendo ciclos del procesador.

Al tener pleno control, además de crear hebras, podemos finalizarlas en cualquier momento que necesitemos, retardar su ejecución y otras funcionalidades diversas.

En versiones anteriores de VB, un problema habitual podría ser el siguiente: disponemos de un formulario en el que al pulsar un botón, ponemos en marcha un proceso que en ocasiones puede prolongarse más de lo que inicialmente habíamos calculado; sin embargo, una vez iniciado, dicho proceso no puede ser cancelado, ya que ha acaparado todos los recursos de la aplicación, impidiéndonos interactuar con cualquier otro elemento de la misma.

En VB.NET, este escenario de trabajo cambia radicalmente: podemos situar el código del proceso que ejecuta el botón en una hebra; esto nos permite seguir actuando con el formulario del modo habitual, mientras que el proceso se ejecuta en segundo plano. Si el proceso se prolonga en exceso, lo detendremos, finalizando la hebra en la que se ejecuta.

La clase Thread

Esta es la clase que nos va a permitir la creación y manipulación de procesos en hebras independientes. Se encuentra ubicada dentro del espacio de nombres Threading, el cual, contiene todas las clases de la plataforma .NET relacionadas con la programación basada en hebras. Entre los miembros de Thread destacaremos los siguientes.

- **New**. Se trata del constructor de la clase, y como parámetro debemos incluir la dirección de entrada del procedimiento que debe ejecutar la hebra. Para indicar dicha dirección del procedimiento utilizaremos el operador AddressOf.
- **Start( )**. Inicia la ejecución de la hebra.
- **Abort( )**. Cancela la ejecución de la hebra.
- **Suspend( )**. Suspende la ejecución de la hebra.
- **Sleep( )**. Permite establecer un retardo en milisegundos para la ejecución de la hebra.
- **CurrentThread**. Devuelve una instancia del objeto Thread actualmente en ejecución.
- **Join( )**. Devuelve un valor lógico que indica si la hebra ya ha finalizado.
- **IsAlive( )**. Devuelve un valor lógico que indica si la hebra está todavía activa o viva.

De forma adicional a los miembros de esta clase, y como veremos en los siguientes ejemplos, en la clase AppDomain disponemos de la propiedad GetCurrentThreadId, que devuelve un valor numérico con el identificador de la hebra que se encuentra actualmente en ejecución.

También disponemos de la estructura SyncLock...End SyncLock, que evita el que varias hebras intenten hacer uso de un mismo objeto.

Ejecutar un proceso en una hebra

Vamos a desarrollar en una aplicación Windows de ejemplo, la situación mencionada anteriormente: crearemos un formulario en el que al pulsar un botón, rellenemos un ListBox con valores durante un determinado tiempo; este proceso lo ejecutaremos desde una hebra, de modo que podremos escribir libremente en otro control TextBox del formulario sin tener que esperar a que el ListBox se haya llenado (hacer clic aquí para acceder al proyecto de ejemplo ThreadBasica).

Una vez que hemos diseñado el formulario, abriremos el editor de código e importaremos el namespace Threading en la cabecera del fichero de código.

A continuación declararemos a nivel de la clase del formulario, una variable de tipo Thread, que será la que nos permita crear la hebra de ejecución separada. Ver Código fuente 484.

```
Imports System.Threading


Public Class Form1
    Inherits System.Windows.Forms.Form
```

```
    Private oHebra As Thread
    '....
```

Código fuente 484

Para terminar, escribiremos un método en la clase que será el que realice el llenado del ListBox, y codificaremos el evento Click del botón del formulario, que crea la hebra y la pone en marcha. Observe el lector, como mientras se produce el llenado del ListBox, podemos escribir texto en el control TextBox. Si no utilizáramos la hebra, hasta que el control de lista no estuviera lleno, no podríamos pasar el foco al TextBox para trabajar con él. Ver Código fuente 485.

```
Private Sub btnIniciar_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnIniciar.Click

    ' instanciar la hebra y ponerla en marcha
    ' en el constructor de la hebra, indicamos
    ' qué procedimiento deberá ejecutar al ser iniciada
    oHebra = New Thread(AddressOf LlenarLista)
    oHebra.Start()

End Sub

Private Sub LlenarLista()
    Dim iContador As Integer
    Dim iCuentaBis As Integer

    For iContador = 1 To 10000
        Me.lstProceso.Items.Add("Contador: " & iContador)

        For iCuentaBis = 1 To 50000
            ' retardo
        Next
    Next

    ' finalizamos la hebra
    oHebra.Abort()
End Sub
```

Código fuente 485

La Figura 283 muestra este ejemplo en ejecución.

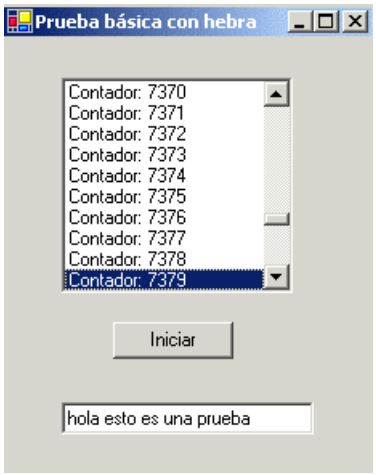


Figura 283. Ejecución de un proceso en una hebra.

Control de procesos indefinidos

Otro escenario de trabajo consistiría en la capacidad de manipular un proceso ejecutado sin un punto determinado de finalización.

El ejemplo del proyecto HebraInfinito (hacer clic aquí para acceder a este ejemplo), consta de un formulario con dos botones y un control de lista. El control btnIniciar pone en funcionamiento, a través de una hebra, la ejecución de un procedimiento que llena un ListBox dentro de un bucle infinito.

En otras circunstancias, este diseño de la aplicación sería impensable, pero en nuestro caso, al estar gestionado por una hebra, vamos a pararla en el momento en que consideremos oportuno mediante la pulsación del otro botón, btnDetener. El Código fuente 486 muestra el código de la clase de este formulario.

```
Imports System.Threading

Public Class Form1
    Inherits System.Windows.Forms.Form

    Private oHebra As Thread
    '....
    '....

    Private Sub btnIniciar_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnIniciar.Click

        ' limpiamos el ListBox
        Me.lstProceso.Items.Clear()

        ' creamos una hebra que ejecute un procedimiento
        oHebra = New Thread(AddressOf LlenarLista)
        ' iniciamos la hebra
        oHebra.Start()

    End Sub

    Private Sub btnDetener_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnDetener.Click

        oHebra.Abort()
        Me.lstProceso.Items.Add("Hebra detenida")

    End Sub

    Private Sub LlenarLista()
        Dim iContador As Integer
        Dim iCuentaBis As Integer

        ' con AppDomain.GetCurrentThreadId obtenemos el
        ' identificador de la hebra que está actualmente
        ' en ejecución
        Me.lstProceso.Items.Add("ID hebra: " & AppDomain.GetCurrentThreadId)

        While True
            iContador += 1
            Me.lstProceso.Items.Add("Contador pasos: " & iContador)

            For iCuentaBis = 1 To 50000
                ' retardo
            Next
```

```
        End While

    End Sub

End Class
```

Código fuente 486

En la Figura 284 podemos ver esta aplicación una vez que la hebra ha sido detenida.

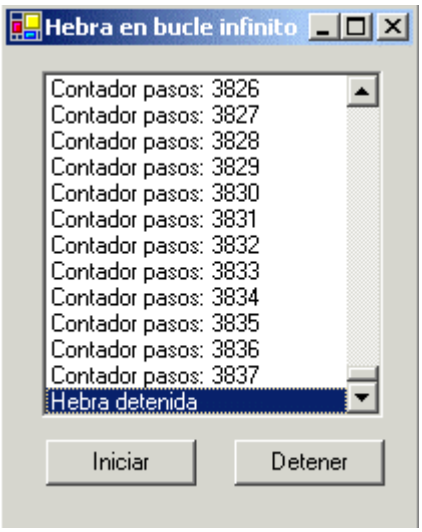


Figura 284. Ejecución de un bucle infinito en una hebra.

El lector habrá observado que en este ejemplo, también incluimos un bucle de retardo al llenar el ListBox. Esto no obstante, no es necesario utilizando la clase Thread, puesto que su método Sleep( ), nos permitirá establecer un tiempo de parada en la ejecución de la hebra, determinado en milisegundos.

Podemos acceder a la hebra actual mediante el método compartido GetCurrentThread de esta clase. Esta es la manera en que ejecutamos, en este ejemplo, el método Sleep( ).

Por lo tanto, si modificamos el código del procedimiento LlenarLista( ) de este ejemplo, por el mostrado en el Código fuente 487 conseguiremos el mismo resultado

```
Private Sub LlenarLista()
    Dim iContador As Integer
    Dim iCuentaBis As Integer

    ' con AppDomain.GetCurrentThreadId obtenemos el
    ' identificador de la hebra que está actualmente
    ' en ejecución
    Me.lstProceso.Items.Add("ID hebra: " & AppDomain.GetCurrentThreadId)

    While True
        iContador += 1
        Me.lstProceso.Items.Add("Contador pasos: " & iContador)

        ' frenar la ejecución de la hebra
        Thread.CurrentThread.Sleep(500)
```

```
    End While
End Sub
```

Código fuente 487

Ejecución multihebra

En los anteriores apartados hemos creado una hebra, y llamado desde la misma a un único procedimiento, para ejecutar un determinado proceso. Sin embargo la verdadera potencia de las hebras radica en su capacidad para ejecutar simultáneamente varios procesos.

En el proyecto de ejemplo VariasHebras (hacer clic aquí para acceder a este ejemplo), se muestra un formulario con las mismas características de los pasados ejemplos: un ListBox y un Button. En esta ocasión, sin embargo, al pulsar el botón, crearemos varias hebras que al ponerse en ejecución, llamarán al mismo procedimiento. Ver el Código fuente 488.

```
Imports System.Threading

Public Class Form1
    Inherits System.Windows.Forms.Form
    '....
    '....
    ' al pulsar este botón
    Private Sub btnIniciar_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnIniciar.Click

        Dim iContador As Integer
        Dim oHebra As Thread

        ' limpiamos el ListBox
        Me.lstValores.Items.Clear()

        ' creamos varias hebras que ejecuten el mismo procedimiento
        For iContador = 1 To 10
            oHebra = New Thread(AddressOf LlenarLista)
            oHebra.Start()
        Next

    End Sub

    ' procedimiento que será ejecutado por las hebras
    Private Sub LlenarLista()
        Dim iContador As Integer

        ' indicamos en que identificador de hebra se está
        ' ejecutando este procedimiento
        Me.lstValores.Items.Add("Sub LlenarLista - ejecutándose en hebra: " &
AppDomain.GetCurrentThreadId)

        ' recorremos un bucle e indicamos
        ' en cada iteración la hebra actual y
        ' el contador de paso
        For iContador = 1 To 7
            Me.lstValores.Items.Add(ControlChars.Tab & _
                AppDomain.GetCurrentThreadId & " paso " & iContador)
        Next

    End Sub
```

```
End Class
```

Código fuente 488

La Figura 285 muestra el resultado de una de las ejecuciones de esta aplicación.

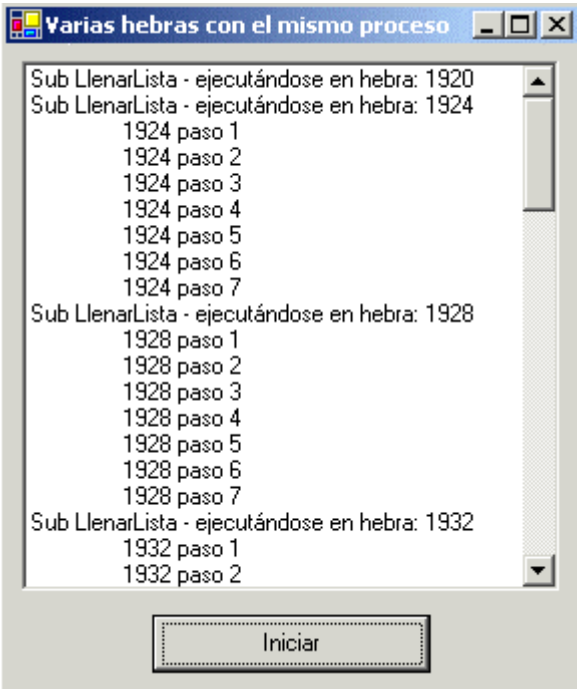


Figura 285. Varias hebras ejecutan el mismo procedimiento.

En la anterior figura aparece un identificador de hebra, del que no se muestran a continuación las correspondientes iteraciones; estos elementos se encuentran más adelante, mezclados con los valores de otro proceso. Esto se debe a una cuestión de sincronización de la que trataremos más adelante.

Tenga en cuenta también el lector, que al probar este ejemplo, puede que los valores le aparezcan correctamente, puesto que no podemos precisar las prioridades que asigna el procesador a la hora de ejecutar las hebras.

Ejecución multihebra de múltiples procesos

Variando el caso expuesto en el ejemplo anterior, en el proyecto VariasHebrasProcesos (hacer clic aquí para acceder a este ejemplo) vamos a ejecutar dos procesos distintos en dos hebras también diferentes, que se ocuparán de llenar el mismo ListBox. El Código fuente 489 muestra este nuevo código con las variantes introducidas respecto al ejemplo anterior.

```
' al pulsar este botón
Private Sub btnIniciar_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnIniciar.Click

    Dim iContador As Integer
    Dim oHebra As Thread
    Dim oHebraBis As Thread
```

```
    ' limpiamos el ListBox
    Me.lstValores.Items.Clear()

    ' creamos varias hebras que ejecuten varios procedimientos
    For iContador = 1 To 10
        oHebra = New Thread(AddressOf LlenarPrimero)
        oHebra.Start()

        oHebraBis = New Thread(AddressOf OtroLlenado)
        oHebraBis.Start()
    Next

End Sub

' procedimientos que ejecutados por las hebras
' -------------------------------------------
Private Sub LlenarPrimero()
    Dim iContador As Integer

    ' indicamos en qué identificador de hebra se está
    ' ejecutando este procedimiento
    Me.lstValores.Items.Add("Sub LlenarPrimero - ejecutándose en hebra: " &
AppDomain.GetCurrentThreadId)

    ' recorremos un bucle e indicamos
    ' en cada iteración la hebra actual y
    ' el contador de paso
    For iContador = 1 To 7
        Me.lstValores.Items.Add(ControlChars.Tab & _
            AppDomain.GetCurrentThreadId & " paso " & iContador)
    Next

End Sub

Private Sub OtroLlenado()
    Dim iContador As Integer

    ' indicamos en qué identificador de hebra se está
    ' ejecutando este procedimiento
    Me.lstValores.Items.Add("Sub OtroLlenado - ejecutándose en hebra: " &
AppDomain.GetCurrentThreadId)

    ' recorremos un bucle e indicamos
    ' en cada iteración la hebra actual y
    ' el contador de paso
    For iContador = 1 To 7
        Me.lstValores.Items.Add(ControlChars.Tab & _
            AppDomain.GetCurrentThreadId & " paso " & iContador)
    Next

End Sub
```

Código fuente 489

Veamos en la Figura 286 el formulario en ejecución. En este caso también aparecen los valores mezclados de las hebras, al igual que anteriormente, por cuestiones de sincronización.

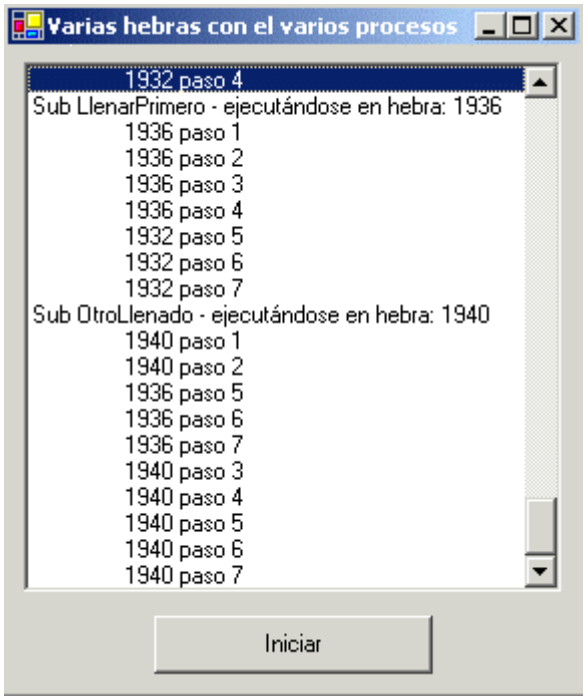


Figura 286. Ejecución de varios procesos en distintas hebras.

Detectando el estado de finalización

Una vez lanzada la ejecución de una hebra, podemos emplear el método Join( ) para averiguar su estado de finalización.

Si queremos esperar un tiempo antes de comprobar dicho estado, pasaremos como parámetro a este método, el tiempo a esperar en milisegundos.

Esto es lo que haremos en el proyecto HebraFinaliz (hacer clic aquí para acceder a este ejemplo), donde también ejecutaremos el proceso de llenado de un ListBox. Ver Código fuente 490.

```
Private Sub btnIniciar_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnIniciar.Click

    Dim Contador As Integer
    Dim Hebra As Thread

    Me.lstValores.Items.Clear()

    Hebra = New Thread(AddressOf AgregarConRetardo)
    Hebra.Start()

    ' si la hebra no se ha terminado, el método Join()
    ' devuelve False; pero si ha terminado devuelve True
    If Hebra.Join(1000) Then
        Me.lblMensaje.Text = "Hebra finalizada"
    Else
        Me.lblMensaje.Text = "Hebra NO finalizada"
    End If

End Sub

'procedimiento que se ejecutará dentro de una hebra
```

```
' especificando un retardo en el bucle del procedimiento
Private Sub AgregarConRetardo()
    Dim Contador As Integer

    Me.lstValores.Items.Add("Sub AgregarConRetardo - ejecutandose en hebra " &
AppDomain.GetCurrentThreadId().ToString())

    For Contador = 1 To 20
        Me.lstValores.Items.Add("     hebra " &
AppDomain.GetCurrentThreadId().ToString() & "  paso " & Contador.ToString())
        Thread.CurrentThread.Sleep(250)
    Next

End Sub
```

Código fuente 490

Veamos el formulario de este proyecto en la Figura 287.

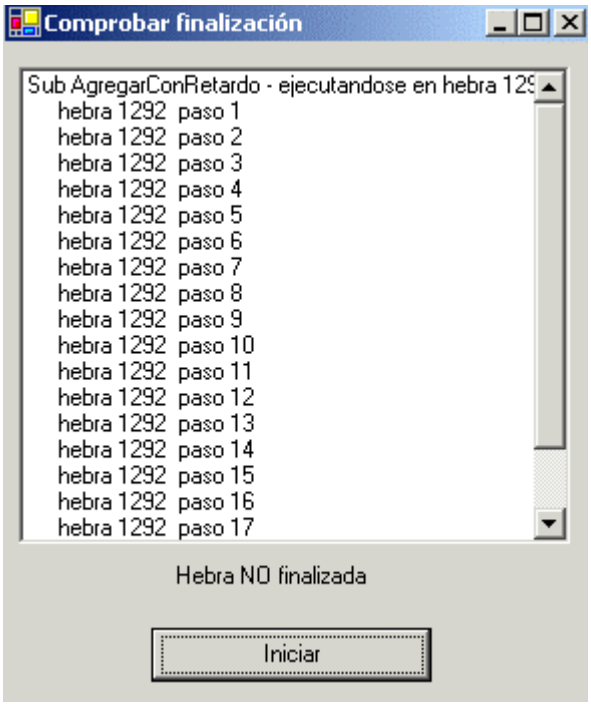


Figura 287. Comprobando la finalización de la hebra.

Ejecución paralela de procesos a distintos ritmos

En el ejemplo HebrasRitmos (hacer clic aquí para acceder a este ejemplo) vamos a llenar dos controles de lista al mismo tiempo con sendas hebras. El aspecto a destacar en este caso, es que los procedimientos a ejecutar en cada hebra, serán ejecutados a distinto ritmos a través de retardos mediante el método Sleep( ), de modo que el procedimiento que debe llenar menos elementos, y que en principio debería terminar antes, será el que termine el último. Ver Código fuente 491.

```
Private Sub btnIniciar_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnIniciar.Click

    Dim Contador As Integer
```

```
    Dim Hebra As Thread
    Dim HebraBis As Thread

    Me.lstValores.Items.Clear()
    Me.lstDatos.Items.Clear()

    Hebra = New Thread(AddressOf LlenaListaValores)
    Hebra.Start()

    HebraBis = New Thread(AddressOf LlenaListaDatos)
    HebraBis.Start()

End Sub

' este procedimiento se ejecutará en una hebra
' y llenará un ListBox
Private Sub LlenaListaValores()
    Dim Contador As Integer
    Me.lstValores.Items.Add("Sub LlenaListaValores - ejecutandose en hebra " &
AppDomain.GetCurrentThreadId().ToString())

    For Contador = 1 To 30
        Me.lstValores.Items.Add("     hebra " &
AppDomain.GetCurrentThreadId().ToString() & "  paso " & Contador.ToString())
        Thread.CurrentThread.Sleep(500)
    Next
End Sub

' este procedimiento se ejecutará en una hebra
' y llenará el otro ListBox
Private Sub LlenaListaDatos()
    Dim Contador As Integer

    Me.lstDatos.Items.Add("Sub LlenaListaDatos - ejecutandose en hebra " &
AppDomain.GetCurrentThreadId().ToString())

    For Contador = 1 To 50
        Me.lstDatos.Items.Add("     hebra " &
AppDomain.GetCurrentThreadId().ToString() & "  paso " & Contador.ToString())
        Thread.CurrentThread.Sleep(150)
    Next
End Sub
```

Código fuente 491.

La Figura 288 muestra un instante de la ejecución de este programa de ejemplo.

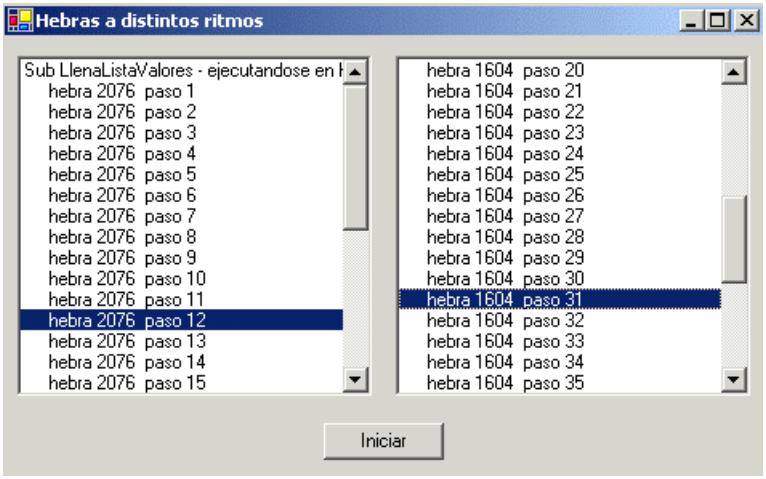


Figura 288. Ejecución de dos hebras a distinto ritmo.

Sincronización de hebras

En un ejemplo anterior, en el cual, varias hebras llamaban a su vez a múltiples procedimientos, es posible como se comentaba al final de dicho apartado, que el resultado obtenido mostrara la ejecución mezclada de varias hebras. El motivo es que todas las hebras creadas trabajaban contra el mismo objeto: el control ListBox del formulario, interfiriéndose mutuamente en algunas ocasiones.

Para evitar este problema, .NET nos proporciona la estructura SyncLock, que nos permite bloquear un objeto que se esté ejecutando dentro del proceso de una hebra, de forma que hasta que dicha hebra no haya finalizado de usar el objeto, no se desbloqueará para ser utilizado por otra hebra, o lo que es igual, SyncLock permite que un objeto sólo pueda ser utilizado por una hebra a la vez.

Para demostrar el uso de SyncLock, crearemos una nueva aplicación con el nombre HebraSincro (hacer clic aquí para acceder a este ejemplo), incluyendo un control Button y un ListBox en su formulario.

Al pulsar el botón de este formulario, crearemos dos hebras que ejecutarán un procedimiento diferente cada una. El procedimiento rellenará de valores el ListBox, con una diferencia respecto a los anteriores ejemplos, consistente en que cuando una de las hebras ejecute su procedimiento, bloqueará el ListBox y hasta que no termine de rellenarlo, no cederá el objeto a la siguiente hebra. Veamos estas operaciones en el Código fuente 492.

```
Private Sub btnIniciar_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnIniciar.Click

    ' crear dos hebras y lanzarlas
    Dim HebraUno As New Thread(AddressOf ManipListaUno)
    Dim HebraDos As New Thread(AddressOf ManipListaDos)
    HebraUno.Start()
    HebraDos.Start()

End Sub

'procedimiento que será ejecutado en una de las hebras
Private Sub ManipListaUno()
    Dim Contador As Integer
    ' si hay hebras que utilizan objetos comunes,
    ' (en este caso el listbox)
    ' es necesario sincronizar el objeto de forma
    ' que sólo pueda ser utilizado por una hebra a la vez
    SyncLock (Me.lstValores)
        For Contador = 1 To 100
            Me.lstValores.Items.Add("Sub ManipListaUno - paso: " & Contador)
        Next
    End SyncLock
End Sub
' procedimiento que será ejecutado en una de las hebras
Private Sub ManipListaDos()
    Dim Contador As Integer
    ' sincronizar también en este procedimiento
    ' el listbox
    SyncLock (Me.lstValores)
        For Contador = 1 To 100
            Me.lstValores.Items.Add("Sub ManipListaDos - paso: " & Contador)
        Next
    End SyncLock
End Sub
```

Código fuente 492

En esta ocasión, no aparecerán en el ListBox, valores entremezclados producto de los intentos de las hebras de acaparar el uso del control del formulario, por el contrario, aparecerán en primer lugar los valores de la ejecución de la primera hebra, y a continuación los de la segunda.

Debido a que el código se ejecuta a gran velocidad, para comprobar mejor este efecto, puede ser buena idea introducir un retardo en cada procedimiento que ejecutan las hebras, de manera que podamos observar con más detenimiento el proceso.

Crear un proceso de monitorización

En este caso vamos a ver la aplicación de hebras desde una perspectiva diferente a la utilizada en las anteriores situaciones.

El objetivo del proyecto HebraMonitor (hacer clic aquí para acceder a este ejemplo), consiste en introducir una clave numérica en un TextBox, que cuando sea de seis números y coincida con un valor que contiene el código de la aplicación, mostrará un mensaje indicando el éxito de la operación. Naturalmente, el código que va a estar permanentemente monitorizando el valor que hay en el TextBox, se ejecutará desde una hebra que iniciaremos o pararemos cuando queramos. Ver el Código fuente 493.

```
Private oHebra As Thread

Private Sub btnIniciar_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnIniciar.Click

    ' instanciar una hebra y ponerla en ejecución
    oHebra = New Thread(AddressOf ComprobarClave)
    oHebra.Start()

End Sub

Private Sub btnParar_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnParar.Click

    ' detener la hebra que está en ejecución
    oHebra.Abort()

    Me.lblContador.Text = ""
    Me.lblEstado.Text = ""

End Sub

' procedimiento que se ejecutará dentro
' de la hebra creada en la aplicación
Public Sub ComprobarClave()
    Dim iContadorPasos As Integer

    While True
        iContadorPasos += 1
        Me.lblContador.Text = iContadorPasos
        If Me.txtClave.Text.Length = 6 Then
            If Me.txtClave.Text = "778899" Then
                Me.lblEstado.Text = "CORRECTA"
            Else
                Me.lblEstado.Text = "NO ES CORRECTA"
            End If
        Else
            Me.lblEstado.Text = ""
        End If
```

```
        ' frenar la ejecución de la hebra
        oHebra.Sleep(250)
    End While

End Sub
```

Código fuente 493

Podemos observar que una vez iniciada la hebra, se comienza la ejecución de un bucle infinito, que muestra en un Label la cantidad de ocasiones en que se realiza la comprobación, y si el valor introducido en el TextBox es o no correcto. En cualquier momento, podemos detener el proceso, pulsando el botón btnParar, que ejecuta el método Abort() de la hebra.

Otro detalle que se nos puede pasar por alto, radica en el hecho de que si iniciamos la hebra y cerramos el formulario, la hebra sigue en ejecución; podemos comprobar esto abriendo el menú Depurar de Visual Studio, en el que sorprendentemente, aparecerá la opción *Detener depuración*, lo cual indica que hay todavía un proceso en ejecución, "¿pero como?, si yo he cerrado el formulario", pues sí, hemos cerrado el formulario, pero no hemos detenido el proceso que pusimos en marcha.

Para que no se quede ningún proceso fuera de control, lo que hacemos en este caso, es codificar el evento Closed( ) del formulario, que se desencadena cuando el formulario es cerrado. En este evento, comprobamos si la hebra está todavía activa mediante su propiedad IsAlive; en el caso de que esta propiedad devuelva True, cancelamos la ejecución de la hebra. Ver Código fuente 494.

```
Private Sub Form1_Closed(ByVal sender As Object, ByVal e As System.EventArgs)
Handles MyBase.Closed

    ' si código de la aplicación no ha finalizado
    ' la hebra, debemos hacerlo antes de terminar
    ' la ejecución del programa
    If Not (oHebra Is Nothing) Then
        If oHebra.IsAlive Then
            oHebra.Abort()
        End If
    End If

End Sub
```

Código fuente 494

La Figura 289 muestra el formulario de este proyecto.

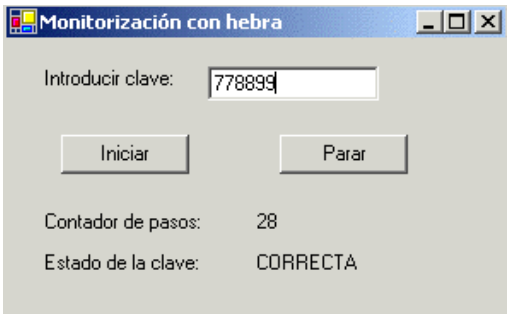


Figura 289. Proceso de monitorización en una hebra.

Inicios de aplicación con dos formularios, empleando hebras

Una situación ante la que nos podríamos encontrar, sería la de crear una aplicación que al iniciarse mostrara dos formularios independientes.

Este escenario, que en versiones anteriores de VB, consistía simplemente en instanciar y mostrar tales formularios de un proyecto, en VB.NET, debido a la arquitectura interna del entorno de ejecución, no es un proceso tan intuitivo como en un principio pudiera pensarse.

La aplicación IniciarVariosForm, descrita en este ejemplo está disponible haciendo clic aquí.

Si tenemos dos formularios en un proyecto: Form1 y Form2, e intentamos mostrarlos, por ejemplo, desde el método Main( ) de una clase que inicie la ejecución del programa, como vemos en el Código fuente 495, solamente se mostrarían por un instante cerrándose inmediatamente.

```
Public Class Inicio

    Public Shared Sub Main()

        ' esto no funciona
        Application.Run(New Form1())
        Application.Run(New Form2())

    End Sub

End Class
```

Código fuente 495

Si optamos por poner en el Form1, un botón que al ser pulsado, abra el Form2, habremos solucionado el problema sólo en parte, puesto que cuando cerremos Form1, también se cerrará Form2 sin que podamos evitarlo.

El motivo de este comportamiento reside en que ambos formularios se ejecutan en la denominada hebra principal de ejecución del programa.

Para solucionar este problema debemos hacer lo siguiente: crear dos nuevos procedimientos o métodos, en los que cada uno instancie una copia de cada uno de los formularios, y desde Main( ) crear dos hebras que ejecuten dichos procedimientos, y los pongan en marcha. De este modo, cada formulario se ejecutaría en su propia hebra independiente, y el cierre del primero no implicaría el cierre del segundo. Veámoslo en el Código fuente 496.

```
Imports System.Threading

Public Class Inicio

    Public Shared Sub Main()
        ' crear dos hebras que apunten a los métodos
        ' que lanzan los formularios
        Dim oHebraUno As New Thread(AddressOf LanzaPrimerForm)
        Dim oHebraDos As New Thread(AddressOf LanzaSegundoForm)
```

```
        ' iniciar las hebras
        oHebraUno.Start()
        oHebraDos.Start()
    End Sub

    Public Shared Sub LanzaPrimerForm()
        Application.Run(New Form1())
    End Sub

    Public Shared Sub LanzaSegundoForm()
        Application.Run(New Form2())
    End Sub

End Class
```

Código fuente 496

La Figura 290 muestra los formularios de este proyecto en ejecución.

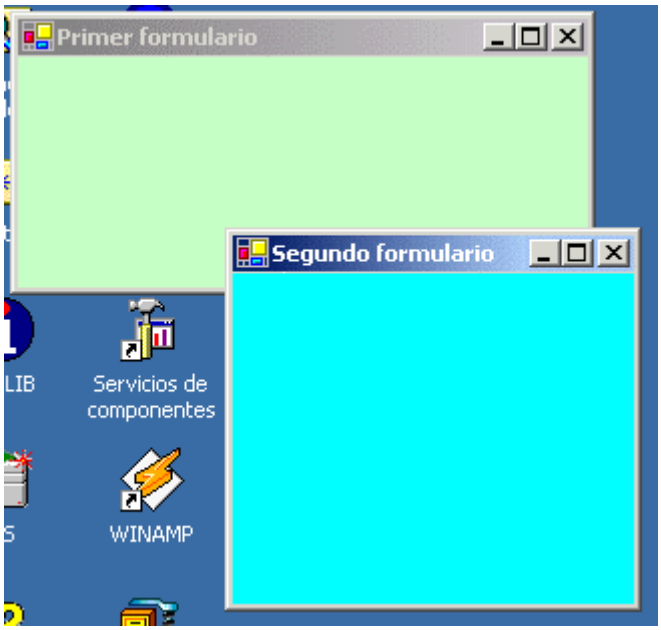


Figura 290. Formularios ejecutándose en hebras independientes.

33

Formularios de interfaz múltiple (MDI)

Aplicaciones de estilo SDI

Una aplicación de tipo o estilo SDI (Single Document Interface), Interfaz de Documento Sencillo, está compuesta fundamentalmente de un único formulario, a través del cual, el usuario realiza toda la interacción con el programa. Como ejemplos de este tipo de aplicación tenemos el Bloc de Notas o la Calculadora de Windows.

Un programa SDI puede tener más de un formulario, aunque no sea algo habitual. Cuando eso ocurre, los formularios se ejecutan independientemente, sin un elemento contenedor que los organice.

Aplicaciones de estilo MDI

Una aplicación de tipo o estilo MDI (Multiple Document Interface), Interfaz de Documento Múltiple, se compone de un formulario principal, también denominado formulario MDI, que actuará como contenedor de otros formularios (documentos) abiertos durante el transcurso del programa, denominados formularios hijos o secundarios MDI. Como ejemplos de este tipo de aplicación tenemos PowerPoint o Access.

A diferencia de lo que ocurría en versiones anteriores de VB, un formulario MDI admite los mismos controles que un formulario normal, aunque dada su orientación de formulario contenedor, se recomienda limitar los controles en un MDI a los estrictamente necesarios. El menú es el ejemplo más identificativo de control idóneo para un formulario MDI, ya que a través de sus opciones, podremos abrir los formularios hijos de la aplicación.

Seguidamente describiremos el proceso de creación de un proyecto que contenga un formulario MDI y dos formularios hijos, así como el comportamiento de estos últimos cuando son abiertos dentro del formulario padre MDI. Este ejemplo tiene el nombre MDIPru, y se debe hacer clic aquí para acceder al mismo.

Una vez creado el nuevo proyecto, cambiaremos el nombre del formulario por defecto a frmPrincipal. Para conseguir que este formulario tenga el comportamiento de un contenedor MDI, debemos asignar el valor True a su propiedad IsMdiContainer. También debemos establecer a este formulario como inicial en las propiedades del proyecto.

Ahora pasaremos a la creación de los formularios hijos del MDI. El primero, frmCarta, permite la escritura en un TextBox multilínea, cuyo contenido podremos grabar a un archivo en disco. La Figura 291 muestra este formulario.

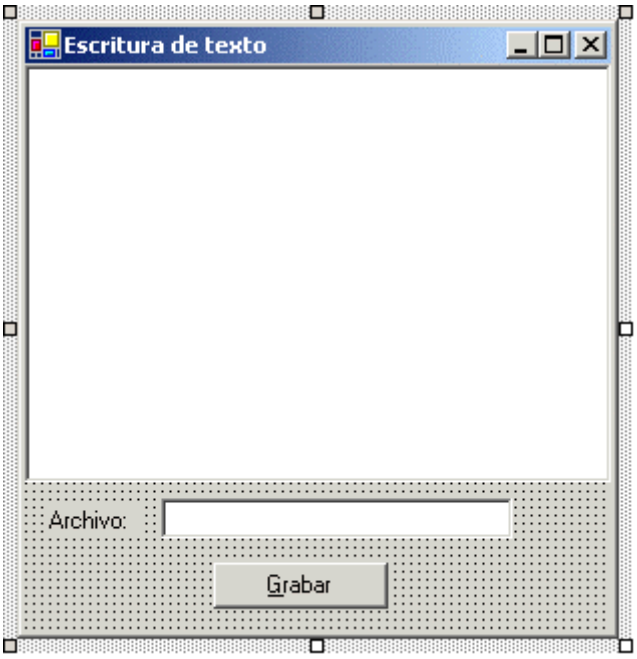


Figura 291. Formulario hijo de MDI para escribir un texto largo.

El código del botón que realiza la grabación del texto lo podemos ver en el Código fuente 497. Debemos importar el espacio de nombres System.IO, ya que en esta clase del formulario hacemos uso de los tipos File y StreamWriter.

```
Private Sub btnGrabar_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnGrabar.Click

    ' escribir en un archivo el contenido
    ' del TextBox
    Dim oEscritor As StreamWriter
    oEscritor = File.CreateText(Me.txtArchivo.Text)
    oEscritor.Write(Me.txtCarta.Text)
    oEscritor.Close()

End Sub
```

Código fuente 497

El otro formulario hijo, frmInfo, muestra la fecha y hora actual; esta última es actualizada a través del control Timer tmrTiempo. Ver la Figura 292.

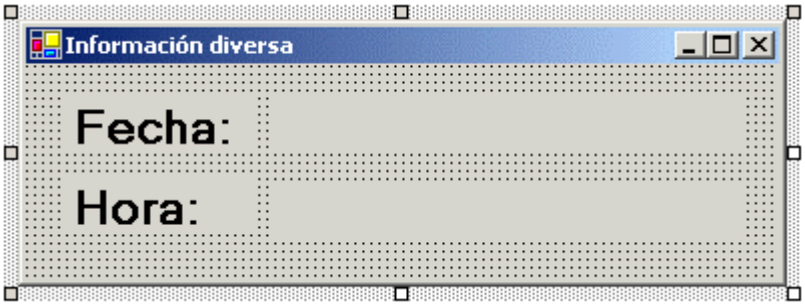


Figura 292. Formulario hijo de MDI para mostrar fecha y hora actuales.

El Código fuente 498 muestra las instrucciones que se ejecutan en el evento Tick del control Timer.

```
Private Sub tmrTiempo_Tick(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles tmrTiempo.Tick

    Dim dtFecha As Date
    dtFecha = DateTime.Today
    Dim dtHora As Date
    dtHora = DateTime.Now

    Me.lblFecha.Text = dtFecha.ToString("d/MMM/yyyy")
    Me.lblHora.Text = dtHora.ToString("h:m:s")

End Sub
```

Código fuente 498

El siguiente paso consiste en crear un menú para poder abrir los formularios hijos a través de sus opciones. Ver Figura 293.

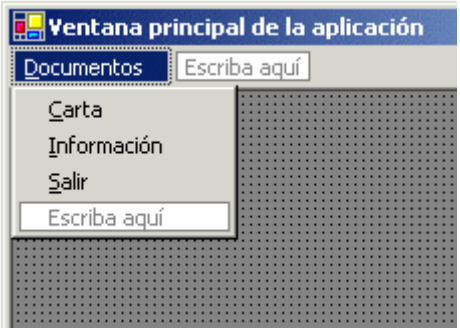


Figura 293. Menú del formulario MDI.

En las opciones Carta e Información del menú, instanciaremos un objeto del formulario correspondiente, teniendo en cuenta que para conseguir que dichos formularios se comporten como hijos del MDI, debemos asignar a su propiedad MdiParent, la instancia actual del formulario en ejecución, es decir, Me. Veamos este punto en el Código fuente 499.

```
Private Sub mnuCarta_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles mnuCarta.Click

    Dim ofrmCarta As New frmCarta()
    ' con la siguiente línea conseguimos que el
    ' formulario se comporte como hijo del actual
    ofrmCarta.MdiParent = Me
    ofrmCarta.Show()

End Sub

Private Sub mnuInformacion_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles mnuInformacion.Click

    Dim ofrmInfo As New frmInfo()
    ' con la siguiente línea conseguimos que el
    ' formulario se comporte como hijo del actual
    ofrmInfo.MdiParent = Me
    ofrmInfo.Show()

End Sub
```

Código fuente 499

En la Figura 294 mostramos el formulario MDI en ejecución, conteniendo a los formularios hijos dependientes.

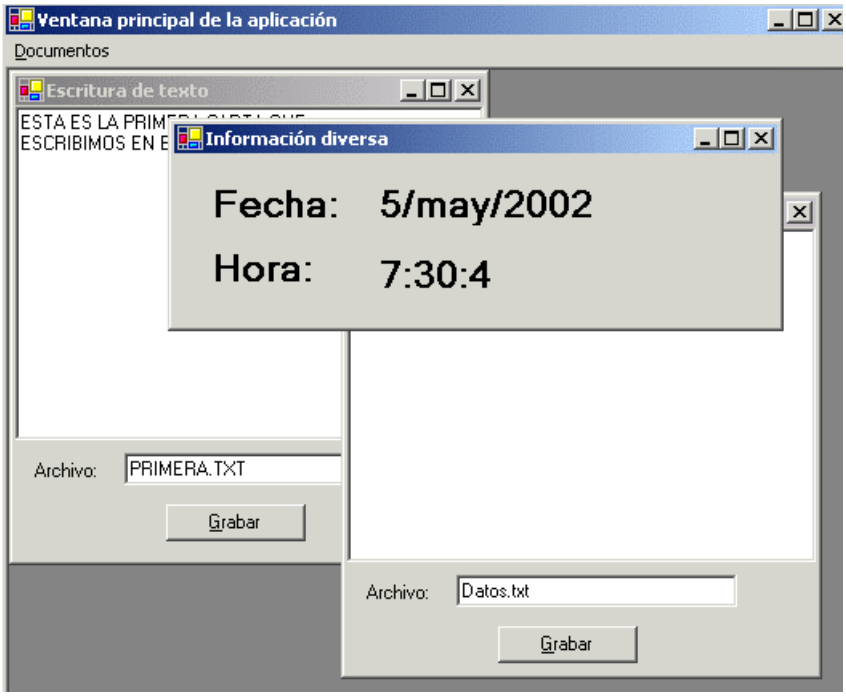


Figura 294. Aplicación MDI en ejecución.

Creación de menús de tipo Ventana, en formularios MDI

Es probable que el lector haya observado, en algunas aplicaciones Windows de tipo MDI, que existe en la barra de menús de la ventana principal, un menú con el nombre Ventana o Window (depende del

idioma del programa), que nos muestra los nombres de los formularios hijos abiertos, permitiéndonos cambiar de formulario activo al seleccionar una de esas opciones.

En nuestras aplicaciones MDI también podemos disponer de un menú de este tipo, añadiendo una nueva opción al menú principal del formulario MDI, y asignando a su propiedad MdiList el valor True.

Adicionalmente, y para darle un aspecto más profesional a este menú, podemos añadir los MenuItem correspondientes a la organización de los formularios hijos en Cascada, Mosaico Horizontal, etc. Para organizar los formularios abiertos en la aplicación en alguno de estos modos, deberemos ejecutar el método LayoutMdi( ) del formulario MDI, pasándole como parámetro uno de los valores correspondiente a la enumeración MdiLayout. El Código fuente 500 muestra las opciones correspondientes a la organización en cascada y en mosaico horizontal de nuestro ejemplo.

```
Private Sub mnuCascada_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles mnuCascada.Click

    Me.LayoutMdi(MdiLayout.Cascade)

End Sub

Private Sub mnuHorizontal_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles mnuHorizontal.Click

    Me.LayoutMdi(MdiLayout.TileHorizontal)

End Sub
```

Código fuente 500

La Figura 295 muestra el mencionado menú Ventana de este proyecto, en cual contiene en este caso los nombres de los formularios abiertos que acaban de ser organizados en mosaico vertical.

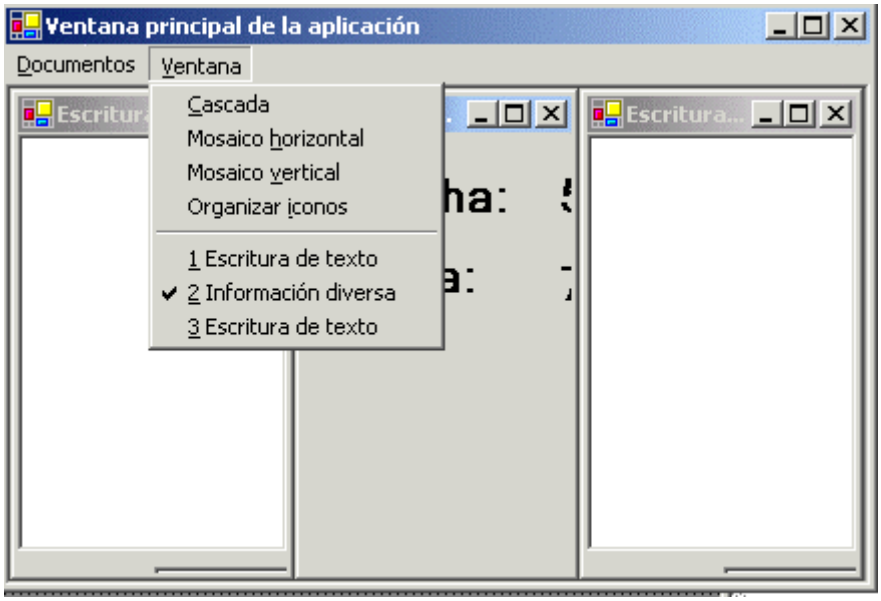


Figura 295. Menú ventana en formulario MDI.

Bloqueo de opciones de menú en formularios MDI

En la aplicación de ejemplo que estamos desarrollando, podemos abrir tantas copias de los formularios hijos como necesitemos.

Respecto al formulario que nos permite escribir un texto para grabar a un archivo, es útil poder tener varios formularios de este tipo, ya que podemos trabajar con diversos archivos a la vez.

Del formulario hijo que muestra la fecha y hora actual sin embargo, tener más de una copia no parece algo muy lógico, ya que se trata simplemente de disponer de una información visualizada, y repetirla a través de la apertura de varios formularios iguales no tiene mucho sentido.

Para conseguir que de un determinado formulario hijo sólo podamos abrir una instancia, debemos hacer dos cosas: en primer lugar, en el manipulador de evento correspondiente a la opción de menú que abre dicho formulario, asignaremos False a la propiedad True de la mencionada opción de menú. Veámoslo en el Código fuente 501.

```
Private Sub mnuInformacion_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles mnuInformacion.Click

    ' deshabilitamos esta opción de menú
    Me.mnuInformacion.Enabled = False   ' <-----

    Dim ofrmInfo As New frmInfo()
    ' con la siguiente línea conseguimos que el
    ' formulario se comporte como hijo del actual
    ofrmInfo.MdiParent = Me
    ofrmInfo.Show()

End Sub
```

Código fuente 501

En segundo lugar, dentro del código del formulario hijo, en nuestro caso frmInfo, debemos escribir el manipulador para el evento Closed del formulario. Este evento se produce cuando se ha cerrado el formulario, por lo que desde aquí volveremos a activar la opción de menú del formulario padre, que habíamos deshabilitado.

Para acceder desde un formulario hijo a su MDI contenedor, disponemos de la propiedad MdiParent, que nos devuelve una referencia de dicho formulario padre. Observe el lector en el Código fuente 502, cómo además de utilizar la mencionada propiedad, la potencia de la función CType( ) nos permite en una sola línea de código, llevar a cabo esta acción.

```
' al cerrar este formulario, activamos de nuevo
' la opción de menú del formulario padre que
' permite crear instancias de este formulario
Private Sub frmInfo_Closed(ByVal sender As Object, ByVal e As System.EventArgs)
Handles MyBase.Closed

    ' utilizando la función CType(), moldeamos
    ' la propiedad MdiParent del formulario al tipo
    ' correspondiente a la clase del formulario MDI;
    ' con ello obtenemos acceso a sus miembros, y en
    ' particular a la opción de menú que necesitamos
```

```
    ' habilitar
    CType(Me.MdiParent, frmPrincipal).mnuInformacion.Enabled = True

End Sub
```

Código fuente 502

La Figura 296 muestra el resultado al ejecutar. Mientras que el formulario de información esté abierto, su opción de menú en el MDI estará deshabilitada.

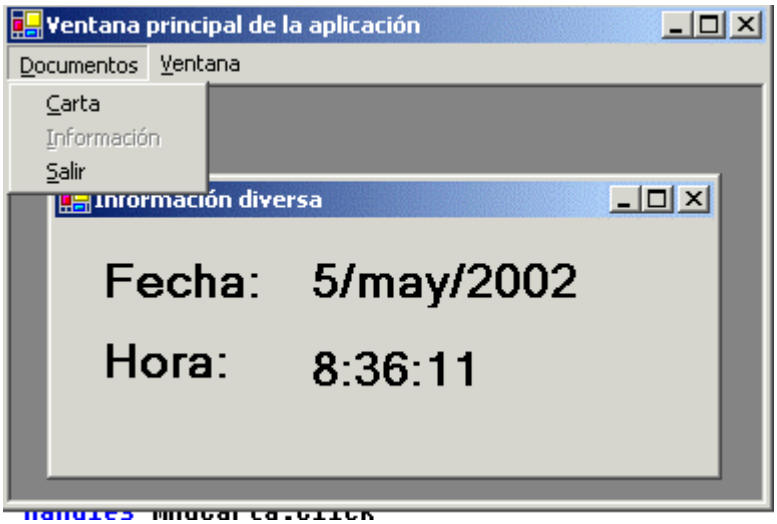


Figura 296. Opción de formulario hijo deshabilitada.

Recorrer los formularios hijos de un MDI

La clase Form tiene la propiedad MdiChildren, que devuelve un array con todos los formularios hijos abiertos hasta el momento.

Esto nos permite recorrer todo este conjunto de formularios para realizar operaciones con alguno de ellos o todos.

El Código fuente 503 muestra un ejemplo de uso de esta propiedad, en el que mostramos el título de cada formulario hijo, y además, cambiamos su color de fondo.

```
Dim oFormHijos() As Form
oFormHijos = Me.MdiChildren

Dim oForm As Form
For Each oForm In oFormHijos
    MessageBox.Show("Título de ventana: " & oForm.Text)
    oForm.BackColor = Color.Beige
Next
```

Código fuente 503

Comportamiento No Modal (Modeless) de formularios

Un formulario de comportamiento no modal, permite el libre cambio de foco entre el resto de formularios de la aplicación.

Una clara muestra la hemos visto en el proyecto de ejemplo realizado durante los últimos apartados del texto. En dicha aplicación, podíamos abrir varios formularios hijos dentro del formulario MDI principal, y pasar de uno a otro sin restricciones.

Otra característica de los formularios no modales reside en que una vez creados y visualizados, el resto del código de la aplicación continúa su ejecución. Ver Código fuente 504.

```
Dim ofrmCarta As New frmCarta()
' crear formulario hijo de un mdi
ofrmCarta.MdiParent = Me
ofrmCarta.Show()

' después de mostrar el formulario hijo
' se muestra a continuación este mensaje
MessageBox.Show("Se acaba de abrir un formulario hijo")
```

Código fuente 504

Comportamiento Modal de formularios

Como contrapartida al anterior apartado tenemos los formularios de comportamiento modal, también denominados cuadros o ventanas de diálogo.

Un formulario modal, al ser visualizado, bloquea el paso a otros formularios de la aplicación hasta que no es cerrado (aceptado o completado) por el usuario.

Como ejemplo de estos formularios se acompaña el proyecto FormDialogos (hacer clic aquí para acceder al ejemplo), del que pasamos a describir su proceso de creación.

Este proyecto contiene un formulario MDI llamado frmPrincipal, y uno hijo con el nombre frmHijo, que abrimos mediante una opción de menú; la creación de este tipo de formularios se ha descrito en apartados anteriores.

A continuación añadimos un nuevo formulario al proyecto con el nombre frmDialogo, que también abriremos a través de la correspondiente opción de menú del formulario MDI.

Para que este formulario tenga un comportamiento modal, debemos mostrarlo ejecutando el método ShowDialog( ) de la clase Form. En el Código fuente 505 tenemos las instrucciones necesarias. Observe también el lector, cómo hasta que el formulario de diálogo no es cerrado, no se mostrará el mensaje que hay a continuación de la llamada a ShowDialog( ). Si además intentamos pasar al formulario hijo, en el caso de que esté abierto, no podremos.

```
Private Sub mnuDialogo_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles mnuDialogo.Click

    ' instanciar el formulario que mostraremos como un diálogo
    Dim ofrmDialogo As New frmDialogo()
```

```
    ' dar una posición al formulario
    ofrmDialogo.StartPosition = FormStartPosition.CenterParent
    ' mostrarlo de forma modal, como cuadro de diálogo
    ofrmDialogo.ShowDialog()

    MessageBox.Show("Se ha cerrado el diálogo")

End Sub
```

Código fuente 505

Para cerrar un formulario modal podemos, al igual que para cualquier formulario, ejecutar su método Close( ). No obstante, un formulario de diálogo suele proporcionar, aunque esto no es obligatorio, los típicos botones para aceptar, cancelar, reintentar, etc.; de modo que una vez cerrado el formulario, podamos averiguar qué botón pulsó el usuario.

Podemos proporcionar este comportamiento en nuestros formularios modales, asignando a la propiedad DialogResult de la clase Form, uno de los valores del tipo enumerado DialogResult. Esto tendrá como efecto adicional el cierre del cuadro de diálogo.

Por lo tanto, vamos a añadir a nuestro formulario frmDialogo, dos controles Button: btnAceptar y btnCancelar, en los que escribiremos las instrucciones del Código fuente 506.

```
Private Sub btnAceptar_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnAceptar.Click

    ' asignar un valor a esta propiedad,
    ' cierra al mismo tiempo el formulario
    Me.DialogResult = DialogResult.OK

End Sub

Private Sub btnCancelar_Click(ByVal sender As Object, ByVal e As System.EventArgs)
Handles btnCancelar.Click

    ' asignar un valor a esta propiedad,
    ' cierra al mismo tiempo el formulario
    Me.DialogResult = DialogResult.Cancel

End Sub
```

Código fuente 506

Como ayuda en la construcción de formularios modales de diálogo, la clase Form dispone de las propiedades AcceptButton y CancelButton, a las que podemos asignar sendos controles Button que serán ejecutados al pulsar las teclas [INTRO] y [ESCAPE] respectivamente.

Esto es lo que haremos en el formulario frmDialogo, asignando a AcceptButton el control btnAceptar, y en CancelButton asignaremos btnCancelar.

Finalmente, en el evento de la opción de menú que abre este formulario modal, correspondiente a frmPrincipal, añadiremos, tras la llamada a ShowDialog( ), el código que comprobará el resultado de la ejecución del formulario de diálogo. Ver el Código fuente 507.

```
Private Sub mnuDialogo_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles mnuDialogo.Click

    ' instanciar el formulario que mostraremos como un diálogo
    Dim ofrmDialogo As New frmDialogo()
    ' dar una posición al formulario
    ofrmDialogo.StartPosition = FormStartPosition.CenterParent
    ' mostrarlo de forma modal, como cuadro de diálogo
    ofrmDialogo.ShowDialog()

    ' comprobar lo que ha hecho el usuario
    ' en el cuadro de diálogo
    Dim Resultado As DialogResult
    Resultado = ofrmDialogo.DialogResult
    If Resultado = DialogResult.OK Then
        MessageBox.Show("Datos del diálogo: " & _
            ofrmDialogo.txtNombre.Text & " " & _
            ofrmDialogo.txtApellidos.Text)

    Else
        MessageBox.Show("Se ha cancelado el diálogo")
    End If

End Sub
```

Código fuente 507

La Figura 297 muestra el programa en ejecución. Como puede comprobar el lector, el formulario modal, debido a su comportamiento, no se encuentra limitado a los bordes del formulario MDI; pero depende de este último, ya que si intentamos pasar el foco a un formulario hijo, no podremos.

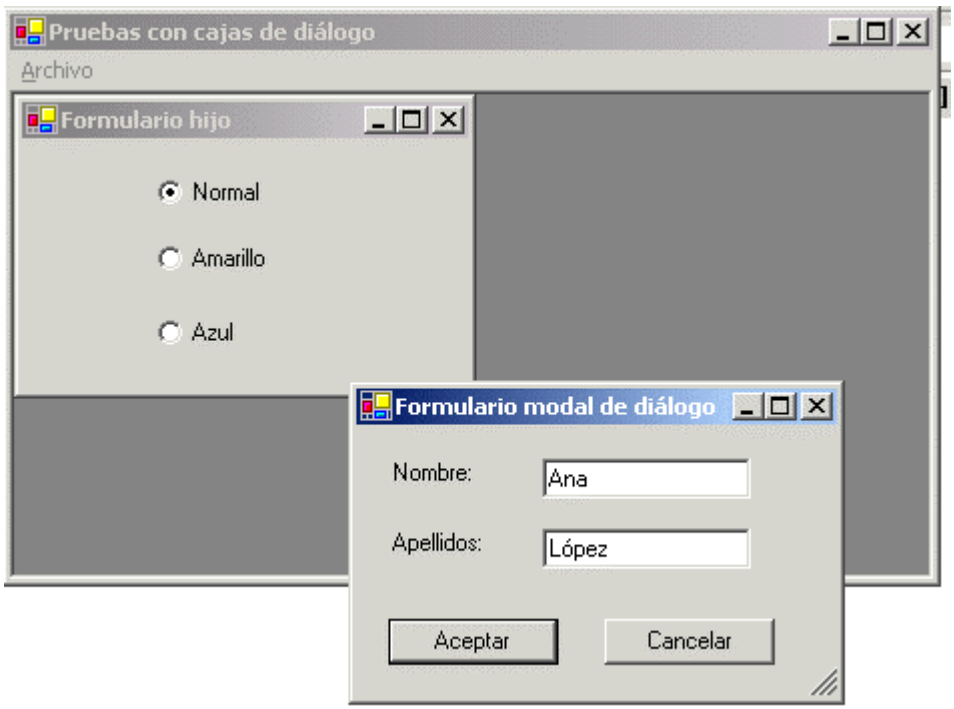


Figura 297. Formulario modal de diálogo en ejecución.

Controles de cuadros de diálogo del sistema

Del conjunto de controles que nos ofrece la ventana Cuadro de herramientas del IDE, existe un grupo que nos permite el acceso a los cuadros de diálogo estándar del sistema operativo, esto es, los cuadros de selección de color, tipo de letra o fuente, apertura-grabación de archivo, etc.

Para ilustrar el uso de algunos de estos controles, vamos a crear un proyecto de ejemplo con el nombre DialogosSistema (hacer clic aquí para acceder al ejemplo), en el que describiremos su modo de uso en los aspectos de diseño y codificación.

Crearemos pues, un nuevo proyecto de tipo aplicación Windows, y en su formulario, insertaremos un menú, añadiendo las siguientes opciones: Abrir, Guardar, Color y Fuente. Cada opción mostrará un tipo de diálogo del sistema.

Seguidamente insertaremos un TextBox, que acoplaremos con la propiedad Dock a todo el espacio del formulario, y que nos servirá como base para las operaciones a realizar mediante los controles de diálogo. La Figura 298 muestra el aspecto de este formulario.

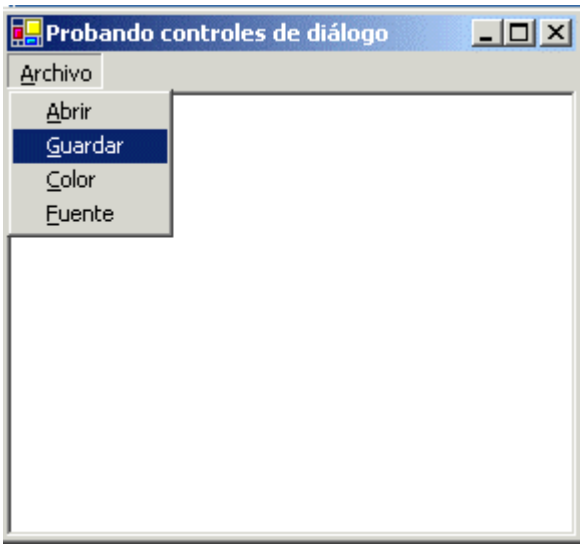


Figura 298. Formulario de pruebas para controles de diálogo estándar.

Una vez dibujado un control de cuadro de diálogo en el formulario, dicho control quedará ubicado en el panel de controles especiales, al igual que sucede con los menús. Para abrir un control de este tipo en tiempo de ejecución, emplearemos su método ShowDialog( ).

A continuación describiremos cada uno de los controles de diálogo utilizados en este ejemplo.

ColorDialog

Este control muestra el cuadro de diálogo del sistema para la selección de colores. Entre sus propiedades podemos destacar las siguientes.

- **Color**. Contiene un tipo de la estructura Color, que nos permite obtener el color seleccionado por el usuario mediante este cuadro de diálogo, para poder aplicarlo sobre alguno de los elementos del formulario.

- **AllowFullOpen**. Contiene un valor lógico que permite habilitar y deshabilitar el botón que muestra el conjunto de colores personalizados del cuadro de diálogo de selección de colores.

Al seleccionar en el formulario, la opción de menú Color, ejecutaremos el Código fuente 508 que nos permitirá, utilizando el control dlgColor, de tipo ColorDialog, elegir un color y aplicarlo a la propiedad BackColor, del control TextBox.

```
Private Sub mnuColor_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles mnuColor.Click

    Me.dlgColor.ShowDialog()
    Me.txtTexto.BackColor = Me.dlgColor.Color

End Sub
```

Código fuente 508

La Figura 299 muestra esta aplicación con el formulario y el cuadro de selección de color abiertos.

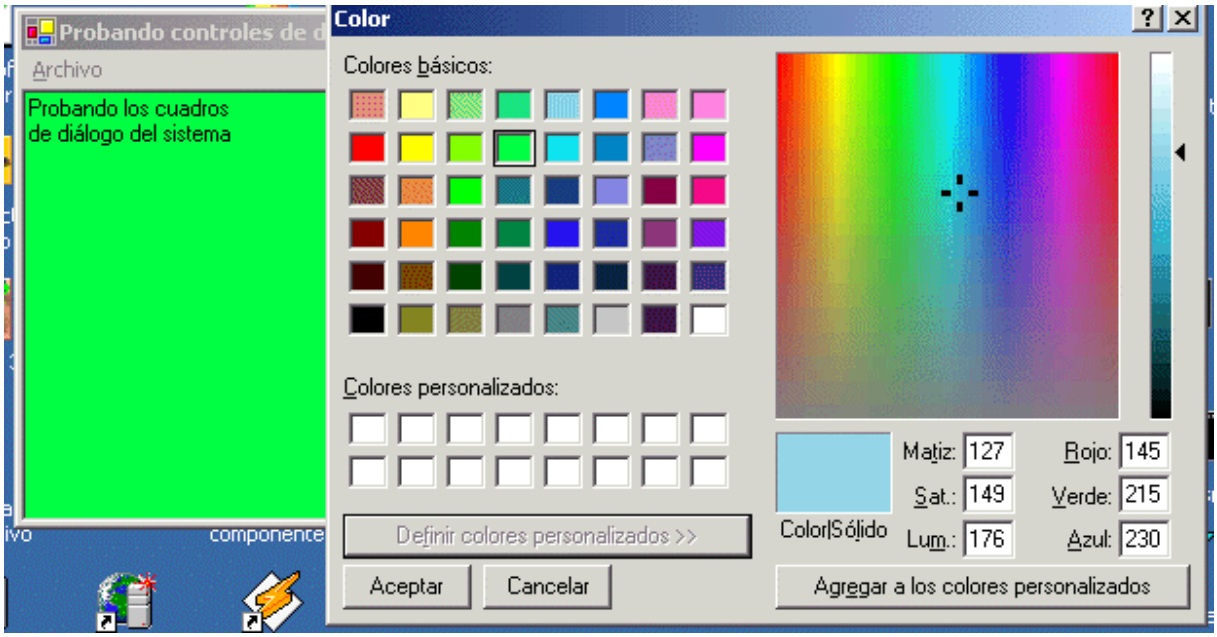


Figura 299. Cuadro de diálogo estándar para selección de colores.

FontDialog

Este control muestra el cuadro de diálogo del sistema para la selección del tipo de fuente. Entre sus propiedades podemos destacar las siguientes.

- **Font**. Contiene un tipo de la clase Font. Una vez seleccionada una fuente por el usuario en el cuadro de diálogo, podremos cambiar el fuente de los controles del formulario.

- **ShowApply**. Contiene un valor lógico que permite mostrar-ocultar el botón Aplicar, que nos permitirá asignar el tipo de letra sin cerrar el diálogo. Al pulsar este botón se desencadenará el

evento Apply de este control de diálogo, en el que podremos escribir el código necesario para aplicar la nueva fuente seleccionada.

Al seleccionar en el formulario la opción de menú Fuente, ejecutaremos el Código fuente 509 que nos permitirá, utilizando el control dlgFuente, de tipo FontDialog, elegir un tipo de letra, y aplicarlo a la propiedad Font del control TextBox; con la particularidad de que el cambio de letra lo haremos tanto al pulsar el botón Aceptar, como Aplicar del cuadro de diálogo.

```
' al hacer clic en este menú, mostramos el cuadro
' de selección de fuente
Private Sub mnuFuente_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles mnuFuente.Click

    Me.dlgFuente.ShowApply = True
    Me.dlgFuente.ShowDialog()
    Me.AplicarFuente()

End Sub

' este método cambia el fuente del TextBox
Private Sub AplicarFuente()

    Me.txtTexto.Font = Me.dlgFuente.Font

End Sub

' al pulsar el botón Aplicar del diálogo de
' selección de fuente, se produce este evento
Private Sub dlgFuente_Apply(ByVal sender As Object, ByVal e As System.EventArgs)
Handles dlgFuente.Apply

    Me.AplicarFuente()

End Sub
```

Código fuente 509

La Figura 300 muestra el cambio de fuente sobre el texto del formulario.

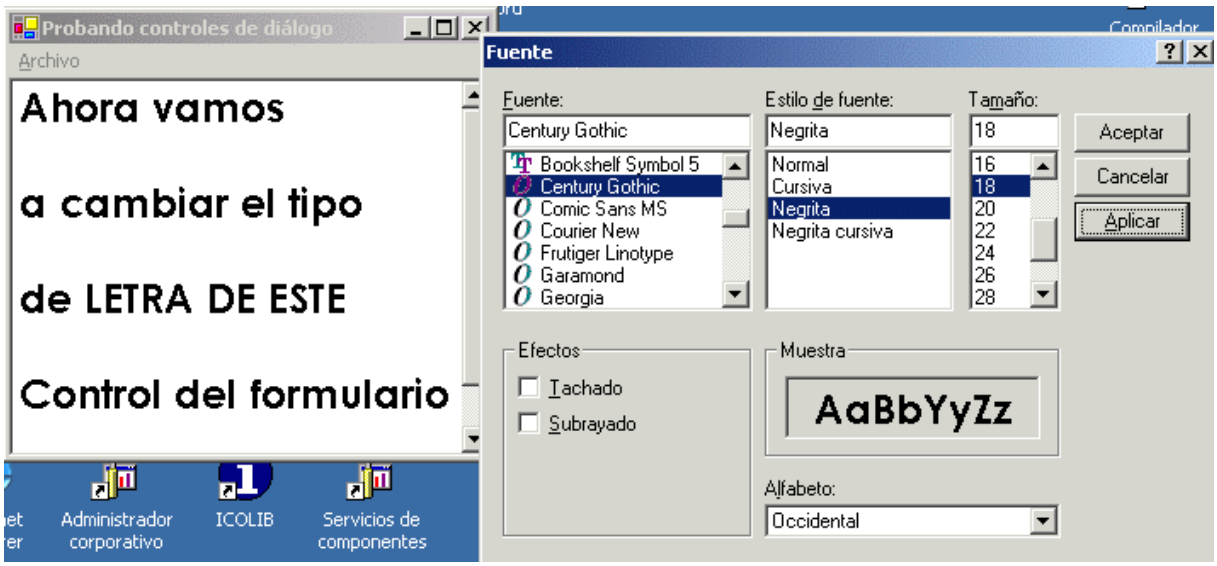


Figura 300. Cuadro de diálogo del sistema para selección de tipo de letra.

SaveFileDialog

Este control muestra el cuadro de diálogo del sistema, mediante el que escribimos un nombre de archivo, y elegimos un directorio, sobre el cual grabaremos información.

Es importante precisar que el control no se ocupa del proceso de grabación de datos; su cometido es el permitirnos navegar por la estructura del sistema de archivos del equipo, y la selección de un nombre para el archivo a grabar.

Entre las propiedades del control, podemos destacar las siguientes.

- **Title**. Contiene una cadena con el título que aparecerá en el cuadro de diálogo.
- **InitialDirectory**. Ruta inicial que mostrará el control al abrirse.
- **Filter**. Cadena con el tipo de archivos que mostrará el cuadro de diálogo al navegar por el sistema de archivos. El formato de esta cadena es el siguiente: NombreArchivo (*.Extensión)|*.Extensión; pudiendo situar varios filtros separados por el carácter de barra vertical (|).
- **FilterIndex**. Número que corresponde a alguno de los tipos de archivo establecidos en la propiedad Filter.
- **FileName**. Nombre del archivo en el que se realizará la escritura
- **DefaultExt**. Cadena con la extensión por defecto a aplicar sobre el nombre del archivo.
- **CheckFileExists**. Valor lógico que nos permite comprobar si el archivo sobre el que vamos a grabar ya existe.
- **ValidateNames**. Valor lógico que comprobará si el nombre de archivo proporcionado contiene caracteres especiales, es decir, si se trata de un nombre válido.

Al seleccionar en el formulario la opción de menú Grabar, ejecutaremos el Código fuente 510, que nos permitirá, utilizando el control dlgGrabar, de tipo SaveFileDialog, seleccionar el nombre de un archivo, y grabar el TextBox del formulario sobre el mismo, mediante un StreamWriter.

```
Private Sub mnuGuardar_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles mnuGuardar.Click

    ' configurar por código el diálogo de grabación de archivos
    Me.dlgGrabar.Filter = "Documento (*.doc)|*.doc|Texto (*.txt)|*.txt"
    Me.dlgGrabar.FilterIndex = 2
    Me.dlgGrabar.ValidateNames = True

    Me.dlgGrabar.ShowDialog()   ' abrir el cuadro de diálogo

    ' si todo es correcto, escribir mediante un objeto Stream
    ' el contenido del TextBox en el archivo indicado por
    ' las propiedades del cuadro de diálogo
    Dim swEscritor As IO.StreamWriter
    swEscritor = New IO.StreamWriter(Me.dlgGrabar.FileName)
    swEscritor.Write(Me.txtTexto.Text)
    swEscritor.Close()

    MessageBox.Show("Texto grabado en archivo")
```

```
End Sub
```

Código fuente 510

La Figura 301 muestra el cuadro de dialogo de grabación de archivo.

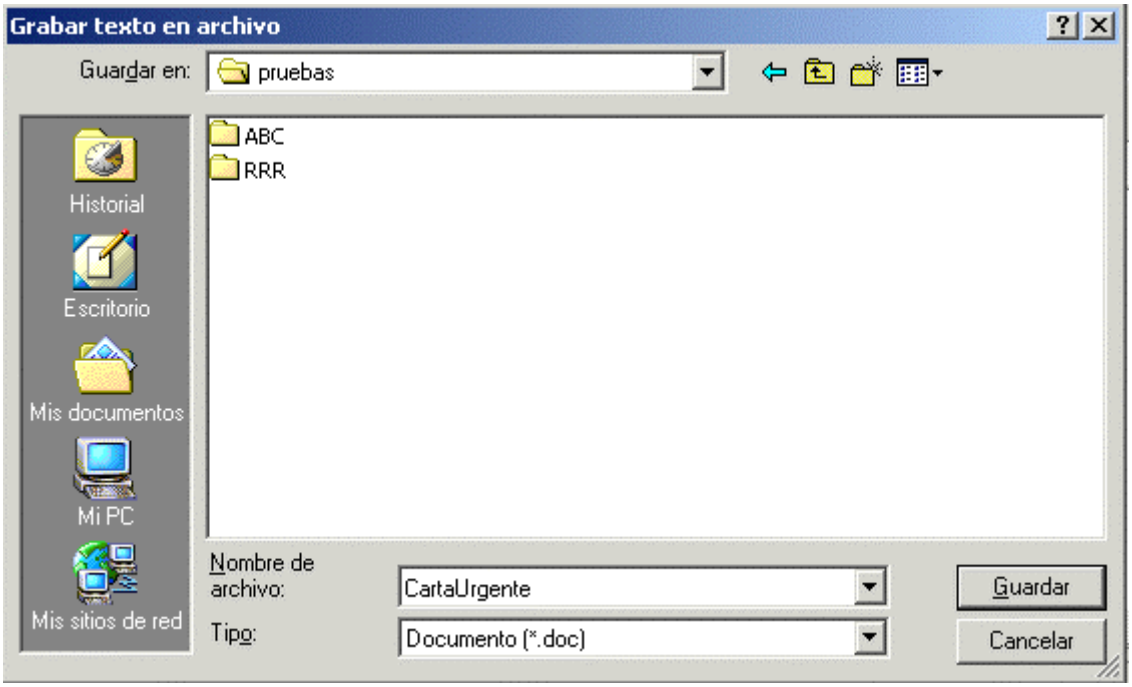


Figura 301. Cuadro de diálogo para la grabación de archivos.

OpenFileDialog

Este control muestra el cuadro de diálogo del sistema, mediante el que seleccionamos un archivo para poder abrirlo posteriormente, y realizar sobre el mismo operaciones de lectura-escritura.

Al igual que en el control anterior, la lectura y escritura de información es algo que deberemos realizar por código, una vez que hayamos elegido el archivo mediante este cuadro de diálogo

Las propiedades de este control son prácticamente las mismas que las de SaveFileDialog, con algunas excepciones como las siguientes.

- **Multiselect**. Contiene un valor lógico, que nos permitirá la selección de múltiples archivos.
- **ShowReadOnly**. Permite mostrar la casilla de verificación para mostrar los archivos de sólo lectura.
- **ReadOnlyChex**. Permite obtener y establecer el valor para la casilla de verificación de sólo lectura del cuadro de diálogo.

Al seleccionar en el formulario la opción de menú Abrir, ejecutaremos el Código fuente 511, que nos permitirá, utilizando el control dlgAbrir, de tipo OpenFileDialog, seleccionar un archivo existente, y pasar su contenido al TextBox del formulario, utilizando un StreamReader.

```
Private Sub mnuAbrir_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles mnuAbrir.Click

    ' configurar el cuadro de diálogo por código
    Me.dlgAbrir.Title = "Seleccionar archivo a leer"
    Me.dlgAbrir.InitialDirectory = "C:\CUBO"
    Me.dlgAbrir.Filter = "Código fuente (*.vb)|*.vb|Texto (*.txt)|*.txt"

    ' abrir el diálogo
    Me.dlgAbrir.ShowDialog()

    ' si se han seleccionado varios archivos
    ' mostrar su nombre
    If Me.dlgAbrir.FileNames.Length > 1 Then
        Dim sArchivo As String
        For Each sArchivo In Me.dlgAbrir.FileNames
            MessageBox.Show("Archivo seleccionado: " & sArchivo)
        Next
    End If

    ' abrir el primer archivo con un Stream
    ' y volcarlo al TextBox
    Dim srLector As New IO.StreamReader(Me.dlgAbrir.FileName)
    Me.txtTexto.Text = srLector.ReadToEnd()

End Sub
```

Código fuente 511

La Figura 302 muestra este cuadro de diálogo en ejecución

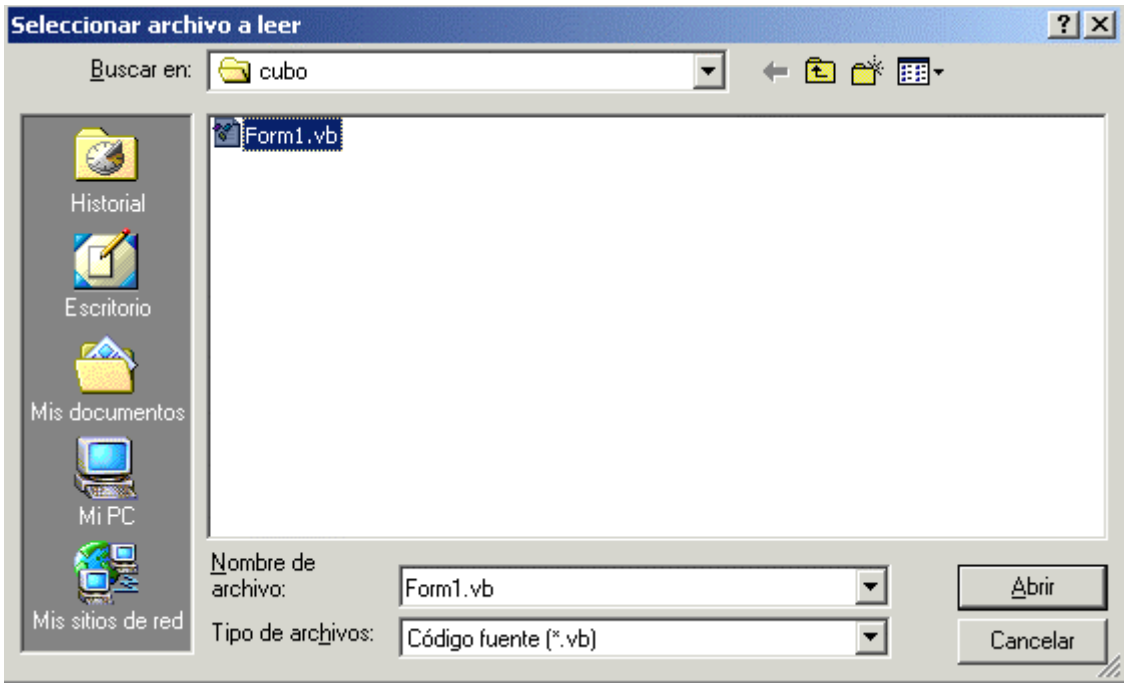


Figura 302. Cuadro de diálogo de apertura de archivos.

34

Formularios dependientes y controles avanzados

Formularios dependientes y fijos en primer plano

Un formulario dependiente, también denominado owned form, consiste en un formulario que es abierto por otro, denominado formulario dueño (owner form), permaneciendo ambos abiertos, sin que el formulario dependiente requiera ser cerrado, en el caso de que necesitemos pasar el foco al formulario dueño.

No necesitamos ir muy lejos para encontrar un ejemplo de este tipo de formularios, en el propio IDE de Visual Studio tenemos muchos casos. En la ventana del editor de código, cuando abrimos la ventana de búsqueda de texto tecleando [CTRL + F], quedan ambas visibles en todo momento, aunque no efectuemos ninguna búsqueda y el foco lo tengamos en el editor de código. En este caso, la ventana Buscar es dependiente de la ventana del editor de código. Ver Figura 303.

Este comportamiento en los formularios contrasta claramente con el que tienen los formularios de diálogo, en los cuales, hasta que no es cerrado el diálogo, no podemos retornar el foco a la ventana que abrió el diálogo.

En versiones anteriores de VB, conseguir un formulario con tal funcionamiento era una ardua tarea, que requería de conocimientos sobre el API de Windows; sin embargo, el nuevo motor de formularios de la plataforma .NET, nos permite a través de una serie de propiedades, que la configuración de formularios dependientes sea un trabajo realmente fácil y rápido de conseguir.

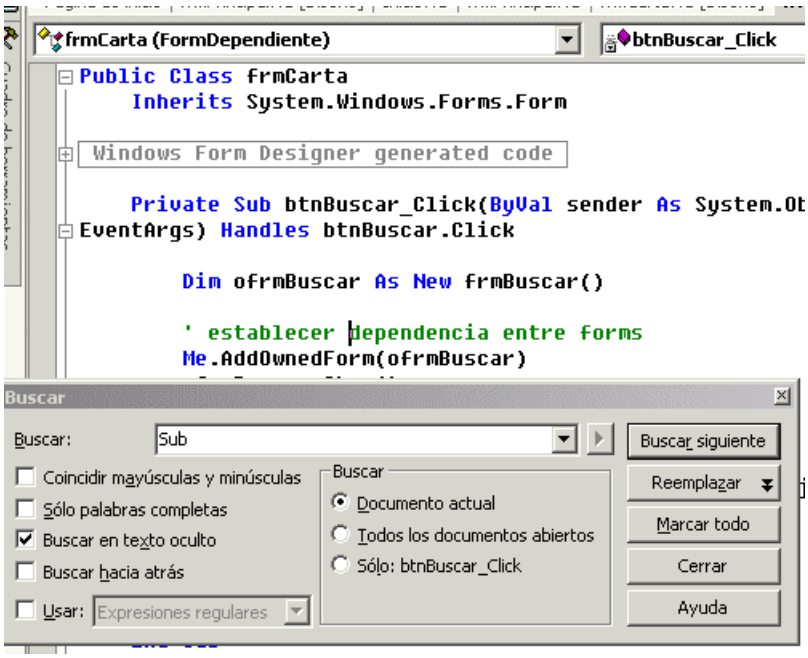


Figura 303. Editor de código de Visual Studio con ventana de búsqueda.

Por otra parte, un formulario fijo en primer plano, también denominado topmost form, consiste en un formulario que siempre aparece en primer plano respecto al resto de formularios de la aplicación. Se trata de una ligera variación de comportamiento respecto al formulario dependiente; mientras que este último, en algunas ocasiones puede ser tapado por otros formularios del programa, un formulario topmost siempre permanece visible en primer plano.

Para ilustrar el modo de creación y funcionamiento de este tipos de formularios, se acompaña el proyecto de ejemplo FormDependiente (hacer clic aquí para acceder al ejemplo), del que comentaremos los pasos principales para su creación.

Una vez creado este proyecto, eliminaremos su formulario por defecto, y añadiremos el formulario frmPrincipal, que configuraremos como contenedor MDI, y al que añadiremos un menú que nos permitirá abrir un formulario hijo para escribir un texto, y otro de diálogo para mostrar un literal. La Figura 304 muestra esta ventana MDI de la aplicación.

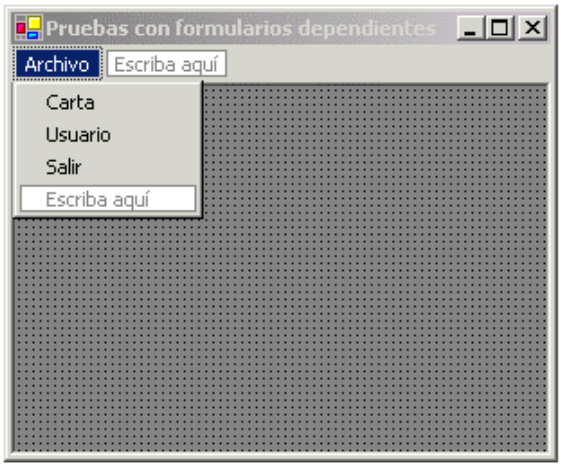


Figura 304. Formulario MDI para el ejemplo con formularios dependientes.

El siguiente paso consistirá en crear el formulario frmCarta, que utilizaremos para abrir los formularios dependientes que crearemos posteriormente en este proyecto. La Figura 305 muestra este formulario.

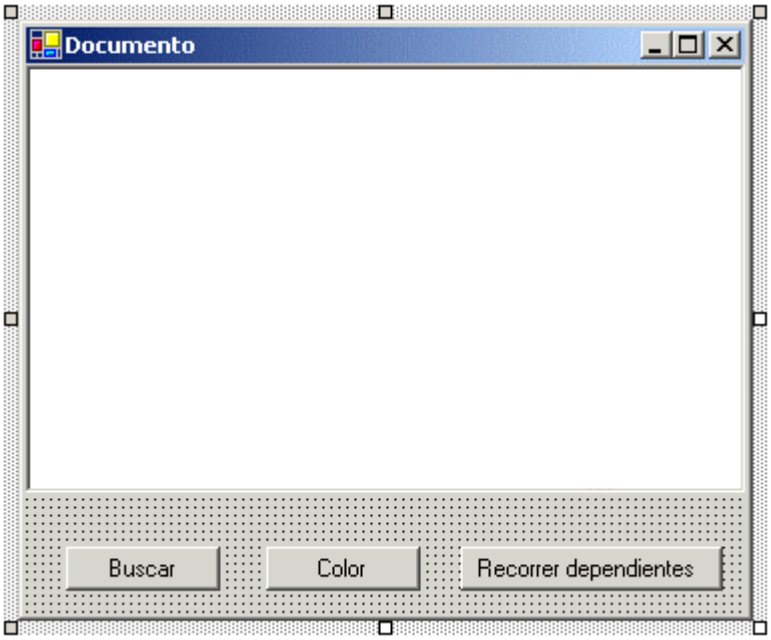


Figura 305. Formulario para escribir un texto y abrir formularios dependientes.

El Código fuente 512 muestra el código del menú de frmPrincipal que instancia este objeto y lo muestra como formulario hijo del MDI.

```
Private Sub mnuCarta_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles mnuCarta.Click

    ' este formulario se abre como hijo del MDI
    Dim ofrmCarta As New frmCarta()
    ofrmCarta.MdiParent = Me
    ofrmCarta.Show()

End Sub
```

Código fuente 512

A continuación agregaremos al proyecto el formulario frmBuscar. Este formulario actuará como dependiente de frmCarta, permitiéndonos buscar una cadena en el TextBox de este último. La Figura 306 muestra el aspecto de frmBuscar. Aunque no sería necesario, para adaptarlo mejor a su funcionamiento, hemos variado mediante la propiedad FormBorderStyle, el estilo de su borde a ventana de herramientas con el valor FixedToolWindow.

Para conseguir que frmBuscar se comporte como formulario dependiente, al pulsar dentro de frmCarta el botón Buscar, instanciaremos un objeto frmBuscar, añadiéndolo a la colección de formularios dependientes de frmCarta mediante el método AddOwnedForm( ), de la clase Form. El Código fuente 513 muestra el código del botón Buscar en el formulario frmCarta.

Figura 306. Formulario dependiente frmBuscar.

```
Private Sub btnBuscar_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnBuscar.Click

    ' crear un objeto frmBuscar
    Dim ofrmBuscar As New frmBuscar()

    ' establecer dependencia entre forms
    Me.AddOwnedForm(ofrmBuscar)
    ofrmBuscar.Show()

End Sub
```

Código fuente 513

Podemos eliminar la asociación entre un formulario propietario y uno dependiente mediante el método RemoveOwnedForm( ) en el formulario propietario. Esto no quiere decir que el formulario dependiente sea eliminado, simplemente se elimina su dependencia con respecto al propietario.

En lo que respecta al código de frmBuscar, al pulsar su botón Buscar, buscamos el contenido del control txtBuscar en el formulario propietario frmCarta.

Si la búsqueda tiene éxito, seleccionamos el texto encontrado dentro del propietario. La propiedad Owner del formulario nos devuelve una referencia del propietario, mientras que para manipular los controles de dicho propietario, realizaremos un moldeado de tipo o type casting sobre Owner utilizando la función CType( ) (observe el lector de nuevo, la enorme potencia que encierra esta función).

Además mostramos una etiqueta en el formulario dependiente, que sólo se visualizará al localizar el texto; cuando volvamos a escribir de nuevo texto a buscar, se ocultará dicha etiqueta. El Código fuente 514 muestra los métodos de frmBuscar que llevan a cabo estas labores.

```
' al pulsar este botón, buscamos en el formulario
' propietario de este dependiente
Private Sub btnBuscar_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnBuscar.Click

    Dim iResultadoBuscar As Integer
    ' la propiedad Owner contiene el formulario propietario
    iResultadoBuscar = CType(Me.Owner,
frmCarta).txtDocumento.Text.IndexOf(Me.txtBuscar.Text)
```

```
    ' si encontramos el texto buscado...
    If iResultadoBuscar > 0 Then
        ' pasamos el foco al TextBox del formulario propietario
        ' y seleccionamos el texto encontrado
        CType(Me.Owner, frmCarta).txtDocumento.Focus()
        CType(Me.Owner, frmCarta).txtDocumento.SelectionStart = iResultadoBuscar
        CType(Me.Owner, frmCarta).txtDocumento.SelectionLength =
Me.txtBuscar.Text.Length

        Me.lblEncontrado.Show()
    End If

End Sub

' al volver a teclear un valor a buscar, se oculta el Label
Private Sub txtBuscar_TextChanged(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles txtBuscar.TextChanged

    Me.lblEncontrado.Hide()

End Sub
```

Código fuente 514

La Figura 307 muestra la aplicación con ambos formularios abiertos. El formulario frmCarta tiene el foco actualmente, pero eso no impide que frmBuscar también permanezca abierto, para poder pasar a él en cualquier momento.

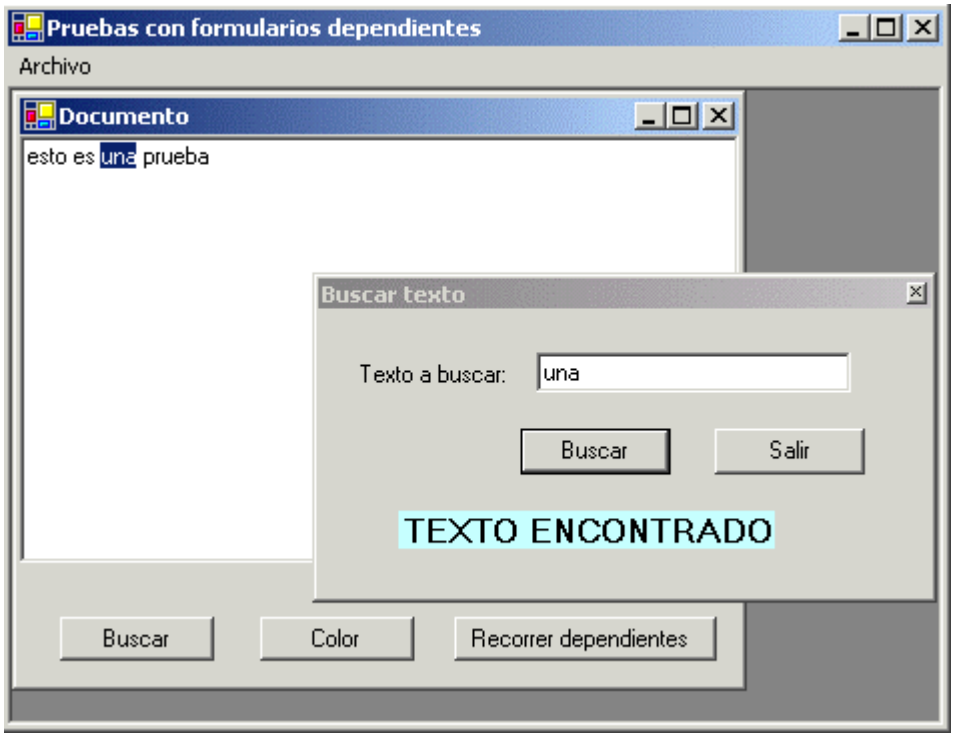


Figura 307. Formulario propietario y dependiente en funcionamiento.

Un formulario dependiente, aunque se muestra en todo momento encima de su propietario, puede ser ocultado por otro formulario de la aplicación. Para demostrarlo, añadiremos al proyecto el formulario frmDatosUsuario, que se mostrará como cuadro de diálogo, visualizando un Label en su interior. Ver Figura 308.

Figura 308. Formulario frmDatosUsuario.

El código de la opción de menú de frmPrincipal que abre este formulario se muestra en el Código fuente 515.

```
Private Sub mnuUsuario_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles mnuUsuario.Click

    ' mostrar este formulario como un diálogo
    Dim ofrmUsuario As New frmDatosUsuario()
    ofrmUsuario.ShowDialog()

End Sub
```

Código fuente 515

La Figura 309 muestra como este formulario oculta parcialmente al de búsqueda.

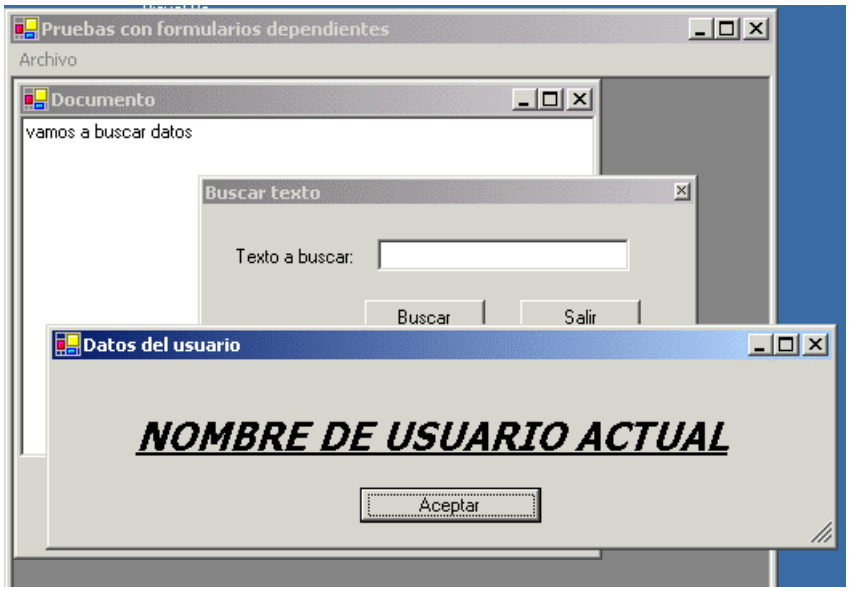


Figura 309. Formulario de diálogo ocultando parte del formulario dependiente.

Para lograr que un formulario se muestre en todo momento por encima del resto de formularios de la aplicación, hemos de asignar el valor True a su propiedad TopMost; obtenemos de esta manera, un formulario con estilo de visualización fijo en primer plano.

Ilustraremos este particular añadiendo un nuevo formulario al proyecto, con el nombre frmPonerColor, en el que asignaremos a su propiedad TopMost el valor True. Ver la Figura 310.

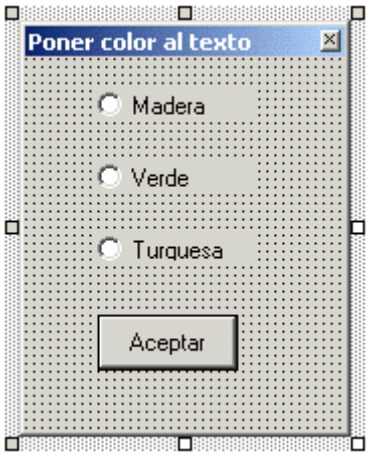


Figura 310. Formulario de estilo TopMost.

El Código fuente 516 muestra el código del botón Color de frmCarta, en el que se crea un formulario frmPonerColor y se visualiza.

```
Private Sub btnColor_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnColor.Click

    ' abrir el formulario para poner color al texto
    Dim ofrmPonerColor As New frmPonerColor(Me)
    ofrmPonerColor.Show()

End Sub
```

Código fuente 516

En este momento debemos hacer dos observaciones: en primer lugar, no añadimos el formulario frmPonerColor a la colección de formularios dependientes del propietario; en segundo lugar, al instanciar el objeto frmPonerColor, estamos pasando al constructor de esta clase la referencia del formulario propietario.

La explicación a este modo de proceder la encontramos dentro del código del formulario dependiente; en donde añadimos dicho formulario, a la lista de formularios dependientes del propietario, utilizando la propiedad Owner de la clase base Form. Esto tiene el mismo efecto que usar el método AddOwnedForm( ). El Código fuente 517 muestra el constructor de la clase frmPonerColor, en donde llevamos a cabo esta operación.

```
Public Class frmPonerColor
    Inherits System.Windows.Forms.Form
'....
```

```
' crear un constructor para establecer
' el formulario propietario
Public Sub New(ByVal frmPropietario As frmCarta)
    Me.New()
    Me.Owner = frmPropietario
End Sub
```

Código fuente 517

Al volver a ejecutar ahora el programa, si abrimos el formulario frmPonerColor y después el cuadro de diálogo, será el formulario de configuración de color el que prevalezca por encima, al ser dependiente y TopMost. Ver Figura 311.

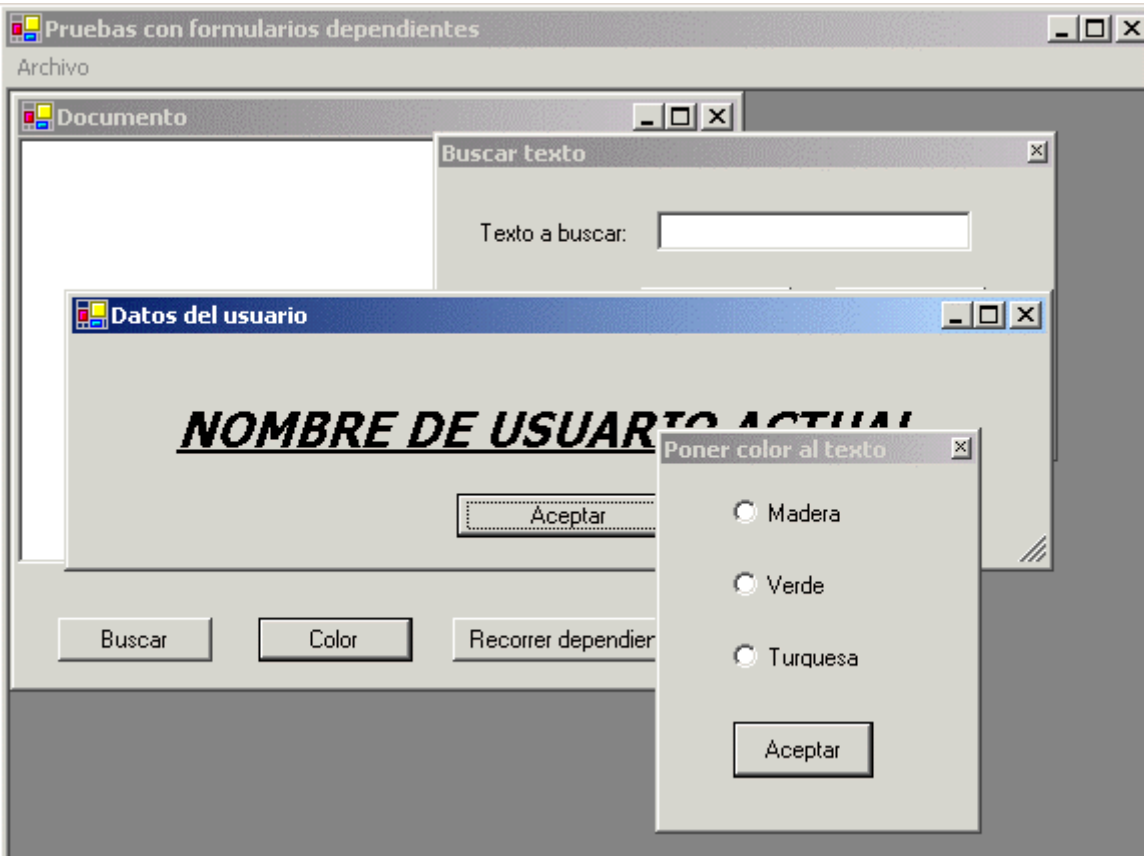


Figura 311. El formulario TopMost siempre se sitúa encima del resto.

Este formulario será abierto también desde frmCarta, mediante su botón Color, y lo utilizaremos para cambiar el color del control de texto de frmCarta. El Código fuente 518 muestra el procedimiento manipulador de evento de los controles RadioButton, en el que se realiza el cambio de color en el formulario propietario.

```
' en este método ponemos el color al TextBox
' del formulario propietario
Private Sub PonerColor(ByVal sender As System.Object, ByVal e As System.EventArgs)
Handles rbtMadera.Click, rbtVerde.Click, rbtTurquesa.Click

    Dim oColor As Color
```

```
    If sender Is Me.rbtMadera Then
        oColor = Color.BurlyWood
    End If

    If sender Is Me.rbtVerde Then
        oColor = Color.MediumSpringGreen
    End If

    If sender Is Me.rbtTurquesa Then
        oColor = Color.Turquoise
    End If

    CType(Me.Owner, frmCarta).txtDocumento.BackColor = oColor

End Sub
```

Código fuente 518

Para finalizar con los formularios dependientes, debemos indicar que la clase Form dispone de la propiedad OwnedForms, que contiene una colección con los formularios dependientes de un formulario que actúe como propietario.

Ya que en este ejemplo es el formulario frmCarta el que se comporta como propietario, añadiremos un botón con el nombre btnDependientes, que nos permitirá recorrer la mencionada colección, y hacer, desde el propietario, una manipulación sobre los formularios dependientes, en el caso de que haya alguno abierto. El Código fuente 519 muestra el código de este botón.

```
Private Sub btnDependientes_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnDependientes.Click

    ' obtener los formularios dependientes
    Dim oFormularios() As Form = Me.OwnedForms
    Dim oFormulario As Form

    ' si existen dependientes...
    If oFormularios.Length > 0 Then
        ' recorrer la colección y
        ' manipular los formularios dependientes
        For Each oFormulario In oFormularios

            Select Case oFormulario.GetType().Name
                Case "frmBuscar"

                    CType(oFormulario, frmBuscar).lblEncontrado.Show()
                    CType(oFormulario, frmBuscar).lblEncontrado.Text =
"¡LOCALIZADO!"

                Case "frmPonerColor"
                    CType(oFormulario, frmPonerColor).rbtTurquesa.Text = "AZULADO"
                    CType(oFormulario, frmPonerColor).rbtTurquesa.BackColor =
Color.Blue

                    ' con el método PerformClick() de un control,
                    ' simulamos una pulsación
                    CType(oFormulario, frmPonerColor).rbtTurquesa.PerformClick()

            End Select

        Next

    End If
```

```
End Sub
```

Código fuente 519

Validación de controles

Los controles Windows vienen provistos de un potente y flexible sistema de validación, que nos permitirá comprobar si el usuario introduce los valores adecuados en un control, de modo que le permitiremos pasar el foco a otro control, u obligarle a permanece en el actual hasta que su valor no sea correcto.

En este esquema de validación, los miembros principales de la clase Control que intervienen son los siguientes.

- **CausesValidation.** Esta propiedad nos permite establecer un valor lógico, de manera que cuando un control capture el foco, provocará la validación para otro control del formulario que la requiera.

- **Validating.** Este evento se produce para que podamos escribir el código de validación oportuno en un manipulador de evento. El procedimiento manejador de evento recibe entre sus parámetros un objeto de tipo CancelEventArgs, por lo que si la validación no es correcta, asignaremos False a la propiedad Cancel de dicho objeto.

- **Validated.** Este evento se produce en el caso de que la validación haya tenido éxito.

El proyecto de ejemplo ValidarControl (hacer clic aquí para acceder a este ejemplo) consta de un formulario con tres controles TextBox. Todos tienen el valor True en su propiedad CausesValidation, y adicionalmente, para el control txtImporte hemos escrito el procedimiento que actuará como manipulador del evento Validating; con ello impediremos el paso desde dicho control a los demás hasta que su contenido no sea numérico. Si pasamos la validación, se ejecutará en ese caso el código del evento Validated. Veamos estos manipuladores de evento en el Código fuente 520.

```
Private Sub txtImporte_Validating(ByVal sender As Object, ByVal e As
System.ComponentModel.CancelEventArgs) Handles txtImporte.Validating

    If Not IsNumeric(Me.txtImporte.Text) Then
        e.Cancel = True
        MessageBox.Show("Se requiere un número")
    End If

End Sub
```

Código fuente 520

La Figura 312 muestra esta aplicación en funcionamiento, durante la ejecución del evento de validación.

En el control txtFecha por otro lado, podemos teclear cualquier valor, aunque no sea fecha, ya que no proporcionamos manipuladores de evento para validar su contenido.

Cuando escribimos código de validación empleando estos miembros de la clase Control hemos de tener presente el comportamiento, a veces no muy intuitivo, del sistema de validación para controles en los formularios Windows.

Figura 312. Validación de un control.

Como hemos mencionado anteriormente, cuando la propiedad CausesValidation de un control contiene True, al recibir el foco dicho control, se provocará el evento de validación para el control que acaba de perder el foco. Pero si pasamos el foco a un control en el que CausesValidation contiene False, la validación no se producirá sobre el control que acaba de perder el foco.

Esto lo podemos comprobar muy fácilmente sobre nuestro proyecto de ejemplo, asignando al control txtFecha el valor False en su CausesValidation. A partir de ahora, cuando estemos situados en el control txtImporte, si este no contiene un número, se producirá la validación si pasamos el foco a txtNombre, pero no se validará si pasamos a txtFecha.

Controles avanzados

Los controles del Cuadro de herramientas del IDE tratados hasta el momento, son los que podríamos considerar básicos o estándar en todas las aplicaciones; no obstante, esta ventana de herramientas dispone de otra serie de controles avanzados o adicionales, que si bien, no son imprescindibles para conseguir la funcionalidad elemental del programa, sirven como un magnífico complemento a la hora de dotar a nuestras aplicaciones de un interfaz de usuario plenamente operativo.

En los siguientes apartados desarrollaremos un proyecto con el nombre ControlAvanzado (hacer clic aquí para acceder a este ejemplo), a través del cual, realizaremos una descripción general de algunos de estos controles adicionales y su modo de uso.

Como primer paso en este proyecto, eliminaremos el formulario por defecto, añadiendo a continuación uno nuevo con el nombre frmPrincipal, al que daremos la característica MDI mediante la propiedad IsMdiContainer. En este formulario crearemos un menú con un conjunto de opciones generales: Abrir, Guardar, Salir, etc.

ImageList

Este control actúa como repositorio de imágenes, del que se alimentarán otros controles del formulario que necesiten mostrar gráficos en su interior.

Una vez añadido este control en el formulario, se situará en el panel de controles especiales del diseñador, y haciendo clic en su propiedad Images, se abrirá la ventana de la Figura 313, en la que podremos añadir y quitar las imágenes que van a formar parte de la lista del control, así como ver en el panel complementario, la información sobre cada imagen asignada.

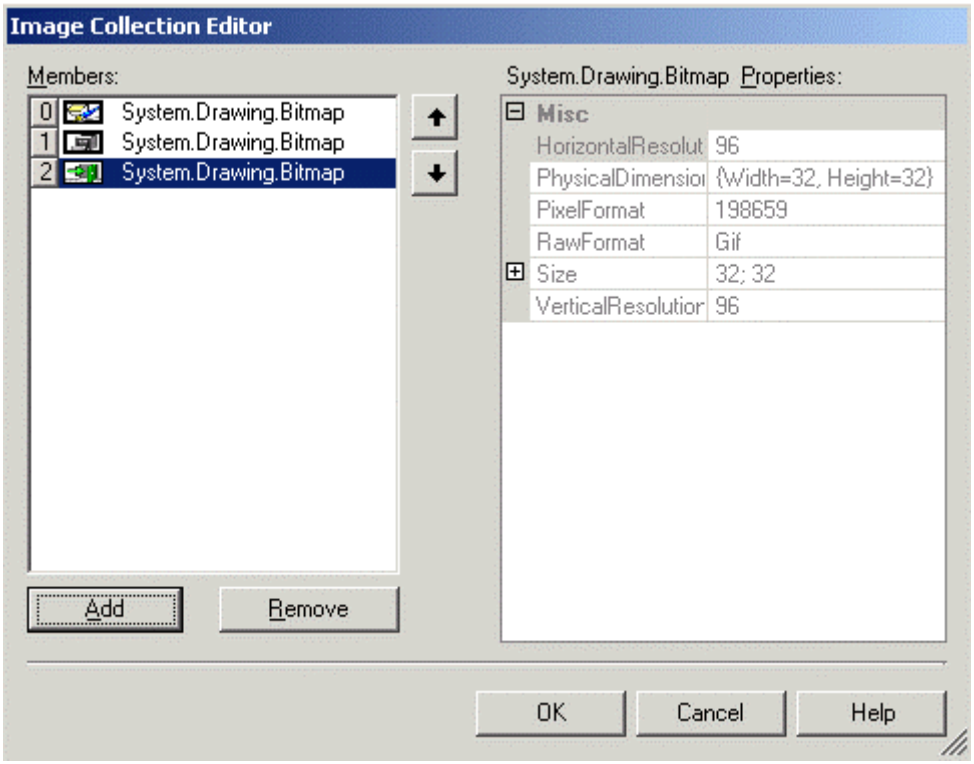


Figura 313. Ventana de administración de imágenes del control ImageList.

Las imágenes que insertamos en el control tienen un tamaño por defecto, en el caso de que necesitemos modificarlo, expandiremos la propiedad ImageSize en la ventana de propiedades y asignaremos nuevos valores en Width y Height.

Otra ventaja de este control es que nos permite manipular las imágenes por código, por ejemplo, para añadir nuevas imágenes, debemos usar el método Add( ) de su propiedad Images, como muestra el Código fuente 521.

```
Me.imlImagenes.Images.Add(New Bitmap("tutorias.gif"))
```

Código fuente 521

ToolBar

Este control representa la barra de herramientas o botones de acceso rápido que facilitan al usuario la ejecución de los procesos principales del programa, evitándole la navegación por el menú del formulario.

Al ser dibujado, este control queda acoplado a la parte superior del formulario. Después de ponerle tbrBarra como nombre, asignaremos a su propiedad ImageList, el control de ese mismo tipo que acabamos de crear; esto nos permitirá asignar los gráficos de la lista a los botones que vayamos creando en el ToolBar. Para establecer el tamaño de los botones de la barra utilizaremos la propiedad ButtonSize de este control.

Seguidamente haremos clic en la propiedad Buttons, que abrirá una ventana con la colección de botones de la barra, en la que podremos crear y configurar dichos botones. Ver Figura 314.

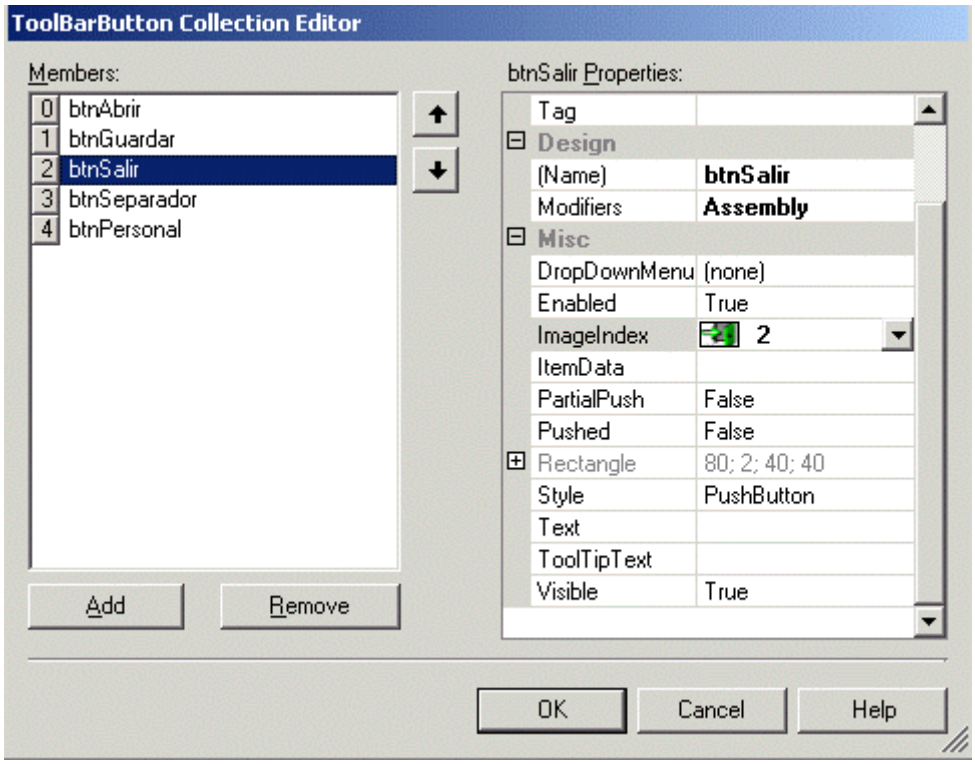


Figura 314. Editor de botones del control ToolBar.

Cada botón en un ToolBar es un objeto de tipo ToolBarButton, del que podemos destacar las siguientes propiedades.

- **Text**. Cadena con el texto que muestra el botón.
- **ImageIndex**. En el caso de asociar el ToolBar con un control ImageList, en esta propiedad asignamos para un botón una de las imágenes del ImageList, indicando el número de orden de la imagen.
- **Style**. Permite establecer el estilo del botón: de pulsación; separador; o de tipo desplegable, que abre un subconjunto de opciones.

- **DropDownMenu**. Si asociamos el botón con una opción de la barra de menú del formulario, y configuramos su estilo como DropDownButton, al pulsar el botón desplegable, se mostrarán las opciones de menú; el efecto será el mismo que si hubiéramos desplegado directamente el menú del formulario.

La Figura 315 muestra la ventana principal de la aplicación con la barra de herramientas

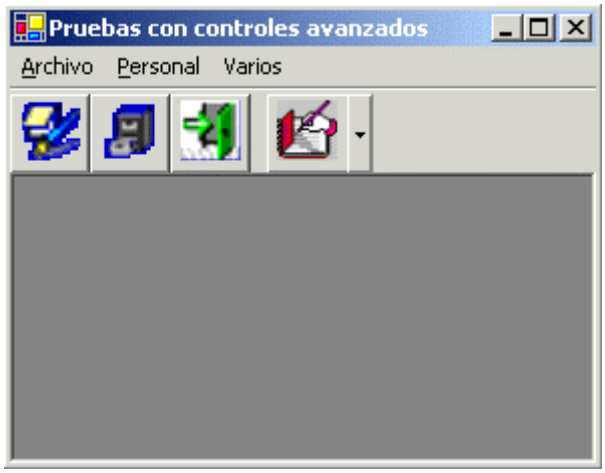


Figura 315. Formulario con ToolBar.

Una vez terminado el diseño del ToolBar, debemos codificar su evento ButtonClick, que será provocado cada vez que se pulse un botón de la barra. Dentro del procedimiento de este evento, comprobaremos qué botón ha sido pulsado y ejecutaremos las acciones oportunas. El Código fuente 522 muestra este evento. Tanto el botón Abrir como la opción de menú del mismo nombre realizan la misma tarea, por lo que llaman al método AbrirArchivo( ), que es quien realmente muestra el formulario necesario.

```
Private Sub tbrBarra_ButtonClick(ByVal sender As System.Object, ByVal e As
System.Windows.Forms.ToolBarButtonClickEventArgs) Handles tbrBarra.ButtonClick

    ' comprobar qué botón de la barra se ha pulsado
    If e.Button Is Me.btnAbrir Then
        ' llamamos al método que abre el
        ' formulario para abrir un archivo
        Me.AbrirArchivo()
    End If

    If e.Button Is Me.btnSalir Then
        ' cerrar la aplicación
        Me.Close()
    End If

End Sub

Private Sub mnuAbrir_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles mnuAbrir.Click

    ' al seleccionar esta opción de menú
    ' llamar al método que abre el formulario
    ' que permite abrir un archivo
    Me.AbrirArchivo()

End Sub
```

```
Private Sub AbrirArchivo()
    Dim ofrmAbrirArchivo As New frmAbrirArchivo()
    ofrmAbrirArchivo.MdiParent = Me
    ofrmAbrirArchivo.Show()
End Sub
```

Código fuente 522

Al haber asignado al botón btnPersonal uno de los menús de la barra del formulario, no será necesario escribir código para detectar este botón en el evento ButtonClick, ya que se ejecutará directamente el código del evento Click de las opciones de menú. El Código fuente 523 muestra el código perteneciente a la opción de menú *Personal + Datos*.

```
Private Sub mnuDatos_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles mnuDatos.Click

    Dim ofrmPersonal As New frmDatosPersonal()
    ofrmPersonal.MdiParent = Me
    ofrmPersonal.Show()

End Sub
```

Código fuente 523

StatusBar

Para mostrar una barra informativa de estado recurriremos a este control, que al dibujarse queda situado en la parte inferior del formulario; como nombre le daremos sbrEstado. De forma similar al ToolBar, un control StatusBar está compuesto de una colección de objetos Panel, que iremos añadiendo al control mediante la propiedad Panels, la cual mostrará una ventana para la creación y configuración de tales paneles. Ver Figura 316.

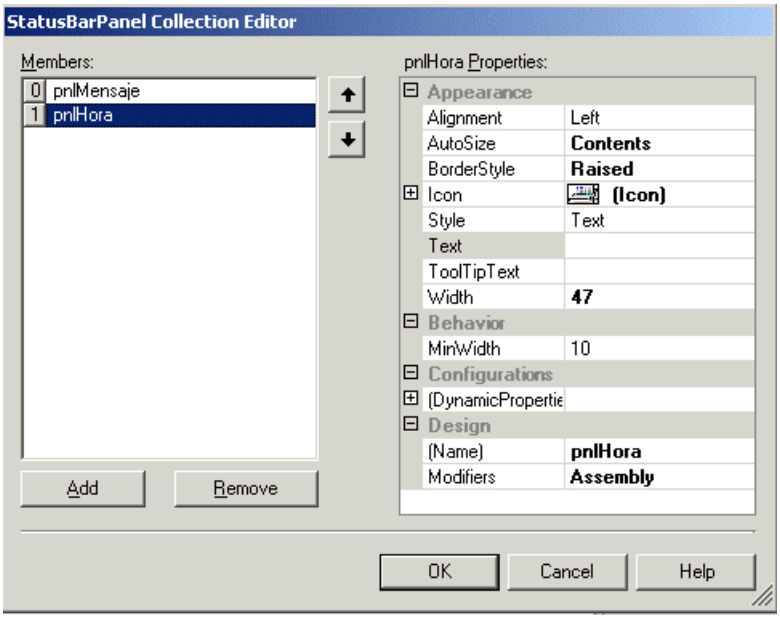


Figura 316. Editor de paneles del control StatusBar.

Entre las propiedades destacables de un objeto Panel podemos mencionar las siguientes.

- **BorderStyle**. Muestra el panel con efecto resaltado, hundido o normal.
- **Icon**. Permite asociar un icono al panel.
- **AutoSize**. Con esta propiedad podemos conseguir que el panel se redimensione ajustándose a su contenido o que tenga un tamaño fijo.

En este ejemplo, hemos añadido dos paneles a la barra de estado del formulario. En uno mostramos un texto fijo; mientras que en el otro, visualizamos la hora actual a través de un objeto Timer que ponemos en marcha en el evento Load del formulario. Veamos los métodos implicados, en el Código fuente 524.

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

    ' al cargar el formulario, creamos un temporizador
    ' le asociamos un manejador para su evento Tick
    ' y lo iniciamos
    Dim oTiempo As New Timer()
    oTiempo.Interval = 1000
    AddHandler oTiempo.Tick, AddressOf PonerHoraActual
    oTiempo.Start()

End Sub

Private Sub PonerHoraActual(ByVal sender As Object, ByVal e As EventArgs)

    ' actualizamos a cada segundo la hora de un panel
    ' de la barra de estado
    Me.sbrEstado.Panels(1).Text = DateTime.Now.ToString("HH:mm:ss")

End Sub
```

Código fuente 524

La Figura 317 muestra el formulario con la barra de estado.

Figura 317. Formulario con StatusBar.

Finalizada la creación del StatusBar, añadiremos al proyecto un formulario con el nombre frmDatosPersonal, en el dibujaremos un conjunto de nuevos controles que iremos describiendo seguidamente.

DateTimePicker

Este control permite la selección e introducción de fechas en una caja de texto con capacidades extendidas, o bien mediante un calendario desplegable que se mostrará al pulsar el botón de expansión que contiene. Ver Figura 318.

Figura 318. Control DateTimePicker.

Para modificar la fecha en el cuadro de texto, debemos situarnos en la parte a modificar de la fecha y teclear el nuevo valor, o bien, con las flechas de dirección arriba-abajo, cambiar esa parte de la fecha. Si expandimos el calendario, podremos realizar la selección de un modo más gráfico.

Por defecto el control muestra la fecha actual, pero con la propiedad Text podemos cambiar la fecha por código, cosa que hacemos al cargar el formulario, asignando una fecha distinta de la actual. Ver Código fuente 525.

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

    ' modificar fecha del DateTimePicker
    Me.dtpFNacim.Text = "15/06/2002"
    '....
End Sub
```

Código fuente 525

Podemos restringir el rango de fechas a mostrar por este control con las propiedades MinDate y MaxDate. Si queremos, por otra parte, que la fecha se muestre con un formato personalizado, aplicaremos dicho formato mediante la propiedad CustomFormat, teniendo en cuenta que no se hará efectivo hasta que a la propiedad Format no le asignemos el valor Custom.

El botón btnCambiosFecha del formulario realiza algunas modificaciones por código sobre el control DateTimePicker dtpFNacim del formulario, que vemos en el Código fuente 526.

```
Private Sub btnCambiosFecha_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnCambiosFecha.Click

    ' configurar por código el
    ' control DateTimePicker
    Me.dtpFNacim.MinDate = "1/4/2002"
    Me.dtpFNacim.MaxDate = "1/10/2002"
    Me.dtpFNacim.CustomFormat = "d-MMM-yy"
    Me.dtpFNacim.Format = DateTimePickerFormat.Custom

End Sub
```

Código fuente 526.

NumericUpDown

Control que muestra una caja de texto con un valor numérico que podremos ir aumentando-disminuyendo al pulsar los botones para esta labor de que dispone el control. La Figura 319 muestra este control en nuestro formulario de pruebas.

Figura 319. Control NumericUpDown.

Entre las propiedades de este control destacaremos las siguientes.

- **Increment**. Número en el que se incrementará el valor del control cuando pulsemos sus botones o teclas de dirección.
- **InterceptArrowKeys**. Permite que las flechas de dirección arriba-abajo tengan el mismo efecto que si pulsamos los botones para incrementar o disminuir, de este control.
- **Maximum, Minimun**. Contienen los límites superior e inferior en cuanto al número que podrá contener el control.
- **TextAlign**. Permite alinear el número dentro la caja de texto del control.
- **UpDownAlign**. Permite situar los botones del control a la izquierda o derecha de la caja de texto que contiene el valor.

Entre los eventos de que dispone este control, ValueChanged se produce cada vez que cambia el valor del control, de modo que en este caso, vamos a cambiar el color de fondo en función del número que contenga. Veamos el Código fuente 527.

```
Private Sub nupEdad_ValueChanged(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles nupEdad.ValueChanged

    Select Case Me.nupEdad.Value
        Case 20 To 30
            Me.nupEdad.BackColor = Color.Gold

        Case 30 To 40
            Me.nupEdad.BackColor = Color.LimeGreen
```

```
        Case Else
            Me.nupEdad.BackColor = Me.nupEdad.DefaultBackColor
    End Select

End Sub
```

Código fuente 527

DomainUpDown

Este control nos permite desplazarnos por una lista de valores, al mismo estilo que el control anterior. Dicha lista de valores la crearemos mediante la propiedad Items, en tiempo de diseño o ejecución.

El Código fuente 528 muestra como al cargar el formulario frmDatosPersonal, con la propiedad Items y su método AddRange( ), añadimos los valores que seleccionaremos en el control en tiempo de ejecución.

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

    '....
    ' crear la lista del DomainUpDown
    Me.dudCategoria.Items.AddRange(New String() {"Auxiliar", "Jefe departamento",
"Coordinador"})

End Sub
```

Código fuente 528

La Figura 320 muestra el control dudCategoría, de este tipo al ser utilizado en el formulario. En el caso de que necesitemos los valores ordenados, asignaremos True a su propiedad Sorted.

Figura 320. Control DomainUpDown.

MonthCalendar

Este control muestra en modo gráfico un calendario por el que podemos desplazarnos para seleccionar una fecha. El control DateTimePicker utiliza internamente un MonthCalendar para mostrar su calendario desplegable.

Por defecto se visualiza un mes, pero si asignamos a su propiedad CalendarDimensions un objeto Size, podemos expandir el tamaño del calendario para que muestre varios meses. El Código fuente 529 muestra el código de un botón del formulario mediante el que cambiamos el tamaño del calendario.

```
Private Sub btnTamCalendario_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnTamCalendario.Click
```

```
    Me.mclCalendario.CalendarDimensions = New Size(2, 2)

End Sub
```

Código fuente 529

En la Figura 321 vemos el resultado de expandir el tamaño del control.

Figura 321. Control MonthCalendar con el tamaño ampliado.

Para detectar la selección de una fecha utilizaremos el evento DateChanged. Debido a que en un control MonthCalendar podemos seleccionar un rango de fechas, las propiedades que tenemos que manipular para averiguar la fecha/s seleccionada/s son: SelectionStart, SelectionEnd y SelectionRange, aunque en muchas ocasiones sólo será necesario utilizar una de ellas. Veamos el Código fuente 530.

```
Private Sub mclCalendario_DateChanged(ByVal sender As System.Object, ByVal e As
System.Windows.Forms.DateRangeEventArgs) Handles mclCalendario.DateChanged

    ' mostrar en un Label la fecha seleccionada
    ' en el control MonthCalendar
    Me.lblCalendario.Text = Me.mclCalendario.SelectionStart

End Sub
```

Código fuente 530

LinkLabel

Este control permite tener en un formulario Windows un enlace hacia una página de Internet, con un comportamiento similar al que encontramos en un hiperenlace de una página web.

Su propiedad Text muestra un literal, de modo que al hacer clic sobre el mismo, se provocará el evento LinkClicked en el que escribiremos el código a ejecutar.

En nuestro formulario de ejemplo, hemos creado un control de este tipo con el nombre lnkEidos, que tiene el aspecto de la Figura 322, ya que además del enlace, le hemos asignado una imagen.

Figura 322. Conrol LinkLabel.

Para conseguir que al hacer clic en este enlace, se abra Internet Explorer y navegue hacia una determinada página, vamos a utilizar la clase Process, que como su nombre indica, nos permite la gestión de procesos del sistema, tales como su inicio y finalización.

En este caso, el método compartido Start( ), de Process, va a ejecutar el navegador al pasarle como parámetro una dirección web en forma de cadena. Veamos el Código fuente 531.

```
Private Sub lnkEidos_LinkClicked(ByVal sender As System.Object, ByVal e As
System.Windows.Forms.LinkLabelLinkClickedEventArgs) Handles lnkEidos.LinkClicked

    ' inicia Internet Explorer y navega hacia una página
    Process.Start("http://www.eidos.es")

End Sub
```

Código fuente 531

Creación y manipulación de elementos en ejecución

El conjunto de controles que acabamos de ver, al igual que los básicos permiten ser creados no sólo mediante el diseñador de formularios, sino también desde código. Como muestra de ello, la opción de menú *Varios + Añadir botón y panel*, añade dos imágenes al control ImageList del formulario frmPrincipal, creando un nuevo botón para el ToolBar y un panel para el StatusBar Ver el Código fuente 532.

```
Private Sub mnuAgregaElementos(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles mnuAgregar.Click

    ' añadir por código imágenes a la lista
    Me.imlImagenes.Images.Add(New Bitmap("tutorias.gif"))
    Me.imlImagenes.Images.Add(New Bitmap("camera1.gif"))

    Dim oBoton As New ToolBarButton()
    oBoton.Text = "TUTORIAS"
    oBoton.ImageIndex = 4
    Me.tbrBarra.Buttons.Add(oBoton)

    Dim oPanel As New StatusBarPanel()
```

```
    oPanel.Text = "BUSCAR"
    oPanel.BorderStyle = StatusBarPanelBorderStyle.Raised
    oPanel.ToolTipText = "Información sobre búsquedas"
    oPanel.Icon = New Icon("magnify.ico")
    Me.sbrEstado.Panels.Add(oPanel)

End Sub
```

Código fuente 532

Para detectar la pulsación del nuevo botón de la barra de herramientas, añadimos el siguiente código en su evento ButtonClick, que vemos en el Código fuente 533.

```
Private Sub tbrBarra_ButtonClick(ByVal sender As System.Object, ByVal e As
System.Windows.Forms.ToolBarButtonClickEventArgs) Handles tbrBarra.ButtonClick
    '....
    If e.Button.Text = "TUTORIAS" Then
        MessageBox.Show("Se ha pulsado el botón de tutorías")
    End If
End Sub
```

Código fuente 533

La Figura 323 muestra este formulario en ejecución tras añadir los nuevos elementos.

Figura 323. Nuevos controles añadidos al formulario en tiempo de ejecución.

NotifyIcon

Este control permite añadir un icono asociado con nuestra aplicación en el panel de iconos del sistema (Windows System Tray) situado en la parte derecha de la barra de tareas de Windows.

Tales iconos suelen utilizarse por aplicaciones que permanecen ocultas, y al hacer clic derecho sobre su icono en este panel, aparece un menú contextual que permite mostrar la aplicación.

En nuestro caso vamos a utilizar este control para ejecutar y parar la calculadora del sistema empleando la clase Process, comentada en un apartado anterior.

Después de agregar un control de este tipo al formulario, asignaremos un icono a su propiedad Icon y una cadena a su propiedad Text, que será mostrada al situar el ratón encima de este control en tiempo de ejecución.

Crearemos después un menú contextual con las opciones Abrir y Cerrar, que asignaremos a la propiedad ContextMenu del control NotifyIcon.

Para poder controlar la calculadora de Windows cuando esté en ejecución, declararemos una variable de tipo Process a nivel de la clase. Al ejecutar la calculadora mediante el método Start( ) de la clase Process, obtendremos un objeto de dicho tipo, que pasaremos a esta variable, y nos permitirá posteriormente, cerrar el proceso en el que se está ejecutando la calculadora mediante el método Kill( ). Veamos esta parte en el Código fuente 534.

```
Public Class frmPrincipal
    Inherits System.Windows.Forms.Form

    Private oCalculadora As Process
    '....

    Private Sub mnuCalcAbrir_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles mnuCalcAbrir.Click

        ' iniciar la calculadora
        oCalculadora = Process.Start("calc.exe")

    End Sub

    Private Sub mnuCalcCerrar_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles mnuCalcCerrar.Click

        ' cerrar la calculadora
        oCalculadora.Kill()

    End Sub

End Class
```

Código fuente 534

Al ejecutar el programa, se mostrará un nuevo icono en la lista del panel de iconos del sistema de la barra de tareas, como muestra la Figura 324.

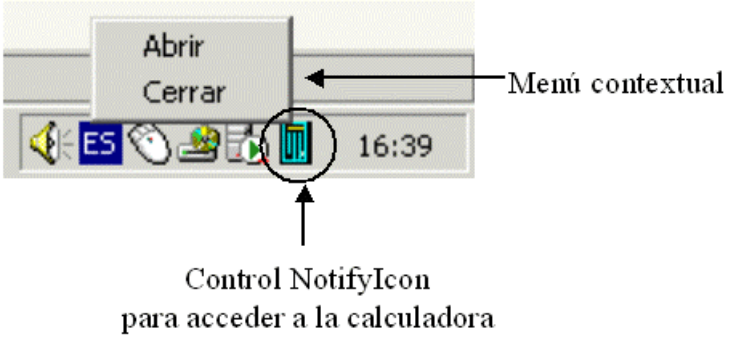


Figura 324. Control NotifyIcon en el panel de iconos del sistema de la barra de tareas.

Como puede comprobar el lector, la clase Process amplía enormemente nuestra capacidad de manipulación de los procesos del sistema.

35

GDI+. Acceso al subsistema gráfico de Windows

GDI+ es la evolución de GDI, el API para la manipulación de gráficos incluido en anteriores versiones de Windows.

Al tratarse de un interfaz de programación independiente del dispositivo físico sobre el que se van a generar los gráficos, el programador gana en flexibilidad, ya que no debe preocuparse de si el gráfico generado se va a mostrar por el monitor, impresora, etc.; esta labor es resuelta por GDI+, que aísla el programa del hardware a manejar.

GDI+ divide su campo de trabajo en tres áreas principales.

- Generación de gráficos vectoriales 2D
- Manipulación de imágenes en los formatos gráficos más habituales.
- Visualización de texto en un amplio abanico de tipos de letra.

En versiones anteriores del lenguaje, el objeto Form disponía de una serie de métodos y controles para el dibujo sobre la superficie del formulario. La mayor parte de esos elementos han desaparecido en la actual versión de VB, integrándose todas las operaciones de dibujo en las clases de .NET Framework; con ello, lo que aprendamos sobre trabajo con gráficos en VB.NET utilizando GDI+, nos servirá igualmente si en un futuro debemos abordar un proyecto en otro lenguaje de la plataforma, ya que las clases pertenecen a .NET Framework y no a un lenguaje en particular.

Otro problema con el que nos enfrentábamos anteriormente era el hecho de que al necesitar alguna manipulación gráfica especial, teníamos que recurrir al API de Windows. A partir de ahora esto no será necesario, ya que como hemos comentado, es el propio entorno de ejecución de .NET el que nos proporciona dicho acceso, por lo que no será necesario acudir al API del sistema operativo.

System.Drawing

Para utilizar las clases relacionadas con la manipulación de gráficos, es preciso importar este espacio de nombres.

System.Drawing contiene el conjunto de clases principales, aunque no es el único namespace de GDI+; para tareas de mayor especialización con gráficos deberemos recurrir a alguno de los siguientes espacios de nombre que están dentro de System.Drawing: Drawing2D, Imaging y Text.

Dibujo con las clases Graphics y Pen

La clase Graphics representa el denominado *Contexto de dispositivo gráfico* (Graphics device context) sobre el que se va a realizar una operación de dibujo, por ejemplo, el formulario.

Debido a la arquitectura del sistema gráfico, no es posible tomar un objeto Form y realizar una operación directa de dibujo sobre el mismo, sino que precisamos en primer lugar, obtener una referencia hacia el área de dibujo de dicho formulario, o contexto gráfico, y una vez obtenida esa referencia, efectuar el dibujo.

Este área lo vamos a obtener mediante el método CreateGraphics( ) de la clase Form, que devuelve un objeto Graphics con la información del contexto de dispositivo gráfico del formulario, que usaremos para dibujar sobre el mismo, mediante el conjunto de métodos DrawXXX( ).

Por otro lado, la clase Pen representa un objeto de tipo lapicero o bolígrafo, que con un determinado color y grosor, utilizará un objeto Graphics para dibujar líneas y formas en un contexto de dispositivo, es decir, el formulario.

El Código fuente 535 muestra un ejemplo de dibujo de un círculo sobre el formulario utilizando el método DrawEllipse( ) de Graphics. Este método recibe como parámetro un objeto Pen con un color y grosor determinados y un objeto Rectangle con las coordenadas y medida necesarias para dibujar el círculo.

```
' crear objeto Pen
Dim oPen As New Pen(Color.DeepPink, 10)

' obtener el contexto de dispositivo
' gráfico del formulario
Dim oGraphics As Graphics = Me.CreateGraphics()

' dibujar en el formulario
oGraphics.DrawEllipse(oPen, New Rectangle(150, 20, 100, 100))
```

Código fuente 535

Asociando este código a la pulsación de un botón en el formulario, el resultado será el dibujo del círculo mostrado en la Figura 325.

Figura 325. Dibujar un círculo con un objeto Pen.

Sin embargo, el dibujo de figuras de esta manera tiene un inconveniente, puesto que si el formulario es ocultado parcial o totalmente por otro, la zona ocultada se dice que ha quedado invalidada, y requiere un repintado, labor que actualmente, no indicamos que se haga en el código del formulario.

Para mantener las figuras dibujadas en el formulario en todo momento, debemos recurrir al evento Paint( ) de la clase Form. Dicho evento se produce cada vez que el formulario necesita repintarse porque parte o la totalidad de su superficie ha quedado invalidada

Vamos por lo tanto a observar las diferencias codificando el mencionado evento Paint( ), en el que dibujaremos un rectángulo. Ver Código fuente 536.

```
Private Sub Form1_Paint(ByVal sender As Object, ByVal e As
System.Windows.Forms.PaintEventArgs) Handles MyBase.Paint

    ' crear objeto Pen
    Dim oPen As New Pen(Color.MediumVioletRed, 6)

    ' obtener el contexto de dispositivo
    ' gráfico del formulario;
    ' en este caso usaremos el objeto que contiene
    ' los argumentos de evento para obtener dicho
    ' contexto de dispositivo
    Dim oGraph As Graphics = e.Graphics

    ' dibujar en el formulario
    oGraph.DrawRectangle(oPen, New Rectangle(40, 80, 70, 70))

End Sub
```

Código fuente 536

Al ejecutar el programa, el rectángulo será mostrado en todo momento, aunque minimicemos el formulario, lo ocultemos parcial, totalmente, etc., ya que este evento se ejecuta automáticamente cada vez que el formulario detecta que su superficie ha quedado invalidada y necesita repintarse. La Figura 326 muestra el formulario con ambas figuras dibujadas, si ocultamos y mostramos de nuevo la ventana, comprobaremos como el círculo ha desaparecido mientras que el rectángulo persiste.

Figura 326. Figuras dibujadas en la superficie del formulario desde un botón y el evento Paint.

El proyecto GraficosGDI (hacer clic aquí para acceder a este ejemplo), contiene a través de las opciones de sus menús, el conjunto de operaciones de dibujo que describiremos seguidamente.

Mediante el menú *Dibujo con Pen*, dibujaremos un círculo, rectángulo, curva, polígono, etc. Consulte el lector sus opciones para comprobar el código para cada tipo de figura. A continuación comentaremos las más destacables.

Al dibujar un rectángulo vamos a modificar el estilo de línea mediante la propiedad DashStyle. Esta propiedad contiene una enumeración con la que podemos hacer que la línea se muestre como guiones, guiones y puntos, etc. Ver Código fuente 537.

```
Private Sub mnuPenRectangulo_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles mnuPenRectangulo.Click

    ' crear objeto Pen
    Dim oPen As New Pen(Color.Firebrick, 4)

    ' aplicar un estilo de línea con la propiedad
    ' DashStyle --> guión, punto
    oPen.DashStyle = Drawing.Drawing2D.DashStyle.DashDot

    ' obtener el contexto de dispositivo
    ' gráfico del formulario
    Dim oGraphics As Graphics = Me.CreateGraphics()

    ' dibujar en el formulario
    oGraphics.DrawRectangle(oPen, New Rectangle(280, 75, 120, 40))

End Sub
```

Código fuente 537

Si queremos aplicar más estilos a la línea del objeto Pen, disponemos también de las propiedades StartCap, EndCap, DashCap. El Código fuente 538 muestra el dibujo de una curva con varios efectos de línea. Al dibujar una curva, necesitamos pasar al método DrawCurve( ) un array de tipos Point, con las coordenadas de referencia a usar para el dibujo de la curva.

```
Private Sub mnuPenCurva_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles mnuPenCurva.Click
```

```
    ' crear objeto Pen
    Dim oPen As New Pen(Color.MediumPurple, 5)

    ' configurar estilo de línea
    oPen.DashStyle = Drawing.Drawing2D.DashStyle.DashDot
    oPen.StartCap = Drawing.Drawing2D.LineCap.Triangle
    oPen.EndCap = Drawing.Drawing2D.LineCap.DiamondAnchor
    oPen.DashCap = Drawing.Drawing2D.DashCap.Triangle

    ' obtener el contexto de dispositivo
    ' gráfico del formulario
    Dim oGraphics As Graphics = Me.CreateGraphics()

    ' crear un array de puntos-coordenadas necesario
    ' para dibujar una curva
    Dim oPuntos(4) As Point
    oPuntos(0) = New Point(10, 200)
    oPuntos(1) = New Point(40, 100)
    oPuntos(2) = New Point(100, 20)
    oPuntos(3) = New Point(130, 100)
    oPuntos(4) = New Point(200, 200)

    ' dibujar en el formulario
    oGraphics.DrawCurve(oPen, oPuntos)

End Sub
```

Código fuente 538

En cuanto a las curvas de tipo Bezier, el método DrawBezier( ) recibe como parámetros, el objeto Pen y una lista de coordenadas para el dibujo. Ver el Código fuente 539.

```
Private Sub mnuPenBezier_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles mnuPenBezier.Click

    ' dibujar curva estilo Bezier
    Dim oGraphics As Graphics = Me.CreateGraphics()
    oGraphics.DrawBezier(New Pen(Color.MediumSeaGreen, 2), 20, 45, 100, 90, 140,
200, 300, 25)

End Sub
```

Código fuente 539

La Figura 327 muestra todas las formas dibujadas con objetos Pen.

Si en un momento dado, necesitamos borrar los elementos gráficos dibujados en la superficie del formulario, utilizaremos el método Invalidate( ) de la clase Form, que en este ejemplo está disponible en la opción de menú *Abrir + Borrar*. Ver Código fuente 540.

```
Private Sub mnuBorrar_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles mnuBorrar.Click
    ' borrar las figuras dibujadas en el formulario
    Me.Invalidate()

End Sub
```

Código fuente 540

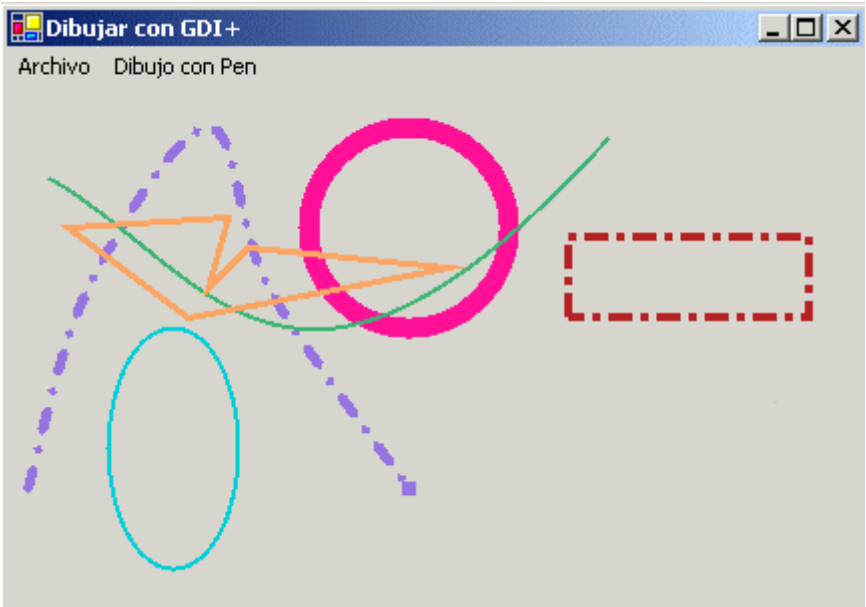


Figura 327. Figuras dibujadas con objetos de la clase Pen.

La clase Brush

Esta clase representa un objeto de tipo brocha, utilizado para rellenar de diferentes formas, las figuras dibujadas sobre el formulario.

Se trata de una clase abstracta, por lo que tendremos que utilizar alguna de sus diversas clases derivadas, según el estilo de brocha que necesitemos aplicar. Debido a las características 2D de algunas de estas clases, tendremos que importar en nuestro código el espacio de nombres Drawing2D.

Los métodos de la clase Graphics que utilizaremos para dibujar con brochas serán los que comienzan por el nombre FillXXX( )

El menú *Dibujo con Brush* del formulario de este ejemplo, muestra algunas de las operaciones de dibujo y estilos de relleno, que podemos aplicar con las clases de tipo Brush.

La clase más básica es SolidBrush, que permite rellenar en un estilo sencillo un área dibujada. Ver el Código fuente 541.

```
Private Sub mnuBrushSolid_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles mnuBrushSolid.Click

    Dim oBrush As New SolidBrush(Color.MidnightBlue)
    Dim oGraphics As Graphics = Me.CreateGraphics()
    oGraphics.FillRectangle(oBrush, New Rectangle(150, 160, 150, 50))

End Sub
```

Código fuente 541

A continuación tenemos la clase HatchBrush, que permite la creación de brochas que al pintar aplican un efecto de tramado con un color de fondo y frente. Ver el Código fuente 542.

```
Private Sub mnuBrushHatch_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles mnuBrushHatch.Click

    ' pintar con brocha de tipo hatch;
    ' para utilizar este tipo de brocha
    ' necesitamos importar System.Drawing.Drawind2D

    ' crear objeto HatchBrush
    Dim oHatchBrush As New HatchBrush(HatchStyle.Vertical, Color.Fuchsia,
Color.Aqua)

    ' dibujar y pintar un polígono
    Dim oGraphics As Graphics = Me.CreateGraphics()
    oGraphics.FillEllipse(oHatchBrush, New Rectangle(25, 40, 150, 50))

End Sub
```

Código fuente 542

Podemos emplear un bitmap como base para la zona de relleno que tendrá que pintarse, para ello usaremos la clase TextureBrush, pasándole como parámetro un objeto Bitmap, que previamente habremos creado, y que contendrá un fichero con la textura necesaria. Ver Código fuente 543.

```
Private Sub mnuBrushTextura_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles mnuBrushTextura.Click

    ' crear un bitmap
    Dim oBitmap As New Bitmap("textura1.bmp")

    ' crear una brocha de textura
    Dim oTextureBrush As New TextureBrush(oBitmap)

    ' dibujar una figura y pintarla con la brocha de textura
    Dim oGraphics As Graphics = Me.CreateGraphics()
    oGraphics.FillEllipse(oTextureBrush, New Rectangle(220, 15, 100, 75))

End Sub
```

Código fuente 543

Para efectos avanzados de relleno, consistentes en degradados de colores, utilizaremos las clases LinearGradientBrush y PathGradientBrush. Una vez creado un objeto Brush de estas clases, aplicaremos un conjunto de colores que serán mezclados para crear un efecto de degradado o fundido en el área a pintar, mediante el constructor y/o las propiedades de la clase que corresponda. Ver Código fuente 544.

```
' pintar figura con brocha LinearGradientBrush
Private Sub mnuBrushLinearG_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles mnuBrushLinearG.Click

    ' crear brocha de tipo LinearGrad.
    Dim oLGB As New LinearGradientBrush(New Rectangle(10, 50, 40, 60),
Color.Aquamarine, Color.Azure, LinearGradientMode.Horizontal)

    ' crear array de coordenadas
    Dim oPuntos(2) As Point
```

```
    oPuntos(0) = New Point(20, 200)
    oPuntos(1) = New Point(75, 100)
    oPuntos(2) = New Point(140, 220)

    ' obtener contexto gráfico
    Dim oGraphics As Graphics = Me.CreateGraphics()

    ' dibujar y pintar una curva cerrada
    oGraphics.FillClosedCurve(oLGB, oPuntos)

End Sub

'-----------------------------------------
' pintar figura con brocha PathGradientBrush
Private Sub mnuBrushPathG_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles mnuBrushPathG.Click

    ' array de coordenadas
    Dim oPuntos(2) As Point
    oPuntos(0) = New Point(100, 150)
    oPuntos(1) = New Point(175, 80)
    oPuntos(2) = New Point(210, 150)

    ' crear brocha de tipo PathGradient,
    ' y configurar el objeto
    Dim oPGB As New PathGradientBrush(oPuntos)
    oPGB.CenterColor = Color.Indigo
    oPGB.SurroundColors = New Color() {Color.Beige, Color.LightGreen}

    ' crear gráfico y pintar polígono
    Dim oGraphics As Graphics = Me.CreateGraphics()
    oGraphics.FillPolygon(oPGB, oPuntos)

End Sub
```

Código fuente 544

La Figura 328 muestra todas las formas dibujadas con objetos de los tipos derivados de Brush.

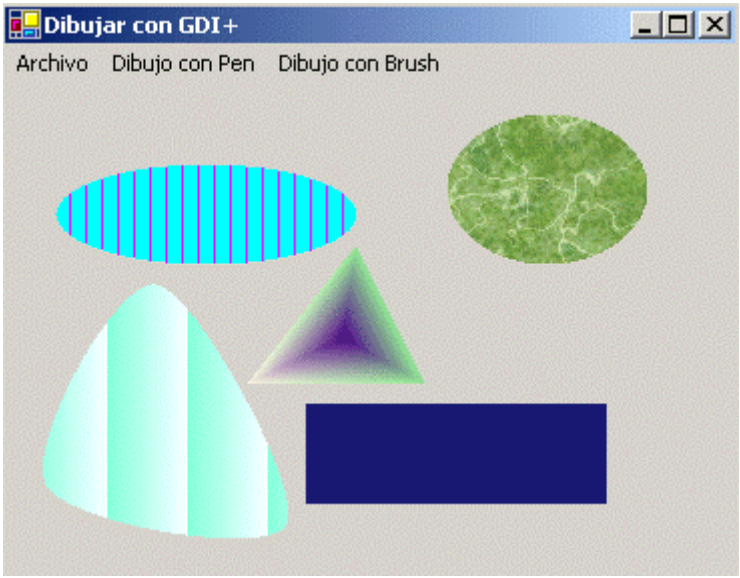


Figura 328. Figuras dibujadas y pintadas con objetos derivados de Brush.

Dibujo de texto en el formulario

Aparte de los controles que nos permiten visualizar y editar texto en un formulario, como Label, TextBox, etc., podemos realizar operaciones de dibujo de texto en la superficie del formulario, empleando el método DrawString( ) de la clase Graphics.

El texto visualizado mediante esta técnica no es evidentemente editable, se encamina fundamentalmente a mostrar texto con efectos adicionales que no podríamos conseguir mediante los controles típicos.

En el formulario de nuestro ejemplo, la opción de menú *Texto + Dibujar texto*, crea un objeto HatchBrush con un tramado específico, un objeto Font de una determinada familia, y con ambos elementos, pinta el texto mediante un objeto Graphics. Ver el Código fuente 545.

```
Private Sub mnuTextoTxt_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles mnuTextoTxt.Click

    ' crear un objeto Brush con efectos
    Dim oBrush As New HatchBrush(HatchStyle.Wave, Color.Aqua, Color.DarkGreen)

    ' crear el tipo de letra
    Dim oFont As New Font(New FontFamily("Georgia"), 50)

    ' obtener el dispositivo gráfico del formulario
    ' y pintar el texto
    Dim oGraf As Graphics = Me.CreateGraphics()
    oGraf.DrawString("Texto en modo gráfico", _
        oFont, oBrush, New RectangleF(20, 20, 500, 200))

End Sub
```

Código fuente 545

La Figura 329 muestra el texto dibujado en el formulario.

Figura 329. Dibujo de texto empleando clases de manipulación de gráficos.

Personalización de la imagen de fondo del formulario

En un tema anterior explicamos que la propiedad BackgroundImage nos permite asignar un fichero de imagen para el fondo del formulario. La facilidad que proporciona esta propiedad tiene desafortunadamente su lado negativo, ya que el tamaño de la imagen asignada, no se adapta automáticamente al del formulario.

Por tal motivo, en este apartado vamos a proporcionar al lector un par de técnicas, que le permitirán crear imágenes de fondo para formularios con ajuste dinámico, para que la imagen adapte sus dimensiones a las del formulario, cuando este cambie su tamaño en tiempo de ejecución.

Manipulación de los eventos de pintado en la clase Form

El primer ejemplo, contenido en el proyecto FondoFormManual (hacer clic aquí para acceder a este ejemplo), se basa en la codificación del evento Paint( ) de la clase Form.

Dentro del procedimiento manipulador de este evento, creamos un objeto Bitmap que contenga la referencia hacia un fichero con el gráfico a mostrar de fondo, y con un objeto Graphics, obtenido del parámetro EventArgs del evento, pintamos el objeto Bitmap. Ver el Código fuente 546.

```
Private Sub Form1_Paint(ByVal sender As Object, ByVal e As
System.Windows.Forms.PaintEventArgs) Handles MyBase.Paint

    ' crear un objeto bitmap
    Dim oBitmap As New Bitmap("LogoWin.gif")

    ' pintar el objeto con el contexto
    ' gráfico del formulario;
    ' el area a pintar abarca desde la
    ' coordenada 0,0 y toma el ancho y alto
    ' del formulario de su propiedad Size
    Dim oGraphics As Graphics = e.Graphics
    oGraphics.DrawImage(oBitmap, 0, 0, Me.Size.Width, Me.Size.Height)

End Sub
```

Código fuente 546

Queda todavía un paso más, ya que aunque la imagen se muestra como fondo del formulario, si redimensionamos este, sólo se repinta la parte nueva redimensionada, produciendo un efecto no deseado. Ver Figura 330.

Para conseguir que se pinte por completo toda la imagen, debemos invalidar la zona gráfica del formulario mediante su método Invalidate( ), en el evento Resize, el cual es provocado cada vez que cambia el formulario de tamaño. Después de escribir el Código fuente 547, cuando volvamos a ejecutar el ejemplo, la imagen de fondo se adaptará en todo momento a las medidas del formulario.

Figura 330. Imagen de fondo con repintado irregular.

```
Private Sub Form1_Resize(ByVal sender As Object, ByVal e As System.EventArgs)
Handles MyBase.Resize

    ' cada vez que cambie el tamaño del formulario
    ' invalidamos su área de dibujo para que
    ' el evento Paint pinte toda la superficie
    Me.Invalidate()

End Sub
```

Código fuente 547

Empleo del control PictureBox

Después de la técnica compleja (sólo en parte), vamos ahora con el modo fácil para crear una imagen de fondo, que pasa por el uso del control PictureBox.

Este control nos permite la visualización de imágenes en el formulario de un modo sencillo, ya que toda la mecánica de generación la lleva incorporada, con lo que el programador se despreocupa de la manipulación del gráfico a mostrar.

El proyecto FondoFormPicB (hacer clic aquí para acceder a este ejemplo) contiene esta aplicación para que el lector pueda realizar las pruebas que requiera.

Una vez insertado un PictureBox en el formulario, asignaremos a su propiedad Dock el valor Fill; de esta forma el control ocupará por completo la superficie del formulario.

A continuación asignaremos el fichero con la imagen a la propiedad Image, y por último, tenemos que establecer la propiedad SizeMode al valor StretchImage. Esto será todo, por lo que si ejecutamos el programa, la imagen quedará en todo momento ajustada al formulario al igual que en el anterior caso. Ver Figura 331.

Figura 331. Imagen de fondo del formulario empleando un PictureBox.

Manipulando el grado de opacidad del formulario

La propiedad Opacity de la clase Form, contiene un valor numérico de tipo Double que permite establecer el grado de opacidad del formulario. Cuando esta propiedad contenga el valor 1, el formulario se mostrará en la forma habitual; pero si el valor es 0, el formulario será transparente.

Podemos asignar valores intermedios de modo que hagamos parcialmente transparente el formulario, mostrando los elementos que quedan bajo el mismo. Debemos tener en cuenta que cuando asignemos este valor mediante la ventana de propiedades del formulario en modo de diseño, el número asignado será el porcentaje de opacidad. La Figura 332 muestra la ventana de propiedades para este caso con un setenta y cinco por ciento de opacidad para el formulario.

Figura 332. Propiedad Opacity establecida desde la ventana de propiedades del IDE.

La Figura 333 muestra este formulario con el anterior porcentaje de opacidad.

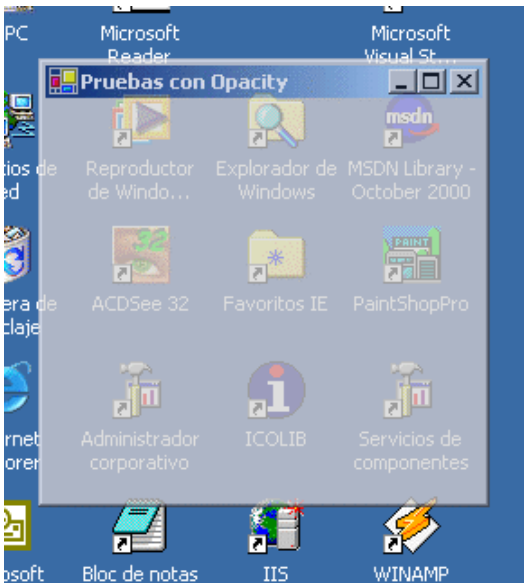


Figura 333. Formulario parcialmente transparente debido a la propiedad Opacity.

El Código fuente 548 establece por código la opacidad del formulario a un grado del cuarenta y cinco por ciento, en el evento DoubleClick del formulario.

```
Private Sub Form1_DoubleClick(ByVal sender As Object, ByVal e As System.EventArgs)
Handles MyBase.DoubleClick

    Me.Opacity = 4.5

End Sub
```

Código fuente 548

El proyecto Opacidad (hacer clic aquí para acceder a este ejemplo) contiene un control TrackBar, que con el estilo de un control de volumen, nos va a permitir graduar el nivel de opacidad del formulario.

Mediante las propiedades Maximum y Minimum establecemos el rango de opacidad respectivamente a diez y cero.

En la propiedad Value asignaremos el valor 10, para partir de la máxima opacidad e ir disminuyendo.

Como efectos visuales de este control, las propiedades Orientation y TickStyle nos permiten establecer la orientación del indicador de posición y su apariencia.

Finalmente, el evento Scroll se producirá cada vez que movamos el indicador de posición en el control, ejecutando el código de su procedimiento manipulador, que vemos en el Código fuente 549.

```
Private Sub tkbOpaco_Scroll(ByVal sender As Object, ByVal e As System.EventArgs)
Handles tkbOpaco.Scroll

    ' cada vez que se mueve el desplazador
    ' del TrackBar se modifica la opacidad
    Select Case Me.tkbOpaco.Value
        Case 0
            Me.Opacity = 0

        Case 10
            Me.Opacity = 1

        Case Else
            Me.Opacity = "0," & Me.tkbOpaco.Value
    End Select

End Sub
```

Código fuente 549

La Figura 334 muestra este ejemplo en ejecución.

Esta útil característica de los formularios nos permite, por ejemplo, proporcionar un efecto de fundido durante su proceso de cierre. Para conseguirlo, escribiremos el manipulador para el evento Closing del formulario, que es producido cuando el formulario está a punto de cerrarse; en dicho procedimiento de evento, cancelaremos en ese momento el cierre del formulario, crearemos un temporizador que conectaremos con un manipulador del evento Tick, y lo pondremos en marcha.

Figura 334. Utilizando un TrackBar para graduar la opacidad de un formulario.

En el procedimiento del evento Tick, crearemos el efecto de fundido, disminuyendo el grado de opacidad del formulario hasta hacerlo invisible, punto en el que cerraremos el formulario. Esto hará que se vuelva a pasar por el evento Closing, pero en esta ocasión, como la opacidad del formulario estará a cero, no se volverá a realizar el proceso de fundido. Todo esto lo vemos en el Código fuente 550.

```
Private Sub Form1_Closing(ByVal sender As Object, ByVal e As
System.ComponentModel.CancelEventArgs) Handles MyBase.Closing

    Dim oTiempo As Timer

    ' el formulario debe tener el valor 1
    ' en Opacity para conseguir el efecto
    If Me.Opacity <> 0 Then
        ' cancelamos el cierre del formulario
        e.Cancel = True

        ' iniciamos un temporizador cada medio segundo
        oTiempo = New Timer()
        oTiempo.Interval = 500
        ' conectamos el temporizador con un manipulador
        ' de su evento Tick
        AddHandler oTiempo.Tick, AddressOf TickTiempo
        oTiempo.Start()
    End If

End Sub

Private Sub TickTiempo(ByVal sender As Object, ByVal e As EventArgs)

    ' este manipulador del evento del temporizador,
    ' cada medio segundo irá disminuyendo el grado
    ' de opacidad del formulario hasta hacerlo invisible
    Static dbContador As Double = 1
    dbContador -= 0.1
    Me.Opacity = dbContador

    ' cuando el formulario es invisible
    If Me.Opacity = 0 Then
        ' parar el temporizador
        CType(sender, Timer).Stop()
        ' cerrar el formulario
        Me.Close()
    End If
```

```
End Sub
```

Código fuente 550

Modificación de la forma del formulario

Las capacidades gráficas del GDI+ nos permiten dotar a los formularios de efectos hasta la fecha muy difíciles de conseguir, si no se conocía en profundidad el API de Windows.

Aspectos como la transparencia del formulario tratada en el apartado anterior, o el cambio en la forma estándar del formulario, que veremos seguidamente.

Para cambiar el aspecto rectangular de un formulario debemos, en primer lugar, importar en nuestro programa, el espacio de nombres System.Drawing.Drawing2D.

A continuación crearemos un objeto de la clase GraphicsPath, que representa un conjunto de líneas y curvas conectadas. A este objeto le añadiremos una forma mediante alguno de sus métodos AddXXX( ).

Finalmente, crearemos un objeto de tipo Region, al que pasaremos el objeto GraphicsPath con la forma creada previamente. Esto creará una región con dicha forma, que asignaremos a la propiedad Region del formulario, consiguiendo de esta manera, modificar el aspecto del formulario en cuanto a su forma.

El proyecto FormasForm (hacer clic aquí para acceder a este ejemplo) muestra un formulario con dos botones que cambian la forma del formulario a un círculo y triángulo al ser pulsados. El Código fuente 551 muestra los eventos Click de ambos botones.

```
Imports System.Drawing.Drawing2D
'....
'....
Private Sub Circulo_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Circulo.Click

    ' crear un objeto gráfico para conectar líneas y curvas
    Dim oGPath As GraphicsPath = New GraphicsPath()
    ' añadir un círculo al objeto gráfico
    oGPath.AddEllipse(New Rectangle(0, 0, 200, 260))

    ' crear una región con el objeto gráfico
    ' y asignarla a la región del formulario
    Me.Region = New Region(oGPath)

End Sub

Private Sub btnPoligono_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnPoligono.Click

    ' mostrar el formulario con el aspecto
    ' de un triángulo
    Dim oPuntos(2) As Point
    oPuntos(0) = New Point(-20, -20)
    oPuntos(1) = New Point(-20, 200)
    oPuntos(2) = New Point(220, 90)

    Dim oGPath As GraphicsPath = New GraphicsPath()
```

```
    oGPath.AddPolygon(oPuntos)

    Me.Region = New Region(oGPath)

End Sub
```

Código fuente 551

La Figura 335 muestra el formulario con forma de elipse.

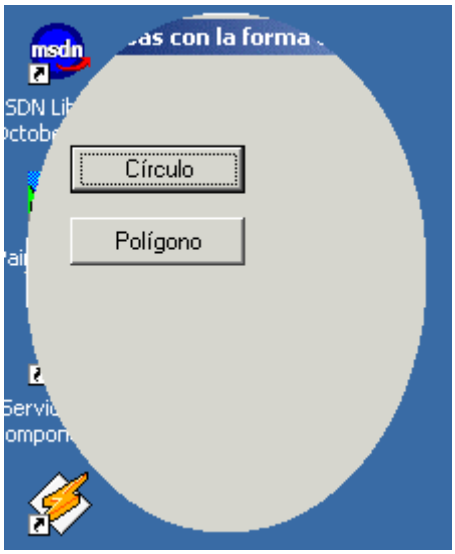


Figura 335. Formulario con forma de elipse.

La Figura 336 muestra el formulario con forma de triángulo.

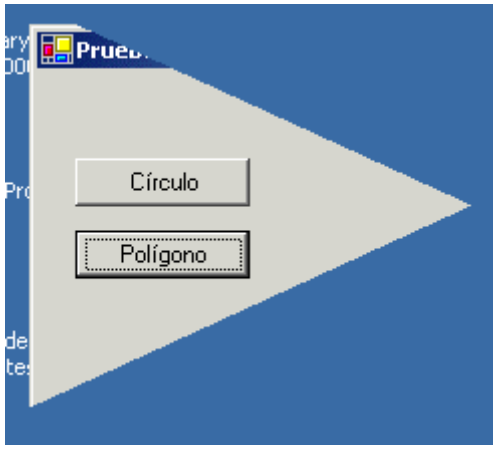


Figura 336. Formulario con forma triangular.

Integrando elementos. Un visualizador de gráficos

Para finalizar con el presente tema, vamos a realizar, a modo de práctica, una aplicación que contendrá un formulario para visualizar archivos de imagen. De esta forma practicaremos con algunos de los puntos ya tratados, y nos servirá para presentar dos nuevos controles: TreeView y ListView.

El ejemplo comentado en este apartado se encuentra en el proyecto VisualizadorGraf (hacer clic aquí para acceder a este ejemplo), y consiste en crear un formulario con un ComboBox que permita, en primer lugar, seleccionar una unidad lógica del equipo. Una vez hecho esto, se llenará el TreeView con la lista de directorios de la unidad elegida. Al seleccionar un directorio del TreeView, se rellenará el ListView con una lista de ficheros de tipo gráfico, de los que al seleccionar uno, se cargará su contenido en un PictureBox; al realizar dicha carga, podremos optar por ajustar la imagen a las dimensiones del PictureBox o bien mantener el tamaño original de la imagen. Debido a que algunos de estos controles necesitan de imágenes asociadas como iconos, utilizaremos también un control ImageList para almacenar estos iconos.

El control TreeView muestra una lista de elementos dispuestos en forma de árbol o nodos expandibles. La propiedad Nodes es una colección que contiene los nodos del control, que en este caso rellenaremos desde el código del programa. Las propiedades ImageIndex y SelectedImageIndex muestran respectivamente una de las imágenes del control ImageList asociado, con el icono indicativo de si un nodo se encuentra o no seleccionado.

Por otro lado, el control ListView consiste en un ListBox al que podemos asociar una imagen para cada elemento o item que muestra en la lista.

Para poder mostrar los elementos de un ListView de esta forma estableceremos la propiedad View al valor Details; crearemos una columna en su propiedad Columns, y asociaremos el control ImageList a las propiedades Large/Small/StateImageList. También es conveniente que la propiedad MultiSelect esté a False para poder seleccionar sólo un elemento en cada ocasión. El llenado de este control también lo haremos desde el código de la aplicación. La Figura 337 muestra el formulario de este proyecto una vez completada su fase de diseño.

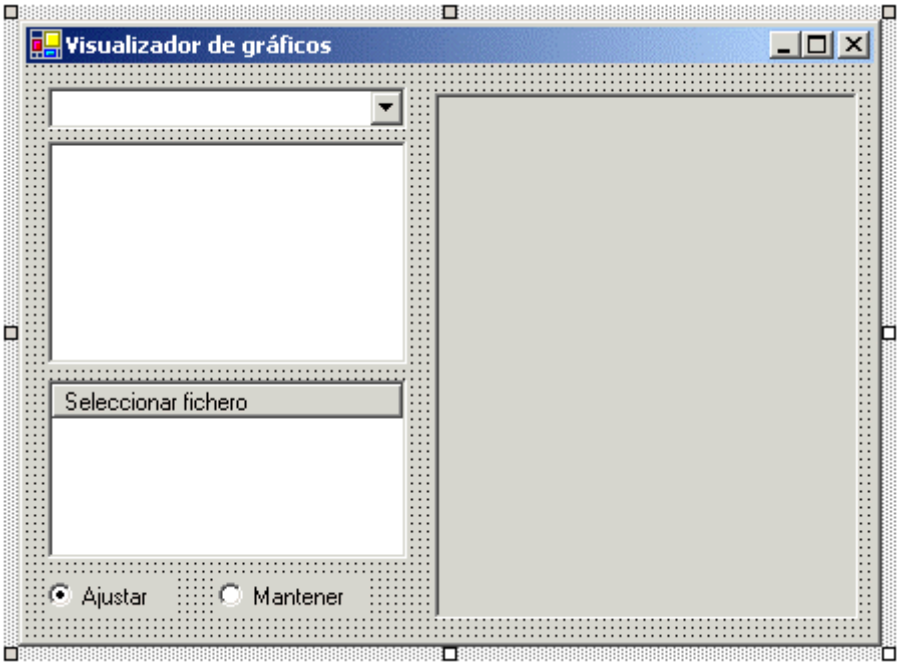


Figura 337. Formulario del visualizador de gráficos.

Pasando al código de la clase del formulario, debido a que vamos a trabajar con objetos que manipulan directorio y ficheros, importaremos en la cabecera del fichero de código el espacio de nombres System.IO. En el evento Load del formulario, cargaremos el ComboBox con las unidades lógicas

detectadas por el sistema, empleando el objeto Environment del sistema, y su método GetLogicalDrives( ). Ver Código fuente 552.

```
Imports System.IO

Public Class Form1
    Inherits System.Windows.Forms.Form

    Private Sub Form1_Load(ByVal sender As Object, ByVal e As System.EventArgs)
Handles MyBase.Load
        ' al cargar el formulario llenar
        ' el combo con las letras de unidad
        Dim sUnidades() As String
        sUnidades = System.Environment.GetLogicalDrives()
        Me.cboUnidades.Items.AddRange(sUnidades)
    End Sub
'....
```

Código fuente 552

Al seleccionar un elemento en el ComboBox, se produce su evento SelectedIndexChanged, en el que nos ocupamos de obtener los directorios raíz de la unidad lógica seleccionada, y con dicha información, rellenar el TreeView. Ver Código fuente 553.

```
Private Sub cboUnidades_SelectedIndexChanged(ByVal sender As Object, ByVal e As
System.EventArgs) Handles cboUnidades.SelectedIndexChanged
    ' este evento se dispara cuando se cambia
    ' el elemento seleccionado del combo

    ' obtener los directorios raíz de la unidad seleccionada
    Dim oDirUnidadSelec As New DirectoryInfo(Me.cboUnidades.Text)
    Dim sDirectorios() As DirectoryInfo
    sDirectorios = oDirUnidadSelec.GetDirectories()

    ' vaciar el treeview
    Me.tvDirectorios.Nodes.Clear()

    ' dibujar cada nombre de directorio raíz en el treeview
    Dim oDirInfo As DirectoryInfo
    Dim oNodo As TreeNode
    For Each oDirInfo In sDirectorios
        oNodo = New TreeNode(oDirInfo.FullName, 0, 1)
        Me.tvDirectorios.Nodes.Add(oNodo)
    Next

End Sub
```

Código fuente 553

El siguiente paso lógico es la selección de un directorio en el TreeView. Cuando esto suceda, se provocará el evento AfterSelect de dicho control; en él comprobaremos si existen directorios anidados al seleccionado, y al mismo tiempo, llenaremos el ListView con los nombres de ficheros del directorio seleccionado, asociando a cada elemento de la lista, una imagen del control ImageList por el número de orden que ocupa la imagen en la lista. Ver Código fuente 554.

```
Private Sub tvDirectorios_AfterSelect(ByVal sender As Object, ByVal e As
System.Windows.Forms.TreeViewEventArgs) Handles tvDirectorios.AfterSelect
    ' si el nodo pulsado no está expandido...
    If Not e.Node.IsExpanded Then
        ' comprobar si tiene subdirectorios
        Dim oSubDirInfo As DirectoryInfo
        oSubDirInfo = New DirectoryInfo(e.Node.FullPath)

        ' obtener subdirectorios del directorio seleccionado
        Dim oSubDirectorios() As DirectoryInfo
        oSubDirectorios = oSubDirInfo.GetDirectories()

        ' crear nodos para cada subdirectorio en el treeview
        Dim oSubDirI As DirectoryInfo
        Dim oNodo As TreeNode
        For Each oSubDirI In oSubDirectorios
            oNodo = New TreeNode(oSubDirI.Name, 0, 1)
            e.Node.Nodes.Add(oNodo.Text)
        Next

        ' obtener los archivos del subdirectorio
        Dim oArchivos() As FileInfo
        oArchivos = oSubDirInfo.GetFiles()

        ' limpiar el listview
        Me.lstFicheros.Items.Clear()

        ' rellenar el listview con los nombres de archivo
        ' que tengan tipo gráfico
        Dim oArchInfo As FileInfo
        For Each oArchInfo In oArchivos
            Select Case oArchInfo.Extension.ToUpper()
                Case ".BMP", ".PNG", ".WMF"
                    Me.lstFicheros.Items.Add(oArchInfo.Name, 3)

                Case ".JPG", ".JPEG"
                    Me.lstFicheros.Items.Add(oArchInfo.Name, 4)

                Case ".GIF"
                    Me.lstFicheros.Items.Add(oArchInfo.Name, 5)
            End Select
        Next
    End If
End Sub
```

Código fuente 554

Finalmente, ya sólo queda comprobar cuándo se pulsa en uno de los ficheros de imagen del ListView, cosa que haremos con su evento SelectedIndexChanged. Al producirse esta situación, lo que haremos será invalidar el área de dibujo del PictureBox, forzando a que se desencadene su evento Paint, en donde realmente realizaremos la carga de la imagen. A pesar de todo, en el Código fuente 555 también se acompaña el código para hacer una carga directa de la imagen en el evento sobre el que nos encontramos.

```
Private Sub lstFicheros_SelectedIndexChanged(ByVal sender As Object, ByVal e As
System.EventArgs) Handles lstFicheros.SelectedIndexChanged

    If Me.lstFicheros.SelectedItems.Count > 0 Then
        ' CARGA MANUAL EN PICTUREBOX:
        ' si invalidamos la región gráfica del picturebox
        ' obligamos a que se produzca su evento Paint,
```

```
        ' y en ese evento escribimos el código que carga
        ' la imagen en el control
        Me.picImagen.Invalidate()

        ' CARGA AUTOMÁTICA DE IMAGEN:
        ' escribimos en este evento el código para
        ' asignar una imagen al picturebox
        'Me.picImagen.Image = New Bitmap( _
        '   Me.tvDirectorios.SelectedNode.FullPath & "\" & _
        '  Me.lstFicheros.SelectedItems(0).Text)
    End If

End Sub
```

Código fuente 555

En el evento Paint del PictureBox, mostramos la imagen seleccionada, ajustada al tamaño del control o con su propio tamaño, según el RadioButton seleccionado del formulario. Adicionalmente, para el caso en el que se redimensione el formulario, también invalidamos el PictureBox, de manera que la imagen que actualmente se esté mostrando será recargada. Veámoslo en el Código fuente 556.

```
Private Sub picImagen_Paint(ByVal sender As Object, ByVal e As
System.Windows.Forms.PaintEventArgs) Handles picImagen.Paint
    ' si el nodo seleccionado tiene contenido
    If Not (IsNothing(Me.tvDirectorios.SelectedNode)) Then
        ' crear imagen a partir del fichero seleccionado
        Dim oBitmap As New Bitmap( _
            Me.tvDirectorios.SelectedNode.FullPath & _
            "\" & Me.lstFicheros.SelectedItems(0).Text)

        ' obtener dispositivo gráfico del picturebox
        Dim oGraf As Graphics = e.Graphics

        If Me.rbtAjustar.Checked Then
            ' dibujar imagen ajustada al picturebox
            oGraf.DrawImage(oBitmap, 0, 0, _
                Me.picImagen.Size.Width, Me.picImagen.Size.Height)
        Else
            ' dibujar imagen con su tamaño original
            oGraf.DrawImage(oBitmap, 0, 0, _
                oBitmap.Size.Width, oBitmap.Size.Height)
        End If
    End If

End Sub

Private Sub picImagen_Resize(ByVal sender As Object, ByVal e As System.EventArgs)
Handles picImagen.Resize
    ' al redimensionar, invalidar área gráfica del picturebox
    Me.picImagen.Invalidate()
End Sub
```

Código fuente 556

Terminada la escritura de código del programa, sólo queda ejecutarlo para comprobar su resultado, como muestra la Figura 338.

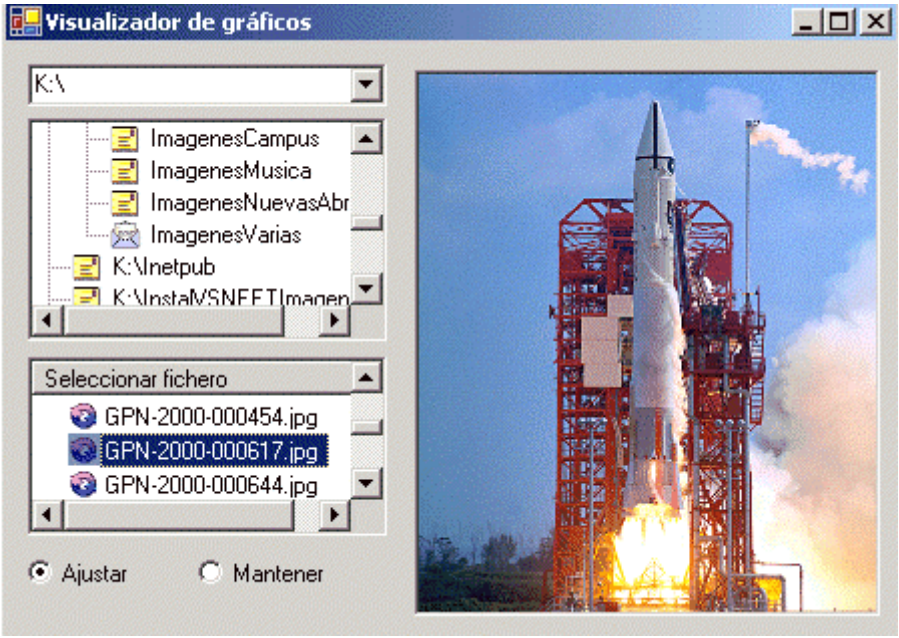


Figura 338. Visualizador de gráficos en ejecución.

36

Acceso a datos con ADO .NET

En los siguientes temas vamos a tratar el acceso a datos desde VB.NET, haciendo uso del nuevo modelo de acceso a datos incluido en la plataforma .NET Framework: ADO .NET.

Mostraremos las tareas básicas para el acceso a datos desde aplicaciones basadas en formularios Windows, empleando la tecnología proporcionada por ADO .NET.

ADO .NET es la nueva versión del modelo de objetos ADO (ActiveX Data Objects), es decir, la estrategia que ofrece Microsoft para el acceso a datos. ADO .NET ha sido ampliado para cubrir todas las necesidades que ADO no ofrecía, y está diseñado para trabajar con conjuntos de datos desconectados, lo que permite reducir el tráfico de red. ADO .NET utiliza XML como formato universal de transmisión de los datos.

ADO .NET posee una serie de objetos que son los mismos que aparecen en la versión anterior de ADO, como pueden ser el objeto Connection o Command, e introduce nuevos objetos tales como el objeto DataReader, DataSet o DataView.

ADO .NET se puede definir como:

- Un conjunto de interfaces, clases, estructuras y enumeraciones que permiten el acceso a datos desde la plataforma .NET de Microsoft

- La evolución lógica del API ADO tradicional de Microsoft

- Permite un modo de acceso desconectado a los datos, los cuales pueden provenir de múltiples fuentes de datos, de diferente arquitectura de almacenamiento

- Soporta un completo modelo de programación y adaptación, basado en el estándar XML

Seguidamente vamos a realizar una descripción genérica de la arquitectura de ADO .NET, y más tarde veremos como utilizarlo desde aplicaciones VB.NET

Comparativa de ADO /ADO .NET

Como punto de partida para comprender la importancia del nuevo diseño de ADO .NET, analizaremos los aspectos distintivos entre ADO .NET y el modelo ADO vigente hasta la fecha. Las diferencias existentes son muchas, y van desde el mismo diseño de los interfaces de las clases, hasta el nivel estructural de los componentes, pasando por el modo en cómo manejan la información. La Tabla 31 muestra estas diferencias.

Clases de objetos mas estructuradas	El modelo de objetos de ADO .NET es mucho más rico que el de ADO. Incorpora nuevas clases de datos y encapsula mucha más potencia de uso a la par que corrige pequeños defectos de las versiones anteriores.
Independiente del lenguaje	Aprovechando la nueva arquitectura de servicios de la plataforma .NET, ADO .NET puede ser usado como un servicio del sistema, pudiendo ser utilizado por cualquier aplicación escrita en cualquier lenguaje.
Representaciones en memoria de los datos	En ADO .NET se emplea el DataSet, mientras que en ADO se emplea el Recordset. No es sólo un cambio de nombre. En memoria y en rendimiento las cosas han cambiado mucho.
Número de tablas	Un Recordset de ADO sólo puede contener un único resultado (fruto de consultas complejas o no). En cambio, un DataSet puede representar un espacio de múltiples tablas simultáneamente. Esto permite obtener una representación de espejo de la base de datos a la que representa.
Navegación mejorada	En ADO sólo nos podemos mover secuencialmente fila a fila. En ADO .NET podremos navegar por filas en un DataSet, y consecuentemente avanzar en una fila/conjunto de filas de otro DataSet asociado a través de una colección de relaciones.

Acceso a datos Off-line (Desconectados)	En ADO .NET todos los accesos a datos se realizan en contextos desconectados de la base de datos. En ADO, es posible tener estructuras desconectadas, pero es una elección a tomar. Por defecto, ADO está pensado para contextos con conexión. Además, ADO .NET realiza todos los accesos a los servicios de datos a través del objeto DataSetCommand, lo que permite personalizar cada comando en función del rendimiento y la necesidad del programa (en ADO, el Recordset puede modificar datos a través de su API de cursores, lo que muchas veces no es óptimo).
Compartimiento de datos entre capas, o bien, entre componentes	El traspaso de un recordset Off-line en ADO es más complejo y pesado pues es necesario establecer un vínculo RPC y un proceso de COM marshalling (verificación de tipos de datos remotos). En ADO .NET para la comunicación entre componentes y capas se emplea un simple stream XML.
Tipos de datos más ricos	Debido al COM Marshalling los tipos de datos que se pueden usar vía RPC están muy limitados. En ADO .NET, gracias a los flujos XML, es posible enviar cualquier tipo de dato de manera sencilla.
Rendimiento	Tanto ADO como ADO .NET requieren de muchos recursos cuando se habla de envíos masivos de datos. El stress del sistema es proporcional al número de filas que se quieren procesar. Pero ADO .NET permite utilizar mucho más eficientemente los recursos, pues no tiene que realizar el COM Marshalling de datos, en lo que ADO sí (y eso supone un canal más de envío de datos de control que ADO .NET ahorra).
Penetración de firewalls	Por motivos de seguridad, las empresas suelen bloquear los sistemas de comunicaciones vía RPC. O bien, implementan pesados mecanismos de control de acceso y envío de la información. Lo que complica los diseños, sus rendimientos y su instalación y distribución. En ADO .NET, puesto que no se realizan llamadas al sistema, sino que se envían flujos de datos en XML, se simplifica enormemente la configuración de estos dispositivos.

Tabla 31

De los anteriores puntos podemos obtener muy buenas conclusiones en cuanto a las mejoras introducidas en el nuevo modelo ADO .NET. Se puede resumir en un mejor mecanismo de comunicación entre procesos gracias a XML y una independencia del cliente con respecto al servidor, que posibilita el funcionamiento autónomo de la aplicación (mejor tolerancia a fallos, independencia del estado de la red).

Beneficios de ADO .NET

ADO .NET ofrece una buena cantidad de mejoras respecto a modelos anteriores de ADO. Los beneficios los podemos agrupar en las categorías descritas a continuación.

Interoperabilidad

Las aplicaciones basadas en ADO .NET obtienen ventaja de la flexibilidad y la masiva aceptación del estándar XML para el intercambio de datos. Puesto que XML es el estándar de envío de información entre capas, cualquier componente capaz de Interpretar los datos XML puede acceder a la información de ADO .NET, se encuentre donde se encuentre, y procesarla. Además, puesto que la información se envía en flujos de XML, no importa la implementación empleada para enviar o recoger la información –así como la plataforma empleada-. Simplemente se exige a los componentes que reconozcan el formato XML empleado para el proceso, envío y recepción de un DataSet.

Mantenimiento

En el ciclo de vida de una aplicación los cambios poco sustanciales y modestos son permisibles. Pero cuando es necesario abordar un cambio estructural o arquitectónico del sistema, la tarea se vuelve demasiado compleja y a veces inviable. Esto es una gran desventaja de los sistemas actuales, pues muchas veces se trata de una cuestión de actualización de los procesos de la propia empresa. Además, cuanto más se aumenta el proceso de la operativa de la empresa, las necesidades de proceso crecen hasta desbordar las máquinas. Es por ello que se separa la estructura de un programa en varias capas. Una de esas capas es la de datos, que es fundamental desarrollar correctamente. Gracias a los DataSets, la tarea de portar y aumentar los procesos de datos y de negocio será mas sencillo: el intercambio de información a través de XML, hace que sea más sencilla la tarea de estructurar en más capas la aplicación, convirtiéndola en más modular y fácil de mantener.

Programación

Los programadores pueden acceder a un API de programación estructurado, de fuerte tipificado y que además se concentra en la correcta forma de presentar los datos. Centra en la estructura del lenguaje lo que un programador necesita para diseñar los programas sin dar muchos rodeos. El Código fuente 557 muestra un ejemplo de código sin tipificar:

```
'....
If CosteTotal > Table("Cliente")("Luis").Column("CreditoDisponible") Then
'....
```

Código fuente 557

Como se puede observar, aparecen nombres de objetos genéricos del sistema que complican la lectura del código, a la par que los operadores complican también la visión de la secuencia de acceso a los datos. Podríamos interpretar lo que hace gracias a que aparecen los nombres propios de los datos que necesitamos. El Código fuente 558 muestra un ejemplo un poco más tipificado:

```
`....
If CosteTotal > DataSet1.Cliente("Luis").CreditoDisponible Then
`....
```

Código fuente 558

El ejemplo es exactamente igual al anterior, pero en este caso, el código se centra más en los objetos reales que en el objeto del lenguaje en sí: las palabras *Table* y *Column* ya no aparecen. En su lugar vemos que aparecen los nombres de los objetos empleados de la vida real, lo que hace el código más legible. Si a esto unimos que los entornos ya son capaces de ayudarnos a escribir el código, todavía lo tenemos más sencillo, ya que podemos ver con nuestras palabras el modelo de objetos de datos que necesitamos en cada momento. Incluso a nivel de ejecución nos vemos respaldado por un sistema de control de tipos y errores que nos permitirán proporcionar una robustez innata, que antes no se tenía sin pasar por el uso de funciones externas.

Rendimiento

Puesto que trabajamos con objetos de datos desconectados, todo el proceso se acelera, ya que no tenemos que estar comunicándonos por Marshalling con el servidor. Además, gracias al modelo de XML la conversión de tipos no es necesaria a nivel de COM. Se reduce pues el ancho de banda disponible, se independiza más el cliente del servidor, y se descarga más a éste, que puede estar dedicado a otras tareas en lo que el cliente analiza sus datos.

Escalabilidad

Las aplicaciones Web tienen un número ilimitado de conexiones potenciales debido a la naturaleza de Internet. Los servidores son capaces de atender muy bien decenas y decenas de conexiones. Pero cuando hablamos de miles y millones, los servidores ya no son capaces de realizar correctamente su trabajo. Esto es debido a que por cada usuario se mantiene una memoria de proceso y conexión, un conjunto de bloqueos de recursos como puedan ser tablas, índices, etc., y una comprobación de sus permisos; todo ello consume tiempo y recursos.
ADO .NET favorece la escalabilidad, puesto que su modelo de conexión Off-Line evita que se mantengan los recursos reservados más tiempo del considerado necesario. Esto permite que más usuarios por unidad de tiempo puedan acceder a la aplicación sin problemas de tiempos. Además se pueden montar servicios en Cluster de alta disponibilidad que serán balanceados automáticamente por el sistema sin afectar a las conexiones ADO. Lo cual garantiza la ampliación del servicio sin representar un cambio de arquitectura de diseño.

Arquitectura de datos desconectados

ADO .NET está basado en una arquitectura desconectada de los datos. En una aplicación de datos se ha comprobado que mantener los recursos reservados mucho tiempo, implica reducir el número de

usuarios conectados y aumenta el proceso del sistema al mantener una política de bloqueos y transacciones. Al mismo tiempo, si la aplicación mantiene más de un objeto simultáneamente, se encuentra con el problema de tener que estar continuamente conectando con el servidor para alimentar las relaciones existentes entre ambas, subiendo y bajando información vía RPC.

Con ADO .NET se consigue estar conectado al servidor sólo lo estrictamente necesario para realizar la operación de carga de los datos en el DataSet. De esta manera se reducen los bloqueos y las conexiones a la mínima expresión. Se pueden soportar muchos más usuarios por unidad de tiempo y disminuyen los tiempos de respuesta, a la par que se aceleran las ejecuciones de los programas.

Tradicionalmente, el recoger información de una base de datos ha ido destinado a realizar un proceso con dicha información: mostrarla por pantalla, procesarla o enviarla a algún componente. Frecuentemente, la aplicación no necesita una única fila, sino un buen conjunto de ellas. Además, también frecuentemente, ese conjunto de filas procede no de una tabla sino de una unión de múltiples tablas (join de tablas). Una vez que estos datos son cargados, la aplicación los trata como un bloque compacto. En un modelo desconectado, es inviable el tener que conectar con la base de datos cada vez que avanzamos un registro para recoger la información asociada a ese registro (condiciones del join). Para solucionarlo, lo que se realiza es almacenar temporalmente toda la información necesaria donde sea necesario y trabajar con ella. Esto es lo que representa un DataSet en el modelo ADO .NET.

Un DataSet es una caché de registros recuperados de una base de datos que actúa como un sistema de almacenamiento virtual, y que contiene una o más tablas basadas en las tablas reales de la base de datos. Adicionalmente, almacena las relaciones y reglas de integridad existentes entre ellas para garantizar la estabilidad e integridad de la información de la base de datos. Muy importante es recalcar, que los DataSets son almacenes pasivos de datos, esto es, no se ven alterados ante cambios subyacentes de la base de datos. Es necesario recargarlos siempre que queramos estar *al día,* en cuanto a datos se refiere.

Una de las mayores ventajas de esta implementación, es que una vez obtenido el DataSet, éste puede ser enviado (en forma de flujo XML) entre distintos componentes de la capa de negocio, como si de una variable más se tratase, ahorrando así comunicaciones a través de la base de datos.

Una consecuencia lógica de este tipo de arquitecturas, es la de conseguir que los DataSets sean independientes de los orígenes de datos. Los drivers OLE-DB transformarán la consulta SQL en un cursor representado con una estructura XML, que es independiente del motor de la base de datos.

Esto nos permitirá trabajar con múltiples orígenes de datos, de distintos fabricante e incluso en formatos que no pertenezcan a bases de datos, por ejemplo, ficheros planos u hojas de cálculo, lo que representa un importante punto de compatibilidad y flexibilidad.

Si a esto unimos el hecho de que disponemos de un modelo consistente de objetos (xmlDOM) que es independiente del origen de datos, las operaciones de los DataSets no se verán afectadas por dicho origen.

La persistencia es un concepto muy interesante en el mundo del desarrollo. Es un mecanismo por el cual un componente puede almacenar su estado (valores de variables, propiedades, datos...en un momento concreto del tiempo) en un soporte de almacenamiento fijo. De manera, que cuando es necesario, se puede recargar el componente tal y como quedó en una operación anterior.

En un sistema de trabajo Off-Line como el que plantea ADO .NET, la persistencia es un mecanismo fundamental. Podemos cerrar la aplicación y mantener persistentes todos los DataSets necesarios, de manera que al reiniciarla, nos encontramos los DataSets tal y como los dejamos. Ahorrando el tiempo que hubiera sido necesario para recuperar de nuevo toda esa información del servidor. Optimizando todavía más el rendimiento del sistema distribuido.

El formato que emplea ADO .NET para almacenar su estado es XML. Puesto que ya es un estándar de la industria, esta persistencia nos ofrece las siguientes cualidades:

- La información puede estar accesible para cualquier componente del sistema que entienda XML.
- Es un formato de texto plano, no binario, que lo hace compatible con cualquier componente de cualquier plataforma, y recuperable en cualquier circunstancia.

DataSet

El API de ADO .NET proporciona una superclase, DataSet, que encapsula lo que sería la base de datos a un nivel lógico: tablas, vistas, relaciones, integridad entre todos ellos, etc., pero siempre con independencia del tipo de fabricante que la diseñó. Aquí se tiene el mejor concepto de datos desconectados: una copia en el cliente de la arquitectura de la base de datos, basada en un esquema XML que la independiza del fabricante, proporcionando al desarrollador la libertad de trabajo independiente de la plataforma. La Figura 339 muestra una representación de este tipo de objeto.

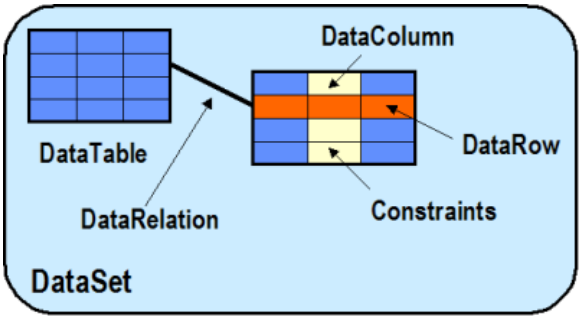


Figura 339. Esquema de un DataSet.

Esta clase se compone a su vez, de clases de soporte, que representan cada una, los elementos arquitecturales de la base de datos: tablas, columnas, filas, sus reglas de chequeo, sus relaciones, las vistas asociadas a la tabla, etc.

ADO .NET y XML

XML se ha convertido en la piedra angular de la informática distribuida de nuestros días. De ahí que gran parte de las motivaciones en cuanto a la redefinición del API de ADO, se deban a la adaptación de los objetos a un modelo de procesos que se apoya en documentos XML, no en objetos específicos de cada plataforma a partir de cursores. Esto permite que las clases de ADO .NET puedan implementar mecanismos de conversión de datos entre plataformas, lectura de datos de cualquier origen, habilitar mecanismos de persistencia en el mismo formato en el que se procesan., etc.

En esta redefinición, Microsoft ha puesto como intermediario entre un cliente y sus datos, un adaptador que transforma cada comando y cada dato en modelos de documentos XML. Tanto para consultas como para actualizaciones. Esto es lo que posibilita la nueva filosofía de acceso a datos desconectados de ADO .NET: primero se cargan en el cliente los documentos necesarios almacenándolos en DataSet, a partir de consultas a tablas, vistas, procedimientos, etc.; se nos da la posibilidad de trabajar con documentos, sin necesidad de estar continuamente consumiendo recursos de la red; y por último, se procesarán los cambios producidos enviándolos a la base de datos, el

adaptador tomará los cambios del documento, y los replicará al servidor. En la Figura 340 se puede ver un esquema de la relación entre ADO .NET y XML.

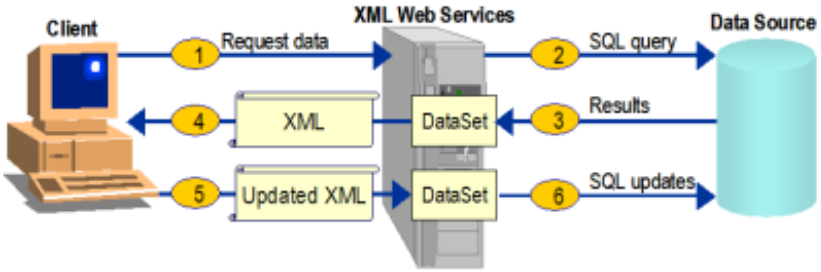


Figura 340. Esquema de relación entre ADO .NET y XML.

Una visión general de ADO .NET

ADO .NET es el modelo de objetos para el acceso a datos incluido en la jerarquía de clases de la plataforma .NET Framework. Se trata de una evolución de ADO, el anterior modelo para la gestión de datos, incluido en VB6.

ADO .NET ha sido ampliado para cubrir todas las necesidades que ADO no ofrecía. Está diseñado para trabajar con conjuntos de datos desconectados, lo que permite reducir el tráfico de red, utilizando XML como formato universal de transmisión de los datos.

ADO .NET posee una serie de objetos que son los mismos que aparecen en la versión anterior de ADO, como pueden ser el objeto Connection o Command, e introduce nuevos objetos tales como el objeto DataReader, DataSet o DataView. A continuación vamos a comentar brevemente los objetos principales que posee ADO .NET.

Los espacios de nombre que utiliza ADO .NET son principalmente System.Data y System.Data.OleDb o System.Data.SqlClient. System.Data ofrece las facilidades de codificación para el acceso y manejo de datos, mientras que System.Data.OleDb y System.Data.SqlClient contienen los proveedores; en el primer caso, los proveedores genéricos de OLE DB, y en el segundo, los proveedores nativos de SQL Server que ofrece la plataforma .NET.

Para el lector que haya seguido la evolución de la plataforma .NET, debemos puntualizar que estos espacios de nombres se denominaban System.Data.ADO y System.Data.SQL en la Beta 1 de la plataforma .NET.

El objeto Connection define el modo en cómo se establece la conexión con el almacén de datos. .NET Framework ofrece dos objetos Connection: SqlConnection y OleDbConnection, que se corresponden con las dos posibilidades de proveedores que disponemos.

Otro objeto importante dentro del modelo de objetos de ADO .NET es el objeto System.Data.DataSet (conjunto de datos). Este nuevo objeto representa un conjunto de datos de manera completa, pudiendo incluir múltiples tablas junto con sus relaciones. No debemos confundir el nuevo objeto DataSet con el antiguo objeto Recordset; el objeto DataSet contiene un conjunto de tablas y las relaciones entre las mismas, sin embargo el objeto Recordset contiene únicamente una tabla. Para acceder a una tabla determinada el objeto DataSet ofrece la colección Tables.

Para poder crear e inicializar las tablas del DataSet debemos hacer uso del objeto DataAdapter, que igualmente, posee las dos versiones, es decir, el objeto SqlDataAdapter para SQL Server y

OleDbDataAdapter genérico de OLE DB. En la Beta 1 de la plataforma .NET el objeto DataAdapter se denominaba DataSetCommand.

Al objeto DataAdapter le pasaremos como parámetro una cadena que representa la consulta que se va a ejecutar y que va a rellenar de datos el DataSet. Del objeto DataAdapter utilizaremos el método Fill(), que posee dos parámetros, el primero es el objeto DataSet que vamos rellenar con datos, y el segundo es una cadena que identifica el objeto DataTable (tabla) que se va a crear dentro del objeto DataSet como resultado de la ejecución de la consulta

Un DataSet puede contener diversas tablas, que se representan mediante objetos DataTable. Para mostrar el contenido de un DataSet, mediante Data Binding, por ejemplo, necesitamos el objeto DataView. Un objeto DataView nos permite obtener un subconjunto personalizado de los datos contenidos en un objeto DataTable. Cada objeto DataTable de un DataSet posee la propiedad DefaultView, que devuelve la vista de los datos por defecto de la tabla.

Otro objeto de ADO .NET es DataReader, que representa un cursor de sólo lectura y que sólo permite desplazamiento hacia adelante (read-only/forward-only), cada vez existe un único registro en memoria, el objeto DataReader mantiene abierta la conexión con el origen de los datos hasta que es cerrado. Al igual que ocurría con otros objetos de ADO .NET, de este objeto tenemos dos versiones, que como el lector sospechará se trata de los objetos SqlDataReader y OleDbDataReader.

Espacios de nombres y clases en ADO .NET

En el presente apartado vamos a enumerar brevemente los principales elementos que forman parte del API de ADO .NET.

Primero vamos a comentar los distintos espacios de nombres que constituyen la tecnología ADO .NET:

- **System.Data**. Clases genéricas de datos de ADO .NET. Integra la gran mayoría de clases que habilitan el acceso a los datos de la arquitectura .NET.

- **System.Data.SqlClient**. Clases del proveedor de datos de SQL Server. Permite el acceso a proveedores SQL Server en su versión 7.0 y superior.

- **System.Data.OleDb**. Clases del proveedor de datos de OleDB. Permite el acceso a proveedores .NET que trabajan directamente contra controladores basados en los ActiveX de Microsoft.

- **System.Data.SqlTypes**. Definición de los tipos de datos de SQL Server. Proporciona la encapsulación en clases de todos los tipos de datos nativos de SQL Server y sus funciones de manejo de errores, ajuste y conversión de tipos, etc.

- **System.Data.Common**. Clases base, reutilizables de ADO .NET. Proporciona la colección de clases necesarias para acceder a una fuente de datos (como por ejemplo una Base de Datos).

- **System.Data.Internal**. Integra el conjunto de clases internas de las que se componen los proveedores de datos.

Dentro del espacio de nombres System.Data encontramos las siguientes clases compartidas, que constituyen el eje central de ADO .NET.

- **DataSet**. Almacén de datos por excelencia en ADO .NET. Representa una base de datos desconectada del proveedor de datos. Almacena tablas y sus relaciones.
- **DataTable**. Un contenedor de datos. Estructurado como un conjunto de filas (DataRow) y columnas (DataColumn).
- **DataRow**. Registro que almacena n valores. Representación en ADO .NET de una fila/tupla de una tabla de la base de datos.
- **DataColumn**. Contiene la definición de una columna. Metadatos y datos asociados a su dominio.
- **DataRelation**. Enlace entre dos o más columnas iguales de dos o mas tablas.
- **Constraint**. Reglas de validación de las columnas de una tabla.
- **DataColumnMapping**. Vínculo lógico existente entre una columna de un objeto del DataSet y la columna física de la tabla de la base de datos.
- **DataTableMapping**. Vínculo lógico existente entre una tabla del DataSet y la tabla física de la base de datos.

Además de estas clases, existe otro grupo de clases consistente en las clases específicas de un proveedor de datos. Estas clases conforman los aspectos particulares de un fabricante de proveedores de datos .NET. Tienen una sintaxis con el formato XXXClase, donde "XXX" es un prefijo que determina el tipo de plataforma de conexión a datos. Se definen en dos espacios de nombre: System.Data.SqlClient y System.Data.OleDb.

En la Tabla 32 se ofrece una descripción de las clases que podemos encontrar en estos espacios de nombre.

Clase	Descripción
SqlCommand OleDbCommand	Clases que representan un comando SQL contra un sistema gestor de datos.
SqlConnection OleDbConnection	Clase que representa la etapa de conexión a un proveedor de datos. Encapsula la seguridad, pooling de conexiones, etc.
SqlCommandBuilder OleDbCommandBuilder	Generador de comandos SQL de inserción, modificación y borrado desde una consulta SQL de selección de datos.
SqlDataReader OleDbDataReader	Un lector de datos de sólo avance, conectado a la base de datos.

SqlDataAdapter OleDbDataAdapter	Clase adaptadora entre un objeto DataSet y sus operaciones físicas en la base de datos (select, insert, update y delete).
SqlParameter OleDbParameter	Define los parámetros empleados en la llamada a procedimientos almacenados.
SqlTransaction OleDbTransaction	Gestión de las transacciones a realizar en la base de datos.

Tabla 32

Para aquellos conocedores de ADO en alguna de sus versiones anteriores, podemos hacer una analogía o comparación entre las antiguas clases de ADO y las nuevas de ADO .NET. En la Figura 341 se puede ver esta aproximación.

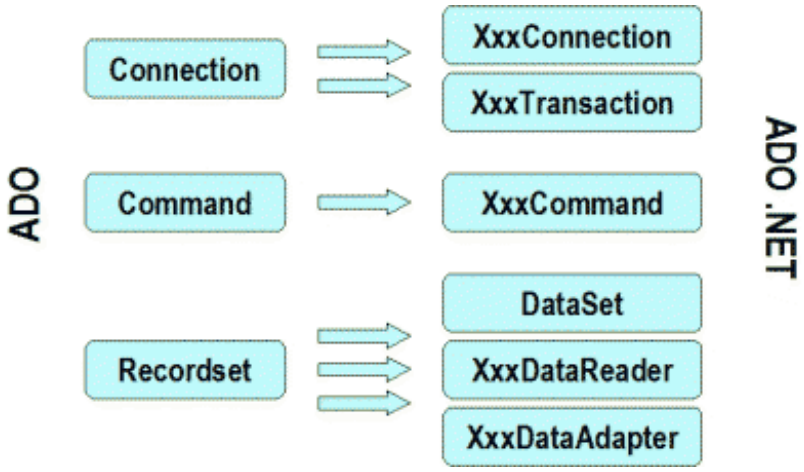


Figura 341. Comparativa entre las clases de ADO y ADO .NET.

Hasta aquí hemos realizado una introducción a la tecnología ADO .NET, repasando su arquitectura y comentando las clases principales. En lo que resta de tema vamos a utilizar las distintas clases que nos ofrece ADO .NET desde VB.NET, para realizar tareas comunes de acceso a datos, como pueden ser establecer una conexión, obtener un conjunto de registros, realizar operaciones con los datos, etc.

Las clases Connection

En los ejemplos con datos que vamos a realizar, se ha utilizado SQL Server 2000 como servidor de datos, y fundamentalmente, la base de datos Northwind.

El primer paso obligado en un acceso a datos consiste en establecer una conexión con un almacén de datos. Esto lo vamos a conseguir gracias a las clases Connection de ADO .NET, que nos permitirán conectarnos a un origen de datos (ya sea una base de datos o no) , al igual que en ADO clásico empleábamos el objeto Connection.

En ADO se podía ejecutar directamente una sentencia contra el almacén de datos, o bien abrir un conjunto de registros (Recordset), pero en ADO .NET no vamos a realizar esta operación con este tipo de objetos.

Debemos recordar que existen dos implementaciones para algunos de los objetos de ADO .NET, cada uno específico del origen de datos con el que nos vamos a conectar. Esto ocurre con el objeto Connection, que tiene dos versiones, una como proveedor de datos de SQL Server, a través de la clase System.Data.SqlClient.SqlConnection, y otra como proveedor de datos OLEDB, a través de la clase Sysem.Data.OleDb.OleDbConnection.

Por norma general, del objeto Connection utilizaremos los métodos Open( ) y Close( ), para abrir y cerrar conexiones respectivamente, con el almacén de datos adecuado. Aunque tenemos el recolector de basura que gestiona de forma automática los recursos y objetos que no son utilizados, es recomendable cerrar las conexiones de forma explícita utilizando el método Close( ).

Las conexiones se abrirán de forma explícita utilizando el método Open(), pero también se puede hacer de forma implícita utilizando un objeto DataAdapter, esta posibilidad la veremos más adelante. Cuando ejecutamos el método Open() sobre un objeto Connection (SqlConnection o OleDbConnection), se abrirá la conexión que se ha indicado en su propiedad ConnectionString, es decir, esta propiedad indicará la cadena de conexión que se va a utilizar para establecer la conexión con el almacén de datos correspondiente. El método Open() no posee parámetros.

El constructor de la clase Connection (al decir clase Connection de forma genérica nos estamos refiriendo en conjunto a las clases SqlConnection y OleDbConnection de ADO .NET) se encuentra sobrecargado, y en una de sus versiones recibe como parámetro una cadena que será la cadena de conexión que se aplique a su propiedad ConnectionString.

Si hacemos uso de la clase SqlConnection, en la cadena de conexión no podremos especificar una DSN de ODBC, ya que la conexión se va a realizar en este caso directamente con SQL Server. Y si utilizamos la clase OleDbConnection debemos especificar el proveedor OLEDB que se va a utilizar para establecer la conexión, una excepción es el proveedor OLEDB para ODBC (MSDASQL), que no puede ser utilizado, ya que el proveedor OLEDB de .NET no soporta el proveedor de ODBC, en este caso deberemos realizar la conexión utilizando el proveedor adecuado al almacén de datos. Los proveedores OLEDB que son compatibles con ADO .NET son:

- SQLOLEDB: Microsoft OLE DB Provider for SQL Server.
- MSDAORA: Microsoft OLE DB Provider for Oracle.
- Microsoft.Jet.OLEDB.4.0: OLE DB Provider for Microsoft Jet.

La sintaxis utilizada para indicar la cadena de conexión, con las particularidades propias de cada proveedor, veremos que es muy similar a la utilizada en ADO clásico. El Código fuente 559 muestra un ejemplo de conexión con un servidor SQL Server 2000, y su posterior desconexión, utilizando un objeto SqlConnection. Debemos importar el espacio de nombres Data.SqlClient para poder utilizar el objeto. Este código lo podemos asociar a la pulsación de un botón en un formulario.

```
Imports System.Data.SqlClient
'....
Try
    ' crear el objeto de conexión
    Dim oConexion As New SqlConnection()
    ' pasar la cadena de conexión
    oConexion.ConnectionString = "server=(local);" & _
```

```
        "database=Xnorthwind;uid=sa;pwd=;"

    ' abrir conexión
    oConexion.Open()
    MessageBox.Show("Conectado")

    ' cerrar conexión
    oConexion.Close()
    MessageBox.Show("Desconectado")

Catch oExcep As SqlException
    ' si se produce algún error,
    ' lo capturamos mediante el objeto
    ' de excepciones particular para
    ' el proveedor de SQL Server
    MessageBox.Show("Error al conectar con datos" & _
    ControlChars.CrLf & _
    oExcep.Message & ControlChars.CrLf & _
    oExcep.Server)
End Try
```

Código fuente 559

El Código fuente 560 muestra la misma operación pero usando el objeto de conexión para el proveedor de OLEDB. Observe el lector las diferencias en las cadenas de conexión y el objeto de excepción con respecto al anterior ejemplo, así como el espacio de nombres a importar.

```
Imports System.Data.OleDb
'....
Try
    ' crear el objeto de conexión
    Dim oConexion As New OleDbConnection()

    oConexion.ConnectionString = "Provider=SQLOLEDB;" & _
        "Server=(local);Database=Northwind;uid=sa;pwd=;"

    ' abrir conexión
    oConexion.Open()
    MessageBox.Show("Conectado")

    ' cerrar conexión
    oConexion.Close()
    MessageBox.Show("Desconectado")

Catch oExcep As OleDbException
    ' si se produce algún error,
    ' lo capturamos mediante el objeto
    ' de excepciones particular para
    ' el proveedor de OLEDB
    MessageBox.Show("Error al conectar con datos" & _
    ControlChars.CrLf & _
    oExcep.Message & ControlChars.CrLf & _
    oExcep.Source())
End Try
```

Código fuente 560

Las clases Command

Establecida una conexión con un almacén de datos, la siguiente operación lógica consiste en enviarle sentencias para realizar los distintos tipos de operaciones que habitualmente realizamos con los datos. Las clases Command de ADO .NET serán las usaremos para realizar tales operaciones.

SqlCommand y OleDbCommand, son muy similares al objeto Command existente en ADO. El objeto Command nos va a permitir ejecutar una sentencia SQL o un procedimiento almacenado sobre la fuente de datos a la que estamos accediendo.

A través de un objeto Command también podremos obtener un conjunto de resultados del almacén de datos. En este caso, los resultados se pasarán a otros objetos de ADO .NET, como DataReader o DataAdapter; estos dos objetos los comentaremos más adelante.

Un objeto Command lo vamos a crear a partir de una conexión ya existente, y va a contener una sentencia SQL para ejecutar sobre la conexión establecida con el origen de datos.

Entre las propiedades que ofrecen los objetos SqlCommand y OleDbCommand, caben destacar las siguientes.

- **CommandText.** Contiene una cadena de texto que va a indicar la sentencia SQL o procedimiento almacenado que se va a ejecutar sobre el origen de los datos.

- **CommandTimeout.** Tiempo de espera en segundos que se va a aplicar a la ejecución de un objeto Command. Su valor por defecto es de 30 segundos.

- **CommandType**. Indica el tipo de comando que se va a ejecutar contra el almacén de datos, es decir, indica como se debe interpretar el valor de la propiedad CommadText. Puede tener los siguientes valores: StoredProcedure, para indicar que se trata de un procedimiento almacenado; TableDirect se trata de obtener una tabla por su nombre (únicamente aplicable al objeto OleDbCommand); y Text que indica que es una sentencia SQL. EL valor por defecto es Text.

- **Connection**. Devuelve el objeto SqlConnection o OleDbConnection utilizado para ejecutar el objeto Command correspondiente.

- **Parameters**. Colección de parámetros que se pueden utilizar para ejecutar el objeto Command, esta colección se utiliza cuando deseamos ejecutar sentencias SQL que hacen uso de parámetros, esta propiedad devuelve un objeto de la clase SqlParameterCollection o un objeto de la clase OleDbParameterCollection. Estas colecciones contendrán objetos de la clase SqlParamter y OleDbParameter, respectivamente, para representar a cada uno de los parámetros utilizados. Estos parámetros también son utilizados para ejecutar procedimientos almacenados.

Una vez vistas algunas de las propiedades de las clases SqlCommand y OleDbCommand, vamos a pasar a comentar brevemente los principales métodos de estas clases.

- **CreateParameter**. Crea un parámetro para el que después podremos definir una serie de características específicas como pueden ser el tipo de dato, su valor, tamaño, etc. Devolverá un objeto de la clase SqlParameter u OleDbParameter.

- **ExecuteNonQuery**. Ejecuta la sentencia SQL definida en la propiedad ComandText contra la conexión definida en la propiedad Connection. La sentencia a ejecutar debe ser de un tipo que no devuelva un conjunto de registros, por ejemplo Update, Delete o Insert. Este método

devuelve un entero indicando el número de filas que se han visto afectadas por la ejecución del objeto Command.

- **ExecuteReader**. Ejecuta la sentencia SQL definida en la propiedad ComandText contra la conexión definida en la propiedad Connection. En este caso, la sentencia sí devolverá un conjunto de registros. El resultado de la ejecución de este será un objeto de la clase SqlDataReader/OleDbDataReader, que nos va a permitir leer y recorrer los resultados devueltos por la ejecución del objeto Command correspondiente.

- **ExecuteScalar**. Este método se utiliza cuando deseamos obtener la primera columna de la primera fila del conjunto de registros, el resto de datos no se tendrán en cuenta. La utilización de este método tiene sentido cuando estamos ejecutando una sentencia SQL del tipo Select Count(*). Este método devuelve un objeto de la clase genérica Object.

- **Prepare**. Este método construye una versión compilada del objeto Command dentro del almacén de datos.

A continuación mostraremos algunos ejemplos de uso de objetos Command.

El Código fuente 561 ilustra la inserción de un registro utilizando un objeto SqlCommand. En primer lugar utilizamos un constructor de la clase, que recibe como parámetro la sentencia a ejecutar y la conexión. Como vamos a ejecutar una sentencia que no produce un conjunto de resultados, emplearemos el método ExecuteNonQuery( ). Observe también el lector en este ejemplo, que la conexión sólo permanece abierta en el momento de ejecutar el comando; esta es la técnica recomendable que debemos utilizar para todas las operaciones con datos: mantener abierta la conexión el menor tiempo posible.

```
' crear conexión
Dim oConexion As New SqlConnection()
oConexion.ConnectionString = "Server=(local);" & _
    "Database=Gestion;uid=sa;pwd=;"

' crear sentencia SQL
Dim sSQL As String
sSQL = "INSERT INTO Clientes (IDCliente,Nombre,FIngreso) " & _
    "VALUES(10,'Alfredo','18/7/2002')"

' crear comando
Dim oComando As New SqlCommand(sSQL, oConexion)

Dim iResultado As Integer
oConexion.Open() ' abrir conexión
iResultado = oComando.ExecuteNonQuery() ' ejecutar comando
oConexion.Close() ' cerrar conexión

MessageBox.Show("Registros añadidos:" & iResultado)
```

Código fuente 561

En el Código fuente 562 realizamos también la inserción con un SqlCommand, pero utilizando en este caso parámetros. En la cadena que tiene la sentencia SQL indicaremos los parámetros con el formato '@NombreParámetro'.

Para crear cada uno de los parámetros utilizaremos la clase SqlParameter, mientras que para añadir los parámetros usaremos la colección Parámeters del objeto SqlCommand y su método Add( ).

Respecto a la creación de los parámetros, podemos observar que es muy flexible, ya que como vemos en este ejemplo, cada uno de ellos se crea de un modo distinto, especificando el nombre, tipo de dato y valor.

```
' crear conexión
Dim oConexion As New SqlConnection()
oConexion.ConnectionString = "Server=(local);" & _
    "Database=Gestion;uid=sa;pwd=;"

' crear sentencia SQL para insertar un registro con
' parámetros; indicamos el nombre del parámetro con
' @NombreParámetro
Dim sSQL As String
sSQL = "INSERT INTO Clientes (IDCliente,Nombre,FIngreso) " & _
    "VALUES(@CodCli,@Nombre,@Fecha)"

' crear comando
Dim oComando As New SqlCommand(sSQL, oConexion)

' añadir parámetros al comando:
' parámetro primer campo
oComando.Parameters.Add(New SqlParameter("@CodCli", _
    SqlDbType.Int))
oComando.Parameters("@CodCli").Value = 25

' parámetro segundo campo
oComando.Parameters.Add(New SqlParameter("@Nombre", "Raquel"))

' parámetro tercer campo
Dim oParametro As New SqlParameter()
oParametro.ParameterName = "@Fecha"
oParametro.SqlDbType = SqlDbType.DateTime
oParametro.Value = "25/10/2002"
oComando.Parameters.Add(oParametro)

Dim iResultado As Integer

oConexion.Open() ' abrir conexión
iResultado = oComando.ExecuteNonQuery() ' ejecutar comando
oConexion.Close() ' cerrar conexión

MessageBox.Show("Registros añadidos:" & iResultado)
```

Código fuente 562

Si empleamos un objeto OleDbCommand, la sintaxis de la sentencia SQL cambia, ya que los parámetros deberemos indicarlos como hacíamos en ADO clásico, utilizando el carácter '?'. Veamos un ejemplo en el Código fuente 563.

```
' crear el objeto de conexión
Dim oConexion As New OleDbConnection()
oConexion.ConnectionString = "Provider=SQLOLEDB;" & _
    "Server=(local);Database=Gestion;uid=sa;pwd=;"

' crear sentencia SQL para modificar un registro con
' parámetros; indicamos el parámetro con ?
Dim sSQL As String
sSQL = "UPDATE Clientes SET Nombre = ? " & _
    "WHERE IDCliente = 2"
```

```
' crear comando
Dim oComando As New OleDbCommand(sSQL, oConexion)
oComando.Parameters.Add(New OleDbParameter("NombreCli", _
    OleDbType.VarChar, 50))
oComando.Parameters("NombreCli").Value = "David"

Dim iResultado As Integer
oConexion.Open() ' abrir conexión
iResultado = oComando.ExecuteNonQuery() ' ejecutar comando
oConexion.Close() ' cerrar conexión

MessageBox.Show("Registros modificados:" & iResultado)
```

Código fuente 563

En el caso de que necesitemos ejecutar un procedimiento almacenado, debemos indicarlo mediante las propiedades CommandType y CommandText del objeto Command que estemos utilizando. En la primera establecemos el tipo de comando (procedimiento almacenado) seleccionando el valor de la enumeración asociada a la propiedad; y en la segunda asignamos una cadena con el nombre del procedimiento almacenado. El Código fuente 564 muestra un ejemplo, en el que podemos comprobar que hemos utilizado un constructor de SqlCommand sin parámetros, por lo que el objeto Connection lo asignamos después mediante la propiedad Connection

```
' crear conexión
Dim oConexion As New SqlConnection()
oConexion.ConnectionString = "Server=(local);" & _
    "Database=Gestion;uid=sa;pwd=;"

' crear comando para ejecutar procedimiento almacenado
' que borra un registro
Dim oComando As New SqlCommand()
oComando.Connection = oConexion
oComando.CommandType = CommandType.StoredProcedure
oComando.CommandText = "BorraCli"

' añadir parámetro al comando
oComando.Parameters.Add(New SqlParameter("@IDCliente", _
    SqlDbType.Int))
oComando.Parameters("@IDCliente").Value = 25

Dim iResultado As Integer

oConexion.Open() ' abrir conexión
iResultado = oComando.ExecuteNonQuery() ' ejecutar comando
oConexion.Close() ' cerrar conexión

MessageBox.Show("Registros borrados:" & iResultado)
```

Código fuente 564

Para obtener el resultado de una función del lenguaje SQL, por ejemplo Count( ), emplearemos el método ExecuteScalar( ) del objeto Command. En el Código fuente 565, la ejecución de este método nos devuelve el número de filas de una tabla de la base de datos, que mostramos en un mensaje.

```
' crear conexión
Dim oConexion As New SqlConnection()
```

```
oConexion.ConnectionString = "Server=(local);" & _
    "Database=Gestion;uid=sa;pwd=;"

' crear comando escalar
Dim sSQL As String
sSQL = "SELECT COUNT(*) FROM Clientes"

' crear comando
Dim oComando As New SqlCommand(sSQL, oConexion)

Dim iResultado As Integer
oConexion.Open() ' abrir conexión
iResultado = oComando.ExecuteScalar() ' ejecutar comando
oConexion.Close() ' cerrar conexión

MessageBox.Show("Número de registros de clientes:" & iResultado)
```

Código fuente 565

Las clases DataReader

Un objeto DataReader permite la navegación hacia delante y de sólo lectura, de los registros devueltos por una consulta. Es lo más parecido al objeto Recordset de ADO de tipo read only/forward only.

A diferencia del resto de objetos, que siguen un esquema desconectado de manipulación de datos, los DataReader permanecen conectados durante todo el tiempo que realizan el recorrido por los registros que contienen. Las clases que implementan este tipo de objeto son SqlDataReader y OleDbDataReader.

Para obtener un DataReader, ejecutaremos el método ExecuteReader( ) de un objeto Command basado en una consulta SQL o procedimiento almacenado.

A continuación vamos a pasar a describir las principales propiedades de las clases SqlDataReader y OleDbDataReader.

- **FieldCount**. Devuelve el número de columnas (campos) presentes en el fila (registro) actual.
- **IsClosed**. Devolverá los valores True o False, para indicar si el objeto DataReader está cerrado o no.
- **Item**. Devuelve en formato nativo, el valor de la columna cuyo nombre le indicamos como índice en forma de cadena de texto.

Una vez vistas las propiedades, vamos a comentar los métodos más destacables.

- **Close( )**. Cierra el objeto DataReader liberando los recursos correspondientes.
- **GetXXX( )**. El objeto DataReader presenta un conjunto de métodos que nos van a permitir obtener los valores de las columnas contenidas en el mismo en forma de un tipo de datos determinado, según el método GetXXX empleado. Existen diversos métodos GetXXX atendiendo al tipo de datos de la columna, algunos ejemplos son GetBoolean(), GetInt32(), GetString(), GetChar(), etc. Como parámetro a este método le debemos indicar el número de orden de la columna que deseamos recuperar.
- **NextResult( )**. Desplaza el puntero actual al siguiente conjunto de registros, cuando la sentencia es un procedimiento almacenado de SQL o una sentencia SQL que devuelve más de

un conjunto de registros, no debemos confundir este método con el método MoveNext() de ADO, ya que en este caso no nos movemos al siguiente registro, sino al siguiente conjunto de registros.

- **Read( )**. Desplaza el cursor actual al siguiente registro permitiendo obtener los valores del mismo a través del objeto DataReader. Este método devolverá True si existen más registros dentro del objeto DataReader, False si hemos llegado al final del conjunto de registros. La posición por defecto del objeto DataReader en el momento inicial es antes del primer registro, por lo tanto para recorrer un objeto DataReader debemos comenzar con una llamada al método Read(), y así situarnos en el primer registro.

El proyecto PruDataReader (hacer clic aquí para acceder al ejemplo), contiene un formulario con algunos controles, que muestran el uso de objetos DataReader.

El botón Empleados crea a partir de un comando, un objeto DataReader que recorremos para llenar un ListBox con los valores de una de las columnas de la tabla que internamente contiene el DataReader. Veamos este caso en el Código fuente 566.

```
Private Sub btnEmpleados_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnEmpleados.Click

    ' crear conexion
    Dim oConexion As New SqlConnection()
    oConexion.ConnectionString = "Server=(local);" & _
        "Database=Northwind;uid=sa;pwd=;"

    ' crear comando
    Dim oComando As New SqlCommand("SELECT * FROM Employees", _
        oConexion)

    ' crear DataReader
    Dim oDataReader As SqlDataReader
    oConexion.Open()
    oDataReader = oComando.ExecuteReader() ' obtener DataReader

    ' recorrer filas
    While oDataReader.Read()
        Me.lstEmpleados.Items.Add(oDataReader("LastName"))
    End While

    oDataReader.Close()
    oConexion.Close()

End Sub
```

Código fuente 566

Como también hemos indicado anteriormente, un objeto Command puede estar basado en múltiples sentencias SQL, separadas por el carácter de punto y coma (;), que se ejecuten en lote. Al crear un DataReader desde un comando de este tipo, podemos recorrer el conjunto de consultas mediante el método NextResult( ) del DataReader. Un ejemplo de este tipo lo tenemos al pulsar el botón Clientes/Productos del formulario, cuyo fuente vemos a continuación en el Código fuente 567. Observe en este caso que conectamos a través de OLE DB, por lo que empleamos los objetos ADO .NET de esta categoría.

```
Private Sub btnCliProd_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnCliProd.Click

    ' crear conexion
    Dim oConexion As New OleDbConnection()
    oConexion.ConnectionString = "Provider=SQLOLEDB;" & _
        "Server=(local);Database=Northwind;uid=sa;pwd=;"

    ' crear comando compuesto por varias consultas
    Dim oComando As New OleDbCommand("SELECT * FROM Customers; SELECT * FROM
Products", oConexion)

    Dim oDataReader As OleDbDataReader
    oConexion.Open()
    oDataReader = oComando.ExecuteReader() ' obtener DataReader

    ' recorrer filas de la primera consulta
    While oDataReader.Read()
        Me.lstClientes.Items.Add(oDataReader("CompanyName"))
    End While

    ' pasar a la siguiente consulta y recorrer
    ' las filas
    oDataReader.NextResult()
    While oDataReader.Read()
        Me.lstProductos.Items.Add(oDataReader("ProductName"))
    End While

    oDataReader.Close()
    oConexion.Close()

End Sub
```

Código fuente 567

La Figura 342 muestra este formulario después de haber rellenado los controles ListBox usando objetos DataReader.

Figura 342. ListBox llenados con objetos DataReader.

37

Conjuntos de datos y enlace (Data Binding)

La clase DataSet

DataSet pertenece al conjunto común de clases de ADO .NET, empleándose para todo tipo de proveedores, por lo que no existe una versión particular para SqlClient u OleDb,. En la introducción que sobre ADO .NET realizamos en el anterior tema, hemos comentado algunos aspectos sobre esta clase.

Básicamente, un objeto DataSet va a ser similar a un objeto Recordset de ADO, pero más potente y complejo. Es el almacén de datos por excelencia en ADO .NET, representando una base de datos en memoria y desconectada del proveedor de datos, que contiene tablas y sus relaciones.

El objeto DataSet nos proporciona el mejor concepto sobre datos desconectados: una copia en el cliente de la arquitectura de la base de datos, basada en un esquema XML que la independiza del fabricante, proporcionando al desarrollador la libertad de trabajo independiente de la plataforma.

Cada tabla contenida dentro de un objeto DataSet se encuentra disponible a través de su propiedad Tables, que es una colección de objetos System.Data.DataTable. Cada objeto DataTable contiene una colección de objetos DataRow que representan las filas de la tabla. Y si seguimos con esta analogía tenemos que decir que cada objeto DataRow, es decir, cada fila, posee una colección de objetos DataColumn, que representan cada una de las columnas de la fila actual. Existen además, colecciones y objetos para representan las relaciones, claves y valores por defecto existentes dentro de un objeto DataSet.

Cada objeto DataTable dispone de una propiedad llamada DefaultView, que devuelve un objeto de la clase DataView, el cual nos ofrece una vista de los datos de la tabla para que podamos recorrer los datos, filtrarlos, ordenarlos, etc.

Para poder crear e inicializar las tablas del DataSet debemos hacer uso del objeto DataAdapter, que posee las dos versiones, es decir, el objeto SqlDataAdapter para SQL Server y OleDbDataAdapter genérico de OLE DB.

Al objeto DataAdapter le pasaremos como parámetro una cadena que representa la consulta que se va a ejecutar, y que va a rellenar de datos el DataSet. Del objeto DataAdapter utilizaremos el método Fill(), que posee dos parámetros; el primero es el DataSet a rellenar de información; y el segundo, una cadena con el nombre que tendrá la tabla creada dentro del DataSet, producto de la ejecución de la consulta.

En el siguiente apartado veremos los objetos DataAdapter, que van a funcionar como intermediarios entre el almacén de datos, y el objeto DataSet, que contiene la versión desconectada de los datos.

Entre los métodos más destacables de la clase DataSet podemos mencionar los siguientes.

- **Clear( )**. Elimina todos los datos almacenados en el objeto DataSet, vaciando todas las tablas contenidas en el mismo.

- **AcceptChanges( )**. Confirma todos los cambios realizados en las tablas y relaciones contenidas en el objeto DataSet, o bien los últimos cambios que se han producido desde la última llamada al método AcceptChanges.

- **GetChanges( )**. Devuelve un objeto DataSet que contiene todos los cambios realizados desde que se cargó con datos, o bien desde que se realizó la última llamada al método AcceptChanges.

- **HasChanges( )**. Devuelve true o false para indicar si se han realizado cambios al contenido del DataSet desde que fue cargado o bien desde que se realizó la última llamada al método AcceptChanges.

- **RejectChanges( )**. Abandona todos los cambios realizados en las tablas contenidas en el objeto DataSet desde que fue cargado el objeto o bien desde la última vez que se lanzó el método AcceptChanges.

- **Merge( )**. Toma los contenidos de un DataSet y los mezcla con los de otro DataSet, de forma que contendrá los datos de ambos objetos DataSet.

En lo que respecta a las propiedades de la clase DataSet, podemos remarcar las siguientes.

- **CaseSensitive**. Propiedad que indica si las comparaciones de texto dentro de las tablas distinguen entre mayúsculas y minúsculas. Por defecto tiene el valor False.

- **DataSetName**. Establece o devuelve mediante una cadena de texto el nombre del objeto DataSet.

- **HasErrors**. Devuelve un valor lógico para indicar si existen errores dentro de las tablas del DataSet.

- **Relations**. Esta propiedad devuelve una colección de objetos DataRelation, que representan todas las relaciones existentes entre las tablas del objeto DataSet.

- **Tables**. Devuelve una colección de objetos DataTable, que representan a cada una de las tablas existentes dentro del objeto DataSet.

En el ejemplo del Código fuente 568 ofrecemos un sencillo ejemplo de creación de un objeto DataSet que llenaremos con un DataAdapter. Una vez listo el DataSet, recorreremos la tabla que contiene y mostraremos valores de sus columnas en un ListBox.

```
' crear conexión
Dim oConexion As New SqlConnection()
oConexion.ConnectionString = "Server=(local);Database=Northwind;uid=sa;pwd=;"

' crear adaptador
Dim oDataAdapter As New SqlDataAdapter("SELECT * FROM Customers ORDER BY
ContactName", oConexion)

' crear conjunto de datos
Dim oDataSet As New DataSet()

oConexion.Open()
' utilizar el adaptador para llenar el dataset con una tabla
oDataAdapter.Fill(oDataSet, "Customers")
oConexion.Close()

' una vez desconectados, recorrer la tabla del dataset
Dim oTabla As DataTable
oTabla = oDataSet.Tables("Customers")

Dim oFila As DataRow
For Each oFila In oTabla.Rows
    ' mostrar los datos mediante un objeto fila
    Me.lstCustomers.Items.Add(oFila.Item("CompanyName") & _
        " - " & oFila.Item("ContactName") & " - " & _
        oFila.Item("Country"))
Next
```

Código fuente 568

La Figura 343 muestra el ListBox llenado a partir del DataSet.

Figura 343. Relleno de un ListBox mediante un DataSet.

Las clases DataAdapter

Como hemos comentado en anteriores apartados, los objetos DataAdapter (SqlDataAdapter y OleDbDataAdapter) van a desempeñar el papel de puente entre el origen de datos y el DataSet, permitiéndonos cargar el DataSet con la información de la fuente de datos, y posteriormente, actualizar el origen de datos con la información del DataSet.

Un objeto DataAdapter puede contener desde una sencilla sentencia SQL, como hemos visto en el apartado anterior, hasta varios objetos Command.

La clase DataAdapter dispone de cuatro propiedades, que nos van a permitir asignar a cada una, un objeto Command (SqlCommand u OleDbCommand) con las operaciones estándar de manipulación de datos. Estas propiedades son las siguientes.

- **InsertCommand**. Objeto de la clase Command, que se va a utilizar para realizar una inserción de datos.
- **SelectCommand**. Objeto de la clase Command que se va a utilizar para ejecutar una sentencia Select de SQL.
- **UpdateCommand**. Objeto de la clase Command que se va a utilizar para realizar una modificación de los datos.
- **DeleteCommand**. Objeto de la clase Command que se va a utilizar para realizar una eliminación de datos.

Un método destacable de las clases SqlDataAdapter/OleDbDataAdapter es el método Fill( ), que ejecuta el comando de selección que se encuentra asociado a la propiedad SelectCommand, los datos obtenidos del origen de datos se cargarán en el objeto DataSet que pasamos por parámetro.

La Figura 344 muestra la relación entre los objetos DataAdapter y el objeto DataSet.

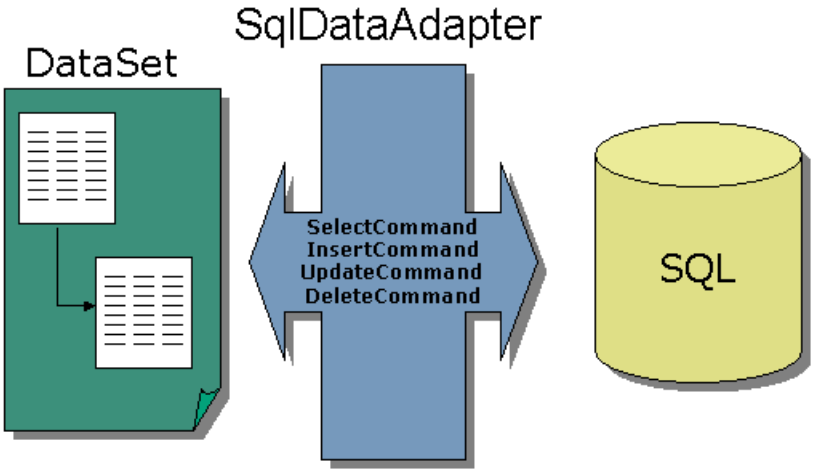


Figura 344. Relación entre objetos DataAdapter y DataSet.

Para demostrar el uso de los objetos DataAdapter vamos a desarrollar un proyecto con el nombre PruDataAdapter (hacer clic aquí para acceder a este ejemplo). En esta aplicación vamos a utilizar el mismo objeto DataAdapter para realizar una consulta contra una tabla e insertar nuevas filas en esa misma tabla.

En primer lugar diseñaremos el formulario del programa. Como novedad, introduciremos el control DataGrid, que trataremos con más profundidad en un próximo apartado. Baste decir por el momento, que a través del DataGrid visualizaremos una o varias tablas contenidas en un DataSet. La Figura 345 muestra el aspecto de esta aplicación en funcionamiento.

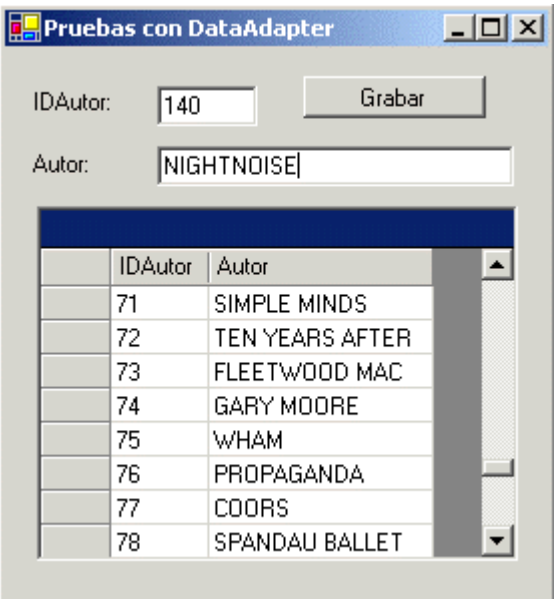


Figura 345. Formulario para operaciones con DataAdapter y DataGrid.

Respecto al código del formulario, en primer lugar, vamos a declarar varios objetos de acceso a datos a nivel de la clase para poder tenerlos disponibles en diversos métodos. Veamos el Código fuente 569.

```
Imports System.Data.SqlClient

Public Class Form1
    Inherits System.Windows.Forms.Form

    Private oConexion As SqlConnection
    Private oDataSet As DataSet
    Private oDataAdapter As SqlDataAdapter
'....
'....
```

Código fuente 569

En el siguiente paso escribiremos el procedimiento del evento Load del formulario, y el método CargarDatos( ), que se ocupa de cargar el DataSet, y asignárselo al DataGrid a través de su propiedad DataSource. Observe el lector que en el método CargarDatos( ) lo primero que hacemos es vaciar el DataSet, puesto que este objeto conserva los datos de tablas y registros; en el caso de que no limpiáramos el DataSet, se acumularían las sucesivas operaciones de llenado de filas sobre la tabla que contiene. Veamos el Código fuente 570.

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

    ' crear conexión
    oConexion = New SqlConnection()
    oConexion.ConnectionString = "Server=(local);Database=MUSICA;uid=sa;pwd=;"

    ' crear adaptador
    oDataAdapter = New SqlDataAdapter()

    ' crear comandos para inserción, consulta con sus parámetros
    ' y asignarlos al adaptador
    Dim oCmdInsercion As New SqlCommand("INSERT INTO AUTORES " & _
        "(IDAutor,Autor) VALUES(@IDAutor,@Autor)", oConexion)
    oDataAdapter.InsertCommand = oCmdInsercion
    oDataAdapter.InsertCommand.Parameters.Add(New SqlParameter("@IDAutor",
SqlDbType.Int))
    oDataAdapter.InsertCommand.Parameters.Add(New SqlParameter("@Autor",
SqlDbType.NVarChar))

    Dim oCmdConsulta As New SqlCommand("SELECT * FROM AUTORES", _
        oConexion)
    oDataAdapter.SelectCommand = oCmdConsulta

    ' crear conjunto de datos
    oDataSet = New DataSet()

    Me.CargarDatos()
End Sub

Private Sub CargarDatos()
    ' vaciar el dataset
    oDataSet.Clear()

    oConexion.Open() ' abrir conexión
    ' utilizar el adaptador para llenar el dataset con una tabla
    oDataAdapter.Fill(oDataSet, "Autores")
    oConexion.Close() ' cerrar conexión

    ' enlazar dataset con datagrid;
    ' en DataSource se asigna el dataset,
    ' en DataMember el nombre de la tabla del
    ' dataset que mostrará el datagrid
    Me.grdDatos.DataSource = oDataSet
    Me.grdDatos.DataMember = "Autores"
End Sub
```

Código fuente 570

Finalmente, en el botón Grabar, escribiremos las instrucciones para insertar un nuevo registro en la tabla. Veamos el Código fuente 571.

```
Private Sub btnGrabar_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnGrabar.Click

    Dim iResultado As Integer

    ' asignar valores a los parámetros para el
    ' comando de inserción
    oDataAdapter.InsertCommand.Parameters("@IDAutor").Value = Me.txtIDAutor.Text
    oDataAdapter.InsertCommand.Parameters("@Autor").Value = Me.txtAutor.Text
```

```
    ' abrir conexión
    oConexion.Open()
    ' ejecutar comando de inserción del adaptador
    iResultado = oDataAdapter.InsertCommand.ExecuteNonQuery()
    ' cerrar conexión
    oConexion.Close()

    Me.CargarDatos()

    MessageBox.Show("Registros añadidos: " & iResultado)

End Sub
```

Código fuente 571

Navegación y edición de registros en modo desconectado

Anteriormente vimos la forma de realizar operaciones de edición, en modo conectado, sobre las tablas de una base de datos, empleando los objetos Command.

Pero como también ya sabemos, la arquitectura de ADO .NET está orientada a un modelo de trabajo desconectado del almacén de datos, al que recurriremos sólo cuando necesitemos obtener los datos para su consulta y manipulación, o bien, cuando esos mismos datos desconectados, los hayamos modificado y tengamos que actualizarlos en la fuente de datos.

El objeto DataSet, combinado con un grupo de objetos enfocados al mantenimiento de datos desconectados, como son DataAdapter, DataTable, DataRow, etc., nos van a permitir realizar tareas como la navegación entre los registros de una tabla del DataSet, además de la modificación de sus datos en las operaciones habituales de inserción, modificación y borrado de filas.

El proyecto NavegaEdita que se acompaña como ejemplo (hacer clic aquí para acceder a este ejemplo), muestra los pasos necesarios que debemos dar para crear un sencillo mantenimiento de datos para una tabla albergada en un DataSet, junto a las típicas operaciones de navegación por las filas de dicha tabla. Seguidamente iremos desgranando el conjunto de pasos a realizar.

Partimos de una sencilla base de datos en SQL Server, que contiene la tabla Clientes, con los campos más característicos de esta entidad de datos: código cliente, nombre, fecha ingreso, crédito. Una vez creado un nuevo proyecto en VB.NET, diseñaremos el formulario de la aplicación que como vemos en la Figura 346, a través de sus controles, nos permitirá realizar las operaciones mencionadas.

Pasando a la escritura del código del programa, en primer lugar importaremos el espacio de nombres System.Data.SqlClient, y declararemos a nivel de clase un conjunto de variables para la manipulación de los datos. Veamos el Código fuente 572.

```
Imports System.Data.SqlClient
Public Class Form1
    Inherits System.Windows.Forms.Form
    ' variables a nivel de clase para
    ' la manipulación de datos
    Private oDataAdapter As SqlDataAdapter
    Private oDataSet As DataSet
    Private iPosicFilaActual As Integer
    '....
    '....
```

Código fuente 572

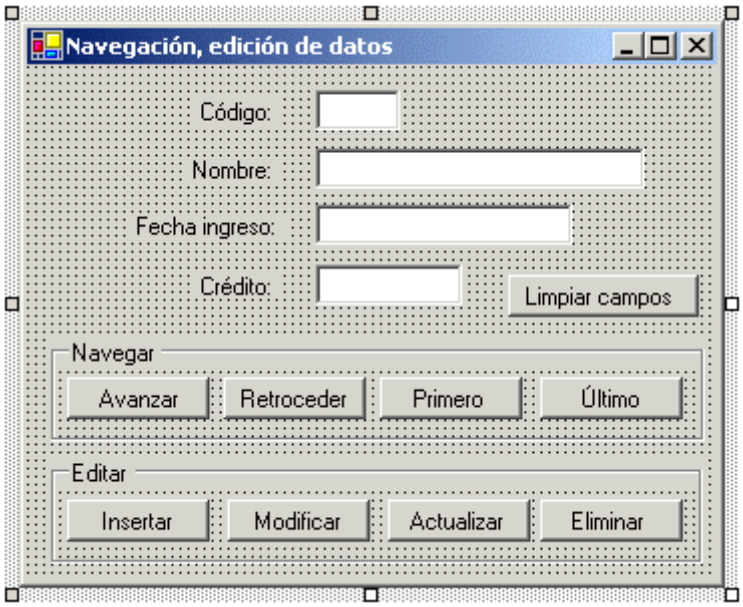


Figura 346. Formulario de navegación y edición manual de datos.

Como siguiente paso, escribiremos el manipulador del evento Load del formulario y un método para cargar los datos del registro actual en los controles del formulario, el Código fuente 573 muestra esta parte.

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

    ' crear conexión
    Dim oConexion As SqlConnection
    oConexion = New SqlConnection()
    oConexion.ConnectionString = "Server=(local);" & _
        "Database=Gestion;uid=sa;pwd=;"

    ' crear adaptador
    Me.oDataAdapter = New SqlDataAdapter("SELECT * FROM Clientes", _
        oConexion)

    ' crear commandbuilder
    Dim oCommBuild As SqlCommandBuilder = New SqlCommandBuilder(oDataAdapter)

    ' crear dataset
    Me.oDataSet = New DataSet()

    oConexion.Open()
    ' llenar con el adaptador el dataset
    Me.oDataAdapter.Fill(oDataSet, "Clientes")
    oConexion.Close()

    ' establecer el indicador del registro
    ' a mostrar de la tabla
    Me.iPosicFilaActual = 0

    ' cargar columnas del registro en
    ' los controles del formulario
    Me.CargarDatos()
End Sub

Private Sub CargarDatos()
```

```
    ' obtener un objeto con la fila actual
    Dim oDataRow As DataRow
    oDataRow = Me.oDataSet.Tables("Clientes").Rows(Me.iPosicFilaActual)

    ' cargar los controles del formulario con
    ' los valores de los campos del registro
    Me.txtIDCliente.Text = oDataRow("IDCliente")
    Me.txtNombre.Text = oDataRow("Nombre")
    Me.txtFIngreso.Text = oDataRow("FIngreso")
    Me.txtCredito.Text = oDataRow("Credito")

    ' mostrar la posición actual del registro
    ' y el número total del registros
    Me.lblRegistro.Text = "Registro: " & _
        Me.iPosicFilaActual + 1 & " de " & _
        Me.oDataSet.Tables("Clientes").Rows.Count

End Sub
```

Código fuente 573

Observe el lector que en el evento Load hemos creado un objeto CommandBuilder, pasándole como parámetro el DataAdapter. Como ya sabemos, un DataAdapter contiene una serie de objetos Command para las operaciones de consulta, inserción, etc. La misión en este caso del objeto CommandBuilder, es la de construir automáticamente tales comandos y asignárselos al DataAdapter, ahorrándonos ese trabajo de codificación.

En cuanto a las operaciones de navegación por la tabla, no hay un objeto, como ocurría con el Recordset de ADO, que disponga de métodos específicos de movimiento como MoveNext( ), MoveLast( ), etc. Lo que debemos hacer en ADO .NET, tal y como muestra el método CargarDatos(), es obtener del DataSet, la tabla que necesitemos mediante su colección Tables, y a su vez, a la colección Rows de esa tabla, pasarle el número de fila/registro al que vamos a desplazarnos. En nuestro ejemplo utilizaremos la variable iPosicFilaActual, definida a nivel de clase, para saber en todo momento, la fila de la tabla en la que nos encontramos.

El Código fuente 574 muestra el código de los botones de navegación, reunidos en el GroupBox Navegar, del formulario.

```
Private Sub btnAvanzar_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnAvanzar.Click

    ' si estamos en el último registro,
    ' no hacer movimiento
    If Me.iPosicFilaActual = _
    (Me.oDataSet.Tables("Clientes").Rows.Count - 1) Then

        MessageBox.Show("Último registro")

    Else
        ' incrementar el marcador de registro
        ' y actualizar los controles con los
        ' datos del registro actual
        Me.iPosicFilaActual += 1
        Me.CargarDatos()
    End If

End Sub
```

```
Private Sub btnRetroceder_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnRetroceder.Click

    ' si estamos en el primer registro,
    ' no hacer movimiento
    If Me.iPosicFilaActual = 0 Then

        MessageBox.Show("Primer registro")

    Else
        ' disminuir el marcador de registro
        ' y actualizar los controles con los
        ' datos del registro actual
        Me.iPosicFilaActual -= 1
        Me.CargarDatos()
    End If

End Sub

Private Sub btnPrimero_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnPrimero.Click

    ' establecer el marcador de registro en el primero
    Me.iPosicFilaActual = 0
    Me.CargarDatos()

End Sub

Private Sub btnUltimo_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnUltimo.Click

    ' establecer el marcador de registro en el primero
    ' obteniendo el número de filas que contiene la tabla menos uno
    Me.iPosicFilaActual = (Me.oDataSet.Tables("Clientes").Rows.Count - 1)
    Me.CargarDatos()

End Sub
```

Código fuente 574

Respecto a las operaciones de edición, debemos utilizar los miembros del objeto tabla del DataSet, como se muestra en el Código fuente 575. Una vez terminado el proceso de edición, actualizaremos el almacén de datos original con el contenido del DataSet, empleando el DataAdapter.

```
Private Sub btnInsertar_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnInsertar.Click

    Dim oDataRow As DataRow
    ' obtener un nuevo objeto fila de la tabla del dataset
    oDataRow = Me.oDataSet.Tables("Clientes").NewRow()

    ' asignar valor a los campos de la nueva fila
    oDataRow("IDCliente") = Me.txtIDCliente.Text
    oDataRow("Nombre") = Me.txtNombre.Text
    oDataRow("FIngreso") = Me.txtFIngreso.Text
    oDataRow("Credito") = Me.txtCredito.Text

    ' añadir el objeto fila a la colección de filas
    ' de la tabla del dataset
    Me.oDataSet.Tables("Clientes").Rows.Add(oDataRow)

End Sub
```

```
Private Sub btnModificar_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnModificar.Click

    Dim oDataRow As DataRow
    ' obtener el objeto fila de la tabla del dataset
    ' en el que estamos posicionados
    oDataRow = Me.oDataSet.Tables("Clientes").Rows(Me.iPosicFilaActual)

    ' modificar las columnas de la fila
    ' excepto la correspondiente al identificador cliente
    oDataRow("Nombre") = Me.txtNombre.Text
    oDataRow("FIngreso") = Me.txtFIngreso.Text
    oDataRow("Credito") = Me.txtCredito.Text

End Sub

Private Sub btnActualizar_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnActualizar.Click

    ' actualizar los cambios realizados en el dataset
    ' contra la base de datos real
    Me.oDataAdapter.Update(Me.oDataSet, "Clientes")

End Sub
```

Código fuente 575

El caso del borrado de filas es algo diferente, por ello lo mostramos aparte del resto de operaciones de edición. En el Código fuente 576 vemos el código del botón Eliminar, dentro del cual, obtenemos la fila a borrar mediante un objeto DataRow, procediendo a su borrado con el método Delete( ). Para actualizar los borrados realizados, empleamos el método GetChanges( ) del objeto DataTable, obteniendo a su vez, un objeto tabla sólo con las filas borradas; información esta, que pasaremos al DataAdapter, para que actualice la información en el origen de datos.

```
Private Sub btnEliminar_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnEliminar.Click

    Dim oDataRow As DataRow
    ' obtener el objeto fila, de la tabla del dataset
    ' en el que estamos posicionados
    oDataRow = Me.oDataSet.Tables("Clientes").Rows(Me.iPosicFilaActual)
    oDataRow.Delete()   ' borrar la fila

    ' mediante el método GetChanges(), obtenemos una tabla
    ' con las filas borradas
    Dim oTablaBorrados As DataTable
    oTablaBorrados =
Me.oDataSet.Tables("Clientes").GetChanges(DataRowState.Deleted)

    ' actualizar en el almacén de datos las filas borradas
    Me.oDataAdapter.Update(oTablaBorrados)

    ' confirmar los cambios realizados
    Me.oDataSet.Tables("Clientes").AcceptChanges()

    ' reposicionar en la primera fila
    Me.btnPrimero.PerformClick()
End Sub
```

Código fuente 576

Data Binding. Enlace de datos a controles

Data Binding es el mecanismo proporcionado por la plataforma .NET, que en aplicaciones con interfaz Windows Forms, enlaza objetos contenedores de datos con los controles del formulario, para poder realizar operaciones automáticas de navegación y edición.

Tipos de Data Binding

Existen dos tipos de enlace de datos: simple y complejo.

- **Enlace simple (Simple Data Binding)**. Este tipo de enlace consiste en una asociación entre un control que puede mostrar un único dato y el objeto que actúa como contenedor de datos. El ejemplo más ilustrativo es el control TextBox.

- **Enlace complejo (Complex Data Binding)**. En este enlace, el control que actúa como interfaz o visualizador de datos, dispone de la capacidad de mostrar varios o todos los datos del objeto que contiene la información. El control más común es el control DataGrid, que ya hemos visto inicialmente en un apartado anterior, y que trataremos con más detenimiento próximamente.

Elementos integrantes en un proceso de Data Binding

El mecanismo de enlace automático de datos a controles está compuesto por un elevado conjunto de elementos del conjunto de tipos de .NET Framework, entre clases, colecciones, enumeraciones, etc. A continuación vamos a mencionar los más importantes, que emplearemos en el ejemplo desarrollado seguidamente.

- **Binding**. Clase que permite crear un enlace (binding) para un control, indicando la propiedad del control que mostrará los datos, el DataSet del que se extraerá la información, y el nombre de la tabla-columna, cuyos datos pasarán a la propiedad del control.

- **DataBindings**. Colección de que disponen los controles, con la información de enlaces a datos. Gracias a su método Add( ), podemos añadir un objeto Binding, para que el control muestre los datos que indica el enlace.

- **BindingContext**. Propiedad de la clase Form, que representa el contexto de enlace a datos establecido en los controles del formulario, es decir, toda la información de enlaces establecida entre los controles y objetos proveedores de datos. Devuelve un objeto de tipo BindingManagerBase.

- **BindingManagerBase**. Objeto que se encarga de administrar un conjunto de objetos de enlace, por ejemplo, los de un formulario, obtenidos a través del BindingContext del formulario.

Empleo de Data Binding simple para navegar y editar datos

En el proyecto DataBindSimple (hacer clic aquí para acceder a este ejemplo) vamos a utilizar los elementos de enlace a datos comentados en el apartado anterior, para construir un formulario en el

que, gracias a la arquitectura de enlace automático proporcionado por la plataforma .NET, simplificaremos en gran medida el acceso a datos hacia una tabla de un DataSet.

El diseño del formulario será muy similar al realizado para el ejemplo de navegación y edición manual, descrito en un apartado anterior. Ver Figura 347.

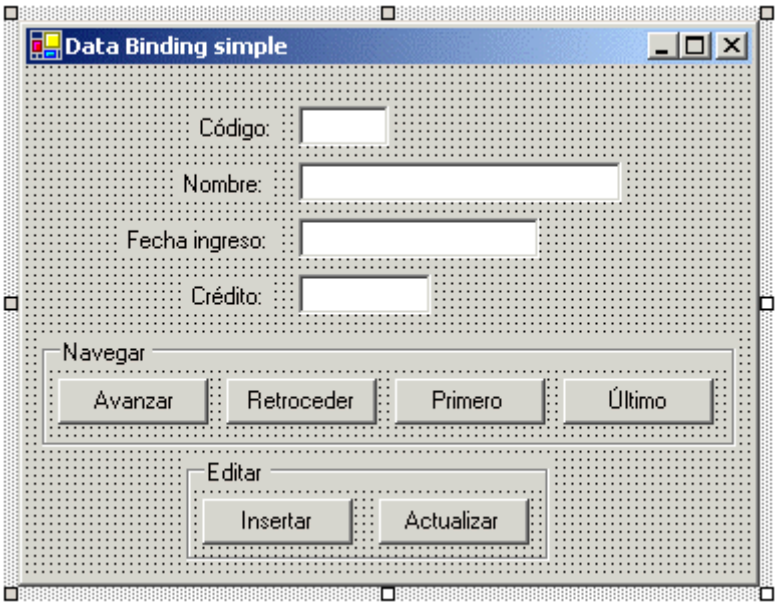


Figura 347. Formulario utilizado en Data Binding automático.

Pasando al código de la clase del formulario, deberemos realizar las siguientes declaraciones a nivel de clase, mostradas en el Código fuente 577.

```
Imports System.Data.SqlClient

Public Class Form1
    Inherits System.Windows.Forms.Form

    Private oDataAdapter As SqlDataAdapter
    Private oDataSet As DataSet
    Private oBMB As BindingManagerBase
    '....
    '....
```

Código fuente 577

En el evento de carga del formulario, aparte de la creación de los objetos de conexión, adaptador, etc., estableceremos el enlace entre los controles y el DataSet, como se muestra en el Código fuente 578.

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

    ' crear conexión
    Dim oConexion As New SqlConnection()
    oConexion.ConnectionString = "Server=(local);" & _
```

```
        "Database=Gestion;uid=sa;pwd=;"

    ' crear adaptador
    oDataAdapter = New SqlDataAdapter("SELECT * FROM Clientes", oConexion)

    ' crear commandbuilder
    Dim oCB As SqlCommandBuilder = New SqlCommandBuilder(oDataAdapter)

    ' crear dataset
    oDataSet = New DataSet()
    oDataAdapter.Fill(oDataSet, "Clientes")

    ' enlazar controles del formulario con el dataset;
    ' se debe utilizar un objeto Binding, al crear este objeto
    ' indicar en su constructor qué propiedad del control
    ' se debe enlazar, el dataset, y el nombre de tabla-columna;
    ' una vez creado el objeto Binding, añadirlo a la colección
    ' de enlaces de datos, DataBindings, del control que necesitemos,
    ' con el método Add() de dicha colección
    Dim oBind As Binding
    oBind = New Binding("Text", oDataSet, "Clientes.IDCliente")
    Me.txtIDCliente.DataBindings.Add(oBind)
    oBind = Nothing

    oBind = New Binding("Text", oDataSet, "Clientes.Nombre")
    Me.txtNombre.DataBindings.Add(oBind)
    oBind = Nothing

    oBind = New Binding("Text", oDataSet, "Clientes.FIngreso")
    'AddHandler oBind.Format, AddressOf FormatoFecha
    Me.txtFIngreso.DataBindings.Add(oBind)
    oBind = Nothing

    oBind = New Binding("Text", oDataSet, "Clientes.Credito")
    Me.txtCredito.DataBindings.Add(oBind)
    oBind = Nothing

    ' obtener del contexto de enlace del formulario,
    ' el enlace de un dataset y una tabla determinadas
    Me.oBMB = Me.BindingContext(oDataSet, "Clientes")

    Me.VerContadorReg()

End Sub

Private Sub VerContadorReg()
    ' mostrar información sobre el número de
    ' registro actual y registros totales
    ' en la tabla del dataset
    Me.lblRegistro.Text = "Registro: " & _
        Me.oBMB.Position + 1 & " de " & Me.oBMB.Count

End Sub
```

Código fuente 578

Debido al enlace automático, el código para las operaciones de navegación se simplifica en gran medida, como muestra el Código fuente 579, en el que vemos los manipuladores de evento para los botones de desplazamiento del formulario.

```
Private Sub btnAvanzar_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnAvanzar.Click
```

```
    ' avanzar a la siguiente fila;
    ' la actualización de los controles con los datos
    ' de la fila en la que acabamos de posicionarnos
    ' es automática, gracias al objeto BindingManagerBase
    Me.oBMB.Position += 1
    Me.VerContadorReg()

End Sub

Private Sub btnRetroceder_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnRetroceder.Click

    Me.oBMB.Position -= 1
    Me.VerContadorReg()

End Sub

Private Sub btnPrimero_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnPrimero.Click

    Me.oBMB.Position = 0
    Me.VerContadorReg()

End Sub

Private Sub btnUltimo_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnUltimo.Click

    Me.oBMB.Position = Me.oBMB.Count - 1
    Me.VerContadorReg()

End Sub
```

Código fuente 579

La Figura 348 muestra este formulario en ejecución.

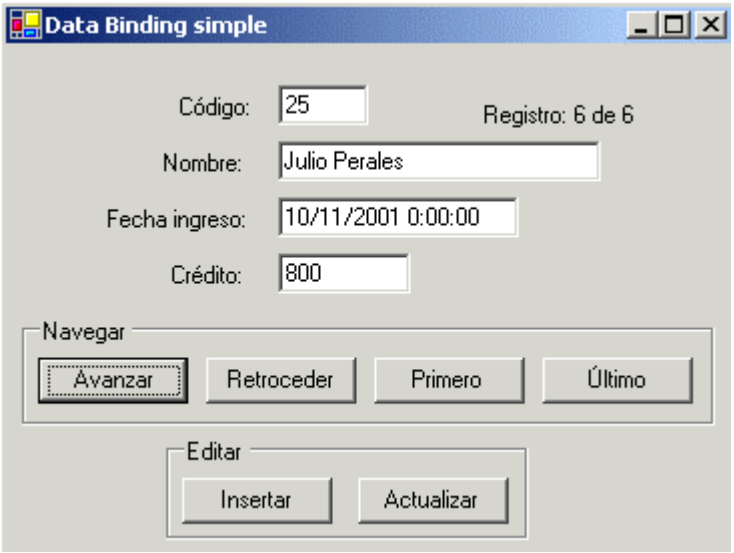


Figura 348. Ejecución del formulario con Data Binding.

Como detalle importante a observar en las operaciones de navegación entre los registros, destacaremos el hecho de que al mostrar el campo que contiene una fecha, dicho dato se muestra con toda la información al completo, fecha y hora, sin ningún formato específico.

Para conseguir en este caso, que la fecha se muestre con el formato que necesitemos, al crear el objeto Binding para ese control, deberemos asignar a su evento Format un procedimiento manipulador, que realice tal formateo y lo devuelva a través del objeto ConvertEventArgs, que recibe ese evento. Veamos estas operaciones en el Código fuente 580.

```
Private Sub Form1_Load(ByVal sender As Object, ByVal e As System.EventArgs) Handles
MyBase.Load
    '....
    oBind = New Binding("Text", oDataSet, "Clientes.FIngreso")
    AddHandler oBind.Format, AddressOf FormatoFecha
    Me.txtFIngreso.DataBindings.Add(oBind)
    oBind = Nothing
    '....
End Sub

' manipulador del Evento format del objeto Binding
Private Sub FormatoFecha(ByVal sender As Object, ByVal e As ConvertEventArgs)

    Dim dtFecha As DateTime
    dtFecha = e.Value
    e.Value = dtFecha.ToString("dd-MMMM-yyyy")

End Sub
```

Código fuente 580

La Figura 349 muestra ahora este control al haberle aplicado el formato.

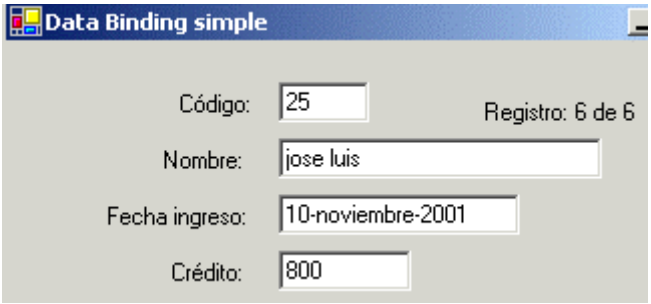


Figura 349. Control enlazado a datos, que muestra una fecha con formato personalizado.

El proceso de edición (inserción en este ejemplo), es muy similar al caso anterior. Aunque debemos tener en cuenta que debido a las particularidades del Data Binding, no podemos borrar el contenido de los TextBox, teclear datos e insertarlos, ya que eso realmente modificaría el registro sobre el que estábamos posicionados. Por ese motivo, en el botón Insertar, asignamos los valores directamente a las columnas del objeto DataRow. Ver el Código fuente 581.

```
Private Sub btnInsertar_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnInsertar.Click

    Dim oDataRow As DataRow
```

```
    ' crear un nuevo objeto fila
    oDataRow = Me.oDataSet.Tables("Clientes").NewRow()
    ' añadir datos a las columnas de la fila
    oDataRow("IDCliente") = 100
    oDataRow("Nombre") = "Isabel"
    oDataRow("FIngreso") = "12/9/01"
    oDataRow("Credito") = 228
    ' añadir el objeto fila a la colección
    ' de filas de la tabla
    Me.oDataSet.Tables("Clientes").Rows.Add(oDataRow)

End Sub

Private Sub btnActualizar_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnActualizar.Click

    Me.oDataAdapter.Update(Me.oDataSet, "Clientes")

End Sub
```

Código fuente 581

38

El control DataGrid, relaciones y vistas

DataGrid

Este control, del que ya realizamos una pequeña demostración en un apartado anterior, nos va a permitir realizar enlace complejo de datos con ADO .NET.

Se trata de la versión mejorada del control DataGrid de ADO, disponible en Visual Basic 6, pero con una serie de funcionalidades optimizadas, y otras nuevas añadidas.

Para utilizar algunas de sus características, crearemos un proyecto de prueba con el nombre DataGridPru (hacer clic aquí para acceder a este ejemplo), consistente en un formulario MDI, con una serie de opciones de menú, a través de las cuales, mostraremos diversas características de este control, y algunas otras adicionales sobre ADO .NET.

La opción de menú *DataGrid + Normal*, mostrará el formulario frmNormal, que contiene un sencillo DataGrid con una tabla. Podemos editar los registros de la tabla y añadir nuevos; al trabajar en desconexión, hasta que no pulsemos el botón Actualizar de este formulario, el objeto DataAdapter del mismo no actualizará los datos del DataSet hacia la base de datos física. Otra característica incluida por defecto es la ordenación de las filas por columna al hacer clic en su título. Finalmente, al redimensionar el formulario, también cambiará el tamaño del DataGrid, puesto que hemos utilizado su propiedad Anchor para anclarlo a todos los bordes de la ventana. La Figura 350 muestra este formulario.

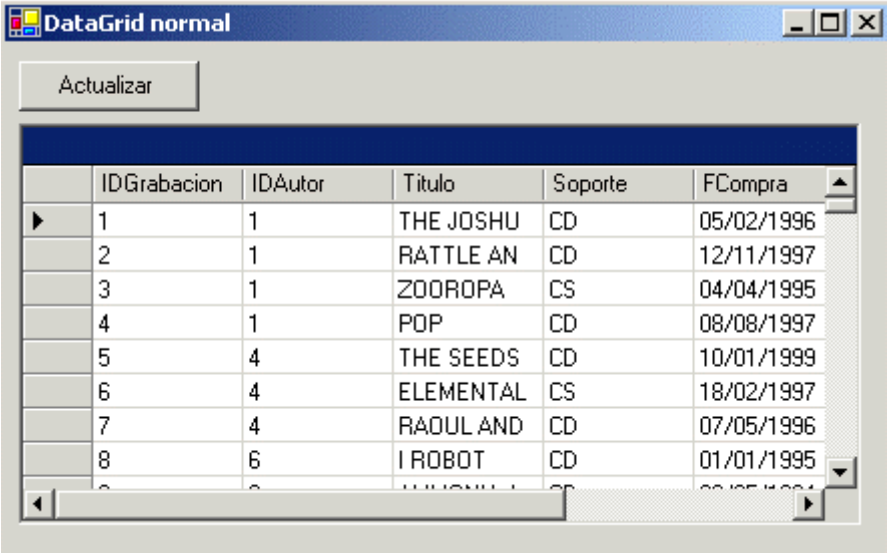


Figura 350. DataGrid editable.

El Código fuente 582 muestra el código principal de este formulario. Recordamos al lector, la necesidad de crear un objeto CommandBuilder para el DataAdapter, ya que en caso contrario, al intentar actualizar el DataSet contra la base de datos, se producirá un error.

```
Private oDataAdapter As SqlDataAdapter
Private oDataSet As DataSet

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

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

    ' crear conexión
    Dim oConexion As New SqlConnection()
    oConexion.ConnectionString = "Server=(local);" & _
        "Database=Musica;uid=sa;pwd=;"

    ' crear adaptador
    oDataAdapter = New SqlDataAdapter("SELECT * FROM Grabaciones", oConexion)

    ' crear commandbuilder
    Dim oCB As SqlCommandBuilder = New SqlCommandBuilder(oDataAdapter)

    ' crear dataset
    oDataSet = New DataSet()
    oDataAdapter.Fill(oDataSet, "Grabaciones")

    ' asignar dataset al datagrid
    Me.grdDatos.DataSource = oDataSet
    Me.grdDatos.DataMember = "Grabaciones"

End Sub
Private Sub btnActualizar_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnActualizar.Click
    Me.oDataAdapter.Update(oDataSet, "Grabaciones")
End Sub
```

Código fuente 582

Creación de un DataGrid a través de los asistentes del IDE

El modo más potente de crear un DataGrid es a través de código, ya que nos permite un mayor grado de manipulación de sus propiedades.

Sin embargo, para aquellas ocasiones en que necesitemos una vista rápida de los datos en un formulario para pruebas o similares, podemos utilizar los asistentes de Visual Studio .NET, en lo que a creación de conexiones, adaptadores, DataGrid, etc., se refiere.

Vamos a crear por lo tanto un nuevo formulario para el proyecto con el nombre frmGridAsist. Una vez añadido el diseñador, abriremos la pestaña *Explorador de servidores*, y haciendo clic derecho en su elemento *Conexiones de datos*, nos mostrará la ventana para la creación de una nueva conexión con una base de datos, en este caso de un servidor SQL Server; en ella introduciremos los valores necesarios para la conexión. Ver Figura 351.

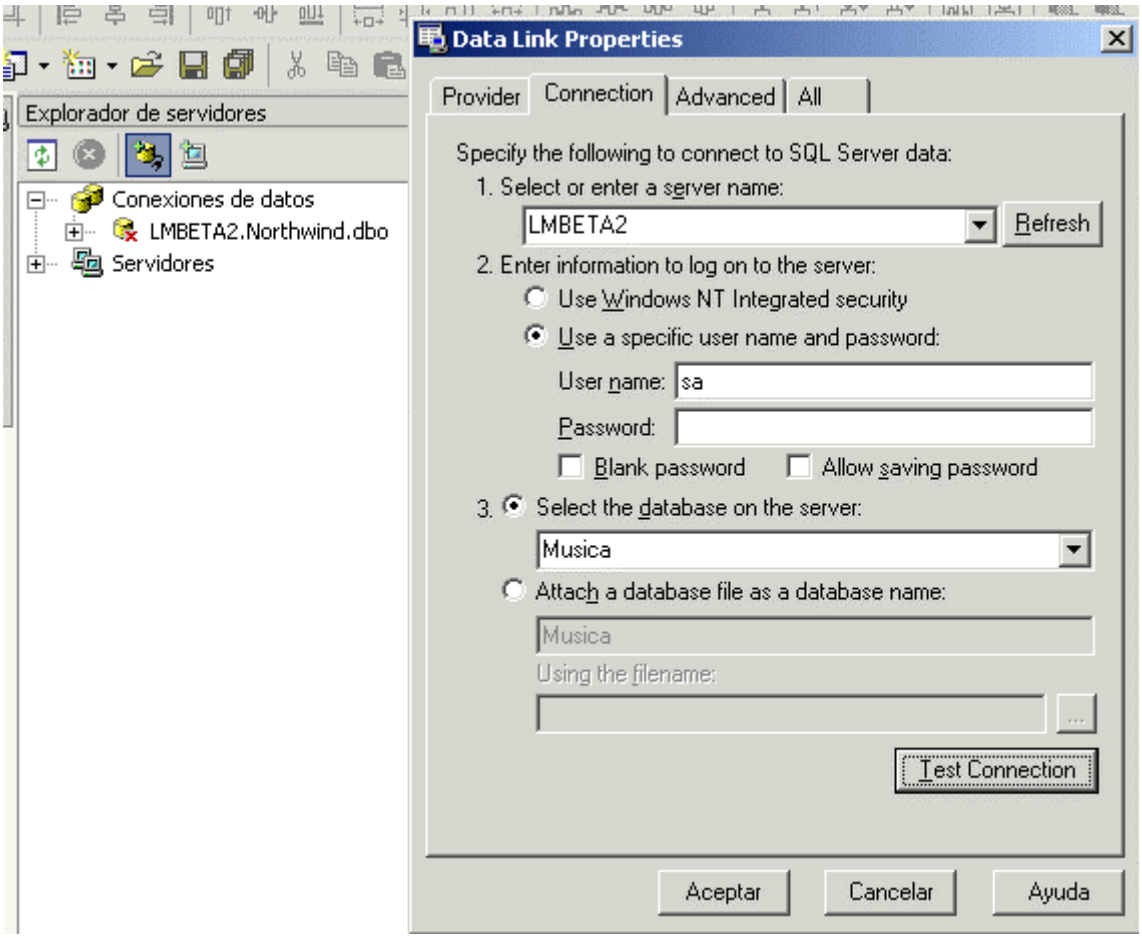


Figura 351. Creación de una conexión desde el Explorador de servidores.

En el siguiente paso, abriremos el *Cuadro de herramientas*, y pulsaremos la ficha Data, añadiendo al formulario un control SqlDataAdapter, lo que abrirá un asistente para la configuración de este control. Ver Figura 352.

Tras la ventana de presentación, al pulsar el botón Siguiente, deberemos elegir la conexión que el adaptador utilizará. Ver Figura 353.

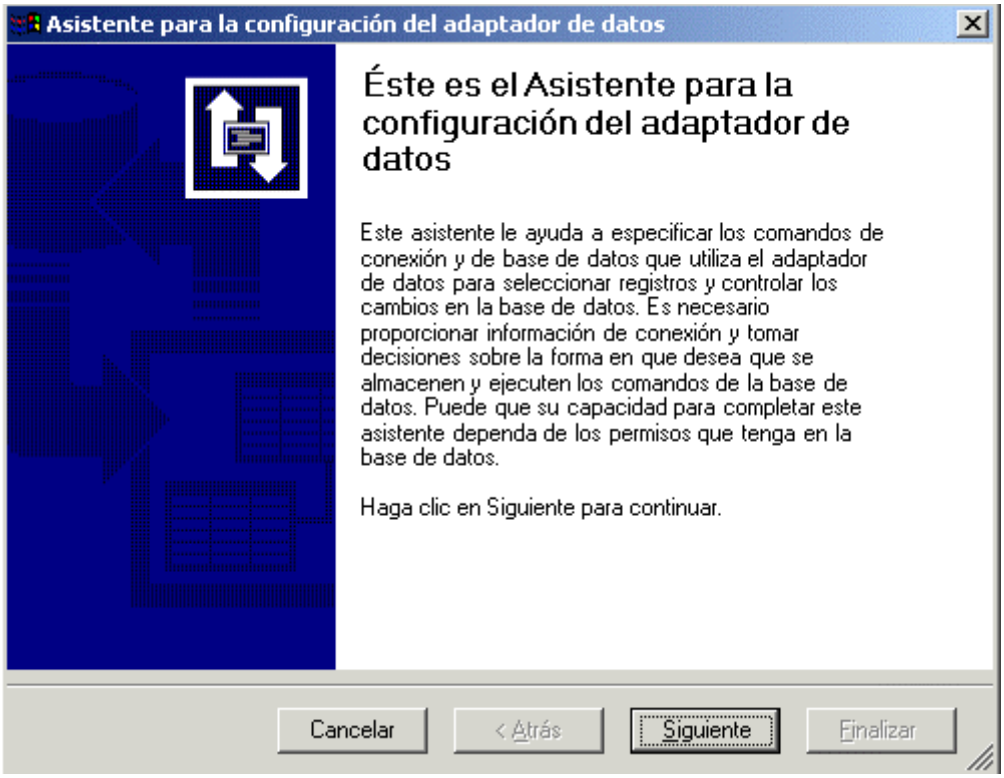


Figura 352. Asistente para configuración del control SqlDataAdapter.

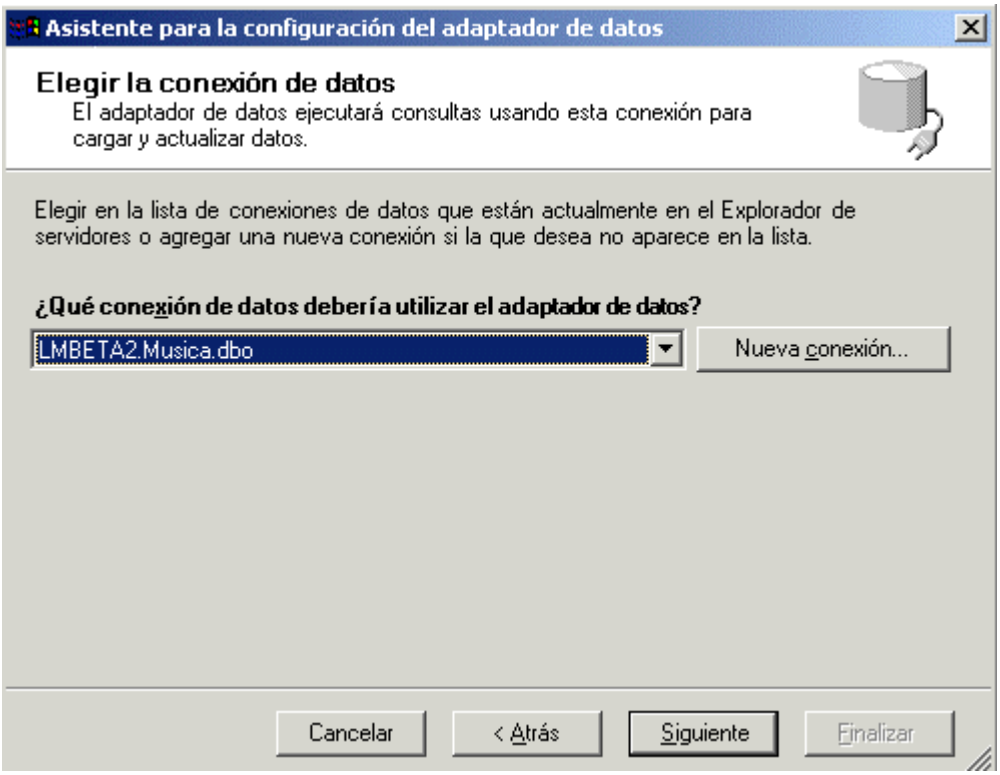


Figura 353. Selección de la conexión de datos.

A continuación seleccionaremos el tipo de consulta, en este caso una sencilla sentencia SQL. Ver Figura 354.

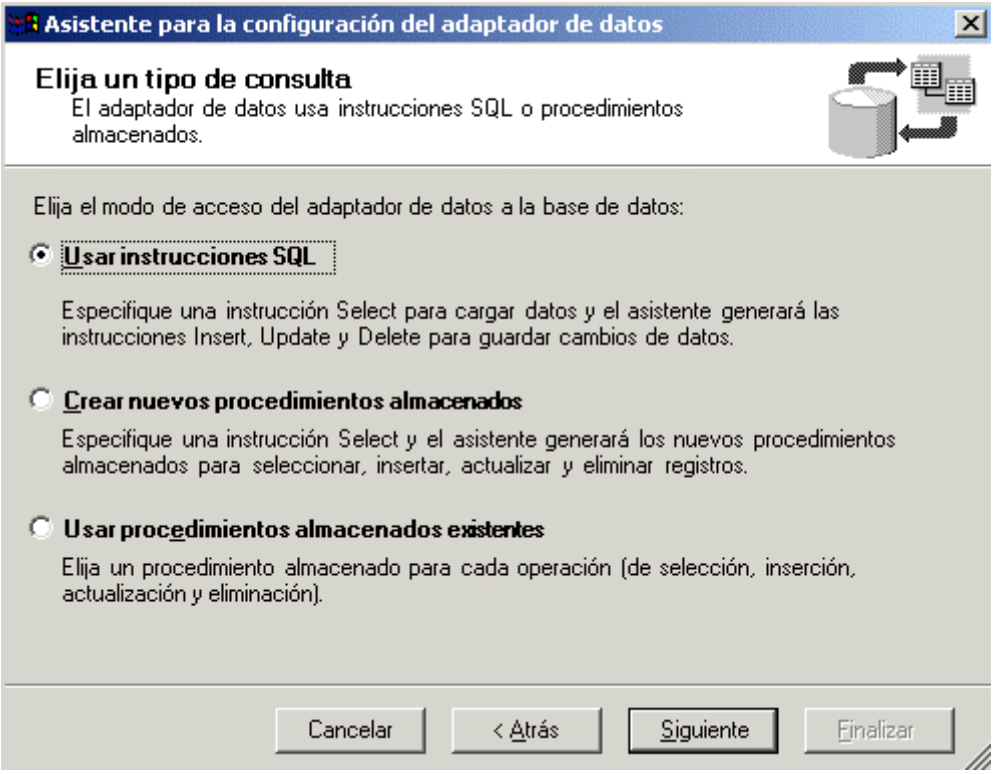


Figura 354. Selección del tipo de consulta que contendrá el adaptador.

Continuaremos con la escritura de la sentencia SQL que quedará incluida en el DataAdapter. Ver Figura 355.

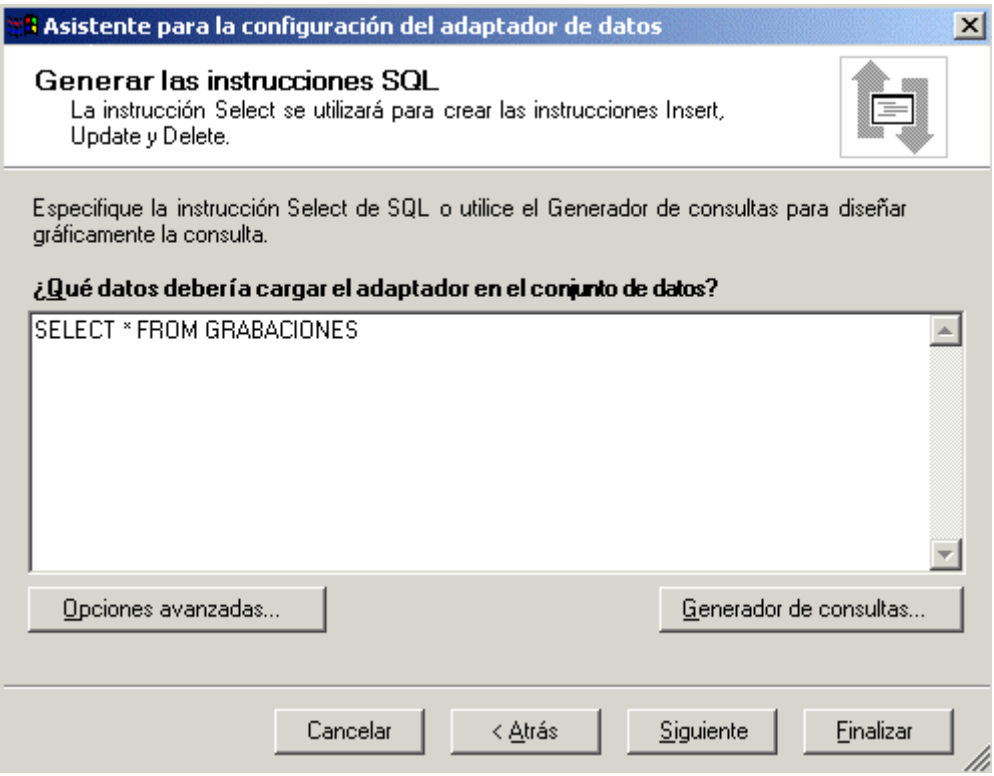


Figura 355. Escritura de la consulta SQL a generar.

Como paso final, se muestra un resumen de lo que este asistente ha generado en el DataAdapter. Figura 356.

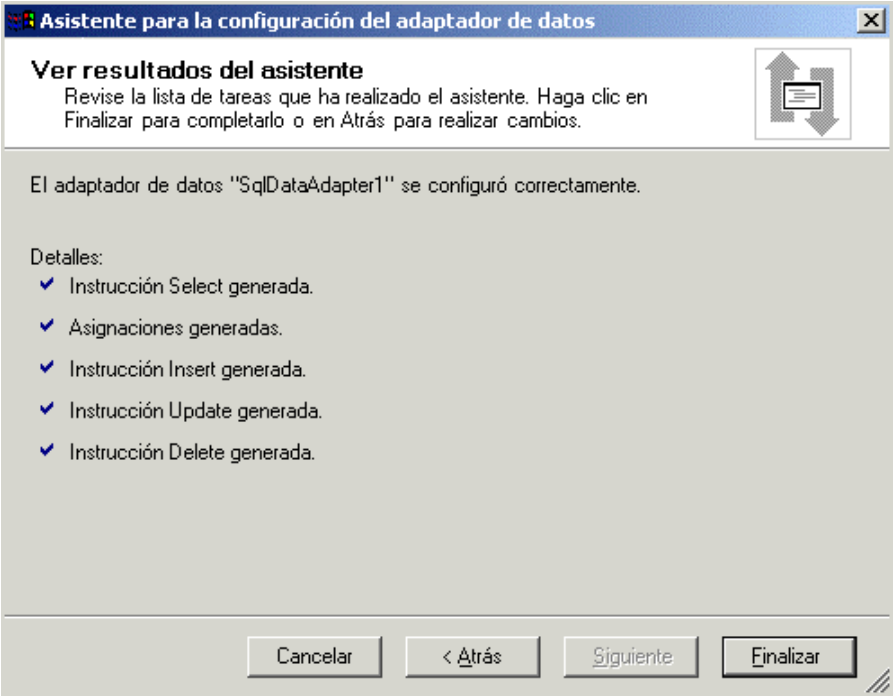


Figura 356. Resultados del asistente de generación del DataAdapter.

Finalizada la creación del adaptador de datos, seleccionaremos el menú Datos + Generar conjunto de datos del IDE, que nos mostrará una ventana en la que daremos el nombre del DataSet que utilizará el formulario, y nos permitirá elegir las tablas que contendrá. Ver Figura 357.

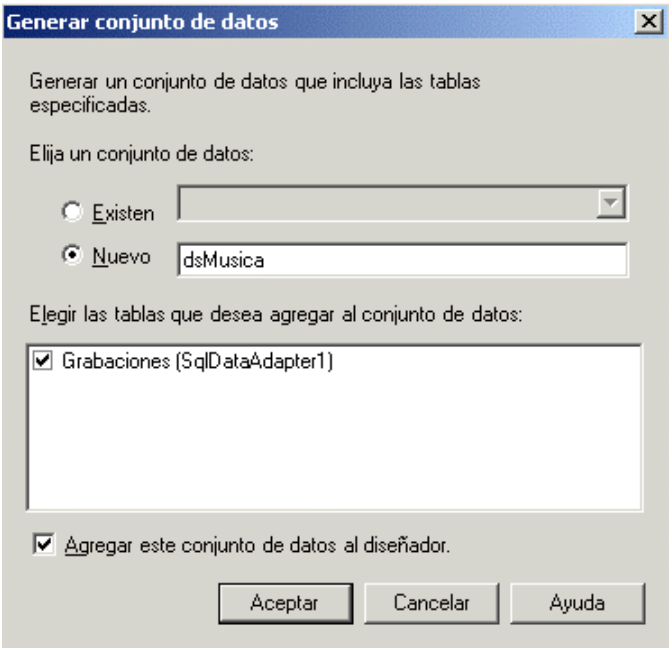


Figura 357. Creación del DataSet.

A continuación dibujaremos un DataGrid en el formulario, y pasaremos a su ventana de propiedades. En la propiedad DataSource asignaremos el DataSet que acabamos de crear, mientras que en la propiedad DataMember, seleccionaremos la tabla del DataSet que va a mostrar el DataGrid. Ver Figura 358.

Figura 358. Propiedades del DataGrid para la obtención de datos.

Completado este último paso, el DataGrid mostrará en tiempo de diseño, la disposición de las columnas de la tabla en su interior. Ver Figura 359.

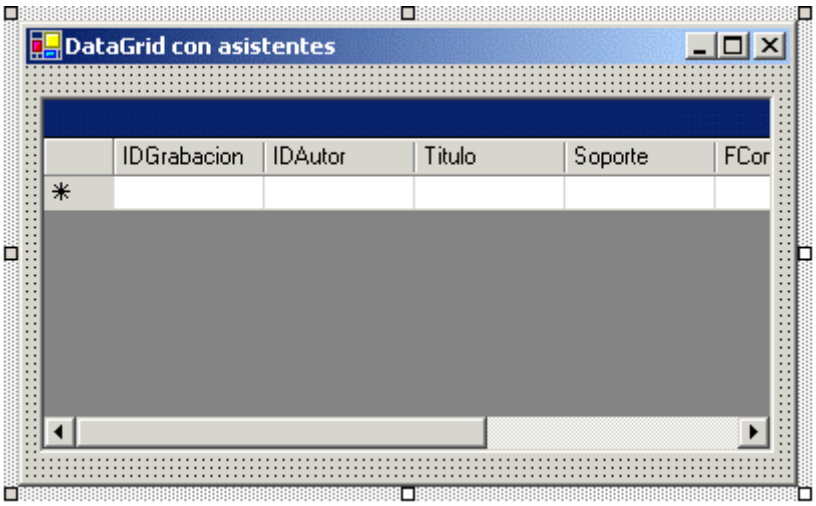


Figura 359. DataGrid mostrando información de las columnas de la tabla del DataSet.

En cuanto al código que debemos escribir, en el evento Load, inicializaremos el DataSet, rellenándolo a continuación mediante el DataAdapter. Ver Código fuente 583.

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

    Me.DsMusica1.Clear()
    Me.SqlDataAdapter1.Fill(Me.DsMusica1)

End Sub
```

Código fuente 583

Podremos ver este formulario en ejecución al seleccionar en el formulario principal del ejemplo, el menú *DataGrid + Asistente*.

Configurar las propiedades del DataGrid

En los casos anteriores, hemos creado un formulario con un DataGrid que tenía la apariencia visual por defecto de este control. Evidentemente, a través de las propiedades del DataGrid, tanto en diseño como en ejecución, podemos de un modo muy flexible y potente, cambiar la apariencia y el comportamiento de este control.

En el formulario frmGridProp, mostramos la misma información que en el anterior ejemplo, pero con una presentación totalmente distinta, al modificar algunas propiedades del DataGrid como BackColor, AlternatingBackColor, CaptionText, etc. Abriremos este formulario con la opción *DataGrid + Propiedades*, de la ventana MDI del proyecto. Ver Figura 360.

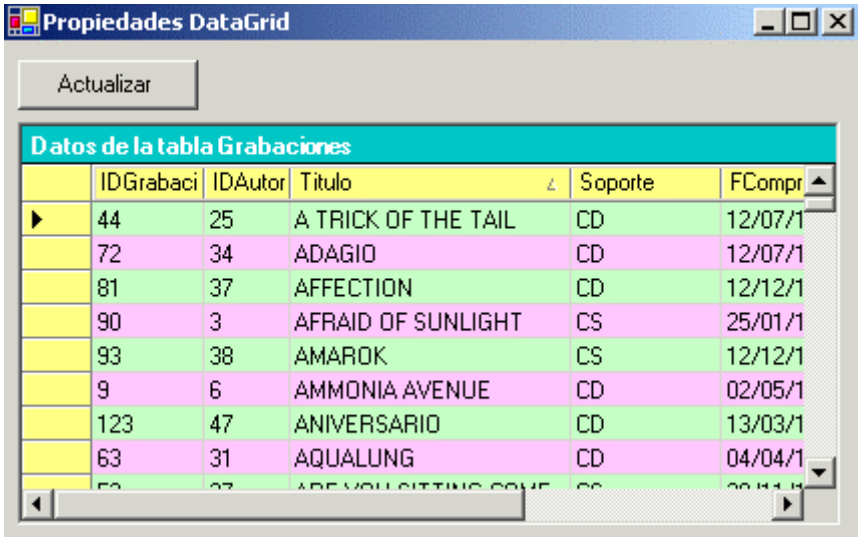


Figura 360. DataGrid con propiedades modificadas.

Configurar por código las propiedades del DataGrid

Supongamos ahora, que necesitamos por código modificar las propiedades no sólo del DataGrid en general, sino de algunas columnas del mismo. Esto es perfectamente factible mediante los objetos manipuladores de estilo, tanto del propio DataGrid, como de cada una de las columnas que lo componen.

La clase DataGridTableStyle, nos permitirá crear objetos que contengan una configuración de tabla personalizada, que después añadiremos al DataGrid.

Por otra parte, mediante la clase DataGridTextBoxColumn, crearemos objetos con la configuración particular para cada columna. La propiedad clave de estos objetos es MappingName, que contiene una cadena con el nombre de la columna de la tabla del DataSet, que será la que muestre dicha columna.

El formulario frmGridPropCod que abriremos con la opción de menú *DataGrid + Prop.código*, hace uso en el evento de carga de la ventana, de estos objetos para variar el aspecto por defecto que tiene su DataGrid. El Código fuente 584 muestra las instrucciones empleadas.

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

```
    ' crear conexión
    Dim oConexion As New SqlConnection()
    oConexion.ConnectionString = "Server=(local);" & _
        "Database=Musica;uid=sa;pwd=;"

    ' crear adaptador
    oDataAdapter = New SqlDataAdapter("SELECT * FROM Grabaciones", oConexion)

    ' crear commandbuilder
    Dim oCB As SqlCommandBuilder = New SqlCommandBuilder(oDataAdapter)

    ' crear dataset
    oDataSet = New DataSet()
    oDataAdapter.Fill(oDataSet, "Grabaciones")

    ' asignar dataset al datagrid
    Me.grdDatos.DataSource = oDataSet
    Me.grdDatos.DataMember = "Grabaciones"

    ' configurar grid por código
    Me.grdDatos.Anchor = AnchorStyles.Bottom + AnchorStyles.Left +
AnchorStyles.Right + AnchorStyles.Top
    Me.grdDatos.CaptionText = "El listado de las grabaciones"
    Me.grdDatos.CaptionBackColor = Color.Turquoise
    Me.grdDatos.CaptionForeColor = Color.Black

    ' crear un objeto para estilos del datagrid
    Dim oEstiloGrid As New DataGridTableStyle()
    oEstiloGrid.MappingName = "Grabaciones"
    oEstiloGrid.BackColor = Color.LightGoldenrodYellow
    oEstiloGrid.AlternatingBackColor = Color.Aquamarine

    ' crear objetos de columna-grid para cada
    ' columna de la tabla a mostrar en el datagrid
    Dim oColGrid As DataGridTextBoxColumn

    ' configurar cada objeto de columna-grid
    oColGrid = New DataGridTextBoxColumn()
    oColGrid.TextBox.Enabled = False
    oColGrid.Alignment = HorizontalAlignment.Center
    oColGrid.HeaderText = "Descripción grabac."
    ' nombre de la columna del dataset que
    ' se mapea hacia esta columna del grid
    oColGrid.MappingName = "Titulo"
    oColGrid.Width = 300
    ' añadir la columna al objeto que contiene
    ' los estilos del datagrid, en concreto,
    ' a la colección de estilos de columna
    oEstiloGrid.GridColumnStyles.Add(oColGrid)
    oColGrid = Nothing

    oColGrid = New DataGridTextBoxColumn()
    oColGrid.TextBox.Enabled = False
    oColGrid.Alignment = HorizontalAlignment.Left
    oColGrid.HeaderText = "Fecha COMPRA"
    oColGrid.MappingName = "FCompra"
    oColGrid.Width = 110
    oColGrid.Format = "ddd, d-MMM-yyy"
    oEstiloGrid.GridColumnStyles.Add(oColGrid)
    oColGrid = Nothing

    oColGrid = New DataGridTextBoxColumn()
    oColGrid.TextBox.Enabled = False
    oColGrid.Alignment = HorizontalAlignment.Right
    oColGrid.HeaderText = "Valor pagado"
    oColGrid.MappingName = "Precio"
```

```
    oColGrid.Width = 85
    oColGrid.Format = "#,#"
    oEstiloGrid.GridColumnStyles.Add(oColGrid)
    oColGrid = Nothing

    ' una vez creadas todas las columnas de
    ' estilos para el grid, añadir el objeto
    ' que contiene el estilo personalizado
    ' a la colección de estilos de tablas
    ' del datagrid
    Me.grdDatos.TableStyles.Add(oEstiloGrid)

End Sub
```

Código fuente 584

La Figura 361 muestra el resultado de esta modificación sobre el DataGrid.

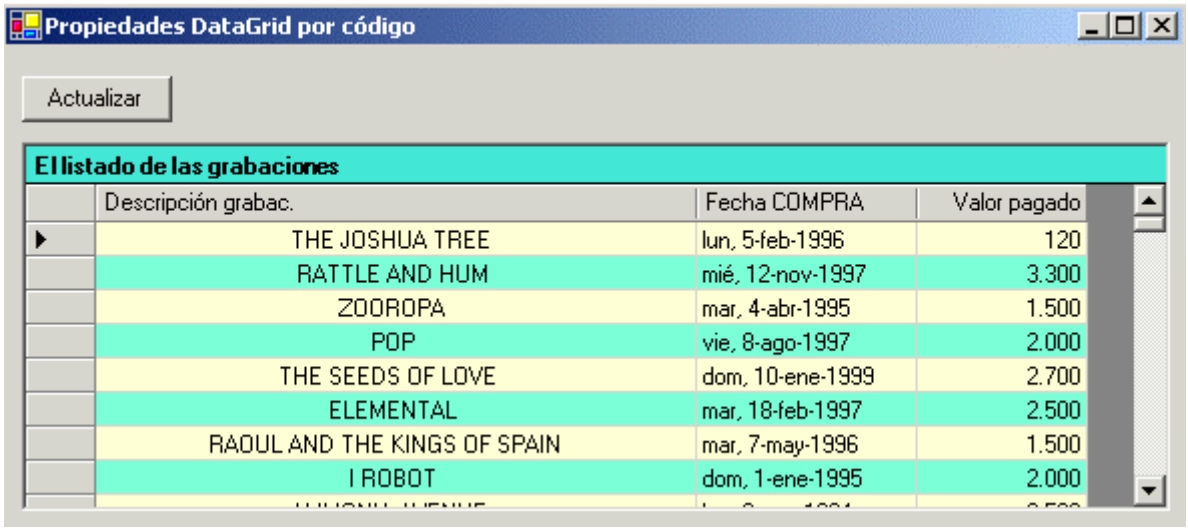


Figura 361. DataGrid modificado totalmente por código.

Selección de tabla en el DataGrid

Al construir un DataSet, podemos utilizar distintos objetos DataAdapter para rellenarlo con diversas tablas. Como hemos visto en los anteriores ejemplos, para mostrar datos en un DataGrid, debemos asignar el DataSet a su propiedad DataSource, y el nombre de la tabla a mostrar en la propiedad DataMember. Sin embargo, si obviamos la asignación a DataMember, gracias a los mecanismos de Data Binding, el propio DataGrid, nos ofrecerá la oportunidad de seleccionar la tabla a mostrar.

El formulario frmGridTablas, que abrimos mediante la opción de menú *DataGrid + Varias tablas* del proyecto de ejemplo, dispone de este comportamiento. En su evento Load crearemos dos DataAdapter que usaremos para llenar un DataSet. Ver Código fuente 585.

```
Private Sub frmGridTablas_Load(ByVal sender As Object, ByVal e As System.EventArgs)
Handles MyBase.Load
    ' crear conexión
    Dim oConexion As New SqlConnection()
    oConexion.ConnectionString = "Server=(local);" & _
```

```
        "Database=Musica;uid=sa;pwd=;"

    ' crear adaptadores
    Dim oDAAutores As New SqlDataAdapter("SELECT * FROM Autores", oConexion)
    Dim oDAGrabaciones As New SqlDataAdapter("SELECT * FROM Grabaciones",
oConexion)

    ' crear dataset
    Dim oDataSet As New DataSet()
    oDAAutores.Fill(oDataSet, "Autores")
    oDAGrabaciones.Fill(oDataSet, "Grabaciones")

    ' asignar dataset a datagrid
    Me.grdDatos.DataSource = oDataSet

End Sub
```

Código fuente 585

Como al asignar el DataSet al DataGrid no hemos indicado qué tabla queremos que muestre, el DataGrid en el formulario visualizará un nodo que al expandir, nos permitirá seleccionar la tabla a mostrar. Podremos contraer dicha tabla para seleccionar otra, y así sucesivamente. Ver Figura 362.

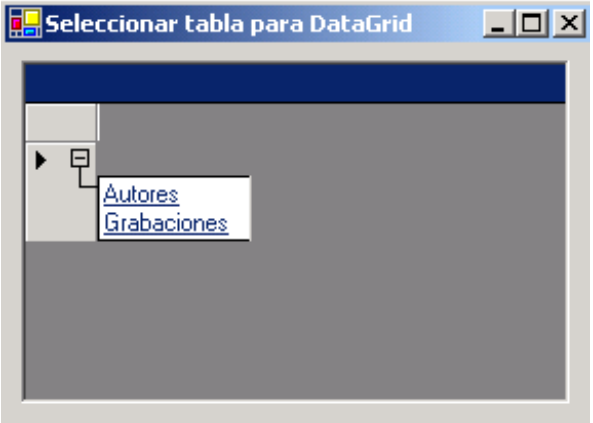


Figura 362. Selección de tabla a mostrar en un DataGrid,

Relaciones entre tablas mediante objetos DataRelation

Los objetos DataRelation nos permiten establecer una relación entre dos tablas (objetos DataTable) de un DataSet, a través de una columna o campo común (objetos DataColumn).

Para demostrar la creación de relaciones con estos objetos, utilizaremos el proyecto de ejemplo RelacionarDatos (hacer clic aquí para acceder a este ejemplo), en el que a través de un formulario MDI, crearemos varios formularios hijos, cada uno con un tipo de relación.

Obtener tablas relacionadas mediante código

En primer lugar, la opción de menú *Relacionar + Manual*, muestra el formulario frmManual, en el que al cargar el formulario, creamos una relación entre dos tablas, Customers y Orders, por un campo clave. Después llenamos un ComboBox con datos de la tabla Customers.

Al seleccionar un valor del ComboBox, se tomarán las filas relacionadas de la tabla Orders y se llenará con ellas un ListBox. El código necesario podemos verlo en el Código fuente 586.

```
Private Sub frmManual_Load(ByVal sender As Object, ByVal e As System.EventArgs)
Handles MyBase.Load
    ' crear conexión
    Dim oConexion As New SqlConnection()
    oConexion.ConnectionString = "server=(local);" & _
        "database=Northwind;uid=sa;pwd=;"

    ' crear adaptadores
    Dim daCustomers As New SqlDataAdapter("SELECT * FROM Customers", oConexion)
    Dim daOrders As New SqlDataAdapter("SELECT * FROM Orders", oConexion)

    ' instanciar dataset
    oDataSet = New DataSet()

    oConexion.Open()
    ' utilizar los dataadapters para llenar el dataset con tablas
    daCustomers.Fill(oDataSet, "Customers")
    daOrders.Fill(oDataSet, "Orders")
    oConexion.Close()

    ' relacionar las dos tablas del dataset por campo común
    oDataSet.Relations.Add("Customers_Orders", _
        oDataSet.Tables("Customers").Columns("CustomerID"), _
        oDataSet.Tables("Orders").Columns("CustomerID"))

    ' llenar el combobox con los nombres de cliente
    Dim oDataRow As DataRow
    For Each oDataRow In oDataSet.Tables("Customers").Rows
        Me.cboCustomers.Items.Add(oDataRow("CustomerID") & _
            "-" & oDataRow("CompanyName"))
    Next
End Sub

' cada vez que se selecciona un valor en el combo
' se produce este evento
Private Sub cboCustomers_SelectedIndexChanged(ByVal sender As Object, ByVal e As
System.EventArgs) Handles cboCustomers.SelectedIndexChanged
    ' limpiar los valores del listbox
    Me.lstOrders.Items.Clear()

    Dim drFilaPadre As DataRow
    ' obtener la fila de la tabla maestra: Customers
    drFilaPadre = oDataSet.Tables("Customers").Rows(Me.cboCustomers.SelectedIndex)

    Dim drFilasHijas() As DataRow
    ' obtener las filas hijas de la tabla Orders,
    ' gracias a la relación Customers-Orders
    drFilasHijas = drFilaPadre.GetChildRows("Customers_Orders")

    Dim drFila As DataRow
    ' rellenar el listbox con valores de las filas hijas
    For Each drFila In drFilasHijas
        Me.lstOrders.Items.Add(drFila("CustomerID") & _
            "-" & drFila("OrderID") & _
            "-" & drFila("OrderDate"))
    Next

End Sub
```

Código fuente 586

La Figura 363 muestra este formulario.

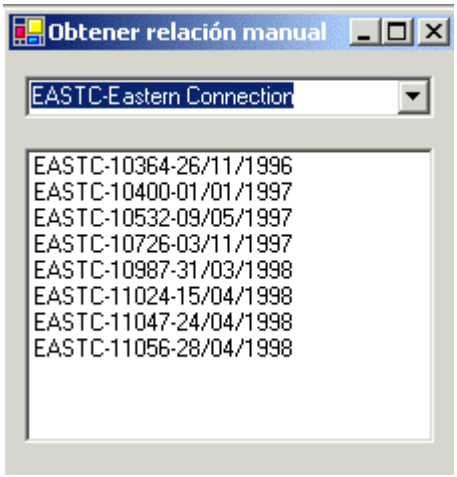


Figura 363. Obtención de filas relacionadas de forma manual.

Visualizar datos relacionados en modo maestro-detalle en un DataGrid

Podemos ahorrar la escritura de la instrucciones que se encargan de obtener las filas hijas, relacionadas con la fila seleccionada de la tabla padre, empleando un DataGrid. Este control implementa de forma transparente todos los mecanismos necesarios gracias al Data Binding, por lo que, una vez creada la relación, sólo hemos de asignar a su propiedad DataSource, la tabla padre del DataSet.

El formulario frmRelacGrid, al que accederemos con el menú *Relacionar* + *DataGrid*, es un ejemplo de este tipo de relación de datos. El código de su evento Load es igual al del anterior formulario, por lo que el Código fuente 587 sólo muestra la creación de la relación en el DataSet, y la asignación de la tabla maestra al DataGrid.

```
' relacionar las dos tablas del dataset por campo común
oDataSet.Relations.Add("Customers_Orders", _
    oDataSet.Tables("Customers").Columns("CustomerID"), _
    oDataSet.Tables("Orders").Columns("CustomerID"))

' asignar la tabla maestra al datagrid
Me.grdDatos.DataSource = oDataSet.Tables("Customers")
```

Código fuente 587

Al abrir este formulario, se visualizarán los datos de la tabla maestra Customers. Cada fila contiene un nodo expandible, que al ser pulsado muestra la relación existente. Si volvemos a hacer clic sobre la relación, se mostrarán en este caso las filas hijas de la tabla Orders, relacionadas con la que hemos seleccionado en la tabla padre. Ver Figura 364 y Figura 365.

En todo momento, desde la vista de las tablas hijas, podemos volver a la vista de la tabla padre, haciendo clic en el icono con forma de flecha situado en el título del DataGrid.

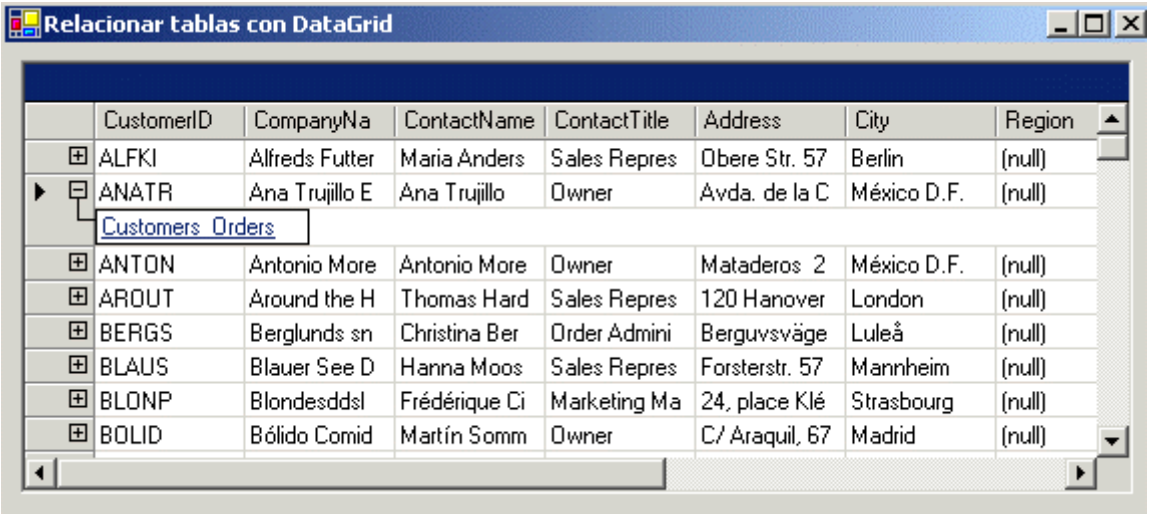


Figura 364. DataGrid mostrando filas de tabla maestra.

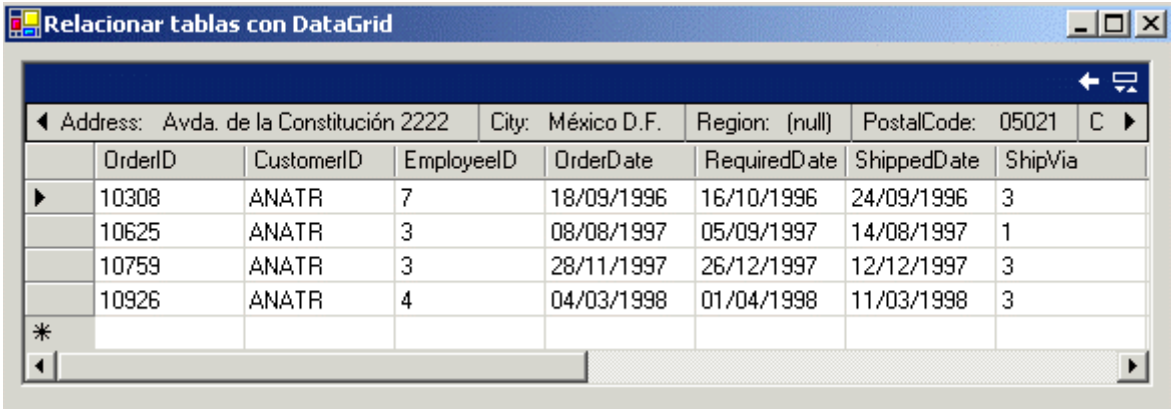


Figura 365. DataGrid mostrando filas de tabla detalle.

Mostrar una relación maestro-detalle en dos DataGrid

Podemos separar la visualización de las tablas maestro y detalle en dos DataGrid independientes. Para sincronizar ambos controles, debemos asignar al que actuará como detalle, una cadena con el nombre de la tabla maestra, junto con el nombre de la relación, empleando el siguiente formato: TablaMaestra.Relación.

El formulario frmDosGrid, que abriremos con la opción de menú *Relacionar + Dos DataGrid*, es un ejemplo de este tipo de organización de datos. En el Código fuente 588 mostramos la parte del evento Load encargada de la creación de la relación entre tablas y asignación a los DataGrid.

```
' relacionar las dos tablas del dataset por campo común
oDataSet.Relations.Add("Customers_Orders", _
    oDataSet.Tables("Customers").Columns("CustomerID"), _
    oDataSet.Tables("Orders").Columns("CustomerID"))

' asignar al datagrid maestro la tabla Customers
Me.grdCustomers.DataSource = oDataSet
Me.grdCustomers.DataMember = "Customers"

' asignar al datagrid detalles la relación
```

```
' que acabamos de crear por código
Me.grdOrders.DataSource = oDataSet
Me.grdOrders.DataMember = "Customers.Customers_Orders"
```

Código fuente 588

La Figura 366 muestra el formulario con ambos DataGrid trabajando en modo conjunto; al hacer clic en una fila del DataGrid maestro, el DataGrid detalle se actualizará con los datos relacionados.

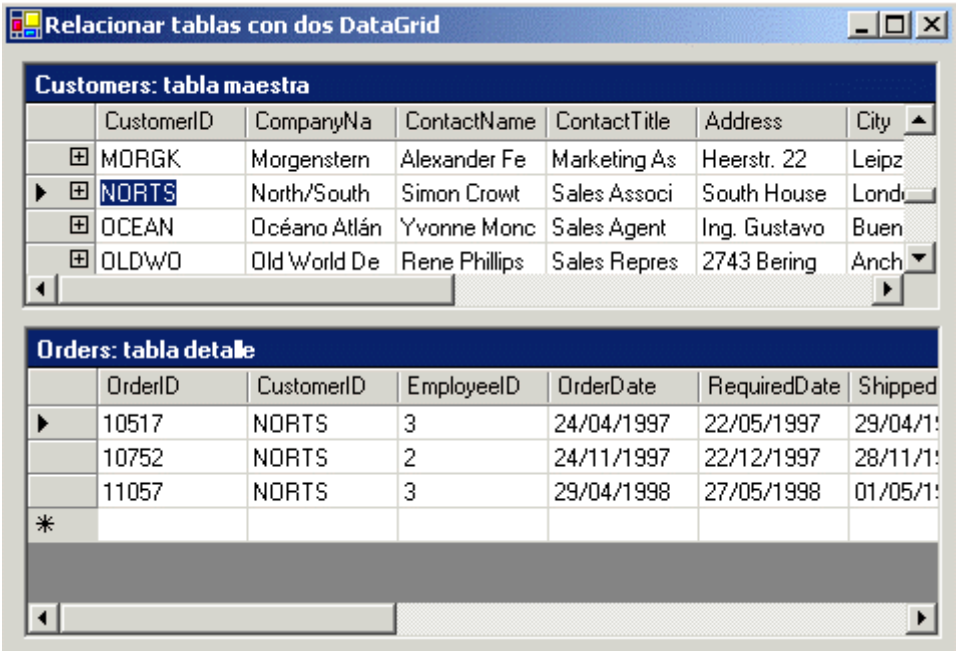


Figura 366. Relación maestro-detalle en dos DataGrid separados.

Relación maestro-detalle en múltiples DataGrid

En este caso se trata de disponer de varios DataGrid maestros y uno para detalles, de forma que al hacer clic sobre cualquiera de los maestros, se muestre la información relacionada en el detalle.

El formulario del proyecto encargado de este ejemplo será frmVariosGrid. Respecto al código, sólo tenemos que asignar al nuevo DataGrid maestro la información de la tabla principal. Ver Código fuente 589.

```
' relacionar las dos tablas del dataset por campo común
oDataSet.Relations.Add("Customers_Orders", _
    oDataSet.Tables("Customers").Columns("CustomerID"), _
    oDataSet.Tables("Orders").Columns("CustomerID"))

' asignar al datagrid maestro la tabla Customers
Me.grdCustomers.DataSource = oDataSet
Me.grdCustomers.DataMember = "Customers"

' asignar al segundo datagrid maestro la tabla Customers
Me.grdCustomersB.DataSource = oDataSet
Me.grdCustomersB.DataMember = "Customers"
```

```
' asignar al datagrid detalles la relación
' que acabamos de crear por código
Me.grdOrders.DataSource = oDataSet
Me.grdOrders.DataMember = "Customers.Customers_Orders"
```

Código fuente 589

Veamos el resultado de la ejecución en la Figura 367.

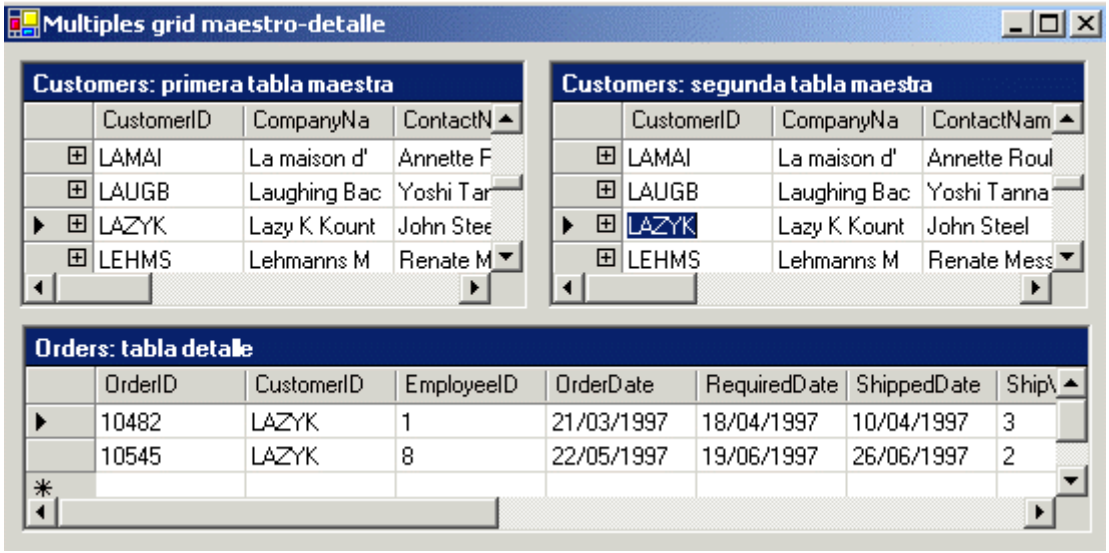


Figura 367. Varios DataGrid maestros contra uno de detalle.

Vistas y ordenación de datos con la clase DataView

La clase DataView nos permite la aplicación de vistas personalizadas a partir de una tabla contenida en un DataSet, así como la ordenación y búsqueda de filas.

En ADO clásico, para disponer de varias vistas de una misma tabla, debíamos crear diferentes objetos Recordset, lo cual provocaba el consumo de una gran cantidad de recursos.

Este aspecto ha cambiado profundamente en ADO .NET, ya que partiendo de un objeto DataTable situado en un DataSet, vamos a definir varias vistas simultáneamente, ordenar y buscar registros, con la ventaja de que el consumo de recursos es menor, puesto que los objetos DataView se alimentan del mismo DataTable. Para realizar algunas pruebas, se acompaña el proyecto Vistas (hacer clic aquí para acceder al ejemplo).

El DataSet del formulario de pruebas va a estar compuesto por dos tablas. El Código fuente 590 muestra el evento de carga del formulario.

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

    ' crear conexión
    Dim oConexion As New SqlConnection()
    oConexion.ConnectionString = "Server=(local);" & _
        "Database=Northwind;uid=sa;pwd=;"
```

```
    'crear dataset
    oDataSet = New DataSet()

    Dim oDataAdapter As SqlDataAdapter
    ' crear un adaptador de datos para la tabla Customers
    oDataAdapter = New SqlDataAdapter("SELECT * FROM Customers", oConexion)

    ' añadir tabla al dataset con el adaptador
    oDataAdapter.Fill(oDataSet, "Customers")
    oDataAdapter = Nothing

    ' crear un adaptador de datos para la tabla Products
    oDataAdapter = New SqlDataAdapter("SELECT * FROM Products", oConexion)

    ' añadir tabla al dataset con el adaptador
    oDataAdapter.Fill(oDataSet, "Products")
    oDataAdapter = Nothing

End Sub
```

Código fuente 590

Vistas por código y DefaultView

Podemos crear una vista instanciando un objeto de la clase DataView, o también obteniendo la denominada vista por defecto de una tabla de un DataSet, a través de la propiedad DefaultView del objeto DataTable. La opción de menú *Vistas + Normal* del formulario, crea dos vistas de esta manera. Ver Código fuente 591.

```
Private Sub mnuNormal_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles mnuNormal.Click
    ' crear una vista por código y asignarla
    ' a un datagrid
    Dim dvNormal As DataView
    dvNormal = New DataView(oDataSet.Tables("Customers"))
    Me.grdDatos.CaptionText = "Customers"
    Me.grdDatos.DataSource = dvNormal

    ' tomar la vista por defecto de una tabla
    ' del dataset y asignarla a un datagrid
    Me.grdDatosBIS.CaptionText = "Products"
    Me.grdDatosBIS.DataSource = oDataSet.Tables("Products").DefaultView

End Sub
```

Código fuente 591

La Figura 368 muestra estas vistas en sendos DataGrid del formulario.

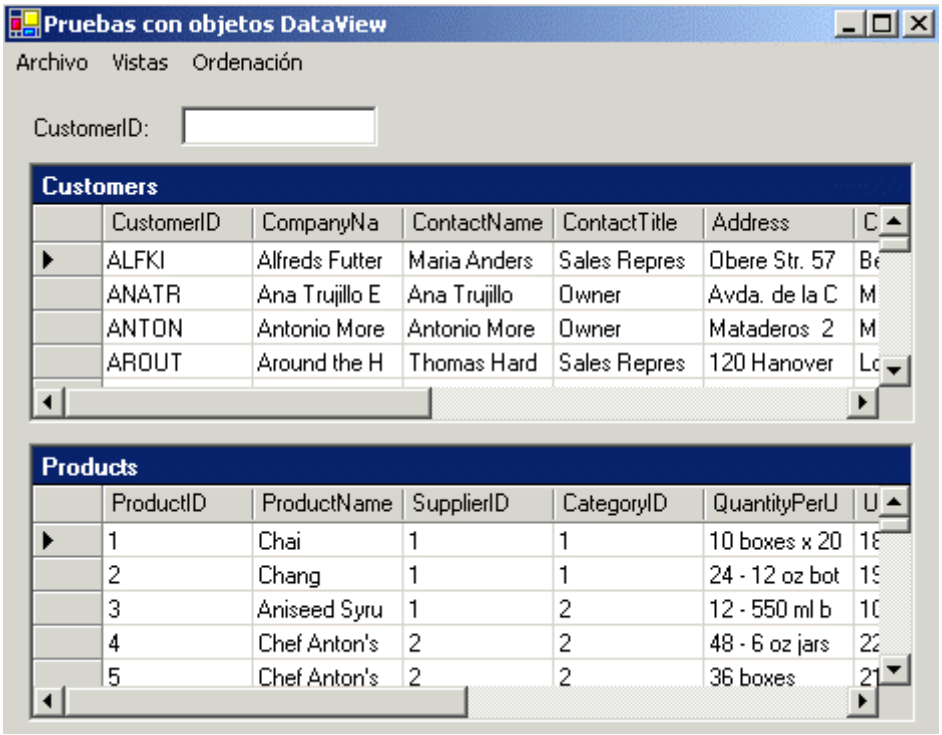


Figura 368. Objetos DataView creados por código y obtenido de DataTable.DefaultView.

Filtros con objetos DataView

La propiedad RowFilter de la clase DataView nos permite asignar a este objeto, una cadena con la expresión de filtro, que en una consulta en lenguaje SQL sería la parte correspondiente a la partícula Where.

El Código fuente 592 muestra el código de la opción de menú *Vistas + País*, del formulario de ejemplo, en la que se crea un filtro que se muestra posteriormente en un DataGrid.

```
Private Sub mnuPais_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles mnuPais.Click

    ' crear dataview
    Dim oDataView As New DataView()
    oDataView.Table = oDataSet.Tables("Customers")
    ' establecer un filtro
    oDataView.RowFilter = "Country='Spain'"

    Me.grdDatos.CaptionText = "Filtrar Customers por país Spain"
    Me.grdDatos.DataSource = oDataView

End Sub
```

Código fuente 592

La Figura 369 muestra las filas de la tabla con el filtro aplicado.

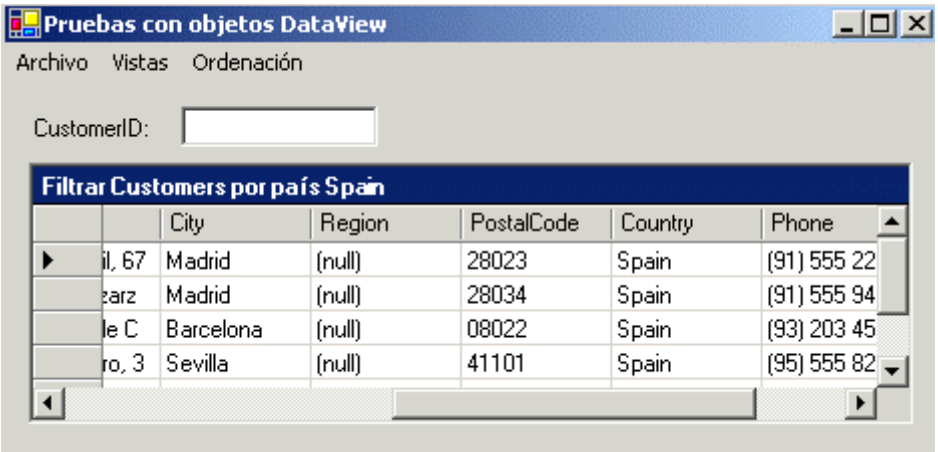


Figura 369. DataView con filtro.

Como hemos comentado anteriormente, a partir de un DataTable podemos obtener varios filtros mediante distintos objetos DataView, sin que ello suponga una penalización en el consumo de recursos. Para demostrar este punto, la opción *Vistas + Combinada*, crea una vista basada en un filtro combinado, y una vista normal, ambas empleando la misma tabla base. Veamos el Código fuente 593.

```
Private Sub mnuCombinada_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles mnuCombinada.Click
    ' tomar la tabla Customers del dataset y aplicar...

    ' ...filtro combinado por dos campos y depositar en un datagrid
    Dim oDataView As New DataView()
    oDataView.Table = oDataSet.Tables("Customers")
    oDataView.RowFilter = "ContactTitle LIKE '%Manager%' AND Country IN
('Spain','USA')"

    Me.grdDatos.CaptionText = "Filtro combinado por campos ContactTitle y Country"
    Me.grdDatos.DataSource = oDataView

    ' ...filtro por un campo y depositar en otro datagrid
    Dim oDV As New DataView()
    oDV.Table = oDataSet.Tables("Customers")
    oDV.RowFilter = "ContactName LIKE '%an%'"

    Me.grdDatosBIS.CaptionText = "Filtro por campo ContactName"
    Me.grdDatosBIS.DataSource = oDV

End Sub
```

Código fuente 593

La Figura 370 muestra el formulario con los diversos filtros establecidos

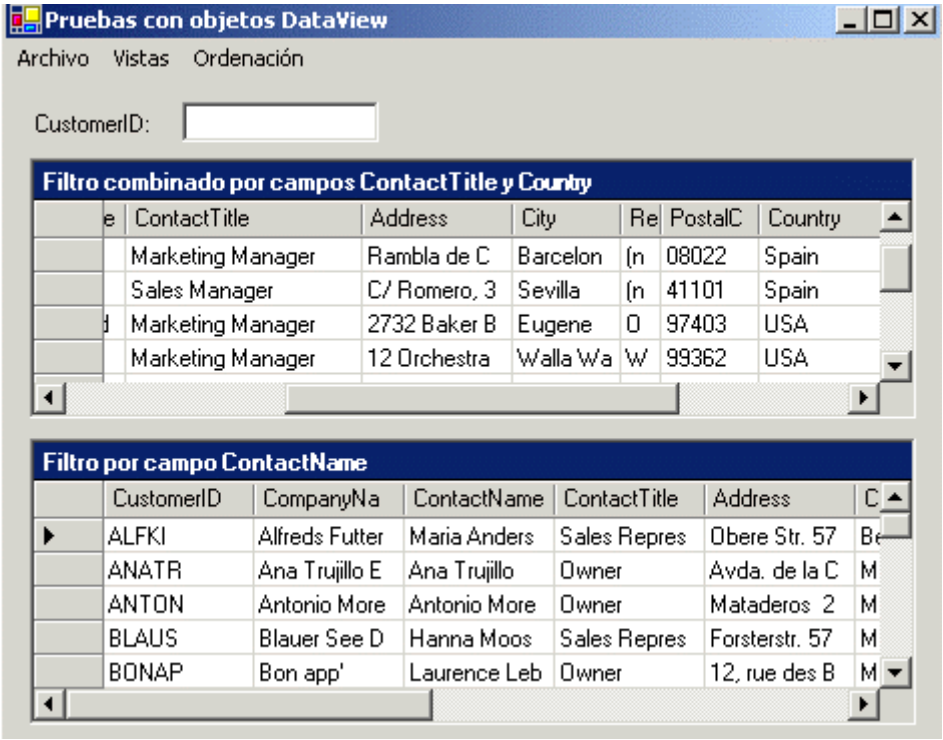


Figura 370. Filtros combinados con objetos DataView.

Búsquedas con DataView

Estableciendo el adecuado filtro a un objeto DataView, podemos realizar búsquedas de registros en tablas, como muestra el Código fuente 594, correspondiente a la opción de menú *Vistas + Buscar fila*, del formulario de ejemplo. Deberemos previamente, haber escrito en el TextBox del formulario, el identificador de la tabla Customers a buscar.

```
Private Sub mnuBuscarFila_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles mnuBuscarFila.Click

    ' crear un dataview y buscar una fila en la vista
    ' estableciendo un filtro
    Dim oDataView As New DataView()
    oDataView.Table = oDataSet.Tables("Customers")
    oDataView.RowFilter = "CustomerID = '" & Me.txtCustomerID.Text & "'"
    Me.grdDatosBIS.CaptionText = "Buscar ID cliente: " & Me.txtCustomerID.Text
    Me.grdDatosBIS.DataSource = oDataView

End Sub
```

Código fuente 594

En la Figura 371 vemos el resultado de una búsqueda, mostrado en uno de los DataGrid del formulario.

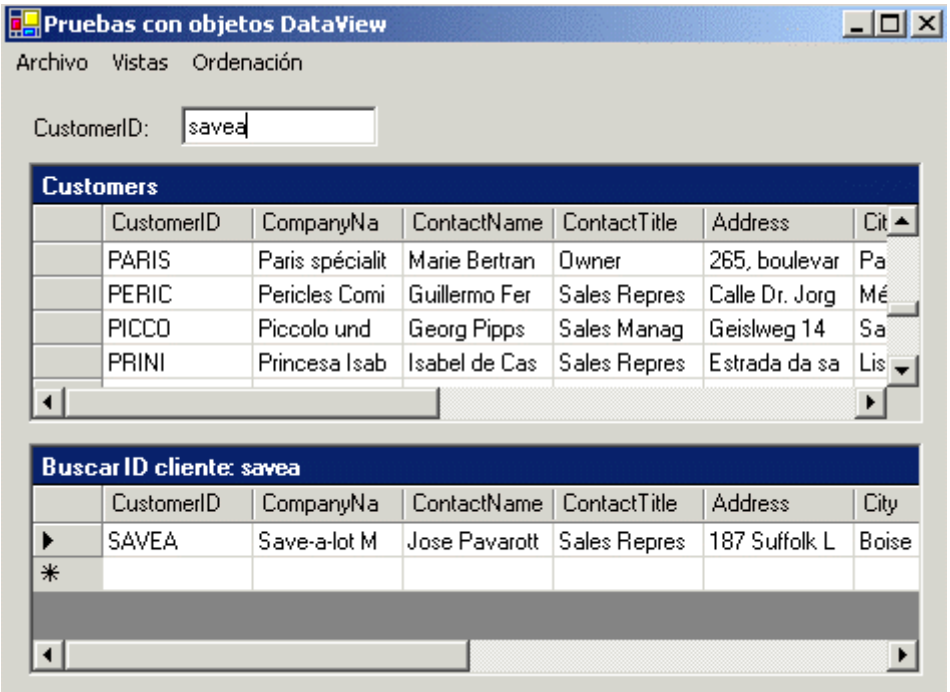


Figura 371. Búsqueda de una fila en una tabla de un DataSet, empleando un DataView.

Ordenación de filas mediante DataView

Para ordenar las filas en un DataView emplearemos su propiedad Sort, asignándole una cadena con el nombre de columna/s a ordenar, tal y como muestra el Código fuente 595, de la opción de menú *Ordenación + Normal*, en el formulario del ejemplo.

```
Private Sub mnuOrdNormal_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles mnuOrdNormal.Click

    ' crear dataview y ordenar las filas
    ' con la propiedad Sort
    Dim oDataView As New DataView()
    oDataView.Table = oDataSet.Tables("Customers")
    oDataView.Sort = "Country"
    Me.grdDatos.CaptionText = "Ordenar por campo Country"
    Me.grdDatos.DataSource = oDataView

End Sub
```

Código fuente 595

Veamos el resultado al ejecutar en la Figura 372.

Si necesitamos ordenar por múltiples columnas de la tabla, sólo tenemos que asignar a Sort una cadena con la lista de columnas requeridas. Ver Código fuente 596.

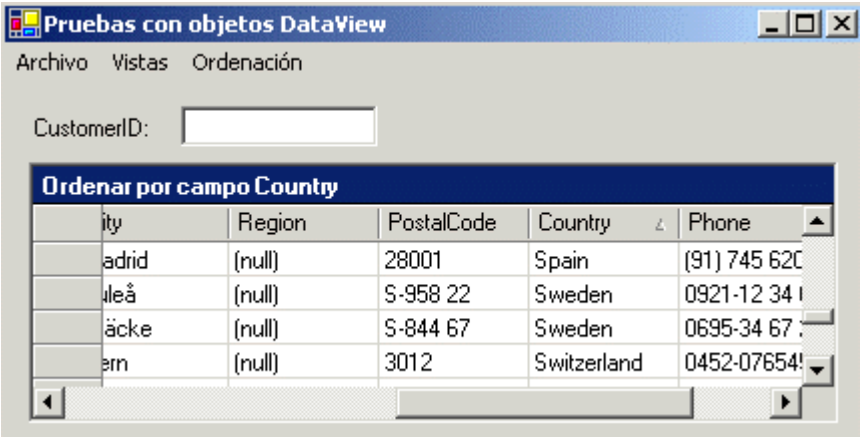


Figura 372. DataView ordenando las filas por la columna Country.

```
oDataView.Sort = "Country, PostalCode"
```

Código fuente 596

También es factible asignar a un DataView una combinación de filtro y ordenación, utilizando en la misma operación las propiedades RowFilter y Sort. El menú del formulario *Ordenación + Con filtro* realiza este trabajo, que vemos en el Código fuente 597.

```
Private Sub mnuOrdenFiltro_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles mnuOrdenFiltro.Click

    Dim oDataView As New DataView()
    oDataView.Table = oDataSet.Tables("Customers")
    ' establecer un filtro al dataview
    oDataView.RowFilter = "Country='USA'"
    ' ordenar las filas del filtro
    oDataView.Sort = "City"
    Me.grdDatos.CaptionText = "Filtrar por USA. Ordenar por campo City"
    Me.grdDatos.DataSource = oDataView

End Sub
```

Código fuente 597

Los datos con el filtro y orden podemos verlos en el DataGrid del formulario, que muestra la Figura 373.

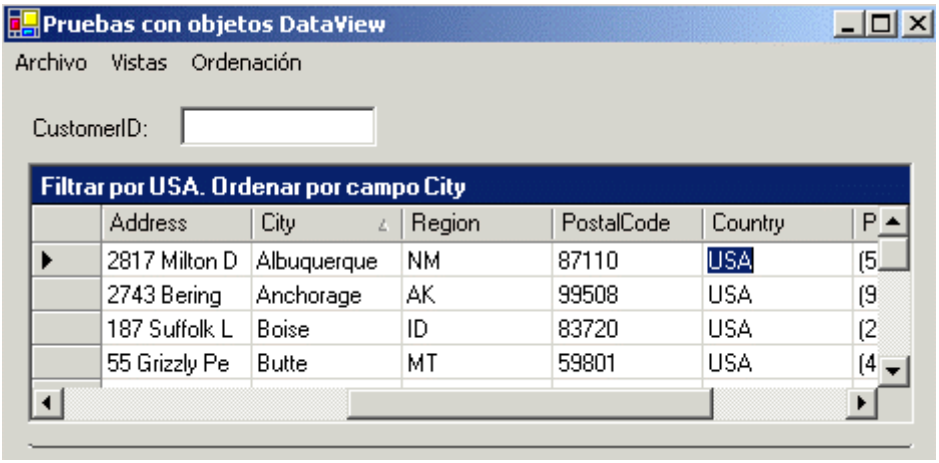


Figura 373. Resultado de DataView con filtro y orden.

Obtener el esquema de un DataSet

El esquema de un DataSet consiste en toda la información contenida por este objeto, acerca de los nombres de tablas, columnas, relaciones, etc.; es decir, se trata de metainformación sobre los datos que contiene el DataSet.

Podemos obtener estos metadatos del DataSet recorriendo la colección que nos interese en cada caso: Tables, Columns, etc.

El Código fuente 598 muestra como tras crear un DataSet, recorremos sus tablas, y dentro de estas, sus columnas, mostrando la información obtenida en un ListBox. Este ejemplo, EsquemaDatos, puede obtenerse haciendo clic aquí.

```
Private Sub btnEsquema_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnEsquema.Click

    ' crear conexión
    Dim oConexion As New SqlConnection()
    oConexion.ConnectionString = "Server=(local);" & _
        "Database=Northwind;uid=sa;pwd=;"

    ' crear dataset
    Dim oDataSet As New DataSet()

    ' crear adaptadores de datos para las tablas
    ' y añadir cada tabla al dataset con el adaptador
    Dim oDataAdapter As SqlDataAdapter

    oDataAdapter = New SqlDataAdapter("SELECT * FROM Customers", oConexion)
    oDataAdapter.Fill(oDataSet, "Customers")
    oDataAdapter = Nothing

    oDataAdapter = New SqlDataAdapter("SELECT * FROM Orders", oConexion)
    oDataAdapter.Fill(oDataSet, "Orders")
    oDataAdapter = Nothing

    oDataAdapter = New SqlDataAdapter("SELECT * FROM Products", oConexion)
    oDataAdapter.Fill(oDataSet, "Products")
    oDataAdapter = Nothing

    oDataAdapter = New SqlDataAdapter("SELECT * FROM Territories", oConexion)
```

```
    oDataAdapter.Fill(oDataSet, "Territories")
    oDataAdapter = Nothing

    ' crear un objeto tabla y columna para mostrar
    ' la información del esquema que el dataset contiene
    Dim oDataTable As DataTable
    Dim oDataColumn As DataColumn

    Me.lstEsquema.Items.Add("Estructura del DataSet")
    ' recorrer la colección de tablas del DataSet
    For Each oDataTable In oDataSet.Tables
        Me.lstEsquema.Items.Add("Tabla: " & oDataTable.TableName)

        ' recorrer la colección de columnas de la tabla
        For Each oDataColumn In oDataTable.Columns
            Me.lstEsquema.Items.Add("Campo: " & _
            oDataColumn.ColumnName & " --- " & _
            "Tipo: " & oDataColumn.DataType.Name)
        Next
    Next

End Sub
```

Código fuente 598

La Figura 374 muestra el ListBox relleno con el esquema del DataSet tras haber pulsado el botón del formulario.

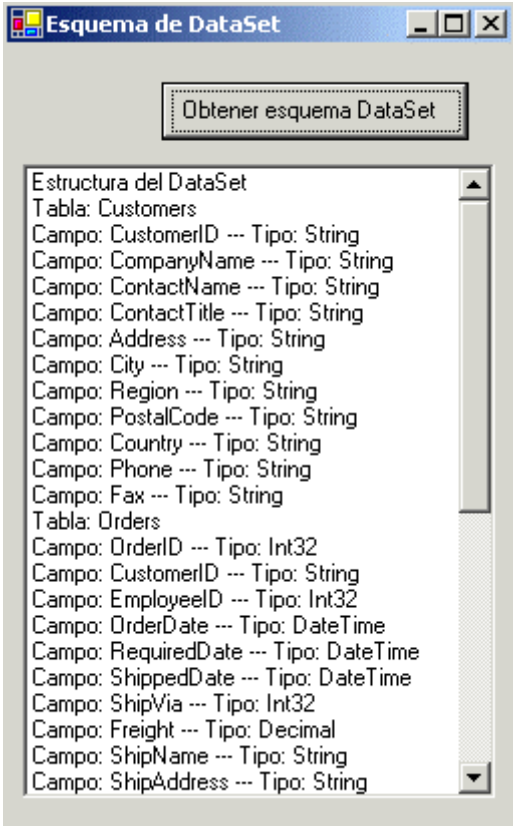


Figura 374. Obtención del esquema de un DataSet.

LaLibreríaDigital.com Libros en formato digital para profesionales de la informática

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 reviata ***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.

© Grupo EIDOS
http://www.eidos.es