El lenguaje de programación C#

Autor: José Antonio González Seco

Curso disponible en DOC y PDF en la página web del autor.

Indice de contenidos

- Bibliografía
- Información en Internet sobre C#
- Portales
- Grupos de noticias y listas de correo

Leer comentarios (36) | Escribir comentario | Puntuación: ■■■■■ (53 votos) Vota

Últimos comentarios [Subir]

esta muy interesante (19/10/2002)
Por jose antonio

como descargo el curso completo de c#, y si tienes curso de java visual 6 informame, pues te agradesco que me contestes pues estoy aprendiendo poco a poco..... Gracias

Felicitaciones (16/10/2002)
Por Enrique López

Me parece un excelente curso, y me gustaría continuar estudiandolo, pero se me dificulta en línea, así que por favor podría tener una copia como documento electrónico,

Muchísimas gracias

como bajar este manual (14/10/2002)
Por fabian rojas

me justaria tener este curso en mi pc
te agradeco si me dices como lo puedo hacer

¿Como descargarlo? (13/10/2002)
Por Araceli

te agradeceria si pudieras decirme como descargar el archivo para poder tenerlo en mi computadora sin tener conexion a internet

gracias

Excelente (08/10/2002)
Por PABLO SOUR

Ojala me pudieras mandar una copia por correo electronico Gracias

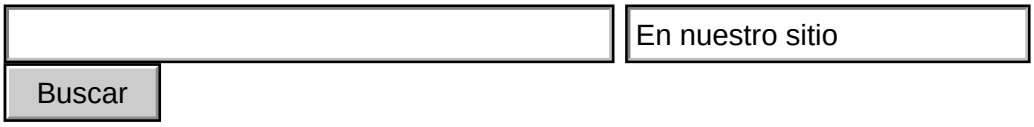
Recomendar este tutorial | Estadísticas

Principio Página

programacion en castellano

Ganamos el Premio iBest 2001

» BUSCADOR



Se pueden buscar frases completas encerrandolas entre comillas dobles (") y usar los operadores lógicos AND, OR y NOT. Por defecto, buscar más de una palabra delimitadas por espacios equivale a utilizar el operador AND.

» DIRECCIONES

Bases de datos
SQL, PostgreSQL...

Entornos de Desarrollo
Delphi, Visual Basic...

Entretenimiento
Juegos, Demos...

Herramientas
Editores, Compiladores...

Internet
HTML, XML, WAP...

Lenguajes imperativos
C, Pascal...

Leng. orientados a objeto
C++, Java...

Lenguajes de *script*
Perl, JavaScript...

Otros lenguajes
Prolog, Haskell...

Sistemas operativos
Windows NT, Linux...

Teoría
Metodología, UML, Algoritmos...

Varios
Generales, Gráfica...

Añadir URL | Últimos enlaces

» NUESTROS CURSOS

Últimos cursos publicados:

- New 2 Java: Construir una Aplicación: 4.- Leer y Escribir Ficheros y Manejar Excepciones [Zona Java]. NUEVO
- Operaciones avanzadas con JDBC y Java [Zona Java]. NUEVO
- Modelo relacional [Zona General]. NUEVO
- El lenguaje de programacion C# [Zona General]. NUEVO
- Curso de Prolog avanzado [Zona General].
- Manejar Conexiones a Bases de Datos con JDBC 3.0 [Zona Java]
- Curso de XML [Zona General].
- Suplementos a New 2 Java [Zona Java].
- Curso práctico de Corba en GNU/Linux [Zona General].
- XML Schema y DTDs [Zona General].

Visita nuestra
Nueva Zona sobre
PHP

» PDF ARTICULOS

Todos los artículos del 2000 en **PDF**.

» FORMACIÓN

» NUESTRAS ZONAS

HTML en castellano
HTML, XML, Javascript, DHTML..

Java en castellano
Información Java

ASP en castellano
ASP, VBScript, etc.

PHP en castellano
PHP, MySQL, etc.

» SECCIONES

Cursos
Artículos
Formación
Foros

» HEMOS GANADO

» NOVEDADES

Más cursos: Todos | Zona HTML | Zona Java | Zona ASP | Zona PHP

» NUESTROS ARTÍCULOS

Últimos artículos publicados:

- SMTP utilizando Sockets en PHP [Zona PHP]. NUEVO
- Desplegar Servlets y Aplicaciones Web en Tomcat y WebLogic Server [Zona Java]. NUEVO
- PHP y funciones FTP [Zona PHP]. NUEVO
- Crear un fichero robots.txt [Zona HTML]. NUEVO
- Trabajar con ficheros en PHP [Zona PHP]. NUEVO
- El Proyecto ASP.NET Web Matrix [Zona General].
- Protección con contraseñas (III): Varios usuarios [Zona HTML].
- JDC Tech Tips 22 de Enero de 2002 [Zona Java].
- JDC Tech Tips 10 de Enero de 2002 [Zona Java].

Más artículos:
Todos | Zona HTML | Zona Java | Zona ASP | Zona PHP

Participa
Sea colaborador de Programación en castellano. Estaremos encantados de publicar sus cursos y articulos sobre programación. Escribenos.

» FOROS

Algunos de nuestros foros:

- Visual Basic NUEVO
- Visual FoxPro NUEVO
- HTML
- Java (básico)
- PHP
- XML
- Bases de datos y SQL
- ASP
- Serv. de Aplicaciones J2EE

Más foros

Ultimos mensajes:

17 - Octubre 2002

Nuevo artículo en **zona PHP**: SMTP utilizando Sockets en PHP.

13 - Octubre 2002

Nuevo artículo en **zona Java**: Desplegar Servlets y Aplicaciones Web en Tomcat y WebLogic Server.

10 - Octubre 2002

Nuevo curso en **zona Java**: New 2 Java: Construir una Aplicación: 4.- Leer y Escribir Ficheros y Manejar Excepciones.

9 - Octubre 2002

Nuevo artículo en **zona PHP**: PHP y funciones FTP.

6 - Octubre 2002

Nuevo curso en **zona Java**: Operaciones avanzadas con JDBC y Java.

3 - Octubre 2002

Segunda y última entrega del curso: Modelo relacional.

24 - Septiembre 2002

Primera entrega del curso: Modelo relacional.

16 - Septiembre 2002

Nuevo artículo en zona HTML: Crear un fichero robots.txt.

13 - Septiembre 2002

Nueva zona temática en Programación en castellano: ZONA PHP.

9 - Septiembre 2002

Nuevo artículo: Trabajar con ficheros en PHP.

6 - Septiembre 2002

» PROGRAMACIÓN EN CASA

Acceso a descarga de archivos PDF
Información sobre programación (novedades, direcciones, artículos, etc.) tranquilamente en el correo de tu casa.

Nombre:

Password:

Enviar

Si ya pertenecías a la lista de distribución de novedades de Programación en castellano, se te ha asignado como **nombre y password tu dirección de correo electrónico**
Por el contrario, si es aún no estas registrado haz clic aquí para darte de alta

Séptima y última entrega del curso: El lenguaje de programación C#.

» RECOMENDADOS

La Variable
Maestros del Web
WebExperto

Ganamos el Premio iBest 2001

Otras Webs: sms gratis, Gratis, Directorio y Buscador, Logos y melodias moviles, WAPes

Contacto - Datos legales
Estadísticas en:

ReD Internet: Melodias Moviles, Logos Nokia | envio sms gratis | Salvapantallas y fondos | **Melodias ericsson** | melodias moviles gratis | logos motorola

Añadir una dirección

Desde aquí puede usted colaborar con Programación en castellano añadiendo tantas direcciones de sitios sobre programación como usted desee. Un miembro de Programación en castellano las visitará y puntuará antes de añadirlas a nuestras páginas. Por favor, rellene el siguiente formulario:

Nombre de la página:

Dirección de la página:
http://

Su nombre:

Su dirección de correo electrónico:

Categoría donde se inscribe:
Ninguna

Comentarios sobre la página:

Enviar (puede tardar) Borrar

¡Gracias por colaborar!

Principio Página

Contacto - Datos legales

Crear una web de Programación en castellano

Tanto si ya tiene una web sobre programación y desea incorporarla a nuestro sitio, como si quiere crear una desde la nada, para **colaborar** con nosotros por favor rellene el siguiente formulario:

Nombre de la página:

X en castellano

Su nombre:

Su dirección de correo electrónico:

¿Cuál es el estado actual de su página?

La página no existe

Si está en el último caso, por favor indiquenos la dirección actual de su página:

http://

Comente la página que desea crear

Enviar (puede tardar) Borrar

¡Gracias por colaborar!

Crear un curso de programación

Si desea enseñar programación, la manera más sencilla es **colaborar** con nosotros creando un curso para Programación en castellano. Si ese es su deseo, por favor rellene el siguiente formulario:

Tema del curso:

Su nombre:

Su dirección de correo electrónico:

¿Cuál es el estado actual de su curso?

El curso no existe

Si está en el último caso, por favor indiquenos la dirección actual de su curso:

http://

Comente el curso que desea crear

Enviar (puede tardar) Borrar

¡Gracias por colaborar!

Direcciones.

Nuestras zonas

Bases de datos

40 direcciones

Recopilación de direcciones relacionadas con las bases de datos y su programación; lenguajes, herramientas, etc...

- Access (3)
- Caché (4)
- Microsoft SQL Server (2)
- MySQL (6)
- Oracle (4)
- PostgreSQL (5)
- SQL (4)
- Sybase (8)
- Visual FoxPro (4)
- Clipper & xBase@ (9)
- Teoría de bases de datos@ (1)

Principio Página

Direcciones. Bases de datos. SQL

4 direcciones

Recopilación de direcciones sobre el lenguaje estándar para el tratamiento de bases de datos SQL.

Nuestras zonas

»EN ESTA PAGINA

Cursos

4 direcciones

[Subir]

Curso de SQL. de aulaClic

http://www.aulaclic.org/sql/f_sql.htm

21914 visitas | Puntuación: ■■■■ (28 votos) Vota

Aprende SQL con este curso realizado con rigor pero con un leguaje claro y sencillo. Con ejemplos. Utiliza como motor de base de datos el Access 2000.

Manual de introducción SQL

http://www.unav.es/cti/manuales/Intro_SQL/indice.html

16817 visitas | Puntuación: ■■■■ (14 votos) Vota

Manual de SQL organizado en 11 lecciones. Forma parte del Centro de Tecnología Informática de la Universidad de Navarra.

Manuales de SQL y bases de datos

http://www.lobocom.es/~claudio/menu.htm

36877 visitas | Puntuación: ■■■■ (111 votos) Vota

Un excelente manual en castellano de SQL (orientado a Access), creado por Claudio Casares, que ha sido completado con cursos más teóricos sobre modelado de datos, modelo E/R, data warehousing, modelo relacional, etc..

Interactive/On-line SQL Tutorial

http://www.sqlcourse.com

11966 visitas | Puntuación: ■■■■ (11 votos) Vota

Excelente introducción al SQL con la posibilidad de probar tus recién adquiridas habilidades con un intérprete de SQL incluido en la web. Podrás teclear tus sentencias mientras estés conectado al curso.

Principio Página

Contacto - Datos legales

Direcciones. Bases de datos. PostgreSQL

Nuestras zonas

5 direcciones

Recopilación de direcciones interesantes sobre el servidor de base de datos gratuito (licencia BSD) PostgreSQL.

»EN ESTA PAGINA

Herramientas

1 dirección

[Subir]

PgAccess

http://flex.ro/pgaccess/

6735 visitas | Puntuación: ■■■■ (6 votos) Vota

Herramienta gráfica para administrar PostgreSQL

Articulos

1 dirección

[Subir]

Instalación de un servidor PostgreSQL

http://www.planetalinux.com.ar/article.php?aid=32

7218 visitas | Puntuación: ■■■ (3 votos) Vota

Artículo en castellano en el que se explica como instalar la base de datos PostgreSQL.

Generales
3 direcciones

[Subir]

PostgreSQL México
http://www.postgres.org.mx/index.html

10915 visitas | Puntuación: ■■■■ (3 votos) Vota

Página con variada información sobre este sistema de bases de datos. En castellano y mantenida por el mexicano Roberto Andrade. Dispone de una lista de correo.

Proyecto de traducción al Español de la documentación de PostgreSQL RDBMS
http://users.servicios.retecal.es/rsantos/index.htm

8272 visitas | Puntuación: ■■■■ (5 votos) Vota

proyecto de traducción al español de la documentación de Postgresql. Se anima a participar.

Página oficial del PostgreSQL
http://www.postgresql.org/

6146 visitas | Puntuación: ■■■■ (6 votos) Vota

Pues eso, la Web oficial de este sistema de bases de datos.

Principio Página

Direcciones.

Lenguajes orientados a objeto

43 direcciones

Recopilación de direcciones sobre lenguajes orientados a objeto, como Java, C++ o Smalltalk.

»EN ESTA PAGINA

Generales

1 dirección

[Subir]

La página orientada a objetos

http://www.ctv.es/USERS/pagullo/

19492 visitas | Puntuación: ■■■■ (45 votos) Vota

Página con información de diversos lenguajes orientados a objeto, como C++ y Java. Contiene enlaces y artículos.

Principio Página

Nuestras zonas

Direcciones. Lenguajes orientados a objeto.

C++

6 direcciones

Recopilación de direcciones sobre C++

»EN ESTA PAGINA

»IMPRESCINDIBLE VISITAR

En castellano

- C ++ con clase

Librerías

1 dirección

[Subir]

MinGW: Minimalist GNU For Windows

http://www.mingw.org

5832 visitas | Puntuación: ■■■■ (10 votos) Vota

Librerías que permiten desarrollar aplicaciones nativas Windows con el conocido compilador gcc. Son sólo librerías, así que es necesario tener el compilador, y no dispone de entorno de desarrollo.

Tutoriales
4 direcciones

[Subir]

C ++ con clase
http://c.conclase.net

10110 visitas | Puntuación: ■■■■■ (26 votos) Vota

Excelente y completo curso de C++, disponible también en PDF.

Introducción a la Programación con C++
http://www.geocities.com/inf135/tutc/tutc.htm

6091 visitas | Puntuación: ■■■ (9 votos) Vota

Tutorial de C++ en el que la teoría y la metodología tienen bastante peso. Muy adecuado, por tanto, para principiantes que no sepan programar ni en C ni en ninguna otra cosa.

Curso de C++
http://www.zator.com/Cpp/

21372 visitas | Puntuación: ■■■ (23 votos) Vota

Interesante curso sobre C++ en el que se hace un repaso de los principios básicos de este lenguaje de programación, sin entrar en los aspectos más complicados del mismo.

Aprenda C++ como si estuviera en primero
http://fcapra.ceit.es/AyudaInf/AprendaInf/Cpp/manualcpp.pdf

61412 visitas | Puntuación: ■■■■ (86 votos) Vota

Completo curso en formato PDF que te ayudará a aprender C++, aunque para seguirlo es recomendable conocer de antemano C y fundamentos de programación.

Generales
1 dirección

[Subir]

El rincon de Chemanuel
http://www.geocities.com/josempadron/esp/introduccion.htm

11847 visitas | Puntuación: ■■■ (10 votos) Vota

Paginas personales donde se recogen gran cantidad de enlaces a sitios dedicados a la programacion para Windows utilizando C++.

Principio Página

Nuestras zonas

Direcciones. Lenguajes orientados a objeto.

Java

27 direcciones

Java 2 Standard Edition -J2SE, es el entorno de desarrollo de aplicaciones Java orientado a las aplicaciones solitarias y los Applets

- J2ME (2)

» EN ESTA PAGINA

» IMPRESCINDIBLE VISITAR

En castellano

- Apple WebObjects 5.1
- Introducción a iPlanet Application Server

En otros idiomas

- Borland Jbuilder
- Sun
- Forte for Java

Entornos de desarrollo

6 direcciones

[Subir]

Apple WebObjects 5.1

http://www.apple.com/es/webobjects

3076 visitas | Puntuación: ■■■■■ (8 votos) Vota

Incluye tanto un entorno de desarrollo basado en Java como un servidor de aplicaciones para la web. Es de pago, eso sí. Lo utilizan más de 3.000 empresas como Adobe, Apple, BBC, Univ. Michigan, etc.

NetBeans

http://www.netbeans.org

2649 visitas | Puntuación: ■■■ (4 votos) Vota

Un gran entorno de desarrollo para Java, hecho en Java y open-source (bajo licencia SPL). El Forte está basado en parte en él.

Forte for Java

http://www.sun.com/forte/ffj/

3917 visitas | Puntuación: ■■■■■ (9 votos) Vota

Entorno de desarrollo para Java desarrollado por Sun en Java. Está basado en NetBeans. Existen dos variantes: "Community Edition" que es gratuita, e "Internet Edition" que es paga.

Borland Jbuilder

http://www.borland.com/jbuilder

10462 visitas | Puntuación: ■■■■■ (25 votos) Vota

Para muchos la mejor herramienta RAD para Java del momento. Es una de las más potentes, de uso sencillo, con soporte 100% puro Java y compatible con las últimas tecnologías y soporte para bases de datos: CORBA, ODBC, JDBC, SQL, etc. El entorno de programación es similar al utilizado por Delphi y Borland C++ Builder, por lo cual los que hayan utilizado esos programas se aconstumbraran fácilmente a JBUilder.

IBM Visual Age for Java

http://www.software.ibm.com/ad/vajava/

8063 visitas | Puntuación: ■■■■ (23 votos) Vota

Herramienta RAD de IBM, disponible para OS/2 y Windows, es uno de los entornos más visuales e intuitivos.

Kawa

http://tek-tools/kawa

19840 visitas | Puntuación: ■■■■ (18 votos) Vota

Aplicación shareware Win32, es un estupendo IDE, potente y de fácil uso. Entorno de varias ventanas, en una de ellas muestra en arbol los paquetes Java, su clases, métodos y variables correspondientes. Posibilidad de establecer diversos modos de compilación, soporte para JDBC, resalte de color y muchas cosas más. Una buena y barata alternativa a los entornos más avanzados.

Generales
8 direcciones

[Subir]

javaHispano
http://www.javahispano.com

13391 visitas | Puntuación: ■■■■ (10 votos) Vota

Pagina con TODO lo relacionado con Java. Tutoriales, artículos, Faqs, noticias, enlaces. Y todo en castellano.

Distrito de Java en Telepolis
http://www.telepolis.com/cgi-bin/t30/!DISTRITOSEC?distrito=Java

10278 visitas | Puntuación: ■■■■ (11 votos) Vota

Distrito de Telepolis dedicado a Java. Con manuales, noticias, ejemplos, cursos y un foro para plantear tus dudas.

Java en BIT
http://www.bit-net.org/java/default.htm

6848 visitas | Puntuación: ■■■■ (4 votos) Vota

Interesante recopilación de enlaces relacionados con Java Muy interesante el minitutorial totalmente práctico que sobre este lenguaje.

Java en la UNAM (Universidad Nacional Autonoma de Mexico)
http://sunsite.unam.mx/java.htm

9275 visitas | Puntuación: ■■■■ (6 votos) Vota

Compendio de información sobre Java. Artículos, direcciones etc.

Cafe au Lait Java
http://metalab.unc.edu/javafaq/

4485 visitas | Puntuación: ■■■■ (1 voto) Vota

Web mantenida por Elliote Rusty Harold con tutoriales, cursor, libros, noticias sobre Java actualizadas diariamente, descripción de APIs, ...

Gamelan
http://www.gamelan.com/

6012 visitas | Puntuación: ■■■■ (3 votos) Vota

La principal fuente de recursos para Java. Todo está ordenado en una gran cantidad de directorios, para poder encontrar fácilmente lo que se busca

JCentral
http://www.ibm.com/developer/java/

4309 visitas | Puntuación: ■■■■ (1 voto) Vota

Sitio central de Java impulsado por IBM, donde se pueden encontrar noticias, consejos para desarrolladores, información sobre aplicaciones y nuevas herramientas, etc

Sun
http://java.sun.com/

5802 visitas | Puntuación: ■■■■■ (8 votos) Vota

La Web de los creadores del Java.

JavaBeans
1 dirección

[Subir]

JavaBeans (JavaSoft)
http://java.sun.com/beans

6315 visitas | Puntuación: ■■ (5 votos) Vota

Desde esta página podréis acceder a todo tipo de material sobre JavaBeans, proporcionado gratuitamente por Sun: API's, documentación, tutoriales, últimas tecnologias relacionadas y herramientas en sus últimas versiones. De visita obligada.

Revistas
1 dirección

[Subir]

JavaWorld
http://www.javaworld.com/

6625 visitas | Puntuación: ■■■■ (7 votos) Vota

Revista electrónica publicada por IDG dedicada a Java. Contiene artículos de muy buena calidad, una sección enorme sobre programas disponibles para Java, revisión de libros, ...

Tutoriales
9 direcciones

[Subir]

Jini - Sistemas distribuidos en Java
http://ciberia.ya.com/pxai/filez/jini.zip

3879 visitas | Puntuación: ■■■ (2 votos) Vota

Jini (Jini Is Not Initials) es una tecnologia de Sun Microsystems para el desarrollo de sistemas distribuidos formados por todo tipo de dispositivos que contengan la maquina virtual Java. Este es un documento que repasa los principios fundamentales de los sistemas distribuidos y describe la tecnologia con ejemplos. En castellano

Struts - Implementacion del patron MVC en Web
http://ciberia.ya.com/pxai/filez/struts.zip

3129 visitas | Puntuación: ■■ (4 votos) Vota

Struts es una plataforma que permite desarrollar aplicaciones web en Java basandonos en el patrón de diseño Model-View-Controller. En este Documento se describe ese patrón y se dan algunos ejemplos de Struts.

Introducción a iPlanet Application Server
http://java.programacion.net/ipintro/

3389 visitas | Puntuación: ■■■■■ (2 votos) Vota

Traducción al castellano del manual de introducción al iPlanet Application Server.

http://java.programacion.net/beaintro/

http://java.programacion.net/beaintro/

6477 visitas | Puntuación: ■■■ (3 votos) Vota

Primero de una serie de tutoriales dedicados a los Servidores de Aplicaciones especialmente enfocados a Java como WebLogic de BEA. A partir de esta serie aprenderemos como publicar nuestras aplicaciones Java en los servidores más importantes.

Tutorial de Cocoon

http://java.programacion.net/cocoon/

6848 visitas | Puntuación: ■■■■ (6 votos) Vota

Cocoon es un sistema de publicación electrónico basado en XML/XSL orientado a documentos.Es 100% Java y está basado en estándares. Además es probablemente el framework de este tipo más maduro y reconocido. En este tutorial se explica como funciona y como podemos utilizarlos para desarrollar aplicaciones Web que presenten la información en HTML, PDF, WML, etc.

Introducción a JSP

http://www.verextremadura.com/miguel/jsp

7661 visitas | Puntuación: ■■■■ (14 votos) Vota

Breve introducción a la programación en internet con JSP y servlets.

Pequeño tutor de Java

http://usuarios.tripod.es/Ozito/index.html

21370 visitas | Puntuación: ■■■■ (36 votos) Vota

Pequeño tutor de Java en la que se estudian de forma muy didactica y comprensible diferentes aspectos de este lenguaje: swing, trabajo en red, fichero Jar, métodos nativos, etc..

Tutorial de Java de Agustin Froufe

http://members.es.tripod.de/froufe/

15151 visitas | Puntuación: ■■■■ (13 votos) Vota

Estupendo tutorial de sobre Java 1.2 de Agustin Froufe.

Thinking in Java

http://www.eckelobjects.com/TIJ2/index.html

13440 visitas | Puntuación: ■■■■ (14 votos) Vota

Un libro excelente (¡Muchos afirman que el mejor!) acerca de Java. El mismo se puede descargar en forma totalmente gratuita.

Principio Página

Direcciones.

Entornos de desarrollo

33 direcciones

Recopilación de direcciones sobre distintos entornos de programación, herramientas de desarrollo rápido de aplicaciones (RAD), etc..

- .NET (2)
- C++ Builder (1)
- Delphi (15)
- Otros (1)
- Power Builder (2)
- Velázquez Visual (1)
- Visual Basic (10)
- Visual C++ (1)

Principio Página

Direcciones. Entornos de desarrollo. Delphi

15 direcciones

Delphi es el entorno de desarrollo de Borland para la programación bajo Windows 95/98/NT, basado en una variación del lenguaje Pascal llamada Object Pascal que convierte este lenguaje en uno orientado a objetos. La mayoría de estas direcciones son igualmente válidas para su herramienta hermana, C++ Builder.

Nuestras zonas

»EN ESTA PAGINA

»IMPRESCINDIBLE VISITAR

En castellano

- Club Delphi

En otros idiomas

- Torry's Delphi Pages

Repositorios de componentes

4 direcciones

[Subir]

Delphi Free Stuff

http://www.delphifreestuff.com

8684 visitas | Puntuación: ■■■ (5 votos) Vota

Esta es la página personal de Brad Stowers, uno de los mejores programadores dedicados a la programación de componentes gratuitos para Delphi. Contiene sus componentes, los de algunos colaboradores y algunos programas gratuitos creados por ellos.

Delphi Super Page

http://delphi.icm.edu.pl

8331 visitas | Puntuación: ■■■■ (13 votos) Vota

Infinitos componentes gratuitos y shareware, muchos de ellos con código fuente. Además la web está perfectamente estructurada de modo que puedas encontrar fácilmente lo que necesitas y cuenta con diversos mirrors que aceleran el proceso de buscar y bajarse componentes.

Jordan Russell's Software Page

http://www.jordanr.dhs.org/

4532 visitas | Puntuación: ■■■■ (5 votos) Vota

Página del programador Jordan Russell, donde encontraréis su famoso componente, Toolbar'97 y sobre todo, el programa gratuito para realizar instalaciones Inno Setup.

Torry's Delphi Pages

http://www.torry.ru

4871 visitas | Puntuación: ■■■■■ (5 votos) Vota

Almacén de componentes, no tan extenso como Delphi Super Page pero igualmente bien clasificado. A veces es bueno mirar aquí para encontrar lo mejor de un determinado tipo de componente.

Cursos

5 direcciones

[Subir]

Curso de Delphi de TSP web

http://orbita.starmedia.com/~tspweb/programacion.htm

2306 visitas | Puntuación: ■■■ (2 votos) Vota

Una pequeña introducción a Delphi, en formato Word. Escrito para Delphi 3, puede que en algunas cosas se haya quedado obsoleto.

Ponga una dll (ISAPI) en su vida.

http://www.dtplan.com/garcia-cuervo/Delphi.htm

2134 visitas | Puntuación: ■■■■ (4 votos) Vota

Un pequeño tutorial que explica paso a paso como realizar extensiones web con Delphi 5.

Ejercicios Resueltos en Delphi

http://www.terra.es/personal/resfer/delphi

12803 visitas | Puntuación: ■■■■ (20 votos) Vota

Más de 40 ejercicios resueltos comenzando desde cero y pasando por tablas del tipo DBF, así como bases de datos de Access.

Cursillo de Delphi

http://www.arrakis.es/~eb1fts/primera.htm

11843 visitas | Puntuación: ■■■■ (12 votos) Vota

Curso para iniciarse en esto de la programación en Delphi.

Curso de Creación de Componentes en Delphi

http://personal.redestb.es/revueltaroche/ccind.htm

6940 visitas | Puntuación: ■■■■ (14 votos) Vota

Magnífico curso que amplía la información que se da en la mayor parte de las web y libros sobre la creación de componentes.

Generales
5 direcciones

[Subir]

Delphi Heaven
http://www.delphiheaven.com

6565 visitas | Puntuación: ■■■■ (4 votos) Vota

Interesante punto de partida para programadores en Delphi. Cursos sobre Delphi y Pascal, artículos, código fuente, componentes, enlaces e incluso una revista propia.

Borland España
http://www.borland.es

4892 visitas | Puntuación: ■■■■ (2 votos) Vota

Página en español de Borland, algo más vacía pero en nuestro idioma.

Club Delphi
http://www.clubdelphi.com

8530 visitas | Puntuación: ■■■■■ (32 votos) Vota

Este es el punto de encuentro de los programadores de Delphi en castellano. Trucos, enlaces, componentes,...

Borland
http://www.borland.com

3855 visitas | Puntuación: ■■■■ (4 votos) Vota

Página de Borland (o Inprise), creadora de Delphi y su hermano C++ Builder.

Project JEDI
http://www.delphi-jedi.org

4512 visitas | Puntuación: ■■■■ (3 votos) Vota

Hogar de diversos proyectos entre los que destacan la conversión de diversos APIs existentes en Windows para poder ser utilizados en Delphi, como DirectX y OpenGL.

Foros
1 dirección

[Subir]

Foro de Delphi en castellano
http://es.groups.yahoo.com/group/DelphiGroup

1969 visitas | Puntuación: ■■■■ (2 votos) Vota

Grupo de discusión de Delphi en castellano (accesible por web o email) para la ayuda mutua y el intercambio de información entre la comunidad programadora que utiliza este entorno, sea cual sea su nivel.

Principio Página

Contacto - Datos legales

Direcciones. Entornos de desarrollo. Visual Basic

10 direcciones

Recopilación de direcciones sobre Visual Basic

Nuestras zonas

»EN ESTA PAGINA

»IMPRESCINDIBLE VISITAR

En castellano

- La página del Visual Basic (el Guille)
- Codigo fuente
- Canal Visual Basic
- Visual Basic Experto

Artículos

1 dirección

[Subir]

VB - web de programación

http://www.terra.es/personal/jrcabrera

13781 visitas | Puntuación: ■■■■ (16 votos) Vota

Artículos principalmente centrados en aplicaciones distribuidas (DNA) en VB... MTS, COM, COM+, trucos, enlaces... Para los que ya dominan el Visual Basic.

Tutoriales
2 direcciones

[Subir]

Isla Programación
http://www.islaprogramacion.com

9577 visitas | Puntuación: ■■■■ (81 votos) Vota

Dispone de un curso para iniciarse en Visual Basic, desde variables y operadores hasta acceso a bases de datos. Además dispone de foro de consulta y enlaces a otras páginas.

Aprenda Visual Basic 6.0
http://fcapra.ceit.es/AyudaInf/AprendaInf/VisualBasic6/vbasic60.pdf

83366 visitas | Puntuación: ■■■■ (108 votos) Vota

Curso de Visual Basic 6.0 en formato PDF desde la Escuela Superior de Ingenieros Industriales de San Sebastián (Universidad de Navarra).

Código
2 direcciones

[Subir]

Codigo fuente
http://www.terra.es/personal2/sfortiz/

5356 visitas | Puntuación: ■■■■■ (11 votos) Vota

Algunos ejemplos con código fuente en VB, relacionados principalemente con base de datos.

Código Fuente de Visual Basic
http://it.internations.net/codigovb

11026 visitas | Puntuación: ■■■■ (17 votos) Vota

Rutinas, ejemplos y codigo fuente de Visual Basic.

Generales
5 direcciones

[Subir]

Canal Visual Basic
http://www.canalvisualbasic.net/

8166 visitas | Puntuación: ■■■■■ (8 votos) Vota

Una buena web con cursos, no sólo de Visual Basic, sino también de metodología, SQL Server, SQL, programación orientada a objetos, etc.. Tiene también foro, ejemplos con código fuente y trucos.

La página del Visual Basic (el Guille)
http://guille.costasol.net/

36999 visitas | Puntuación: ■■■■■ (130 votos) Vota

La mejor Web sobre Visual Basic en castellano. Con gran cantidad de información y actualizada de forma constante. Mantenida por el Guille. De visita indispensable.

La web de Oscar Grosso

http://www.geocities.com/vbgrosso/

14038 visitas | Puntuación: ■■■ (11 votos) Vota

Pequeña web con algunos trucos y diversos controles OCX sacados de Internet.

Visual Basic 6

http://www.visualb6.com/

37446 visitas | Puntuación: ■■■■ (25 votos) Vota

Interesante web argentina, con multitud de artículos diseñados para resolver dudas frecuentes.

Visual Basic Experto

http://www.eidos.es/VeXPERT/

32406 visitas | Puntuación: ■■■■■ (54 votos) Vota

Web con gran cantidad de artículos relacionadas con Visual Basic, organizados en diferentes apartados: Bases de datos, Generales, Componentes, Matemáticas e Ingenieria, etc. Mantenida por Harvey Triana.

Principio Página

Direcciones.

Nuestras zonas

Lenguajes de script

42 direcciones

Recopilación de direcciones sobre distintos lenguajes de script, o lenguajes de macro.

- Javascript (27)
- Perl (10)
- Python (5)

Principio Página

Direcciones. Lenguajes de script. Perl

10 direcciones

Recopilación de direcciones con documentacion (tutoriales, artículos, ejemplos, etc..) sobre Perl, el lenguaje de script para Unix más popular.

Nuestras zonas

»EN ESTA PAGINA

»IMPRESCINDIBLE VISITAR

En castellano

- Perl Chile
- Perl 5.0: Un lenguaje multiuso
- Centro de Recursos Perl

Archivos

1 dirección

[Subir]

Comprehensive Perl Archive Network (CPAN)

http://www.cpan.org

4401 visitas | Puntuación: ■■■■ (4 votos) Vota

Inmenso archivo (786 Mb en estos momentos), con información, herramientas, librerías, etc..

Generales
5 direcciones

[Subir]

IndicePERL
http://lmsaizarroba.tripod.com/IndicePERL.html

817 visitas | Puntuación: ■■■ (2 votos) Vota

Pequeña página donde se indica donde conseguir PERL y se dan algunos ejemplos de uso.

Centro de Recursos Perl
http://informatica.kipelhouse.com/perl.html

3548 visitas | Puntuación: ■■■■■ (20 votos) Vota

Es sólo una página (literalmente) pero llena de información. Incluye apuntes, cursos, manuales, bolsa de empleo, y un tablón de preguntas y respuestas.

Código Perl
http://www.tres.com/perl/

6988 visitas | Puntuación: ■■■■ (6 votos) Vota

Aceptable punto de partida para aquellos que deseen utilizar Perl para el desarrollo de CGIs.

Perl Chile
http://www.perl.cl

6648 visitas | Puntuación: ■■■■■ (5 votos) Vota

Excelente página sobre Perl, con noticias, artículos y un pequeño y tutorial de este lenguaje.

Página oficial de Perl
http://www.perl.com/pub

4762 visitas | Puntuación: ■■■ (2 votos) Vota

Aparte de poder obtener las últimas versiones de este lenguaje, contiene la famosa 'Perl Reference', donde podrás obtener toda clase de recursos relacionados con Perl.

Tutoriales
4 direcciones

[Subir]

Perl 5.0: Un lenguaje multiuso
http://www.iespana.es/perl-es

4302 visitas | Puntuación: ■■■■■ (13 votos) Vota

Completo tutorial que incluye temas como acceso a BBDD, CGI, XML, y programación orientada a objetos. Sin embargo, profundiza poco en cada tema, resultando bastante ardua en ocasiones la comprensión de los temas.

Tutorial de DBI
http://geneura.ugr.es/~javi/dbi/index.htm

5161 visitas | Puntuación: ■■■ (5 votos) Vota

Completísimo tutorial de DBI. DBI (Interfaz de Base de Datos-Data Base Interfaz-) es un módulo de Perl para acceso a bases de datos, es decir, mediante DBI podremos acceder a bases de datos con nuestros scripts en en Perl

Introducción al lenguaje Perl

http://www.uco.es/~i22oscav/tutor.html

8643 visitas | Puntuación: ■■■■ (4 votos) Vota

Antiguo tutorial contenido en una sola página. Está, pues eso, un poco antiguo, y pequeño.

Tutorial de Perl

http://www.granavenida.com/perl/

11061 visitas | Puntuación: ■■■■ (8 votos) Vota

Atractivo tutorial muy indicado para iniciarse en el estudio de Perl, ya que abarca desde el proceso de instalación hasta el uso de tuberías y expresiones regulares. Dispone de una versión comprimida para llevarselo a casa.

Principio Página

Direcciones. Lenguajes de script. Javascript

27 direcciones

Recopilación de direcciones con documentacion (tutoriales, artículos, ejemplos, etc..) sobre Javascript y JScript.

Nuestras zonas

»EN ESTA PAGINA

»IMPRESCINDIBLE VISITAR

En castellano

- /* El código */
- Taller Web
- Gamarod JavaScript
- Scriptonario de via-modem.com

En otros idiomas

- Manual de referencia de Netscape
- Doc Javascript

Direcciones de caracter general

3 direcciones

[Subir]

/* El código */

http://www.elcodigo.net/

20812 visitas | Puntuación: ■■■■■ (22 votos) Vota

Web dedicada principalmente a albergar tutoriales y trucos para Javascript, aunque posee también una sección para ayudar a los usuarios de Netscape. Un poco vacía aún, pero promete.

Doc Javascript

http://www.webreference.com/docjs

11621 visitas | Puntuación: ■■■■■ (3 votos) Vota

El mejor lugar para el experto, pues pone a tu disposición artículos de periodicidad quincenal y un truco nuevo cada día (pequeñito, eso sí). Unas páginas que querrías guardar enteras en tu disco duro.

Webmonkey programming: Javascript

http://www.hotwired.com/webmonkey/programming/javascript/

7431 visitas | Puntuación: ■■■ (3 votos) Vota

Dispone de un par de excelentes tutoriales y de un gran repositorio de rutinas en Javascript.

Rutinas y ejemplos en Javascript

9 direcciones

[Subir]

mapbDhtml

http://perso.wanadoo.es/mapintanel/mapbdhtml/es/index.html

11070 visitas | Puntuación: ■■■■ (19 votos) Vota

Página con scripts, que por ahora sólo incorpora un menú desplegable con soporte para múltiples menús por página, profundidad variable y su utilización con frames.

Gamarod JavaScript

http://www.gamarod.com.ar

20665 visitas | Puntuación: ■■■■■ (43 votos) Vota

Recursos para WebMasters, este sitio contiene más de 50 Rutinas Javascript, en castellano y Gratuitas, listas para usar en tu web. Además incluyen ejemplos y explicaciones para su correcto uso. Ideal para las personas que recien se inician en el diseño de páginas web.

Librerías de JavaScript

http://javascript.raulnd.com

21970 visitas | Puntuación: ■■■■ (27 votos) Vota

En esta página encontrarás librerías *.js para descargar y utilizar gratuitamente

JavaScript to the Bone!

http://jsbone.fadlan.com

9588 visitas | Puntuación: ■■■■ (9 votos) Vota

Ejemplos prácticos (publicados en la -extinta- revista Netmani@)

Biblioteca de CGI's y rutinas Javascript

http://www.webviva.com/biblioteca

14091 visitas | Puntuación: ■■■■ (16 votos) Vota

Un señor barbudo nos guiará por varias rutinas en Javascript separadas por grupos. No tiene demasiadas, pero la mayoría son recomendables.

Scriptonario de via-modem.com

http://www.via-modem.com/scriptionario/

12959 visitas | Puntuación: ■■■■■ (5 votos) Vota

Excelente recolección de scripts, bien organizados e interesantes. También disponde de rutinas DHTML y applets Java.

Taller Web

http://html.programacion.net/tweb.php

14086 visitas | Puntuación: ■■■■■ (8 votos) Vota

Nuestro taller web, pese a incluir artículos de otros temas, está enfocado principalmente a aportar rutinas y ejemplos prácticos en Javascript.

Ejemplos de JavaScript

http://www.js-examples.com/js/

4134 visitas | Puntuación: ■■■■ (2 votos) Vota

Un buen número de scripts. Sin embargo, no resulta demasiado cómoda, puesto que las categorías no estásn muy bien escogidas.

javascripts.com

http://www.javascripts.com

10683 visitas | Puntuación: ■■■■ (6 votos) Vota

Un enorme repositorio de trucos y pequeños programas en Javascript. A pesar de estar bien estructurado, su mayor problema es precisamente su tamaño: lo normal es que encuentres demasiados scripts iguales y en ocasiones lo que necesitas resulta difícil de hallar.

Especificaciones y referencias

5 direcciones

[Subir]

DOM de Microsoft Internet Explorer

http://msdn.microsoft.com/workshop/author/om/doc_object.asp

4734 visitas | Puntuación: ■■■■ (3 votos) Vota

En esta web está la guía del DOM (Document Object Model) del Explorer de Microsoft. Desafortunadamente no dispone de opción para bajartelo.

Especificación de componentes ECMAScript

http://www.ecma.ch/stand/ECMA-290.htm

4929 visitas | Puntuación: ■■■■ (2 votos) Vota

Ambos navegadores han intentado colocar soluciones propietarias al problema de los componentes Javascript. En junio se aprobó un estándar oficial ECMA que regula dichos componentes, pero aún no es soportado por ningún navegador.

Especificación ECMAScript

http://www.ecma.ch/stand/ECMA-262.htm

4951 visitas | Puntuación: ■■■ (3 votos) Vota

Esta especificación regula el lenguaje base de Javascript. Ambos navegadores son compatibles con dicha especificación. De todos modos, es difícil de leer, así que es mejor leer la referencia de JScript de Microsoft para saber qué soporta y qué no soporta ECMAScript.

Manual de referencia de Microsoft

http://msdn.microsoft.com/scripting/default.htm

5042 visitas | Puntuación: ■■ (3 votos) Vota

En esta web encontrarás el Manual de referencia de la versión 5.0 de JScript, el nombre que recibe el Javascript utilizado en los productos de Microsoft. Lamentablemente sólo fija su atención en el lenguaje base (común a ambos navegadores) y se olvida del DOM, o modelo de objetos del documento. Te lo puedes bajar en formato CHM.

Manual de referencia de Netscape

http://developer.netscape.com/docs/manuals/js/client/jsguide/index.htm

6249 visitas | Puntuación: ■■■■■ (4 votos) Vota

Manual de referencia de la versión 1.3 del Javascript utilizado por los navegadores de Netscape. Se puede bajar en formato ZIP y PDF y resulta muy completo e instructivo. Sus únicos problemas son estar en inglés y hablar sólo de los navegadores de Netscape.

Tutoriales

7 direcciones

[Subir]

Tutorial JavaScript

http://www.iespana.es/cnlasrozas/tutjs/index.html

21129 visitas | Puntuación: ■■■■ (19 votos) Vota

Tutorial de JavaScript que empieza desde cero y termina en la creación avanzada de objetos. Recomendado a principiantes y a los que quieran mejorar sus técnicas.

Tutorial de Javascript en Terra

http://www.terra.com.ar/canales/tecnologia/11/11823.html

9028 visitas | Puntuación: ■■■■ (7 votos) Vota

Tutorial aceptable aunque algo parco. Eso sí, está en español y dispone de ejemplos bastante apropiados para principiantes.

Curso de Javascript 1.2

http://html.programacion.net/js/titjs.html

19582 visitas | Puntuación: ■■■■ (17 votos) Vota

Introducción al Javascript que, aunque escrita en principio para ser utilizada por usuarios de Netscape, ha sido adaptada para funcionar con Explorer.

Introducción a Javascript, de Stephan Koch

http://200.25.9.3/enlaces/javascript/index.html

10649 visitas | Puntuación: ■■■ (4 votos) Vota

Traducción del que posiblemente sea el manual clásico de Javascript más famoso en Internet. Muchos aprendimos con la versión en inglés de este tutorial, cuyo principal problema es haberse quedado algo anticuado.

Javascript for the total non-programmer

http://www.webteacher.com/javatour/framehol.htm

4571 visitas | Puntuación: ■■■ (1 voto) Vota

Ignoro si los que no sepan programar sacarán tanto provecho de este curso como augura su nombre, pero sí que es una buena introducción a Javascript.

The Javascript Primers

http://www.htmlgoodies.com/primers/jsp

6783 visitas | Puntuación: ■■■■ (6 votos) Vota

Un manual que permite aprender Javascript en 30 lecciones. Muy bien estructurado, expone cada concepto con un ejemplo y un ejercicio para que el lector asiente sus conocimientos.

Voodoo's Intro to Javascript, de Stephan Koch

http://rummelplatz.uni-mannheim.de/~skoch/js/tutorial.htm

10230 visitas | Puntuación: ■■■■ (1 voto) Vota

El clásico se actualiza, aunque sólo en inglés. Llega a tratar temas de HTML dinámico en un lenguaje conciso y concreto. Te lo puedes bajar en HTML, PS y PDF.

Articulos

3 direcciones

[Subir]

Artículo en Maestros del Web sobre Internet Explorer 5

http://quik.guate.com/cvdhs/webmasters/ie5.shtml

4362 visitas | Puntuación: ■■ (2 votos) Vota

Interesante artículo sobre las extensiones realizadas al modelo de DHTML de Explorer en su versión 5.

Artículo de Danny Goodman sobre el soporte del W3C DOM en los navegadores de quinta generación.

http://developer.iplanet.com/viewsource/goodman_cross/goodman_cross.htm

4999 visitas | Puntuación: ■■■■ (2 votos) Vota

Aunque ataña principalmente al DHTML, el Document Object Model va a variar la manera en que accedemos a las propiedades del documento desde Javascript, así que conviene estar preparados para el mismo. Este artículo es una buena introducción.

Artículo en Inquiry de Shelley Powers sobre Javascript y Netscape 5.0

http://www.inquiry.com/techtips/dhtml_pro/10min/10min1299/10min1299.asp

3866 visitas | Puntuación: ■■ (1 voto) Vota

Artículo que nos explica cómo adaptar nuestras rutinas en Javascript para que funcionen correctamente en el futuro Netscape 5.

Principio Página

Direcciones. Entretenimiento

Nuestras zonas

19 direcciones

Enlaces a sitios relacionados con programación enfocada al ocio: juegos, demos, etc...

- Demos (9)
- Juegos (10)

Principio Página

Direcciones. Entretenimiento.

Nuestras zonas

Juegos

10 direcciones

Recopilación de direcciones sobre programación de videojuegos.

»EN ESTA PAGINA

»IMPRESCINDIBLE VISITAR

En otros idiomas

- The Game Programming Megasite

Cursos

2 direcciones

[Subir]

CPV (Curso de Programación de Videojuegos)

http://www.arrakis.es/~ppriego/softpaco/wincpv.htm

11709 visitas | Puntuación: ■■■■ (16 votos) Vota

Todo un clásico dentro de Fidonet, este curso ofrecía una excelente introducción al mundo de la programación de videojuegos bajo MS-DOS. Aunque antiguo, destaca su incontestable calidad y la inclusión del código fuente de cuatro juegos completos. Ahora está disponible completo (era shareware) pero la página oficial parece haber desaparecido.

Curso de DIV Games Studio

http://www.areaint.com/Curso_Div.htm

7989 visitas | Puntuación: ■■■■ (15 votos) Vota

Curso sobre el entorno de programación de videojuegos DIV Games Studio, que permite la creación de los mismos de una manera más rápida y sencilla, si bien es cierto que no resulta útil para programar profesionalmente.

Generales
7 direcciones

[Subir]

WaterShip Dreams
http://wsd2002.iespana.es/wsd2002/index3.htm

1761 visitas | Puntuación: ■■■■ (3 votos) Vota

Web con trucos y recursos destinada, principalmente, al desarrollo de juegos en DIV2, aunque también incluye recursos para entornos como Delphi o Dark Basic.

Artebinario
http://artebinario.cjb.net

2248 visitas | Puntuación: ■■■■ (4 votos) Vota

Página está orientada a las personas que programan videojuegos en múltiples plataformas. Incluye un curso de programación de videojuegos en Allegro y C++.

DX Lab
http://www.geocities.com/dxlab/

4520 visitas | Puntuación: ■■■■ (8 votos) Vota

Recursos y código fuente sobre programación de DirectX 8 bajo Delphi.

RE genial.com: Desarrolladores
http://www.regenial.com/gameprogramming/

6040 visitas | Puntuación: ■■■■ (5 votos) Vota

Pequeña web dedicada principalmente a los engine 3D de los juegos, con pequeños artículos que explican los primeros pasos en DirectX y OpenGL.

Game programming and graphics programming
http://gameprogrammer.com/

3838 visitas | Puntuación: ■■■■ (3 votos) Vota

Web con artículos sobre programación gráfica y de videojuegos. Desgraciadamente, son pocos los artículos y además algo antiguos.

The Game Programming Megasite
http://www.perplexed.com/GPMega/index.htm

8773 visitas | Puntuación: ■■■■■ (4 votos) Vota

Fabulosa web con artículos sobre programación de videojuegos, útiles, numerosos, y perfectamente ordenados. También desde aquí puedes bajarte el código fuente de juegos completos y de diversas librerías. Falla un poco en el tema del sonido, pero todo lo demás ralla la perfección.

X2FTP
ftp://x2ftp.oulu.fi/pub/msdos/programming/

42871 visitas | Puntuación: ■■■■ (2 votos) Vota

Este sitio de ftp pone a nuestra disposición incontables recursos para programar juegos bajo MS-DOS, siendo la elección de sistema operativo lo peor que tiene.

Revistas

1 dirección

[Subir]

Gamasutra

http://www.gamasutra.com

5519 visitas | Puntuación: ■■■ (6 votos) Vota

Publicación sobre programación de videojuegos. Escriben muchos programadores del, digamos, 'mundo real' y se actualiza cada semana.

Principio Página

Contacto - Datos legales

Direcciones. Entretenimiento.

Nuestras zonas

Demos

9 direcciones

Recopilación de direcciones sobre programación de demos, presentaciones gráficas y sonoras de gran espectacularidad generadas en tiempo real.

- Ensamblador@ (6)

»EN ESTA PAGINA

»IMPRESCINDIBLE VISITAR

En castellano

- Efectos gráficos y otros artículos de PC-Manía

En otros idiomas

- Hornet
- Hugi

Artículos

4 direcciones

[Subir]

Efectos gráficos y otros artículos de PC-Manía

http://www.hobbypress.com/PCMANIA/PC044/DE/pc044dedemos0000.html

7826 visitas | Puntuación: ■■■■■ (1 voto) Vota

Se puede acceder desde Internet a todos los artículos publicados en la revista PC-Manía sobre la programación de demos y escritos por Miquel Barceló y Eduard Sánchez Palazón. Son muchos, perfectamente explicados y con código de ejemplo, pero desafortunadamente no existen un índice que permita acceder a todos ellos, por lo que deberemos ir pulsando a los enlaces de 'Siguiente' para leerlos todos.

Página de Null

http://pagina.de/null

3487 visitas | Puntuación: ■■■ (2 votos) Vota

En este página encontrarás una sección de tutoriales por ahora limitada a un buen artículo sobre el efecto Blobs.

Awesome Fire Tut
http://backlit.digitis.net/awesome.htm

12195 visitas | Puntuación: ■ (1 voto) Vota

Buen artículo que explica cómo realizar el efecto del fuego, usado ya en muchas demos, pero que puede servir para introducirse en los métodos usados en el mundillo.

Witchlord
http://www.witchlord.com

4806 visitas | Puntuación: ■■■■ (2 votos) Vota

En este lugar podrás encontrar diversos artículos que te guiarán en la producción de efectos utilizando Visual C++ y las DirectX. Excelente para dar tus primeros pasos en estos temas. También contiene una excelente recopilación de artículos de otras web.

Cursos
2 direcciones

[Subir]

Demonomicron
http://www.encomix.es/~chispa/

3545 visitas | Puntuación: ■■■ (3 votos) Vota

Interesantísima iniciativa de la escena española por enseñar las bases que debe tener todo buen programador de demos o 'coder'. Desafortunadamente, parece no haber tenido continuidad.

Abe's Demoschool
http://www.mds.mdh.se/föreningar/small/abe/

3811 visitas | Puntuación: ■■ (1 voto) Vota

Pequeña introducción al mundillo que, desafortunadamente, se ha quedado demasiado antigua.

Generales
2 direcciones

[Subir]

Faqsys
http://www.neutralzone.org/home/faqsys

6652 visitas | Puntuación: ■■■ (2 votos) Vota

Excelente web con artículos y tutoriales, generalmente en formato texto y con código de ejemplo.

Hornet
http://www.hornet.org/code

3941 visitas | Puntuación: ■■■■■ (3 votos) Vota

Hornet fue el archivo mundial del mundo de la demoscene hasta su cierre en 1998. Por ello cuenta con la mejor recopilación de tutoriales y código fuente que puedes encontrar, siempre y cuando no busques material demasiado moderno. Además, todo lo que puedas buscar está comentado y puntuado.

Publicaciones

1 dirección

[Subir]

Hugi

http://www.hugi.de

3440 visitas | Puntuación: ■■■■■ (3 votos) Vota

Revista electrónica para PCs, que en cada número tiene una importante remesa de artículos sobre programación de demos, entre otras muchas cosas.

Principio Página

Direcciones.

Nuestras zonas

Otros lenguajes

11 direcciones

Recopilación de direcciones lenguajes no incluidos en otras categorías, como pueden ser los lenguajes declarativos o los funcionales. El principal interés de muchos de ellos es sobre todo académico.

- Haskell (3)
- J (1)
- Prolog (7)

Principio Página

Direcciones. Otros lenguajes. Prolog

7 direcciones

Recopilación de direcciones sobre Prolog, el más popular de los lenguajes declarativos. Estos lenguajes se caracterizan porque en ellos se le indica al ordenador qué es lo que se quiere que haga, pero no cómo se quiere que lo haga. Sin embargo, y por ahora, su uso está restringido sobre todo a universidades.

»EN ESTA PAGINA

Compiladores

4 direcciones

[Subir]

Visual Prolog

http://www.e-zapac87.f2s.com

12372 visitas | Puntuación: ■■■■ (11 votos) Vota

Entorno de desarrollo completo, con IDE, depurador, etc., para Prolog. Disponible en entorno Windows, Unix y OS/2.

CIAO Prolog

http://clip.dia.fi.upm.es/Software/Ciao/

5999 visitas | Puntuación: ■■■■ (14 votos) Vota

Potente compilador GNU, que interpreta el estándar ISO de Prolog y dispone de una cantidad ingente de librerías y una herramienta para la generación automática de documentación.

Sicstus Prolog

http://www.sics.se/isl/sicstus.html

3902 visitas | Puntuación: ■■■■ (2 votos) Vota

El más completo compilador de Prolog, incorpora añadidos habituales en compiladores de otros lenguajes pero habitualmente ausentes cuando hablamos de Prolog. También dispone de al posibilidad de enlazar código en este lenguaje dentro de programas escritos en otros lenguajes como, por ejemplo, Visual Basic. Eso sí, es de pago.

SWI Prolog

http://swi.psy.uva.nl/projects/SWI-Prolog/

5164 visitas | Puntuación: ■■■■ (15 votos) Vota

Pequeña, completa y útil implementación del Prolog de la Universidad de Edinmburgo. Principalmente realizada con fines académicos, se distribuye bajo licencia GPL.

Generales

1 dirección

[Subir]

Programación Lógica

http://clip.dia.fi.upm.es/proglog/

5796 visitas | Puntuación: ■■■ (9 votos) Vota

Interesante página con un curso completo sobre Programación Lógica de la Facultad de Informática de Madrid en transparencias, a veces no muy bien explicadas, aparte de enlaces a un par de intérpretes gratuitos. Es triste que en una universidad española sólo tengan estas páginas en inglés, pero al menos es de lo mejorcito que hemos encontrado.

Tutoriales

2 direcciones

[Subir]

Tutorial de Prolog en Castellano

http://www.programacion.net/cursos/prolog1/

14868 visitas | Puntuación: ■■■ (20 votos) Vota

Este tutorial pretende ser una guía básica de introducción al lenguaje de programación Prolog. Se engloba dentro de una serie de cursos que pretende profundizar en la materia a medida que se adquiere dominio de dicho lenguaje.

Tutorial de Prolog

http://proton.ucting.udg.mx/tutorial/prolog/index.htm

11843 visitas | Puntuación: ■■■ (8 votos) Vota

Tutorial de una sola página (eso si, muy grande), que parece insistir más en el manejo del intérprete de la Universidad de Edinburgo que en la comprensión del lenguaje. Aconsejable para los que ya hayan pasado de ahí.

Principio Página

Direcciones. Otros lenguajes. Haskell

3 direcciones

Recopilación de direcciones sobre Haskell, el más moderno estándar entre los lenguajes funcionales. Estos lenguajes se caracterizan por tener como unidad lógica de procesamiento la función, en lugar de la sentencia como sucede en los lenguajes imperativos u orientados a objetos.

Generales

1 dirección

[Subir]

The Haskell Home Page

http://www.haskell.org

3732 visitas | Puntuación: ■■■ (3 votos) Vota

El punto de partida de todo programador de Haskell. Incluye enlaces, tutoriales, compiladores, etc.

Tutoriales

2 direcciones

[Subir]

Curso de Haskell de Jeroen Fokker

http://www.cs.uu.nl/people/jeroen/courses/fp-sp.pdf

1004 visitas | Puntuación: ■■■■ (5 votos) Vota

Curso empleado en las clases de la Universidad de Utrecht. Aún cuando se refiera a Gofer, en realidad el lenguaje es Haskell, que por lo visto tiene hasta apodos. Es muy sencillo y está muy bien traducido.

Introducción a Haskell

http://horru.lsi.uniovi.es/~labra/FTP/IntHaskell98.pdf

7798 visitas | Puntuación: ■■■■ (12 votos) Vota

Completo pero conciso tutorial de Haskell, escrito por José Labra. Comienza razonando la necesidad de la creación de Haskell, para ir desgranando cada una de las características principales de este lenguaje.

Principio Página

Contacto - Datos legales

Direcciones. Herramientas

Nuestras zonas

51 direcciones

Recopilación de direcciones de herramientas de programación, como compiladores, editores, etc..

- Compiladores (6)
- Desarrollo web (37)
- Editores (1)
- Lotus Notes & Domino (7)

Principio Página

Direcciones. Herramientas.

Nuestras zonas

Editores

1 dirección

Recopilación de sitios que albergan editores o a páginas de compañias dedicadas a la construccion de los mismos.

»EN ESTA PAGINA

Gratuitos

1 dirección

[Subir]

Grasp

http://www.eng.auburn.edu/grasp/

8221 visitas | Puntuación: ■■■■ (17 votos) Vota

Editor multilenguaje cuya mayor originalidad consiste en la generación de CSD (Control Structure Diagram); unos gráficos situados a la izquierda del código que ayudan a comprender rápidamente su estructura. Edita archivos ADA, C, C++, Java y VHDL y es multiplataforma.

Principio Página

Direcciones. Herramientas. Compiladores

Nuestras zonas

6 direcciones

Recopilación de sitios que albergan compiladores o a páginas de compañias dedicadas a la construccion de los mismos.

»EN ESTA PAGINA

»IMPRESCINDIBLE VISITAR

En otros idiomas

- Compilador GNAT

Gratuitos

6 direcciones

[Subir]

CIAO Prolog

http://clip.dia.fi.upm.es/Software/Ciao/

5999 visitas | Puntuación: ■■■■ (14 votos) Vota

Potente compilador GNU, que interpreta el estándar ISO de Prolog y dispone de una cantidad ingente de librerías y una herramienta para la generación automática de documentación.

Compilador GNAT

http://www.gnat.com

4973 visitas | Puntuación: ■■■■■ (3 votos) Vota

Compilador GNU de ADA, disponible para diversas plataformas y distribuido a lo largo y ancho de Internet en miles de sitios FTP.

DJGPP

http://www.delorie.com/djgpp/

9795 visitas | Puntuación: ■■■■ (15 votos) Vota

La página del famoso compilador de C para DOS/Windows. Es una versión del GNU C presente en muchos Unix para estos sistemas operativos.

Freepascal

http://www.freepascal.org/

8408 visitas | Puntuación: ■■■■ (22 votos) Vota

Página que alberga el proyecto Freepascal, que intenta crear un compilador gratuito y multiplataforma de Object Pascal, la versión orientada a objeto del lenguaje Pascal creada por Borland.

Harbour Project

http://www.winfakt.com/harbour-project/

5669 visitas | Puntuación: ■■■■ (13 votos) Vota

En estas páginas encontrareis un compilador de xBase (Clipper) freeware todavía algo inmaduro.

SWI Prolog

http://swi.psy.uva.nl/projects/SWI-Prolog/

5164 visitas | Puntuación: ■■■■ (15 votos) Vota

Pequeña, completa y útil implementación del Prolog de la Universidad de Edinmburgo. Principalmente realizada con fines académicos, se distribuye bajo licencia GPL.

Principio Página

Contacto - Datos legales

Direcciones.

Nuestras zonas

Sistemas operativos

25 direcciones

Recopilación de sitios que albergan información para programar bajo un sistema operativo específico.

- AS/400 (3)
- Linux (14)
- OS/2 (1)
- PalmOS (1)
- WindowsNT (5)

»EN ESTA PAGINA

Cursos

1 dirección

[Subir]

Sistemas Operativos

http://exa.unne.edu.ar/depar/areas/informatica/SistemasOperativos/SOF.htm

2053 visitas | Puntuación: ■■■ (3 votos) Vota

Completo libro online cuyos contenidos corresponden a un curso universitario de Sistemas Operativos Convencionales y Distribuidos. Se puede descargar el curso en PDF. La página principal, no obstante, está sobrecargada a la extenuación.

Principio Página

Nuestras zonas

Direcciones. Sistemas operativos. WindowsNT

5 direcciones

Recopilación de direcciones sobre Windows NT/2000.

»EN ESTA PAGINA

Cursos

2 direcciones

[Subir]

Ayudas y guías para Windows NT

http://www.win-nt.com.ar

3964 visitas | Puntuación: ■■■ (12 votos) Vota

Manuales y trucos sobre Windows NT Server, bastante completos.

Portal Certificación MCSE en español

http://www.certificacionmcse.com

8663 visitas | Puntuación: ■■■■ (14 votos) Vota

Primer portal en español dedicado a las certificaciones Cisco y Microsoft para el mundo hispanoparlante. tests, foros de debates, rankings, exámenes, trucos, enlaces imprescindibles, etc...

Generales
2 direcciones

[Subir]

Intranet con Windows NT
http://www.globalnet.com.mx/intranet/

8306 visitas | Puntuación: ■■ (13 votos) Vota

Implementación de una Intranet para el trabajo en grupo sobre Windows NT. Por José González Moreno

Windows NT Resource Site
http://www.interlacken.com/winnt/default.htm

5224 visitas | Puntuación: ■■■■ (4 votos) Vota

En Inglés. Utilidades para la administración de sistemas y sites NT. Trucos para la Administración de NT, herramientas para los servidores Web NT, etc. Así como recomendaciones de libros.

Revistas
1 dirección

[Subir]

Windows NT Magazine
http://www.wntmag.com/

6618 visitas | Puntuación: ■■■■ (8 votos) Vota

Pagina Web esta conocida revista sobre el sistema operativo Window NT.

Principio Página

Direcciones. Sistemas operativos. Linux

14 direcciones

Recopilación de información sobre Linux

Nuestras zonas

»EN ESTA PAGINA

»IMPRESCINDIBLE VISITAR

En castellano

- ZonaLinux
- Anillo de Linux en castellano

Distribuciones

3 direcciones

[Subir]

Debian/GNU Linux

http://www.es.debian.org/

5424 visitas | Puntuación: ■■■■ (9 votos) Vota

Información en castellano sobre esta distribución de Linux.

ESware

http://www.esware.com/

4237 visitas | Puntuación: ■■■■ (4 votos) Vota

Distribución española. Con la línea de comandos castellanizada, páginas man en castellano, y soporte técnico.

Eurielec

http://www.etsit.upm.es/~eurielec/linux/

3791 visitas | Puntuación: ■■■■ (2 votos) Vota

Distribución española. Basada en la distribución de Red Hat.

Generales
8 direcciones

[Subir]

INSFLUG
http://www.insflug.org/

668 visitas | Puntuación: ■■■ (1 voto) Vota

Site donde se coordina la traducción "oficial" de documentos breves, como los COMOs (HOW-TOs) y PUFs o Preguntas de Uso Frecuente, las FAQs en inglés.

Linux Start
http://es.linuxstart.com/category.php?file=/development/

7816 visitas | Puntuación: ■■■■ (2 votos) Vota

Un portal de Linux muy básico, cuyo mayor interés reside en su buscador y, por supuesto, su categoría de desarrollo.

Anillo de Linux en castellano
http://linux-es.uio.no/Linuxring/

15304 visitas | Puntuación: ■■■■■ (3 votos) Vota

El 'Anillo Linux en Castellano' es un projecto mediante el cual todas las paginas sobre Linux en castellano estaran enlazadas unas con otras, formando parte de un anillo de paginas con un tema en comun, Linux en castellano.

Hispalinux
http://www.hispalinux.es/

4812 visitas | Puntuación: ■■■■ (6 votos) Vota

Asociación de usuarios de Linux en español.

Linux de Novato a Novato
http://personal2.redestb.es/traque/

10527 visitas | Puntuación: ■■■■ (12 votos) Vota

Web sobre Linux dirigida esencialmente a los que empiezan con este magnifico sistema operativo. Mantenida por Fernando Travesedo.

LinuxBusca
http://www.linuxbusca.lanets.net/

11289 visitas | Puntuación: ■■■■ (2 votos) Vota

Primer directorio en español con motor de búsqueda sobre recursos Linux.

ZonaLinux
http://www.zonalinux.com/

7581 visitas | Puntuación: ■■■■■ (5 votos) Vota

ZonaLinux.com, página web con todo tipo de recursos de Linux. Noticias diarias, documentación, boletín mensual, foros de discusión, etc. Mantenida por David Lizano.

Linux Web Ring
http://linuxwebring.org/

2941 visitas | Puntuación: ■■■ (1 voto) Vota

Anillo sobre de páginas sobre Linux en Ingles.

Noticias

2 direcciones

[Subir]

Linux Preview

http://linux.ncc.org.ve/

7286 visitas | Puntuación: ■■■ (1 voto) Vota

Página web con noticias, 'reviews', y enlaces.

SoloLinux

http://www.sololinux.com/

5399 visitas | Puntuación: ■■■■ (2 votos) Vota

Noticias sobre Linux en castellano. Esta sección pertenece a Noticas.com, noticias intercom.

Revistas

1 dirección

[Subir]

Linux Focus en castellano

http://www.linuxfocus.org/Castellano/

5561 visitas | Puntuación: ■■■■ (10 votos) Vota

Es una revista internacional y libre sobre linux. Es una organización sin ánimo de lucro, y la revista es llevada por voluntarios de todas partes del mundo.

Principio Página

Direcciones.

Nuestras zonas

Internet

164 direcciones

Recopilación de sitios que albergan información para programar en Internet.

- ASP (12)
- CGI (1)
- ColdFusion (4)
- CSS (2)
- Flash (8)
- Generales (6)
- HTML (10)
- PHP (21)
- SVG y VML (11)
- WAP (30)
- XHTML (1)
- XML (30)
- XSL, XSLT y Xpath (28)
- Java@ (27)
- Javascript@ (27)

Principio Página

Direcciones. Internet. HTML

10 direcciones

Recopilación de direcciones sobre HTML y recursos a la hora de realizar páginas web. Para profundizar más en este tema, recomendamos visitar HTML en castellano.

- Desarrollo web@ (37)

Nuestras zonas

»EN ESTA PAGINA

»IMPRESCINDIBLE VISITAR

En castellano

- Tutor HTML
- HTML 4.01
- Tutorial HTML & CSS
- WebMaestro

Cursos

6 direcciones

[Subir]

Tutorial HTML & CSS

http://www.xiniom.com/users/Jorge/html/

3699 visitas | Puntuación: ■■■■■ (12 votos) Vota

Extenso tutorial enfocado a las nuevas versiones de HTML y CSS (hojas de estilos). Liberado bajo los términos de la Licencia de Documentación Libre GNU (LDLG o GFDL en inglés).

Curso de HTML de AulaFácil

http://www.aulafacil.org/CursoHtml/temario.htm

1874 visitas | Puntuación: ■■■ (2 votos) Vota

Un curso algo corto y bastante parco. Está dividido en lecciones muy pequeñas.

Curso Visual de HTML

http://www10.brinkster.com/sausant/nociones_html.html

5905 visitas | Puntuación: ■■■■ (13 votos) Vota

Curso básico de HTML, consta de lecciones explicadas por medio de diapositivas empleando Viewlets. A veces puede resultar un poco pesado este sistema, pero es perfecto para los que se pierdan con cursos más "ásperos".

Tutorial de HTML de "Aprende en Internet"

http://www.iespana.es/querol/tutoriales/cursohtm.htm

5330 visitas | Puntuación: ■■■■ (10 votos) Vota

Un pequeño tutorial que toma como proyecto la construcción de una sencilla web de seis páginas, que va creandose a medida que se aprenden nuevos conceptos.

WebMaestro

http://www.american.edu.co/vs/cursoweb/docs/portada.html

6494 visitas | Puntuación: ■■■■■ (5 votos) Vota

Copia del curso, ya desaparecido, de Francisco Arocena, posiblemente el más celebre de Internet en nuestro idioma.

Tutor HTML

http://gias720.dis.ulpgc.es/Gias/Cursos/Tutorial_html/indice.htm

4656 visitas | Puntuación: ■■■■■ (7 votos) Vota

Un buen curso de HTML, bastante completo.

Especificaciones

2 direcciones

[Subir]

HTML 3.2

http://dns.uncor.edu/info/html/rec-sp.htm

1536 visitas | Puntuación: ■■■ (1 voto) Vota

Ya se ha quedado un poco antigua pero resulta algo más pequeña y manejable que la 4.0.

HTML 4.01

http://html.conclase.net/w3c/html401-es/cover.html

6156 visitas | Puntuación: ■■■■■ (5 votos) Vota

Especificación en español de la última revisión del HTML, la 4.01. Conviene tenerla a mano para resolver dudas.

Generales
1 dirección

[Subir]

Tejedores del web
http://www.tejedoresdelweb.com/

2902 visitas | Puntuación: ■■■■ (2 votos) Vota

Uno de los más antiguos y venerables sitios dedicados a enseñar HTML y temas afines. Se actualiza con frecuencia con todo tipo de cursos.

Listas de correo
1 dirección

[Subir]

Web-ES
http://www.rediris.es/list/info/web-es.es.html

1469 visitas | Puntuación: ■■■■ (2 votos) Vota

La lista española con más solera, ofrecida por RedIRIS.

Principio Página

Contacto - Datos legales

Direcciones.
Internet.
XML

30 direcciones

En estas páginas encontraréis todo tipo de documentación sobre XML tanto en ingles como en castellano. Para profundizar más en este tema, recomendamos visitar HTML en castellano.

Nuestras zonas

»EN ESTA PAGINA

»IMPRESCINDIBLE VISITAR

En castellano

- Entorno a SGML/XML
- SGML-ESP
- Recursos XML en RAMON.ORG
- Desarrollo de aplicaciones Web con JSP y XML

En otros idiomas

- XMLHack
- Cafe con Leche
- XML-DEV
- Web sobre SGML/XML de Robin Cover
- XMLINFO

Artículos generales

7 direcciones

[Subir]

XML Roadmap

http://www.aqs.es/web/files/xml-roadmap.pdf

11208 visitas | Puntuación: ■■■■ (5 votos) Vota

XML Roadmap es una presentación de las diversas tecnologías que están relacionadas con XML. Este documento permite establecer una base de información para entrar en el mundo de XML sin perderse, encontrando la información más actualizada posible. A su vez, muestra las distintas aplicaciones y las herramientas necesarias, para el ámbito del desarrollo en XML.

Recursos XML en RAMON.ORG

http://www.ramon.org/xml/index2.htm

10240 visitas | Puntuación: ■■■■■ (11 votos) Vota

Buena recopilación de artículos, tutoriales y enlaces sobre XML. A destacar su excelente cursos de VML.

Entrevista a Tim Bray

http://www.revistaweb.com/entrevistes/tbrayen9.html

10744 visitas | Puntuación: ■■■ (2 votos) Vota

Entrevista a Tim Bray, uno de los coeditores de las especificaciones del XML, en la revista WEB.

Introducción al XML por Fernando Santamaria

http://fesabid98.florida-uni.es/Comunicaciones/f_santamaria/f_santamaria.htm

9907 visitas | Puntuación: ■■■■ (7 votos) Vota

Interesante introducción al XML de la mano de Fernando Santamaria. Este artículo fue presentado en las VI Jornadas Españolas de documentación.

Tutorial de introducción al XML de Oscar Lechuga Gomez.

http://face.el.uma.es/imasd/xml/xml.html

10349 visitas | Puntuación: ■■■ (9 votos) Vota

Tutorial de introducción al XML de Oscar Luis Lechuga Gómez de la Universidad de Málaga.

XML y comercio electrónico

http://www.marketingycomercio.com/numero5/5xml.htm

4450 visitas | Puntuación: ■■■ (1 voto) Vota

Artículo sobre XML y el comercio electrónico, publicado en la revista Marketing y comercio electrónico.

XML, Java y el futuro de la Web

http://metalab.unc.edu/pub/sun-info/standards/xml/why/xmlapps.htm

5242 visitas | Puntuación: ■■■■ (2 votos) Vota

Excelente artículo sobre XML, su futuro y su relación con el Java. Escrito por Jon Bosak, uno de los padres del XML.

Especificaciones
3 direcciones

[Subir]

XML 1.0 en castellano
http://slug.ctv.es/~olea/sgml-esp/Rec-xml.html

20475 visitas | Puntuación: ■■■■ (3 votos) Vota

Traducción al castellano de la especificación del XML 1.0

La especificación del XML comentada.
http://www.xml.com/axml/axml.html

4585 visitas | Puntuación: ■■■■ (2 votos) Vota

La recomendación del XML comentada por Tim Bray, uno de sus coeditores.

XML 1.0
http://www.w3.org/TR/1998/REC-xml-19980210

3891 visitas | Puntuación: ■■■■ (2 votos) Vota

XML 1.0. Recomendación definitiva del W3C del 10 de Febrero de 1998.

FAQs
4 direcciones

[Subir]

Perl XML Faq
http://cronopio.net/perl/faqs/perlxmlfaq.html

3777 visitas | Puntuación: ■■■ (3 votos) Vota

Faq sobre como trabajar con XML desde Perl

XML FAQ en castellano
http://slug.ctv.es/~olea/sgml-esp/xfaq13.html

11373 visitas | Puntuación: ■■■■ (2 votos) Vota

Traducción del FAQ sobre XML de Peter Flynn.

El XML en 20 preguntas
http://builder.com/Authoring/Xml20/index.html

4775 visitas | Puntuación: ■■■ (1 voto) Vota

20 preguntas y respuestas que ayudan a saber que es el XML.

XML FAQ de Peter Flynn
http://www.ucc.ie/xml/

3810 visitas | Puntuación: ■■■■ (1 voto) Vota

FAQ sobre XML de Peter Flynn. Es sin duda el más completo y utilizado.

WEBs de información general sobre XML

6 direcciones

[Subir]

Entorno a SGML/XML

http://orion.deusto.es/~abaitua/konzeptu/sgml.htm

8537 visitas | Puntuación: ■■■■■ (2 votos) Vota

Web de Joseba Abaitua dedicada al SGML y XML. Posiblemente una de las primeras Webs en castellano dedicado al SGML. Muy recomendables los artículos que tiene escritos sobre el tema y en especial su tutorial sobre SGML.

Cafe con Leche

http://metalab.unc.edu/xml/

4791 visitas | Puntuación: ■■■■■ (1 voto) Vota

Recopilación de noticias y recursos sobre XML.

Web sobre SGML/XML de Robin Cover

http://www.oasis-open.org/cover/

4388 visitas | Puntuación: ■■■■■ (1 voto) Vota

La mejor Web sobre SGML/XML que existe. Indispensable para estar al dia.

XML.COM

www.xml.com

18408 visitas | Puntuación: ■■■ (2 votos) Vota

De lo mas completito con gran cantidad de artículos y recursos.

XMLHack

http://www.xmlhack.com

4551 visitas | Puntuación: ■■■■■ (1 voto) Vota

Nuevo Web dirigido a los desarrolladores en XML, con noticias, opiniones y cualquier información que pueda resultar útil a los desarrolladores.

XMLINFO

http://www.xmlinfo.com/

5700 visitas | Puntuación: ■■■■■ (1 voto) Vota

Web mantenida por James Taubercon gran cantidad de recursos sobre XML.

Listas de correo y news

2 direcciones

[Subir]

SGML-ESP

http://slug.ctv.es/~olea/sgml-esp/

12583 visitas | Puntuación: ■■■■■ (1 voto) Vota

Es una lista de distribucion sobre SGML y XML en castellano.

XML-DEV

http://www.vsms.nottingham.ac.uk/vsms/xml/jewels.html

8270 visitas | Puntuación: ■■■■■ (1 voto) Vota

Es una lista de correo para desarrolladores de aplicaciones XML.

Tutoriales

8 direcciones

[Subir]

Desarrollo de aplicaciones Web con JSP y XML

http://java.programacion.net/jspyxml/index.php

8385 visitas | Puntuación: ■■■■■ (11 votos) Vota

Muy buen tutorial de SUN en castellano sobre como desarrollar aplicaciones Web utilizando de forma combinada JSP y XML.

Tutorial de XML de Angel Barbero

http://www.dat.etsit.upm.es/~abarbero/curso/xml/xmltutorial.html

15373 visitas | Puntuación: ■■■■ (22 votos) Vota

Buen tutorial de XML de Angel Barbero.

XML tutorial en Zvon

http://zvon.org/xxl/XMLTutorial/General/book_en.html

4942 visitas | Puntuación: ■■■■ (2 votos) Vota

Breve pero conciso tutorial sobre XML escrito por Milosvac Nic.

Las entidades en el XML

http://www.xml.com/xml/pub/98/08/xmlqna0.html

4179 visitas | Puntuación: ■■ (2 votos) Vota

Completísima introducción a la correcta utilización de las entidades en el XML. Su autor es Norman Walsh.

Tutorial de XML de Frank Boumphrey

http://www.hypermedic.com/style/xml/xmlindex.htm

5232 visitas | Puntuación: ■■■ (1 voto) Vota

En esta dirección encontraras un estupendo tutorial sobre XML. Esta dividido en dos partes: una primera más sencilla en la que aprendemos como podemos construir nuestros documentos en función de una DTD y como podemos mostrarlos y luego una segunda parte en la que se abordan temas más avanzados como son los Xlink, Xpointer, RDF, namespaces etc.

Tutorial de XML en Microsoft

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/xmlsdk30/htm/xmtutxmltutorial.asp

6993 visitas | Puntuación: ■■■■ (2 votos) Vota

Buen tutorial sobre XML de Microsoft. Indispensable si queremos aprender a utilizar el XML desde el Explorer.

Tutoriales de XML en IBM

http://www.software.ibm.com/xml/education/tutorial-prog/abstract.html

6017 visitas | Puntuación: ■■■■ (1 voto) Vota

Índice de los tutoriales que IBM tiene en su Web sobre XML. Podeis obtenerlos tanto en formato HTML como PDF.

XML atributos y entidades

http://www.hotwired.com/webmonkey/98/45/index3a.html?tw=eg1998453

3930 visitas | Puntuación: ■■■ (1 voto) Vota

Tutorial en el que se profundiza en la utilización de los atributos y las entidades en los documentos XML

Principio Página

Direcciones.
Internet.
WAP

30 direcciones

Recopilación de direcciones sobre WAP..

Nuestras zonas

»EN ESTA PAGINA

»IMPRESCINDIBLE VISITAR

En castellano

- WapColombia
- Comunidad sobre comunicaciones móviles en ICT-NET
- WMLCLUB
- WAPes

En otros idiomas

- Desarrolladores WAP en Nokia
- Phone Developer Zone
- Wireless Developer Network

ASP
5 direcciones

[Subir]

Corobori
http://www.corobori.com/wap/

4227 visitas | Puntuación: ■■■ (1 voto) Vota

Buena página con unos cuantos ejemplos sobre como desarrollar aplicaciones WAP utilizando ASPs. Esta en tres idiomas y entre ellos en castellano. Los ejemplos se pueden bajar en formato ZIP.

An Online Shopping Cart WAP Application using WML and ASP
http://www.asptoday.com/articles/20000309.htm

3488 visitas | Puntuación: ■■■ (2 votos) Vota

Estupendo artículo de By Wei Meng Lee sobre como implementar un carrito de la compra utilizando ASP y WML.

Configurar el IIS para trabajar con WAP
http://www.anywhereyougo.com/ayg/ayg/wap/Article.po?type=WAP_Tutorial&page=12720

5880 visitas | Puntuación: ■■■ (1 voto) Vota

Articulo en el que se explica como debemos configurar el IIS para poder trabajar con aplicaciones WAP.

WAP y ASP
http://www.asptoday.com/articles/19991115.htm

3787 visitas | Puntuación: ■■■■ (2 votos) Vota

Buen artículo de introducción a WAP y como utilizarlo con los ASPs. Esta es la primera parte de dos artículos sobre el tema. Escrito por Luca Passani.

WAP, ASP y XML
http://www.webtechniques.com/archives/2000/03/passani/

3705 visitas | Puntuación: ■■ (2 votos) Vota

Estupendo artículo de Luca Passani sobre generar aplicaciones WAP utilizando XML, ASP y XSL. Destacamos de este artículo la combinación de XML y XSLT para la generación de WML.

Generales
12 direcciones

[Subir]

ChileWAP
http://www.chilewap.cl

3739 visitas | Puntuación: ■■■■ (3 votos) Vota

Gran variedad de recursos en español sobre WAP, tutoriales, artículos, noticias, etc.

Pyweb.com
http://www.pyweb.com

3472 visitas | Puntuación: ■■■■ (5 votos) Vota

Web que ofrece la creación de sitios WAP desde su contenido HTML. PyWeb.com traduce dinámicamente y gratis todo el contenido de páginas HTML (incluyendo imágenes, formularios, marcos, etc.), o solamente la información crítica de su web, para que sea accesible desde todos los móviles WAP. Ademas : Un emulador WAP especial que contiene nuestras herramientas de traducción, para fijar las versiones WML de cualquier sitio , esté en HTML o WML. Un constructor de WBMP para ayudarle dentro de la creación de los argumentos utilizados con nuestra etiqueta "traductor de imágenes".

WapColombia

http://wapcolombia.ucauca.edu.co

7748 visitas | Puntuación: ■■■■■ (7 votos) Vota

Es Un portal dinámico en Wap cuyo principal objetivo es crear una comunidad para Colombia de desarrolladores interesados en la tecnología Wap

Comunidad sobre comunicaciones móviles en ICT-NET

http://www.ictnet.es/esp/comunidades/movil/

3838 visitas | Puntuación: ■■■■■ (2 votos) Vota

Comunidad sobre comunicaciones moviles en ICT-NET. Gran cantidad de artículos, documentación y enlaces. Mantenida por Luis Pont.

Bigwapsite

http://www.gate-keeper.org.uk/

1853 visitas | Puntuación: ■■■■ (1 voto) Vota

Un tutorial sencillo y comprensible, FAQ, descargas y, sobre todo, un directorio de enlaces sobre WAP con casi doscientas direcciones.

anywhereyougo

http://www.anywhereyougo.com/ayg/ayg/Index.po?

5156 visitas | Puntuación: ■■■■ (2 votos) Vota

Muy buena recopilación de información sobre Wireles, WAP y Bluetooth.

Desarrolladores WAP en Nokia

http://www.forum.nokia.com/main/1,6668,1_1,00.html

3944 visitas | Puntuación: ■■■■■ (1 voto) Vota

Zona de Nokia de la Web de WAP dedicada a los desarrolladores.

Ericsson WAP developer Zone

http://www.ericsson.com/developerszone/

3196 visitas | Puntuación: ■■■■ (1 voto) Vota

Zona de la Web de Ericsson dedicada a los desarrolladores WAP

Phone Developer Zone

http://updev.phone.com/

3406 visitas | Puntuación: ■■■■■ (1 voto) Vota

Sección de Phone dedicada a los desarrolladores WAP.

WAPForum

http://www.wapforum.com/

3107 visitas | Puntuación: ■■■■ (1 voto) Vota

Página oficial de WAP. Esta Web esta formada por las organizaciones y empresas encargadas de desarrollar las especificaciones relacionadas con este tema.

Wapholesun
http://www.wapholesun.com/

7212 visitas | Puntuación: ■■■■ (1 voto) Vota

Web sobre WAP en ingles desarrollada por el español Carlos Fernández. Indispensable para los que esten interesados en desarrollar juegos para móviles.

Wireless Developer Network
http://www.wirelessdevnet.com/

3493 visitas | Puntuación: ■■■■■ (1 voto) Vota

Web de indispensable visita para estar al día en el desarrollo de aplicaciones Wireless.

Java
3 direcciones

[Subir]

Generar WML desde un Servlet
http://java.programacion.net/taller/wmlservlet.htm

3265 visitas | Puntuación: ■■■■ (1 voto) Vota

Artículo en castellano desde el que se explica como generar WML desde un servlet.

Building Servlets to Output WML Content
http://www.anywhereyougo.com/ayg/ayg/wap/Article.po?type=WAP_Tutorial&page=10743

4975 visitas | Puntuación: ■■■ (1 voto) Vota

Articulo en el que se explica como generar WML desde un servlet.

Developing Wireless Applications with WAP, WML, and JSP
http://www.xml.com/pub/2000/06/26/xmldevcon/wirelessapps.html

3182 visitas | Puntuación: ■■■ (1 voto) Vota

Artículo en el que David Sims explica las ventajas de desarrollar aplicaciones WAP utilizando JSP, XML y XSLT.

Listas de correo
2 direcciones

[Subir]

WAPes
http://es.egroups.com/group/wapes

4681 visitas | Puntuación: ■■■■■ (1 voto) Vota

Lista sobre tecnología WAP en castellano.

wmlprogramming
http://www.egroups.com/group/wmlprogramming

3008 visitas | Puntuación: ■■■■ (1 voto) Vota

Lista en ingles sobre WML y WMLScript dirigida a desarrolladores de aplicaciones WAP.

Otros artículos
1 dirección

[Subir]

Configurar Apache para Wireless Browsers
http://mikal.org/interests/articles/article0001.jsp

3283 visitas | Puntuación: ■■■ (1 voto) Vota

Artículo en el que se explica como debemos configurar nuestro servidor Apache para que pueda servir páginas WML.

Perl
1 dirección

[Subir]

Z-Y-G-O
http://wap.z-y-g-o.com/tools/

3120 visitas | Puntuación: ■■■■ (1 voto) Vota

Libreria de ejemplos en Perl para trabajar con WML: WML to WMLc, HTML to WML, etc.

PHP
1 dirección

[Subir]

HAWHAW
http://www.hawhaw.de

3281 visitas | Puntuación: ■■■■ (1 voto) Vota

Librería de clases PHP para generar facilmente tanto páginas HTML como páginas WML.

Tutoriales
3 direcciones

[Subir]

ABC del WAP
http://usuarios.lycos.es/manualeswap

1271 visitas | Puntuación: ■■■■ (3 votos) Vota

Manual de WAP, más enfocado al protocolo en sí que a los lenguajes WML y WMLScript.

Wapfacil
http://wapfacil.esgratis.net/

9985 visitas | Puntuación: ■■■■ (1 voto) Vota

Una de las primeras Webs sobre Wap en castellano. Tiene un tutorial de WML, ejemplos de WMLScript y una recopilación de manuales y documentos sobre WAP traducidos al castellano.

WMLCLUB

http://www.wmlclub.com

4700 visitas | Puntuación: ■■■■■ (7 votos) Vota

Interesante Web con gran cantidad de información sobre WAP. Tiene un tutorial de WML, otro de WMLScript y un montón de secciones: enlaces, ofertas de trabajo, un estupendo FAQ, etc..

XML

2 direcciones

[Subir]

A Mobile Window on our Portal

http://www.xml.com/pub/2000/05/31/style/index.html

3150 visitas | Puntuación: ■■■■ (1 voto) Vota

Interesantisimo artículo de Didier Martin sobre como generar un portal WAP y HTML utilizando XML y XSLT.

Content Management and Distribution Using XML

http://www.wirelessdevnet.com/articles/jun2000/xmlcast.html

3073 visitas | Puntuación: ■■■■ (1 voto) Vota

Muy buen artículo sobre como podemos crear contenido para nuestra Web en diferentes formatos (HTML, RSS, WML) utilizando XML.

Principio Página

Direcciones.

Nuestras zonas

Teoría

9 direcciones

Recopilación de direcciones sobre teoría de la programación.

- Algoritmos (3)
- Metodología (2)
- Redes (1)
- Teoría de bases de datos (1)
- UML (2)

Principio Página

Direcciones.
Teoría.
Metodología

2 direcciones

Recopilación de direcciones sobre metodología de la programación.

Nuestras zonas

Cursos

2 direcciones

[Subir]

IntroProgra

http://ar.geocities.com/luis_pirir/

5479 visitas | Puntuación: ■■■■ (9 votos) Vota

Pequeño curso, no demasiado completo, de introducción a la programacion, centrada en la elaboración de algoritmos.

Curso de Metodología de la programación

http://64.226.188.26/sivnetwork-www/cursos_gratis/Indice%20Meto.htm

24640 visitas | Puntuación: ■■■■ (75 votos) Vota

Completo curso de metodología de la programación. Ideal para aquellos que no hayan programado nunca.

Principio Página

Direcciones. Teoría. UML

2 direcciones

Direcciones sobre Unified Modeling Language, el estándar de representación de diseño orientado a objetos, nacido de las cenizas de OMT y Booch.

Documentacion

2 direcciones

[Subir]

Taller UML en Vico.org

http://www.vico.org

23065 visitas | Puntuación: ■■■■ (68 votos) Vota

Concreta y práctica recopilación de información sobre UML. Muy interesantes las plantillas que ofrece para: matricular casos de uso, proyectos, etc.

Curso de Análisis y Diseño Orientado a Objetos

http://www.dsic.upv.es/~uml/

25912 visitas | Puntuación: ■■■■ (51 votos) Vota

Recopilación en diferentes formatos (PDF, PowerPoint) de un curso de UML impartido por dos profesores de la Universidad Politecnica de Valencia. Dispone de prácticas a realizar con la herramienta Rational Rose.

Principio Página

Direcciones.
Teoría.
Algoritmos

3 direcciones

Recopilación de direcciones con algoritmos o sobre creación de algoritmos (algorítmica).

Nuestras zonas

»EN ESTA PAGINA

Recopilaciones de algoritmos

3 direcciones

[Subir]

ALGORITMIA

http://www.algoritmia.cjb.net

18091 visitas | Puntuación: ■■■■ (74 votos) Vota

Recopilación de los algoritmos más habituales detalladamente explicados y organizados por secciones.

Algorithm Archive

http://wannabe.guru.org/alg/

20875 visitas | Puntuación: ■■■ (27 votos) Vota

Interesante y bien ordenado índice de algoritmos de lo más diverso, que van de la ordenación a la Inteligencia Artificial en los juegos. Desafortunadamente, está bastante incompleto.

Numerical Recipes in C

http://www.nr.com

10561 visitas | Puntuación: ■■■ (16 votos) Vota

Versión online (en PS y PDF) del famoso libro de algoritmos de cálculo numérico. No se puede bajar por completo, sino algoritmo por algoritmo, lo que lo hace más adecuado para resolver algún problema en concreto. Además los algoritmos no se pueden usar sin permiso de los autores.

Principio Página

Direcciones.

Nuestras zonas

Lenguajes imperativos

45 direcciones

Recopilación de direcciones sobre lenguajes imperativos.

- Ada (8)
- C (9)
- Clipper & xBase (9)
- Cobol (4)
- Ensamblador (6)
- Euphoria (1)
- Fortran (1)
- Pascal (5)
- QuickBasic (2)

Principio Página

Direcciones. Lenguajes imperativos. C

Nuestras zonas

9 direcciones

Recopilación de direcciones sobre el popular lenguaje C.

»EN ESTA PAGINA

»IMPRESCINDIBLE VISITAR

En castellano

- Gorka´s Web Page
- Lenguaje C

En otros idiomas

- UNIX System Calls and Subroutines using C

Cursos

4 direcciones

[Subir]

Curso de 'Peregrino a sueldo'

http://www.geocities.com/SiliconValley/Drive/1035/

11917 visitas | Puntuación: ■■■■ (14 votos) Vota

Un buen y completo curso de C, aunque parece dedicar un espacio excesivo a la programación gráfica en un sistema concreto (el DOS), que es de interés algo más marginal.

Gorka´s Web Page

http://www.geocities.com/g_urrutia/

8125 visitas | Puntuación: ■■■■■ (7 votos) Vota

Contiene un buen curso de C, desafortunadamente incompleto, y diversas páginas sobre el compilador DJGPP, incluyendo ayuda para su instalación e información sobre su traducción al castellano.

Programación en C, por Virgilio Gómez

http://www.geocities.com/SiliconValley/Garage/8211/frontal/progc.htm

14488 visitas | Puntuación: ■■■■ (13 votos) Vota

Lo mejor de este manual es la ayuda que ofrece al principiante enseñando algunos conceptos básicos.

UNIX System Calls and Subroutines using C

http://www.cs.cf.ac.uk/Dave/C/

5991 visitas | Puntuación: ■■■■■ (3 votos) Vota

Un libro completo disponible en HTML en el que no sólo enseña C, sino como acceder desde este lenguaje a las llamadas del sistema de Unix.

Generales

5 direcciones

[Subir]

Lenguaje C

http://www.jeanpaul.com.ar

2723 visitas | Puntuación: ■■■■ (4 votos) Vota

Página con enlaces a manuales, tutoriales e informacion diversa sobre programacion en C y temás afines (C++, visual C++, C#, OpenGL, DirectX, Linux, etc.).

MundoC.net

http://www.mundoc.net

3543 visitas | Puntuación: ■■■■ (3 votos) Vota

Portal dedicado a C y C++ con un buen curso de C y un buen número de ejemplos de código fuente y algunas utilidades, varios compiladores, trucos, foro, chat e, incluso, ofertas de empleo.

Lenguaje C

http://www.lenguaje-c.es.vg

6155 visitas | Puntuación: ■■■■■ (5 votos) Vota

Página en la que podrá encontrar información, manuales, artículos, etc, sobre el Lenguaje C

El Rincón del C

http://www.elrincondelc.com

12929 visitas | Puntuación: ■■■■ (18 votos) Vota

Cursos de programación en C , mucho código fuente, boletines, listas de correo e información sobre compiladores de C/C++. El punto de encuentro de los programadores de C.

DJGPP

http://www.delorie.com/djgpp/

9795 visitas | Puntuación: ■■■■ (15 votos) Vota

La página del famoso compilador de C para DOS/Windows. Es una versión del GNU C presente en muchos Unix para estos sistemas operativos.

Principio Página

Direcciones. Lenguajes imperativos. Pascal

5 direcciones

Recopilación de direcciones sobre el lenguaje Pascal y su sucesor, Turbo Pascal.

»EN ESTA PAGINA

Cursos

3 direcciones

[Subir]

Ejercicios resueltos

http://usuarios.lycos.es/VictorSanchez2/tutoriales/tutoriales.htm

4710 visitas | Puntuación: ■■■■ (21 votos) Vota

Página donde puedes encontrar más de 200 ejercicios resueltos en Pascal. Además, si tienes alguna duda puedes preguntar a su autor para que intente echarte una mano.

Tutorial de Turbo Pascal 7.0

http://www.terra.es/personal/raul_zm/

11508 visitas | Puntuación: ■■■■ (40 votos) Vota

Parco tutorial sobre Turbo Pascal 7.0. No entra en profundidad en las posibilidades del lenguaje ni explica demasiado los conceptos, por lo que sólo resulta adecuado para los ya iniciados en el mundo de la programación.

Tutorial de Turbo Pascal

http://members.xoom.com/tutoriales

11350 visitas | Puntuación: ■■■■ (21 votos) Vota

Muy buen tutorial de Turbo Pascal.

Generales
2 direcciones

[Subir]

Página de Pascal de Nacho Cabanes
http://members.es.tripod.de/ncabanes/pascal.htm

7980 visitas | Puntuación: ■■■■ (9 votos) Vota

Un buen punto de introducción al Pascal, con cursos y enlaces a compiladores incluidos.

Freepascal
http://www.freepascal.org/

8408 visitas | Puntuación: ■■■■ (22 votos) Vota

Página que alberga el proyecto Freepascal, que intenta crear un compilador gratuito y multiplataforma de Object Pascal, la versión orientada a objeto del lenguaje Pascal creada por Borland.

Principio Página

Direcciones.

Nuestras zonas

Varios

27 direcciones

Categoría que engloba aquellas direcciones de programación que no pudieron ser catalogadas en otras páginas.

- Generales (14)
- Gráfica (5)
- Historia (4)
- Personales (1)
- Publicaciones (1)
- Software libre (2)

Principio Página

Direcciones.
Varios.
Generales

14 direcciones

Direcciones que engloban demasiados aspectos de la programación como para poder ser restringidos en una sola categoría.

Nuestras zonas

» EN ESTA PAGINA

» IMPRESCINDIBLE VISITAR

En castellano

- La web del programador
- El Guru Programador
- MagiOS

Buscadores

1 dirección

[Subir]

Buscadoc

http://buscadoc.ods.org

2089 visitas | Puntuación: ■■■■ (5 votos) Vota

Buscador de documentacion informática solamente en castellano. Tiene indexadas la mayoria de webs en castellano que ofrecen documentación informatica (incluyendo la nuestra).

Cursos
2 direcciones

[Subir]

Cybertutos.com
http://www.cybertutos.com

5867 visitas | Puntuación: ■■ (2 votos) Vota

Dispone de algunos tutoriales y manuales de los mas usados lenguajes de programacion, herramientas de diseño web y grafico y algunos enlaces relacionados. Escaso de material.

Aprenda Informática como si estuviera en Primero
http://fcapra.ceit.es/AyudaInf/

9916 visitas | Puntuación: ■■■■ (10 votos) Vota

Diversos cursos en formato PDF ofrecidos a sus estudiantes por la Universidad de Navarra. Claros y completos, en ocasiones pueden resultar difíciles al lector no iniciado.

Generales
10 direcciones

[Subir]

El Tutorial
http://www.eltutorial.com

644 visitas | Puntuación: ■■■■ (3 votos) Vota

Portal de documentación técnica. Consiste básicamente en un buen número de enlaces bien organizados a artículos, tutoriales, libros, etc.. También dispone de foros propios y un glosario de términos técnicos.

MagiOS
http://www.magios.com

1046 visitas | Puntuación: ■■■■■ (17 votos) Vota

Portal en crecimiento dedicado a la programación y la seguridad informática, con noticias, artículos y algún tutorial.

El Guru Programador
http://www.elguruprogramador.com.ar

2553 visitas | Puntuación: ■■■■■ (7 votos) Vota

Agradable página dedicada a la programacion y el desarrollo web. Tiene bastantes articulos, algunos tutoriales, foros, y enlaces. Zonas especializadas de ASP, Visual Basic, Flash, JavaScript, SQL, XML, HTML y otros lenguajes.

CGR Software.com
http://www.cgrsoftware.com

1199 visitas | Puntuación: ■■■■ (2 votos) Vota

"Megabase" (según sus autores) de recursos para programadores. Incluye manuales, ejemplos de código, programas y recursos varios para programadores.

El Rincón del Programador (bis)
http://www.elrincondelprogramador.com

1362 visitas | Puntuación: ■■■■ (2 votos) Vota

Artículos, noticias, trucos y diversos recursos para los programadores. Dispone de canales especializados de ASP, C y C++, Delphi, Java, HTML, Flash y muchos otros. Nada que ver con el otro rincón del programador, que se sepa.

El Rincón del Programador

http://rinconprog.metropoli2000.com/

4853 visitas | Puntuación: ■■■■ (7 votos) Vota

Un rincón para todos aquellos interesados por la informática en general, y por la programación en particular, con interesantes tutoriales sobre Python, C, Programación gráfica, etc.

La web del programador

http://www.lawebdelprogramador.com/

5270 visitas | Puntuación: ■■■■■ (7 votos) Vota

Posiblemente, la web más visitada de programación en castellano. Posee un gran índice de recursos y documentación para el programador. Su mayor defecto es una estructura que dificulta la navegación. Ultimamente ha añadido un buscador.

laVariable

http://www.lavariable.com

3132 visitas | Puntuación: ■■ (2 votos) Vota

En laVariable.com se pueden encontrar trucos, artículos y tutoriales sobre ASP, CGI, XML, Java, SQL, JavaScript, VBScript y otras nuevas tecnologías.

CODE.BOX.SK

http://code.box.sk

2573 visitas | Puntuación: ■■■■ (1 voto) Vota

Un buen lugar para programadores, con foros, enlaces y tutoriales. Especialmente interesantes algunas de sus secciones, como la de PHP.

Programmer's heaven

http://www.programmersheaven.com/

3304 visitas | Puntuación: ■■■■ (9 votos) Vota

Una excelente recopilación de enlaces (unos 2400) y ficheros de utilidad para el programador (ejemplos, herramientas) totalmente gratuitos.

Listas de correo

1 dirección

[Subir]

Latium Software

http://www.latiumsoftware.com

1016 visitas | Puntuación: ■■■■ (2 votos) Vota

El principal interés de esta web son sus boletines. Posee de Pascal (Delphi, Kylix), Visual Basic, y uno genérico para todo tipo de programadores.

Principio Página

Contacto - Datos legales

Direcciones.
Varios.
Gráfica

5 direcciones

Direcciones sobre programación gráfica, librerías, 3D, etc.. Sin restricción de lenguajes utilizados.

Nuestras zonas

»EN ESTA PAGINA

»IMPRESCINDIBLE VISITAR

En castellano

- Programación Gráfica
- 3Dup.com - El Portal del Diseño 3D y Multimedia

Generales

3 direcciones

[Subir]

El mundo del caos

http://www.elmundodelcaos.tk/

1551 visitas | Puntuación: ■■■■ (4 votos) Vota

Página dedicada a la programación gráfica en C y C++, con especial hincapié en la programación de fractales (funciones matemáticas en variable compleja).

3Dup.com - El Portal del Diseño 3D y Multimedia

http://es.3dup.com

5446 visitas | Puntuación: ■■■■■ (46 votos) Vota

Versión en Español del Portal especializado en Diseño 3D y Multimedia que cuenta con Motor de Búsqueda, Noticias, Webs y Correo Gratuitos, Forums, Chats, Tienda Online, Galería de Artistas y Zona de Download con Texturas, Modelos 3D y Plugins.

Programación Gráfica

http://www.geocities.com/valcoey/index.html

8478 visitas | Puntuación: ■■■■■ (9 votos) Vota

Completa página con numerosos artículos sobre programación gráfica en VisualC++, OpenGL y Visual Basic. También examina a fondo la teoría de fractales, gráficos 2D y 3D y las matemáticas necesarias para poder entender todo lo demás, aunque sea poco.

Cursos

2 direcciones

[Subir]

Breve Tutorial de programación con Motif

http://www.compulinux.com/diego/data/110/gm_indice.html

2114 visitas | Puntuación: ■■■■ (2 votos) Vota

Breve Tutorial de programación con Motif con ejemplos y textos sencillos y didácticos.

Tutorial de 3D

http://usuarios.maptel.es/josecpujol/

4983 visitas | Puntuación: ■■■■ (10 votos) Vota

Buen tutorial sobre los fundamentos de la programación en 3D. Aunque pueda echar para atrás el hecho de que los ejemplos y programas estén hechos en Turbo Pascal, conviene echarle un vistazo para entender cómo funciona a bajo nivel la programación en 3D.

Principio Página

Los últimos enlaces

En esta página podrás acceder a los últimos enlaces añadidos a nuestra base de datos.

Nuestras zonas

PHP y funciones de FTP NUEVO
http://programacion.com/php/articulo.fli_phpftp.html
En Internet / PHP
143 visitas | Puntuación: ■■■■■ (3 votos) Vota
Articulo en castellano donde se explica como realizar FTP desde aplicaciones en PHP.

Tutorial de XPath. Ver. 1.0 NUEVO
http://geneura.ugr.es/~victor/cursillos/xml/XPath/
En Internet / XSL, XSLT y Xpath
139 visitas | Puntuación: ■■■■ (2 votos) Vota
Tutorial introductorio a la tecnología XPath.

Generación de páginas Web usando XSLT y XML NUEVO
http://geneura.ugr.es/~jmerelo/XSLT/
En Internet / XSL, XSLT y Xpath
324 visitas | Puntuación: ■■■■ (2 votos) Vota
Pequeño tutorial sobre XSLT y cómo generar webs a partir de ficheros XML usando Saxon o Xalan.

Web de fLIPIS
http://www.flipis.net
En Internet / PHP
694 visitas | Puntuación: ■■■■ (1 voto) Vota
La página del encargado de PHP en castellano; consta principalmente de artículos y tutoriales sobre PHP y tecnologías afines.

IndicePERL
http://lmsaizarroba.tripod.com/IndicePERL.html
En Lenguajes de script / Perl
818 visitas | Puntuación: ■■■ (2 votos) Vota
Pequeña página donde se indica donde conseguir PERL y se dan algunos ejemplos de uso.

PHP para torpes
http://php-hispano.net
En Internet / PHP
2297 visitas | Puntuación: ■■■■■ (20 votos) Vota
Web del canal #php_para_torpes del IRC-Hispano. Dispone de tutoriales, scripts y foros, con contenidos propios. Desafortunadamente hay que registrarse para acceder.

Curso de Haskell de Jeroen Fokker

http://www.cs.uu.nl/people/jeroen/courses/fp-sp.pdf

En Otros lenguajes / Haskell

1004 visitas | Puntuación: ■■■■ (5 votos) Vota

Curso empleado en las clases de la Universidad de Utrecht. Aún cuando se refiera a Gofer, en realidad el lenguaje es Haskell, que por lo visto tiene hasta apodos. Es muy sencillo y está muy bien traducido.

Sistemas Operativos

http://exa.unne.edu.ar/depar/areas/informatica/SistemasOperativos/SOF.htm

En Sistemas operativos

2053 visitas | Puntuación: ■■■ (3 votos) Vota

Completo libro online cuyos contenidos corresponden a un curso universitario de Sistemas Operativos Convencionales y Distribuidos. Se puede descargar el curso en PDF. La página principal, no obstante, está sobrecargada a la extenuación.

Tutorial HTML & CSS

http://www.xiniom.com/users/Jorge/html/

En Internet / HTML

3699 visitas | Puntuación: ■■■■■ (12 votos) Vota

Extenso tutorial enfocado a las nuevas versiones de HTML y CSS (hojas de estilos). Liberado bajo los términos de la Licencia de Documentación Libre GNU (LDLG o GFDL en inglés).

Planeta Silius

http://welcome.to/planetasilius

En Varios / Personales

366 visitas | Puntuación: ■■■■ (3 votos) Vota

Dispone de varios programas creados por el autor, historia de algunos lenguajes, compiladores, tutoriales en ZIP, criptografía, ejemplos de código, prácticas de su carrera, etc.

Principio Página

New 2 Java: Construir una Aplicación: 4.- Leer y Escribir Ficheros y Manejar Excepciones

Autor: Sun
Traductor: Juan Antonio Palos (Ozito)

Puedes encontrar la Version Original en Ingles en (http://java.sun.com)

Indice de contenidos

Operaciones avanzadas con JDBC y Java

Autor: IBM
Traductor: Juan Antonio Palos (Ozito)

Puedes encontrar la Version Original en Ingles en (http://ibm.com/developerworks)

Indice de contenidos

Leer comentarios (5) | Escribir comentario | Puntuación: ■■■■■ (3 votos) Vota

Últimos comentarios

[Subir]

Imprimir en Java (19/10/2002)
Por Juan Antonio Palos (Ozito)

Hola, en el tutorial http://programacion.com/java/tutorial.2d.html podrás encontrar un capítulo completo sobre impresión en Java

todo bueno pero falta.... (17/10/2002)
Por omar velez

He sido fiel a leer los tutoriales de OZITO y me parecen de lo mejor y han sido parte de una buena programacion en java, pero falta, falta un manual completo de impresiones en java, para poder programar un reportes 10/10

programas (15/10/2002)
Por myriam

me gusto su pagina sobre este programa, quisiera por favor si me podrian mandar algunos programas en java... gracias

programas (15/10/2002)
Por myriam

me gusto su pagina sobre este programa, quisiera por favor si me podrian mandar algunos programas en java... gracias

Muy interesante (09/10/2002)
Por Manuel

Gracias. El tutorial me ha resultado muy util.

Recomendar este tutorial | Estadísticas

Modelo relacional

Autor: Claudio Casares

Leer comentarios (3) | Escribir comentario | Puntuación: ■■■■■ (4 votos) Vota
Recomendar este tutorial | Estadísticas

Indice de contenidos

Leer comentarios (3) | Escribir comentario | Puntuación: ■■■■■ (4 votos) Vota

Últimos comentarios

[Subir]

BUSCO EN TODA LA WEB (15/10/2002)
Por Cansada

Por que no pueden hacer un tutorial completo donde se pueda encontrar definiciones que son básicas.
Que es transaccionalidad??
Que tipo de info contiene un archivo de un a base de datos??

A medias (07/10/2002)
Por Daniel

Si, MySQL sigue el modelo relacional, pero no implementa muchas funciones típicas de los mismos, como puedan ser las claves extranjeras.

MySQL (04/10/2002)
Por Angel

¿ MySQL es una base ralacionada?
¿Hay algun libro en el mercado sobre el tema?

Gracias

Recomendar este tutorial | Estadísticas

Principio Página

Curso avanzado de Prolog

Autor: Angel Fernández Pineda

Indice de contenidos

Principio Página

Manejar Conexiones a Bases de Datos con JDBC 3.0

Autor: IBM
Traductor: Juan Antonio Palos (Ozito)

Indice de contenidos

Leer comentarios (22) | Escribir comentario | Puntuación: ■■■■ (37 votos) Vota

Últimos comentarios

[Subir]

Applet (10/10/2002)
Por Desperado

Hola,
He estado siguiendo este manual y todo perfecto hasta que he querido probar el acceso a la bbdd mediante un applet.
Alguien se ha encontrado con ese problema??
Me puede ayudar??
Gracias por adelantado.

Dudas (03/10/2002)
Por Raymundo Galvan Nieves

Trabajo en el area de operacion de sistemas, quiero desarrollar algo para llevar ciertos controles ante los requerimientos y necesidades de los usuarios, tipo un helpdesk, no quiero invertir mucho tiempo en programacion, JAVA me puedes servir ?, JAVA maneja base de datos o tengo que interactuar con otra que exista en el mercado? que tan facil y rapido es programar con JAVA?, que requiero en determinado momento para iniciar ?, donde lo consigo ?.
Gracias y disculpa tantas preguntas.

Excelente (06/09/2002)
Por Galo

Muy bueno pero seria formidable aun mas con una pequeña aplicacion
gracias de todas maneras

Bastante completo (29/08/2002)
Por Ignacio Alcázar Contell

Podría mejorarse con ejemplos más concretos. No sé si hay algún tutorial de JDBC 2.0, pero si no es así también mejoraría con los tipos de ResultSet, llamada a procedimientos almacenados, sentencias preparadas, etc

Muy agradecido (28/08/2002)
Por Jorge

Hola soy Jorge desde Argentina y quiero decirles que soy nuevo en el asunto de la creación y diseño de páginas web, y sus trutoriales me han sido de gran ayuda. Espero que sigan publicándolos y que ayuden a muchas personas como yo que estamos interesadas en aprender alg más.
Desde Argentina los felicito y les digo: "sigan adelante"

Recomendar este tutorial | Estadísticas

Curso de XML

Autor: Joaquin Bravo Montero

Indice de contenidos

Leer comentarios (33) | Escribir comentario | Puntuación: ■■■■■ (50 votos) Vota

Últimos comentarios

[Subir]

Bueno (18/10/2002)
Por THEMASTER

Hasta ahora empecé vamos a ver si aprendo alguito y puedo ganar dinero con este nuevo conocimiento.

Buenisimo (10/10/2002)
Por Jorge

Esta bueno ...

Alguna pega tenia q tener (09/10/2002)
Por Freddy

Me uno a los comentarios anteriores,deberias tener un .pdf .zip o algo parecido poder descargar este manual.Respecto al manual me parece muy bueno para la gente q no sabe nada d XML,es una buena referencia.

muy bueno (04/10/2002)
Por alfonso

esta muy bueno pero hace falta la opción de carga.
si lo hicieran seria excelente.

Agregar Opción (02/10/2002)
Por Luis Jose

Por que no agregan una opcion para la descarga completa *.zip o un pdf., luego todo esta excelente.

Recomendar este tutorial | Estadísticas

Principio Página

New 2 Java: Suplementos

Autor: Sun
Traductor: Juan Antonio Palos (Ozito)

Puedes encontrar la Version Original en Ingles en (http://developer.java.sun.com/developer/onlineTraining/new2java/supplements/)

Indice de contenidos

Leer comentarios (5) | Escribir comentario | Puntuación: ■■■■ (27 votos) Vota

Últimos comentarios

[Subir]

DESCARGAR JAVA (19/10/2002)
Por ION MAÑAS

NECESITO DESCARGARME EL JAVA.ME LE BAJO YO Y SIEMPRE ME DA ERROR.TENGO WINDOWS XP.GRACIAS.

Muchas Gracias (09/10/2002)
Por Raúl

Quiero dar las GRACIAS a las personas que están dedicando su tiempo a la elaboración de todo este material. Su trabajo es IMPRESIONANTE. MUCHAS GRACIAS por su esfuerzo y espero que sigan mucho más tiempo en esta labor.

como asignar un valor a una variable (04/10/2002)
Por felipe

estoy conociendo java y me parece muy interesante
pero no e podido saber como capturo un valor de teclado, me gustaria saberlo..... gracias

cuando estará disponoble la 4ta parte (14/09/2002)
Por hernan

hasta ahora me a parecido un exelente tutorial(el mejor),pero quiero saber porqué se estan tomando tanto tiempo en la 4ta parte de este tutorial donde se implementan las demas clases faltantes de la aplicacion divelog

URGENTE (11/09/2002)
Por andrew

necesito saber en que pagina puedo descargar java
para windows es de caracter URGENTE

GRACIAS.....

Recomendar este tutorial | Estadísticas

Curso práctico de Corba en GNU/Linux

Autor: Alvaro del Castillo

Indice de contenidos

- Por dónde seguir
- Conclusiones

Leer comentarios (13) | Escribir comentario | Puntuación: ■■■■■ (11 votos) Vota

Últimos comentarios

[Subir]

tienen razon (27/09/2002)
Por martin

por favor... hagan un zip o pdf de este tutorial

Hagan un pdf o zip (18/09/2002)
Por Manu G.

Por favor, pasen el curso a pdf o comprímanlo en zip.

Sobre Corba (30/08/2002)
Por sergio

¿Por que dices que corba tiene los dias contado?
no cuestiono tu comentario por molestar si no para saber si vale la pena estudiarlo o no. O hicistes el comentario solo por molestar. por lo que te pido, que si alguno de ustedes tiene argumentos mas solidos sobre el futuro de corba, podria ponerlos como comentarios, para saber si vale la pena estudiarlo.

Verdad (17/07/2002)
Por freddy rente

Es Verdad debe hacer un .zip o un pdf gracias por la infor.
Suerte

Adhiero al resto (16/07/2002)
Por Diego

No sean tan haraganes y hagan un zip o pdf.

Recomendar este tutorial | Estadísticas

Principio Página

XML Schema y DTDs

Autor: Advanced Quality Solutions

Leer comentarios (2) | Escribir comentario | Puntuación: ■■■■ (6 votos) Vota

Recomendar este tutorial | Estadísticas

Indice de contenidos

Leer comentarios (2) | Escribir comentario | Puntuación: ■■■■ (6 votos) Vota

Últimos comentarios

[Subir]

quiero cursos (05/06/2002)
Por inmaculada

No hay alguna forma de descargarse los cursos completos?

Endavant! (16/04/2002)
Por Mercè Vázquez

Endavant, nois! Això està molt bé.

Recomendar este tutorial | Estadísticas

Principio Página

Contacto - Datos legales

Programación en castellano

Documentacion | Direcciones | Taller PHP

Novedades y más información en nuestra lista de correo

Secciones

- Tutoriales
- Taller PHP
- Recursos
- Formación
- Buscador
- Downloads
- Foros

Tutoriales

- Tutoriales propios

Taller PHP

- Artículos propios

Foros

- Foro PHP

Premio IBEST2001 a la mejor página personal

En nuestro sitio

Búsqueda avanzada | Buscar

Documentación propia

Tutoriales | Taller PHP

SMTP utilizando Sockets en PHP

Por: Alejandro Almunia

Vamos a ver, paso a paso, la creación de una clase que nos permita mandar sencillos correos de texto (sin ficheros adjuntos), usando PHP y las funciones de sockets que lleva incorporadas. Asimismo, usaremos comandos SMTP para comunicarnos con el servidor de correo.Publicado el *17 de Octubre 2002*

PHP y funciones FTP

Por: Alejandro Almunia

Entre las casi innumerables librerías de PHP, disponemos de una que nos permite conectarnos por FTP. Estudiaremos las funciones de dicha librería en este nuevo artículo.Publicado el *9 de Octubre 2002*

Trabajar con PHP y ficheros

Por: Alejandro Almunia

En este artículo se nos explica como trabajar con ficheros desde PHP. Como podemos escribirlos, leerlos, subirlos a la web, etc.Publicado el *10 de Septiembre 2002*

Paginar los resultados de una consulta en PHP (II)

Por: Daniel Rodriguez Herrera

Explicamos otra manera de paginar empleando una sola consulta y también indicamos como reducir el número de enlaces a otras páginas, cuando tenemos muchos registros.Publicado el *20 de Agosto 2002*

Paginar los resultados de una consulta en PHP

Por: Daniel Rodriguez Herrera

Realizar una consulta a una base de datos MySQL desde PHP y presentarla es sencillo. El problema viene cuando debemos presentar 3000 registros; para solucionarlo utilizamos la paginación.Publicado el *3 de Abril 2002*

Como interactuar con una base de datos MySQL usando PHP

Por: Agustín Dondo

Realizar una consulta a una base de datos MySQL desde PHP y presentarla es sencillo. El problema viene cuando debemos presentar 3000 registros; para solucionarlo utilizamos la paginación.Publicado el *2 de Marzo 2002*

Tutoriales | Taller PHP

Registrate

Nombre:

Password:

Enviar

Ultimos Tutoriales

Bases de datos en la Web

Webs dinámicos con PHP

Tutorial de PHP y MySQL

Los mejores libros en:

Recursos

Enlaces a direcciones en las que encontrara recursos y aplicaciones que haran más fácil la elaboración de tus aplicaciones utilizando PHP.

Más

Principio Página

Mantenida por: Alejandro y Daniel.
PHP en castellano.
Estadísticas en:

Zona de download

Nuestras zonas

Recopilación de los tutoriales, artículos, etc. que están en nuestra Web y que se pueden descargar en formato zip. Para poderlos descargar **es necesario entrar como usuario** registrado.

Artículos 2000 (01/02/2001)

http://www.programacion.net/download.php?id=1

55415 descargas (595277 kb)

Todos los artículos que hemos publicado a los largo del 2000 en un único fichero PDF. Incluye tanto los artículos de la zona principal como de cada una de las zonas.

Online en: http://www.programacion.com

Curso de Javascript 1.2 (19/02/2001)

http://www.programacion.net/download.php?id=2

41349 descargas (83129 kb)

Curso de Javascript 1.2 de HTML en castellano, en formato HTML comprimido en un fichero ZIP.

Online en: http://html.programacion.net/js/

Curso de HTML 4.0 (19/02/2001)

http://www.programacion.net/download.php?id=3

62583 descargas (102038 kb)

Curso de HTML 4.0 de HTML en castellano, en formato HTML comprimido en un fichero ZIP.

Online en: http://html.programacion.net/curso/

Curso de XHTML 1.0 (19/02/2001)

http://www.programacion.net/download.php?id=4

16072 descargas (41304 kb)

Curso de XHTML 1.0 de HTML en castellano, en formato HTML comprimido en un fichero ZIP.

Online en: http://html.programacion.net/xhtml/

Introducción a ASP (30/03/2001)

http://www.programacion.net/download.php?id=5

28575 descargas (31661 kb)

Curso de inicación a la tecnología ASP: sus requisitos técnicos, los tipos de elementos que componen las páginas ASP y su correcta utilización.

Online en: http://asp.programacion.net/tutoriales/asp_basics

VBSCRIPT (30/03/2001)

http://www.programacion.net/download.php?id=9

21481 descargas (140835 kb)

Visual Basic Script es el lenguaje de script con el que se escriben las páginas ASP. Aprende su sintaxis en este curso.

Online en: http://asp.programacion.net/tutoriales/vbscript/

ASP y WAP (30/03/2001)
http://www.programacion.net/download.php?id=8

16345 descargas (283124 kb)

Implementación de una aplicación WAP utilizando tecnología ASP.

Online en: http://asp.programacion.net/tutoriales/aspywap/

Servlets y JSP (13/05/2001)
http://www.programacion.net/download.php?id=12

28704 descargas (689991 kb)

Los Servlets son las respuesta de la tecnología Java a la programación CGI. Son programas que se ejecutan en un servidor Web y construyen páginas Web. Java Server Pages (JSP) es una tecnología que nos permite mezclar HTML estático con HTML generado dinámicamente mediante codigo java.

Online en: http://java.programacion.net/servlets_jsp

Api Java Mail (01/06/2001)
http://www.programacion.net/download.php?id=13

17024 descargas (109708 kb)

Introducción al API JavaMail para enviar y recibir e-mails desde programas Java

Online en: http://java.programacion.net/javamail/

Tutorial de introduccion a Java (07/05/2001)
http://www.programacion.net/download.php?id=10

33515 descargas (64439 kb)

Tutorial de introducción a Java.

Online en: http://java.programacion.net/intjava/

Java: Programación en Red (13/05/2001)
http://www.programacion.net/download.php?id=11

21688 descargas (43881 kb)

Tutorial sobre programación en Red utilizando Java.

Online en: http://java.programacion.net/

Curso de FrontPage 2000 (05/06/2001)
http://www.programacion.net/download.php?id=15

16790 descargas (1013206 kb)

Curso completo de creación de páginas web con FrontPage 2000, en formato Word.

Online en: http://html.programacion.net/frontpage/

Primera taza en Java (23/06/2001)
http://www.programacion.net/download.php?id=16

32589 descargas (146 kb)

Fichero PDF de una presentación muy gráfica de los primeros pasos en Java para las plataformas Windows, UNIX/Linux y MAC OS.

Online en: http://java.programacion.net/primera_taza/index.php

TutorJava nivel básico (23/06/2001)
http://www.programacion.net/download.php?id=17

38768 descargas (389 kb)

Fichero PDF de un acercamiento a los conceptos básicos sobre Objetos, Clases e Interfaces, así como la explicación de los principales paquetes Java.

Online en: http://java.programacion.net/java_basico/index.php

Manejo de Errores Utilizando Excepciones Java (23/06/2001)
http://www.programacion.net/download.php?id=18

23606 descargas (132 kb)

Introducción a las excepciones en el lenguaje Java. Manejo y control de errores. En PDF.

Online en: http://java.programacion.net/excepciones/index.php

Trabajo en Red (04/09/2001)
http://www.programacion.net/download.php?id=19

15440 descargas (273 kb)

Inicio a la programación en Red en el lenguaje Java, incluyendo Sockets, Datagramas, etc. en formato PDF con los ficheros fuente adjuntados

Online en: http://java.programacion.net/red/index.php

Servlets (04/09/2001)
http://www.programacion.net/download.php?id=20

21361 descargas (219 kb)

Inicio a la programación de servlets en lenguaje Java, incluyendo las partes básicas para comunicar programas cliente/servidores. Formato PDF con los ficheros fuente adjuntados

Online en: http://java.programacion.net/servlets_basico/index.php

Java Native Interface (JNI) (04/09/2001)
http://www.programacion.net/download.php?id=21

11943 descargas (220 kb)

Explicación de como comunicar programas escritos en Java con programas escritos en otros lenguajes como C++... Formato PDF con los ficheros fuente adjuntados

Online en: http://java.programacion.net/jni/index.php

Invocación Remota de Métodos (RMI) (04/09/2001)
http://www.programacion.net/download.php?id=22

12059 descargas (117 kb)

Explica el modo de llamar a métodos de otras aplicaciones que se están ejecutando en otras máquinas de la red. (programación distribuida)

Online en: http://java.programacion.net/rmi/index.php

Beans (básico) (04/09/2001)
http://www.programacion.net/download.php?id=23

15288 descargas (197 kb)

Introducción a los componentes reutilizables (Beans) en Java

Online en: http://java.programacion.net/beans/index.php

Gráficos en Java 2D (04/09/2001)
http://www.programacion.net/download.php?id=24

20699 descargas (856 kb)

Programación de gráficos en 2D, utilizando los nuevos APIs que la plataforma Java presenta a partir del JDK 1.2.

Online en: http://java.programacion.net/2d/index.php

Principio Página

Cursos

Nuestras zonas

108 cursos

Las mejores plumas (bueno... teclados) escriben para Programación en castellano sobre temas de interés para el programador.

- Bases de datos (5)
- Entornos de desarrollo (1)
- Herramientas (2)
- Internet (26)
- Lenguajes de script (4)
- Lenguajes orientados a objeto (65)
- Otros lenguajes (3)
- Sistemas operativos (1)
- Teoría (1)

Principio Página

Artículos

Nuestras zonas

107 artículos

Las mejores plumas (bueno... teclados) escriben para Programación en castellano sobre temas de interés para el programador.

- Bases de datos (2)
- Entornos de desarrollo (2)
- Herramientas (2)
- Internet (50)
- Lenguajes de script (23)
- Lenguajes imperativos (1)
- Lenguajes orientados a objeto (27)

Principio Página

Formación en la Red

Nuestras web

Desde esta página podras acceder a cursos de formación en informática y nuevas tecnologías.

CENTROS COLABORADORES

Ciberaula | BIT

Ciberaula

Ciberaula es una empresa dedicada a la formación on-line en informática y nuevas tecnologías en España e Hispanoamérica. Por ser usuario **registrado** de Programación en castellano obtendras 101 euros de descuento en los siguentes cursos:

- PHP - MySQL - Comercio electrónico
- ASP - SQL - Comercio electrónico
- XML
- Webmaster Nivel I
- Photoshop 6.0

Condiciones de la oferta

BIT

CARRERAS 2002

PRECIOS ESPECIALES
para las inscripciones anteriores al 31 Diciembre 2001
Ahórrese hasta 84.000 Pts.

- Analista Programador de aplicaciones informáticas
- Analista Programador en entorno Visual
- Analista Programador en entorno Internet (JAVA y C#)
- Técnico en Microinformática
- Técnico en Microinformática
- Diseñador Web Site
- Programador Web Site (ASP)
- Análisis y diseño estructurado + UML
- Responsable de Xarxa
- Gestor de Telecomunicaciones

Condiciones de la oferta

Principio Página

Foros de debate

Nuestras zonas

Buscar: Dale

Estos son los foros disponibles en Programación en castellano:

Foro	Mensajes	Último mensaje
ASP	1929	19 de octubre de 2002
Bases de datos y SQL	655	18 de octubre de 2002
C / C++	372	19 de octubre de 2002
Delphi / Kylix / C++ Builder	203	19 de octubre de 2002
General	316	17 de octubre de 2002
HTML	1001	17 de octubre de 2002
Java (básico)	2515	20 de octubre de 2002
Java (Servlets y JSP)	902	19 de octubre de 2002
Java y XML	181	17 de octubre de 2002
Javascript	719	10 de octubre de 2002
Petición de foros nuevos	227	19 de octubre de 2002
PHP	645	19 de octubre de 2002
Servidores de Aplicaciones J2EE	101	18 de octubre de 2002
Visual Basic	473	20 de octubre de 2002
Visual FoxPro	153	16 de octubre de 2002
XML	180	17 de octubre de 2002

Que los disfrutes.

Principio Página

Taller PHP. SMTP utilizando Sockets en PHP

Autor: Alejandro Almunia

El primer script usando sockets y SMTP

Comenzaremos por mostrar como se usa la función fsockopen(), porque la vamos a necesitar antes de conectarnos al servidor SMTP. Esta función toma dos argumenstos más tres opcionales, pero a nosotros solo nos interesan los dos obligatorios. El primero es la IP o nombre del servidor al que conectar, y el segundo es el puerto. Así, nosotros, usaremos el puerto 25, y, como servidor, uno que conozcamos que funcione.

Así pues, primero vamos a ver un ejemplo de como abrir un socket, y luego encapsularemos en una clase las funciones que vamos a necesitar en la clase a crear. En este sencillo ejemplo, solo vamos a conectar a un servidor de correo y decirle "Hola", en su lenguaje. El código es como sigue:

```
<?

//Abrimos la conexión
$Conexion = fsockopen("programacion.com","25");

//Sacamos el contenido de lo que nos devuelve el servidor antes de seguir
$str_recibe = fgets($Conexion,256);
        echo "<pre>";
        print_r($str_recibe);
        echo "</pre>";


//Esta es la cadena que dice "Hola". Fíjate en en \r\n final que marca
//un "Enter", para confirmar el comando
$str_hola = "EHLO programacion.com\r\n";

//Le soltamos la cadena al socket
if(!fputs($Conexion,$str_hola))
{
        echo "Imposible decirle \"hola\" al servidor.";
        exit;
}
else
```

```
{

        $str_recibimos = fgets($Conexion,256);
        echo "<pre>";
        print_r($str_recibimos);
        echo "</pre>";

}

//Salimos de la sesión, primero diciendo "Adios" al servidor
//y luego cerrando el socket con fclose()
if(!fputs($Conexion,"QUIT\r\n"))
{
        echo "No se pudo salir de la sesión con el servidor SMTP";
        exit;
}
else
{
        fclose($Conexion);
        echo "Cerrado y terminado";
        exit;
}


?>
```

Como vemos, no es tan complicado como parece. A continuación vamos a mostrar una lista de comandos SMTP que nos serán imprescindibles a la hora de tratar con un servidor de correo. Asimismo, veremos los distintos códigos que nos puede devolver el servidor.

Comandos SMTP y respuestas del servidor

A continuación, muestro una lista de comandos SMTP junto con su significado.

Comando	Significado
HELO	Hola sencillo. Es el modo más antiguo y seguro de decir "hola". Si el servidor es capaz de responder a él, soporta las funcionalidades básicas que vamos a ver en este tutorial.
EHLO	Hola extendido. Es el modo de decirle hola al servidor más moderno. Si es capaz de responder, significa que posee muchas más características que si solo pudiese responder a HELO. Muchas de ellas son muy avanzadas y no las vamos a ver en este tutorial.
MAIL FROM: correo@dominio	Con este comando se especifica de quién proviene el correo a enviar. A menudo el dominio ha de ser uno válido, pues ciertos servidores comprueban la existencia del mismo, devolviendo un mensaje de error si no pueden ser identificados.

RCPT TO: correo@dominio	Con este comando se especifica el receptor (o receptores, pues el comando puede ser repetido tantas veces como receptores haya), del mensaje enviado. Es importante tener en cuenta que ciertos servidores poseen una tabla de dominios a los que pueden entregar el correo. Por ejemplo, mucha gente configura el servidor para que solo pueda admitir correo para la máquina, rechazando todas las otras direcciones cuyos registros DNS no se encuentren en la misma. Como regla general, es seguro usar el servidor de correo del dominio al cual pertenece el correo de la persona a la que va dirigida el mensaje.
DATA\r\n (Subject: \r\n) \r\n.\r\n	Este comando, pese a su apariencia compleja, no lo es tanto en realidad. Primero va la palabra clave DATA, que indica al servidor de correo que se inicia la transferencia de los datos del correo en si. Tras esta sentencia, y, antes del correo, podemos incluir cabeceras, siendo la más común Subject: , para indicar el tema. Tras todo el mensaje, debe escribirse **\r\n.\r\n**. Si eso es. Un "Enter", seguido de un punto y otro "Enter". Así se le dice al servidor que el mensaje ha terminado. Una vez pulsemos el último "Enter", el mensaje será enviado a la cola de proceso, listo para su entrega.

Pues ya ves que no son tantos. En realidad solo cinco, para hacer el trabajo más básico. Con esto se pueden mandar e-mails en formato de texto simple, sin ficheros adjuntos. No es mucho, pero, para estar trabajando a este nivel, ya es.

A continuación, una lista de los códgos numéricos más usuales de un servidor SMTP.

Código	Significado
502	No implementado. Es el error que devuelve cuando uno de los comandos no está implementado en el servidor. Por ejemplo, programacion.com responde 502 a EHLO.
250	Todo correcto. El comando introducido ha funcionado correctamnte y la acción requerida ha sido realizada sin problemas. También es lo que devuelve el servidor cuando un mensaje ha sido enviado correctamente a la cola de proceso.
553	El Recipiente del mensaje (RCPT TO:), no está incluido en la lista de dominios a los que el servidor esté configurado para entregar el correo.
354	Sigue adelante enviando el mensaje. Esto es lo que devuelve el servidor tras introducirle correctamente la palabra DATA.
221	Significa "Adios" en la jerga SMTP

Y, para demostrar que todo lo anterior es cierto, aquí está la transcripción de mi sesión telnet con el servidor de programacion.com

```
220 us-8.34web.com ESMTP
EHLO programacion.com
502 unimplemented (#5.5.1)
HELO programacion.com
250 us-8.34web.com
```

```
MAIL FROM:flipis@flipis.net
250 ok
RCPT TO:multivac@internautas.org
553 sorry, that domain isn't in my list of allowed rcpthosts (#5.7.1)
RCPT TO:info@programacion.com
250 ok
DATA
354 go ahead
Subject: Esto es una prueba del webmaster de la seccion de php fLIPIS
Este es el mensaje de prueba
.
250 ok 1033728059 qp 1944
quit
221 Goodbye
```

Escribiendo la clase SMTP

Llegados a este punto, hemos visto cual es la secuencia de comandos necesaria para mandarle al servidor lo que nos hace falta para enviar un correo sencillo, de solo texto. Dado que el código de la clase es considerablemente largo, no lo incluyo aquí. Descargadlo en el sguiente link. Hay dos scripts, uno de ejemplo y otro que es la clase en si.

Descargate los ficheros del artículo

Mantenida por: Alejandro y Daniel.

Taller Java. Desplegar Servlets y Aplicaciones Web en Tomcat y WebLogic Server

Autor: Sun
Traductor: Juan Antonio Palos (Ozito)

Introducción

En este artículo revisaremos los pasos implicados en el despliegue de un servlet, describe cómo tomar un servlet y crear una aplicación Web - tanto en formato expandido como en un WAR. Ilustra cómo desplegar una aplicación Web en Apache Tomcat y en WebLogic Server 6.0, un completo servidor de aplicaciones J2EE.

Empezaremos con una breve recapitulación sobre los fundamentos del desarrollo de Servlets, luego mostraremos como construir una aplicación Web para contenerlos. Explicaremos el uso de **Web Application Archives** (WARs), y luego veremos como desplegar una aplicación web en los entornos anteriormente nombrados.

Desarrollo de Servlets

Los servlets fueron diseñados para permitir la extensión de un servidor proporcionando cualquier servicio. Sin embargo, actualmente sólo se soportan HTTP y páginas JSP. En el futuro, un desarrollador podría extender un servidor FTP o un servidor SMTP usando servlets.

Servlets Genéricos

Un servlet amplía las funcionalidades de un servidor ofreciendo un servicio específico dentro de un marco de trabajo bien definido. Es una pequeña pieza de código Java - normalmente una sóla clase -- que porporciona un servicio específico. Por ejemplo, un servlet HTTP podría proporciona a un cliente de un banco los detalles de sus depositos y reintegros recientes. Otro servlet HTTP podría permitir a un cliente, ver, e incluso editar su dirección de correo.

Para desplegar un servlet, normalmente se requiere la configuración de un servidor de aplicaciones. Cuando el servidor encuentra un tipo particular de solicitud, invoca al servlet, pasándole los detalles sobre la solicitud y un objeto `response` para devolver el resultado.

Todos lo servlets implementan el interface `javax.servlet.Servlet` bien directamente -- en el caso de los servelts genéricos -- o indirectamente, en el caso de los servlets HTTP o JSP. El interface `javax.servlet.Servlet` incluye los siguientes métodos importantes:

- `init()`:
  Define cualquier código de inicialización que debería ejecutarse cuando se carga el servlet en memoria.
- `service()`:
  El método principal, llamado cuando el servlet recibe una solicitud de servicio. Define un paquete de lógica de procesamiento proporcionado por el servlet.
- `destroy()`:
  Define cualquier código de limpieza requerido antes de eliminar el servlet de la memoria.

Cuando el contenedor servlet carga por primera vez un servlet invoca al método `init()` del servlet para inicializarlo. Luego según se hacen solicitudes para ejecutar el servlet, el contenedor servlet llama repetidamente la método `service()` del servlet. Finalmente, cuando el contenedor servlet no necesita el servlet, llama al método `destroy()` del servlet y lo descarga de la memoria. Observa que durante el tiempo de vida de un simple ejemplar servlet, los métodos `init()` y `destroy()` sólo son invocados una vez, mientras que el método `service()` será invocado muchas veces -- una cada vez que se haga una solicitud para ejecutar el servlet.

Servlets HTTP

Los servlets HTTP extienden la clase `javax.servlet.http.HttpServlet`. Esta clase extiende la clase `javax.servlet.GenericServlet`, que a su vez implementa `javax.servlet.Servlet`. La clase `HttpServlet` sobreescribe el método `service()` de forma que puede manejar diferentes tipos de solicitudes HTTP: **DELETE**, **GET**, **OPTIONS**, **POST**, **PUT**, y **TRACE**. Por cada uno de estos tipos de solicitud, la clase `HttpServlet` proporciona su correspondiente método `doXXX()`.

Aunque podemos sobreescribir el método `service()` en nuestra clase servlet, raramente hay alguna necesidad de hacerlo. Más frecuentemente querremos sobreescribir métodos `doXXX()` individuales. Si sobreescribimos el método `service()`, debemos tener cuidado de que los métodos `doXXX()` por defecto, sólo sean llamados si llamamos a `super.service` o los invocamos directamente.

Para la mayoría de las aplicaciones querremos sobreescribir los métodos `doPost()` y `doGet()`, ya que ellos manejan normalmente los datos enviados por un formulario de usuario desde un FORM HTML.

Para sumarizar, cuando escribamos nuestros servlets HTTP, deberíamos:

1. Importar como mínimo las clases servlets:

- javax.servlet.ServletException
- javax.servlet.http.HttpServlet
- javax.servlet.http.HttpServletRequest
- javax.servlet.http.HttpServletResponse

2. Hacer la clase public
3. Hacer que la clase extienda HttpServlet
4. Sobreescribir los métodos doXXX() apropiados para implementar nuestra lógica de solicitud/respuesta..

Un Servlet de Ejemplo: RequestDetails

En el ejemplo de abajo hemos ilustrado un simple servlet HTTP. La primera línea simplemente define a qué paquete pertenece el servlet. El siguiente bloque de código importa las clases usadas en este servlet. Luego viene la definición de la clase servlet. Como puedes ver, la clase RequestDetails extiende HttpServlet.

El cuerpo de RequestDetails define dos métodos: doGet() y doPost(). El método doGet() define la funcionalidad principal de este servlet. El método doPost()simplemente llama a doGet(). Por lo tanto, el servlet maneja las peticiones **GET** y **POST** de la misma forma.

El método doGet() construye una página HTML que contiene detalles sobre la solicitud HTTP enviada al servidor. Observa las dos primeras líneas del método. La primera línea selecciona el tipo de contenido de la respuesta. En general, construiremos una página HTML, en cuyo caso el tipo de contenido debe configurarse como text/html. La segunda línea del método doGet() obtiene una referencia a un stream de salida PrintWriter. Toda la salida a devolver para el cliente se escribe en este stream de salida:

```
package org.stevengould.javaworld;

import java.io.IOException;
import java.io.PrintWriter;
import java.util.Date;
import java.util.Enumeration;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
* This class provides a simple example of a servlet, and
* illustrates some of the information available from an
* HTTP request.
*/
public class RequestDetails extends HttpServlet {
  /**
   * Handler for all GET requests. We simply dump out the
   * requestheader information, followed by the body of
   * the request.
   * @param request the HTTP request submitted to the
   *        server for processing. It is this object that
   *        contains the details of the requested URL, and
```

```
     *        it is the details of this object that we
     *        output as a response.
     * @param response the response object to be used to
     *        send a result back to the client.
     * @exception IOException thrown if a communications
     *        error occurs.
     *@exception ServletException if the GET request could
     *        could not be handled
     */
    public void doGet(HttpServletRequest request, HttpServletResponse response)
                            throws IOException, ServletException {
      response.setContentType("text/html");
      PrintWriter out = response.getWriter();
      out.println("<html>");
      out.println("<head>");
      out.println("<title>Request Details Example</title>");
      out.println("</head>");
      out.println("<body>");

      out.println("<h3>HTTP Request Header</h3>");
      out.println("<table border='1'>");
      out.println(" <tr bgcolor=#e0e0e0>");
      out.println("  <td><strong>Name</strong></td>");
      out.println("  <td><strong>Value</strong></td>");
      out.println(" </tr>");
      Enumeration e = request.getHeaderNames();
      while (e.hasMoreElements()) {
          String name = (String)e.nextElement();
          String value = request.getHeader(name);
          out.println(" <tr>");
          out.println("  <td bgcolor=#e0e0e0>"+name+"</td>");
          out.println("  <td>"+value+"</td>");
          out.println(" </tr>");
      }
      out.println("</table>");

      out.println("<h3>HTTP Request Information</h3>");
      out.println("<table border='1'>");
      out.println(" <tr bgcolor=#e0e0e0>");
      out.println("  <td><strong>Name</strong></td>");
      out.println("  <td><strong>Value</strong></td>");
      out.println(" </tr>");
      out.println(" <tr>");
      out.println("  <td bgcolor=#e0e0e0>Method:</td>");
      out.println("  <td>"+request.getMethod()+"</td>");
      out.println(" </tr>");
      out.println(" <tr>");
      out.println("  <td bgcolor=#e0e0e0>Request URI:</td>");
      out.println("  <td>"+request.getRequestURI()+"</td>");
      out.println(" </tr>");
      out.println(" <tr>");
      out.println("  <td bgcolor=#e0e0e0>Protocol:</td>");
      out.println("  <td>"+request.getProtocol()+"</td>");
```

```
    out.println(" </tr>");
    out.println(" <tr>");
    out.println("  <td bgcolor=#e0e0e0>PathInfo:</td>");
    out.println("  <td>"+request.getPathInfo()+"</td>");
    out.println(" </tr>");
    out.println(" <tr>");
    out.println("  <td bgcolor=#e0e0e0>Remote Address:</td>");
    out.println("  <td>"+request.getRemoteAddr()+"</td>");
    out.println(" </tr>");
    out.println("</table>");

    out.println("<hr>");
    Date date = new Date();
    out.println("<p align=center>Page generated on "+date);

    out.println("</body>");
    out.println("</html>");

    out.close();
    }

  /**
   * For POST requests, we will simply perform the same
   *operations as for GET requests. The best way to do this
   * is to simply invoke the doGet() method with the appropriate
   * parameters.
   * @param request the HTTP request submitted to the server
   *        for processing. It is this object that contains
   *        the details of the requested URL, and it is the
   *        details of this object that we output as a
   *        response.
   * @param response the response object to be used to send a
   *        result back to the client.
   */
  public void doPost(HttpServletRequest request, HttpServletResponse response)
                     throws IOException, ServletException {
    doGet(request, response);
    }
  }
```

Compilar el Servlet

Como los servlets usan clases de extensión Java (clases que no forman parte del JDK principal) debes asegurarte de seleccionar correctamente el CLASSPATH antes de intentar compilar cualquier servlet. El compilador Java necesita poder encontrar las clases y paquetes `javax.servlet.*` (que se encuentra en el fichero **j2ee.jar** en el J2EESDK 1.4). A parte de esto, la compilación se realiza igual que con otro programa Java:

```
javac RequestDetails.java
```

Crear una Aplicación Web

Ahora que hemos creado el servlet, necesitamos pensar en desplegarlo. La especificación Java Servlet 2.2 presentó al menos dos caracterísiticas importantes: una aplicación Web y un archivo de aplicación Web (WAR). de acuerdo a las especificaciones Servlets 2.2:

Nota:
Una aplicación Web es una colección de servlets, páginas HTML, clases, y otros recursos que se pueden empaquetar y ejecutar en varios contenedores de distintos venderores.

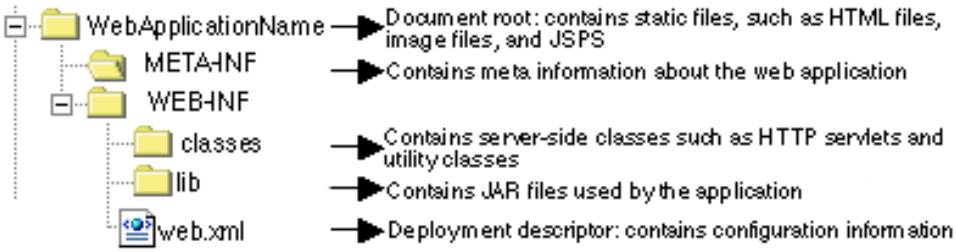
Los WARs simplemente son archivos Java de una aplicación Web con una extensión diferente para diferenciarlos de los comunmente usados JARs.

Antes de la especificación Servlet 2.2, era bastante diferente desplegar Servlets entre diferentes contenedores servlets -- anteriormente también llamados motores servlet. La especificación 2.2 estandarizó el despliegue entre contenedores, llevando así la portabilidad del código Java un paso más allá. Veremos el poder de esto más adelante en este artículo, cuando ilustremos la creación de una sencilla aplicación Web que se despliegue tanto en Apache Tomcat como en WebLogic Server sin ninguna modificación o recompilación.

Estructura de Directorios de la Aplicación Web

La especificación Servlet 2.2 define la estructura de directorios para los ficheros de una aplicación Web. El directorio superior -- o directorio raíz -- debería tener el nombre de la aplicación y definirá la **raíz de documentos** para nuestra aplicación Web. Todos los ficheros debajo de esta raíz pueden servirse al cliente excepto aquellos ficheros que están bajo los directorios especiales `META-INF` y `WEB-INF` en el directorio raíz. Todos los ficheros privados -- como los ficheros class de los servlets -- deberían almacenarse bajo el directorio `WEB-INF`.

En la siguiente figura podemos ver la estructura de directorios de una aplicación Web:



Para crear una aplicación Web, empezamos creando esta estructura de directorio. Toma tu fichero de la clase del servlet compilado y sitúala en el directorio `WEB-INF/classes`. Si hemos definido que nuestro servlet pertenece a un paquete, debemos seguir las reglas estandar de Java y crear los subdirectorios apropiados para que la JVM puedan encontrar nuestras clases. Por ejemplo, si nuestro servlet está definido en un paquete `com.mycompany.myproject`, deberíamos crear la siguiente estructura de directorios:

```
.../WEB-INF
    |--
    classes
        |--
        com
            |--
            mycompany
                |--
                myproject
```

Sitúa tus clases Java en el subdirectorio `myproject`.

Una alternativa útil para copiar los ficheros de clases al directorio apropiado es configurar nuestro entorno de construcción (un Makefile o IDE) para salvar las clases compiladas directamente en los directorios requeridos. Hacer esto nos ahorrará este paso durante el desarrollo.

Modificar el Descriptor de Despliegue

Ahora deberíamos tener todos nuestros ficheros en su lugar para crear nuestra primera aplicación Web. En este punto, necesitamos realizar otra tarea: actualizar el descriptor de despliegue para registrar nuestros servlets con el contenedor. Para crear fácilmente un descriptor de despliegue, simplemente editamos uno existente. Abajo tenemos un esqueleto de un fichero `web.xml`:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app PUBLIC
    "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
    "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">

<web-app>

    <!-- Tus definiciones van aquí  -->

</web-app>
```

Insertamos nuestros descriptores de despliegue de servlets entre las etiquetas `<web-app>` y `</web-app>` de este fichero. El descriptor de despliegue de un servlet debe incluir las siguientes etiquetas (en este orden):

```
  <servlet>

    <servlet-name>nombre</servlet-name>

    <servlet-class>package.nombre.MiClass</servlet-class>

  </servlet>
```

También están permitidas antes de la etiqueta de cierre `</servlet>` varias etiquetas opcionales, que definen las propiedades de ejecución del servlet,. Estas etiquetas definen propiedades como parámetros de inicilización, si el servlet debería o no cargarse en la arranada, los roles de seguidad, y las propiedades de pantalla (incluyendo iconos grandes y pequeños, nombre de pantalla y una descripción).

Hasta ahora, nuestro descriptor de despliegue ha descrito el servlet al contenedor de servlets. Luego, debemos describir cuándo el contenedor de servlets debe invocar al servlet -- nos referimos a esto como *mapeo*. En otras palabras, debemos describir cómo se mapea una URL al servlet. En el fichero `web.xml`, las URLs se mapean de esta forma:

```
  <servlet-mapping>

    <servlet-name>nombre</servlet-name>

    <url-pattern>pattern</url-pattern>
```

```
  </servlet-mapping>
```

OK, suficiente teoría. Veamos un ejemplo de un descritor de despliegue de una aplicación Real. Abajo podemos ver el fichero `web.xml` mínimo que describe nuestro servlet de ejemplo `RequestDetails`:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app PUBLIC
    "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
    "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">

<web-app>

<servlet>
  <servlet-name>RequestDetails</servlet-name>
  <servlet-class>org.stevengould.javaworld.RequestDetails</servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>RequestDetails</servlet-name>
  <url-pattern>SampleServlet</url-pattern>
</servlet-mapping>

</web-app>
```

Como puedes ver en la etiqueta de mapeo, hemos elegido la URL `/SampleServlet` para mapear nuestro servlet `RequestDetails`.

Ya está! Hemos creado nuestra primera aplicación Web que contiene un sólo servlet. Ahora deberíamos poder desplegar esta aplicación en cualquier contenedor servlet compatible con la especificación 2.2.

De esta forma sabremos como trabajar con ellas y como desplegar nuestras aplicaciones Web en un modo de desarrollo. Sin embargo, en un entorno de producción, mantener juntos los ficheros relacionados es más conveniente. En la siguiente sección, veremos como crear ficheros archivos de aplicaciones Web (WARs) que hacen exactamente esto.

Crear WARs

Como se mencionó anteriormente, un fichero WAR simplemente es un fichero JAR con la extensión cambiada para reflejar su propósito diferente. Ya hemos visto la estructura de directorios requerida para una aplicación Web. Para crear un fichero WAR, usamos esta misma estructura de directorio.

Para crear un WAR para nuestra aplicación , vamos al directorio raíz que contiene nuestra aplicación Web y tecleamos el siguiente comando:

```
jar cv0f myWebApp.war .
```

Observa el punto obligatorio que hay al final de la línea; le dice al programa **jar** que archive el directorio actual.

El comando `jar` anterior creará un fichero WAR llamado `myWebApp.war`. Luego, veremos como desplegar este fichero WAR en Tomcat 3.2 y en WebLogic Server 6.0.

Desplegar Aplicaciones Web en Tomcat 3.2

Tomcat 3.2 sirve como implementación de referencia para la especificación Java Servlet 2.2. Para el propósito de este artículo hemos asumido que estás usando Tomcat 3.2.1 o posterior.

Para desplegar nuestra aplicación Web en Tomcat, copiamos el directorio raíz de nuestra aplicación -- el que contiene `web.xml` y sus subidirectorios -- al subdirectorio `webapps/ROOT/` de nuestra instalación de Tomcat. Podríamos querer salvar una copia de la aplicación Web por defecto antes de sobreescribirla.

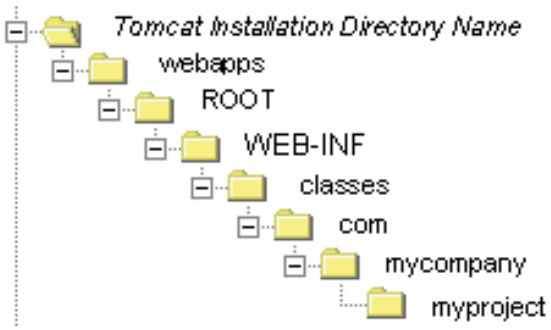
Bajo Unix, por ejemplo, si hemos instalado Tomcat en el directorio `/opt/jakarta-tomcat-3.2.1`, deberíamos copiar las clases del servlet debajo del directorio:

```
/opt/jakarta-tomcat-3.2.1/webapps/ROOT/
```

Si ejecutamos Tomcat bajo Windows y los hemos instalado en el directorio `C:\Program Files\Jakarta-Tomcat-3.2.1`, deberíamos copiar las clases del servlet bajo el directorio:

```
C:\Program Files\Jakarta-Tomcat-3.2.1\webapps\ROOT\
```

El subdirectorio `webapps/ROOT/WEB-INF/classes` es el directorio por defecto en el que Tomcat buscará nuestras clases Java. Si hemos definido que nuestros servlets pertenecen a un paquete, deberíamos seguir las reglas estándar de Java y crear los subdirectorios apropiados para que la JVM pueda encontrar nuestras clases, como hicimos anteriormente. Por ejemplo, si definimos nuestro servlet en un paquete `com.mycompany.myproject`, entonces deberíamos tener la estructura de directorios que se ve en la siguiente figura:



Nuestras clases Java estarán en el subdirectorio `myproject`.

Esto es todo lo necesario. No hay más configuración posterior. Siguiendo con el ejemplo `RequestDetails`, intenta copiar los ficheros de la aplicación Web en la aplicación Web por defecto de Tomcat.

Probar el Servlet

Para probar nuestro servlet, arranca el servidor Tomcat, abre tu navegador Web, y escribe una URL con la siguiente forma:

```
http://:/
```

Donde:

- `address` es el nombre o dirección IP de la máquina que está ejecutando

Tomcat. Podemos usar `localhost` si el navegador se está ejecutando sobre la misma máquina que Tomcat.
- `port` es el puerto en el que escucha Tomcat. Por defecto, es el puerto 8080.
- `servletName` es el nombre del servlet que queremos invocar. Debería corresponder al valor contenido en las etiquetas `<url-pattern></url-pattern>` del fichero descriptor de despliegue.

Por ejemplo, si Tomcat se está ejecutando en la misma máquina que el navegador y está escuchando el puerto por defecto (8080), podemos probar nuestro servlet de ejemplo `RequestDetails` (que está mapeado a la URL `SampleServlet`) abriendo la siguiente URL:

```
http://localhost:8080/SampleServlet
```

Observa el poco trabajo necesario para desplegar una aplicación Web. Copiar algunos ficheros y probar. Esta facilidad de uso la hacen posible la especificación Java Servlet 2.2 y el uso de los descriptores de despliegue.

Ahora que hemos visto como desplegar servlets en Tomcat, veamos como desplegar un servlet en WebLogic Server.

Desplegar Aplicaciones Web en WebLogic Server 6.0

Aunque WebLogic Server 5.1 fue la primera versión de WebLogic Server en proporcionar soporte para la especificación Java Servlet 2.2 y aplicaciones Web, WebLogic Server 6.0 tiene algunas mejoras importantes que simplifican el despliegue de aplicaciones Web (tanto en formato expandido como empaquetadas como un fichero WAR).

Desplegar WARs Usando la Consola

Con nuestro ejemplar de WebLogic Server ejecutándose, arrancamos la **WebLogic Server Console**. Asumiendo una instalación por defecto, la consola se puede traer desde la máquina `localhost` abriendo la siguiente URL en un navegador Web:

```
http://localhost:7001/console
```

Se nos pedíra el nombre y la password del usuario system antes de permitirnos el paso a la **Consola**.

Para desplegar nuestro fichero WAR, una vez que hemos accedido a la **Consola**:

1. Pulsamos sobre le nodo **Web Applications** en el panel izquierdo de la **Consola**.
2. En el panel derecho, pulsamos **"Install a new Web Application..."**
3. Tecleamos el path completo y el nombre de fichero de nuestro WAR, o usamos el botón **Browse...** para localizarlo.
4. Pulsamos el botón **Upload**.

Esto es todo. Si todo fue correctamente, deberías ver nuestra aplicación Web listada bajo **Web Applications** en el panel izquierdo de la **Consola**. Podrías necesitar refrescar la vista para que apareciera.

Como alternativa al uso de la **WebLogic Server Console**, es posible copiar la estructura de directorios completa de la aplicación Web como lo hicimos cuando la desplegamos en Tomcat.

Desplegar Aplicaciones Web Manualmente

Normalmetne cuando uno piensa en hacer las tareas manualmente, o a mano, la reacción automática es esperar que la tarea sea un poquito más complicada que su equivalente automático. En el caso de desplegar aplicaciones Web bajo WebLogic Server 6.0, la aproximación manual es tan facíl, si no más, que utilizar la **Consola**.

Simplemente copiamos nuestro fichero WAR o la estructura de directorios de nuestra aplicación Web completa al subdirectorio `config/mydomain/applications` de nuestra distribución de WebLogic Server (donde *mydomain* es el nombre de nuestro dominio WebLogic Server). Tan pronto como nuestros ficheros hayan sido copiados, WebLogic Server despliega la aplicación Web.

Probar el Servlet

Para probar nuestro servlet, abre tu navegador Web, y escribe una URL con la siguiente forma:

```
http://:/
```

Donde:

- `address` es el nombre o dirección IP de la máquina que está ejecutando WebLogic Server. Podemos usar `localhost` si el navegador se está ejecutando sobre la misma máquina que WebLogic Server.
- `port` es el puerto en el que escucha WebLogic Server. Por defecto, es el puerto 7001.
- `servletName` es el nombre del servlet que queremos invocar. Debería corresponder al valor contenido en las etiquetas `<url-pattern></url-pattern>` del fichero descriptor de despliegue.

Por ejemplo, si WebLogic Server se está ejecutando en la misma máquina que el navegador y está escuchando el puerto por defecto (7001), podemos probar nuestro servlet de ejemplo `RequestDetails` (que estám mapeado a la URL `SampleServlet`) abriendo la siguiente URL:

```
http://localhost:7001/SampleServlet
```

De nuevo, el despliegue de nuestra aplicación Web o fichero WAR sólo ha requerido que copiemos unos cuantos ficheros y probemos el servlet -- no se necesita configuración.

Reconfigurar Aplicaciones Web

Una vez que hemos desplegado nuestras aplicaciones Web en WebLogic Server, podemos usar la **Consola** para configurar y reconfigurar la aplicación. Según hagamos los cambios en cualquiera de las configuraciones, los detalles se escribirán automáticamente en el fichero `config.xml` de WebLogic Server. La próxima vez que lo arranquemos usará este fichero para configurar nuestra aplicación.

Taller PHP. PHP y funciones FTP

Autor: Alejandro Almunia

Vamos a examinar las funciones FTP. A menudo pueden sernos útiles para algún proyecto que se esté llevando a cabo, así que nunca viene de más conocerlas. Además, son pocas, ¡¡ y muy fáciles !!

Configurando el fichero php.ini para admitir funciones FTP

Antes de nada, tenemos que configurar nuestro fichero php.ini para que admita funciones de FTP. Es muy simple. En la parte en la que se listan todas las extensiones, debe haber una con el nombre php_ftp.dll Descomentadla, y, en teoría, eso es todo lo que hay que hacer para disponer de ellas. Para estar del todo seguro, ejecuta phpinfo() y lo sabrás.

Usando las funciones FTP más simples

Para este ejemplo, vamos a usar un poco de código fuente comentado. Creo que es bastante explicativo por si mismo.

```
<?

//Hazte cuenta de que puede tardar más de 30 segundos.
set_time_limit(0);

//Conectamos al host
$FtpConn = ftp_connect("ftp.cdrom.com");

//Nos autentificamos como usuarios registrados o anónimos
if(!ftp_login($FtpConn,"anonymous","me@you.net")){
        echo "No se ha podido realizar la conexión";
        exit;
}

//Obtenemos el directorio actual
$directorio = ftp_pwd($FtpConn);

//Obtenemos el listado del directorio actual
$lista = array();
$lista = ftp_nlist($FtpConn,$directorio);
```

```
//Mostramos sus contenidos
echo "<B><CENTER>CONTENIDOS DEL DIRECTORIO / (RAÍZ)</CENTER></B>";

echo "<pre>";
print_r($lista);
echo "</pre>";


//Cambiamos al directorio /pub
if(!ftp_chdir($FtpConn,"pub")){
        echo "Se ha producido un error al entrar en el directorio /pub";
        exit;
}

//Almacenamos el directorio actual
$directorio2 = ftp_pwd($FtpConn);

//Obtenemos el listado del directorio actual
$list = array();
$list = ftp_nlist($FtpConn,$directorio2);

//Mostramos su contenido
echo "<B><CENTER>CONTENIDOS DEL DIRECTORIO /pub</CENTER></B>";

echo "<pre>";
print_r($list);
echo "</pre>";



//Tansferimos un fichero
if(!ftp_get($FtpConn,"test.txt","test.txt",FTP_BINARY)){
        echo "Imposible recuperar fichero test.txt";
        exit;
}

/*

ESTO NO ES POSIBLE VERLO FUNCIONANDO
//Cargamos un fichero
if(!ftp_put($FtpConn,"ind.txt","",FTP_BINARY)){
        echo "Imposible cargar el fichero";
        exit;
}

*/
?>
```

Como ves, es muy sencillo usar FTP con PHP. Si tienes alguna duda, escribeme a flipis@flipis.net y la resolveré, si puedo, :-)

Últimos comentarios

[Subir]

Para benjo (19/10/2002)
Por Victor

Si lo que quieres es generar graficas de barras y demas, echale un vistazo a phplot, con esa clase he podido llevar a cabo un proyecto de una empresa que consistia en eso.

Redes-Linux, la web de redes bajo linux.

una ayuda en graficos en PHP (16/10/2002)
Por benjo

hola quetal necesito ayuda tengo que mostrar datos en tortas y barras en PHP por favor me pueden ayudar...

gracias....

Me ha resultado muy util (13/10/2002)
Por manuel

Gracias Alejandro.

Me ha resultado muy util este articulo. Es justo lo que estaba buscando, para terminar un programilla que estaba realizando.

Un saludo.

Recomendar este tutorial | Estadísticas

Taller PHP. Trabajando con PHP y ficheros

Autor: Alejandro Almunia

Leer comentarios (4) | Escribir comentario | Puntuación: ■■■■■ (12 votos) Vota

1 . Abriendo un fichero de texto, lectura, escritura y añadido
2 . Subir ficheros al servidor
3 . Forzar descarga de ficheros al navegador

Recomendar este tutorial | Estadísticas

En este nuevo tutorial, voy a tratar el tema de los ficheros y como se trabaja en ellos desde PHP. No va a ser exhaustivo ni mucho menos, pretende proporcionaros las bases para que experimentéis por vuestra cuenta y riesgo. Vamos a aprender unas cuantas cosas útiles al respecto de los ficheros, así que, si estáis listos, empezamos. Para este tutorial solo váis a necesitar PHP, nada más (ni MySQL ni otra cosa).

Abriendo un fichero de texto, lectura, escritura y añadido

Lo primero que vamos a hacer es escribir un sencillo fichero de texto. Lo abriremos, escribiremos un par de líneas dentro de él y luego lo cerraremos. El código que realiza esto se puede ver a continuación.

```
<?

#Abrimos el fichero en modo de escritura
$DescriptorFichero = fopen("fichero_prueba.txt","w");

#Escribimos la primera línea dentro de él
$string1 = "Esta es la primera línea de texto\r\n";
fputs($DescriptorFichero,$string1);

#Escribimos la segunda línea de texto
$string2 = "Y esta es la segunda línea de texto\r\n";
fputs($DescriptorFichero,$string2);

#Cerramos el fichero
fclose($DescriptorFichero);

?>
```

Así pues, el script anterior lo único que hace es abrir un fichero llamado **fichero_prueba.txt**, y escribe dentro de él dos líneas de texto. Os habréis fijado en el \r\n de detrás de las líneas de texto, en las variables $string1 y $string2. Esto se debe a que, si no estuviese puesto, el programa escribiría todo seguido. Para comprobarlo, quitadlo y ejecutad de nuevo el programa. Con solo \n no sirva, al menos en mi sistema Windows 2000. :-) En Linux, basta con un \n.

Otra de las cosas importantes del anterior script es algo que quizás no hayamos visto de cerca. Fijémonos en la siguiente línea:

```
$DescriptorFichero = fopen("fichero_prueba.txt","w");
```

La función fopen sirve para abrir un fichero en un **modo**. Los modos pueden ser seis y son los siguientes.

Además de listarlos, explicaré las diferencias (no siempre tan obvias), al respecto de ellos.

Modo de apertura	Qué significa
r	Modo de solo lectura. Se abre el fichero y el cursor se coloca al principio del mismo, permitiendo leerlo hasta el final.
r+	Modo de lectura/escritura. Se abre el fichero y el cursor se coloca al principio del mismo,permitiendo leer o escribir en el fichero.
w	Modo de solo escritura. Se crea el fichero si no existiese, y, si existe, se borra todo su contenido, se sitúa el cursor al principio del fichero permitiendonos escribir.
w+	Modo de escritura/lectura. Si el fichero no existe, se crea, y, si existiese, se borra todo su contenido, se sitúa el cursor al principio del fichero permitiéndonos escribir y leer.
a	Modo de añadido. Abre el fichero, sitúa el cursor al final del mismo y permite escribir. Si el fichero no existe, lo crea, pero, en caso de existir, no borra su contenido.
a+	Modo de añadido/lectura. Sitúa el cursor al final del fichero y permite escribir y leer. Si el fichero no existe, lo crea, pero, si existe, no borra su contenido.

Así pues, estos son los seis modos de abrir un fichero. Vamos ahora a ver un ejemplo en código del uso de los mismos. El siguiente script va a hacer las siguientes tareas:

- Crear un fichero y escribir en él dos líneas de texto.
- Abrir el fichero de nuevo, esta vez en modo añadido, y escribir otras dos líneas.

Es poco, pero la lectura de ficheros la veremos al final de esta parte del tutorial. De momento, aquí está el código del script de PHP.

escribir2.php

```
<?

#Abrimos el fichero en modo de escritura
$DescriptorFichero = fopen("fichero_prueba.txt","w");

#Escribimos la primera línea dentro de él
$string1 = "Esta es la primera línea de texto\r\n";
fputs($DescriptorFichero,$string1);

#Escribimos la segunda línea de texto
$string2 = "Y esta es la segunda línea de texto\r\n";
fputs($DescriptorFichero,$string2);

#Cerramos el fichero
fclose($DescriptorFichero);

#Volvemos a abrir el fichero, esta vez en modo de añadir
```

```
$Descriptor2 = fopen("fichero_prueba.txt","a");

#Añadimos la tercera línea de texto
fputs($Descriptor2,"Esta es la tercera línea, añadida con modo \"a\"\r\n");

#Añadimos la cuarta línea de texto
fputs($Descriptor2,"Esta es la cuarta línea, añadida con modo \"a\"\r\n");

#Cerramos el fichero
fclose($Descriptor2);

?>
```

Como podéis comprobar si abrís el fichero recién creado, éste contiene cuatro líneas, dos de ellas escritas con modo "w" y otras dos con modo "a". Si ya tenéis más o menos claro como funciona, vamos a pasar a ver dos funciones muy útiles para leer ficheros de texto: `fgets()` y `feof()`. A través de `fgets()` podemos leer una línea del fichero de texto cada vez. `feof()` sirva para saber si hemos llegado al final del fichero. Para ver como funcionan, crearemos un script que leerá el fichero que hemos creado con los dos scripts anteriores.

leer.php

```
<?

#Abrimos el fichero en modo lectura
$DescriptorFichero = fopen("fichero_prueba.txt","r");

#Hasta que no lleguemos al final del fichero
while(!feof($DescriptorFichero)){

    #Capturamos 4096 caracteres dentro de la línea,
    #o menos si hay un retorno de carro antes
    #(\r\n en Win32, \r en UNIX)
    $buffer = fgets($DescriptorFichero,4096);

    #Soltamos el texto, añadiendo <BR> detrás
    echo $buffer."<BR>";
}

?>
```

Como véis, este script lee el fichero de texto línea a línea y lo va mostrando en el navegador. La función `feof()` devuelve TRUE cuando ha llegado al final del fichero. `fgets()`, va, pues, leyendo linea a linea y almacenándolo en una varible llamada `$buffer`.

Ahora vamos a ver como funcionan los modos w+, r+ y a+. Veréis que son diferentes de los anteriores en el sentido de que permiten dos operaciones, y tambien en el sentido de como tratan los ficheros. Empezaremos con w+. El siguiente script explica qué es lo que hace este modo con los ficheros.

leer_wplus.php

```
<?

#Abrimos el fichero en modo w+
$Descriptor1 = fopen("nuevo_fichero.txt","w+");

#Vamos a escribir un par de líneas en el fichero
fputs($Descriptor1,"Esta es la primera línea de texto\r\n");
```

```
fputs($Descriptor1,"Esta es la segunda línea de texto\r\n");

#Ahora cerraremos el fichero
fclose($Descriptor1);

#Volvemos a abrirlo en modo w+
$Descriptor2 = fopen("nuevo_fichero.txt","w+");

#Escribimos un par de líneas
fputs($Descriptor2,"Esta es la tercera línea de texto\r\n");
fputs($Descriptor2,"Esta es la cuarta línea de texto\r\n");

#Volvemos al principio del fichero
rewind($Descriptor2);

#Vamos leyendo líneas y mostrándolas
while(!feof($Descriptor2)){

    $buffer = fgets($Descriptor2,4096);
    echo $buffer."<BR>";

}

#Cerramos el fichero
fclose($Descriptor2);

?>
```

Como véis, al ejecutarlo, el resultado es el siguiente:

```
Esta es la tercera línea de texto
Esta es la cuarta línea de texto
```

¿Por qué no aparecen la primera y la segunda línea escritas? Observemos lo que hemos hecho. Primero abrimos el fichero y escribimos dentro de él dos líneas de texto. Tras esto, lo cerramos y lo volvemos a abrir, en modo w+. Este modo **BORRA EL CONTENIDO ANTERIOR** del fichero, por lo que en este solo aparecen las dos últimas líneas. Como véis, se puede utilizar este modo para leer desde el fichero con `fgets()`.

Ahora vamos a ver un ejemplo con r+. Vamos a crear un script que haga lo mismo que el anteriro, pero en vez de abrir los ficheros con w+, los abrirá con r+.

leer_rplus.php

```
<?

#Abrimos el fichero en modo w+
$Descriptor1 = fopen("nuevo_fichero.txt","w");

#Vamos a escribir un par de líneas en el fichero
fputs($Descriptor1,"Esta es la primera línea de texto\r\n");
fputs($Descriptor1,"Esta es la segunda línea de texto\r\n");

#Ahora cerraremos el fichero
fclose($Descriptor1);

#Volvemos a abrirlo en modo w+
$Descriptor2 = fopen("nuevo_fichero.txt","r+");
```

```
#Escribimos un par de líneas
fputs($Descriptor2,"Esta es la tercera línea de texto\r\n");
fputs($Descriptor2,"Esta es la cuarta línea de texto\r\n");

#Volvemos al principio del fichero
rewind($Descriptor2);

#Vamos leyendo líneas y mostrándolas
while(!feof($Descriptor2)){

    $buffer = fgets($Descriptor2,4096);
    echo $buffer."<BR>";

}

#Cerramos el fichero
fclose($Descriptor2);

?>
```

Si ejecutáis el script, quizás observéis, sorprendidos, que el resultado es el mismo que en el anterior. Pero lo que ha sucedido, en cambio, no es lo mismo. Vamos a anlizarlo por partes. Primero, hemos abierto el fichero en modo w (escritura), para meter dos líneas en el fichero. Tras esto, lo cerramos, y lo abrimos en modo r+ (lectura/escritura). Al abrirlo de este modo, el cursor se sitúa al principio del fichero, por lo que al escribir las siguientes dos líneas, borra el contenido de las dos líneas anteriroes.. Antes de mostrar el contenido del fichero usamos la función `rewind()`, que rebobina el cursor hasta el principio del fichero. Para añadir al final de fichero, necesitamos el modo a+, como veremos en el siguiente ejemplo.

leer_aplus.php

```
<?
#Abrimos el fichero en modo w+
$Descriptor1 = fopen("nuevo_fichero.txt","w+");

#Vamos a escribir un par de líneas en el fichero
fputs($Descriptor1,"Esta es la primera línea de texto\r\n");
fputs($Descriptor1,"Esta es la segunda línea de texto\r\n");

#Ahora cerraremos el fichero
fclose($Descriptor1);

#Volvemos a abrirlo en modo w+
$Descriptor2 = fopen("nuevo_fichero.txt","a+");

#Escribimos un par de líneas
fputs($Descriptor2,"Esta es la tercera línea de texto\r\n");
fputs($Descriptor2,"Esta es la cuarta línea de texto\r\n");

#Volvemos al principio del fichero
rewind($Descriptor2);

#Vamos leyendo líneas y mostrándolas
while(!feof($Descriptor2)){

    $buffer = fgets($Descriptor2,4096);
    echo $buffer."<BR>";
```

```
}

#Cerramos el fichero
fclose($Descriptor2);

?>
```

El resultado de este método es el esperado. Se abre el fichero en modo escritura y se insertan las dos líneas de texto. Se cierra este descriptor, y se abre otro en modo a+. El cursor se sitúa al final del fichero, y comienza a añadir el texto. El resultado son las cuatro líneas dentro del fichero de texto.

Esto es todo en cuanto a modos de apertura. En la siguiente sección vamos a ver como subir ficheros al servidor, algo muy útil cuando se trata de páginas web.

Subir ficheros al servidor

Para ejemplificar la subida de archivos al servidor, vamos a ver un script de ejemplo. El script tiene dos partes; la primera, el formulario, en el que se introduce el fichero a cargar, y la segunda parte, en la que se procesa la subida y se informa al usuario del éxito o fracaso de la carga.

upload.php

```
<?

if(!isset($cargar)){

?>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>:: Formulario de carga de ficheros ::</TITLE>
</HEAD>

<BODY>

<FORM NAME="elForm" METHOD="POST"
      ACTION="<? echo $PHP_SELF; ?>?cargar=1"
      ENCTYPE="multipart/form-data">

    <TABLE WIDTH="80%" STYLE="font-family:Arial;font-size:9pt;">

    <TR>
        <TD ALIGN="LEFT"><INPUT TYPE="FILE" NAME="elFichero"></INPUT></TD>
    </TR>

    <TR>
        <TD ALIGN="LEFT"><INPUT TYPE="SUBMIT" VALUE="Subir el fichero">
    </TR>

    </TABLE>

</FORM></BODY></HTML>

<?

}
```

```
#Aquí realizamos la carga del fichero
if(isset($cargar)){

    #Le damos al fichero un nuevo nombre
    $nuevositio = "nuevo_fichero_caragado.000";

    #Lo copiamos
    if(!copy($elFichero,$nuevositio)){
        echo "NO SE HA PODIDO SUBIR EL FICHERO";
    }
    else{
        echo "FICHERO SUBIDO CON ÉXITO";
    }

}

?>
```

Como vemos, el script realiza la carga del fichero. Para rizar un poco más el rizo os propongo lo siguiente: un script que suba un número de ficheros introducido por el usuario. Se compondrá de tres partes. La primera, en la que se pregunta al usuario el número de ficheros que desea introducir. La segunda, en la que aparecen todos los campos de tipo fichero, y la tercera, en la que se procesa la carga. El código es como sigue:

upload_multiple.php

```
<?

/* Este script se encarga de subir múltiples ficheros al servidor. */

#Formulario en el que se pregunta el número de ficheros
if(!isset($fich) && !isset($cargar)){

    $txt ="<HTML><HEAD>\n";
    $txt.="<TITLE>:: ¿Cuántos ficheros quiere subir hoy? ::</TITLE>\n";
    $txt.="</HEAD><BODY>\n";
    $txt.="<FORM NAME=\"frmNumFicheros\"
                 METHOD=\"POST\"
                 ACTION=\"".$PHP_SELF."?fich=1\">\n";

    $txt.="<BR><BR><BR><BR>\n";
    $txt.="<DIV ALIGN=\"CENTER\">\n";
    $txt.="<INPUT TYPE=\"TEXT\" NAME=\"numFicheros\">\n";
    $txt.="      ";
    $txt.="<B>Introduce el número de ficheros</B>\n";
    $txt.="<BR><BR>\n";
    $txt.="<INPUT TYPE=\"SUBMIT\"
                  VALUE=\"Mostrar campos para subir ficheros\">\n";
    $txt.="<BR></DIV>\n";

    $txt.="</FORM></BODY></HTML>\n";

    echo $txt;
}

#Formulario en el que se muestran los campos tipo fichero
if(isset($fich)){
```

```
    $txt ="<HTML><HEAD>\n";
    $txt.="<TITLE>:: ¿Cuántos ficheros quiere subir hoy? ::</TITLE>\n";
    $txt.="</HEAD><BODY>\n";
    $txt.="<FORM ENCTYPE=\"multipart-form/data\"
                 NAME=\"frmCargaFicheros\"
                 METHOD=\"POST\"
ACTION=\"".$PHP_SELF."?cargar=1&cantidad=".$HTTP_POST_VARS["numFicheros"]."\">\n";

    for($i=0;$i<$HTTP_POST_VARS["numFicheros"];$i++){

        $txt.="<INPUT TYPE=\"FILE\" NAME=\"fichero_$i\"><BR>\n";

    }

    $txt.="<INPUT TYPE=\"SUBMIT\" VALUE=\"cargar\">\n";

    $txt.="</FORM></BODY></HTML>\n";


    echo $txt;
}

#Parte que gestiona el proceso de carga
if(isset($cargar)){


    for($n=0;$n<$cantidad;$n++){

        #Creamos la "variable variable"
        $nomvar = "fichero_$n";
        $valvar = $;

        #Extraemos el nombre del fichero sin la ruta
        $nomfichero = basename($valvar);

        #Le damos al fichero su nombre, metiéndolo dentro del directorio /subidas
        $nuevositio = "subidas/".$nomfichero."";

        #Lo copiamos
        if(!copy($valvar,$nuevositio)){
            echo "NO SE HA PODIDO SUBIR EL FICHERO";
        }
        else{
            echo "FICHERO SUBIDO CON ÉXITO";
        }
    }


}

?>
```

De este modo podemos cargar varios ficheros al mismo tiempo. Tendrás que crear el directorio /subidas manualmente. Vamos a termnar con esta sección y pasar a la siguiente, en la que se explica como forzar al cliente a descargarse el fichero en vez de verlo on-line.

Forzar descarga de ficheros al navegador

A veces puede ser interesante que el usuario se descargue el fichero en vez de verlo on-line. Para realizar esta operación, solo necesitamos utilizar el siguiente código que voy a explicar a continuación. El script consta de una sola parte. Vamos a descargarnos un fichero .html, en vez de verlo en el navegador. El nombre del fichero será prueba_descarga.html. El código es como sigue:

descargar.php

```
<?

function Descargar($ElFichero){

    $TheFile = basename($ElFichero);

    header( "Content-Type: application/octet-stream");
    header( "Content-Length: ".filesize($ElFichero));
    header( "Content-Disposition: attachment; filename=".$TheFile."");
    readfile($ElFichero);
}

Descargar("prueba_descarga.html");

?>
```

Como ves, el script se ejecuta y el fichero, pese a ser HTML, e interpretable por el navegador, es forzado a ser descargado, igual que si hubiéramos pulsado el botón derecho.

Y con esto termino este tutorial. Espero que os sea útil, aunque se que es corto. Es posible que en otro tutorial próximo me extienda más sobre este tema (sobre todo si mis conocimeintos aumentan, jeje).

Descargate los ficheros del artículo

Pues la verdad no tengo conocimientos en PHP, pero si en HTML javascript y C++, Por lo que aprender PHP se me has aun mas interesante aprenderlo... Por el momento estoy biendo el ambiente de php ya que Desconosco hasta el compilador..

SALUDOS..!!

Gracias (11/09/2002)
Por manuel

Gracias por el artículo me ha resultado muy util.

Recomendar este tutorial | Estadísticas

El Proyecto ASP.NET Web Matrix

Autor: Web Matrix

Leer comentarios (6) | Escribir comentario | Puntuación: ■■■■■ (4 votos) Vota

1 . Presentación
2 . Descripción de Características
  2.1 . Sencillo / Fácil de usar
  2.2 . Ligera
  2.3 . Integración con la Comunidad
3 . Más información

Recomendar este tutorial | Estadísticas

Presentación

El proyecto ASP.NET Web Matrix es una herramienta ligera y fácil de usar para el desarrollo de aplicaciones web con ASP.NET. Sus características orientadas a la comunidad de desarrollo permiten la integración de usuarios noveles y expertos para compartir sus conocimientos en ASP.NET. El Proyecto Web Matrix une a la comunidad ASP.NET al utilizar características como la mensajería instantánea, la integración de un cliente de Chat así como ayuda basada en comunidades. Todo esto unido a un diseñador de formularios web tipo WYSIWYG (What You See Is What You Get - lo que ves es lo que obtienes) está disponible en un paquete de un tamaño mínimo ~1MB.

El Proyecto ASP.NET Web Matrix es:

- Una herramienta ligera y sencilla de usar, orientada a la comunidad, para construir aplicaciones ASP.NET
- Construida íntegramente usando el .NET Framework con lenguaje C#.
- · Incluido con MSDE - Microsoft Data Engine (como una descarga separada)

El Proyecto ASP.NET Web Matrix está diseñado para:

- Conectar a los desarrolladores a la comunidad ASP.NET
- Ayudar a los desarrolladores a facilitar su opinión sobre nuevas características
- Experimentar con nuevas ideas y posibilidades

El Proyecto ASP.NET Web Matrix es complementario a Visual Studio .NET:

- Web Matrix ha sido diseñado pensando en el desarrollador aficionado - Visual Studio .NET es una impresionante herramienta para el desarrollador profesional.
- Web Matrix es una estupenda forma de mejorar las habilidades con ASP.NET, para más adelante migrar a Visual Studio .NET.

Descripción de Características

Sencillo / Fácil de usar

Característica	Descripcion
Diseñadores WYSIWYG	Construya aplicaciones web arrastrando y soltando controles desde la caja de herramientas de Web Matrix.
Ejemplos de Aplicaciones	El Proyecto Web Matrix facilita varios ejemplos de aplicaciones y páginas incluyendo: páginas marcadas por fecha, servicios web, caching de salida, páginas de login, y más.
Integración de Datos	Construya fácilmente aplicaciones web orientadas a datos al integrar bases de datos MSDE o SQL Server con aplicaciones web. Simplemente arrastre y suelte las tablas de datos desde el diseñador para conectarlas a su página.
Galería de Controles	Mejore sus aplicaciones y desarrolle a mayor velocidad usando controles de la galería en línea en www.asp.net
Generador Proxy de Servicios Web XML	Convierta cualquier clase de VB.NET o C# en un Servicio Web XML automáticamente.
Visor de Clases .NET	Navegue rápidamente por la biblioteca de clases del .NET. Framework para encontrar las clases que necesita para construir su aplicación.
Editor de texto coloreado de rica sintaxis	Aprender VB.NET y C# es sencillo con el editor de texto coloreado.
Ayudantes y plantillas basados en tareas	Ayuda para guiar a los usuarios a través de las tareas más comunes.
Constructores de Código	Especifique unos pocos parámetros y el código es generado automáticamente.
Trabajo por archivos (no se requiere un proyecto)	El trabajo por archivos proporciona una alternativa más ligera al habitual trabajo por proyectos.
Soporte para alojamiento FTP	Aloje sus aplicaciones en una selección de terceras compañías vía FTP

Ligera

Característica	Descripcion

Pequeño tamaño / descarga rápida	Descarga ~1MB - ¡cabe en un disco flexible!
Trabajo por archivos (no se requiere un proyecto)	El trabajo por archivos proporciona una alternativa más ligera al habitual trabajo por proyectos.
Servidor Web para pruebas incluido	Incluye su propio servidor web ligero - no se requiere una instalación previa de IIS (también se proporcionan enlaces a compañías de alojamiento)

Integración con la Comunidad

Característica	Descripcion
Integración de Mensajería Instantánea	Comenta con tus colegas ASP.NET a través del MSN messenger integrado.
Extensibilidad	Descargue controles web y add-ins de la galería en línea - o escriba los suyos propios y póngalos a disposición de otros para su uso.
Integración en Listserv / Newsgroup	Obtenga respuesta rápida a sus preguntas al conectar directamente con los servidores de listas y noticias de ASP.NET.
Cliente de Chat integrado	Acceso directo a los "Chat rooms" de ASP.NET.
Seleccionador de Controles	Navegue y descargue controles en linea - revise y puntúe los construidos por otros.
Compartidor de Fragmentos de código	Comparta sus fragmentos de código con otros desarrolladores desde la propia herramienta.

Más información

Más información y descarga a partir del 17 de junio de 2002 en:

- http://www.asp.net/webmatrixproject
- http://www.webmatrixproject.net

Le escribo para comentarle que su proyecto ASP.NET tiene el mismo nombre que un proyecto que terminamos de diseñar el semestre pasado en nuetra universidad con fines sociales y educativos.

Muy respetuosamente solicito a usted nos envie una copia del proyecto o un informe del por qué de sus siglas, ya que el proyecto de nosotros esta registrado local y nacionalmente con el mismo nombre y nos interesaria conocer de su propuesta y proyecto para tener claridad de su aplicación y funcionamiento.

Agradecemos su atencion a la presente.

atentamente,

DAVID F. BASTIDAS
RUBEN D. OROZCO

Diseñadores del Software ASP.NET 1.0

Traducción (24/07/2002)
Por Alejandro López

Hola, me gustaría proponer la traducción de la documentación del proyecto, ya he traducido parte de él, en cuanto lo tenga listo me gustaría subirlo a la página, para su revisión, ya que no soy traductor de profesión.. =P, además quiero saber con quién debo contactarme para que lo pongan el linea.

ASP.NET web MAtrix (26/06/2002)
Por Jhon Cuadros

Seria genial si estuviese en castellano, además para poderlo bajar en un zip

Cuando lo suben en español? (18/06/2002)
Por LomaxCyrrus

Como ya es de costubre estoy leyendos sus infos, y este de asp lo encuentro genial, pero si lo suben en español seria mucho mejor.

Recomendar este tutorial | Estadísticas

Principio Página

Taller Java. JDC Tech Tips (20 de Diciembre de 2001)

Autor: Sun
Traductor: Juan Antonio Palos (Ozito)

Presentación

Bienvenido a los Consejos Técnicos de la Conexión del Desarrollador Java (JDC), del 20 de Diciembre de 2001. Esta edición cubre:

- Crear Modal Internal Frames, Aproximación I.
- Crear Modal Internal Frames, Aproximación II.

Estos consejos fueron desarrollados usando Java(tm) 2 SDK, Standard Edition v 1.3

Se puede ver esta edición (en su original en inglés) de los Consejos Técnicos en formato html en http://java.sun.com/jdc/JDCTechTips/2001/tt1220.html

Crear Modal Internal Frames, Aproximación I

El conjunto de componentes **Java Foundation Classes (JFC) Project Swing** proporciona el componente `JOptionPane` para mostrar simples díalogos estándards. Un diálogo es una ventana que muestra información, pero también espera una respuesta del usuario. Por ejemplo, un diálogo podría avisar al usuario de un problema potencial, y también mostar un boton **OK** para que el usuario pueda reconocer el aviso. Podemos mostrar un diálogo en una ventana popup o un marco interno, es decir, una ventana dentro de otra ventana.

Para mostrar un diálogo en una ventana de popup, usamos uno de los métodos `JOptionPaneshowXXXDialog` como `showMessageDialog`. La ventana desplegable en este caso es modal. Esto significa que el usuario debe responder a la ventana antes de que el programa continúe su ejecución. Pero hay más -- modal también significa que el usuario no puede interactuar con otras partes del programa. Para mostrar un diálogo en un frame interno, usamos uno de los métodos `showInternalXXXDialog` de `JOptionPane`. Estos diálogos de marco interno no son modales. Sin embargo hay veces que podríamos querer que un diálogo interno sea modal. Este truco mostrará como crear un diálogo modal en un marco interno.

Hay un límite para soportar la modalidad en marcos internos creados por un `JOptionPane`. Para

aprovecharnos de este límite, necesitamos situar el frame interno en el glass pane del frame donde aparece el desktop pane.

Si hemos trabajado con frames internos, sabemos que normalmente añadimos los frames internos, es decir ejemplares de `JInternalFrame`, a un desktop pane, es decir, un ejemplar de `JDesktopPane`. Un desktop pane es un panel de capas que maneja múltiples frames internos solapados. Glass pane es parte del panel raíz con el que trata una ventana de alto nivel. Un root pane se compone de tres partes (el `glass pane`, el `layered pane`, y el `content pane`), y una cuarta parte opcional (la barra de menú). El `content pane` contiene los componentes visibles del `root pane`. La barra de menú opcional contiene los menús del `root pane`. Y el `layered pane` posiciona los contenidos del `content pane` y del **menu bar** opcional. El `glass pane` es útil en la intercepción de eventos que de otra forma podrían pasar a través de los componentes subyacentes.

Por eso, para repetir, podemos crear un diálogo modal de alguna forma en un frame interno creado por `JOptionPane`. Para hacer esto, ponemos el `internal frame` en el `glass pane` del marco donde aparezca el `desktop pane`. Esta técnica restringe la entrada sólo a ese marco especificado. El `internal frame` en este caso no es realmente modal. Para ser realmente modal, un `internal frame` necesita bloquearse una vez que se ha mostrado. Sin embargo, la aproximación no restringe la entrada a un sólo componente.

El primer paso de esta técnica es crear un diálogo dentro de un internal frame. `JOptionPane`, y luego usar los métodos `createInternalXXX` para crear y mostrar los componentes `message` necesarios. Por ejemplo, lo siguiente crea un mensaje de diálogo dentro de un `internal frame`:

```
JOptionPane optionPane = new JOptionPane();
optionPane.setMessage("Hello, World");
optionPane.setMessageType(
  JOptionPane.INFORMATION_MESSAGE);
JInternalFrame modal =
  optionPane.createInternalFrame(desktop, "Modal");
```

El siguiente paso es situar el componente en el `glass pane` de la ventana donde está localizado el `desktop pane`. El `glass pane` puede ser cualquier componente. Por eso, la forma más fácil de hacer esto es crear un `JPanel` transparente:

```
JPanel glass = new JPanel();
glass.setOpaque(false);
glass.add(modal);
frame.setGlassPane(glass);
glass.setVisible(true);
modal.setVisible(true);
```

Los últimos pasos son configurar el `glass pane` para que intercepte los eventos, y para ocultarse cuando el `internal frame` se cierre. Para que el `glass pane` intercepte eventos, debemos adjuntar un `MouseListener` y un `MouseMotionListener`. Para ocultar el `glass pane` cuando se cierra el `internal frame`, necesitamos adjuntar un `InternalFrameListener` al `internal frame`:

```
class ModalAdapter extends InternalFrameAdapter {
  Component glass;

  public ModalAdapter(Component glass) {
```

```
this.glass = glass;

// Associate dummy mouse listeners
// Otherwise mouse events pass through
MouseInputAdapter adapter = new MouseInputAdapter(){};
glass.addMouseListener(adapter);
glass.addMouseMotionListener(adapter);
    }

    public void internalFrameClosed(InternalFrameEvent e) {
glass.setVisible(false);
    }
  }
```

Aquí tenemos un programa que pone todas las piezas juntas. Crea un `JDesktopPane` con un sólo `internal frame`. En este marco hay un botón. Cuando se pulsa el botón, aparece el diálogo de mensaje que bloquea el `internal frame`. Mientras que está visible, no podemos pulsar el primer botoón. Una vez pulsado el boton **OK** de la ventana del mensaje, podemos interactúar con el primer `internal frame`.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

public class Modal  {

  static class ModalAdapter
extends InternalFrameAdapter {
    Component glass;

    public ModalAdapter(Component glass) {
this.glass = glass;

// Associate dummy mouse listeners
// Otherwise mouse events pass through
MouseInputAdapter adapter =
  new MouseInputAdapter(){};
glass.addMouseListener(adapter);
glass.addMouseMotionListener(adapter);
    }

    public void internalFrameClosed(
  InternalFrameEvent e) {
glass.setVisible(false);
    }
  }

  public static void main(String args[]) {
    final JFrame frame = new JFrame(
"Modal Internal Frame");
    frame.setDefaultCloseOperation(
```

```
JFrame.EXIT_ON_CLOSE);

    final JDesktopPane desktop = new JDesktopPane();

    ActionListener showModal =
  new ActionListener() {
public void actionPerformed(ActionEvent e) {

  // Manually construct a message frame popup
  JOptionPane optionPane = new JOptionPane();
  optionPane.setMessage("Hello, World");
  optionPane.setMessageType(
    JOptionPane.INFORMATION_MESSAGE);
  JInternalFrame modal = optionPane.
    createInternalFrame(desktop, "Modal");

  // create opaque glass pane
  JPanel glass = new JPanel();
  glass.setOpaque(false);

  // Attach modal behavior to frame
  modal.addInternalFrameListener(
    new ModalAdapter(glass));

  // Add modal internal frame to glass pane
  glass.add(modal);

  // Change glass pane to our panel
  frame.setGlassPane(glass);

  // Show glass pane, then modal dialog
  glass.setVisible(true);
  modal.setVisible(true);

  System.out.println("Returns immediately");
}
    };

    JInternalFrame internal =
new JInternalFrame("Opener");
    desktop.add(internal);

    JButton button = new JButton("Open");
    button.addActionListener(showModal);

    Container iContent = internal.getContentPane();
    iContent.add(button, BorderLayout.CENTER);
    internal.setBounds(25, 25, 200, 100);
    internal.setVisible(true);

    Container content = frame.getContentPane();
    content.add(desktop, BorderLayout.CENTER);
    frame.setSize(500, 300);
```

```
    frame.setVisible(true);
  }
}
```

Crear Modal Internal Frames, Aproximación II

Aunque la aproximación anterior proporcionan marcos internos que bloqueaban las entradas de otros marcos internos, los marcos no eran realmente modales. Para ser realmente modal, un `internal frame` necesita bloquearse una vez que se ha mostrado. El `internal frame` de arriba no hace esto.

Para poder hacer un `internal frame` realmente modal, debemos hacernos con el despacho de eventos cuando el frame se ha mostrado. Esto sería además de mostrar el frame en el `glass pane`. Todavía podemos usar el `JOptionPane` para crear los diálogos de mensaje y de entrada, pero también necesitamos añadir algún comportamiento que normalmente es manejado por nosotros cuando usamos uno de los métodos `showInternalXXX`. Debido a la necesidad de un comportamiento personalizado, y cuándo es necesitado, es necesario crear una subclase de `JInternalFrame`. Hacer esto también nos permite mover dentro de la subclase mucho del comportamiento que hacíamos anteriormente en el `ActionListener`.

La mayoría del trabajo que necesitamos para crear un `internal frame` realmente modal implica completar el constructor de la subclase. Simplemente copiando el código del `ActionListener` de la aproximación anterior al constructor proporcionamos un marco de trabajo en el que podemos construirlo. Pasandolo en el `JRootPane`, podemos usar este `JInternalFrame` modal en un `JApplet` así como en un `JFrame`.

```
  public ModalInternalFrame(String title,
JRootPane rootPane, Component desktop,
JOptionPane pane) {
    super(title);

    // create opaque glass pane
    final JPanel glass = new JPanel();
    glass.setOpaque(false);

    // Attach mouse listeners
    MouseInputAdapter adapter =
new MouseInputAdapter(){};
    glass.addMouseListener(adapter);
    glass.addMouseMotionListener(adapter);

    // Add in option pane
    getContentPane().add(pane, BorderLayout.CENTER);

    // *** Remaining code to be added here ***

    // Add modal internal frame to glass pane
    glass.add(this);

    // Change glass pane to our panel
    rootPane.setGlassPane(glass);
```

```
    // Show glass pane, then modal dialog
    glass.setVisible(true);
  }
```

Observa que el único código no copiado desde el `ActionListener` es la llamada final a `setVisible(true)` del `internal frame`.

Alguna de las otras tareas que realizan los métodos `showInternalXXX` de `JOptionPane` incluyen la sección de un diálogo de cierre una vez que el botón se ha seleccionado (o se ha introducido una entrada), y algunas tareas relacionadas con la apariencia que casi siempre tienen algo que ver con el tamaño. Debido a que no estámos usando el método `showInternalXXX`, debemos realizar otras tareas nosotros mismos.

La forma de configurar el cierre del `internal frame` es adjuntar un `PropertyChangeListener` al `option pane`. En el `JOptionPane`, cuando se selecciona un botón o se introduce una entrada, dispara la generación de un `PropertyChangeEvent`. Podemos cerrar el marco cuando suceda este evento. Aquí tenemos el código para este comportamiento:

```
    // Define close behavior
    PropertyChangeListener pcl =
  new PropertyChangeListener() {
public void propertyChange(
    PropertyChangeEvent event) {
  if (isVisible() &&
    (event.getPropertyName().equals(
JOptionPane.VALUE_PROPERTY) ||
     event.getPropertyName().equals(
JOptionPane.INPUT_VALUE_PROPERTY))) {
    try {
setClosed(true);
    } catch (PropertyVetoException ignored) {
    }
    ModalInternalFrame.this.setVisible(false);
    glass.setVisible(false);
  }
}
    };
    pane.addPropertyChangeListener(pcl);
```

Hay tres tareas relacionadas con la apariencia que necesitamos realizar. Los diálogos de marcos internos están definidos para tener un borde diferente al borde normal de los `internal frames`. Por eso, necesitamos configurar una propiedad cliente para el marco. La segunda tarea es inicializar el tamaño y la posición del `internal frame`. Podríamos codificar "a mano" un tamaño (sin embargo, el siguiente código centra el marco). La última tarea es marcar el `internal frame` como el seleccionado. Aquí está el código encargado de realizar estas tres tareas:

```
    // Change frame border
    putClientProperty("JInternalFrame.frameType",
"optionDialog");
```

```
    // Size frame
    Dimension size = getPreferredSize();
    Dimension rootSize = desktop.getSize();

    setBounds((rootSize.width - size.width) / 2,
  (rootSize.height - size.height) / 2,
  size.width, size.height);
    desktop.validate();
    try {
setSelected(true);
    } catch (PropertyVetoException ignored) {
    }
```

Añadiendo estos dos bloques de código en el medio de nuestro constructor completamos la inicialización de la subclase de `JInternalFrame`.

Lo último que tenemos que hacer es tomar el despacho de eventos después de que el `internal frame` se haya mostrado. Normalmente el despacho de eventos se maneja en la clase `EventQueue`. Sin embargo, como estámos bloqueando el thread de manejo de eventos cuando hacemo modal al `internal frame`, el `EventQueue` nunca verá los eventos, Por eso debemos reemplazar su funcionalidad.

Para poder despachar eventos nosotros mismos, todo lo que tenemos que hacer es copiar el código del método `dispatchEvent()` de `EventQueue`. Si el `internal frame` se hace visible desde un thread distinto al thread de despacho de eventos, incluso ni necesitamos copiar el código del método `dispatchEvent()`. En este caso, todo lo que tenemos que hacer es llamar a `wait()` para bloquear. Luego, cuando se cierre el marco, necesitamos ser notificados. Aquí está el código de despacho de eventos:

```
   public void setVisible(boolean value) {
    super.setVisible(value);
    if (value) {
startModal();
    } else {
stopModal();
    }
  }

  private synchronized void startModal() {
    try {
if (SwingUtilities.isEventDispatchThread()) {
  EventQueue theQueue =
    getToolkit().getSystemEventQueue();
  while (isVisible()) {
    AWTEvent event = theQueue.getNextEvent();
    Object source = event.getSource();
    if (event instanceof ActiveEvent) {
((ActiveEvent)event).dispatch();
    } else if (source instanceof Component) {
((Component)source).dispatchEvent(
  event);
    } else if (source instanceof
```

```
  MenuComponent) {
((MenuComponent)source).dispatchEvent(
  event);
    } else {
System.err.println(
  "Unable to dispatch: " + event);
    }
  }
} else {
  while (isVisible()) {
    wait();
  }
}
    } catch (InterruptedException ignored) {
    }
  }

  private synchronized void stopModal() {
    notifyAll();
  }
```

Aquí hay un ejemplo que lo pone todo junto. En lugar de simplemente mostrar un mensaje de diálogo, pide al usuario una respuesta a una pregunta **Si/No** cuando se muestra el `internal frame` modal. Observa que todo lo que tenemos que hacer después de crear el `internal frame` es mostrarlo:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
import java.beans.*;

public class ModalInternalFrame extends JInternalFrame {

  public ModalInternalFrame(String title, JRootPane
rootPane, Component desktop, JOptionPane pane) {
    super(title);

    // create opaque glass pane
    final JPanel glass = new JPanel();
    glass.setOpaque(false);

    // Attach mouse listeners
    MouseInputAdapter adapter =
new MouseInputAdapter(){};
    glass.addMouseListener(adapter);
    glass.addMouseMotionListener(adapter);

    // Add in option pane
    getContentPane().add(pane, BorderLayout.CENTER);

    // Define close behavior
```

```
    PropertyChangeListener pcl =
  new PropertyChangeListener() {
public void propertyChange(PropertyChangeEvent
    event) {
  if (isVisible() &&
    (event.getPropertyName().equals(
JOptionPane.VALUE_PROPERTY) ||
     event.getPropertyName().equals(
JOptionPane.INPUT_VALUE_PROPERTY))) {
    try {
setClosed(true);
    } catch (PropertyVetoException ignored) {
    }
    ModalInternalFrame.this.setVisible(false);
    glass.setVisible(false);
  }
}
    };
    pane.addPropertyChangeListener(pcl);

    // Change frame border
    putClientProperty("JInternalFrame.frameType",
"optionDialog");

    // Size frame
    Dimension size = getPreferredSize();
    Dimension rootSize = desktop.getSize();

    setBounds((rootSize.width - size.width) / 2,
  (rootSize.height - size.height) / 2,
   size.width, size.height);
    desktop.validate();
    try {
setSelected(true);
    } catch (PropertyVetoException ignored) {
    }

    // Add modal internal frame to glass pane
    glass.add(this);

    // Change glass pane to our panel
    rootPane.setGlassPane(glass);

    // Show glass pane, then modal dialog
    glass.setVisible(true);
  }

  public void setVisible(boolean value) {
    super.setVisible(value);
    if (value) {
startModal();
    } else {
```

```
stopModal();
    }
  }

  private synchronized void startModal() {
    try {
if (SwingUtilities.isEventDispatchThread()) {
  EventQueue theQueue =
    getToolkit().getSystemEventQueue();
  while (isVisible()) {
    AWTEvent event = theQueue.getNextEvent();
    Object source = event.getSource();
    if (event instanceof ActiveEvent) {
((ActiveEvent)event).dispatch();
    } else if (source instanceof Component) {
((Component)source).dispatchEvent(
  event);
    } else if (source instanceof MenuComponent) {
((MenuComponent)source).dispatchEvent(
  event);
    } else {
System.err.println(
  "Unable to dispatch: " + event);
    }
  }
} else {
  while (isVisible()) {
    wait();
  }
}
    } catch (InterruptedException ignored) {
    }
  }

  private synchronized void stopModal() {
    notifyAll();
  }

  public static void main(String args[]) {
    final JFrame frame = new JFrame(
"Modal Internal Frame");
    frame.setDefaultCloseOperation(
JFrame.EXIT_ON_CLOSE);

    final JDesktopPane desktop = new JDesktopPane();

    ActionListener showModal =
  new ActionListener() {
Integer ZERO = new Integer(0);
Integer ONE = new Integer(1);
public void actionPerformed(ActionEvent e) {
```

```
  // Manually construct an input popup
  JOptionPane optionPane = new JOptionPane(
    "Print?", JOptionPane.QUESTION_MESSAGE,
    JOptionPane.YES_NO_OPTION);

  // Construct a message internal frame popup
  JInternalFrame modal =
    new ModalInternalFrame("Really Modal",
    frame.getRootPane(), desktop, optionPane);

  modal.setVisible(true);

  Object value = optionPane.getValue();
  if (value.equals(ZERO)) {
    System.out.println("Selected Yes");
  } else if (value.equals(ONE)) {
    System.out.println("Selected No");
  } else {
    System.err.println("Input Error");
 }
}
    };

    JInternalFrame internal =
new JInternalFrame("Opener");
    desktop.add(internal);

    JButton button = new JButton("Open");
    button.addActionListener(showModal);

    Container iContent = internal.getContentPane();
    iContent.add(button, BorderLayout.CENTER);
    internal.setBounds(25, 25, 200, 100);
    internal.setVisible(true);

    Container content = frame.getContentPane();
    content.add(desktop, BorderLayout.CENTER);
    frame.setSize(500, 300);
    frame.setVisible(true);
  }
}
```

Para aprender más sobre `internal frames`, `root panes`, y su `glass pane` puedes ver la lecciones del **Java Tutorial** "How to Use Internal Frames" y "How to Use Root Panes".

Copyright y notas de la traducción

Nota respecto a la traducción

El original en inglés de la presente edición de los JDC Tech Tips fue escrita por Glen McCluskey, la traducción no oficial fue hecha por Juan A. Palos (Ozito), cualquier sugerencia o corrección hágala al

correo tutorjava@hotmail.com , sugerencia respecto a la edición original a mailto:jdc-webmaster@sun.com

Nota (Respecto a la edición via email)

Sun respeta su tiempo y su privacidad. La lista de correo de la Conexión del desarrollador Java se usa sólo para propósitos internos de Sun Microsystems(tm). Usted ha recibido este email porque se ha suscrito a la lista. Para desuscribirse vaya a la página de suscripciones, desmarque casilla apropiada y haga clic en el botón Update.

Suscripciones

Para suscribirse a la lista de correo de noticias de la JDC vaya a la página de suscripciones, elija los boletines a los que quiera suscribirse, y haga clic en Update.

Realimentación

¿Comentarios?, envie su sugerencias a los Consejos Técnicos de la JDC a mailto:jdc-webmaster@sun.com

Archivos

Usted encontrará las ediciones de los Consejos Técnicos de la JDC (en su original en inglés) en http://java.sun.com/jdc/TechTips/index.html

Copyright

Enlaces a sitios fuera de Sun

Los Consejos Técnicos de la JDC pueden dar enlaces a otros sitios y recursos. Ya que Sun no tiene control sobre esos sitios o recursos usted reconoce y acepta que Sun no es responsable por la disponibilidad de tales sitios o recursos, y no se responsabiliza por cualquier contenido, anuncios , productos u otros materiales disponibles en tales sitios o recursos. Sun no será responsable, directa o indirectamente, por cualquier daño o pérdida causada o supuestamente causada por o en relación con el uso de o seguridad sobre cualquier tal contenido, bienes o servicios disponibles en o através de cualquier sitio o recurso.

El original en Ingles de esta edición de los Consejos técnicos fue escrita por Glen McCluskey.

JDC Tech Tips December 20, 2001

Foros de debate

Nuestras zonas

Visual Basic

Buscar

Lista de foros | Nuevo asunto

Asunto	Usuario	Fecha	Respuestas	Última respuesta
Formato de botones	daniloide	20 de octubre de 2002	0	-
Pregunta Sobre Comando Shell	EnErGiE	20 de octubre de 2002	0	-
Creación de Objetos	gaby	16 de octubre de 2002	4	20 de octubre de 2002
SI TENEIS ALGUNA DUDA	ALBERTO	5 de septiembre de 2002	7	20 de octubre de 2002
Cadenas de más de 60.000 char	J.P.L.	8 de octubre de 2002	1	19 de octubre de 2002
Existe algún api del microsoft internet explorer para VBasic?????	panoramixxx	10 de octubre de 2002	1	19 de octubre de 2002
Máscara en un TextBox	eperez	9 de octubre de 2002	3	19 de octubre de 2002
tipo de dato booleano (Sí/No)	paky	15 de octubre de 2002	3	19 de octubre de 2002
Esperar a que un proceso termine	Empar	18 de octubre de 2002	2	19 de octubre de 2002
seguridad ante todo	manolo	18 de octubre de 2002	2	19 de octubre de 2002
Enviar pulsos al Lpt1	Esli Moreno	17 de octubre de 2002	2	19 de octubre de 2002
controlar que no haya registros duplicados	paky	29 de agosto de 2002	5	19 de octubre de 2002
Ajecutar en la Barra de Tareas	Wilbert	18 de octubre de 2002	1	19 de octubre de 2002
DAO y ADO	Rubén	17 de octubre de 2002	1	18 de octubre de 2002

Conexiones telefónicas con VB	Ernesto	10 de octubre de 2002	1	18 de octubre de 2002
VB A -> Java Servlet	fromer	18 de octubre de 2002	0	-
Tengo un problema muy grave (Visual Basic)	ansoba	31 de agosto de 2002	2	18 de octubre de 2002
VISUAL BASIC EN PAGINAS WEB	ISAEL ARENAS	12 de octubre de 2002	3	18 de octubre de 2002
Urgente por favor	licjrz	15 de octubre de 2002	3	18 de octubre de 2002
Busco programador de visual Basic. Pago en euros.	Ivan	9 de octubre de 2002	1	17 de octubre de 2002

« **1** 2 3 4 5 6 7 8 9 10 »

Principio Página

Foros de debate

Nuestras zonas

Visual FoxPro

Lista de foros | Nuevo asunto

Buscar

Asunto	Usuario	Fecha	Respuestas	Última respuesta
Visual Fox vs Macros en Excel	Osvaldo	16 de octubre de 2002	0	-
Disenador de Archivos	carlosrubio	16 de octubre de 2002	0	-
PROGRAMA EN VISUAL FOXPRO	Valentin Guzman	5 de septiembre de 2002	27	16 de octubre de 2002
necesito programa de acceso a cualquier sistema	rtafur_flores	14 de octubre de 2002	0	-
Novato	rtafur_flores	14 de octubre de 2002	0	-
URGENTE!!!!!!!!!!!!!	Paul	11 de octubre de 2002	0	-
Fopen	marcosnoya	10 de octubre de 2002	0	-
Ayuda..Como crear archivos Memo..	saul	10 de octubre de 2002	0	-
El mejor RAD	Lucas	9 de octubre de 2002	0	-
Problemas con Vistas	Carlos	9 de octubre de 2002	0	-
Ayuda porfavor!	Esteban	3 de septiembre de 2002	4	9 de octubre de 2002
Cual es su opinion ??	Lucas	8 de octubre de 2002	0	-
Função para obter o numero de dias uteis	Martin	8 de octubre de 2002	0	-
Servidor Com .Exe	renar	2 de octubre de 2002	0	-
programa de seguros en fox pro	alejandro	1 de octubre de 2002	2	2 de octubre de 2002

Mostrar datos en Grid con rango de fechas	juan	2 de octubre de 2002	0	-
COMO UTILIZAR MENUS Y FORMULARIO EN UNA MISMA PANTALLA	DAVID	30 de septiembre de 2002	1	1 de octubre de 2002
Imprimir Campo Numerico de color, con If/else/endif	MarcarvaS	1 de octubre de 2002	0	-
Ing Computación	Arturo Aquino	30 de septiembre de 2002	0	-
De .txt o .dbf a VFP 7.0	ozdapava	28 de septiembre de 2002	0	-

« **1** 2 3 »

Principio Página

Foros de debate

Nuestras zonas

HTML

Lista de foros | Nuevo asunto

Buscar

Asunto	Usuario	Fecha	Respuestas	Última respuesta
Formularios	Mónica	17 de octubre de 2002	0	-
a ver q os parece esta web, gracias	zapamix	17 de octubre de 2002	0	-
aspecto barras de desplazamiento	newhtm	14 de octubre de 2002	2	17 de octubre de 2002
Refresco en netscape6	Javi	16 de octubre de 2002	0	-
Editar Fuentes	nomecopies	15 de octubre de 2002	0	-
web gratis sin publicidad	sandrups	4 de octubre de 2002	2	12 de octubre de 2002
¿como agregar un chat a una pagina?	antonio ortega	10 de octubre de 2002	1	11 de octubre de 2002
pdf	maria	11 de octubre de 2002	0	-
showmodaldialog	Jordi	11 de octubre de 2002	0	-
forms html	loren	9 de octubre de 2002	2	9 de octubre de 2002
autorun.inf	miguel	22 de mayo de 2002	3	9 de octubre de 2002
cambiar estilo a un select	Manuel	3 de octubre de 2002	2	9 de octubre de 2002
Problemas con marcos	Belen	9 de octubre de 2002	0	-
Listas de Correo - Alguien sabe como puedo obtener direcciones de mail	Juan	22 de junio de 2002	1	7 de octubre de 2002
Formularios	Giovanni	3 de octubre de 2002	0	-

convertir html a formato pdf	max	26 de septiembre de 2002	3	3 de octubre de 2002
Salto de página	Scherzo	11 de abril de 2002	1	2 de octubre de 2002
Como cargar un frame especial	barahona	2 de octubre de 2002	0	-
eliminar barra de menu	sandrups	27 de septiembre de 2002	1	2 de octubre de 2002
COMO GRABO UN CAMPO DE LA FORMA EN UN ARCHIVO TEXTO	juan manuel	6 de septiembre de 2002	3	2 de octubre de 2002

« **1** 2 3 4 5 6 7 8 9 10 11 … »

Principio Página

Foros de debate

Nuestras zonas

Java (básico)

Buscar

Asunto	Usuario	Fecha	Respuestas	Última respuesta
abrir y guardar formatos gráficos desde Java	jluisgil	13 de octubre de 2002	1	20 de octubre de 2002
Empezar a trabajar con Java	pedro	9 de octubre de 2002	3	20 de octubre de 2002
Division entre cero	Jorge Molina	22 de junio de 2002	2	20 de octubre de 2002
AYUUUUDAAAAAAAA	BelThaSar	18 de octubre de 2002	4	20 de octubre de 2002
zoom	luisperez_graterol	12 de octubre de 2002	2	19 de octubre de 2002
Ayuda!! con coordenadas	Luis Eduardo	19 de octubre de 2002	0	-
churro de errores	r0466	15 de octubre de 2002	6	19 de octubre de 2002
Manual de VisualAge en castellano	GARCIAJ	19 de febrero de 2002	54	19 de octubre de 2002
LINUX-JAVA-POSTGRESQL	lobmb	17 de octubre de 2002	1	18 de octubre de 2002
ICONO DE JAVA	Garciaj	16 de octubre de 2002	3	18 de octubre de 2002
Manifest en los JAR	garion	17 de octubre de 2002	2	18 de octubre de 2002
String a array de caracteres	lab	17 de octubre de 2002	2	18 de octubre de 2002

Lanzar un HELP!	Luis Alberto	17 de octubre de 2002	0	-
¿Convertir int y short a byte para almacenarlo en tabla de bytes?	Raygmar	30 de abril de 2002	4	17 de octubre de 2002
Sobre la generación de números aleatorios en Java	omardf18	15 de octubre de 2002	3	17 de octubre de 2002
Consulta JTable	Rosi	17 de octubre de 2002	0	-
Error cargando el driver jdbc para mysql	irbo	16 de octubre de 2002	1	17 de octubre de 2002
Con datos hacer graficas estadisticas en java	sandra	15 de octubre de 2002	1	17 de octubre de 2002
Classpath en linux, y versión del kit de desarrollo	josé Luis	15 de octubre de 2002	1	17 de octubre de 2002
Como obtener el path	ramonal	16 de octubre de 2002	0	-

« **1** 2 3 4 5 6 7 8 9 10 11 ... »

Principio Página

Foros de debate

Nuestras zonas

PHP

Lista de foros | Nuevo asunto

Buscar

Asunto	Usuario	Fecha	Respuestas	Última respuesta
Extracción de un número de un linea de un txt	biglione	19 de octubre de 2002	0	-
duda urgente	gladys	19 de octubre de 2002	0	-
ayudaaa por favor	Ismael	18 de octubre de 2002	0	-
No funciona	laangie	17 de octubre de 2002	1	17 de octubre de 2002
Les ruego me ayuden	rocio	17 de octubre de 2002	0	-
PAGINAR CAMPOS MEMO DE FOX	rflores	17 de octubre de 2002	0	-
Problema al Escribir un TXT!!!!...AYUDA PLS!!	LEGION	14 de octubre de 2002	2	17 de octubre de 2002
problemas con actualizaciones en php	cristina	17 de octubre de 2002	1	17 de octubre de 2002
Un problema de diseño	LOUIAH	17 de octubre de 2002	1	17 de octubre de 2002
php con sql server	jose	18 de marzo de 2002	4	16 de octubre de 2002
problemas con oracle y php	crimax	16 de octubre de 2002	0	-
Impresion de papel en php	LANTE	15 de octubre de 2002	1	15 de octubre de 2002

Error en programacion.com/php	Grogie	14 de octubre de 2002	1	14 de octubre de 2002
¿Como controlar errores?	MetalManiac	14 de octubre de 2002	3	14 de octubre de 2002
PHP COMO?	antares	13 de octubre de 2002	2	14 de octubre de 2002
PHP y MySQL	Juanma	14 de octubre de 2002	0	-
ASP o PHP?	MetalManiac	7 de octubre de 2002	3	12 de octubre de 2002
'Solapas' en lugar de menú?	Roberto	11 de octubre de 2002	1	11 de octubre de 2002
Como ejecutar una query de SAP en PHP?	rcanavate	9 de octubre de 2002	1	10 de octubre de 2002
ayuda por favor, soy nuevo en php	Jonathan	9 de octubre de 2002	3	10 de octubre de 2002

« **1** 2 3 4 5 6 7 8 9 10 11 ... »

Principio Página

Foros de debate

Nuestras zonas

XML

Lista de foros | Nuevo asunto

Buscar

Asunto	Usuario	Fecha	Respuestas	Última respuesta
XML y bases de datos	yomisma	17 de octubre de 2002	0	-
Como genero un link desde XML a traves de XSL con Asp??	Victor	13 de octubre de 2002	0	-
XML con asp y base de datos	Luis Alonso	23 de septiembre de 2002	0	-
Tratamiento Ficheros XML con Java	Mir	18 de septiembre de 2002	0	-
XML y Archivos Planos	groblesmonroy	18 de junio de 2002	2	13 de septiembre de 2002
mi proyecto...	larraitz	4 de septiembre de 2002	0	-
Parseo incorrecto XML-XSL	tito	20 de agosto de 2002	2	27 de agosto de 2002
La palabra clave xsl:sort no se puede utilizar en el espacio de nombres http://www.w3.org/	vagalume	25 de julio de 2002	1	21 de agosto de 2002
Tablas en XML	Marcos	6 de agosto de 2002	1	20 de agosto de 2002
generar HTML	Diego	7 de agosto de 2002	1	20 de agosto de 2002
como visualiso lo que he echo en xml	eduardo	7 de agosto de 2002	1	8 de agosto de 2002
Schemas	casi416	7 de agosto de 2002	0	-
XSL. Atributo de un XML en un TextBox	Mariano	6 de agosto de 2002	1	7 de agosto de 2002
Schemas y DTDs	casi416	5 de agosto de 2002	0	-

rutas dinamicas	Txiki_3	5 de agosto de 2002	0	-
superindices en xml	eli	25 de enero de 2002	3	2 de agosto de 2002
Coneccion a SQL server	aleblake	23 de julio de 2002	1	2 de agosto de 2002
DTDs	casi416	23 de julio de 2002	0	-
leer un xml desde el final hasta el principio	Vagalume	22 de julio de 2002	0	-
Xml y bases de datos	dani	7 de julio de 2002	2	22 de julio de 2002

« **1** 2 3 4 5 »

Principio Página

Foros de debate

Nuestras zonas

Bases de datos y SQL

Lista de foros | Nuevo asunto

Buscar

Asunto	Usuario	Fecha	Respuestas	Última respuesta
consulta en access	LATER	18 de octubre de 2002	0	-
¿Se pueden crear bases de datos con el mysql?	Wro	18 de octubre de 2002	0	-
Actualizar datos de un formulario, help!	Mox	18 de octubre de 2002	0	-
Duda de VB i Acces	Marc	18 de octubre de 2002	1	18 de octubre de 2002
base de datos	gachas	15 de octubre de 2002	2	18 de octubre de 2002
ayudaaaaa	Ismael	18 de octubre de 2002	0	-
ERROR EN BD	Marco Mendoza	17 de octubre de 2002	0	-
Como realizar esta consulta	Rodrigo	17 de octubre de 2002	0	-
migracion de datos (DBase a SQL)	Rafal D.J.	17 de octubre de 2002	0	-
migracion de datos (DBase a SQL)	Rafal D.J. Novoa Hdez	17 de octubre de 2002	0	-
informes de access y mail	txus2002	17 de octubre de 2002	1	17 de octubre de 2002
Convertidor de Fox a SQL o Acces	mva	15 de octubre de 2002	1	16 de octubre de 2002
Access	J.Luis	16 de octubre de 2002	0	-
Pregunta sobre access	J.Luis	16 de octubre de 2002	0	-
¿para qué sirven y como se utilizan los CHECKS?	skanat	15 de mayo de 2002	1	16 de octubre de 2002

access	enrique	15 de octubre de 2002	4	16 de octubre de 2002
Cómo vincular access con visual basic	Graciela	16 de octubre de 2002	0	-
Acceso remoto a una BD en FOX-convirtierndo a SQL o Acces	mva	15 de octubre de 2002	0	-
access	jose ramon	15 de octubre de 2002	2	15 de octubre de 2002
parbulitos	markoss	15 de octubre de 2002	0	-

« **1** 2 3 4 5 6 7 8 9 10 11 … »

Principio Página

Foros de debate

Nuestras zonas

ASP

Lista de foros | Nuevo asunto

Buscar

Asunto	Usuario	Fecha	Respuestas	Última respuesta
cuando voy a usar ASP?????	freddy	19 de octubre de 2002	0	-
Pago por Internet (validar tarjetas, etc...)	J.L.Dengra	19 de octubre de 2002	0	-
INSERCION EN BASE DE DATOS ACCESS	Thehighlander	16 de octubre de 2002	1	18 de octubre de 2002
seguridad con asp	ivan	6 de junio de 2002	2	18 de octubre de 2002
chilisoft asp gratis!!!!!!!	fofo	17 de octubre de 2002	0	-
Busqueda por ASP	Daniel	17 de octubre de 2002	0	-
Como crear HTML de manera Dinámica con codigo asp .NET	Luis Ernesto "Canuto"	11 de octubre de 2002	1	15 de octubre de 2002
estudiante	paty	12 de octubre de 2002	1	15 de octubre de 2002
Objeto Session	matarese	9 de octubre de 2002	1	15 de octubre de 2002
JavaScript	yareaj	15 de octubre de 2002	0	-
Cantidad de visitas	Nicomix	15 de octubre de 2002	1	15 de octubre de 2002
Error de ASP e IIS	hugofajardo	3 de julio de 2002	1	15 de octubre de 2002
Corregir código sin necesidad de conectarme	Carlos	14 de octubre de 2002	1	15 de octubre de 2002

Conexión a KmySQL con ASP	Ricardo	15 de octubre de 2002	0	-
Como corro asp desde mi ordenador con windows me	Hyunkel	14 de octubre de 2002	0	-
Request.Form de un campo de texto con varias líneas	Sergio	11 de octubre de 2002	2	14 de octubre de 2002
Diseñar páginas ASP sin conexión	matilda	10 de octubre de 2002	4	11 de octubre de 2002
Redireccionar e-mail con ASP	kuntent	17 de septiembre de 2002	1	11 de octubre de 2002
comparar fechas	ivanillo	10 de octubre de 2002	4	11 de octubre de 2002
Nombre de campo	o|O	10 de octubre de 2002	1	11 de octubre de 2002

« **1** 2 3 4 5 6 7 8 9 10 11 ... »

Principio Página

Foros de debate

Nuestras zonas

Servidores de Aplicaciones J2EE

Lista de foros | Nuevo asunto

Buscar

Asunto	Usuario	Fecha	Respuestas	Última respuesta
problemas con ejb y jboss	Yicart	18 de octubre de 2002	0	-
Tomcat como Servicio NT	Edgar	10 de septiembre de 2002	2	17 de octubre de 2002
Lanzar un HELP	Luis Alberto	16 de octubre de 2002	0	-
conectar con EJBs	Loco	14 de octubre de 2002	1	16 de octubre de 2002
problemas con setEntityResolver	guille	11 de octubre de 2002	1	15 de octubre de 2002
tengo que reiniciar tomcat 4.1 al modifcar un class !!!	jperez	27 de septiembre de 2002	1	15 de octubre de 2002
Colaboración Apache+Tomcat	LN	15 de octubre de 2002	0	-
Acentos y Ñ's en Weblogic 6.1	Lillo	9 de octubre de 2002	0	-
Documentacion de arquitecturas multicapa	Cesar	8 de octubre de 2002	0	-
apache2+tomcat	izas	8 de octubre de 2002	0	-
inconsistent thread	kusturica	3 de octubre de 2002	0	-
Websphere	JESUSHC	26 de septiembre de 2002	3	2 de octubre de 2002
MS Visual Studio . Net Enterprise Architect ES	MMvisualnet	1 de octubre de 2002	0	-
modificar el proceso de login/password de windows 2000	kaos	30 de septiembre de 2002	0	-

duda: jdbc+oracle+tomcat	guille	20 de septiembre de 2002	3	28 de septiembre de 2002
¿Qué hace que pueda ejecutar Java 2?	manuparres	31 de julio de 2002	1	26 de septiembre de 2002
JBuilder 7 con Weblogic	Raquel	26 de septiembre de 2002	0	-
WebLogic class	JulenK	18 de septiembre de 2002	1	25 de septiembre de 2002
Tomcat no encuentra mis properties !!!	jperez	5 de septiembre de 2002	1	20 de septiembre de 2002
Tomcat y Apache en Win 2000	JPérez	19 de septiembre de 2002	1	20 de septiembre de 2002

« **1** 2 3 »

Principio Página

Programación en castellano

Documentacion | Recursos | Taller Java

Novedades y más información en nuestra lista de correo

Secciones

- Tutoriales
- Taller Java
- Recursos
- Formación
- Buscador
- Downloads
- Foros

Formación

J2EE: Aplicaciones avanzadas de Java para entornos profesionales

Curso a distancia ofrecido por la UNED.

bea Noticias

Bea y Sun crean un centro de soporte conjunto para clientes

Bea optimiza Weblogic Jrockit 7.0 para todos los servidores Intel

Bea systems y Hp fortalecen su alianza

PALM y BEA se alian para ofrecer Web Services

BEA refuerza su compromiso con LINUX como plataforma estratégica

Más

En nuestro sitio

Búsqueda avanzada

Buscar

Documentación propia

Tutoriales | Taller Java

Desplegar Servlets y Aplicaciones Web en Apache Tomcat y BEA WebLogic Server

Por: Juan A. Palos (Ozito)

En este artículo revisaremos los pasos implicados en el despliegue de un servlet, describe cómo tomar un servlet y crear una aplicación Web - tanto en formato expandido como en un WAR. Ilustra cómo desplegar una aplicación Web en Apache Tomcat y en WebLogic Server 6.0, un completo servidor de aplicaciones J2EE. Publicado el *12 de Octubre 2002*

New2Java 4: Leer y Escribir Ficheros y Manejar Excepciones

Por: Juan A. Palos (Ozito)

Cuarta entrega de este tutorial para principiantes en Java, en la que veremos como leer y escribir ficheros, como manejar los errores y excepciones en Java y como utilizar el controlador de distribución de componentes GUI "BoxLayout" Publicado el *9 de Octubre 2002*

Operaciones Avanzadas con Bases de Datos Usando JDBC 3.0

Por: Juan A. Palos (Ozito)

Segundo de la serie de tutoriales en los que veremos entre otras cosas las "Sentencias Preparadas" y los "Tipos de Datos Avanzados". Publicado el *4 de Octubre 2002*

Manejar Conexiones a Bases de Datos con JDBC 3.0

Por: Juan A. Palos (Ozito)

Primero de una serie de tutoriales en los que veremos operaciones avanzadas con bases de datos, utilizando fuentes de datos y almacenes de conexiones Publicado el *15 de Junio 2002*

Suplementos a New 2 Java

Por: Juan A. Palos (Ozito)

En este tutorial os iremos ofreciendo "suplementos" del Tutorial sobre iniciación a Java "New2Java" en sus páginas encontraremos explicaciones de las clases y conceptos Java más importantes así como unos pequeños ejercicios con su solución. Publicado el *27 de Abril 2002*

El API Apache SOAP v2.2

Por: Juan A. Palos (Ozito)

Siguiendo con los tutoriales sobre los marcos de trabajo más interesantes que encontramos, esta vez os entregamos uno sobre el API Apache SOAP, utilizado para enviar y recibir mensajes SOAP

Novedades

Tutoriales

- Tutoriales básicos
- New 2 Java
- Tutoriales avanzados
- Java y XML
- Serv. Aplicaciones
- FrameWorks Java
- Otros tutoriales

Taller Java

- Artículos propios
- Traducción JDC Tips

Foros

- Java Básico
- Servlets-JSP
- Java & XML
- Serv. Aplicaciones

Otras Webs

entre aplicaciones. Publicado el *13 de Abril 2002*

JDC Tech Tips 22 de Enero de 2002

Por: Juan A. Palos (Ozito)

Traducción de los JDC Tech Tips del 22 de enero de 2002 en los que se tratan los siguientes temas:

- Recuperar Mail con el API JavaMai
- Trabajar con el API Java Communications (puertos serie y paralelo).

Publicado el *13 de Abril 2002*

Comparación de APIs Java para XML

Por: Juan A. Palos (Ozito)

Comparativa de los diferentes APIs Java utilizados para trabajar en XML, con una página dedicada a trucos para mejorar el rendimiento de nuestras aplicaciones Java que trabajan con datos y documentos XML. Publicado el *5 de Abril 2002*

JDC Tech Tips 10 de Enero de 2002

Por: Juan A. Palos (Ozito)

Traducción de los JDC Tech Tips del 10 de enero de 2002 en los que se tratan los siguientes temas:

- Usar Excepciones
- Dimensionar Texto con FontMetrics

Publicado el *5 de Abril 2002*

Manual Básico de Struts

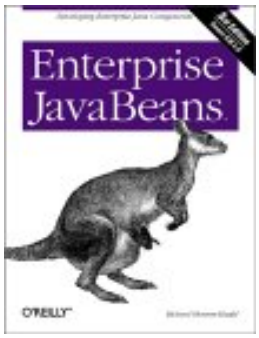
Por: Javier Antoniucci

Manual básico sobre el API Struts de Apache Publicado el *3 Abril 2002*

Tutoriales | Taller Java

Libros

Java and XSLT



Enterprise JavaBeans, 3rd Edition...

Recursos

Enlaces a direcciones en las que encontrara recursos y aplicaciones que haran más fácil la elaboración de tus aplicaciones Java.

Más

12 - Octubre 2002
Nuevo artículo Desplegar Servlets y Aplicaciones Web enTomcat y WebLogic Server en la zona Taller Java.

9 - Octubre 2002
New2Java: Leer y Escribir Ficheros y Manejar Excepciones en la sección New 2 Java del TutorJava

4 - Octubre 2002
Operaciones Avanzadas de Bases de Datos con JDBC 3.0 en el TutorJava

15 - junio 2002
Manejar Conexiones a Bases de Datos con JDBC 3.0 en el TutorJava

23 - abril 2002
Suplementos a New 2 Java en la sección New 2 Java del TutorJava

13 - abril 2002
El API Apache SOAP v2.2 en la sección Marcos de Trabajo del TutorJava

13 - Abril 2002
Nuevo artículo JDC Tech Tips del 22 de Enero de 2002 en la zona Taller Java.

5 - abril 2002
Comparativa de APIs Java para XML en el TutorJava

5 - Abril 2002
Nuevo artículo JDC Tech Tips del 10 de Enero de 2002 en la zona Taller Java.

3 - Abril 2002
Manual Básico de Struts en el TutorJava

23 - Marzo 2002
Nuevo artículo JDC Tech Tips del 20 de Diciembre de 2001 en la zona Taller Java.

Java Hispano

Java en Telepolis

Java World

Servlets.com

On Java

Recomendamos

23 - Marzo 2002
Nuevo artículo JDC Tech Tips del 4 de Diciembre de 2001 en la zona Taller Java.

17 - Marzo 2002
Introdución a iPlanet Application Server en la sección Servidores Web del TutorJava

16 - Marzo 2002
Nuevo artículo JDC Tech Tips del 20 de Noviembre de 2001 en la zona Taller Java.

Principio Página

Anillo Java: Anterior | Al Azar | Siguiente (visita las mejores páginas en castellano)

Programación en castellano

Html en castellano

Documentación | Recursos | Taller Web

Novedades y más información en nuestra lista de correo

Hemos ganado

Cursos propios

- HTML 4.0
- DHTML
- JavaScript 1.2
- XML
- XHTML
- FrontPage
- Dreamweaver NUEVO

Foros

- HTML
- Javascript
- XML

Y muchos más

Libros

En Amazón:

Otros

- Mapa del Web
- Lista de correo

Otros Webs

En nuestro sitio

Búsqueda avanzada

Buscar

Documentación

Recopilación de documentación con la que aprendera a realizar sus páginas Web.

- Tutoriales propios
- Manuales, listas de correo, news, en la Red
- Libros, revistas.
- y mucho más...

Monográficos sobre:

- XML
- XSL, XSLT y XPath
- JavaScript
- WAP, WML, etc.

Taller Web

Sección en la que periodicamente iremos publicando **trucos y consejos**.

Lo último...

- Crear un fichero robots.txt. NUEVO
- Protección con contraseñas (III): Varios usuarios.
- Validación del número de cuenta (CCC).

Recursos

Enlaces a direcciones en las que encontrara recursos y aplicaciones que haran más fácil la elaboración de tus páginas.

FORMACIÓN

Si quieres un curso profesional de HTML, Javascript, Dreamweaver o Flash, visita en nuestro canal de formación la oferta de Ciberaula.

Formación en nuevas tecnologías

PDF ARTÍCULOS

Todos los artículos del 2000 en un fichero PDF.

EMPLEO

Ofertas de trabajo en Tecnologías de la Información.

Novedades

16 - Septiembre 2002
Nuevo **artículo** en el **Taller Web**: Crear un fichero robots.txt.

20 - Mayo 2002
Nuevo **artículo** en el **Taller Web**: Protección con contraseñas (III): Varios usuarios.

5 - Abril 2002
Nuevo **capítulo** del **Curso de Javascript 1.2**: Expresiones regulares.

6 - Diciembre 2001
Nuevo **artículo** en el **Taller Web**: Validación del número de cuenta (CCC).

18 - Noviembre 2001
Inaugurados los **foros** de Programación en castellano.

7 - Octubre 2001
Quinta y última entrega del curso de Dreamweaver.

- Recursos gratis
- Java en castellano
- ASP en castellano
- WebEstilo
- DWEs

30 - Septiembre 2001
Cuarta entrega del curso de Dreamweaver.

Principio Página

Trabajando con PHP y ficheros

Autor: Alejandro Almunia

Leer comentarios (4) | Escribir comentario | Puntuación: ■■■■■ (12 votos) Vota

1 . Abriendo un fichero de texto, lectura, escritura y añadido
2 . Subir ficheros al servidor
3 . Forzar descarga de ficheros al navegador

Recomendar este tutorial | Estadísticas

En este nuevo tutorial, voy a tratar el tema de los ficheros y como se trabaja en ellos desde PHP. No va a ser exhaustivo ni mucho menos, pretende proporcionaros las bases para que experimentéis por vuestra cuenta y riesgo. Vamos a aprender unas cuantas cosas útiles al respecto de los ficheros, así que, si estáis listos, empezamos. Para este tutorial solo váis a necesitar PHP, nada más (ni MySQL ni otra cosa).

Abriendo un fichero de texto, lectura, escritura y añadido

Lo primero que vamos a hacer es escribir un sencillo fichero de texto. Lo abriremos, escribiremos un par de líneas dentro de él y luego lo cerraremos. El código que realiza esto se puede ver a continuación.

```
<?

#Abrimos el fichero en modo de escritura
$DescriptorFichero = fopen("fichero_prueba.txt","w");

#Escribimos la primera línea dentro de él
$string1 = "Esta es la primera línea de texto\r\n";
fputs($DescriptorFichero,$string1);

#Escribimos la segunda línea de texto
$string2 = "Y esta es la segunda línea de texto\r\n";
fputs($DescriptorFichero,$string2);

#Cerramos el fichero
fclose($DescriptorFichero);

?>
```

Así pues, el script anterior lo único que hace es abrir un fichero llamado **fichero_prueba.txt**, y escribe dentro de él dos líneas de texto. Os habréis fijado en el \r\n de detrás de las líneas de texto, en las variables $string1 y $string2. Esto se debe a que, si no estuviese puesto, el programa escribiría todo seguido. Para comprobarlo, quitadlo y ejecutad de nuevo el programa. Con solo \n no sirva, al menos en mi sistema Windows 2000. :-) En Linux, basta con un \n.

Otra de las cosas importantes del anterior script es algo que quizás no hayamos visto de cerca. Fijémonos en la siguiente línea:

```
$DescriptorFichero = fopen("fichero_prueba.txt","w");
```

La función fopen sirve para abrir un fichero en un **modo**. Los modos pueden ser seis y son los siguientes. Además de listarlos, explicaré las diferencias (no siempre tan obvias), al respecto de ellos.

Modo de apertura	Qué significa
r	Modo de solo lectura. Se abre el fichero y el cursor se coloca al principio del mismo, permitiendo leerlo hasta el final.
r+	Modo de lectura/escritura. Se abre el fichero y el cursor se coloca al principio del mismo,permitiendo leer o escribir en el fichero.
w	Modo de solo escritura. Se crea el fichero si no existiese, y, si existe, se borra todo su contenido, se sitúa el cursor al principio del fichero permitiendonos escribir.
w+	Modo de escritura/lectura. Si el fichero no existe, se crea, y, si existiese, se borra todo su contenido, se sitúa el cursor al principio del fichero permitiéndonos escribir y leer.
a	Modo de añadido. Abre el fichero, sitúa el cursor al final del mismo y permite escribir. Si el fichero no existe, lo crea, pero, en caso de existir, no borra su contenido.
a+	Modo de añadido/lectura. Sitúa el cursor al final del fichero y permite escribir y leer. Si el fichero no existe, lo crea, pero, si existe, no borra su contenido.

Así pues, estos son los seis modos de abrir un fichero. Vamos ahora a ver un ejemplo en código del uso de los mismos. El siguiente script va a hacer las siguientes tareas:

- Crear un fichero y escribir en él dos líneas de texto.
- Abrir el fichero de nuevo, esta vez en modo añadido, y escribir otras dos líneas.

Es poco, pero la lectura de ficheros la veremos al final de esta parte del tutorial. De momento, aquí está el código del script de PHP.

escribir2.php

```
<?

#Abrimos el fichero en modo de escritura
$DescriptorFichero = fopen("fichero_prueba.txt","w");

#Escribimos la primera línea dentro de él
$string1 = "Esta es la primera línea de texto\r\n";
fputs($DescriptorFichero,$string1);

#Escribimos la segunda línea de texto
$string2 = "Y esta es la segunda línea de texto\r\n";
fputs($DescriptorFichero,$string2);

#Cerramos el fichero
fclose($DescriptorFichero);
```

```
#Volvemos a abrir el fichero, esta vez en modo de añadir
$Descriptor2 = fopen("fichero_prueba.txt","a");

#Añadimos la tercera línea de texto
fputs($Descriptor2,"Esta es la tercera línea, añadida con modo \"a\"\r\n");

#Añadimos la cuarta línea de texto
fputs($Descriptor2,"Esta es la cuarta línea, añadida con modo \"a\"\r\n");

#Cerramos el fichero
fclose($Descriptor2);

?>
```

Como podéis comprobar si abrís el fichero recién creado, éste contiene cuatro líneas, dos de ellas escritas con modo "w" y otras dos con modo "a". Si ya tenéis más o menos claro como funciona, vamos a pasar a ver dos funciones muy útiles para leer ficheros de texto: `fgets()` y `feof()`. A través de `fgets()` podemos leer una línea del fichero de texto cada vez. `feof()` sirva para saber si hemos llegado al final del fichero. Para ver como funcionan, crearemos un script que leerá el fichero que hemos creado con los dos scripts anteriores.

leer.php

```
<?

#Abrimos el fichero en modo lectura
$DescriptorFichero = fopen("fichero_prueba.txt","r");

#Hasta que no lleguemos al final del fichero
while(!feof($DescriptorFichero)){

    #Capturamos 4096 caracteres dentro de la línea,
    #o menos si hay un retorno de carro antes
    #(\r\n en Win32, \r en UNIX)
    $buffer = fgets($DescriptorFichero,4096);

    #Soltamos el texto, añadiendo <BR> detrás
    echo $buffer."<BR>";
}

?>
```

Como véis, este script lee el fichero de texto línea a línea y lo va mostrando en el navegador. La función `feof()` devuelve TRUE cuando ha llegado al final del fichero. `fgets()`, va, pues, leyendo linea a linea y almacenándolo en una varible llamada `$buffer`.

Ahora vamos a ver como funcionan los modos w+, r+ y a+. Veréis que son diferentes de los anteriores en el sentido de que permiten dos operaciones, y tambien en el sentido de como tratan los ficheros. Empezaremos con w+. El siguiente script explica qué es lo que hace este modo con los ficheros.

leer_wplus.php

```
<?

#Abrimos el fichero en modo w+
$Descriptor1 = fopen("nuevo_fichero.txt","w+");
```

```
#Vamos a escribir un par de líneas en el fichero
fputs($Descriptor1,"Esta es la primera línea de texto\r\n");
fputs($Descriptor1,"Esta es la segunda línea de texto\r\n");

#Ahora cerraremos el fichero
fclose($Descriptor1);

#Volvemos a abrirlo en modo w+
$Descriptor2 = fopen("nuevo_fichero.txt","w+");

#Escribimos un par de líneas
fputs($Descriptor2,"Esta es la tercera línea de texto\r\n");
fputs($Descriptor2,"Esta es la cuarta línea de texto\r\n");

#Volvemos al principio del fichero
rewind($Descriptor2);

#Vamos leyendo líneas y mostrándolas
while(!feof($Descriptor2)){

    $buffer = fgets($Descriptor2,4096);
    echo $buffer."<BR>";

}

#Cerramos el fichero
fclose($Descriptor2);

?>
```

Como véis, al ejecutarlo, el resultado es el siguiente:

```
Esta es la tercera línea de texto
Esta es la cuarta línea de texto
```

¿Por qué no aparecen la primera y la segunda línea escritas? Observemos lo que hemos hecho. Primero abrimos el fichero y escribimos dentro de él dos líneas de texto. Tras esto, lo cerramos y lo volvemos a abrir, en modo w+. Este modo **BORRA EL CONTENIDO ANTERIOR** del fichero, por lo que en este solo aparecen las dos últimas líneas. Como véis, se puede utilizar este modo para leer desde el fichero con `fgets()`.

Ahora vamos a ver un ejemplo con r+. Vamos a crear un script que haga lo mismo que el anteriro, pero en vez de abrir los ficheros con w+, los abrirá con r+.

leer_rplus.php

```
<?

#Abrimos el fichero en modo w+
$Descriptor1 = fopen("nuevo_fichero.txt","w");

#Vamos a escribir un par de líneas en el fichero
fputs($Descriptor1,"Esta es la primera línea de texto\r\n");
fputs($Descriptor1,"Esta es la segunda línea de texto\r\n");

#Ahora cerraremos el fichero
fclose($Descriptor1);
```

```
#Volvemos a abrirlo en modo w+
$Descriptor2 = fopen("nuevo_fichero.txt","r+");

#Escribimos un par de líneas
fputs($Descriptor2,"Esta es la tercera línea de texto\r\n");
fputs($Descriptor2,"Esta es la cuarta línea de texto\r\n");

#Volvemos al principio del fichero
rewind($Descriptor2);

#Vamos leyendo líneas y mostrándolas
while(!feof($Descriptor2)){

    $buffer = fgets($Descriptor2,4096);
    echo $buffer."<BR>";

}

#Cerramos el fichero
fclose($Descriptor2);

?>
```

Si ejecutáis el script, quizás observéis, sorprendidos, que el resultado es el mismo que en el anterior. Pero lo que ha sucedido, en cambio, no es lo mismo. Vamos a anlizarlo por partes. Primero, hemos abierto el fichero en modo w (escritura), para meter dos líneas en el fichero. Tras esto, lo cerramos, y lo abrimos en modo r+ (lectura/escritura). Al abrirlo de este modo, el cursor se sitúa al principio del fichero, por lo que al escribir las siguientes dos líneas, borra el contenido de las dos líneas anteriroes.. Antes de mostrar el contenido del fichero usamos la función `rewind()`, que rebobina el cursor hasta el principio del fichero. Para añadir al final de fichero, necesitamos el modo a+, como veremos en el siguiente ejemplo.

leer_aplus.php

```
<?
#Abrimos el fichero en modo w+
$Descriptor1 = fopen("nuevo_fichero.txt","w+");

#Vamos a escribir un par de líneas en el fichero
fputs($Descriptor1,"Esta es la primera línea de texto\r\n");
fputs($Descriptor1,"Esta es la segunda línea de texto\r\n");

#Ahora cerraremos el fichero
fclose($Descriptor1);

#Volvemos a abrirlo en modo w+
$Descriptor2 = fopen("nuevo_fichero.txt","a+");

#Escribimos un par de líneas
fputs($Descriptor2,"Esta es la tercera línea de texto\r\n");
fputs($Descriptor2,"Esta es la cuarta línea de texto\r\n");

#Volvemos al principio del fichero
rewind($Descriptor2);

#Vamos leyendo líneas y mostrándolas
while(!feof($Descriptor2)){

    $buffer = fgets($Descriptor2,4096);
```

```
    echo $buffer."<BR>";

}

#Cerramos el fichero
fclose($Descriptor2);

?>
```

El resultado de este método es el esperado. Se abre el fichero en modo escritura y se insertan las dos líneas de texto. Se cierra este descriptor, y se abre otro en modo a+. El cursor se sitúa al final del fichero, y comienza a añadir el texto. El resultado son las cuatro líneas dentro del fichero de texto.

Esto es todo en cuanto a modos de apertura. En la siguiente sección vamos a ver como subir ficheros al servidor, algo muy útil cuando se trata de páginas web.

Subir ficheros al servidor

Para ejemplificar la subida de archivos al servidor, vamos a ver un script de ejemplo. El script tiene dos partes; la primera, el formulario, en el que se introduce el fichero a cargar, y la segunda parte, en la que se procesa la subida y se informa al usuario del éxito o fracaso de la carga.

upload.php

```
<?

if(!isset($cargar)){

?>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>:: Formulario de carga de ficheros ::</TITLE>
</HEAD>

<BODY>

<FORM NAME="elForm" METHOD="POST"
      ACTION="<? echo $PHP_SELF; ?>?cargar=1"
      ENCTYPE="multipart/form-data">

    <TABLE WIDTH="80%" STYLE="font-family:Arial;font-size:9pt;">

    <TR>
        <TD ALIGN="LEFT"><INPUT TYPE="FILE" NAME="elFichero"></INPUT></TD>
    </TR>

    <TR>
        <TD ALIGN="LEFT"><INPUT TYPE="SUBMIT" VALUE="Subir el fichero">
    </TR>

    </TABLE>

</FORM></BODY></HTML>

<?
```

```
}

#Aquí realizamos la carga del fichero
if(isset($cargar)){

    #Le damos al fichero un nuevo nombre
    $nuevositio = "nuevo_fichero_caragado.000";

    #Lo copiamos
    if(!copy($elFichero,$nuevositio)){
        echo "NO SE HA PODIDO SUBIR EL FICHERO";
    }
    else{
        echo "FICHERO SUBIDO CON ÉXITO";
    }

}

?>
```

Como vemos, el script realiza la carga del fichero. Para rizar un poco más el rizo os propongo lo siguiente: un script que suba un número de ficheros introducido por el usuario. Se compondrá de tres partes. La primera, en la que se pregunta al usuario el número de ficheros que desea introducir. La segunda, en la que aparecen todos los campos de tipo fichero, y la tercera, en la que se procesa la carga. El código es como sigue:

upload_multiple.php

```
<?

/* Este script se encarga de subir múltiples ficheros al servidor. */

#Formulario en el que se pregunta el número de ficheros
if(!isset($fich) && !isset($cargar)){

    $txt ="<HTML><HEAD>\n";
    $txt.="<TITLE>:: ¿Cuántos ficheros quiere subir hoy? ::</TITLE>\n";
    $txt.="</HEAD><BODY>\n";
    $txt.="<FORM NAME=\"frmNumFicheros\"
                METHOD=\"POST\"
                ACTION=\"".$PHP_SELF."?fich=1\">\n";

    $txt.="<BR><BR><BR><BR>\n";
    $txt.="<DIV ALIGN=\"CENTER\">\n";
    $txt.="<INPUT TYPE=\"TEXT\" NAME=\"numFicheros\">\n";
    $txt.="      ";
    $txt.="<B>Introduce el número de ficheros</B>\n";
    $txt.="<BR><BR>\n";
    $txt.="<INPUT TYPE=\"SUBMIT\"
                 VALUE=\"Mostrar campos para subir ficheros\">\n";
    $txt.="<BR></DIV>\n";

    $txt.="</FORM></BODY></HTML>\n";

    echo $txt;
}
```

```
#Formulario en el que se muestran los campos tipo fichero
if(isset($fich)){

    $txt ="<HTML><HEAD>\n";
    $txt.="<TITLE>:: ¿Cuántos ficheros quiere subir hoy? ::</TITLE>\n";
    $txt.="</HEAD><BODY>\n";
    $txt.="<FORM ENCTYPE=\"multipart-form/data\"
                NAME=\"frmCargaFicheros\"
                METHOD=\"POST\"
ACTION=\"".$PHP_SELF."?cargar=1&cantidad=".$HTTP_POST_VARS["numFicheros"]."\">\n";

    for($i=0;$i<$HTTP_POST_VARS["numFicheros"];$i++){

        $txt.="<INPUT TYPE=\"FILE\" NAME=\"fichero_$i\"><BR>\n";

    }

    $txt.="<INPUT TYPE=\"SUBMIT\" VALUE=\"cargar\">\n";

    $txt.="</FORM></BODY></HTML>\n";


    echo $txt;
}

#Parte que gestiona el proceso de carga
if(isset($cargar)){


    for($n=0;$n<$cantidad;$n++){

        #Creamos la "variable variable"
        $nomvar = "fichero_$n";
        $valvar = $;

        #Extraemos el nombre del fichero sin la ruta
        $nomfichero = basename($valvar);

        #Le damos al fichero su nombre, metiéndolo dentro del directorio /subidas
        $nuevositio = "subidas/".$nomfichero."";

        #Lo copiamos
        if(!copy($valvar,$nuevositio)){
            echo "NO SE HA PODIDO SUBIR EL FICHERO";
        }
        else{
            echo "FICHERO SUBIDO CON ÉXITO";
        }
    }


}

?>
```

De este modo podemos cargar varios ficheros al mismo tiempo. Tendrás que crear el directorio /subidas manualmente. Vamos a termnar con esta sección y pasar a la siguiente, en la que se explica como forzar al cliente a descargarse el fichero en vez de verlo on-line.

Forzar descarga de ficheros al navegador

A veces puede ser interesante que el usuario se descargue el fichero en vez de verlo on-line. Para realizar esta operación, solo necesitamos utilizar el siguiente código que voy a explicar a continuación. El script consta de una sola parte. Vamos a descargarnos un fichero .html, en vez de verlo en el navegador. El nombre del fichero será prueba_descarga.html. El código es como sigue:

descargar.php

```
<?

function Descargar($ElFichero){

    $TheFile = basename($ElFichero);

    header( "Content-Type: application/octet-stream");
    header( "Content-Length: ".filesize($ElFichero));
    header( "Content-Disposition: attachment; filename=".$TheFile."");
    readfile($ElFichero);
}

Descargar("prueba_descarga.html");

?>
```

Como ves, el script se ejecuta y el fichero, pese a ser HTML, e interpretable por el navegador, es forzado a ser descargado, igual que si hubiéramos pulsado el botón derecho.

Y con esto termino este tutorial. Espero que os sea útil, aunque se que es corto. Es posible que en otro tutorial próximo me extienda más sobre este tema (sobre todo si mis conocimeintos aumentan, jeje).

Descargate los ficheros del artículo

Leer comentarios (4) | Escribir comentario | Puntuación: ■■■■■ (12 votos) Vota

Últimos comentarios [Subir]

Muchas gracias. (25/09/2002)
Por Salvador

Te agradezco la claridad en los ejemplos

una mejora? (16/09/2002)
Por mikil

basado en este ejemplo se me plantean dos cuestiones:
1.- como puedo preasignar el valor a la variable que recoge el nombre del fichero? He probado con value y me presenta el campo en blanco
2.- puedo subir un fichero sin que el usuario sepa donde esta? o sea, yo se que fichero es y donde esta, pues lo subo al servidor y lo dejo en su sitio. Es esto posible

Excelente..!! (13/09/2002)
Por Daniel Ramos

Pues la verdad no tengo conocimientos en PHP, pero si en HTML javascript y C++, Por lo que aprender PHP se me has aun mas interesante aprenderlo... Por el momento estoy biendo el ambiente de php ya que Desconosco hasta el compilador..

SALUDOS..!!

Gracias (11/09/2002)
Por manuel

Gracias por el artículo me ha resultado muy util.

Recomendar este tutorial | Estadísticas

Principio Página

Registrarse en Programación en castellano

Si quieres participar en foros, recibir las últimas novedades de programación en castellano mediante correo electrónico o acceder a la documentación en formato PDF o ZIP, deberas registrarte. Para ello, rellena el siguiente formulario teniendo en cuenta que los campos señalado por (*) son obligatorios.

Nombre: (*)

Apellidos: (*)

E-mail: (*)

País: AFGANISTAN (*)

Usuario: (*)

Contraseña: (*)

Repetir contraseña: (*)

Domicilio:

Número: Piso / planta / escalera / puerta

Código postal:

Población:

Provincia: (*) Usuarios de España

Provincia:

Sexo: ○ Hombre ○ Mujer

Enviar

Principio Página

Formulario de contacto

Si desea ponerse en contacto con Programación en castellano S.L., por favor rellene el siguiente formulario. Recuerde que para realizar consultas existen unos foros de discusión donde les responderán mucho mejor y más rápido de lo que nunca podríamos lograr nosotros solos:

Su dirección de correo electrónico:

Motivo de su consulta:

-- Elegir opción --

Su mensaje:

Enviar Borrar

Principio Página

Datos legales

Programacion en Castellano, S.L.
B13344544
Cl. Mesones, 9
13640 Herencia
Ciudad Real
España

Teléfono 902 333 932 (no se da ningun tipo de soporte por telefono)
Fax. 926 574037

Email: webmaster@programacion.com

Principio Página

Búsqueda

En nuestro sitio

Buscar

Se pueden buscar frases completas encerrandolas entre comillas dobles (") y usar los operadores lógicos AND, OR y NOT. Por defecto, buscar más de una palabra delimitadas por espacios equivale a utilizar el operador AND.

Principio Página

Cursos.

Bases de datos

5 cursos

Las mejores plumas (bueno... teclados) escriben para Programación en castellano sobre temas de interés para el programador.

- Oracle (1)
- SQL (1)
- Teoría de bases de datos@ (1)

Operaciones avanzadas con JDBC y Java

Leer comentarios (5) | Puntuación: ■■■■■ (3 votos)

Operaciones Avanzadas con Bases de Datos Usando JDBC **Por IBM.**

Manejar Conexiones a Bases de Datos con JDBC 3.0

Leer comentarios (22) | Puntuación: ■■■■ (37 votos)

Operaciones Avanzadas con Bases de Datos Usando JDBC **Por IBM.**

Acceso a Bases de Datos [JDBC]

Leer comentarios (149) | Puntuación: ■■■■■ (126 votos)

Este tutorial presenta los tópicos necesarios para la programación de Acceso a Bases de Datos en Java con JDBC **Por Sun.**

Principio Página

Cursos.

Nuestras zonas

Entornos de desarrollo

1 curso

Las mejores plumas (bueno... teclados) escriben para Programación en castellano sobre temas de interés para el programador.

- Visual Basic (1)

Principio Página

Cursos. Herramientas

Nuestras zonas

2 cursos

Las mejores plumas (bueno... teclados) escriben para Programación en castellano sobre temas de interés para el programador.

- Desarrollo web (2)

Principio Página

Cursos.

Internet

26 cursos

Las mejores plumas (bueno... teclados) escriben para Programación en castellano sobre temas de interés para el programador.

Nuestras zonas

- ASP (4)
- Flash (2)
- HTML (2)
- PHP (3)
- SVG y VML (1)
- WAP (1)
- XML (11)
- XSL, XSLT y Xpath (2)
- Java@ (64)
- Javascript@ (1)

Principio Página

Cursos.

Lenguajes de script

4 cursos

Las mejores plumas (bueno... teclados) escriben para Programación en castellano sobre temas de interés para el programador.

- Javascript (1)
- Perl (1)
- Python (2)

Principio Página

Cursos.

Nuestras zonas

Lenguajes orientados a objeto

65 cursos

Las mejores plumas (bueno... teclados) escriben para Programación en castellano sobre temas de interés para el programador.

- C# (1)
- Java (64)

Principio Página

Cursos.

Nuestras zonas

Otros lenguajes

3 cursos

Las mejores plumas (bueno... teclados) escriben para Programación en castellano sobre temas de interés para el programador.

- Prolog (3)

Principio Página

Cursos.

Nuestras zonas

Sistemas operativos

1 curso

Las mejores plumas (bueno... teclados) escriben para Programación en castellano sobre temas de interés para el programador.

- Linux (1)

Principio Página

Cursos.

Nuestras zonas

Teoría

1 curso

Las mejores plumas (bueno... teclados) escriben para Programación en castellano sobre temas de interés para el programador.

- Teoría de bases de datos (1)

Principio Página

Artículos.

Nuestras zonas

Bases de datos

2 artículos

Las mejores plumas (bueno... teclados) escriben para Programación en castellano sobre temas de interés para el programador.

- MySQL (1)

APIs XML para Bases de Datos

Leer comentarios (4) | Puntuación: ■■■■ (17 votos)

Uso de las APIs XML SAX y DOM para acceder a bases de datos. **Por Sun.**

Principio Página

Artículos.

Nuestras zonas

Entornos de desarrollo

2 artículos

Las mejores plumas (bueno... teclados) escriben para Programación en castellano sobre temas de interés para el programador.

- Visual Basic (1)
- Visual C++ (1)

Principio Página

Artículos. Herramientas

Nuestras zonas

2 artículos

Las mejores plumas (bueno... teclados) escriben para Programación en castellano sobre temas de interés para el programador.

- Desarrollo web (1)
- Editores (1)

Principio Página

Artículos.

Nuestras zonas

Internet

50 artículos

Las mejores plumas (bueno... teclados) escriben para Programación en castellano sobre temas de interés para el programador.

- ASP (17)
- CSS (2)
- Generales (1)
- HTML (5)
- PHP (10)
- SVG y VML (2)
- WAP (1)
- XML (7)
- XSL, XSLT y Xpath (5)
- Java@ (27)
- Javascript@ (22)

Principio Página

Artículos.

Nuestras zonas

Lenguajes de script

23 artículos

Las mejores plumas (bueno... teclados) escriben para Programación en castellano sobre temas de interés para el programador.

- Javascript (22)
- Python (1)

Principio Página

Artículos.

Nuestras zonas

Lenguajes imperativos

1 artículo

Las mejores plumas (bueno... teclados) escriben para Programación en castellano sobre temas de interés para el programador.

- Ensamblador (1)

Principio Página

Artículos.

Nuestras zonas

Lenguajes orientados a objeto

27 artículos

Las mejores plumas (bueno... teclados) escriben para Programación en castellano sobre temas de interés para el programador.

- Java (27)

Principio Página

Foros de debate

Nuestras zonas

Buscar

C / C++

Lista de foros | Nuevo asunto

Asunto	Usuario	Fecha	Respuestas	Última respuesta
necesito codigo fuente	yo_	19 de octubre de 2002	0	-
Manejo de sockets bajo c++6	JOSE	18 de octubre de 2002	0	-
Por favor necesito una ayuda	César	28 de mayo de 2002	3	18 de octubre de 2002
Ayuda! Como configuro Turbo 3.1 for windows	JuanJo	18 de octubre de 2002	0	-
editor c/c++ hilos linux	jaqy	18 de octubre de 2002	0	-
mis propias librerias	nabucco	17 de octubre de 2002	0	-
Sonido en C	romeroandres	11 de octubre de 2002	1	16 de octubre de 2002
Pasenme programas en c y c++	Juan	16 de octubre de 2002	0	-
Como crear una libreria en "c"	arkan	16 de octubre de 2002	0	-
arboles b+	Alguien	15 de octubre de 2002	0	-
problema con este programa	rafa	13 de octubre de 2002	2	14 de octubre de 2002
Alguien sabe cual es el algoritmo de la funcion que genera numeros aleatorios	Pepe	17 de septiembre de 2002	2	13 de octubre de 2002
Librería system.h	Jose Luis	11 de octubre de 2002	2	13 de octubre de 2002
Manejo del puerto serie en C.	fafis	14 de marzo de 2002	3	11 de octubre de 2002

manual de dev-c++ 4 en español???	josepe8219	11 de octubre de 2002	0	-
Visual C ejecutando Visual Basic	Jordi	11 de octubre de 2002	0	-
rs232 y c	manolo	10 de octubre de 2002	0	-
Error de linkado al utilizar la libreria windows.h	David	8 de octubre de 2002	1	10 de octubre de 2002
Libro de C++	.:Gaby:.	30 de septiembre de 2002	3	10 de octubre de 2002
modificar el proceso de login/password de windows 2000	nuevo	2 de octubre de 2002	1	10 de octubre de 2002

« **1** 2 3 4 5 6 7 8 9 10 »

Principio Página

Foros de debate

Nuestras zonas

Delphi / Kylix / C++ Builder

Buscar

Lista de foros | Nuevo asunto

Asunto	Usuario	Fecha	Respuestas	Última respuesta
Busco programadores para proyecto SIN FINES DE LUCRO...	Mauricio	31 de marzo de 2002	3	19 de octubre de 2002
creación de dbf desde delphi para ser importados desde otros programas como WinFax	triniti	24 de septiembre de 2002	1	19 de octubre de 2002
Necesito ayuda para hacer algoritmo,estructura,programa	Sayra Veronica Pulido M.	6 de octubre de 2002	1	19 de octubre de 2002
Ayuda por favor	liuxus	19 de octubre de 2002	0	-
!!!!!!!!!!! urgente ¡¡¡¡¡¡¡¡¡	liuxus	19 de octubre de 2002	0	-
consulta ADOQuery	liuxus	19 de octubre de 2002	0	-
handles de ventanas!!	elia	14 de octubre de 2002	0	-
red, base de datos en delphi	walter	12 de febrero de 2002	2	12 de octubre de 2002
AnsiString a una Función	GreenGo	30 de septiembre de 2002	1	9 de octubre de 2002
Como puedo copiar toda la pantalla al portapapeles?	Wolverick	9 de octubre de 2002	0	-
El mejor RAD	Lucas	9 de octubre de 2002	0	-
Datos por paralelo. En Delphi	Douglas J.	9 de octubre de 2002	0	-
Denme su opinion	Lucas	8 de octubre de 2002	0	-
errores en bases d datos	Ja	7 de octubre de 2002	0	-

Distribuir BCB 1	Flip	4 de octubre de 2002	0	-
integrales	rogert	1 de octubre de 2002	2	4 de octubre de 2002
Operar con variables int y double en C++Builder	David Urpí	6 de febrero de 2002	2	29 de septiembre de 2002
asistring	Eduadan	26 de septiembre de 2002	0	-
ASOCIACION OLIVARES 2000	OLIVARES 2000	24 de septiembre de 2002	0	-
C++ Builder para Linux	Esteban	9 de enero de 2002	1	20 de septiembre de 2002

« **1** 2 3 4 5 6 »

Principio Página

Foros de debate

Nuestras zonas

General

Lista de foros | Nuevo asunto

Buscar

Asunto	Usuario	Fecha	Respuestas	Última respuesta
Registrar LOG en BD	Andrea	17 de octubre de 2002	0	-
a ver q os parece esta web, gracias	zapamix	17 de octubre de 2002	0	-
INFERNO y LIMBO	Don	16 de octubre de 2002	0	-
Urgente!!!!!!!!!!!!!	PEKE	16 de octubre de 2002	0	-
¿Diferencias entre torres ATX y AT?	ÇR	10 de octubre de 2002	0	-
¿EXE de Vb6 sin instalacion?	Anonimo	10 de octubre de 2002	0	-
qué programa utilizar para crear un tutorial de mi aplicación	anaa	9 de octubre de 2002	0	-
Crear un icono	eperez	2 de octubre de 2002	1	3 de octubre de 2002
Mundo Internet.	Azazel	23 de septiembre de 2002	1	1 de octubre de 2002
Archivos desconocidos	Jazmin E.L.	28 de septiembre de 2002	1	28 de septiembre de 2002
busqueda de programa para tarjetas	jsarrio	28 de septiembre de 2002	0	-
Terminal Server	emolinah	28 de septiembre de 2002	0	-
AYUDA : COMANDO BUSQUEDA DESAPARECIÓ..	R.	27 de septiembre de 2002	0	-
lenguajes de computadora	matador	21 de septiembre de 2002	1	23 de septiembre de 2002

CD/DVD-ROM	Xiber	22 de septiembre de 2002	0	-
Abrir archivos .dat	Pedro Quijano	12 de marzo de 2002	2	22 de septiembre de 2002
Se me abren cientos de paginas web	Nieves	19 de septiembre de 2002	0	-
EN CANARIAS HAY DOS PROVINCIAS!!	chacho	18 de septiembre de 2002	0	-
Alguien me puede explicar como trabajan los instaladores de programas	Angel40	12 de agosto de 2002	3	18 de septiembre de 2002
Ayuda con Flash	Santi	18 de septiembre de 2002	0	-

« **1** 2 3 4 5 6 7 8 9 »

Principio Página

Contacto - Datos legales

Foros de debate

Nuestras zonas

Java (Servlets y JSP)

Lista de foros | Nuevo asunto

Buscar

Asunto	Usuario	Fecha	Respuestas	Última respuesta
Como imprimir tablas dinamicas con jsp	Cris	19 de octubre de 2002	0	-
manejo del log de tomcat	Gustavo Gutierrez A.	18 de octubre de 2002	0	-
VB A -> Java Servlet	fromer	18 de octubre de 2002	0	-
Pasar el valor de una variable a java desde html o javascript	jolin21	17 de octubre de 2002	0	-
problemas para llamar servlets y jsp	mario	15 de octubre de 2002	0	-
Clases gráficas y servidores X	gerarunha	15 de octubre de 2002	0	-
Problemas con parámetros	rbld	8 de octubre de 2002	2	15 de octubre de 2002
Cerrar y abrir el servidor Tomcat continuamente	Q	14 de octubre de 2002	1	15 de octubre de 2002
Problemas al compilar un servlet	lab	10 de octubre de 2002	1	15 de octubre de 2002
JSP	stibcasa	11 de diciembre de 2001	1	15 de octubre de 2002
Cerrar una pag HTML creada con un servlet sin que pida confirmacion	Gema Nuñez	9 de octubre de 2002	4	14 de octubre de 2002
JSP y Oracle 8i/9i	Root	12 de octubre de 2002	0	-
Alguien me puede ayudar con JSP	silvita	11 de octubre de 2002	1	11 de octubre de 2002

Como puede ejecutar servelts	David	9 de octubre de 2002	1	11 de octubre de 2002
Problemas al compilar un servlet II	lab	10 de octubre de 2002	1	11 de octubre de 2002
JSP	Rosi	10 de octubre de 2002	1	11 de octubre de 2002
Instalación de tomcat	David	11 de octubre de 2002	1	11 de octubre de 2002
Carga de ficheros	PaMaY	9 de enero de 2002	3	10 de octubre de 2002
Quieres Visual Age for Java 4?, WebSphere Studio?	jasistemas	9 de octubre de 2002	0	-
Alguien me puede explicar esto por favor!!!!	lab	9 de octubre de 2002	1	9 de octubre de 2002

« **1** 2 3 4 5 6 7 8 9 10 11 ... »

Principio Página

Foros de debate

Nuestras zonas

Java y XML

Lista de foros | Nuevo asunto

Buscar

Asunto	Usuario	Fecha	Respuestas	Última respuesta
excel a xml	Edgar	8 de octubre de 2002	1	17 de octubre de 2002
¿Como hago para hacer correr soap en Apache tomcat?	Tito	10 de octubre de 2002	0	-
XSLFO, Como poder escribir texto en vertical	Jose Manuel	13 de septiembre de 2002	1	10 de octubre de 2002
JavaWebStart	arielelmejor	9 de octubre de 2002	0	-
compilar/ejecutable	ratitallorona	2 de octubre de 2002	0	-
modificar el proceso de login/password de windows 2000	kaos	30 de septiembre de 2002	0	-
como borrar en un jtree	maria	18 de septiembre de 2002	1	25 de septiembre de 2002
XML en servidor y applet	jbarreiro	24 de septiembre de 2002	3	25 de septiembre de 2002
Applets y xml	pakico	25 de septiembre de 2002	1	25 de septiembre de 2002
llamadas al S.O desde Java	Juan	6 de septiembre de 2002	1	23 de septiembre de 2002
Base de Datos en XML con JAVA	Michael ET	22 de septiembre de 2002	0	-
¿Como envio un e-mail con la API Java Mail?,porfavooooooooor	elbercial23	30 de julio de 2002	2	19 de septiembre de 2002
Como hacer un compilador para sentencias sql en java?	César	18 de septiembre de 2002	0	-

Pasar xml a otro xml	srarroba	20 de enero de 2002	2	12 de septiembre de 2002
Hacer un compilador en Java	Marleny	9 de septiembre de 2002	0	-
libro xml & java de Oreilly	tanny	9 de agosto de 2002	1	2 de septiembre de 2002
Parseo incorrecto XML-XSL	tito	20 de agosto de 2002	3	27 de agosto de 2002
Parsear validando con esquemas	casi416	27 de agosto de 2002	0	-
variables en xml?	Txiki_3	5 de agosto de 2002	1	21 de agosto de 2002
Impresion	Hugo	16 de mayo de 2002	3	17 de agosto de 2002

« **1** 2 3 4 5 »

Principio Página

Foros de debate

Nuestras zonas

Javascript

Lista de foros | Nuevo asunto

Buscar

Asunto	Usuario	Fecha	Respuestas	Última respuesta
decimales	jesus	26 de septiembre de 2002	1	10 de octubre de 2002
Convercion	Irma	9 de octubre de 2002	1	10 de octubre de 2002
paso de variable javascript	jesus	9 de octubre de 2002	0	-
Un obsequio de Wolverine para mis amigos del foro: "Reloj en la barra de estado"	Wolverine	24 de marzo de 2002	3	8 de octubre de 2002
Ayuda!!!!! Problema con los menus desplegables	marcos	8 de octubre de 2002	0	-
Problemon con los menus desplegables	Marcos	8 de octubre de 2002	0	-
Cambiar una imagen de un frame desde otro frame	POX	7 de diciembre de 2001	3	8 de octubre de 2002
Seleccionar texto dentro de un TextArea	Aramram	3 de octubre de 2002	1	7 de octubre de 2002
Crear un archivo txt en un localhost	Ivan	29 de septiembre de 2002	1	5 de octubre de 2002
Trabajar con Word	molin	25 de abril de 2002	1	5 de octubre de 2002
Descergar Ficheros	Dexter	4 de octubre de 2002	0	-
ocultar y desocultar botones	jesus	3 de octubre de 2002	1	3 de octubre de 2002

Menú desplegable (obsequio de Wolverine a mis amigos del foro)	Wolverine	23 de febrero de 2002	10	1 de octubre de 2002
javascript eliminar los saltos de linea de un textarea	forenai	28 de septiembre de 2002	1	30 de septiembre de 2002
comunicacion entre ventanas	inyaki	26 de septiembre de 2002	1	27 de septiembre de 2002
JAVA SCRIPT	Erika Alexandra Villabona León	19 de septiembre de 2002	2	27 de septiembre de 2002
Frames	Rastro	25 de septiembre de 2002	1	26 de septiembre de 2002
Problema con menu desplegable	mendrugo	23 de septiembre de 2002	2	25 de septiembre de 2002
problemas con netscape	Loreto	25 de septiembre de 2002	0	-
Combo editable	Juls	25 de septiembre de 2002	0	-

« **1** 2 3 4 5 6 7 8 9 10 11 ... »

Principio Página

Foros de debate

Nuestras zonas

Petición de foros nuevos

Lista de foros | Nuevo asunto

Buscar

Asunto	Usuario	Fecha	Respuestas	Última respuesta
foro j2me para moviles	mi_nick	14 de febrero de 2002	3	19 de octubre de 2002
Nuevo Foro De Scripting	]][a][n[u][b][i][s][[	17 de marzo de 2002	1	17 de octubre de 2002
Foros de Oracle Reports, Oracle Forms y PL/SQL	yvan_terry	15 de septiembre de 2002	1	16 de octubre de 2002
¿ PARA CUÁNDO UNO DE ASSEMBLER ME CAGO EN DIOS ?	Onit	9 de septiembre de 2002	3	16 de octubre de 2002
Cambiar password grupo de trabajo.	Oscar	7 de octubre de 2002	0	-
vaciar el listbox	Rosa	22 de marzo de 2002	1	7 de octubre de 2002
Foro sobre CMS	Triki	3 de octubre de 2002	0	-
Programar para Palm!!!!	Taufpate	28 de diciembre de 2001	2	3 de octubre de 2002
Redimension de programa	Txino	2 de octubre de 2002	0	-
MSAccess 97 a MCAccess XP	Oscar	2 de octubre de 2002	0	-
PowerBuilder 7.0	Sergio Hernandez	22 de enero de 2002	3	21 de septiembre de 2002
HOLA mi nombre es Ana	HOLA mi nombre es Ana	8 de marzo de 2002	1	19 de septiembre de 2002
Un foro de C#	fLIPIS	17 de septiembre de 2002	0	-

busco descompilador urgente	david	21 de marzo de 2002	1	17 de septiembre de 2002
Foro de Perl	alex	30 de agosto de 2002	0	-
Sobre c#	pepato	30 de agosto de 2002	0	-
Pascal, el lenguaje de los universitarios	VictorSanchez2	30 de noviembre de 2001	7	26 de agosto de 2002
Problema con MsAccess	Fran	23 de agosto de 2002	0	-
¿Para cuándo un foro de Python?	herraiz	13 de agosto de 2002	0	-
Softonic.com con J2ME !!!	Xan	8 de agosto de 2002	0	-

« **1** 2 3 4 5 6 »

Principio Página

Warning: Supplied argument is not a valid MySQL result resource in **/chs/p1/programacion.com/home/html/registroinit.php** on line **11**

Registrarse en Programación en castellano

Si quieres participar en foros, recibir las últimas novedades de programación en castellano mediante correo electrónico o acceder a la documentación en formato PDF o ZIP, deberas registrarte. Para ello, rellena el siguiente formulario teniendo en cuenta que los campos señalado por (*) son obligatorios.

Nombre: (*)

Apellidos: (*)

E-mail: (*)

País: ESPAÑA (*)

Usuario: (*)

Contraseña: (*)

Repetir contraseña: (*)

Domicilio:

Número: Piso / planta / escalera / puerta

Código postal:

Población:

Provincia: (*) Usuarios de España

Provincia:

Sexo: ○ Hombre ○ Mujer

Enviar

Canal de Formación - Ciberaula

Acuerdo de colaboración entre Ciberaula y Programacion

Recuerda que debes ser usuario registrado de Programación en castellano para aprovechar esta oferta.

Condiciones de la oferta

(1) Descuento del 101 Euros (16.805 ptas.) en la matrícula de cualquiera de los siguientes cursos:

- PHP - MySQL - Comercio electrónico sobre Apache o IIS con PHP - MySQL
- ASP - SQL - Comercio electrónico sobre IIS con ASP y Access
- XML
- Webmaster Nivel I
- Photoshop 6.0

Este descuento es válido para todas las modalidades de curso "Predefinido" y "Personalizado", quedando fuera del mismo la modalidad "por libre".

(2) Libre acceso al campus virtual de Ciberaula y a posibles actualizaciones o ampliaciones del material didáctico durante 3 meses extra una vez finalizado el curso. Este acceso post-curso permite el uso del material allí alojado, sin asistencia de tutor.

Requisitos para utilizar esta oferta

(1) Demostrar que eres suscriptor de programacion.net, enviando el mensaje donde anunciamos a nuestra lista de usuarios registrados la existencia de esta oferta a prognet@ciberaula.com indicando la dirección de correo desde la que te registraste en el cuerpo del mensaje.

(2) La matrícula en el curso que te interese deberás formalizarla (o al menos abonar una reserva de plaza) **antes del día 22 de febrero**, fecha en que la oferta deja de tener validez. No importa si el curso en el que te matriculas deseas comenzarlo en una fecha posterior, el único requisito es que el pago de la matrícula o reserva de plaza se efectúe **antes del día 22 de febrero**.

Más información

Tels: Desde España: 91 3035800 - 91 7781509
Desde fuera de España, añadir prefijo 34 antes de marcar el número
Fax: Desde España: 91 3803641

Desde fuera de España añadir prefijo 34 antes de marcar el número
Dirección: c/ Villalobos, 135 - 28038 Madrid (España)
E-mail: prognet@ciberaula.com
Para consultar disponibilidad de plazas y forma de matricularse envíanos un mensaje.

IMPORTANTE: Recuerda que debes estar registrado en Programación en Castellano.

Principio Página

Corporación serinter.com

Microsoft CERTIFIED Partner

MINISTERIO DE EDUCACIÓN, CULTURA Y DEPORTE

UNIVERSITY of CAMBRIDGE CENTRO EXAMINADOR OFICIAL

e-mail
36226 subscritos

Subscribirse

Boletín de noticias

Formación

FORMACION
- Ciclos formativos
- Desarrollo
- Diseño gráfico
- Ingeniería de sistemas
- Ofimática & Secretariado

MARCAS
- MICROSOFT
- MACROMEDIA
- ORACLE
- Sun Microsystems

TECNOLOGIAS
- .NET
- XML

VARIOS
- Quienes Somos
- Formación a Distancia
- Plano de situación
- Instalaciones
- Alquiler de aulas
- Conversor de monedas
- AFILIACIÓN/WEBMASTERS
- Contactar

Usuario Contraseña Entrar

Usuarios registrados

Especiales

Todos nuestros ciclos formativos están homologados por el Ministerio de Educación, Cultura y Deporte y por la Universidad de Cambridge. (click aquí).

Serinter informa

La gran mayoría de las profesiones que te ofrecemos están avaladas por certificaciones oficiales o empresas líderes. Entre los organismos que podrán avalar tu prestigio y reconocimiento personal se encuentra el Ministerio de educación y ciencia, Sun Microsystems, Microsoft y Oracle.

Masters especializados

MASTER EN PROGRAMACIÓN EN VISUAL STUDIO.NET (Disponible ON-LINE)

Tu mejor AVAL PROFESIONAL.

- Fundamentos de programación.
- SQL.
- XML.
- Plataforma .NET.
- Lenguaje C#.
- Acceso a Base de datos con ADO.NET.
- Desarrollo de aplicaciones Windows con .NET.
- Desarrollo de aplicaciones Web con ASP.NET.
- Web Services con .NET.

MASTER DE MODELIZACIÓN Y DESARROLLO DE APLICACIONES EN JAVA

Aprende el lenguaje de programación de desarrollo más demandado en la actualidad por las empresas de desarrollo de aplicaciones y de Internet.

Con este master el alumno aprenderá a analizar los requerimientos de una aplicación (de gestión o de Internet), modelarlos y desarrollarlos en lenguaje JAVA. Así mismo el alumno aprenderá los conocimientos necesarios de Bases de Datos para poder implementar dichas soluciones.

MASTER EXPERTO EN DISEÑO Y CREACIÓN DE SITIOS WEB (Disponible ON-LINE)

Domina las herramientas utilizadas por los profesionales del diseño de páginas web: Macromedia Fireworks MX, Dreamweaver MX, Flash MX,...

Aprenderás también los fundamentos de Internet necesarios para conectar tus páginas web con bases de datos.

MASTER DE ADMINISTRACIÓN Y DISEÑO DE SOLUCIONES EN ORACLE (Disponible ON-LINE)

ORACLE, Aprende a administrar, optimizar y resolver problemas en la base de datos Oracle. Define los niveles de seguridad, así como sus privilegios. Aprende todo lo necesario sobre migración de información.

Aprende los conceptos necesarios sobre Sql y Sql Plus. Diseña y desarrolla aplicaciones y bases de datos mediante Designer 2000 y DEVELOPER 2000.

MASTER PROGRAMACIÓN INTERNET (Presencial Madrid)

Comprenderás y sabrás abarcar las diferentes etapas del desarrollo de un proyecto en nuevas tecnologías. Habrá desarrollado ejemplos reales de aplicaciones en INTERNET, tales como portales y

aplicaciones B2B.
· Recopilar información sobre los procesos de negocio, sus retos y la visión.
· Determinar requerimientos de usuario.
· Modelizarás procesos de negocio existentes.
· Analizarás los requerimientos en el contexto del propio negocio.
· Crearás un plan director de desarrollo.
· Crearás un documento de especificaciones orgánicas.
· Desarrollarás la aplicación resultante en JAVA, JAVASCRIPT, ASP dependiendo de los requerimientos.
· Desarrollarás presentaciones en FLASH para tu sitio web.
· Dominar conceptos de Usabilidad y EXPERIENCIA de usuario en diseño de páginas WEB.

Cursos monográficos

CURSO DE FLASH MX Y FLASH EN 3D (Disponible ON-LINE)

Macromedia Flash MX está llegando, sus posibilidades son tan increibles que podría decirse que el equipo de Macromedia se ha superado una vez más.
Con este curso el alumno dominará la tecnología de Flash, así como algunas de las posibilidades que Flash MX nos ofrecerá en un futuro muy próximo.

CURSO DE OFIMÁTICA & SECRETARIADO

Curso de ofimática, es la herramienta de productividad indispensable para todo usuario de ordenador.
Desde un punto de vista empresarial se analizan y aprenden las diferentes aplicaciones que contiene esta suite: Microsoft Word, Microsoft Office, Microsoft Excel y Power Point.
Así mismo se aprenderán los conceptos necesarios para utilizar INTERNET como herramienta de trabajo.

MASTER PROGRAMACIÓN INTERNET (PRESENCIAL MADRID)

Comprenderás y sabrás abarcar las diferentes etapas del desarrollo de un proyecto en nuevas tecnologías. Habrá desarrollado ejemplos reales de aplicaciones en INTERNET, tales como **Portales y aplicaciones B2B**

- Recopilar información sobre los **procesos de negocio**, sus retos y la visión.
- Determinar **requerimientos de usuario**
- Modelizarás **procesos de negocio** existentes.
- Analizarás los requerimientos en el contexto del propio negocio.
- Crearás un **plan director de desarrollo**
- Crearás un documento de **especificaciones orgánicas**.
- **Desarrollarás la aplicación resultante** en JAVA, JAVASCRIPT, ASP dependiendo de los requerimientos.
- **Desarrollarás presentaciones en FLASH para tu sitio web.**
- **Dominar conceptos de Usabilidad y EXPERIENCIA de usuario en diseño de páginas WEB.**

Profesiones

- Web developer Manager
- Web Developer Specialist
- System Developer Specialist

Certificaciones

- Sun Certification Programmer for Java 2
- SERINTER Web Developer Specialist

Calendario

- 01-10-02 - 26-06-03

este sitio web ha sido desarrollado y promocionado por http://www.marketingenlaweb.com

Autor

Nombre:
José Antonio González Seco

Email:
josanguapo@hotmail.com

URL:
http://www.josanguapo.com

Visitas totales:
17876

Cursos

1. El lenguaje de programación C# (17876 visitas)

Principio Página

El lenguaje de programación C#

Comentarios de los usuarios

El eje central de la obra es el lenguaje de programación C#, del que no sólo se describe su sintaxis sino que también se intenta explicar cuáles son las razones que justifican las decisiones tomadas en su diseño y cuáles son los errores más difíciles de detectar que pueden producirse al desarrollar de aplicaciones con él.

Puede también escribir su comentario o regresar a donde estaba leyendo.

Comentarios (1/3)

36 comentarios

[Subir]

« **1** 2 3 »

esta muy interesante (19/10/2002)
Por jose antonio

como descargo el curso completo de c#, y si tienes curso de java visual 6 informame, pues te agradesco que me contestes pues estoy aprendiendo poco a poco..... Gracias

Felicitaciones (16/10/2002)
Por Enrique López

Me parece un excelente curso, y me gustaría continuar estudiandolo, pero se me dificulta en línea, así que por favor podría tener una copia como documento electrónico,

Muchísimas gracias

como bajar este manual (14/10/2002)
Por fabian rojas

me justaria tener este curso en mi pc
te agradeco si me dices como lo puedo hacer

¿Como descargarlo? (13/10/2002)
Por Araceli

te agradeceria si pudieras decirme como descargar el archivo para poder tenerlo en mi computadora sin tener conexion a internet

gracias

Excelente (08/10/2002)
Por PABLO SOUR

Ojala me pudieras mandar una copia por correo electronico Gracias

Excelente (06/10/2002)
Por Carlos Medina

Es un buen manual para aquellos principiantes como yo, te pediría que me dijeras como obtener el archivo del manual completo para bajarlo y tenerlo en la computadora

descarga de este tutorial (03/10/2002)
Por miguel pescador

estaria agradecido si me pudieras mandar un archivo con el tutorial a poder ser en pdf

descarga completa de manual (01/10/2002)
Por Julio

Me gustaria realizar una descarga completa del manual dime como puedo hacerlo. Te lo agradeceré,

Sin conexion a internet (26/09/2002)
Por Martin (MAC)

buenas... gente primero excelente trabajo!!
... una forma que puede usar para que la puedan ver sin estar conectado a intenet es agregar a favoritos. la direccion;http://www.programacion.com/tutorial.csharp.html, chequear la opcion disponible sin conexion, presionar el boton de personlaizar, y seleccionar decargar un nivel de vinculos. todo esto estan conectado a internet.., deconectarse y accesar la info entrando por su link de favoritos.. saludos...

Genial (24/09/2002)
Por Borja

Me parece una referencia utilísima y muy completa. Estoy en la misma situación que la mayoría de los que han firmado antes que yo, ¿como puedo conseguirla para consultarla fuera de internet? Gracias.

obtener tutorial (23/09/2002)
Por victor

hola: me gustaria obtener el tutorial que presentas si podrias decirme como te lo agradecere escribeme a mi correo
gracias

como lo obtengo (23/09/2002)
Por Erick

Informarme como lo puedo obtener

UNA DE ROMANOS (23/09/2002)
Por elena nito del bosque

Hola me gustaria saber si c3 es una copia barata de java, porque el guiilermo puertas desde que perdio el juicio con java no sabe lo que hacer, SERÁ IGUAL DE FULL QUE SU VISUAL BASIC? DARA PANTALLAZOS POR TODOS LOS LADOS? AHORA QUE PROBLEMA SIMILA AL DE LAS DLL APARECRÁ?...TODO ESTO LO VEREMOS EN LOS PROXIMOS CAPITULOS DE MOCOSOFT...

Bien hecho! (20/09/2002)
Por Luis

Excelente trabajo, pero ahora lo más importante: cómo puedo obtenerlo?

Manual de C# (19/09/2002)
Por Ramses

Grandioso, muy util en este momento que es la nueva moda el frame .NET
Me gustaria tenerlo para poder leerlo mas detenidamente cuando no estoy online...
Hay alguna forma de bajarlo?
si alguien puede enviarlo a mi correo:
RamsesReinoso@hotmail.com

Gracias
Buen Trabajo

Principio Página

Nuestras zonas

Escribir comentario

El lenguaje de programación C#

El eje central de la obra es el lenguaje de programación C#, del que no sólo se describe su sintaxis sino que también se intenta explicar cuáles son las razones que justifican las decisiones tomadas en su diseño y cuáles son los errores más difíciles de detectar que pueden producirse al desarrollar de aplicaciones con él.

Rellene el siguiente formulario para realizar un comentario sobre "El lenguaje de programación C#":

Nombre:

Dirección de correo electrónico:

Título del comentario:

Texto del comentario:

Enviar

Principio Página

El lenguaje de programación C#

Comentarios de los usuarios

El eje central de la obra es el lenguaje de programación C#, del que no sólo se describe su sintaxis sino que también se intenta explicar cuáles son las razones que justifican las decisiones tomadas en su diseño y cuáles son los errores más difíciles de detectar que pueden producirse al desarrollar de aplicaciones con él.

Puede también escribir su comentario o regresar a donde estaba leyendo.

Comentarios (1/3)

36 comentarios

[Subir]

« 1 **2** 3 »

¡Como obtener el tutorial! (14/09/2002)
Por Xavier Ortega

Me gustarìa saber como puedo bajarme el tutorial ya que està muy interesante pero muchas veces no tengo la oportunidad de utilizar el Internet.

Manual muy interesante (11/09/2002)
Por FREDY ALCOCER SANCHEZ

Un manual basico para los que se inician en la programacion C#, no nocesitas saber programar en C++.
Saludos

Como??? (10/09/2002)
Por MeMe

La verdad es que me gusta mucho lo que es programar en C, ya que alo que he oido es muy bueno, pero mi pregunta es, como se llama el software para programar en C? o si existen varios cual es el mejor...

Disculpen mi ignoracia pero creo que asi todos empesamos, si alguien sabe el nombre me lo podrian mandar ami email por favor??

manuellegarcoa@hotmail.com

Gracias!!

necesito ayuda (10/09/2002)
Por denice

HOLA QUE TAL ESPERO PUEDAN AYUDARME RESULTA QUE NECESITO SABER LOS SIGNIFICADOS DE DIAGRAMAS LINEALES SUS TIPOS DE OPERADORES OPERADOSRES ARITMETICOS OPERADORES RELACIONAL4ES LOGICOS CONSPETOS EXPRESIONES CONTADORES ACUMULADORES SU DEFINICIONE Y CLASIFICACIO DE DIAGRAMAS SIMBOLOGIA ESTRUCTURA DE CONTROL SECUENCIALES SELECTIVOS IFIFELSE ESTRUCTURAS REPETITIVAS EXPRESION FOR WHILE DO WHILE ELABORACION DE DIAGRAMAS. SEUDO CODIGO PRUEBAS DE ESCRITORIO DISSEÑO MOLECULAR SUS FUNCIONES SIN PARAMETROS CON PARAMETROS.
SI NO ES POSIBLE ENVIAR LA INFORMACION ME PODRIAN DAR UNA PAGINA DE DONDE BAJARLA
ESPERO NO SEA MUCHA MOLESTIA LES AGRADEZCO
GRACIAS

Que Nota! (10/09/2002)
Por jpcc

Excelente el manual...
gracias por ese aporte!

Simplemente magnífico (s/t) (10/09/2002)
Por Juan

Libro en .doc (10/09/2002)
Por José Antonio González Seco

Hola amigos,

Ante todo agradecer los comentarios favorables sobre mi obra que he leido. Después, deciros que lo que cierto correo de por aquí pedía sobre pasar el libro a .doc no es necesario. En mi página web www.josanguapo.com lo podreis encontrar tanto en .doc como .pdf

Saludos
José Antonio.

Paranoia (10/09/2002)
Por paugas

La verdad es que todo esto me hace sospechar ¿para que ejecutar un programar remotamente? es un intento para que los datos esten controlados por microsoft, creo que esto fracasara.

quien será el primero (09/09/2002)
Por Fentinak

a ver quién es el primero que lo pasa a un .doc y los comparte con los demás...fentinak@hotmail.com

MUY BUENO (09/09/2002)
Por Esteban Marín Cervantes

ESTE TUTORIAL ESTA BUENISIMO, LA INTERESANTE APRENDER DE .NET. OKK..

.NET wa! (09/09/2002)
Por Roberto Rodriguez G.

Me veo en la obligacion de Leer este Tutorial (de cual estoy muy agradecido), y como siempre es de los mejores de la Red,

pero creo que como siempre Microsoft Copia lo bueno y destruye al Creador. Aunque creo dificil botar a SUN, porque C# es una copia de JAVA, del cual soy un seguidor, pero como el mercado manda hay que aprender .NET

RobertoChi :D

Muy bueno (09/09/2002)
Por cifra

Es muy bueno, extenso, facil de manejar y muy bien estructurado

Dios te dè màs (07/09/2002)
Por A.C. Colombia

Te felicito, amigo. Gracias por darnos esta informaciòn. Recibiràs màs conforme has dado. Dios te bendiga. A.C.

c# (07/09/2002)
Por Daniel

Simplemente genial, por ser uno de los primeros manuales y por estar tan bien estructurado y explicado. Gracias

Aquí tenéis este magnífico tutorial para descargarlo (06/09/2002)
Por Eduardo

En el apartado downloads de mi página:

http://usuarios.tripod.es/CSharp/index.htm

Principio Página

El lenguaje de programación C#

Comentarios de los usuarios

El eje central de la obra es el lenguaje de programación C#, del que no sólo se describe su sintaxis sino que también se intenta explicar cuáles son las razones que justifican las decisiones tomadas en su diseño y cuáles son los errores más difíciles de detectar que pueden producirse al desarrollar de aplicaciones con él.

Puede también escribir su comentario o regresar a donde estaba leyendo.

Comentarios (1/3)

36 comentarios

[Subir]

« 1 2 **3** »

GRACIAS José Antonio González Seco (03/09/2002)
Por BENIGNO

Esta muy bien estructurado en cada tema te agradezco que pongas a nuestro alcance este tutorial no entiendo a la gente que se queja de que haber si lo tienen en tal o cual formato encima que les ponen en la palma de la mano un trabajo bueno encima se quejan, por otro lado estoy mirando por trabajos en VB.NET parece que lo tienen dejadito me extraña pues ya es de objetos verdaderamente. Termino dando de nuevo mi gratitud al autor y al que lo aloja claro!.

Este tutorial es... (30/08/2002)
Por Manu

Sencillamente fabuloso. Muy bien, y gracias al autor.

excelente manual (28/08/2002)
Por Ferney

el manuel sobre C sharp es excelente, pero me uno a los que dicen que deberia estar en PDF o Word, gracias al autor por esta estupenda guia y gracias a programacion.com

Felicitaciones (23/08/2002)
Por AC

Felicitaciones, amigo. Muy buena obra. Sería bueno bajarla toda. Estoy esperando las nuevas entregas.

MANUAL (21/08/2002)

Por sanchez

Donde puedo descargar el manual.

Muy bueno, aúnque.... (19/08/2002)
Por Marc

El tutorial es muy bueno, pero poco dinámico y cómodo.
Obtener la copia en Word o PDF e imprimirlo es algo mejor.

Aún así felicito al autor.

Principio Página

Contacto - Datos legales

Puntuar

Nuestras zonas

El lenguaje de programación C#

Elija una puntuación para "El lenguaje de programación C#":

Puntuación:

Elija una

Puntuar

Principio Página

Recomendar a un amigo

Nuestras zonas

El lenguaje de programación C#

Rellene el siguiente formulario para recomendar "El lenguaje de programación C#" a un amigo:

Dirección de correo del amigo: (*Obligatorio*)

Tu Nombre: (*Obligatorio*)

Tu dirección de correo electrónico: (*Obligatorio*)

Comentario:

Enviar

Principio Página

Estadísticas

Curso:
El lenguaje de
programación C#

Visitas totales:
17876
Visitas desglosadas por meses (desde Julio de 2002)

1. Octubre de 2002: 3698 visitas
2. Septiembre de 2002: 11879 visitas
3. Agosto de 2002: 10409 visitas
4. Julio de 2002: 1783 visitas

Principio Página

El lenguaje de programación C#

En esta página:

Introducción a la obra

Requisitos previos recomendados

En principio, para entender con facilidad esta obra es recomendable estar familiarizado con los conceptos básicos de programación orientada a objetos, en particular con los lenguajes de programación C++ o Java de los que C# deriva.

Sin embargo, estos no son requisitos fundamentales para entenderla ya que cada vez que en ella se introduce algún elemento del lenguaje se definen y explican los conceptos básicos que permiten entenderlo. Aún así, sigue siendo recomendable disponer de los requisitos antes mencionados para poder moverse con mayor soltura por el libro y aprovecharlo al máximo.

Estructura de la obra

Básicamente el eje central de la obra es el lenguaje de programación C#, del que no sólo se describe su sintaxis sino que también se intenta explicar cuáles son las razones que justifican las decisiones tomadas en su diseño y cuáles son los errores más difíciles de detectar que pueden producirse al desarrollar de aplicaciones con él. Sin embargo, los 20 temas utilizados para ello pueden descomponerse en tres grandes bloques:

- **Bloque 1: Introducción a C# y .NET**: Antes de empezar a describir el lenguaje

es obligatorio explicar el porqué de su existencia, y para ello es necesario antes introducir la plataforma .NET de Microsoft con la que está muy ligado. Ese es el objetivo de los temas 1 y 2, donde se explican las características y conceptos básicos de C# y .NET, las novedosas aportaciones de ambos y se introduce la programación y compilación de aplicaciones en C# con el típico ¡Hola Mundo!

- **Bloque 2: Descripción del lenguaje:** Este bloque constituye el grueso de la obra y está formado por los temas comprendidos entre el 3 y el 19. En ellos se describen pormenorizadamente los aspectos del lenguaje mostrando ejemplos de su uso, explicando su porqué y avisando de cuáles son los problemas más difíciles de detectar que pueden surgir al utilizarlos y cómo evitarlos.

- **Bloque 3: Descripción del compilador**: Este último bloque, formado solamente por el tema 20, describe cómo se utiliza el compilador de C# tanto desde la ventana de consola como desde la herramienta Visual Studio.NET. Como al describir el lenguaje, también se intenta dar una explicación lo más exhaustiva, útil y fácil de entender posible del significado, porqué y aplicabilidad de las opciones de compilación que ofrece.

Convenios de notación

Para ayudar a resaltar la información clave se utilizan diferentes convenciones respecto a los tipos de letra usados para representar cada tipo de contenido. Éstas son:

- El texto correspondiente a explicaciones se ha escrito usando la fuente Verdana, como es el caso de este párrafo.
- Los fragmentos de código fuente se han escrito usando la fuente de paso fijo tal y como se muestra a continuación:

```
class HolaMundo
{
  static void Main()
  {
    System.Console.WriteLine("¡Hola Mundo!");
  }
}
```

Esta misma fuente es la que se usará desde las explicaciones cada vez que se haga referencia a algún elemento del código fuente. Si además dicho elemento es una palabra reservada del lenguaje o viene predefinido en la librería de .NET, su nombre se escribirá en negrita para así resaltar el carácter especial del mismo

- Las referencias a textos de la interfaz del sistema operativo (nombres de ficheros y directorios, texto de la línea de comandos, etc.) se han escrito usando la misma fuente de paso fijo. Por ejemplo:

```
csc HolaMundo.cs
```

Cuando además este tipo de texto se utilice para hacer referencia a elementos predefinidos tales como extensiones de ficheros recomendadas o nombres de aplicaciones incluidas en el

SDK, se escribirá en negrita.

- Al describirse la sintaxis de definición de los elementos del lenguaje se usará fuente de paso fija y se representarán en cursiva los elementos opcionales en la misma, en negrita los que deban escribirse tal cual, y sin negrita y entre símbolos < y > los que representen de texto que deba colocarse en su lugar. Por ejemplo, cuando se dice que una clase ha de definirse así:

```
class  <nombreClase>
{
  <miembros>
}
```

Lo que se está diciendo es que ha de escribirse la palabra reservada class, seguida de texto que represente el nombre de la clase a definir, seguido de una llave de apertura ({), seguido opcionalmente de texto que se corresponda con definiciones de miembros y seguido de una llave de cierre (})

Principio Página

El lenguaje de programación C#

En esta página:

Tema 1: Introducción a Microsoft.NET

Microsoft.NET

Microsoft.NET es el conjunto de nuevas tecnologías en las que Microsoft ha estado trabajando durante los últimos años con el objetivo de obtener una plataforma sencilla y potente para distribuir el software en forma de servicios que puedan ser suministrados remotamente y que puedan comunicarse y combinarse unos con otros de manera totalmente independiente de la plataforma, lenguaje de programación y modelo de componentes con los que hayan sido desarrollados. Ésta es la llamada **plataforma .NET**, y a los servicios antes comentados se les denomina **servicios Web**.

Para crear aplicaciones para la plataforma .NET, tanto servicios Web como aplicaciones tradicionales (aplicaciones de consola, aplicaciones de ventanas, servicios de Windows NT, etc.), Microsoft ha publicado el denominado kit de desarrollo de software conocido como **.NET Framework SDK**, que incluye las herramientas necesarias tanto para su desarrollo como para su distribución y ejecución y **Visual Studio.NET**, que permite hacer todo la anterior desde una interfaz visual basada en ventanas. Ambas herramientas puede descargarse gratuitamente desde http://www.msdn.microsoft.com/net, aunque la última sólo está disponible para subscriptores

MSDN Universal (los no subscriptores pueden pedirlo desde dicha dirección y se les enviará gratis por correo ordinario)

El concepto de Microsoft.NET también incluye al conjunto de nuevas aplicaciones que Microsoft y terceros han (o están) desarrollando para ser utilizadas en la plataforma .NET. Entre ellas podemos destacar aplicaciones desarrolladas por Microsoft tales como Windows.NET, Hailstorm, Visual Studio.NET, MSN.NET, Office.NET, y los nuevos servidores para empresas de Microsoft (SQL Server.NET, Exchange.NET, etc.)

Common Language Runtime (CLR)

El **Common Language Runtime** (**CLR**) es el núcleo de la plataforma .NET. Es el motor encargado de gestionar la ejecución de las aplicaciones para ella desarrolladas y a las que ofrece numerosos servicios que simplifican su desarrollo y favorecen su fiabilidad y seguridad. Las principales características y servicios que ofrece el CLR son:

- **Modelo de programación consistente:** A todos los servicios y facilidades ofrecidos por el CLR se accede de la misma forma: a través de un modelo de programación orientado a objetos. Esto es una diferencia importante respecto al modo de acceso a los servicios ofrecidos por los algunos sistemas operativos actuales (por ejemplo, los de la familia Windows), en los que a algunos servicios se les accede a través de llamadas a funciones globales definidas en DLLs y a otros a través de objetos (objetos COM en el caso de la familia Windows)

- **Modelo de programación sencillo:** Con el CLR desaparecen muchos elementos complejos incluidos en los sistemas operativos actuales (registro de Windows, GUIDs, HRESULTS, IUnknown, etc.) El CLR no es que abstraiga al programador de estos conceptos, sino que son conceptos que no existen en la plataforma .NET

- **Eliminación del "infierno de las DLLs":** En la plataforma .NET desaparece el problema conocido como "infierno de las DLLs" que se da en los sistemas operativos actuales de la familia Windows, problema que consiste en que al sustituirse versiones viejas de DLLs compartidas por versiones nuevas puede que aplicaciones que fueron diseñadas para ser ejecutadas usando las viejas dejen de funcionar si las nuevas no son 100% compatibles con las anteriores. En la plataforma .NET las versiones nuevas de las DLLs pueden coexistir con las viejas, de modo que las aplicaciones diseñadas para ejecutarse usando las viejas podrán seguir usándolas tras instalación de las nuevas. Esto, obviamente, simplifica mucho la instalación y desinstalación de software.

- **Ejecución multiplataforma:** El CLR actúa como una máquina virtual, encargándose de ejecutar las aplicaciones diseñadas para la plataforma .NET. Es decir, cualquier plataforma para la que exista una versión del CLR podrá ejecutar cualquier aplicación .NET. Microsoft ha desarrollado versiones del CLR para la mayoría de las versiones de Windows: Windows 95, Windows 98, Windows ME, Windows NT 4.0, Windows 2000, Windows XP y Windows CE (que puede ser usado en CPUs que no sean de la familia x86) Por otro lado Microsoft ha firmado un acuerdo con Corel para portar el CLR a Linux y también hay terceros que están desarrollando de manera independiente versiones de libre distribución del CLR para Linux. Asímismo, dado que la arquitectura del CLR está totalmente abierta, es posible que en el futuro se diseñen versiones del mismo para otros sistemas

operativos.

- **Integración de lenguajes:** Desde cualquier lenguaje para el que exista un compilador que genere código para la plataforma .NET es posible utilizar código generado para la misma usando cualquier otro lenguaje tal y como si de código escrito usando el primero se tratase. Microsoft ha desarrollado un compilador de C# que genera código de este tipo, así como versiones de sus compiladores de Visual Basic (Visual Basic.NET) y C++ (C++ con extensiones gestionadas) que también lo generan y una versión del intérprete de JScript (JScript.NET) que puede interpretarlo. La integración de lenguajes esta que es posible escribir una clase en C# que herede de otra escrita en Visual Basic.NET que, a su vez, herede de otra escrita en C++ con extensiones gestionadas.

- **Gestión de memoria:** El CLR incluye un **recolector de basura** que evita que el programador tenga que tener en cuenta cuándo ha de destruir los objetos que dejen de serle útiles. Este recolector es una aplicación que se activa cuando se quiere crear algún objeto nuevo y se detecta que no queda memoria libre para hacerlo, caso en que el recolector recorre la memoria dinámica asociada a la aplicación, detecta qué objetos hay en ella que no puedan ser accedidos por el código de la aplicación, y los elimina para limpiar la memoria de "objetos basura" y permitir la creación de otros nuevos. Gracias a este recolector se evitan errores de programación muy comunes como intentos de borrado de objetos ya borrados, agotamiento de memoria por olvido de eliminación de objetos inútiles o solicitud de acceso a miembros de objetos ya destruidos.

- **Seguridad de tipos:** El CLR facilita la detección de errores de programación difíciles de localizar comprobando que toda conversión de tipos que se realice durante la ejecución de una aplicación .NET se haga de modo que los tipos origen y destino sean compatibles.

- **Aislamiento de procesos:** El CLR asegura que desde código perteneciente a un determinado proceso no se pueda acceder a código o datos pertenecientes a otro, lo que evita errores de programación muy frecuentes e impide que unos procesos puedan atacar a otros. Esto se consigue gracias al sistema de seguridad de tipos antes comentado, pues evita que se pueda convertir un objeto a un tipo de mayor tamaño que el suyo propio, ya que al tratarlo como un objeto de mayor tamaño podría accederse a espacios en memoria ajenos a él que podrían pertenecer a otro proceso. También se consigue gracias a que no se permite acceder a posiciones arbitrarias de memoria.

- **Tratamiento de excepciones:** En el CLR todo los errores que se puedan producir durante la ejecución de una aplicación se propagan de igual manera: mediante excepciones. Esto es muy diferente a como se venía haciendo en los sistemas Windows hasta la aparición de la plataforma .NET, donde ciertos errores se transmitían mediante códigos de error en formato Win32, otros mediante HRESULTs y otros mediante excepciones.

  El CLR permite que excepciones lanzadas desde código para .NET escrito en un cierto lenguaje se puedan capturar en código escrito usando otro lenguaje, e incluye mecanismos de depuración que pueden saltar desde código escrito para .NET en un determinado lenguaje

a código escrito en cualquier otro. Por ejemplo, se puede recorrer la pila de llamadas de una excepción aunque ésta incluya métodos definidos en otros módulos usando otros lenguajes.

- **Soporte multihilo:** El CLR es capaz de trabajar con aplicaciones divididas en múltiples hilos de ejecución que pueden ir evolucionando por separado en paralelo o intercalándose, según el número de procesadores de la máquina sobre la que se ejecuten. Las aplicaciones pueden lanzar nuevos hilos, destruirlos, suspenderlos por un tiempo o hasta que les llegue una notificación, enviarles notificaciones, sincronizarlos, etc.

- **Distribución transparente:** El CLR ofrece la infraestructura necesaria para crear objetos remotos y acceder a ellos de manera completamente transparente a su localización real, tal y como si se encontrasen en la máquina que los utiliza.

- **Seguridad avanzada:** El CLR proporciona mecanismos para restringir la ejecución de ciertos códigos o los permisos asignados a los mismos según su procedendecia o el usuario que los ejecute. Es decir, puede no darse el mismo nivel de confianza a código procedente de Internet que a código instalado localmente o procedente de una red local; puede no darse los mismos permisos a código procedente de un determinado fabricante que a código de otro; y puede no darse los mismos permisos a un mismo códigos según el usuario que lo esté ejecutando o según el rol que éste desempeñe. Esto permite asegurar al administrador de un sistema que el código que se esté ejecutando no pueda poner en peligro la integridad de sus archivos, la del registro de Windows, etc.

- **Interoperabilidad con código antiguo:** El CLR incorpora los mecanismos necesarios para poder acceder desde código escrito para la plataforma .NET a código escrito previamente a la aparición de la misma y, por tanto, no preparado para ser ejecutando dentro de ella. Estos mecanismos permiten tanto el acceso a objetos COM como el acceso a funciones sueltas de DLLs preexistentes (como la API Win32)

Como se puede deducir de las características comentadas, el CLR lo que hace es gestionar la ejecución de las aplicaciones diseñadas para la plataforma .NET. Por esta razón, al código de estas aplicaciones se le suele llamar **código gestionado**, y al código no escrito para ser ejecutado directamente en la plataforma .NET se le suele llamar **código no gestionado**.

Microsoft Intermediate Language (MSIL)

Todos los compiladores que generan código para la plataforma .NET no generan código máquina para CPUs x86 ni para ningún otro tipo de CPU concreta, sino que generan código escrito en el lenguaje intermedio conocido como Microsoft Intermediate Lenguage (MSIL) El CLR da a las aplicaciones las sensación de que se están ejecutando sobre una máquina virtual, y precisamente MSIL es el código máquina de esa máquina virtual. Es decir, MSIL es el único código que es capaz de interpretar el CLR, y por tanto cuando se dice que un compilador genera código para la plataforma .NET lo que se está diciendo es que genera MSIL.

MSIL ha sido creado por Microsoft tras consultar a numerosos especialistas en la escritura de compiladores y lenguajes tanto del mundo académico como empresarial. Es un lenguaje de un nivel de abstracción mucho más alto que el de la mayoría de los códigos máquina de las CPUs existentes, e incluye instrucciones que permiten trabajar directamente con objetos (crearlos, destruirlos,

inicializarlos, llamar a métodos virtuales, etc.), tablas y excepciones (lanzarlas, capturarlas y tratarlas)

Ya se comentó que el compilador de C# compila directamente el código fuente a MSIL, que Microsoft ha desarrollado nuevas versiones de sus lenguajes Visual Basic (Visual Basic.NET) y C++ (C++ con extensiones gestionadas) cuyos compiladores generan MSIL, y que ha desarrollado un intérprete de JScript (JScript.NET) que genera código MSIL. Pues bien, también hay numerosos terceros que han anunciado estar realizando versiones para la plataforma .NET de otros lenguajes como APL, CAML, Cobol, Eiffel, Fortran, Haskell, Java (J#), Mercury, ML, Mondrian, Oberon, Oz, Pascal, Perl, Python, RPG, Scheme y Smalltalk.

La principal ventaja del MSIL es que facilita la ejecución multiplataforma y la integración entre lenguajes al ser independiente de la CPU y proporcionar un formato común para el código máquina generado por todos los compiladores que generen código para .NET. Sin embargo, dado que las CPUs no pueden ejecutar directamente MSIL, antes de ejecutarlo habrá que convertirlo al código nativo de la CPU sobre la que se vaya a ejecutar. De esto se encarga un componente del CLR conocido como compilador JIT (Just-In-Time) o jitter que va convirtiendo dinámicamente el código MSIL a ejecutar en código nativo según sea necesario. Este jitter se distribuye en tres versiones:

- **jitter normal:** Es el que se suele usar por defecto, y sólo compila el código MSIL a código nativo a medida que va siendo necesario, pues así se ahorra tiempo y memoria al evitarse tener que compilar innecesariamente código que nunca se ejecute. Para conseguir esto, el cargador de clases del CLR sustituye inicialmente las llamadas a métodos de las nuevas clases que vaya cargando por llamadas a funciones auxiliares (stubs) que se encarguen de compilar el verdadero código del método. Una vez compilado, la llamada al stub es sustituida por una llamada directa al código ya compilado, con lo que posteriores llamadas al mismo no necesitarán compilación.

- **jitter económico:** Funciona de forma similar al jitter normal solo que no realiza ninguna optimización de código al compilar sino que traduce cada instrucción MSIL por su equivalente en el código máquina sobre la que se ejecute. Esta especialmente pensado para ser usado en dispositivos empotrados que dispongan de poca potencia de CPU y poca memoria, pues aunque genere código más ineficiente es menor el tiempo y memoria que necesita para compilar. Es más, para ahorrar memoria este jitter puede descargar código ya compilado que lleve cierto tiempo sin ejecutarse y sustituirlo de nuevo por el stub apropiado. Por estas razones, este es el jitter usado por defecto en Windows CE, sistema operativo que se suele incluir en los dispositivos empotrados antes mencionados.

  Otra utilidad del jitter económico es que facilita la adaptación de la plataforma .NET a nuevos sistemas porque es mucho más sencillo de implementar que el normal. De este modo, gracias a él es posible desarrollar rápidamente una versión del CLR que pueda ejecutar aplicaciones gestionadas aunque sea de una forma poco eficiente, y una vez desarrollada es posible centrarse en desarrollar el jitter normal para optimizar la ejecución de las mismas.

- **prejitter:** Se distribuye como una aplicación en línea de comandos llamada **ngen.exe** mediante la que es posible compilar completamente cualquier ejecutable o librería (cualquier ensamblado en general, aunque este concepto se verá más adelante) que contenga código gestionado y convertirlo a código nativo, de modo que posteriores ejecuciones del mismo se harán usando esta versión ya compilada y no se perderá tiempo en hacer la compilación dinámica.

La actuación de un jitter durante la ejecución de una aplicación gestionada puede dar la sensación de hacer que ésta se ejecute más lentamente debido a que ha de invertirse tiempo en las compilaciones dinámicas. Esto es cierto, pero hay que tener en cuenta que es una solución mucho más eficiente que la usada en otras plataformas como Java, ya que en .NET cada código no es interpretado cada vez que se ejecuta sino que sólo es compilado la primera vez que se llama al método al que pertenece. Es más, el hecho de que la compilación se realice dinámicamente permite que el jitter tenga acceso a mucha más información sobre la máquina en que se ejecutará la aplicación del que tendría cualquier compilador tradicional, con lo que puede optimizar el código para ella generado (por ejemplo, usando las instrucciones especiales del Pentium III si la máquina las admite, usando registros extra, incluyendo código *inline*, etc.) Además, como el recolector de basura de .NET mantiene siempre compactada la memoria dinámica las reservas de memoria se harán más rápido, sobre todo en aplicaciones que no agoten la memoria y, por tanto, no necesiten de una recolección de basura. Por estas razones, los ingenieros de Microsoft piensan que futuras versiones de sus jitters podrán incluso conseguir que el código gestionado se ejecute más rápido que el no gestionado.

Metadatos

En la plataforma .NET se distinguen dos tipos de **módulos** de código compilado: **ejecutables** (extensión **.exe**) y **librerías de enlace dinámico** (extensión **.dll** generalmente) Ambos son ficheros que contienen definiciones de tipos de datos, y la diferencia entre ellos es que sólo los primeros disponen de un método especial que sirve de punto de entrada a partir del que es posible ejecutar el código que contienen haciendo una llamada desde la línea de comandos del sistema operativo. A ambos tipos de módulos se les suele llamar **ejecutables portables** (PE), ya que su código puede ejecutarse en cualquiera de los diferentes sistemas operativos de la familia Windows para los que existe alguna versión del CLR.

El contenido de un módulo no sólo MSIL, sino que también consta de otras dos áreas muy importantes: la cabecera de CLR y los metadatos:

La **cabecera de CLR** es un pequeño bloque de información que indica que se trata de un módulo gestionado e indica es la versión del CLR que necesita, cuál es su firma digital, cuál es su punto de entrada (si es un ejecutable), etc.

Los **metadatos** son un conjunto de datos organizados en forma de tablas que almacenan información sobre los tipos definidos en el módulo, los miembros de éstos y sobre cuáles son los tipos externos al módulo a los que se les referencia en el módulo. Los metadatos de cada modulo los genera automáticamente el compilador al crearlo, y entre sus tablas se incluyen:

Tabla	Descripción
ModuleDef	Define las características del módulo. Consta de un único elemento que almacena un identificador de versión de módulo (GUID creado por el compilador) y el nombre de fichero que se dio al módulo al compilarlo (así este nombre siempre estará disponible, aunque se renombre el fichero)
TypeDef	Define las características de los tipos definidos en el módulo. De cada tipo se almacena su nombre, su tipo padre, sus modificadores de acceso y referencias a los elementos de las tablas de miembros correspondientes a sus miembros.
MethodDef	Define las características de los métodos definidos en el módulo. De cada método se guarda su nombre, signatura (por cada parámetro se incluye una referencia al elemento apropiado en la tabla ParamDef), modificadores y posición del módulo donde comienza el código MSIL de su cuerpo.

ParamDef	Define las características de los parámetros definidos en el módulo. De cada parámetro se guarda su nombre y modificadores.
FieldDef	Define las características de los campos definidos en el módulo. De cada uno se almacena información sobre cuál es su nombre, tipo y modificadores.
PropertyDef	Define las características de las propiedades definidas en el módulo. De cada una se indica su nombre, tipo, modificadores y referencias a los elementos de la tabla MethodDef correspondientes a sus métodos set/get.
EventDef	Define las características de los eventos definidos en el módulo. De cada uno se indica su nombre, tipo, modificadores. y referencias a los elementos de la tabla MethodDef correspondientes a sus métodos add/remove.
AssemblyRef	Indica cuáles son los ensamblados externos a los que se referencia en el módulo. De cada uno se indica cuál es su nombre de fichero (sin extensión), versión, idioma y marca de clave pública.
ModuleRef	Indica cuáles son los otros módulos del mismo ensamblado a los que referencia el módulo. De cada uno se indica cuál es su nombre de fichero.
TypeRef	Indica cuáles son los tipos externos a los que se referencia en el módulo. De cada uno se indica cuál es su nombre y, según donde estén definidos, una referencia a la posición adecuada en la tabla AssemblyRef o en la tabla ModuleRef.
MemberRef	Indican cuáles son los miembros definidos externamente a los que se referencia en el módulo. Estos miembros pueden ser campos, métodos, propiedades o eventos; y de cada uno de ellos se almacena información sobre su nombre y signatura, así como una referencia a la posición de la tabla TypeRef donde se almacena información relativa al tipo del que es miembro.

Tabla 1: Principales tablas de metadatos

Nótese que el significado de los metadatos es similar al de otras tecnologías previas a la plataforma .NET como lo son los ficheros IDL. Sin embargo, los metadatos tienen dos ventajas importantes sobre éstas: contiene más información y siempre se almacenan incrustados en el módulo al que describen, haciendo imposible la separación entre ambos. Además, como se verá más adelante, es posible tanto consultar los metadatos de cualquier módulo a través de las clases del espacio de nombres `System.Reflection` de la BCL como añadirles información adicional mediante **atributos** (se verá más adelante)

Ensamblados

Un **ensamblado** es una agrupación lógica de uno o más módulos o ficheros de recursos (ficheros .GIF, .HTML, etc.) que se engloban bajo un nombre común. Un programa puede acceder a información o código almacenados en un ensamblado sin tener porqué sabe cuál es el fichero en concreto donde se encuentran, por lo que los ensamblados nos permiten abstraernos de la ubicación física del código que ejecutemos o de los recursos que usemos. Por ejemplo, podemos incluir todos los tipos de una aplicación en un mismo ensamblado pero colocando los más frecuentemente usados en un cierto módulo y los menos usados en otro, de modo que sólo se descarguen de Internet los últimos si es que se van a usar.

Todo ensamblado contiene un **manifiesto**, que son metadatos con información sobre las características del ensamblado. Este manifiesto puede almacenarse cualquiera de los módulos que formen el ensamblado o en uno específicamente creado para ello, caso éste último necesario cuando es un **ensamblado satélite** (sólo contiene recursos)

Las principales tablas incluidas en los manifiestos son las siguientes:

Tabla	Descripción
AssemblyDef	Define las características del ensamblado. Consta de un único elemento que almacena el nombre del ensamblado sin extensión, versión, idioma, clave pública y tipo de algoritmo de dispersión usado para hallar los valores de dispersión de la tabla FileDef.
FileDef	Define cuáles son los archivos que forman el ensamblado. De cada uno se da su nombre y valor de dispersión. Nótese que sólo el módulo que contiene el manifiesto sabrá qué ficheros que forman el ensamblado, pero el resto de ficheros del mismo no sabrán si pertenecen o no a un ensamblado (no contienen metadatos que les indique si pertenecen a un ensamblado)
ManifestResourceDef	Define las características de los recursos incluidos en el módulo. De cada uno se indica su nombre y modificadores de acceso. Si es un recurso incrustado se indica dónde empieza dentro del PE que lo contiene, y si es un fichero independiente se indica cuál es el elemento de la tabla FileDef correspondiente a dicho fichero.
ExportedTypesDef	Indica cuáles son los tipos definidos en el ensamblado y accesibles desde fuera del mismo. Para ahorrar espacio sólo recogen los que no pertenezcan al módulo donde se incluye el manifiesto, y de cada uno se indica su nombre, la posición en la tabla FileDef del fichero donde se ha implementado y la posición en la tabla TypeDef correspondiente a su definición.
AssemblyProccesorDef	Indica en qué procesadores se puede ejecutar el ensamblado, lo que puede ser útil saberlo si el ensamblado contiene módulos con código nativo (podría hacerse usando C++ con extensiones gestionadas) Suele estar vacía, lo que indica que se puede ejecutar en cualquier procesador; pero si estuviese llena, cada elemento indicaría un tipo de procesador admitido según el formato de identificadores de procesador del fichero WinNT.h incluido con Visual Studio.NET (por ejemplo, 586 = Pentium, 2200 = Arquitectura IA64, etc.)
AssemblyOSDef	Indica bajo qué sistemas operativos se puede ejecutar el ensamblado, lo que puede ser útil si contiene módulos con tipos o métodos disponibles sólo en ciertos sistemas. Suele estar vacía, lo que indica que se puede ejecutar en cualquier procesador; pero si estuviese llena, indicaría el identificador de cada uno de los sistemas admitidos siguiendo el formato del WinNT.h de Visual Studio.NET (por ejemplo, 0 = familia Windows 9X, 1 = familia Windows NT, etc.) y el número de la versión del mismo a partir de la que se admite.

Tabla 2: Principales tablas de un manifiesto

Para asegurar que no se haya alterado la información de ningún ensamblado se usa el criptosistema de clave pública RSA. Lo que se hace es calcular el código de dispersión SHA-1 del módulo que contenga el manifiesto e incluir tanto este valor cifrado con RSA (**firma digital**) como la clave pública necesaria para descifrarlo en algún lugar del módulo que se indicará en la cabecera de CLR. Cada vez que se vaya a cargar en memoria el ensamblado se calculará su valor de dispersión de nuevo y se comprobará que es igual al resultado de descifrar el original usando su clave pública. Si no fuese así se detectaría que se ha adulterado su contenido.

Para asegurar también que los contenidos del resto de ficheros que formen un ensamblado no hayan sido alterados lo que se hace es calcular el código de dispersión de éstos antes de cifrar el

ensamblado y guardarlo en el elemento correspondiente a cada fichero en la tabla FileDef del manifiesto. El algoritmo de cifrado usado por defecto es SHA-1, aunque en este caso también se da la posibilidad de usar MD5. En ambos casos, cada vez que se accede al fichero para acceder a un tipo o recurso se calculará de nuevo su valor de dispersión y se comprobará que coincida con el almacenado en FileDef.

Dado que las claves públicas son valores que ocupan muchos bytes (2048 bits), lo que se hace para evitar que los metadatos sean excesivamente grandes es no incluir en las referencias a ensamblados externos de la tabla AssemblyRef las claves públicas de dichos ensamblados, sino sólo los 64 últimos bits resultantes de aplicar un algoritmo de dispersión a dichas claves. A este valor recortado se le llama **marca de clave pública**.

Hay dos tipos de ensamblados: **ensamblados privados** y **ensamblados compartidos**. Los privados se almacenan en el mismo directorio que la aplicación que los usa y sólo puede usarlos ésta, mientras que los compartidos se almacenan en un **caché de ensamblado global** (GAC) y pueden usarlos cualquiera que haya sido compilada referenciándolos.

Los compartidos han de cifrase con RSA ya que lo que los identifica es en el GAC es su nombre (sin extensión) más su clave pública, lo que permite que en el GAC puedan instalarse varios ensamblados con el mismo nombre y diferentes claves públicas. Es decir, es como si la clave pública formase parte del nombre del ensamblado, razón por la que a los ensamblados así cifrados se les llama **ensamblados de nombre fuerte**. Esta política permite resolver los conflictos derivados de que se intente instalar en un mismo equipo varios ensamblados compartidos con el mismo nombre pero procedentes de distintas empresas, pues éstas tendrán distintas claves públicas.

También para evitar problemas, en el GAC se pueden mantener múltiples versiones de un mismo ensamblado. Así, si una aplicación fue compilada usando una cierta versión de un determinado ensamblado compartido, cuando se ejecute sólo podrá hacer uso de esa versión del ensamblado y no de alguna otra más moderna que se hubiese instalado en el GAC. De esta forma se soluciona el problema del **infierno de las DLL** comentado al principio del tema.

En realidad es posible modificar tanto las políticas de búsqueda de ensamblados (por ejemplo, para buscar ensamblados privados fuera del directorio de la aplicación) como la política de aceptación de ensamblados compartidos (por ejemplo, para que se haga automáticamente uso de las nuevas versiones que se instalen de DLLs compartidas) incluyendo en el directorio de instalación de la aplicación un fichero de configuración en formato XML con las nuevas reglas para las mismas. Este fichero ha de llamarse igual que el ejecutable de la aplicación pero ha de tener extensión **.cfg**.

Librería de clase base (BCL)

La Librería de Clase Base (BCL) es una librería incluida en el *.NET Framework* formada por cientos de tipos de datos que permiten acceder a los servicios ofrecidos por el CLR y a las funcionalidades más frecuentemente usadas a la hora de escribir programas. Además, a partir de estas clases prefabricadas el programador puede crear nuevas clases que mediante herencia extiendan su funcionalidad y se integren a la perfección con el resto de clases de la BCL. Por ejemplo, implementando ciertos interfaces podemos crear nuevos tipos de colecciones que serán tratadas exactamente igual que cualquiera de las colecciones incluidas en la BCL.

Esta librería está escrita en MSIL, por lo que puede usarse desde cualquier lenguaje cuyo compilador genere MSIL. A través de las clases suministradas en ella es posible desarrollar cualquier tipo de aplicación, desde las tradicionales aplicaciones de ventanas, consola o servicio de Windows NT hasta los novedosos servicios Web y páginas ASP.NET. Es tal la riqueza de servicios que ofrece que puede crearse lenguajes que carezcan de librería de clases propia y sólo usen la BCL -como C#.

Dado la amplitud de la BCL, ha sido necesario organizar las clases en ella incluida en **espacios de nombres** que agrupen clases con funcionalidades similares. Por ejemplo, los espacios de nombres más usados son:

Espacio de nombres	Utilidad de los tipos de datos que contiene
System	Tipos muy frecuentemente usados, como los los tipos básicos, tablas, excepciones, fechas, números aleatorios, recolector de basura, entrada/salida en consola, etc.
System.Collections	Colecciones de datos de uso común como pilas, colas, listas, diccionarios, etc.
System.Data	Manipulación de bases de datos. Forman la denominada arquitectura **ADO.NET**.
System.IO	Manipulación de ficheros y otros flujos de datos.
System.Net	Realización de comunicaciones en red.
System.Reflection	Acceso a los metadatos que acompañan a los módulos de código.
System.Runtime.Remoting	Acceso a objetos remotos.
System.Security	Acceso a la política de seguridad en que se basa el CLR.
System.Threading	Manipulación de hilos.
System.Web.UI.WebControls	Creación de interfaces de usuario basadas en ventanas para aplicaciones Web.
System.Winforms	Creación de interfaces de usuario basadas en ventanas para aplicaciones estándar.
System.XML	Acceso a datos en formato XML.

Tabla 3: Espacios de nombres de la BCL más usados

Common Type System (CTS)

El **Common Type System** (CTS) o Sistema de Tipo Común es el conjunto de reglas que han de seguir las definiciones de tipos de datos para que el CLR las acepte. Es decir, aunque cada lenguaje gestionado disponga de sus propia sintaxis para definir tipos de datos, en el MSIL resultante de la compilación de sus códigos fuente se ha de cumplir las reglas del CTS. Algunos ejemplos de estas reglas son:

- Cada tipo de dato puede constar de cero o más miembros. Cada uno de estos miembros puede ser un campo, un método una propiedad o un evento.
- No puede haber herencia múltiple, y todo tipo de dato ha de heredar directa o indirectamente de `System.Object`.
- Los modificadores de acceso admitidos son:

Modificador	Código desde el que es accesible el miembro
public	Cualquier código
private	Código del mismo tipo de dato

family	Código del mismo tipo de dato o de hijos de éste.
assembly	Código del mismo ensamblado
family and assembly	Código del mismo tipo o de hijos de éste ubicado en el mismo ensamblado
family or assembly	Código del mismo tipo o de hijos de éste, o código ubicado en el mismo ensamblado

Tabla 4: Modificadores de acceso a miembros admitidos por el CTS

Common Language Specification (CLS)

El **Common Language Specification** (CLS) o Especificación del Lenguaje Común es un conjunto de reglas que han de seguir las definiciones de tipos que se hagan usando un determinado lenguaje gestionado si se desea que sean accesibles desde cualquier otro lenguaje gestionado. Obviamente, sólo es necesario seguir estas reglas en las definiciones de tipos y miembros que sean accesibles externamente, y no la en las de los privados. Además, si no importa la interoperabilidad entre lenguajes tampoco es necesario seguirlas. A continuación se listan algunas de reglas significativas del CLS:

- Los tipos de datos básicos admitidos son `bool`, `char`, `byte`, `short`, `int`, `long`, `float`, `double`, `string` y `object` Nótese pues que no todos los lenguajes tienen porqué admitir los tipos básicos enteros sin signo o el tipo `decimal` como lo hace C#.

- Las tablas han de tener una o más dimensiones, y el número de dimensiones de cada tabla ha de ser fijo. Además, han de indexarse empezando a contar desde 0.

- Se pueden definir tipos abstractos y tipos sellados. Los tipos sellados no pueden tener miembros abstractos.

- Las excepciones han de derivar de `System.Exception`, los delegados de `System.Delegate`, las enumeraciones de `System.Enum`, y los tipos por valor que no sean enumeraciones de `System.ValueType`.

- Los métodos de acceso a propiedades en que se traduzcan las definiciones get/set de éstas han de llamarse de la forma `get_X` y `set_X` respectivamente, donde X es el nombre de la propiedad; los de acceso a indizadores han de traducirse en métodos `get_Item` y `setItem`; y en el caso de los eventos, sus definiciones add/remove han de traducirse en métodos de `add_X` y `remove_X`.

- En las definiciones de atributos sólo pueden usarse enumeraciones o datos de los siguientes tipos: `System.Type`, `string`, `char`, `bool`, `byte`, `short`, `int`, `long`, `float`, `double` y `object`.

- En un mismo ámbito no se pueden definir varios identificadores cuyos nombres sólo difieran en la capitalización usada. De este modo se evitan problemas al acceder a ellos usando lenguajes no sensibles a mayúsculas.

- Las enumeraciones no pueden implementar interfaces, y todos sus campos han de ser estáticos y del mismo tipo. El tipo de los campos de una enumeración sólo puede ser uno de estos cuatro tipos básicos: `byte`, `short`, `int` o `long`.

Principio Página

El lenguaje de programación C#

En esta página:

Tema 2: Introducción a C#

Origen y necesidad de un nuevo lenguaje

C# (leído en inglés "C Sharp" y en español "C Almohadilla") es el nuevo lenguaje de propósito general diseñado por Microsoft para su plataforma .NET. Sus principales creadores son Scott Wiltamuth y Anders Hejlsberg, éste último también conocido por haber sido el diseñador del lenguaje Turbo Pascal y la herramienta RAD Delphi.

Aunque es posible escribir código para la plataforma .NET en muchos otros lenguajes, C# es el único que ha sido diseñado específicamente para ser utilizado en ella, por lo que programarla usando C# es mucho más sencillo e intuitivo que hacerlo con cualquiera de los otros lenguajes ya que C# carece de elementos heredados innecesarios en .NET. Por esta razón, se suele decir que C# es el **lenguaje nativo de .NET**

La sintaxis y estructuración de C# es muy similar a la C++, ya que la intención de Microsoft con C# es facilitar la migración de códigos escritos en estos lenguajes a C# y facilitar su aprendizaje a los desarrolladores habituados a ellos. Sin embargo, su sencillez y el alto nivel de productividad son equiparables a los de Visual Basic.

Un lenguaje que hubiese sido ideal utilizar para estos menesteres es Java, pero debido a problemas con la empresa creadora del mismo -Sun-, Microsoft ha tenido que desarrollar un nuevo lenguaje que añadiese a las ya probadas virtudes de Java las modificaciones que Microsoft tenía pensado

añadirle para mejorarlo aún más y hacerlo un lenguaje orientado al desarrollo de componentes.

En resumen, C# es un lenguaje de programación que toma las mejores características de lenguajes preexistentes como Visual Basic, Java o C++ y las combina en uno solo. El hecho de ser relativamente reciente no implica que sea inmaduro, pues Microsoft ha escrito la mayor parte de la BCL usándolo, por lo que su compilador es el más depurado y optimizado de los incluidos en el *.NET Framework SDK*

Características de C#

Con la idea de que los programadores más experimentados puedan obtener una visión general del lenguaje, a continuación se recoge de manera resumida las principales características de C# Alguna de las características aquí señaladas no son exactamente propias del lenguaje sino de la plataforma .NET en general. Sin embargo, también se comentan aquí también en tanto que tienen repercusión directa en el lenguaje, aunque se indicará explícitamente cuáles son este tipo de características cada vez que se toquen:

- **Sencillez:** C# elimina muchos elementos que otros lenguajes incluyen y que son innecesarios en .NET. Por ejemplo:
  - El código escrito en C# es **autocontenido**, lo que significa que no necesita de ficheros adicionales al propio fuente tales como ficheros de cabecera o ficheros IDL
  - El tamaño de los tipos de datos básicos es fijo e independiente del compilador, sistema operativo o máquina para quienes se compile (no como en C++), lo que facilita la portabilidad del código.
  - No se incluyen elementos poco útiles de lenguajes como C++ tales como macros, herencia múltiple o la necesidad de un operador diferente del punto (`.`) acceder a miembros de espacios de nombres (`::`)

- **Modernidad:** C# incorpora en el propio lenguaje elementos que a lo largo de los años ha ido demostrándose son muy útiles para el desarrollo de aplicaciones y que en otros lenguajes como Java o C++ hay que simular, como un tipo básico `decimal` que permita realizar operaciones de alta precisión con reales de 128 bits (muy útil en el mundo financiero), la inclusión de una instrucción `foreach` que permita recorrer colecciones con facilidad y es ampliable a tipos definidos por el usuario, la inclusión de un tipo básico `string` para representar cadenas o la distinción de un tipo `bool` específico para representar valores lógicos.

- **Orientación a objetos:** Como todo lenguaje de programación de propósito general actual, C# es un lenguaje orientado a objetos, aunque eso es más bien una característica del CTS que de C#. Una diferencia de este enfoque orientado a objetos respecto al de otros lenguajes como C++ es que el de C# es más puro en tanto que no admiten ni funciones ni variables globales sino que todo el código y datos han de definirse dentro de definiciones de tipos de datos, lo que reduce problemas por conflictos de nombres y facilita la legibilidad del código.

  C# soporta todas las características propias del paradigma de programación orientada a objetos: **encapsulación**, **herencia** y **polimorfismo**.

  En lo referente a la encapsulación es importante señalar que aparte de los típicos modificadores `public`, `private` y `protected`, C# añade un cuarto modificador llamado

`internal`, que puede combinarse con `protected` e indica que al elemento a cuya definición precede sólo puede accederse desde su mismo ensamblado.

Respecto a la herencia -a diferencia de C++ y al igual que Java- C# sólo admite herencia simple de clases ya que la múltiple provoca más quebraderos de cabeza que facilidades y en la mayoría de los casos su utilidad puede ser simulada con facilidad mediante herencia múltiple de interfaces. De todos modos, esto vuelve a ser más bien una característica propia del CTS que de C#.

Por otro lado y a diferencia de Java, en C# se ha optado por hacer que todos los métodos sean por defecto sellados y que los redefinibles hayan de marcarse con el modificador `virtual` (como en C++), lo que permite evitar errores derivados de redefiniciones accidentales. Además, un efecto secundario de esto es que las llamadas a los métodos serán más eficientes por defecto al no tenerse que buscar en la tabla de funciones virtuales la implementación de los mismos a la que se ha de llamar. Otro efecto secundario es que permite que las llamadas a los métodos virtuales se puedan hacer más eficientemente al contribuir a que el tamaño de dicha tabla se reduzca.

- **Orientación a componentes:** La propia sintaxis de C# incluye elementos propios del diseño de componentes que otros lenguajes tienen que simular mediante construcciones más o menos complejas. Es decir, la sintaxis de C# permite definir cómodamente **propiedades** (similares a campos de acceso controlado), **eventos** (asociación controlada de funciones de respuesta a notificaciones) o **atributos** (información sobre un tipo o sus miembros)

- **Gestión automática de memoria:** Como ya se comentó, todo lenguaje de .NET tiene a su disposición el recolector de basura del CLR. Esto tiene el efecto en el lenguaje de que no es necesario incluir instrucciones de destrucción de objetos. Sin embargo, dado que la destrucción de los objetos a través del recolector de basura es indeterminista y sólo se realiza cuando éste se active -ya sea por falta de memoria, finalización de la aplicación o solicitud explícita en el fuente-, C# también proporciona un mecanismo de liberación de recursos determinista a través de la instrucción `using`.

- **Seguridad de tipos:** C# incluye mecanismos que permiten asegurar que los accesos a tipos de datos siempre se realicen correctamente, lo que permite evita que se produzcan errores difíciles de detectar por acceso a memoria no perteneciente a ningún objeto y es especialmente necesario en un entorno gestionado por un recolector de basura. Para ello se toman medidas del tipo:
  - Sólo se admiten **conversiones entre tipos compatibles**. Esto es, entre un tipo y antecesores suyos, entre tipos para los que explícitamente se haya definido un operador de conversión, y entre un tipo y un tipo hijo suyo del que un objeto del primero almacenase una referencia del segundo (**downcasting**) Obviamente, lo último sólo puede comprobarlo en tiempo de ejecución el CLR y no el compilador, por lo que en realidad el CLR y el compilador colaboran para asegurar la corrección de las conversiones.
  - No se pueden usar **variables no inicializadas**. El compilador da a los campos un valor por defecto consistente en ponerlos a cero y controla mediante análisis del flujo de control del fuente que no se lea ninguna variable local sin que se le haya asignado previamente algún valor.
  - Se comprueba que todo **acceso a los elementos de una tabla** se realice

con índices que se encuentren dentro del rango de la misma.
    - Se puede controlar la **producción de desbordamientos** en operaciones aritméticas, informándose de ello con una excepción cuando ocurra. Sin embargo, para conseguirse un mayor rendimiento en la aritmética estas comprobaciones no se hacen por defecto al operar con variables sino sólo con constantes (se pueden detectar en tiempo de compilación)
    - A diferencia de Java, C# incluye **delegados**, que son similares a los punteros a funciones de C++ pero siguen un enfoque orientado a objetos, pueden almacenar referencias a varios métodos simultáneamente, y se comprueba que los métodos a los que apunten tengan parámetros y valor de retorno del tipo indicado al definirlos.
    - Pueden definirse métodos que admitan un número indefinido de parámetros de un cierto tipo, y a diferencia lenguajes como C/C++, en C# siempre se comprueba que los valores que se les pasen en cada llamada sean de los tipos apropiados.

- **Instrucciones seguras:** Para evitar errores muy comunes, en C# se han impuesto una serie de restricciones en el uso de las instrucciones de control más comunes. Por ejemplo, la guarda de toda condición ha de ser una expresión condicional y no aritmética, con lo que se evitan errores por confusión del operador de igualdad (`==`) con el de asignación (`=`); y todo caso de un `switch` ha de terminar en un `break` o `goto` que indique cuál es la siguiente acción a realizar, lo que evita la ejecución accidental de casos y facilita su reordenación.

- **Sistema de tipos unificado:** A diferencia de C++, en C# todos los tipos de datos que se definan siempre derivarán, aunque sea de manera implícita, de una clase base común llamada **System.Object**, por lo que dispondrán de todos los miembros definidos en ésta clase (es decir, serán "objetos")

  A diferencia de Java, en C# esto también es aplicable a los tipos de datos básicos Además, para conseguir que ello no tenga una repercusión negativa en su nivel de rendimiento, se ha incluido un mecanismo transparente de **boxing** y **unboxing** con el que se consigue que sólo sean tratados como objetos cuando la situación lo requiera, y mientras tanto puede aplicárseles optimizaciones específicas.

  El hecho de que todos los tipos del lenguaje deriven de una clase común facilita enormemente el diseño de colecciones genéricas que puedan almacenar objetos de cualquier tipo.

- **Extensibilidad de tipos básicos:** C# permite definir, a través de **estructuras**, tipos de datos para los que se apliquen las mismas optimizaciones que para los tipos de datos básicos. Es decir, que se puedan almacenar directamente en pila (luego su creación, destrucción y acceso serán más rápidos) y se asignen por valor y no por referencia. Para conseguir que lo último no tenga efectos negativos al pasar estructuras como parámetros de métodos, se da la posibilidad de pasar referencias a pila a través del modificador de parámetro `ref`.

- **Extensibilidad de operadores:** Para facilitar la legibilidad del código y conseguir que los nuevos tipos de datos básicos que se definan a través de las estructuras estén al mismo nivel que los básicos predefinidos en el lenguaje, al igual que C++

y a diferencia de Java, C# permite redefinir el significado de la mayoría de los operadores -incluidos los de conversión, tanto para conversiones implícitas como explícitas- cuando se apliquen a diferentes tipos de objetos.

Las redefiniciones de operadores se hacen de manera inteligente, de modo que a partir de una única definición de los operadores `++` y `--` el compilador puede deducir automáticamente como ejecutarlos de manera prefijas y postfija; y definiendo operadores simples (como `+`), el compilador deduce cómo aplicar su versión de asignación compuesta (`+=`) Además, para asegurar la consistencia, el compilador vigila que los operadores con opuesto siempre se redefinan por parejas (por ejemplo, si se redefine `==`, también hay que redefinir `!=`)

También se da la posibilidad, a través del concepto de **indizador**, de redefinir el significado del operador `[]` para los tipos de dato definidos por el usuario, con lo que se consigue que se pueda acceder al mismo como si fuese una tabla. Esto es muy útil para trabajar con tipos que actúen como colecciones de objetos.

Extensibilidad de modificadores: C# ofrece, a través del concepto de **atributos**, la posibilidad de añadir a los metadatos del módulo resultante de la compilación de cualquier fuente información adicional a la generada por el compilador que luego podrá ser consultada en tiempo ejecución a través de la librería de reflexión de .NET . Esto, que más bien es una característica propia de la plataforma .NET y no de C#, puede usarse como un mecanismo para definir nuevos modificadores.

- **Versionable:** C# incluye una **política de versionado** que permite crear nuevas versiones de tipos sin temor a que la introducción de nuevos miembros provoquen errores difíciles de detectar en tipos hijos previamente desarrollados y ya extendidos con miembros de igual nombre a los recién introducidos.

  Si una clase introduce un nuevo método cuyas redefiniciones deban seguir la regla de llamar a la versión de su padre en algún punto de su código, difícilmente seguirían esta regla miembros de su misma signatura definidos en clases hijas previamente a la definición del mismo en la clase padre; o si introduce un nuevo campo con el mismo nombre que algún método de una clase hija, la clase hija dejará de funcionar. Para evitar que esto ocurra, en C# se toman dos medidas:

  - Se obliga a que toda redefinición deba incluir el modificador `override`, con lo que la versión de la clase hija nunca sería considerada como una redefinición de la versión de miembro en la clase padre ya que no incluiría `override`. Para evitar que por accidente un programador incluya este modificador, sólo se permite incluirlo en miembros que tengan la misma signatura que miembros marcados como redefinibles mediante el modificador `virtual`. Así además se evita el error tan frecuente en Java de creerse haber redefinido un miembro, pues si el miembro con `override` no existe en la clase padre se producirá un error de compilación.
  - Si no se considera redefinición, entonces se considera que lo que se desea es ocultar el método de la clase padre, de modo que para la clase hija sea como si nunca hubiese existido. El compilador avisará de esta decisión a través de un mensaje de aviso que puede suprimirse incluyendo el modificador `new` en la definición del miembro en la clase hija para así indicarle explícitamente la intención de ocultación.

- **Eficiente:** En principio, en C# todo el código incluye numerosas restricciones para asegurar su seguridad y no permite el uso de punteros. Sin embargo, y a diferencia de Java, en C# es posible saltarse dichas restricciones manipulando objetos a través de punteros. Para ello basta marcar regiones de código como inseguras (modificador `unsafe`) y podrán usarse en ellas punteros de forma similar a cómo se hace en C++, lo que puede resultar vital para situaciones donde se necesite una eficiencia y velocidad procesamiento muy grandes.

- **Compatible:** Para facilitar la migración de programadores, C# no sólo mantiene una sintaxis muy similar a C, C++ o Java que permite incluir directamente en código escrito en C# fragmentos de código escrito en estos lenguajes, sino que el CLR también ofrece, a través de los llamados **Platform Invocation Services** (**PInvoke**), la posibilidad de acceder a código nativo escrito como funciones sueltas no orientadas a objetos tales como las DLLs de la API Win32. Nótese que la capacidad de usar punteros en código inseguro permite que se pueda acceder con facilidad a este tipo de funciones, ya que éstas muchas veces esperan recibir o devuelven punteros.

  También es posible acceder desde código escrito en C# a objetos COM. Para facilitar esto, el *.NET Framework SDK* incluye una herramientas llamadas `tlbimp` y `regasm` mediante las que es posible generar automáticamente clases proxy que permitan, respectivamente, usar objetos COM desde .NET como si de objetos .NET se tratase y registrar objetos .NET para su uso desde COM.

  Finalmente, también se da la posibilidad de usar controles ActiveX desde código .NET y viceversa. Para lo primero se utiliza la utilidad **aximp**, mientras que para lo segundo se usa la ya mencionada **regasm**.

Escritura de aplicaciones

Aplicación básica ¡Hola Mundo!

Básicamente una aplicación en C# puede verse como un conjunto de uno o más ficheros de código fuente con las instrucciones necesarias para que la aplicación funcione como se desea y que son pasados al compilador para que genere un ejecutable. Cada uno de estos ficheros no es más que un fichero de texto plano escrito usando caracteres Unicode y siguiendo la sintaxis propia de C#.

Como primer contacto con el lenguaje, nada mejor que el típico programa de iniciación "¡Hola Mundo!" que lo único que hace al ejecutarse es mostrar por pantalla el mensaje ¡Hola Mundo! Su código es:

```
1:          class HolaMundo
2:          {
3:             static void Main()
4:             {
5:               System.Console.WriteLine("¡Hola Mundo!");
6:             }
7:          }
```

Todo el código escrito en C# se ha de escribir dentro de una definición de clase, y lo que en la línea **1:** se dice es que se va a definir una clase (**class**) de nombre HolaMundo1 cuya definición estará

comprendida entre la llave de apertura de la línea **2:** y su correspondiente llave de cierre en la línea línea **7:**

Dentro de la definición de la clase (línea **3:**) se define un método de nombre Main cuyo código es el indicado entre la llave de apertura de la línea **4:** y su respectiva llave de cierre (línea **6:**) Un método no es más que un conjunto de instrucciones a las que se les asocia un nombre, de modo que para posteriormente ejecutarlas baste referenciarlas por su nombre en vez de tener que reescribirlas.

La partícula que antecede al nombre del método indica cuál es el tipo de valor que se devuelve tras la ejecución del método, y en este caso es **`void`** que significa que no se devuelve nada. Por su parte, los paréntesis que se colocado tras el nombre del método indican cuáles son los parámetros éste toma, y como en este caso están vacíos ello significa que el método no toma parámetros. Los parámetros de un método permiten variar el resultado de su ejecución según los valores que se les dé en cada llamada.

La palabra **`static`** que antecede a la declaración del tipo de valor devuelto es un **modificador** del significado de la declaración de método que indica que el método está asociado a la clase dentro de la que se define y no a los objetos que se creen a partir de ella. Main() es lo que es denomina el **punto de entrada** de la aplicación, que no es más que el método por el que comenzará su ejecución. Necesita del modificador **`static`** para evitar que para llamarlo haya que crear algún objeto de la clase donde se haya definido.

Finalmente, la línea **5:** contiene la instrucción con el código a ejecutar, que lo que se hace es solicitar la ejecución del método **`WriteLine()`** de la clase **`Console`** definida en el espacio de nombres **`System`** pasándole como parámetro la cadena de texto con el contenido ¡Hola Mundo! Nótese que las cadenas de textos son secuencias de caracteres delimitadas por comillas dobles aunque dichas comillas no forman parte de la cadena. Por su parte, un espacio de nombres puede considerarse que es algo similar para las clases a lo que un directorio es para los ficheros; es decir, es una forma de agruparlas.

El método **`WriteLine()`** se usará muy a menudo en los próximos temas, por lo que es conveniente señalar ahora que una forma de llamarlo que se utilizará en repetidas ocasiones consiste en pasarle un número indefinido de otros parámetros de cualquier tipo e incluir en el primero subcadenas de la forma `{i}`. Con ello se consigue que se muestre por la ventana de consola la cadena que se le pasa como primer parámetro pero sustituyéndole las subcadenas `{i}` por el valor convertido en cadena de texto del parámetro que ocupe la posición i+2 en la llamada a **`WriteLine()`**. Por ejemplo, la siguiente instrucción mostraría Tengo 5 años por pantalla si x valiese 5:

```
System.Console.WriteLine("Tengo {0} años", x);
```

Para indicar cómo convertir cada objeto en un cadena de texto basta redefinir su método **`ToString()`**, aunque esto es algo que no se verá hasta el *Tema 5: Clases*.

Antes de seguir es importante resaltar que C# es sensible a las mayúsculas, los que significa que no da igual la capitalización con la que se escriban los identificadores. Es decir, no es lo mismo escribir Console que COnsole o CONSOLE, y si se hace de alguna de las dos últimas formas el compilador producirá un error debido a que en el espacio de nombres **`System`** no existe ninguna clase con dichos nombres. En este sentido, cabe señalar que un error común entre programadores acostumbrados a Java es llamar al punto de entrada main en vez de Main, lo que provoca un error al compilar ejecutables en tanto que el compilador no detectará ninguna definición de punto de entrada.

Puntos de entrada

Ya se ha dicho que el **punto de entrada** de una aplicación es un método de nombre Main que contendrá el código por donde se ha de iniciar la ejecución de la misma. Hasta ahora sólo se ha visto una versión de `Main()` que no toma parámetros y tiene como tipo de retorno **`void`**, pero en realidad todas sus posibles versiones son:

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

Como se ve, hay versiones de `Main()` que devuelven un valor de tipo **int**. Un **int** no es más que un tipo de datos capaz de almacenar valor enteros comprendidos entre -2.147$_1$483.648 y 2.147$_1$483.647, y el número devuelto por `Main()` sería interpretado como código de retorno de la aplicación. Éste valor suele usarse para indicar si la aplicación a terminado con éxito (generalmente valor 0) o no (valor según la causa de la terminación anormal), y en el *Tema 8: Métodos* se explicará como devolver valores.

También hay versiones de `Main()` que toman un parámetro donde se almacenará la lista de argumentos con los que se llamó a la aplicación, por lo que sólo es útil usar estas versiones del punto de entrada si la aplicación va a utilizar dichos argumentos para algo. El tipo de este parámetro es **`string[]`**, lo que significa que es una tabla de cadenas de texto (en el *Tema 5: Campos* se explicará detenidamente qué son las tablas y las cadenas), y su nombre -que es el que habrá de usarse dentro del código de `Main()` para hacerle referencia- es `args` en el ejemplo, aunque podría dársele cualquier otro

Compilación en línea de comandos

Una vez escrito el código anterior con algún editor de textos -como el **Bloc de Notas** de Windows- y almacenado en formato de texto plano en un fichero `HolaMundo.cs`, para compilarlo basta abrir una ventana de consola (MS-DOS en Windows), colocarse en el directorio donde se encuentre y pasárselo como parámetro al compilador así:

```
csc HolaMundo.cs
```

`csc.exe` es el compilador de C# incluido en el .NET Framework SDK para Windows de Microsoft, y es posible llamarlo desde cualquier directorio en tanto que al instalarlo se añade una referencia al mismo en el **path**. Si utiliza otros compiladores de C# puede que varie la forma en que se realice la compilación, por lo que lo que aquí se explica en principio sólo podría ser válido para el compilador de Microsoft para Windows.

Tras la compilación se obtendría un ejecutable llamado HolaMundo.exe cuya ejecución produciría la siguiente salida por la ventana de consola:

```
¡Hola Mundo!
```

Si la aplicación que se vaya a compilar no utilizase la ventana de consola para mostrar su salida sino una interfaz gráfica de ventanas, entonces habría que compilarla pasando al compilador la opción **/t** con el valor **winexe** antes del nombre del fichero a compilar. Si no se hiciese así se abriría la ventana de consola cada vez que ejecutase la aplicación de ventanas, lo que suele ser

indeseable en este tipo de aplicaciones. Así, para compilar Ventanas.cs como ejecutable de ventanas sería conveniente escribir:

```
csc /t:winexe Ventanas.cs
```

Nótese que aunque el nombre **winexe** dé la sensación de que este valor para la opción **/t** sólo permite generar ejecutables de ventanas, en realidad lo que permite es generar ejecutables sin ventana de consola asociada. Por tanto, también puede usarse para generar ejecutables que no tengan ninguna interfaz asociada, ni de consola ni gráfica.

Si en lugar de un ejecutable -ya sea de consola o de ventanas- se desea obtener una librería, entonces al compilar hay que pasar al compilador la opción **/t** con el valor **library**. Por ejemplo, siguiendo con el ejemplo inicial habría que escribir:

```
csc /t:library HolaMundo.cs
```

En este caso se generaría un fichero HolaMundo.dll cuyos tipos de datos podrían utilizarse desde otros fuentes pasando al compilador una referencia a los mismos mediante la opción **/r**. Por ejemplo, para compilar como ejecutable un fuente A.cs que use la clase HolaMundo de la librería HolaMundo.dll se escribiría:

```
csc /r:HolaMundo.dll A.cs
```

En general **/r** permite referenciar a tipos definidos en cualquier ensamblado, por lo que el valor que se le indique también puede ser el nombre de un ejecutable. Además, en cada compilación es posible referenciar múltiples ensamblados ya sea incluyendo la opción **/r** una vez por cada uno o incluyendo múltiples referencias en una única opción **/r** usando comas o puntos y comas como separadores. Por ejemplo, las siguientes tres llamadas al compilador son equivalentes:

```
csc /r:HolaMundo.dll;Otro.dll;OtroMás.exe  A.cs

csc /r:HolaMundo.dll,Otro.dll,OtroMás.exe A.cs

csc /t:HolaMundo.dll /r:Otro.dll /r:OtroMás.exe A.cs
```

Hay que señalar que aunque no se indique nada, en toda compilación siempre se referencia por defecto a la librería **mscorlib.dll** de la BCL, que incluye los tipos de uso más frecuente. Si se usan tipos de la BCL no incluidos en ella habrá que incluir al compilar referencias a las librerías donde estén definidos (en la documentación del SDK sobre cada tipo de la BCL puede encontrar información sobre donde se definió)

Tanto las librerías como los ejecutables son ensamblados. Para generar un módulo de código que no forme parte de ningún ensamblado sino que contenga definiciones de tipos que puedan añadirse a ensamblados que se compilen posteriormente, el valor que ha de darse al compilar a la opción **/t** es **module**. Por ejemplo:

```
csc /t:module HolaMundo.cs
```

Con la instrucción anterior se generaría un módulo llamado HolaMundo.netmodule que podría ser añadido a compilaciones de ensamblados incluyéndolo como valor de la opción **/addmodule**. Por ejemplo, para añadir el módulo anterior a la compilación del fuente librería Lib.cs como librería se escribiría:

```
csc /t:library /addmodule:HolaMundo.netmodule Lib.cs
```

Aunque hasta ahora todas las compilaciones de ejemplo se han realizado utilizando un único fichero de código fuente, en realidad nada impide que se puedan utilizar más. Por ejemplo, para compilar los ficheros A.cs y B.cs en una librería A.dll se ejecutaría:

```
csc /t:library A.cs B.cs
```

Nótese que el nombre que por defecto se dé al ejecutable generado siempre es igual al del primer fuente especificado pero con la extensión propia del tipo de compilación realizada (**.exe** para ejecutables, **.dll** para librerías y **.netmodule** para módulos) Sin embargo, puede especificárse como valor en la opción **/out** del compilador cualquier otro tal y como muestra el siguiente ejemplo que compila el fichero A.cs como una librería de nombre Lib.exe:

```
csc /t:library /out:Lib.exe A.cs
```

Véase que aunque se haya dado un nombre terminado en **.exe** al fichero resultante, éste sigue siendo una librería y no un ejecutable e intentar ejecutarlo produciría un mensaje de error. Obviamente no tiene mucho sentido darle esa extensión, y sólo se le ha dado en este ejemplo para demostrar que, aunque recomendable, la extensión del fichero no tiene porqué corresponderse realmente con el tipo de fichero del que se trate.

A la hora de especificar ficheros a compilar también es pueden utilizar los caracteres de comodín típicos del sistema operativo. Por ejemplo, para compilar todos los ficheros con extensión .cs del directorio actual en una librería llamada Varios.dll se haría:

```
csc /t:library /out:varios.dll *.cs
```

Con lo que hay que tener cuidado, y en especial al compilar varios fuentes, es con que no se compilen a la vez más de un tipo de dato con punto de entrada, pues entonces el compilador no sabría cuál usar como inicio de la aplicación. Para orientarlo, puede especificarse como valor de la opción **/main** el nombre del tipo que contenga el Main() ha usar como punto de entrada. Así, para compilar los ficheros A.cs y B.cs en un ejecutable cuyo punto de entrada sea el definido en el tipo Principal, habría que escribir:

```
csc /main:Principal A.cs B.cs
```

Obviamente, para que esto funcione A.cs o B.cs tiene que contener alguna definición de algún tipo llamado Principal con un único método válido como punto de entrada. (obviamente si contiene varias se volvería a tener el problema de no saber cuál usar)

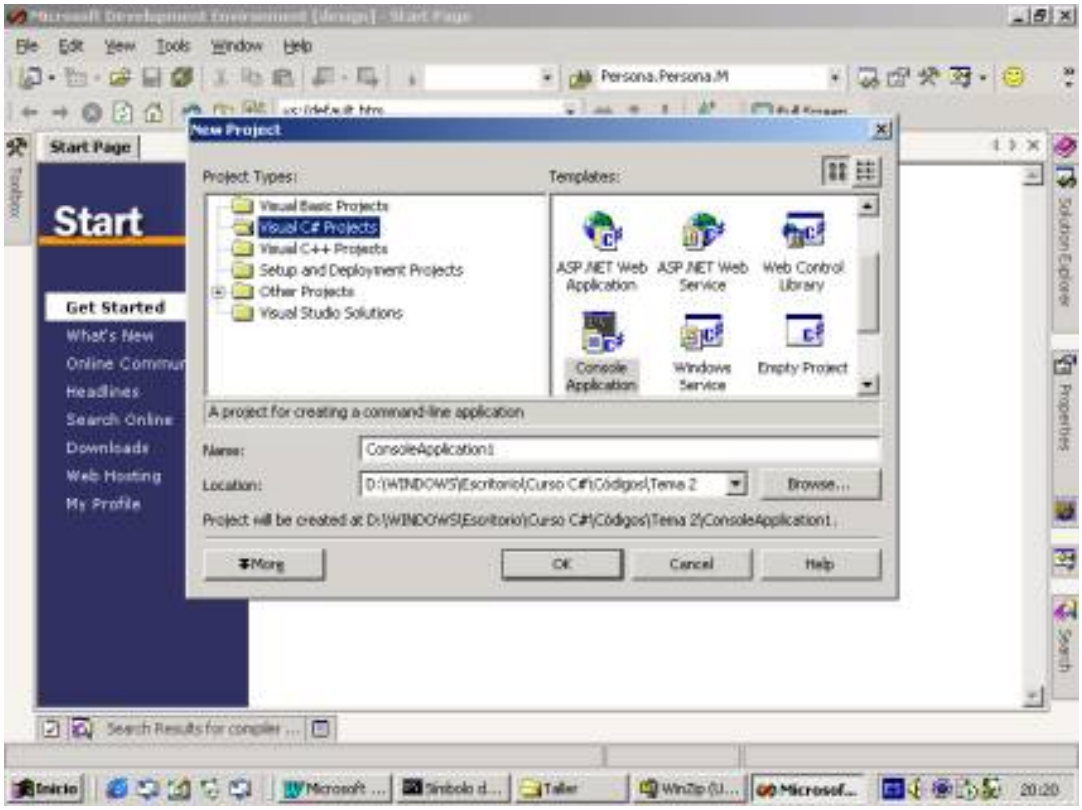
Compilación con Visual Studio.NET

Para compilar una aplicación en Visual Studio.NET primero hay que incluirla dentro de algún proyecto. Para ello basta pulsar el botón **New Project** en la página de inicio que se muestra nada más arrancar dicha herramienta, tras lo que se obtendrá una pantalla con el aspecto mostrado en la **Ilustración 1**.

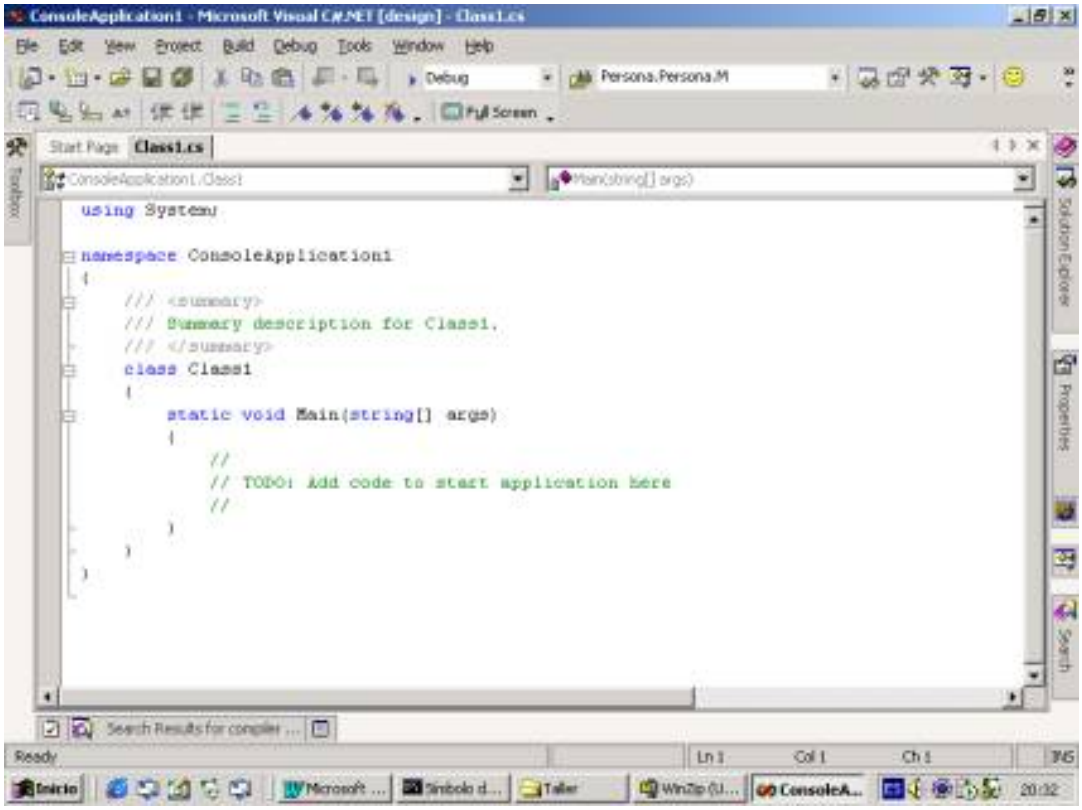
En el recuadro de la ventana mostrada etiquetado como **Project Types** se ha de seleccionar el tipo de proyecto a crear. Obviamente, si se va a trabajar en C# la opción que habrá que escoger en la misma será siempre Visual C# Projects.

En el recuadro **Templates** se ha de seleccionar la plantilla correspondiente al subtipo de proyecto dentro del tipo indicado en **Project Types** que se va a realizar. Para realizar un ejecutable de consola, como es nuestro caso, hay que seleccionar el icono etiquetado como Console Application. Si se quisiese realizar una librería habría que seleccionar Class Library, y si se quisies realizar un ejecutable de ventanas habría que seleccionar Windows Application. Nótese que no se ofrece ninguna plantilla para realizar módulos, lo que se debe a que desde Visual Studio.NET no pueden crearse.

Por último, en el recuadro de texto **Name** se ha de escribir el nombre a dar al proyecto y en **Location** el del directorio base asociado al mismo. Nótese que bajo de **Location** aparecerá un mensaje informando sobre cual será el directorio donde finalmente se almacenarán los archivos del proyecto, que será el resultante de concatenar la ruta especificada para el directorio base y el nombre del proyecto.



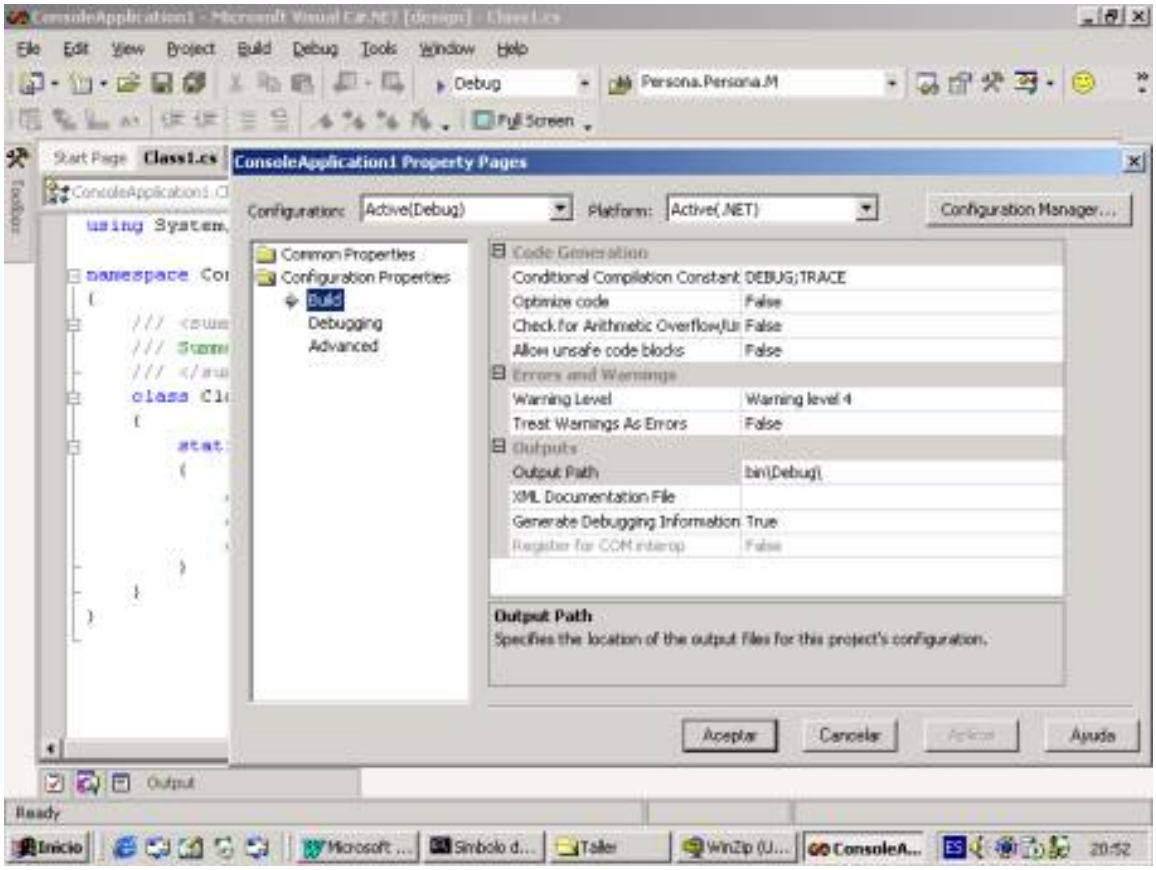
Una vez configuradas todas estas opciones, al pulsar botón **OK** Visual Studio creará toda la infraestructura adecuada para empezar a trabajar cómodamente en el proyecto. Como puede apreciarse en la **Ilustración 2**, esta infraestructura consistirá en la generación de un fuente que servirá de plantilla para la realización de proyectos del tipo elegido (en nuestro caso, aplicaciones de consola en C#):



A partir de esta plantilla, escribir el código de la aplicación de ejemplo es tan sencillo con simplemente teclear `System.Console.WriteLine("¡Hola Mundo!")` dentro de la definición del método `Main()` creada por Visual Studio.NET. Claro está, otra posibilidad es borrar toda la plantilla y sustituirla por el código para HolaMundo mostrado anteriormente.

Se haga como se haga, para compilar y ejecutar tras ello la aplicación sólo hay que pulsar **CTRL+F5** o seleccionar **Debug -> Start Without Debugging** en el menú principal de Visual Studio.NET. Para sólo compilar el proyecto, entonces hay que seleccionar **Build -> Rebuild All**. De todas formas, en ambos casos el ejecutable generado se almacenará en el subdirectorio Bin\Debug del directorio del proyecto.

En el extremo derecho de la ventana principal de Visual Studio.NET puede encontrar el denominado **Solution Explorer** (si no lo encuentra, seleccione **View -> Solution Explorer**), que es una herramienta que permite consultar cuáles son los archivos que forman el proyecto. Si selecciona en él el icono correspondiente al proyecto en que estamos trabajando y pulsa **View -> Property Pages** obtendrá una hoja de propiedades del proyecto con el aspecto mostrado en la **Ilustración 3**:



Esta ventana permite configurar de manera visual la mayoría de opciones con las que se llamará al compilador en línea de comandos. Por ejemplo, para cambiar el nombre del fichero de salida (opción **/out**) se indica su nuevo nombre en el cuadro de texto **Common Properties -> General -> Assembly Name**; para cambiar el tipo de proyecto a generar (opción **/t**) se utiliza **Common Properties -> General -> Output Type** (como verá si intenta cambiarlo, no es posible generar módulos desde Visual Studio.NET); y el tipo que contiene el punto de entrada a utilizar (opción **/main**) se indica en **Common Properties -> General -> Startup Object**

Finalmente, para añadir al proyecto referencias a ensamblados externos (opción **/r**) basta seleccionar **Project -> Add Reference** en el menú principal de VS.NET.

Principio Página

En esta página:

Tema 3: El preprocesador

Concepto de preprocesador

El **preprocesado** es un paso previo a la compilación mediante el que es posible controlar la forma en que se realizará ésta. El **preprocesador** es el módulo auxiliar que utiliza el compilador para realizar estas tareas, y lo que finalmente el compilador compila es el resultado de aplicar el preprocesador al fichero de texto fuente, resultado que también es un fichero de texto. Nótese pues, que mientras que el compilador hace una traducción de texto a binario, lo que el preprocesador hace es una traducción de texto a texto.

Aquellos que tengan experiencia en el uso del preprocesador en lenguajes como C++ y conozcan los problemas que implica el uso del mismo pueden respirar tranquilos, ya que en C# se han eliminado la mayoría de características de éste que provocaban errores difíciles de detectar (macros, directivas de inclusión, etc.) y prácticamente sólo se usa para permitir realizar compilaciones condicionales de código.

Directivas de preprocesado

Concepto de directiva. Sintaxis

El preprocesador no interpreta de ninguna manera el código fuente del fichero, sino que sólo interpreta de dicho fichero lo que se denominan **directivas de preprocesado**. Estas directivas son líneas de texto del fichero fuente que se caracterizan porque en ellas el primer carácter no blanco que aparece es una almohadilla (carácter **#**) Por ejemplo:

```
#define TEST
#error Ha habido un error fatal
```

No se preocupe ahora si no entiendo el significado de estas directivas, ya que se explicarán más adelante. Lo único debe saber es que el nombre que se indica tras el símbolo # es el nombre de la directiva, y el texto que se incluye tras él (no todas las directivas tienen porqué incluirlo) es el valor que se le da. Por tanto, la sintaxis de una directiva es:

```
#<nombreDirectiva> <valorDirectiva>
```

Es posible incluir comentarios en la misma línea en que se declara una directiva, aunque estos sólo pueden ser comentarios de una línea que empiecen con `//` Por ejemplo, el siguiente comentario es válido:

```
#define TEST  // Ha habido algún error durante el preprocesado
```

Pero este otro no, pues aunque ocupa una línea tiene la sintaxis de los comentarios que pueden ocupar varias líneas:

```
#define TEST /* Ha habido algún error durante el preprocesado */
```

Definición de identificadores de preprocesado

Como ya se ha comentado, la principal utilidad del preprocesador en C# es la de permitir determinar cuáles regiones de código de un fichero fuente se han de compilar. Para ello, lo que se hace es encerrar las secciones de código opcionales dentro de directivas de compilación condicional, de modo que sólo se compilarán si determinados identificadores de preprocesado están definidos. Para definir un identificador de este tipo la directiva que se usa sigue esta sintaxis:

```
#define <nombreIdentificador>
```

Esta directiva define un identificador de preprocesado `<nombreIdentificador>`. Aunque más adelante estudiaremos detalladamente cuáles son los nombres válidos como identificadores en C#, por ahora podemos considerar que son válidos aquellos formados por uno o más caracteres alfanuméricos tales que no sean ni **true** ni **false** y no empiecen con un numero. Por ejemplo, para definir un identificador de preprocesado de nombre PRUEBA se haría:

```
#define PRUEBA
```

Por convenio se da a estos identificadores nombres en los que todas las letras se escriben en mayúsculas, como en el ejemplo anterior. Aunque es sólo un convenio y nada obliga a usarlo, ésta será la nomenclatura que usaremos en el presente documento, que es la usada por Microsoft en sus códigos de ejemplo. Conviene familiarizarse con ella porque por un lado hay mucho código escrito que la usa y por otro usarla facilita la lectura de nuestro código a los demás al ser la notación que esperarán encontrar.

Es importante señalar que cualquier definición de identificador ha de preceder a cualquier aparición de código en el fichero fuente. Por ejemplo, el siguiente código no es válido, pues antes del **#define** se ha incluido código fuente (el `class A`):

```
class A
#define PRUEBA
{}
```

Sin embargo, aunque no pueda haber código antes de un #define, sí que es posible incluir antes de él otras directivas de preprocesado con total libertad.

Existe una forma alternativa de definir un identificador de preprocesado y que además permite que dicha definición sólo sea válida en una compilación en concreto. Esta forma consiste en pasarle al compilador en su llamada la opción `/d:<nombreIdentificador>` (forma abreviada de `/define:<nombreIdentificador>`), caso en que durante la compilación se considerará que al principio de todos los ficheros fuente a compilar se encuentra definido el identificador indicado. Las siguientes tres formas de llamar al compilador son equivalentes y definen identificadores de preprocesado de nombres `PRUEBA` y `TRAZA` durante la compilación de un fichero fuente de nombre `ejemplo.cs`:

```
csc /d:PRUEBA /d:TRAZA ejemplo.cs
csc /d:PRUEBA,TRAZA ejemplo.cs
csc /d:PRUEBA;TRAZA ejemplo.cs
```

Nótese en el ejemplo que si queremos definir más de un identificador usando esta técnica tenemos dos alternativas: incluir varias opciones **/d** en la llamada al compilador o definir varios de estos identificadores en una misma opción **/d** separándolos mediante caracteres de coma (**,**) o punto y coma (**;**)

Si se trabaja con Visual Studio.NET en lugar de directamente con el compilador en línea de comandos, entonces puede conseguir mismo efecto a través de **View -> Property Pages -> Configuration Options -> Build -> Conditional Compilation Constants**, donde nuevamente usado el punto y coma (**;**) o la coma (**,**) como separadores, puede definir varias constantes. Para que todo funcione bien, antes de seleccionar **View** ha de seleccionar en el **Solution Explorer** (se abre con **View -> Solution Explorer**) el proyecto al que aplicar la definición de las constantes.

Finalmente, respecto al uso de **#define**sólo queda comentar que es posible definir varias veces una misma directiva sin que ello provoque ningún tipo de error en el compilador, lo que permite que podamos pasar tantos valores a la opción **/d** del compilador como queramos sin temor a que entren en conflicto con identificadores de preprocesado ya incluidos en los fuentes a compilar.

Eliminación de identificadores de preprocesado

Del mismo modo que es posible definir identificadores de preprocesado, también es posible eliminar definiciones de este tipo de identificadores previamente realizadas. Para ello la directiva que se usa tiene la siguiente sintaxis:

```
#undef <nombreIdentificador>
```

En caso de que se intente eliminar con esta directiva un identificador que no haya sido definido o cuya definición ya haya sido eliminada no se producirá error alguna, sino que simplemente la directiva de eliminación será ignorada. El siguiente ejemplo muestra un ejemplo de esto en el que el segundo **#undef** es ignorado:

```
#define VERSION1
#undef VERSION1
#undef VERSION1
```

Al igual que ocurría con las directivas **#define**, no se puede incluir código fuente antes de las directivas **#undef**, sino que, todo lo más, lo único que podrían incluirse antes que ellas serían directivas de preprocesado.

Compilación condicional

Como se ha repetido varias veces a lo largo del tema, la principal utilidad del preprocesador en C# es la de permitir la compilación de código condicional, lo que consiste en sólo permitir que se compile determinadas regiones de código fuente si las variables de preprocesado definidas cumplen alguna condición determinada. Para conseguir esto se utiliza el siguiente juego de directivas:

```
#if <condición1>
  <código1>
#elif <condición2>
  <código2>
...
#else
  <códigoElse>
#endif
```

El significado de una estructura como esta es que si se cumple <condición1> entonces se pasa al compilador el <código1>, si no ocurre esto pero se cumple <condición2> entonces lo que se pasaría al compilador sería <código2>, y así continuamente hasta que se llegue a una rama **#elif** cuya condición se cumpla. Si no se cumple ninguna pero hay una rama **#else** se pasará al compilador el <códigoElse>, pero si dicha rama no existiese entonces no se le pasaría código alguno y se continuaría preprocesando el código siguiente al **#endif** en el fuente original.

Aunque las ramas **#else** y **#endif** son opcionales, hay que tener cuidado y no mezclarlas ya que la rama **#else** sólo puede aparecer como última rama del bloque **#if**...**#endif**.

Es posible anidar varias estructuras **#if**...**#endif**, como muestra el siguiente código:

```
#define PRUEBA

using System;

class A
{
  public static void Main()
  {
    #if PRUEBA
      Console.Write ("Esto es una prueba");
      #if TRAZA
        Console.Write(" con traza");
      #elif !TRAZA
        Console.Write(" sin traza");
      #endif
    #endif
  }
}
```

Como se ve en el ejemplo, las condiciones especificadas son nombres de identificadores de preprocesado, considerándose que cada condición sólo se cumple si el identificador que se indica en ella está definido. O lo que es lo mismo: un identificador de preprocesado vale cierto (**true** en C#) si está definido y falso (**false** en C#) si no.

El símbolo **!** incluido en junto al valor de la directiva **#elif** es el símbolo de "no" lógico, y el **#elif** en el que se usa lo que nos permite es indicar que en caso de que no se encuentre definido el identificador de preprocesado TRAZA se han de pasar al compilador las instrucciones a continuación indicadas (o sea, el `Console.Write("sin traza");`)

El código fuente que el preprocesador pasará al compilador en caso de que compilemos sin especificar ninguna opción **/d** en la llamada al compilador será:

```
using System;

class A
{
  public static void Main()
  {
    Console.Write("Esto es una prueba");
    Console.Write(" sin traza");
  }
}
```

Nótese como en el código que se pasa al compilador ya no aparece ninguna directiva de preprocesado, pues lo que el preprocesador le pasa es el código resultante de aplicar al original las directivas de preprocesado que contuviese.

Asimismo, si compilásemos el código fuente original llamando al compilador con `/d:TRAZA`, lo que el preprocesador pasaría al compilador sería:

```
using System;

class A
{
  public static void Main()
  {
    Console.Write ("Esto es una prueba");
    Console.Write(" sin traza");
  }
}
```

Hasta ahora solo hemos visto que la condición de un **#if** o **#elif** puede ser un identificador de preprocesado, y que este valdrá `true` o `false` según esté o no definido. Pues bien, estos no son el único tipo de condiciones válidas en

C#, sino que también es posible incluir condiciones que contengan expresiones lógicas formadas por identificadores de preprocesado, operadores lógicos (**!** para "not", **&&** para "and" y **||** para "or"), operadores relacionales de igualdad (**==**) y desigualdad (**!=**), paréntesis (**(** y **)**) y los identificadores especiales **true**y **false**. Por ejemplo:

```
#if TRAZA // Se cumple si TRAZA esta definido.
#if TRAZA==true // Idem al ejemplo anterior aunque con una sintaxis menos cómoda
#if !TRAZA // Sólo se cumple si TRAZA no está definido.
#if TRAZA==false // Idema al ejemplo anterior aunque con una sintaxis menos cómoda
#if TRAZA == PRUEBA //  Solo se cumple si tanto TRAZA como PRUEBA están
                    // definidos o si no ninguno lo está.
#if TRAZA != PRUEBA //  Solo se cumple si TRAZA esta definido y PRUEBA no o
                    // viceversa
#if TRAZA && PRUEBA // Solo se cumple si están definidos TRAZA y PRUEBA.
#if TRAZA || PRUEBA // Solo se cumple si están definidos TRAZA o PRUEBA.
#if false // Nunca se cumple (por lo que es absurdo ponerlo)
#if true // Siempre se cumple (por lo que es absurdo ponerlo)
```

Es fácil ver que la causa de la restricción antes comentada de que no es válido dar un como nombre **true** o **false** a un identificador de preprocesado se debe al significado especial que estos tienen en las condiciones de los **#if** y **#elif**

Generación de avisos y errores

El preprocesador de C# también ofrece directivas que permiten generar avisos y errores durante el proceso de preprocesado en caso de que ser interpretadas por el preprocesador. Estas directivas tienen la siguiente sintaxis:

```
#warning <mensajeAviso>
#error <mensajeError>
```

La directiva **#warning** lo que hace al ser procesada es provocar que el compilador produzca un mensaje de aviso que siga el formato estándar usado por éste para ello y cuyo texto descriptivo tenga el contenido indicado en <mensajeAviso>; y **#error** hace lo mismo pero provocando un mensaje de error en vez de uno de aviso.

Usando directivas de compilación condicional se puede controlar cuando se han de producir estos mensajes, cuando se han de procesar estas directivas. De hecho la principal utilidad de estas directivas es permitir controlar errores de asignación de valores a los diferentes identificadores de preprocesado de un código, y un ejemplo de ello es el siguiente:

```
#warning Código aun no revisado
#define PRUEBA
#if PRUEBA && FINAL
  #error Un código no puede ser simultáneamente de prueba y versión final
#endif
class A
{}
```

En este código siempre se producirá el mensaje de aviso, pero el **#if** indica que sólo se producirá el mensaje de error si se han definido simultáneamente los identificadores de preprocesado PRUEBA y FINAL.

Como puede deducirse del ejemplo, el preprocesador de C# considera que los mensajes asociados a directivas **#warning** o **#error**son todo el texto que se encuentra tras el nombre de dichas directivas y hasta el final de la línea donde éstas aparecen. Por tanto, todo comentario que se incluya en una línea de este tipo será considerado como parte del mensaje a mostrar, y no como comentario como tal. Por ejemplo, ante la directiva:

```
#error La compilación ha fallado // Error
```

Lo que se mostrará en pantalla es un mensaje de la siguiente forma:

```
Fichero.cs(3,5): error CS1029: La compilación ha fallado // Error
```

Cambios en la numeración de líneas

Por defecto el compilador enumera las líneas de cada fichero fuente según el orden normal en que estas aparecen en el mismo, y este orden es el que sigue a la hora de informar de errores o de avisos durante la compilación. Sin embargo, hay situaciones en las que interesa cambiar esta numeración, y para ello se ofrece una directiva con la siguiente sintaxis:

```
#line <número> "<nombreFichero>"
```

Esta directiva indica al preprocesador que ha de considerar que la siguiente línea del fichero fuente en que aparece es la línea cuyo número se le indica, independientemente del valor que tuviese según la numeración usada en ese momento. El valor indicado en *"<nombreFichero>"* es opcional, y en caso de aparecer indica el nombre que se ha de considerar que tiene el fichero a la hora de dar mensajes de error. Un ejemplo:

```
#line 127 "csmace.cs"
```

Este uso de **#line**indica que el compilador ha de considerar que la línea siguiente es la línea 127 del fichero `csmace.cs`. A partir de ella se seguirá usando el sistema de numeración normal (la siguiente a esa será la 128 de `csmace.cs`, la próxima la 129, etc.) salvo que más adelante se vuelva a cambiar la numeración con otra directiva **#line**.

Aunque en principio puede parecer que esta directiva es de escasa utilidad, lo cierto es que suele venir bastante bien para la escritura de compiladores y otras herramientas que generen código en C# a partir de código escrito en otros lenguajes.

Marcación de regiones de código

Es posible marcar regiones de código y asociarles un nombre usando el juego de directivas **#region**y **#endregion**. Estas directivas se usan así:

```
#region <nombreRegión>
  <código>
#endregion
```

La utilidad que se dé a estas marcaciones depende de cada herramienta, pero en el momento de escribir estas líneas la única herramienta disponible que hacía uso de ellas era Visual Studio.NET, donde se usa para marcar código de modo que desde la ventana de código podamos expandirlo y contraerlo con una única pulsación de ratón. En concreto, en la ventana de código de Visual Studio aparecerá un símbolo [-] junto a las regiones de código así marcadas de manera que pulsando sobre él todo el código contenido en la región se comprimirá y será sustituido por el nombre dado en `<nombreRegión>`. Tras ello, el [-] se convertirá en un [+] y si volvemos a pulsarlo el código contraído se expandirá y recuperará su aspecto original. A continuación se muestra un ejemplo de cada caso:

Ilustración 4: Código de región expandido

Ilustración 5: Código de región contraído

Hay que tener cuidado al anidar regiones con directivas de compilación condicional, ya que todo bloque **#if**...**#endif**que comience dentro de una región ha de terminar también dentro de ella. Por tanto, el siguiente uso de la directiva **#region** no es valido ya que RegiónErrónea termina estando el bloque**#if**...**#endif** abierto:

```
#region RegiónErrónea
  #if A
#endregion
  #endif
```

Principio Página

Contacto - Datos legales

El lenguaje de programación C#

En esta página:

Tema 4: Aspectos léxicos

Comentarios

Un **comentario** es texto que incluido en el código fuente de un programa con la idea de facilitar su legibilidad a los programadores y cuyo contenido es, por defecto, completamente ignorado por el compilador. Suelen usarse para incluir información sobre el autor del código, para aclarar el significado o el porqué de determinadas secciones de código, para describir el funcionamiento de los métodos de las clases, etc.

En C# hay dos formas de escribir comentarios. La primera consiste en encerrar todo el texto que se desee comentar entre caracteres /* y */ siguiendo la siguiente sintaxis:

```
/*<texto>*/
```

Estos comentarios pueden abarcar tantas líneas como sea necesario. Por ejemplo:

```
/* Esto es un comentario
   que ejemplifica cómo se escribe comentarios que ocupen varias líneas */
```

Ahora bien, hay que tener cuidado con el hecho de que no es posible anidar comentarios de este tipo. Es decir, no vale escribir comentarios como el siguiente:

```
/*  Comentario contenedor /* Comentario contenido */  */
```

Esto se debe a que como el compilador ignora todo el texto contenido en un comentario y sólo busca la secuencia */ que marca su final, ignorará el segundo /* y cuando llegue al primer */ considerará que ha acabado el comentario abierto con el primer /* (no el abierto con el segundo) y pasará a buscar código. Como el */ sólo lo admite si ha detectado antes algún comentario abierto y aún no cerrado (no mientras busca código), cuando llegue al segundo */ considerará que ha habido un error ya que encontrará el */ donde esperaba encontrar código

Dado que muchas veces los comentarios que se escriben son muy cortos y no suelen ocupar más de una línea, C# ofrece una sintaxis alternativa más compacta para la escritura este tipo de comentarios en las que se considera como indicador del

comienzo del comentario la pareja de caracteres **//** y como indicador de su final el fin de línea. Por tanto, la sintaxis que siguen estos comentarios es:

```
// <texto>
```

Y un ejemplo de su uso es:

```
// Este comentario ejemplifica como escribir comentarios abreviados de una sola línea
```

Estos comentarios de una sola línea sí que pueden anidarse sin ningún problema. Por ejemplo, el siguiente comentario es perfectamente válido:

```
// Comentario contenedor // Comentario contenido
```

Identificadores

Al igual que en cualquier lenguaje de programación, en C# un **identificador** no es más que, como su propio nombre indica, un nombre con el que identificaremos algún elemento de nuestro código, ya sea una clase, una variable, un método, etc.

Típicamente el nombre de un identificador será una secuencia de cualquier número de caracteres alfanuméricos -incluidas vocales acentuadas y eñes- tales que el primero de ellos no sea un número. Por ejemplo, identificadores válidos serían: `Arriba`, `caña`, `C3P0`, `áëÎò`, etc; pero no lo serían `3com`, `127`, etc.

Sin embargo, y aunque por motivos de legibilidad del código no se recomienda, C# también permite incluir dentro de un identificador caracteres especiales imprimibles tales como símbolos de diéresis, subrayados, etc. siempre y cuando estos no tengan un significado especial dentro del lenguaje. Por ejemplo, también serían identificadores válidos `_barco_`, `c¨k` y `A·B` pero no `C#` (`#` indica inicio de directiva de preprocesado) o `a!b` (**!** indica operación lógica "not")

Finalmente, C# da la posibilidad de poder escribir identificadores que incluyan caracteres Unicode que no se puedan imprimir usando el teclado de la máquina del programador o que no sean directamente válidos debido a que tengan un significado especial en el lenguaje. Para ello, lo que permite es escribir estos caracteres usando **secuencias de escape**, que no son más que secuencias de caracteres con las sintaxis:

```
\u<dígito><dígito><dígito><dígito>
```

ó

```
\U<dígito><dígito><dígito><dígito><dígito><dígito><dígito><dígito>
```

Estos dígitos indican es el código Unicode del carácter que se desea incluir como parte del identificador, y cada uno de ellos ha de ser un dígito hexadecimal válido. (`0-9`, `a-f` ó `A-F`) Hay que señalar que el carácter **u** ha de escribise en minúscula cuando se indiquen caracteres Unicode con 4 dígitos y en mayúscula cuando se indiquen con caracteres de ocho. Ejemplos de identificadores válidos son `C\u0064` (equivale a `C#`, pues 64 es el código de `#` en Unicode) ó a\U00000033b (equivale a `a!b`).

Palabras reservadas

Aunque antes se han dado una serie de restricciones sobre cuáles son los nombres válidos que se pueden dar en C# a los identificadores, falta todavía por dar una: los siguientes nombres no son válidos como identificadores ya que tienen un significado especial en el lenguaje:

```
abstract, as, base, bool, break, byte, case, catch, char, checked, class, const,
continue, decimal, default, delegate, do, double, else, enum, event, explicit,
extern, false, finally, fixed, float, for, foreach, goto, if, implicit, in, int,
interface,
internal, lock, is, long, namespace, new, null, object, operator, out, override,
params, private, protected, public, readonly, ref, return, sbyte, sealed, short,
sizeof, stackalloc, static, string, struct, switch, this, throw, true, try, typeof,
uint, ulong, unchecked, unsafe, ushort, using, virtual, void, while
```

Aparte de estas palabras reservadas, si en futuras implementaciones del lenguaje se decidiese incluir nuevas palabras reservadas, Microsoft dice que dichas palabras habrían de incluir al menos dos símbolos de subrayado consecutivos (__) Por tanto, para evitar posibles conflictos futuros no se recomienda dar a nuestros identificadores nombres que contengan dicha

secuencia de símbolos.

Aunque directamente no podemos dar estos nombres a nuestros identificadores, C# proporciona un mecanismo para hacerlo indirectamente y de una forma mucho más legible que usando secuencias de escape. Este mecanismo consiste en usar el carácter `@` para prefijar el nombre coincidente con el de una palabra reservada que queramos dar a nuestra variable. Por ejemplo, el siguiente código es válido:

```
class @class
{
  static void @static(bool @bool)
  {
    if (@bool)
      Console.WriteLine("cierto");
    else
      Console.WriteLine("falso");
  }
}
```

Lo que se ha hecho en el código anterior ha sido usar **@** para declarar una clase de nombre `class` con un método de nombre `static` que toma un parámetro de nombre `bool`, aún cuando todos estos nombres son palabras reservadas en C#.

Hay que precisar que aunque el nombre que nosotros escribamos sea por ejemplo `@class`, el nombre con el que el compilador va a tratar internamente al identificador es solamente `class`. De hecho, si desde código escrito en otro lenguaje adaptado a .NET distinto a C# hacemos referencia a éste identificador y en ese lenguaje su nombre no es una palabra reservada, el nombre con el que deberemos referenciarlo es `class`, y no `@class` (si también fuese en ese lenguaje palabra reservada habría que referenciarlo con el mecanismo que el lenguaje incluyese para ello, que quizás también podría consistir en usar `@` como en C#)

En realidad, el uso de @ no se tiene porqué limitar a preceder palabras reservadas en C#, sino que podemos preceder cualquier nombre con él. Sin embargo, hacer esto no se recomienda, pues es considerado como un mal hábito de programación y puede provocar errores muy sutiles como el que muestra el siguiente ejemplo:

```
class A
{
  int a;     // (1)
  int @a; // (2)

  public static void Main()
  {}
}
```

Si intentamos compilar este código se producirá un error que nos informará de que el campo de nombre a ha sido declarado múltiples veces en la clase A. Esto se debe a que como `@` no forma parte en realidad del nombre del identificador al que precede, las declaraciones marcadas con comentarios como (1) y (2) son equivalentes.

Hay que señalar por último una cosa respecto al carácter `@`: sólo puede preceder al nombre de un identificador, pero no puede estar contenido dentro del mismo. Es decir, identificadores como `i5322@fie.us.es` no son válidos.

Literales

Un **literal** es la representación explícita de los valores que pueden tomar los tipos básicos del lenguaje. A continuación se explica cuál es la sintaxis con que se escriben los literales en C# desglosándolos según el tipo de valores que representan:

- **Literales enteros:** Un número entero se puede representar en C# tanto en formato decimal como hexadecimal. En el primer caso basta escribir los dígitos decimales (`0-9`) del número unos tras otros, mientras que en el segundo hay que preceder los dígitos hexadecimales (`0-9`, `a-f`, `A-F`) con el prefijo **`0x`**. En ambos casos es posible preceder el número de los operadores **`+`** ó **`-`** para indicar si es positivo o negativo, aunque si no se pone nada se considerará que es positivo. Ejemplos de literales enteros son `0`, `5`, `+15`, `-23`, `0x1A`, `-0x1a`, etc

  En realidad, la sintaxis completa para la escritura de literales enteros también puede incluir un sufijo que indique el tipo de dato entero al que ha de pertenecer el literal. Esto no lo veremos hasta el *Tema 7: Variables y tipos de datos*.

- **Literales reales:** Los números reales se escriben de forma similar a los enteros, aunque sólo se pueden escribir en forma decimal y para separar la parte entera de la real usan el tradicional punto decimal (carácter `.`) También es posible representar los reales en formato científico, usándose para indicar el exponente los caracteres **e** ó **E**. Ejemplos de literales reales son `0.0`, `5.1`, `-5.1`, `+15.21`, `3.02e10`, `2.02e-2`, `98.8E+1`, etc.

  Al igual que ocurría con los literales enteros, los literales reales también pueden incluir sufijos que indiquen el tipo de dato real al que pertenecen, aunque nuevamente no los veremos hasta el *Tema 7: Variables y tipos de datos*

- **Literales lógicos:** Los únicos literales lógicos válidos son **true** y **false**, que respectivamente representan los valores lógicos cierto y falso.

- **Literales de carácter:** Prácticamente cualquier carácter se puede representar encerrándolo entre comillas simples. Por ejemplo, `'a'` (letra a), `' '` (carácter de espacio), '?' (símbolo de interrogación), etc. Las únicas excepciones a esto son los caracteres que se muestran en la **Tabla 4.1**, que han de representarse con secuencias de escape que indiquen su valor como código Unicode o mediante un formato especial tal y como se indica a continuación:

Carácter	Código de escape Unicode	Código de escape especial
Comilla simple	\u0027	\'
Comilla doble	\u0022	\"
Carácter nulo	\u0000	\0
Alarma	\u0007	\a
Retroceso	\u0008	\b
Salto de página	\u000C	\f
Nueva línea	\u000A	\n
Retorno de carro	\u000D	\r
Tabulación horizontal	\u0009	\t
Tabulación vertical	\u000B	\v
Barra invertida	\u005C	\\

Tabla 4.1: Códigos de escape especiales

En realidad, de la tabla anterior hay que matizar que el carácter de comilla doble también puede aparecer dentro de un literal de cadena directamente, sin necesidad de usar secuencias de escape. Por tanto, otros ejemplos de literales de carácter válidos serán `'\"'`, `'"'`, `'\f'`, `'\u0000'`, `'\\'`, `'\''`, etc.

Aparte de para representar los caracteres de la tabla anterior, también es posible usar los códigos de escape Unicode para representar cualquier código Unicode, lo que suele usarse para representar literales de caracteres no incluidos en los teclados estándares.

Junto al formato de representación de códigos de escape Unicode ya visto, C# incluye un formato abreviado para representar estos códigos en los literales de carácter si necesidad de escribir siempre cuatro dígitos aún cuando el código a representar tenga muchos ceros en su parte izquierda. Este formato consiste en preceder el código de **\x** en vez de **\u**. De este modo, los literales de carácter `'\U00000008'`, `'\u0008'`, `'\x0008'`, `'\x008'`, `'\x08'` y `'\x8'` son todos equivalentes. Hay que tener en cuenta que este formato abreviado sólo es válido en los literales de carácter, y no a la hora de dar nombres a los identificadores.

- **Literales de cadena:** Una **cadena** no es más que una secuencia de caracteres encerrados entre comillas dobles. Por ejemplo `"Hola, mundo"`, `"camión"`, etc. El texto contenido dentro estos literales puede estar formado por cualquier número de literales de carácter concatenados y sin las comillas simples, aunque si incluye comillas dobles éstas han de escribirse usando secuencias de escape porque si no el compilador las interpretaría como el final de la cadena.

Aparte del formato de escritura de literales de cadenas antes comentado, que es el comúnmente usado en la mayoría de lenguajes de programación, C# también admite un nuevo formato para la escritura estos literales tipo de literales consistente en precederlas de un símbolo **@**, caso en que todo el contenido de la cadena sería interpretado tal cual, sin considerar la existencia de secuencias de escape. A este tipo de literales se les conoce como **literales de cadena planos** y pueden incluso ocupar múltiples líneas. La siguiente tabla recoge algunos ejemplos de cómo se interpretan:

Literal de cadena	Interpretado como...
"Hola\tMundo"	Hola Mundo
@"Hola\tMundo"	Hola\tMundo
@"Hola Mundo"	Hola Mundo
@"""Hola Mundo"""	"Hola Mundo"

Tabla 4.2: Ejemplos de literales de cadena planos

El último ejemplo de la tabla se ha aprovechado para mostrar que si dentro de un literal de cadena plano se desea incluir caracteres de comilla doble sin que sean confundidos con el final de la cadena basta duplicarlos.

- **Literal nulo:** El literal nulo es un valor especial que se representa en C# con la palabra reservada **null** y se usa como valor de las variables de objeto no inicializadas para así indicar que contienen referencias nulas.

Operadores

Un **operador** en C# es un símbolo formado por uno o más caracteres que permite realizar una determinada operación entre uno o más datos y produce un resultado.

A continuación se describen cuáles son los operadores incluidos en el lenguaje clasificados según el tipo de operaciones que permiten realizar, aunque hay que tener en cuenta que C# permite la redefinición del significado de la mayoría de los operadores según el tipo de dato sobre el que se apliquen, por lo que lo que aquí se cuenta se corresponde con los usos más comunes de los mismos:

- **Operaciones aritméticas:** Los operadores aritméticos incluidos en C# son los típicos de suma (+), resta (-), producto (*), división (/) y módulo (%) También se incluyen operadores de "menos unario" (-) y "más unario" (+)

  Relacionados con las operaciones aritméticas se encuentran un par de operadores llamados **checked** y **unchecked** que permiten controlar si se desea detectar los desbordamientos que puedan producirse si al realizar este tipo de operaciones el resultado es superior a la capacidad del tipo de datos de sus operandos. Estos operadores se usan así:

  ```
  checked (<expresiónAritmética>)
  unchecked(<expresiónAritmética>)
  ```

  Ambos operadores calculan el resultado de <expresiónAritmética> y lo devuelven si durante el cálculo no se produce ningún desbordamiento. Sin embargo, en caso de que haya desbordamiento cada uno actúa de una forma distinta: **checked** provoca un error de compilación si <expresiónAritmética> es una expresión constante y una excepción **System.OverflowException** si no lo es, mientras que **unchecked** devuelve el resultado de la expresión aritmética truncado para modo que quepa en el tamaño esperado.

  Por defecto, en ausencia de los operadores **checked** y **unchecked** lo que se hace es evaluar las operaciones aritméticas entre datos constantes como si se les aplicase **checked** y las operaciones entre datos no constantes como si se les hubiese aplicado **unchecked**.

- **Operaciones lógicas:** Se incluyen operadores que permiten realizar las operaciones lógicas típicas: "and" (&& y &), "or" (|| y |), "not" (!) y "xor" (^)

  Los operadores && y || se diferencia de & y | en que los primeros realizan evaluación perezosa y los segundos no. La evaluación perezosa consiste en que si el resultado de evaluar el primer operando permite deducir el resultado de la operación, entonces no se evalúa el segundo y se devuelve dicho resultado directamente, mientras que la evaluación no perezosa consiste en evaluar siempre ambos operandos. Es decir, si el primer operando de una operación && es falso se devuelve false directamente, sin evaluar el segundo; y si el primer operando de una || es cierto se

devuelve `true` directamente, sin evaluar el otro.

- **Operaciones relacionales:** Se han incluido los tradicionales operadores de igualdad (`==`), desigualdad (`!=`), "mayor que" (`>`), "menor que" (`<`), "mayor o igual que" (`>=`) y "menor o igual que" (`<=`)

- **Operaciones de manipulación de bits:** Se han incluido operadores que permiten realizar a nivel de bits operaciones "and" (`&`), "or" (`|`), "not" (`~`), "xor" (`^`), desplazamiento a izquierda (`<<`) y desplazamiento a derecha (`>>`) El operador `<<` desplaza a izquierda rellenando con ceros, mientras que el tipo de relleno realizado por `>>` depende del tipo de dato sobre el que se aplica: si es un dato con signo mantiene el signo, y en caso contrario rellena con ceros.

- **Operaciones de asignación:** Para realizar asignaciones se usa en C# el operador `=`, operador que además de realizar la asignación que se le solicita devuelve el valor asignado. Por ejemplo, la expresión `a = b` asigna a la variable a el valor de la variable b y devuelve dicho valor, mientras que la expresión `c = a = b` asigna a c y a el valor de b (el operador `=` es asociativo por la derecha)

También se han incluido operadores de asignación compuestos que permiten ahorrar tecleo a la hora de realizar asignaciones tan comunes como:

```
temperatura = temperatura + 15;   // Sin usar asignación compuesta
temperatura += 15;                // Usando asignación compuesta
```

Las dos líneas anteriores son equivalentes, pues el operador compuesto `+=`lo que hace es asignar a su primer operando el valor que tenía más el valor de su segundo operando. Como se ve, permite compactar bastante el código.

Aparte del operador de asignación compuesto `+=`, también se ofrecen operadores de asignación compuestos para la mayoría de los operadores binarios ya vistos. Estos son: `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `<<=` y `>>=`. Nótese que no hay versiones compuestas para los operadores binarios `&&` y `||`.

Otros dos operadores de asignación incluidos son los de incremento(`++`) y decremento (`--`) Estos operadores permiten, respectivamente, aumentar y disminuir en una unidad el valor de la variable sobre el que se aplican. Así, estas líneas de código son equivalentes:

```
temperatura = temperatura + 1;  // Sin usar asignación compuesta ni incremento
temperatura += 1;               // Usando asignación compuesta
temperatura++;                  // Usando incremento
```

Si el operador `++` se coloca tras el nombre de la variable (como en el ejemplo) devuelve el valor de la variable antes de incrementarla, mientras que si se coloca antes, devuelve el valor de ésta tras incrementarla; y lo mismo ocurre con el operador `--`. Por ejemplo:

```
c = b++; // Se asigna a c el valor de b y luego se incrementa b
c = ++b; // Se incrementa el valor de b y luego se asigna a c
```

La ventaja de usar los operadores `++` y `--` es que en muchas máquinas son más eficientes que el resto de formas de realizar sumas o restas de una unidad, pues el compilador traducirlos en una única instrucción en código máquina.

- **Operaciones con cadenas:** Para realizar operaciones de concatenación de cadenas se puede usar el mismo operador que para realizar sumas, ya que en C# se ha redefinido su significado para que cuando se aplique entre operandos que sean cadenas o que sean una cadena y un carácter lo que haga sea concatenarlos. Por ejemplo, "Hola"+" mundo" devuelve "Hola mundo", y "Hola mund" + "o" también.

- **Operaciones de acceso a tablas:** Una **tabla** es un conjunto de ordenado de objetos de tamaño fijo. Para acceder a cualquier elemento de este conjunto se aplica el operador postfijo `[]` sobre la tabla para indicar entre corchetes la posición que ocupa el objeto al que se desea acceder dentro del conjunto. Es decir, este operador se usa así:

  ```
  [<posiciónElemento>]
  ```

  Un ejemplo de su uso en el que se asigna al elemento que ocupa la posición 3 en una tabla de nombre tablaPrueba el valor del elemento que ocupa la posición 18 de dicha tabla es el siguiente:

```
tablaPrueba[3] = tablaPrueba[18];
```

Las tablas se estudian detenidamente en el *Tema 7: Variables y tipos de datos*

- **Operador condicional:** Es el único operador incluido en C# que toma 3 operandos, y se usa así:

```
<condición> ? <expresión1> : <expresión2>
```

El significado del operando es el siguiente: se evalúa `<condición>` Si es cierta se devuelve el resultado de evaluar `<expresión1>`, y si es falsa se devuelve el resultado de evaluar `<condición2>`. Un ejemplo de su uso es:

```
b = (a>0)? a : 0;       // Suponemos a y b de tipos enteros
```

En este ejemplo, si el valor de la variable a es superior a 0 se asignará a `b` el valor de `a`, mientras que en caso contrario el valor que se le asignará será 0.

Hay que tener en cuenta que este operador es asociativo por la derecha, por lo que una expresión como `a?b:c?d:e` es equivalente a `a?b:(c?d:e)`

No hay que confundir este operador con la instrucción condicional **if** que se tratará en el *Tema 8:Instrucciones*, pues aunque su utilidad es similar al de ésta, `?` devuelve un valor e `if` no.

- **Operaciones de delegados:** Un **delegado** es un objeto que puede almacenar en referencias a uno o más métodos y a través del cual es posible llamar a estos métodos. Para añadir objetos a un delegado se usan los operadores `+` y `+=`, mientras que para quitárselos se usan los operadores `-` y `-=`. Estos conceptos se estudiarán detalladamente en el *Tema 13: Eventos y delegados*

- **Operaciones de acceso a objetos:** Para acceder a los miembros de un objeto se usa el operador `.`, cuya sintaxis es:

```
<objeto>.<miembro>
```

Si a es un objeto, ejemplos de cómo llamar a diferentes miembros suyos son:

```
a.b = 2;  // Asignamos a su propiedad a el valor 2
a.f();    // Llamamos a su método f()
a.g(2);   // Llamamos a su método g() pasándole como parámetro el valor entero 2
a.c += new adelegado(h) // Asociamos a su evento c el código del método h() de
                        // "tipo" adelegado
```

No se preocupe si no conoce los conceptos de métodos, propiedades, eventos y delegados en los que se basa este ejemplo, pues se explican detalladamente en temas posteriores.

- **Operaciones con punteros:** Un puntero es una variable que almacena una referencia a una dirección de memoria. Para obtener la dirección de memoria de un objeto se usa el operador `&`, para acceder al contenido de la dirección de memoria almacenada en un puntero se usa el operador `*`, para acceder a un miembro de un objeto cuya dirección se almacena en un puntero se usa `->`, y para referenciar una dirección de memoria de forma relativa a un puntero se le aplica el operador `[]` de la forma puntero[desplazamiento]. Todos estos conceptos se explicarán más a fondo en el *Tema 18: Código inseguro*.

- **Operaciones de obtención de información sobre tipos:** De todos los operadores que nos permiten obtener información sobre tipos de datos el más importante es `typeof`, cuya forma de uso es:

```
typeof(<nombreTipo>)
```

Este operador devuelve un objeto de tipo **`System.Type`** con información sobre el tipo de nombre `<nombreTipo>` que podremos consultar a través de los miembros ofrecidos por dicho objeto. Esta información incluye detalles tales como cuáles son sus miembros, cuál es su tipo padre o a qué espacio de nombres pertenece.

Si lo que queremos es determinar si una determinada expresión es de un tipo u otro, entonces el operador a usar es `is`, cuya sintaxis es la siguiente:

```
<expresión> is <nombreTipo>
```

El significado de este operador es el siguiente: se evalúa `<expresión>`. Si el resultado de ésta es del tipo cuyo nombre se indica en <nombreTipo>se devuelve **true**; y si no, se devuelve **false**. Como se verá en el *Tema 5: Clases*, este operador suele usarse en métodos polimórficos.

Finalmente, C# incorpora un tercer operador que permite obtener información sobre un tipo de dato: **sizeof** Este operador permite obtener el número de bytes que ocuparán en memoria los objetos de un tipo, y se usa así:

```
sizeof(<nombreTipo>)
```

sizeof sólo puede usarse dentro de código inseguro, que por ahora basta considerar que son zonas de código donde es posible usar punteros. No será hasta el *Tema 18: Código inseguro* cuando lo trataremos en profundidad.

Además, **sizeof** sólo se puede aplicar sobre nombres de tipos de datos cuyos objetos se puedan almacenar directamente en pila. Es decir, que sean estructuras (se verán en el *Tema 13*) o tipos enumerados (se verán en el *Tema 14*)

- **Operaciones de creación de objetos**: El operador más típicamente usado para crear objetos es **new**, que se usa así:

```
new <nombreTipo>(<parametros>)
```

Este operador crea un objeto de `<nombreTipo>` pasándole a su método constructor los parámetros indicados en `<parámetros>` y devuelve una referencia al mismo. En función del tipo y número de estos parámetros se llamará a uno u otro de los constructores del objeto. Así, suponiendo que a1 y a2 sean variables de tipo Avión, ejemplos de uso del operador **new** son:

```
Avión a1 = new Avión(); // Se llama al constructor sin parámetros de Avión
Avión a2  = new Avión("Caza"); // Se llama al constructor de Avión que toma
                               // como parámetro una cadena
```

En caso de que el tipo del que se haya solicitado la creación del objeto sea una clase, éste se creará en memoria dinámica, y lo que **new** devolverá será una referencia a la dirección de pila donde se almacena una referencia a la dirección del objeto en memoria dinámica. Sin embargo, si el objeto a crear pertenece a una estructura o a un tipo enumerado, entonces éste se creará directamente en la pila y la referencia devuelta por el **new** se referirá directamente al objeto creado. Por estas razones, a las clases se les conoce como **tipos referencia** ya que de sus objetos en pila sólo se almacena una referencia a la dirección de memoria dinámica donde verdaderamente se encuentran; mientras que a las estructuras y tipos enumerados se les conoce como **tipos valor** ya sus objetos se almacenan directamente en pila.

C# proporciona otro operador que también nos permite crear objetos. Éste es **stackalloc**, y se usa así:

```
stackalloc <nombreTipo>[<nElementos>]
```

Este operador lo que hace es crear en pila una tabla de tantos elementos de tipo `<nombreTipo>` como indique `<nElementos>` y devolver la dirección de memoria en que ésta ha sido creada. Por ejemplo:

stackalloc sólo puede usarse para inicializar punteros a objetos de tipos valor declarados como variables locales. Por ejemplo:

```
int * p = stackalloc[100]; // p apunta a una tabla de 100 enteros.
```

- **Operaciones de conversión:** Para convertir unos objetos en otros se utiliza el operador de conversión, que no consiste más que en preceder la expresión a convertir del nombre entre paréntesis del tipo al que se desea convertir el resultado de evaluarla. Por ejemplo, si l es una variable de tipo **long** y se desea almacenar su valor dentro de una variable de tipo **int** llamada i, habría que convertir previamente su valor a tipo **int** así:

```
i = (int) l;  // Asignamos a i el resultado de convertir el valor de l a tipo int
```

Los tipos **int** y **long** están predefinidos en C# y permite almacenar valores enteros con signo. La capacidad de **int** es de 32 bits, mientras que la de **long** es de 64 bits. Por tanto, a no ser que hagamos uso del operador de conversión, el compilador no nos dejará hacer la asignación, ya que al ser mayor la capacidad de los **long**, no todo valor que se pueda almacenar en un **long** tiene porqué poderse almacenar en un **int**. Es decir, no es válido:

```
i = l; //ERROR: El valor de l no tiene porqué caber en i
```

Esta restricción en la asignación la impone el compilador debido a que sin ella podrían producirse errores muy difíciles de detectar ante truncamientos no esperados debido al que el valor de la variable fuente es superior a la capacidad de la variable destino.

Existe otro operador que permite realizar operaciones de conversión de forma muy similar al ya visto. Éste es el operador **as**, que se usa así:

```
<expresión> as <tipoDestino>
```

Lo que hace es devolver el resultado de convertir el resultado de evaluar <expresión> al tipo indicado en <tipoDestino> Por ejemplo, para almacenar en una variable p el resultado de convertir un objeto t a tipo tipo Persona se haría:

```
p = t as Persona;
```

Las únicas diferencias entre usar uno u otro operador de conversión son:

- **as** sólo es aplicable a tipos referencia y sólo a aquellos casos en que existan conversiones predefinidas en el lenguaje. Como se verá más adelante, esto sólo incluye conversiones entre un tipo y tipos padres suyos y entre un tipo y tipos hijos suyos.

  Una consecuencia de esto es que el programador puede definir cómo hacer conversiones de tipos por él definidos y otros mediante el operador **()**, pero no mediante **as**.

  Esto se debe a que as únicamente indica que se desea que una referencia a un objeto en memoria dinámica se trate como si el objeto fuese de otro tipo, pero no implica conversión ninguna. Sin embargo, **()** sí que implica conversión si el `<tipoDestino>` no es compatible con el tipo del objeto referenciado. Obviamente, el operador se aplirará mucho más rápido en los casos donde no sea necesario convertir.

- En caso de que se solicite hacer una conversión inválida **as** devuelve **null** mientras que **()** produce una excepción **System.InvalidCastException**.

Principio Página

El lenguaje de programación C#

En esta página:

Tema 5: Clases

Definición de clases

Conceptos de clase y objeto

C# es un lenguaje orientado a objetos puro, lo que significa que todo con lo que vamos a trabajar en este lenguaje son objetos. Un **objeto** es un agregado de datos y de métodos que permiten manipular dichos datos, y un programa en C# no es más que un conjunto de objetos que interaccionan unos con otros a través de sus métodos.

Una **clase** es la definición de las características concretas de un determinado tipo de objetos. Es decir, de cuáles son los datos y los métodos de los que van a disponer todos los objetos de ese tipo. Por esta razón, se suele decir que el **tipo de dato** de un objeto es la clase que define las características del mismo.

Sintaxis de definición de clases

La sintaxis básica para definir una clase es la que a continuación se muestra:

```
class <nombreClase>
{
  <miembros>
}
```

De este modo se definiría una clase de nombre <nombreClase> cuyos miembros son los definidos en <miembros> Los **miembros** de una clase son los datos y métodos de los que van a disponer todos los objetos de la misma. Un ejemplo de cómo declarar una clase de nombre A que no tenga ningún miembro es la siguiente:

```
class A
```

```
{}
```

Una clase así declarada no dispondrá de ningún miembro a excepción de los implícitamente definidos de manera común para todos los objetos que creemos en C#. Estos miembros los veremos dentro de poco en este mismo tema bajo el epígrafe *La clase primigenia: System.Object*.

Aunque en C# hay muchos tipos de miembros distintos, por ahora vamos a considerar que estos únicamente pueden ser campos o métodos y vamos a hablar un poco acerca de ellos y de cómo se definen:

- **Campos:** Un **campo** es un dato común a todos los objetos de una determinada clase. Para definir cuáles son los campos de los que una clase dispone se usa la siguiente sintaxis dentro de la zona señalada como `<miembros>` en la definición de la misma:

```
<tipoCampo> <nombreCampo>;
```

El nombre que demos al campo puede ser cualquier identificador que queramos siempre y cuando siga las reglas descritas en el *Tema 4: Aspectos Léxicos* para la escritura de identificadores y no coincida con el nombre de ningún otro miembro previamente definido en la definición de clase.

Los campos de un objeto son a su vez objetos, y en `<tipoCampo>` hemos de indicar cuál es el tipo de dato del objeto que vamos a crear. Éste tipo puede corresponderse con cualquiera que los predefinidos en la BCL o con cualquier otro que nosotros hallamos definido siguiendo la sintaxis arriba mostrada. A continuación se muestra un ejemplo de definición de una clase de nombre Persona que dispone de tres campos:

```
class Persona
{
  string Nombre;   // Campo de cada objeto Persona que almacena su nombre
  int Edad;        // Campo de cada objeto Persona que almacena su edad
  string NIF;      // Campo de cada objeto Persona que almacena su NIF
}
```

Según esta definición, todos los objetos de clase Persona incorporarán campos que almacenarán cuál es el nombre de la persona que cada objeto representa, cuál es su edad y cuál es su NIF. El tipo **int** incluido en la definición del campo Edad es un tipo predefinido en la BCL cuyos objetos son capaces de almacenar números enteros con signo comprendidos entre -2.147.483.648 y 2.147.483.647 (32 bits), mientras que **string** es un tipo predefinido que permite almacenar cadenas de texto que sigan el formato de los literales de cadena visto en el *Tema 4: Aspectos Léxicos*

Para acceder a un campo de un determinado objeto se usa la sintaxis:

```
<objeto>.<campo>
```

Por ejemplo, para acceder al campo Edad de un objeto Persona llamado p y cambiar su valor por 20 se haría:

```
p.Edad = 20;
```

En realidad lo marcado como `<objeto>` no tiene porqué ser necesariamente el nombre de algún objeto, sino que puede ser cualquier expresión que produzca como resultado una referencia no nula a un objeto (si produjese `null` se lanzaría una excepción del tipo predefinido `System.NullPointerException`)

- **Métodos:** Un **método** es un conjunto de instrucciones a las que se les asocia un nombre de modo que si se desea ejecutarlas basta referenciarlas a través de dicho nombre en vez de tener que escribirlas. Dentro de estas instrucciones es posible acceder con total libertad a la información almacenada en los campos pertenecientes a la clase dentro de la que el método se ha definido, por lo que como al principio del tema se indicó, los métodos permiten manipular los datos almacenados en los objetos.

  La sintaxis que se usa en C# para definir los métodos es la siguiente:

```
<tipoDevuelto> <nombreMétodo> (<parametros>)
{
  <instrucciones>
}
```

Todo método puede devolver un objeto como resultado de la ejecución de las instrucciones que lo forman, y el tipo de dato al que pertenece este objeto es lo que se indica en <tipoDevuelto>. Si no devuelve nada se indica **void**, y si devuelve algo es obligatorio finalizar la ejecución de sus instrucciones con alguna instrucción **return** <objeto>; que indique qué objeto ha de devolverse.

Opcionalmente todo método puede recibir en cada llamada una lista de objetos a los que podrá acceder durante la ejecución de sus instrucciones. En *<parametros>* se indica es cuáles son los tipos de dato de estos objetos y cuál es el nombre con el que harán referencia las instrucciones del método a cada uno de ellos. Aunque los objetos que puede recibir el método pueden ser diferentes cada vez que se solicite su ejecución, siempre han de ser de los mismos tipos y han de seguir el orden establecido en *<parametros>*.

Un ejemplo de cómo declarar un método de nombre Cumpleaños es la siguiente modificación de la definición de la clase Persona usada antes como ejemplo:

```
class Persona
{
  string Nombre;    // Campo de cada objeto Persona que almacena su nombre
  int Edad;         // Campo de cada objeto Persona que almacena su edad
  string NIF;       // Campo de cada objeto Persona que almacena su NIF
  void Cumpleaños() // Incrementa en uno de la edad del objeto Persona
  {
    Edad++;
  }
}
```

La sintaxis usada para llamar a los métodos de un objeto es la misma que la usada para llamar a sus campos, sólo que ahora tras el nombre del método al que se desea llamar hay que indicar entre paréntesis cuáles son los valores que se desea dar a los parámetros del método al hacer la llamada. O sea, se escribe:

```
<objeto>.<método>(<parámetros>)
```

Como es lógico, si el método no tomase parámetros se dejarían vacíos los parámetros en la llamada al mismo. Por ejemplo, para llamar al método Cumpleaños() de un objeto Persona llamado p se haría:

```
p.Cumpleaños();   // El método no toma parámetros, luego no le pasamos ninguno
```

Es importante señalar que en una misma clase pueden definirse varios métodos con el mismo nombre siempre y cuando tomen diferente número o tipo de parámetros. A esto se le conoce como **sobrecargar de métodos**, y es posible ya que cuando se les llame el compilador sabrá a cual llamar a partir de *<parámetros>* pasados en la llamada.

Sin embargo, lo que no es permite es definir varios métodos que sólo se diferencien en su valor de retorno, ya que como éste no se tiene porqué indicar al llamarlos no podría diferenciarse a que método en concreto se hace referencia en cada llamada. Por ejemplo, a partir de la llamada:

```
p.Cumpleaños();
```

Si además de la versión de Cumpleaños() que no retorna nada hubiese otra que retornase un **int**, ¿cómo sabría entonces el compilador a cuál llamar?

Antes de continuar es preciso señalar que en C# todo, incluido los literales, son objetos del tipo de cada literal y por tanto pueden contar con miembros a los que se accedería tal y como se ha explicado. Para entender esto no hay nada mejor que un ejemplo:

```
string s = 12.ToString();
```

Este código almacena el literal de cadena "12" en la variable s, pues 12 es un objeto de tipo **int** (tipo que representa enteros) y cuenta cuenta con el método común a todos los **int**s llamado **ToString()** que lo que hace es devolver una cadena cuyos caracteres son los dígitos que forman el entero representado por el **int** sobre el que se aplica; y como la variable s es de tipo **string** (tipo que representa cadenas) es perfectamente posible almacenar dicha cadena en ella, que es lo que se hace en el código anterior.

Creación de objetos

Operador new

Ahora que ya sabemos cómo definir las clases de objetos que podremos usar en nuestras aplicaciones ha llegado el momento de explicar cómo crear objetos de una determinada clase. Algo de ello ya se introdujo en el *Tema 4: Aspectos Léxicos* cuando se comentó la utilidad del operador **new**, que precisamente es crear objetos y cuya sintaxis es:

```
new <nombreTipo>(<parametros>)
```

Este operador crea un nuevo objeto del tipo cuyo nombre se le indica y llama durante su proceso de creación al constructor del mismo apropiado según los valores que se le pasen en <parametros>, devolviendo una referencia al objeto recién creado. Hay que resaltar el hecho de que **new** no devuelve el propio objeto creado, sino una referencia a la dirección de memoria dinámica donde en realidad se ha creado.

El antes comentado **constructor** de un objeto no es más que un método definido en la definición de su tipo que tiene el mismo nombre que la clase a la que pertenece el objeto y no tiene valor de retorno. Como **new** siempre devuelve una referencia a la dirección de memoria donde se cree el objeto y los constructores sólo pueden usarse como operandos de **new**, no tiene sentido que un constructor devuelva objetos, por lo que no tiene sentido incluir en su definición un campo <tipoDevuelto> y el compilador considera erróneo hacerlo (aunque se indique **void**)

El constructor recibe ese nombre debido a que su código suele usarse precisamente para construir el objeto, para inicializar sus miembros. Por ejemplo, a nuestra clase de ejemplo Persona le podríamos añadir un constructor dejándola así:

```
class Persona
{
  string Nombre;  // Campo de cada objeto Persona que almacena su nombre
  int Edad;       // Campo de cada objeto Persona que almacena su edad
  string NIF;     // Campo de cada objeto Persona que almacena su NIF

  void Cumpleaños()   // Incrementa en uno la edad del objeto Persona
  {
    Edad++;
  }

  Persona (string nombre, int edad, string nif) // Constructor
  {
    Nombre = nombre;
    Edad = edad;
    NIF = nif;
  }
}
```

Como se ve en el código, el constructor toma como parámetros los valores con los que deseemos inicializar el objeto a crear. Gracias a él, podemos crear un objeto Persona de nombre José, de 22 años de edad y NIF 12344321-A así:

```
new Persona("José", 22, "12344321-A")
```

Nótese que la forma en que se pasan parámetros al constructor consiste en indicar los valores que se ha de dar a cada uno de los parámetros indicados en la definición del mismo separándolos por comas. Obviamente, si un parámetro se definió como de tipo **string** habrá que pasarle una cadena, si se definió de tipo **int** habrá que pasarle un entero y, en general, ha todo parámetro habrá que pasarle un valor de su mismo tipo (o de alguno convertible al mismo), produciéndose un error al compilar si no se hace así.

En realidad un objeto puede tener múltiples constructores, aunque para diferenciar a unos de otros es obligatorio que se diferencien en el número u orden de los parámetros que aceptan, ya que el nombre de todos ellos ha de coincidir con el nombre de la clase de la que son miembros. De ese modo, cuando creemos el objeto el compilador podrá inteligentemente determinar cuál de los constructores ha de ejecutarse en función de los valores que le pasemos al **new**.

Una vez creado un objeto lo más normal es almacenar la dirección devuelta por **new** en una variable del tipo apropiado para el objeto creado. El siguiente ejemplo -que como es lógico irá dentro de la definición de algún método- muestra cómo

crear una variable de tipo Persona llamada p y cómo almacenar en ella la dirección del objeto que devolvería la anterior aplicación del operador **new**:

```
Persona p;  // Creamos variable p
p = new Persona("Jose", 22, "12344321-A");
          // Almacenamos en p el objeto creado con new
```

A partir de este momento la variable p contendrá una referencia a un objeto de clase Persona que representará a una persona llamada José de 22 años y NIF 12344321-A. O lo que prácticamente es lo mismo y suele ser la forma comúnmente usada para decirlo: la variable p representa a una persona llamada José de 22 años y NIF 12344321-A.

Como lo más normal suele ser crear variables donde almacenar referencias a objetos que creemos, las instrucciones anteriores pueden compactarse en una sola así:

```
Persona p = new Persona("José", 22, "12344321-A");
```

De hecho, una sintaxis más general para la definición de variables es la siguiente:

```
<tipoDato> <nombreVariable> = <valorInicial>;
```

La parte = <valorInicial> de esta sintaxis es en realidad opcional, y si no se incluye la variable declarada pasará a almacenar una referencia nula (contendrá el literal **null**)

Constructor por defecto

No es obligatorio definir un constructor para cada clase, y en caso de que no definamos ninguno el compilador creará uno por nosotros sin parámetros ni instrucciones. Es decir, como si se hubiese definido de esta forma:

```
<nombreTipo>()
{
}
```

Gracias a este constructor introducido automáticamente por el compilador, si Coche es una clase en cuya definición no se ha incluido ningún constructor, siempre será posible crear uno nuevo usando el operador **new** así:

```
Coche c = new Coche();
    // Crea coche c llamando al constructor por defecto de Coche
```

Hay que tener en cuenta una cosa: el constructor por defecto es sólo incluido por el compilador si no hemos definido ningún otro constructor. Por tanto, si tenemos una clase en la que hayamos definido algún constructor con parámetros pero ninguno sin parámetros no será válido crear objetos de la misma llamando al constructor sin parámetros, pues el compilador no lo habrá definido automáticamente. Por ejemplo, con la última versión de la clase de ejemplo Persona es inválido hacer:

```
Persona p = new Persona();
    // ERROR: El único constructor de persona toma 3 parámetros
```

Referencia al objeto actual con this

Dentro del código de cualquier método de un objeto siempre es posible hacer referencia al propio objeto usando la palabra reservada **this**. Esto puede venir bien a la hora de escribir constructores de objetos debido a que permite que los nombres que demos a los parámetros del constructor puedan coincidir nombres de los campos del objeto sin que haya ningún problema. Por ejemplo, el constructor de la clase Persona escrito anteriormente se puede reescribir así usando **this**:

```
Persona (string Nombre, int Edad, string NIF)
{
  this.Nombre = Nombre;
  this.Edad = Edad;
  this.NIF = NIF;
}
```

Es decir, dentro de un método con parámetros cuyos nombres coincidan con campos, se da preferencia a los parámetros y para hacer referencia a los campos hay que prefijarlos con el `this` tal y como se muestra en el ejemplo.

El ejemplo anterior puede que no resulte muy interesante debido a que para evitar tener que usar `this` podría haberse escrito el constructor tal y como se mostró en la primera versión del mismo: dando nombres que empiecen en minúscula a los parámetros y nombres que empiecen con mayúsculas a los campos. De hecho, ese es el convenio que Microsoft recomienda usar. Sin embargo, como más adelante se verá sí que puede ser útil `this` cuando los campos a inicializar a sean privados, ya que el convenio de escritura de identificadores para campos privados recomendado por Microsoft coincide con el usado para dar identificadores a parámetros (obviamente otra solución sería dar cualquier otro nombre a los parámetros del constructor o los campos afectados, aunque así el código perdería algo legibilidad)

Un uso más frecuente de `this` en C# es el de permitir realizar llamadas a un método de un objeto desde código ubicado en métodos del mismo objeto. Es decir, en C# siempre es necesario que cuando llamemos a algún método de un objeto precedamos al operador `.` de alguna expresión que indique cuál es el objeto a cuyo método se desea llamar, y si éste método pertenece al mismo objeto que hace la llamada la única forma de conseguir indicarlo en C# es usando `this`.

Finalmente, una tercera utilidad de `this` es permitir escribir métodos que puedan devolver como objeto el propio objeto sobre el que el método es aplicado. Para ello bastaría usar una instrucción `return this;` al indicar el objeto a devolver

Herencia y métodos virtuales

Concepto de herencia

El mecanismo de **herencia** es uno de los pilares fundamentales en los que se basa la programación orientada a objetos. Es un mecanismo que permite definir nuevas clases a partir de otras ya definidas de modo que si en la definición de una clase indicamos que ésta deriva de otra, entonces la primera -a la que se le suele llamar **clase hija**- será tratada por el compilador automáticamente como si su definición incluyese la definición de la segunda -a la que se le suele llamar **clase padre** o **clase base**. Las clases que derivan de otras se definen usando la siguiente sintaxis:

```
class <nombreHija>:<nombrePadre>
{
  <miembrosHija>
}
```

A los miembros definidos en `<miembrosHijas>` se le añadirán los que hubiésemos definido en la clase padre. Por ejemplo, a partir de la clase `Persona` puede crearse una clase `Trabajador` así:

```
class Trabajador:Persona
{
  public int Sueldo;

  public Trabajador(string nombre, int edad, string nif, int sueldo)
              : base(nombre, edad, nif)
  {
    Sueldo = sueldo;
  }
}
```

Los objetos de esta clase `Trabajador` contarán con los mismos miembros que los objetos `Persona` y además incorporarán un nuevo campo llamado `Sueldo` que almacenará el dinero que cada trabajador gane. Nótese además que a la hora de escribir el constructor de esta clase ha sido necesario escribirlo con una sintaxis especial consistente en preceder la llave de apertura del cuerpo del método de una estructura de la forma:

```
: base(<parametrosBase>)
```

A esta estructura se le llama **inicializador base** y se utiliza para indicar cómo deseamos inicializar los campos heredados de la clase padre. No es más que una llamada al constructor de la misma con los parámetros adecuados, y si no se incluye el compilador consideraría por defecto que vale `:base()`, lo que sería incorrecto en este ejemplo debido a que `Persona` carece de constructor sin parámetros.

Un ejemplo que pone de manifiesto cómo funciona la herencia es el siguiente:

```
using System;

class Persona
{
  public string Nombre; // Campo de cada objeto Persona que almacena su nombre
  public int Edad;      // Campo de cada objeto Persona que almacena su edad
  public string NIF;    // Campo de cada objeto Persona que almacena su NIF

  void Cumpleaños()   // Incrementa en uno de edad del objeto Persona
  {
    Edad++;
  }

  public Persona (string nombre, int edad, string nif) // Constructor de Persona
  {
    Nombre = nombre;
    Edad = edad;
    NIF = nif;
  }
}

class Trabajador: Persona
{
  public int Sueldo; // Campo de cada objeto Trabajador que almacena cuánto gana

  Trabajador(string nombre, int edad, string nif, int sueldo):
             base(nombre, edad, nif)
  {          // Inicializamos cada Trabajador en base al constructor de Persona
    Sueldo = sueldo;
  }

  public static void Main()
  {
    Trabajador p = new Trabajador("Josan", 22, "77588260-Z", 100000);

    Console.WriteLine ("Nombre="+p.Nombre);
    Console.WriteLine ("Edad="+p.Edad);
    Console.WriteLine ("NIF="+p.NIF);
    Console.WriteLine ("Sueldo="+p.Sueldo);
  }
}
```

Nótese que ha sido necesario prefijar la definición de los miembros de Persona del palabra reservada **public**. Esto se debe a que por defecto los miembros de una tipo sólo son accesibles desde código incluido dentro de la definición de dicho tipo, e incluyendo **public** conseguimos que sean accesibles desde cualquier código, como el método Main() definido en Trabajador. **public** es lo que se denomina un **modificador de acceso**, concepto que se tratará más adelante en este mismo tema bajo el epígrafe titulado *Modificadores de acceso*.

Llamadas por defecto al constructor base

Si en la definición del constructor de alguna clase que derive de otra no incluimos inicializador base el compilador considerará que éste es **:base()** Por ello hay que estar seguros de que si no se incluye **base** en la definición de algún constructor, el tipo padre del tipo al que pertenezca disponga de constructor sin parámetros.

Es especialmente significativo reseñar el caso de que no demos la definición de ningún constructor en la clase hija, ya que en estos casos la definición del constructor que por defecto introducirá el compilador será en realidad de la forma:

```
<nombreClase>(): base()
{}
```

Es decir, este constructor siempre llama al constructor sin parámetros del padre del tipo que estemos definiendo, y si ése no dispone de alguno se producirá un error al compilar.

Métodos virtuales

Ya hemos visto que es posible definir tipos cuyos métodos se hereden de definiciones de otros tipos. Lo que ahora vamos a ver es que además es posible cambiar dicha definición en la clase hija, para lo que habría que haber precedido con la palabra reservada **virtual** la definición de dicho método en la clase padre. A este tipo de métodos se les llama **métodos virtuales**, y la sintaxis que se usa para definirlos es la siguiente:

```
virtual <tipoDevuelto> <nombreMétodo>(<parámetros>)
{
  <código>
}
```

Si en alguna clase hija quisiésemos dar una nueva definición del *<código>* del método, simplemente lo volveríamos a definir en la misma pero sustituyendo en su definición la palabra reservada **virtual** por **override**. Es decir, usaríamos esta sintaxis:

```
override <tipoDevuelto> <nombreMétodo>(<parámetros>)
{
  <nuevoCódigo>
}
```

Nótese que esta posibilidad de cambiar el código de un método en su clase hija sólo se da si en la clase padre el método fue definido como **virtual**. En caso contrario, el compilador considerará un error intentar redefinirlo.

El lenguaje C# impone la restricción de que toda redefinición de método que queramos realizar incorpore la partícula **override** para forzar a que el programador esté seguro de que verdaderamente lo que quiere hacer es cambiar el significado de un método heredado. Así se evita que por accidente defina un método del que ya exista una definición en una clase padre. Además, C# no permite definir un método como **override** y **virtual** a la vez, ya que ello tendría un significado absurdo: estaríamos dando una redefinición de un método que vamos a definir.

Por otro lado, cuando definamos un método como **override** ha de cumplirse que en alguna clase antecesora (su clase padre, su clase abuela, etc.) de la clase en la que se ha realizado la definición del mismo exista un método virtual con el mismo nombre que el redefinido. Si no, el compilador informará de error por intento de redefinición de método no existente o no virtual. Así se evita que por accidente un programador crea que está redefiniendo un método del que no exista definición previa o que redefina un método que el creador de la clase base no desee que se pueda redefinir.

Para aclarar mejor el concepto de método virtual, vamos a mostrar un ejemplo en el que cambiaremos la definición del método `Cumpleaños()` en los objetos `Persona` por una nueva versión en la que se muestre un mensaje cada vez que se ejecute, y redefiniremos dicha nueva versión para los objetos Trabajador de modo que el mensaje mostrado sea otro. El código de este ejemplo es el que se muestra a continuación:

```
using System;

class Persona
{
  public string Nombre;      // Campo de cada objeto Persona que almacena su nombre
  public int Edad;              // Campo de cada objeto Persona que almacena su edad
  public string NIF;            // Campo de cada objeto Persona que almacena su NIF

  public virtual void Cumpleaños() // Incrementa en uno de la edad del objeto Persona
  {
    Console.WriteLine("Incrementada edad de persona");
  }

  public Persona (string nombre, int edad, string nif) // Constructor de Persona
  {
    Nombre = nombre;
    Edad = edad;
    NIF = nif;
  }
}
```

```
class Trabajador: Persona
{
  public int Sueldo; // Campo de cada objeto Trabajador que almacena cuánto gana

  Trabajador(string nombre, int edad, string nif, int sueldo): base(nombre, edad, nif)
  {               // Inicializamos cada Trabajador en base al constructor de Persona
    Sueldo = sueldo;
  }

  public override Cumpleaños()
  {
    Edad++;
    Console.WriteLine("Incrementada edad de persona");
  }

  public static void Main()
  {
    Persona p = new Persona("Carlos", 22, "77588261-Z", 100000);
    Trabajador t = new Trabajador("Josan", 22, "77588260-Z", 100000);

    t.Cumpleaños();
    p.Cumpleaños();
  }
}
```

Nótese cómo se ha añadido el modificador **virtual** en la definición de Cumpleaños() en la clase Persona para habilitar la posibilidad de que dicho método puede ser redefinido en clase hijas de Persona y cómo se ha añado **override** en la redefinición del mismo dentro de la clase Trabajador para indicar que la nueva definición del método es una redefinición del heredado de la clase. La salida de este programa confirma que la implementación de **Cumpleaños()** es distinta en cada clase, pues es de la forma:

```
Incrementada edad de trabajador
Incrementada edad de persona
```

También es importante señalar que para que la redefinición sea válida ha sido necesario añadir la partícula **public** a la definición del método original, pues si no se incluyese se consideraría que el método sólo es accesible desde dentro de la clase donde se ha definido, lo que no tiene sentido en métodos virtuales ya que entonces nunca podría ser redefinido. De hecho, si se excluyese el modificador **public** el compilador informaría de un error ante este absurdo. Además, este modificador también se ha mantenido en la redefinición de Cumpleaños() porque toda redefinición de un método virtual ha de mantener los mismos modificadores de acceso que el método original para ser válida.

Clases abstractas

Una **clase abstracta** es aquella que forzosamente se ha de derivar si se desea que se puedan crear objetos de la misma o acceder a sus miembros estáticos (esto último se verá más adelante en este mismo tema) Para definir una clase abstracta se antepone **abstract** a su definición, como se muestra en el siguiente ejemplo:

```
public abstract class A
{
  public abstract void F();
}

abstract public class B: A
{
  public void G() {}
}
class C: B
{
  public override void F()
  {}
}
```

Las clases A y B del ejemplo son abstractas, y como puede verse es posible combinar en cualquier orden el modificador

abstract con modificadores de acceso.

La utilidad de las clases abstractas es que pueden contener métodos para los que no se dé directamente una implementación sino que se deje en manos de sus clases hijas darla. No es obligatorio que las clases abstractas contengan métodos de este tipo, pero sí lo es marcar como abstracta a toda la que tenga alguno. Estos métodos se definen precediendo su definición del modificador **abstract** y sustituyendo su código por un punto y coma (**;**), como se muestra en el método F() de la clase A del ejemplo (nótese que B también ha de definirse como abstracta porque tampoco implementa el método F() que hereda de A)

Obviamente, como un método abstracto no tiene código no es posible llamarlo. Hay que tener especial cuidado con esto a la hora de utilizar **this** para llamar a otros métodos de un mismo objeto, ya que llamar a los abstractos provoca un error al compilar.

Véase que todo método definido como abstracto es implícitamente virtual, pues si no sería imposible redefinirlo para darle una implementación en las clases hijas de la clase abstracta donde esté definido. Por ello es necesario incluir el modificador **override** a la hora de darle implementación y es redundante marcar un método como **abstract** y **virtual** a la vez (de hecho, hacerlo provoca un error al compilar)

Es posible marcar un método como **abstract** y **override** a la vez, lo que convertiría al método en abstracto para sus clases hijas y forzaría a que éstas lo tuviesen que reimplementar si no se quisiese que fuesen clases abstractas.

La clase primigenia: System.Object

Ahora que sabemos lo que es la herencia es el momento apropiado para explicar que en .NET todos los tipos que se definan heredan implícitamente de la clase **System.Object** predefinida en la BCL, por lo que dispondrán de todos los miembros de ésta. Por esta razón se dice que **System.Object** es la raíz de la jerarquía de objetos de .NET.

A continuación vamos a explicar cuáles son estos métodos comunes a todos los objetos:

- **public virtual bool Equals(object o)**: Se usa para comparar el objeto sobre el que se aplica con cualquier otro que se le pase como parámetro. Devuelve **true** si ambos objetos son iguales y **false** en caso contrario.

  La implementación que por defecto se ha dado a este método consiste en usar igualdad por referencia para los tipos por referencia e igualdad por valor para los tipos por valor. Es decir, si los objetos a comparar son de tipos por referencia sólo se devuelve **true** si ambos objetos apuntan a la misma referencia en memoria dinámica, y si los tipos a comparar son tipos por valor sólo se devuelve **true** si todos los bits de ambos objetos son iguales, aunque se almacenen en posiciones diferentes de memoria.

  Como se ve, el método ha sido definido como **virtual**, lo que permite que los programadores puedan redefinirlo para indicar cuándo ha de considerarse que son iguales dos objetos de tipos definidos por ellos. De hecho, muchos de los tipos incluidos en la BCL cuentan con redefiniciones de este tipo, como es el caso de **string**, quien aún siendo un tipo por referencia, sus objetos se consideran iguales si apuntan a cadenas que sean iguales carácter a carácter (aunque referencien a distintas direcciones de memoria dinámica)

  El siguiente ejemplo muestra cómo hacer una redefinición de **Equals()** de manera que aunque los objetos Persona sean de tipos por referencia, se considere que dos Personas son iguales si tienen el mismo NIF:

```
public override bool Equals(object o)
{
  if (o==null)
    return this==null;
  else
    return (o is Persona) && (this.NIF == ((Persona) o).NIF);
}
```

  Hay que tener en cuenta que es conveniente que toda redefinición del método **Equals()** que hagamos cumpla con una serie de propiedades que muchos de los métodos incluidos en las distintas clases de la BCL esperan que se cumplan. Estas propiedades son:

  - **Reflexividad**: Todo objeto ha de ser igual a sí mismo. Es decir, x.Equals(x) siempre ha de

devolver **true**.
- **Simetría**: Ha de dar igual el orden en que se haga la comparación. Es decir, `x.Equals(y)` ha de devolver lo mismo que `y.Equals(x)` .
- **Transitividad**: Si dos objetos son iguales y uno de ellos es igual a otro, entonces el primero también ha de ser igual a ese otro objeto. Es decir, si `x.Equals(y)` e `y.Equals(z)` entonces `x.Equals(z)` .
- **Consistencia:** Siempre que el método se aplique sobre los mismos objetos ha de devolver el mismo resultado.
- **Tratamiento de objetos nulos:** Si uno de los objetos comparados es nulo (**null**), sólo se ha de devolver **true** si el otro también lo es.
- Hay que recalcar que el hecho de que redefinir `Equals()` no implica que el operador de igualdad (**==**) quede también redefinido. Ello habría que hacerlo de independientemente como se indica en el *Tema 11: Redefinición de operadores*.

- **`public virtual int GetHashCode()`**: Devuelve un código de dispersión (hash) que representa de forma numérica al objeto sobre el que el método es aplicado. **GetHashCode()** suele usarse para trabajar con tablas de dispersión, y se cumple que si dos objetos son iguales sus códigos de dispersión serán iguales, mientras que si son distintos la probabilidad de que sean iguales es ínfima.

  En tanto que la búsqueda de objetos en tablas de dispersión no se realiza únicamente usando la igualdad de objetos (método `Equals()`) sino usando también la igualdad de códigos de dispersión, suele ser conveniente redefinir **`GetHashCode()`** siempre que se redefina `Equals()` De hecho, si no se hace el compilador informa de la situación con un mensaje de aviso.

- **`public virtual string ToString()`**: Devuelve una representación en forma de cadena del objeto sobre el que se el método es aplicado, lo que es muy útil para depurar aplicaciones ya que permite mostrar con facilidad el estado de los objetos.

  La implementación por defecto de este método simplemente devuelve una cadena de texto con el nombre de la clase a la que pertenece el objeto sobre el que es aplicado. Sin embargo, como lo habitual suele ser implementar `ToString()` en cada nueva clase que se defina, a continuación mostraremos un ejemplo de cómo redefinirlo en la clase `Persona` para que muestre los valores de todos los campos de los objetos `Persona`:

```
public override string ToString()
{
  string cadena = "";

  cadena += "DNI = " + this.DNI + "\n";
  cadena +="Nombre = " + this.Nombre + "\n";
  cadena +="Edad = " + this.Edad + "\n";

  return cadena;
}
```

  Es de reseñar el hecho de que en realidad los que hace el operador de concatenación de cadenas (**+**) para concatenar una cadena con un objeto cualquiera es convertirlo primero en cadena llamando a su método **`ToString()`** y luego realizar la concatenación de ambas cadenas.

  Del mismo modo, cuando a **`Console.WriteLine()`** y **`Console.Write()`** se les pasa como parámetro un objeto lo que hacen es mostrar por la salida estándar el resultado de convertirlo en cadena llamando a su método **`ToString();`** y si se les pasa como parámetros una cadena seguida de varios objetos lo muestran por la salida estándar esa cadena pero sustituyendo en ella toda subcadena de la forma {`<número>`} por el resultado de convertir en cadena el parámetro que ocupe la posición `<número>+2` en la lista de valores de llamada al método.

- **`protected object MemberWiseClone():`** Devuelve una copia **shallow copy** del objeto sobre el que se aplica. Esta copia es una copia bit a bit del mismo, por lo que el objeto resultante de la copia mantendrá las mismas referencias a otros que tuviese el objeto copiado y toda modificación que se haga a estos objetos a través de la copia afectará al objeto copiado y viceversa.

  Si lo que interesa es disponer de una copia más normal, en la que por cada objeto referenciado se crease una copia del mismo a la que referenciase el objeto clonado, entonces el programador ha de escribir su propio método

clonador pero puede servirse de `MemberwiseClone()` como base con la que copiar los campos que no sean de tipos referencia.

- **`public System.Type GetType()`**: Devuelve un objeto de clase **`System.Type`** que representa al tipo de dato del objeto sobre el que el método es aplicado. A través de los métodos ofrecidos por este objeto se puede acceder a metadatos sobre el mismo como su nombre, su clase padre, sus miembros, etc. La explicación de cómo usar los miembros de este objeto para obtener dicha información queda fuera del alcance de este documento ya que es muy larga y puede ser fácilmente consultada en la documentación que acompaña al .NET SDK.

- **`protected virtual void Finalize()`**: Contiene el código que se ejecutará siempre que vaya ha ser destruido algún objeto del tipo del que sea miembro. La implementación dada por defecto a **`Finalize()`** consiste en no hacer nada.

  Aunque es un método virtual, en C# no se permite que el programador lo redefina explícitamente dado que hacerlo es peligroso por razones que se explicarán en el *Tema 8: Métodos* (otros lenguajes de .NET podrían permitirlo).

Aparte de los métodos ya comentados que todos los objetos heredan, la clase **`System.Object`** también incluye en su definición los siguientes métodos de tipo:

- **`public static bool Equals(object objeto1, object objeto2)`**: Versión estática del método **`Equals()`** ya visto. Indica si los objetos que se le pasan como parámetros son iguales, y para compararlos lo que hace es devolver el resultado de calcular **`objeto1.Equals(objeto2)`** comprobando antes si alguno de los objetos vale **`null`** (sólo se devolvería **`true`** sólo si el otro también lo es)

  Obviamente si se da una redefinición al **`Equals()`** no estático, esta también se aplicará al estático.

- **`public static bool ReferenceEquals(object objeto1, object objeto2)`**: Indica si los dos objetos que se le pasan como parámetro se almacenan en la misma posición de memoria dinámica. A través de este método, aunque se hayan redefinido `Equals()` y el operador de igualdad (**`==`**) para un cierto tipo por referencia, se podrán seguir realizando comparaciones por referencia entre objetos de ese tipo en tanto que redefinir de Equals() no afecta a este método. Por ejemplo, dada la anterior redefinición de Equals() para objetos Persona:

```
Persona p = new Persona("José", 22, "83721654-W");
Persona q = new Persona("Antonio", 23, "83721654-W");
Console.WriteLine(p.Equals(q));
Console.WriteLine(Object.Equals(p, q));
Console.WriteLine(Object.ReferenceEquals(p, q));
Console.WriteLine(p == q);
```

  La salida que por pantalla mostrará el código anterior es:

```
True
True
False
False
```

  En los primeros casos se devuelve **`true`** porque según la redefinición de `Equals()` dos personas son iguales si tienen el mismo DNI, como pasa con los objetos p y q. Sin embargo, en los últimos casos se devuelve **`false`** porque aunque ambos objetos tienen el mismo DNI cada uno se almacena en la memoria dinámica en una posición distinta, que es lo que comparan `ReferenceEquals()` y el operador **`==`** (éste último sólo por defecto)

Polimorfismo

Concepto de polimorfismo

El **polimorfismo** es otro de los pilares fundamentales de la programación orientada a objetos. Es la capacidad de almacenar objetos de un determinado tipo en variables de tipos antecesores del primero a costa, claro está, de sólo poderse acceder a través de dicha variable a los miembros comunes a ambos tipos. Sin embargo, las versiones de los

métodos virtuales a las que se llamaría a través de esas variables no serían las definidas como miembros del tipo de dichas variables, sino las definidas en el verdadero tipo de los objetos que almacenan.

A continuación se muestra un ejemplo de cómo una variable de tipo `Persona` puede usarse para almacenar objetos de tipo `Trabajador`. En esos casos el campo `Sueldo` del objeto referenciado por la variable no será accesible, y la versión del método `Cumpleaños()` a la que se podría llamar a través de la variable de tipo `Persona` sería la definida en la clase `Trabajador`, y no la definida en `Persona`:

```
using System;

class Persona
{
  public string Nombre;  // Campo de cada objeto Persona que almacena su nombre
  public int Edad;       // Campo de cada objeto Persona que almacena su edad
  public string NIF;     // Campo de cada objeto Persona que almacena su NIF

  // Incrementa en uno la edad del objeto Persona
  public virtual void Cumpleaños()
  {
    Console.WriteLine("Incrementada edad de persona");
  }

  // Constructor de Persona
  public Persona (string nombre, int edad, string nif)
  {
    Nombre = nombre;
    Edad = edad;
    NIF = nif;
  }

}

class Trabajador: Persona
{
  int Sueldo; // Campo de cada objeto Trabajador que almacena cuánto gana

  Trabajador(string nombre, int edad, string nif, int sueldo):
             base(nombre, edad, nif)
  {          // Inicializamos cada Trabajador en base al constructor de Persona
    Sueldo = sueldo;
  }

  public override Cumpleaños()
  {
    Edad++;
    Console.WriteLine("Incrementada edad de trabajador");
  }

  public static void Main()
  {
    Persona p = new Trabajador("Josan", 22, "77588260-Z", 100000);

    p.Cumpleaños();
    // p.Sueldo++; //ERROR: Sueldo no es miembro de Persona
  }
}
```

El mensaje mostrado por pantalla al ejecutar este método confirma lo antes dicho respecto a que la versión de `Cumpleaños()` a la que se llama, ya que es:

```
Incrementada edad de trabajador
```

Métodos genéricos

El polimorfismo es muy útil ya que permite escribir métodos genéricos que puedan recibir parámetros que sean de un determinado tipo o de cualquiera de sus tipos hijos. Es más, en tanto que cómo se verá en el epígrafe siguiente, en C# todos los tipos derivan implícitamente del tipo **System.Object**, podemos escribir métodos que admitan parámetros de cualquier tipo sin más que definirlos como métodos que tomen parámetros de tipo **System.Object**. Por ejemplo:

```
public void MétodoGenérico(object o)
{
  // Código del método
}
```

Nótese que en vez de **System.Object** se ha escrito **object**, que es el nombre abreviado incluido en C# para hacer referencia de manera compacta a un tipo tan frecuentemente usado como **System.Object**.

Determinación de tipo. Operador is

Dentro de una rutina polimórifica que, como la del ejemplo anterior, admita parámetros que puedan ser de cualquier tipo, muchas veces es conveniente poder consultar en el código de la misma cuál es el tipo en concreto del parámetro que se haya pasado al método en cada llamada al mismo. Para ello C# ofrece el operador **is**, cuya forma sintaxis de uso es:

```
<expresión> is <nombreTipo>
```

Este operador devuelve **true** en caso de que el resultado de evaluar <expresión> sea del tipo cuyo nombre es <nombreTipo> y **false** en caso contrario. Gracias a ellas podemos escribir métodos genéricos que puedan determinar cuál es el tipo que tienen los parámetros que en cada llamada en concreto se les pasen. O sea, métodos como:

```
public void MétodoGenérico(object o)
{
  if  (o is int)       // Si o es de tipo int (entero)...
    // ...Código a ejecutar si el objeto o es de tipo int
  else if (o is string) // Si no, si o es de tipo string (cadena)...
    // ...Código a ejecutar si o es de tipo string
  //... Idem para otros tipos
}
```

El bloque **if...else** es una instrucción condicional que permite ejecutar un código u otro en función de si la condición indicada entre paréntesis tras el **if** es cierta (**true**) o no (**false**) Esta instrucción se explicará más detalladamente en el *Tema 16: Instrucciones*

Acceso a la clase base

Hay determinadas circunstancias en las que cuando redefinamos un determinado método nos interese poder acceder al código de la versión original. Por ejemplo, porque el código redefinido que vayamos a escribir haga lo mismo que el original y además algunas cosas extras. En estos casos se podría pensar que una forma de conseguir esto sería convirtiendo el objeto actual al tipo del método a redefinir y entonces llamar así a ese método, como por ejemplo en el siguiente código:

```
using System;

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

class B:A
{
  public override void F()
  {
    Console.WriteLine("Antes");
```

```
    ((A) this).F();                    // (2)
    Console.WriteLine("Después");
  }

  public static void Main()
  {
    B b = new B();
    b.F();
  }
}
```

Pues bien, si ejecutamos el código anterior veremos que la aplicación nunca termina de ejecutarse y está constantemente mostrando el mensaje `Antes por pantalla`. Esto se debe a que debido al polimorfismo se ha entrado en un bucle infinito: aunque usemos el operador de conversión para tratar el objeto como si fuese de tipo `A`, su verdadero tipo sigue siendo `B`, por lo que la versión de `F()` a la que se llamará en `(2)` es a la de `B` de nuevo, que volverá a llamarse así misma una y otra vez de manera indefinida.

Para solucionar esto, los diseñadores de C# han incluido una palabra reservada llamada **`base`** que devuelve una referencia al objeto actual semejante a **`this`** pero con la peculiaridad de que los accesos a ella son tratados como si el verdadero tipo fuese el de su clase base. Usando **`base`**, podríamos reemplazar el código de la redefinición de `F()` de ejemplo anterior por:

```
public override void F()
{
  Console.WriteLine("Antes");
  base.F();
  Console.WriteLine("Después");
}
```

Si ahora ejecutamos el programa veremos que ahora sí que la versión de `F()` en `B` llama a la versión de `F()` en A, resultando la siguiente salida por pantalla:

```
Antes
A
Después
```

A la hora de redefinir métodos abstractos hay que tener cuidado con una cosa: desde el método redefinidor no es posible usar **`base`** para hacer referencia a métodos abstractos de la clase padre, aunque sí para hacer referencia a los no abstractos. Por ejemplo:

```
abstract class A
{
  public abstract void F();
  public void G()
  {}
}

class B: A
{
  public override void F()
  {
    base.G();         // Correcto
    base.F();         // Error, base.F() es abstracto
  }
}
```

Downcasting

Dado que una variable de un determinado tipo puede estar en realidad almacenando un objeto que sea de algún tipo hijo del tipo de la variable y en ese caso a través de la variable sólo puede accederse a aquellos miembros del verdadero tipo del objeto que sean comunes con miembros del tipo de la variable que referencia al objeto, muchas veces nos va a interesar que una vez que dentro de un método genérico hayamos determinado cuál es el verdadero tipo de un objeto

(por ejemplo, con el operador **is**) podamos tratarlo como tal. En estos casos lo que hay es que hacer una conversión del tipo padre al verdadero tipo del objeto, y a esto se le llama **downcasting**

Para realizar un downcasting una primera posibilidad es indicar preceder la expresión a convertir del tipo en el que se la desea convertir indicado entre paréntesis. Es decir, siguiendo la siguiente sintaxis:

```
(<tipoDestino>) <expresiónAConvertir>
```

El resultado de este tipo de expresión es el objeto resultante de convertir el resultado de <expresiónAConvertir> a <tipoDestino>. En caso de que la conversión no se pudiese realizar se lanzaría una excepción del tipo predefinido **System.InvalidCastException**

Otra forma de realizar el downcasting es usando el operador **as**, que se usa así:

```
<expresiónAConvertir> as <tipoDestino>
```

La principal diferencia de este operador con el anterior es que si ahora la conversión no se pudiese realizar se devolvería **null** en lugar de lanzarse una excepción. La otra diferencia es que **as** sólo es aplicable a tipos referencia y sólo a conversiones entre tipos de una misma jerarquía (de padres a hijos o viceversa)

Los errores al realizar conversiones de este tipo en métodos genéricos se producen cuando el valor pasado a la variable genérica no es ni del tipo indicado en <tipoDestino> ni existe ninguna definición de cómo realizar la conversión a ese tipo (cómo definirla se verá en el *Tema 11: Redefinición de operadores*).

Clases y métodos sellados

Una **clase sellada** es una clase que no puede tener clases hijas, y para definirla basta anteponer el modificador **sealed** a la definición de una clase normal. Por ejemplo:

```
sealed class ClaseSellada
{
}
```

Una utilidad de definir una clase como sellada es que permite que las llamadas a sus métodos virtuales heredados se realicen tan eficientemente como si fuesen no virtuales, pues al no poder existir clases hijas que los redefinan no puede haber polimorfismo y no hay que determinar cuál es la versión correcta del método a la que se ha de llamar. Nótese que se ha dicho métodos virtuales heredados, pues lo que no se permite es definir miembros virtuales dentrode este tipo de clases, ya que al no poderse heredarse de ellas es algo sin sentido en tanto que nunca podráin redefinirse.

Ahora bien, hay que tener en cuenta que sellar reduce enormemente su capacidad de reutilización, y eso es algo que el aumento de eficiencia obtenido en las llamadas a sus métodos virtuales no suele compensar. En realidad la principal causa de la inclusión de estas clases en C# es que permiten asegurar que ciertas clases críticas nunca podrán tener clases hijas. Por ejemplo, para simplificar el funcionamiento del CLR y los compiladores se ha optado porque todos los tipos de datos básicos excepto **System.Object** estén sellados, pues así las operaciones con ellos siempre se realizarán de la misma forma al no influirles el polimorfismo.

Téngase en cuenta que es absurdo definir simultáneamente una clase como **abstract** y **sealed**, pues nunca podría accederse a la misma al no poderse crear clases hijas suyas que definan sus métodos abstractos. Por esta razón, el compilador considera erróneo definir una clase con ambos modificadores a la vez.

Aparte de para sellar clases, también se puede usar **sealed** como modificador en la redefinición de un método para conseguir que la nueva versión del mismo que se defina deje de ser virtual y se le puedan aplicar las optimizaciones arriba comentadas. Un ejemplo de esto es el siguiente:

```
class A
{
  public abstract F();
}

class B:A
{
  public sealed override F() // F() deja de ser redefinible
```

```
  {}
}
```

Ocultación de miembros

Hay ocasiones en las que puede resultar interesante usar la herencia únicamente como mecanismo de reutilización de código pero no necesariamente para reutilizar miembros. Es decir, puede que interese heredar de una clase sin que ello implique que su clase hija herede sus miembros tal cual sino con ligeras modificaciones.

Esto puede muy útil al usar la herencia para definir versiones especializadas de clases de uso genérico. Por ejemplo, los objetos de la clase **`System.Collections.ArrayList`** incluida en la BCL pueden almacenar cualquier número de objetos **`System.Object`**, que al ser la clase primigenia ello significa que pueden almacenar objetos de cualquier tipo. Sin embargo, al recuperarlos de este almacén genérico se tiene el problema de que los métodos que para ello se ofrecen devuelven objetos **`System.Object`**, lo que implicará que muchas veces haya luego que reconvertirlos a su tipo original mediante downcasting para poder así usar sus métodos específicos. En su lugar, si sólo se va a usar un **`ArrayList`** para almacenar objetos de un cierto tipo puede resultar más cómodo usar un objeto de alguna clase derivada de **`ArrayList`** cuyo método extractor de objetos oculte al heredado de **`ArrayList`** y devuelva directamente objetos de ese tipo.

Para ver más claramente cómo hacer la ocultación, vamos a tomar el siguiente ejemplo donde se deriva de una clase con un método `void F()` pero se desea que en la clase hija el método que se tenga sea de la forma `int F()`:

```
class Padre
{
  public void F()
  {}
}

class Hija:Padre
{
  public int  F()
  {return 1;}
}
```

Como en C# no se admite que en una misma clase hayan dos métodos que sólo se diferencien en sus valores de retorno, puede pensarse que el código anterior producirá un error de compilación. Sin embargo, esto no es así sino que el compilador lo que hará será quedarse únicamente con la versión definida en la clase hija y desechar la heredada de la clase padre. A esto se le conoce como **ocultación de miembro** ya que hace desparacer en la clase hija el miembro heredado, y cuando al compilar se detecte se generará el siguiente de aviso (se supone que clases.cs almacena el código anteiror):

```
clases.cs(9,15): warning CS0108: The keyword new is required on
            'Hija.F()' because it hides inherited member 'Padre.F()';
```

Como generalmente cuando se hereda interesa que la clase hija comparta los mismos miembros que la clase padre (y si acaso que añada miembros extra), el compilador emite el aviso anterior para indicar que no se está haciendo lo habitual. Si queremos evitarlo hemos de preceder la definición del método ocultador de la palabra reservada **`new`** para así indicar explíctamente que lo que queremos hacer es ocultar el`F()` heredado:

```
class Padre
{
  public void F()
  {}
}

class Hija:Padre
{
  new public int F()
  {return 1;}
}
```

En realidad la ocultación de miembros no implica los miembros ocultados tengan que ser métodos, sino que también pueden ser campos o cualquiera de los demás tipos de miembro que en temas posteriores se verán. Por ejemplo, puede

que se desee que un campo X de tipo **int** esté disponible en la clase hija como si fuese de tipo **string**.

Tampoco implica que los miembros métodos ocultados tengan que diferenciarse de los métodos ocultadores en su tipo de retorno, sino que pueden tener exáctamente su mismo tipo de retorno, parámetros y nombre. Hacer esto puede dar lugar a errores muy sutiles como el incluido en la siguiente variante de la clase Trabajador donde en vez de redefinirse Cumpleaños() lo que se hace es ocultarlo al olvidar incluir el **override**:

```
using System;

class Persona
{
  public string Nombre;  // Campo de cada objeto Persona que almacena su nombre
  public int Edad;       // Campo de cada objeto Persona que almacena su edad
  public string NIF;     // Campo de cada objeto Persona que almacena su NIF

  // Incrementa en uno la edad del objeto Persona
  public virtual void Cumpleaños()
  {
    Console.WriteLine("Incrementada edad de persona");
  }

  // Constructor de Persona
  public Persona (string nombre, int edad, string nif)
  {
    Nombre = nombre;
    Edad = edad;
    NIF = nif;
  }

}

class Trabajador: Persona
{
  int Sueldo; // Campo de cada objeto Trabajador que almacena cuánto gana

  Trabajador(string nombre, int edad, string nif, int sueldo):
             base(nombre, edad, nif)
  {          // Inicializamos cada Trabajador en base al constructor de Persona
    Sueldo = sueldo;
  }

  public Cumpleaños()
  {
    Edad++;
    Console.WriteLine("Incrementada edad de trabajador");
  }

  public static void Main()
  {
    Persona p = new Trabajador("Josan", 22, "77588260-Z", 100000);

    p.Cumpleaños();
    // p.Sueldo++; //ERROR: Sueldo no es miembro de Persona
  }
}
```

Al no incluirse **override** se ha perdido la capacidad de polimorifsmo, y ello puede verse en que la salida que ahora mostrara por pantalla el código:

```
Incrementada edad de persona
```

Errores de este tipo son muy sutiles y podrían ser difíciles de detectar. Sin embargo, en C# es fácil hacerlo gracias a que el compilador emitirá el mensaje de aviso ya visto por haber hecho la ocultación sin **new**. Cuando el programador lo vea

podrá añadir **new** para suprimirlo si realmente lo que quería hacer era ocultar, pero si esa no era su intención así sabrá que tiene que corregir el código (por ejemplo, añadiendo el **override** olvidado)

Como su propio nombre indica, cuando se redefine un método se cambia su definición original y por ello las llamadas al mismo ejecutaran dicha versión aunque se hagan a través de variables de la clase padre que almacenen objetos de la clase hija donde se redefinió. Sin embargo, cuando se oculta un método no se cambia su definición en la clase padre sino sólo en la clase hija, por lo que las llamadas al mismo realizadas a través de variables de la clase padre ejecutarán la versión de dicha clase padre y las realizadas mediante variables de la clase hija ejecutarán la versión de la clase hija.

En realidad el polimorfismo y la ocultación no son conceptos totalmente antagónicos, y aunque no es válido definir métodos que simultánemente cuenten con los modificadores **override** y **new** ya que un método ocultador es como si fuese la primera versión que se hace del mismo (luego no puede redefinirse algo no definido), sí que es posible combinar **new** y **virtual** para definir métodos ocultadores redefinibles. Por ejemplo:

```
using System;

class A
{
  public virtual void F() { Console.WriteLine("A.F"); }
}
class B: A
{
  public override void F() { Console.WriteLine("B.F"); }
}
class C: B
{
  new public virtual void F() { Console.WriteLine("C.F"); }
}
class D: C
{
  public override void F() { Console.WriteLine("D.F"); }
}

class Ocultación
{
  public static void Main()
  {
    A a = new D();
    B b = new D();
    C c = new D();
    D d = new D();

    a.F();
    b.F();
    c.F();
    d.F();
  }
}
```

La salida por pantalla de este programa es:

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

Aunque el verdadero tipo de los objetos a cuyo método se llama en `Main()` es `D`, en las dos primeras llamadas se llama al `F()` de `B`. Esto se debe a que la redefinición dada en `B` cambia la versión de `F()` en `A` por la suya propia, pero la ocultación dada en `C` hace que para la redefinición que posteriormente se da en `D` se considere que la versión original de `F()` es la dada en `C` y ello provoca que no modifique la versiones de dicho método dadas en `A` y `B` (que, por la redefinición dada en `B`, en ambos casos son la versión de `B`)

Un truco nemotécnico que puede ser útil para determinar a qué versión del método se llamará en casos complejos como el anterior consiste en considerar que el mecanismo de polimorfismo funciona como si buscase el verdadero tipo del objeto a cuyo método se llama descendiendo en la jerarquía de tipos desde el tipo de la variable sobre la que se aplica el método y de manera que si durante dicho recorrido se llega a alguna versión del método con **new** se para la búsqueda y se queda con la versión del mismo incluida en el tipo recorrido justo antes del que tenía el método ocultador.

Hay que tener en cuenta que el grado de ocultación que proporcione **new** depende del nivel de accesibilidad del método ocultador, de modo que si es privado sólo ocultará dentro de la clase donde esté definido. Por ejemplo, dado:

```
using System;

class A
{
  public virtual void F()       // F() es un método redefinible
  {
    Console.WriteLine("F() de A");
  }
}

class B: A
{
  new private void F() {}  // Oculta la versión de F() de A sólo dentro de B
}

class C: B
{
  public override void F() // Válido, pues aquí sólo se ve el F() de A
  {
    base.F();
    Console.WriteLine("F() de B");
  }

  public static void Main()
  {
    C obj = new C();
    obj.F();
  }
}
```

La salida de este programa por pantalla será:

```
F() de A
F() de B
```

Pese a todo lo comentado, hay que resaltar que la principal utilidad de poder indicar explícitamente si se desea redefinir u ocultar cada miembro es que facilita enormemente la resolución de problemas de **versionado de tipos** que puedan surgir si al derivar una nueva clase de otra y añadirle miembros adicionales, posteriormente se la desea actualizar con una nueva versión de su clase padre pero ésta contiene miembros que entran en conflictos con los añadidos previamente a la clase hija cuando aún no existían en la clase padre. En lenguajes como Java donde todos los miembros son implícitamente virtuales estos da lugar a problemas muy graves debidos sobre todo a:

- Que por sus nombres los nuevos miembros de la clase padre entre en conflictos con los añadidos a la clase hija cuando no exisitían. Por ejemplo, si la versión inicial de de la clase padre no contiene ningún método de nombre `F()`, a la clase hija se le añade `void F()` y luego en la nueva versión de la clase padre se incorporado `int F()`, se producirá un error por tenerse en la clase hija dos métodos `F()`

  En Java para resolver este problema una posibilidad sería pedir al creador de la clase padre que cambiase el nombre o parámetros de su método, lo cual no es siempre posible ni conveniente en tanto que ello podría trasladar el problema a que hubiesen derivado de dicha clase antes de volverla a modificar. Otra posibilidad sería modificar el nombre o parámetros del método en la clase hija, lo que nuevamente puede llevar a incompatibilidades si también se hubiese derivado de dicha clase hija.

- Que los nuevos miembros tengan los mismos nombres y tipos de parámetros que los incluidos en las

clases hijas y sea obligatorio que toda redefinición que se haga de ellos siga un cierto esquema.

Esto es muy problemático en lenguajes como Java donde toda definición de método con igual nombre y parámetros que alguno de su clase padre es considerado implícitamente redefinición de éste, ya que difícilmente en una clase hija escrita con anterioridad a la nueva versión de la clase padre se habrá seguido el esquema necesario. Por ello, para resolverlo habrá que actualizar la clase hija para que lo siga y de tal manera que los cambios que se le hagan no afecten a sus subclases, lo que ello puede ser más o menos difícil según las características del esquema a seguir.

Otra posibilidad sería sellar el método en la clase hija, pero ello recorta la capacidad de reutilización de dicha clase y sólo tiene sentido si no fue redefinido en ninguna subclase suya.

En C# todos estos problemas son de fácil solución ya que pueden resolverse con sólo ocultar los nuevos miembros en la clase hija y seguir trabajando como si no existiesen.

Miembros de tipo

En realidad, dentro la definición de un tipo de dato no tiene porqué incluirse sólo definiciones de miembros comunes a todos sus objetos, sino también pueden definirse miembros ligados al tipo como tal y no a los objetos del mismo. Para ello basta preceder la definición de ese miembro de la palabra reservada **static**, como muestra este ejemplo:

```
class A
{
  int x;
  static int y;
}
```

Los objetos de clase A sólo van a disponer del campo x, mientras que el campo y va a pertenecer a la clase A. Por esta razón se dice que los miembros con modificador **static** son **miembros de tipo** y que los no lo tienen son **miembros de objeto**.

Para acceder a un miembro de clase ya no es válida la sintaxis hasta ahora vista de <objeto>.<miembro>, pues al no estar estos miembros ligados a ningún objeto no podría ponerse nada en el campo <objeto>. La sintaxis a usar para acceder a estos miembros será <nombreClase>.<miembro>, como muestra ejemplo donde se asigna el valor 2 al miembro y de la clase A definida más arriba:

```
A.y = 2;
```

Nótese que la inclusión de miembros de clase rompe con la afirmación indicada al principio del tema en la que se decía que C# es un lenguaje orientado a objetos puro en el que todo con lo que se trabaja son objetos, ya que a los miembros de tipo no se les accede a través de objetos sino nombres de tipos.

Es importante matizar que si definimos una función como **static**, entonces el código de la misma sólo podrá acceder implícitamente (sin sintaxis <objeto>.<miembro>) a otros miembros **static** del tipo de dato al que pertenezca. O sea, no se podrá acceder a ni a los miembros de objeto del tipo en que esté definido ni se podrá usar **this** ya que el método no está asociado a ningún objeto. O sea, este código sería inválido:

```
int x;
static void Incrementa()
{
  x++; //ERROR: x es  miembro de objeto e Incrementa() lo es de clase.
}
```

También hay que señalar que los métodos estáticos no entran dentro del mecanismo de redefiniciones descrito en este mismo tema. Dicho mecanismo sólo es aplicable a métodos de objetos, que son de quienes puede declararse variables y por tanto puede actuar el polimorifsmo. Por ello, incluir los modificadores **virtual**, **override** o **abstract** al definir un método **static** es considerado erróneo por el compilador; aunque ello no significan que los miembros **static** no se hereden, sino que sólo tiene sentido redefinirlos.

Encapsulación

Ya hemos visto que la herencia y el polimorfismo eran dos de los pilares fundamentales en los que es apoya la programación orientada a objetos. Pues bien, el tercero y último es la **encapsulación**, que es un mecanismo que permite

a los diseñadores de tipos de datos determinar qué miembros de los tipos creen pueden ser utilizados por otros programadores y cuáles no. Las principales ventajas que ello aporta son:

- Se facilita a los programadores que vaya a usar el tipo de dato (programadores clientes) el aprendizaje de cómo trabajar con él, pues se le pueden ocultar todos los detalles relativos a su implementación interna y sólo dejarle visibles aquellos que puedan usar con seguridad. Además, así se les evita que cometan errores por manipular inadecuadamente miembros que no deberían tocar.
- Se facilita al creador del tipo la posterior modificación del mismo, pues si los programadores clientes no pueden acceder a los miembros no visibles, sus aplicaciones no se verán afectadas si éstos cambian o se eliminan. Gracias a esto es posible crear inicialmente tipos de datos con un diseño sencillo aunque poco eficiente, y si posteriormente es necesariomodificarlos para aumentar su eficiencia, ello puede hacerse sin afectar al código escrito en base a la no mejorada de tipo.

La encapsulación se consigue añadiendo **modificadores de acceso** en las definiciones de miembros y tipos de datos. Estos modificadores son partículas que se les colocan delante para indicar desde qué códigos puede accederse a ellos, entendiéndose por acceder el hecho de usar su nombre para cualquier cosa que no sea definirlo, como llamarlo si es una función, leer o escribir su valor si es un campo, crear objetos o heredar de él si es una clase, etc.

Por defecto se considera que los miembros de un tipo de dato sólo son accesibles desde código situado dentro de la definición del mismo, aunque esto puede cambiarse precediendolos de uno los siguientes modificadores (aunque algunos de ellos ya se han explicado a lo largo del tema, aquí se recogen todos de manera detallada) al definirlos:

public: Puede ser accedido desde cualquier código.

protected: Desde una clase sólo puede accederse a miembros **protected** de objetos de esa misma clase o de subclases suyas. Así, en el siguiente código las instrucciones comentadas con `// Error` no son válidas por lo escrito junto a ellas:

```
public class A
{
  protected int x;

  static void F(A a, B b, C c)
  {
    a.x = 1;  // Ok
    b.x = 1;  // Ok
    c.x = 1;  // OK
  }
}

public class B: A
{
  static void F(A a, B b, C c)
  {
    //a.x = 1;  // Error, ha de accederse a traves de objetos tipo B o C
    b.x = 1;  // Ok
    c.x = 1;  // Ok
  }
}

public class C: B
{
  static void F(A a, B b, C c)
  {
    //a.x = 1; // Error, ha de accederse a traves de objetos tipo C
    //b.x = 1; // Error, ha de accederse a traves de objetos tipo C
    c.x = 1; // Ok
  }
}
```

Obviamente siempre que se herede de una clase se tendrá total acceso en la clase hija -e implícitamente sin necesidad de usar la sintaxis `<objeto>.<miembro>`- a los miembros que ésta herede de su clase padre, como muestra el siguiente ejemplo:

```
using System;

class A
{
  protected int x=5;
}

class B:A
{
  B()
  {
    Console.WriteLine("Heredado x={0} de clase A", x);
  }

  public static void Main()
  {
    new B();
  }
}
```

Como es de esperar, la salida por pantalla del programa de ejemplo será:

```
Heredado x=5 de clase A
```

private: Sólo puede ser accedido desde el código de la clase a la que pertenece. Es lo considerado por defecto.

internal: Sólo puede ser accedido desde código perteneciente al ensamblado en que se ha definido.

protected internal: Sólo puede ser accedido desde código perteneciente al ensamblado en que se ha definido o desde clases que deriven de la clase donde se ha definido.

Es importante recordar que toda redefinición de un método virtual o abstracto ha de realizarse manteniendo los mismos modificadores que tuviese el método original. Es decir, no podemos redefinir un método protegido cambiando su accesibilidad por pública, pues si el creador de la clase base lo definió así por algo sería.

Respecto a los tipos de datos, por defecto se considera que son accesibles sólo desde el mismo ensamblado en que ha sido definidos, aunque también es posible modificar esta consideración anteponiendo uno de los siguientes modificadores a su definición:

public: Es posible acceder a la clase desde cualquier ensamblado.

internal: Sólo es posible acceder a la clase desde el ensamblado donde se declaró. Es lo considerado por defecto.

También pueden definirse tipos dentro de otros (**tipos internos**) En ese caso serán considerados miembros del tipo contenedor dentro de la que se hayan definido, por lo que les serán aplicables todos los modificadores válidos para miembros y por defecto se considerará que, como con cualquier miembro, son privados. Para acceder a estos tipos desde código externo a su tipo contenedor (ya sea para heredar de ellos, crear objetos suyos o acceder a sus miembros estáticos), además de necesitarse los permisos de acceso necesarios según el modificador de accesibilidad al definirlos, hay que usar la notación <nombreTipoContendera>.<nombreTipoInterno> como muestra este ejemplo:

```
class A  // No lleva modificador, luego se considera que es internal
{
  public class AInterna {}  // Si ahora no se pusiese public se consideraría private
}

class B:A.AInterna   // B deriva de la clase interna AInterna definida dentro de A. Es
{}                   // válido porque A.AInterna es pública
```

Principio Página

El lenguaje de programación C#

En esta página:

Tema 6: Espacios de nombres

Concepto de espacio de nombres

Del mismo modo que los ficheros se organizan en directorios, los tipos de datos se organizan en **espacios de nombres**.

Por un lado estos espacios permiten tener más organizados los tipos de datos, lo que facilita su localización. De hecho, esta es la forma en que se encuentra organizada la BCL, de modo que todas las clases más comúnmente usadas en cualquier aplicación pertenecen al espacio de nombres llamado `System`, las de acceso a bases de datos en `System.Data`, las de realización de operaciones de entrada/salida en `System.IO`, etc

Por otro lado, los espacios de nombres también permiten poder usar en un mismo programa varias clases con igual nombre si pertenecen a espacios diferentes. La idea es que cada fabricante defina sus tipos dentro de un espacio de nombres propio para que así no hayan conflictos si varios fabricantes definen clases con el mismo nombre y se quieren usar a la vez en un mismo programa. Obviamente para que esto funcione no han de coincidir los nombres los espacios de cada fabricante, y una forma de conseguirlo es dándoles el nombre de la empresa fabricante, o su nombre de dominio en Internet, etc.

Definición de espacios de nombres

Para definir un espacio de nombres se utiliza la siguiente sintaxis:

```
namespace <nombreEspacio>
{
  <tipos>
}
```

Los tipos que se definan en `<tipos>` pasarán a considerase pertenecientes al espacio de nombres llamado `<nombreEspacio>`. Como veremos más adelante, aparte de clases esto tipos pueden ser también interfaces, estructuras, tipos enumerados y delegados. A continuación se muestra un ejemplo en el que definimos una clase de nombre `ClaseEjemplo` perteneciente a un espacio de nombres llamado `EspacioEjemplo`:

```
namespace EspacioEjemplo
{
  class ClaseEjemplo
  {}
}
```

El verdadero nombre de una clase, al que se denomina **nombre completamente calificado**, es el nombre que le demos al declararla prefijado por la concatenación de todos los espacios de nombres a los que pertenece ordenados del más externo al más interno y seguido cada uno de ellos por un punto (carácter **.**) Por ejemplo, el verdadero nombre de la clase `ClaseEjemplo` antes definida es `EspacioEjemplo.ClaseEjemplo`. Si no definimos una clase dentro de una definición de espacio de nombres -como se ha hecho en los ejemplos de temas previos- se considera que ésta pertenece al denominado **espacio de nombres global** y su nombre completamente calificado coincidirá con el nombre que le demos al definirla..

Aparte de definiciones de tipo, también es posible incluir como miembros de un espacio de nombres a otros espacios de nombres. Es decir, como se muestra el siguiente ejemplo es posible anidar espacios de nombres:

```
namespace EspacioEjemplo
{
  namespace EspacioEjemplo2
  {
    class ClaseEjemplo
    {}
  }
}
```

Ahora `ClaseEjemplo` tendrá `EspacioEjemplo.EspacioEjemplo2.ClaseEjemplo` como nombre completamente calificado. En realidad es posible compactar las definiciones de espacios de nombres anidados usando esta sintaxis de calificación completa para dar el nombre del espacio de nombres a definir. Es decir, el último ejemplo es equivalente a:

```
namespace EspacioEjemplo.EspacioEjemplo2
{
  class ClaseEjemplo
```

```
  {}
}
```

En ambos casos lo que se ha definido es una clase llamada `ClaseEjemplo` perteneciente al espacio de nombres llamado `EspacioEjemplo2` que, a su vez, pertenece al espacio de nombres llamado `EspacioEjemplo`.

Importación de espacios de nombres

Sentencia using

En principio, si desde código perteneciente a una clase definida en un cierto espacio de nombres se desea hacer referencia a tipos definidos en otros espacios de nombres, se ha de referir a los mismos usando su nombre completamente calificado. Por ejemplo:

```
namespace EspacioEjemplo.EspacioEjemplo2
{
  class ClaseEjemplo
  {}
}

class Principal  // Pertenece al espacio de nombres global
{
  public static void Main ()
  {
    EspacioEjemplo.EspacioEjemplo2.ClaseEjemplo c  = new
              EspacioEjemplo.EspacioEjemplo2.ClaseEjemplo();
  }
}
```

Como puede resultar muy pesado tener que escribir nombres tan largos en cada referencia a tipos así definidos, en C# se ha incluido un mecanismo de importación de espacios de nombres que usa la siguiente sintaxis:

```
using <espacioNombres>;
```

Este tipo de sentencias siempre ha de aparecer dentro de una definición de espacio de nombres antes que cualquier definición de miembros de la misma y permiten indicar cuáles serán los espacios de nombres que se usarán implícitamente dentro de ese espacio de nombres. A los miembros de los espacios de nombres así importados se les podrá hacer referencia sin tener que usar calificación completa, como muestra la siguiente versión del último ejemplo:

```
using EspacioEjemplo.EspacioEjemplo2;

namespace EspacioEjemplo.EspacioEjemplo2
{
  class ClaseEjemplo
  {}
}
```

```
// (1)
class Principal    // Pertenece al espacio de nombres global
{
  public static void ()
  {
    // EspacioEjemplo.EspacioEjemplo2. está implícito
    ClaseEjemplo c  = new ClaseEjemplo();
  }
}
```

Nótese que la sentencia **using** no podría haberse incluido en la zona marcada en el código como (1) el código porque entonces se violaría la regla de que todo **using** ha aparecer en un espacio de nombres antes que cualquier definición de miembro, ya que la definición del espacio de nombres EspacioEjemplo.EspacioEjemplo2 es un miembro del espacio de nombres global. Sin embargo, el siguiente código si que sería válido:

```
namespace EspacioEjemplo.EspacioEjemplo2
{
  class ClaseEjemplo
  {}
}

namespace Principal
{
  using EspacioEjemplo.EspacioEjemplo2;

  class Principal    // Pertenece al espacio de nombres global
  {
    public static void Main()
    {
      ClaseEjemplo c  = new ClaseEjemplo();
    }
  }
}
```

En este caso el **using** aparece antes que cualquier otra definición de tipos dentro del espacio de nombres en que se incluye (Principal) Sin embargo, ahora la importación hecha con el **using** sólo será válida dentro de código incluido en ese mismo espacio de nombres, mientras que en el caso anterior era válida en todo el fichero al estar incluida en el espacio de nombres global.

Si una sentencia **using** importa miembros de igual nombre que miembros definidos en el espacio de nombres donde se incluye, el **using** no se produce error alguno pero se da preferencia a los miembros no importados. Un ejemplo:

```
namespace N1.N2
{
  class A {}
  class B {}
}

namespace N3
```

```
{
  using N1.N2;

  class A {}
  class C: A {}
}
```

En este ejemplo `C` deriva de `N3.A` en vez de `N1.N2.A`. Si queremos que ocurra lo contrario tendremos que referenciar a `N1.N2.A` por su nombre completo al definir `C` o, como se explica a continuación, usar un **alias**.

Especificación de alias

Aún en el caso de que usemos espacios de nombres distintos para diferenciar clases con igual nombre pero procedentes de distintos fabricantes, podrían darse conflictos sin usamos sentencias **`using`** para importar los espacios de nombres de dichos fabricantes ya que entonces al hacerse referencia a una de las clases comunes con tan solo su nombre simple el compilador no podrá determinar a cual de ellas en concreto nos referimos.

Por ejemplo, si tenemos una clase de nombre completamente calificado `A.Clase`, otra de nombre `B.Clase`, y hacemos:

```
using A;
using B;

class EjemploConflicto: Clase {}
```

¿Cómo sabrá el compilador si lo que queremos es derivar de A.Clase o de B.Clase? En realidad el compilador no puede determinarlo y producirá un error informando de que hay una referencia ambigua a Clase.

Para resolver ambigüedades de este tipo podría hacerse referencia a los tipos en conflicto usando siempre sus nombres completamente calificados, pero ello puede llegar a ser muy fatigoso sobre todo si sus nombres son muy largos. Para solucionar los conflictos de nombres sin tener que escribir tanto se ha incluido en C# la posibilidad de definir **alias** para cualquier tipo de dato, que son sinónimos para los mismos que se definen usando la siguiente sintaxis:

```
using <alias> = <nombreCompletoTipo>;
```

Como cualquier otro **using**, las definiciones de alias sólo pueden incluirse al principio de las definiciones de espacios de nombres y sólo tienen validez dentro de las mismas.

Definiendo alias distintos para los tipos en conflictos se resuelven los problemas de ambigüedades. Por ejemplo, el problema del ejemplo anterior se podría resolver así:

```
using A;
using B;
using ClaseA = A.Clase;

class EjemploConflicto: ClaseA {} // Heredamos de A.Clase
```

Los alias no tienen porqué ser sólo referentes a tipos, sino que también es posible escribir alias de espacios de nombres como muestra el siguiente ejemplo:

```
namespace N1.N2
{
  class A {}
}
namespace N3
{
  using R1 = N1;
  using R2 = N1.N2;
  class B
  {
    N1.N2.A a;  // Campo de nombre completamente calificado N1.N2.A
    R1.N2.A b;  // Campo de nombre completamente calificado N1.N2.A
    R2.A c;       // Campo de nombre completamente calificado N1.N2.A
  }
}
```

Al definir alias hay que tener cuidado con no definir en un mismo espacio de nombres varios con igual nombre o cuyos nombres coincidan con los de miembros de dicho espacio de nombres. También hay que tener en cuenta que no se pueden definir unos alias en función de otro, por lo que códigos como el siguiente son incorrectos:

```
namespace N1.N2 {}
namespace N3
{
  using R1 = N1;
  using R2 = N1.N2;
  using R3 = R1.N2; // ERROR: No se puede definir R3 en función de R1
}
```

Espacio de nombres distribuidos

Si hacemos varias definiciones de un espacio de nombres en un mismo fichero o en diferentes y se compilan todas juntas, el compilador las fusionará en una sola definición cuyos miembros serán la concatenación de los miembros definidos en cada una de las definiciones realizadas. Por ejemplo:

```
namespace A   // (1)
{
  class B1 {}
}

namespace A    // (2)
{
  class B2 {}
}
```

Hacer una definición como la anterior es tratada por el compilador exactamente igual que si se hubiese hecho:

```
namespace A

{
  class B1 {}
  class B2 {}
}
```

Lo mismo ocurriría si las definiciones marcadas como (1) y (2) se hubiesen hecho en ficheros separados que se compilasen conjuntamente.

Hay que tener en cuenta que las sentencias **using**, ya sean de importación de espacios de nombres o de definición de alias, no son consideradas miembros de los espacios de nombres y por tanto no participan en sus fusiones. Así, el siguiente código es inválido:

```
namespace A
{
  class ClaseA {}
}

namespace B
{
  using A;
}

namespace B
{
  // using A;
  class Principal: ClaseA {}
}
```

Este código no es válido debido a que aunque se importa el espacio de nombres `A` al principio de una definición del espacio de nombres donde se ha definido `Principal`, no se importa en la definición en donde se deriva `Principal` de `A.ClaseA`. Para que todo funcionase a la perfección habría que descomentar la línea comentada en el ejemplo.

Principio Página

El lenguaje de programación C#

En esta página:

Tema 7: Variables y tipos de datos

Definición de variables

Una **variable** puede verse simplemente como un almacén de objetos de un determinado tipo al que se le da un cierto nombre. Por tanto, para definir una variable sólo hay que decir cuál será el nombre que se le dará y cuál será el tipo de datos que podrá almacenar, lo que se hace con la la siguiente sintaxis:

```
<tipoVariable> <nombreVariable>;
```

Una variable puede ser definida dentro de una definición de clase, en cuyo caso se correspondería con el tipo de miembro que hasta ahora hemos denominado **campo**. También puede definirse como un **variable local** a un método, que es una variable definida dentro del código del método a la que sólo puede accederse desde dentro de dicho código. Otra posibilidad es definirla como **parámetro** de un método, que son variables que almacenan los valores de llamada al método y que, al igual que las variables locales, sólo puede ser accedidas desde código ubicado dentro del método. El siguiente ejemplo muestra como definir variables de todos estos casos:

```
class A
{
  int x, z;
  int y;
```

```
  void F(string a, string b)
  {
    Persona p;
  }
}
```

En este ejemplo las variables `x`, `z` e `y` son campos de tipo **`int`**, mientras que **`p`** es una variable local de tipo `Persona` y `a` y `b` son parámetros de tipo **`string`**. Como se muestra en el ejemplo, si un método toma varios parámetros las definiciones de éstos se separan mediante comas (carácter `,`), y si queremos definir varios campos o variables locales (no válido para parámetros) de un mismo tipo podemos incluirlos en una misma definición incluyendo en `<nombreVariable>` sus nombres separados por comas.

Con la sintaxis de definición de variables anteriormente dada simplemente definimos variables pero no almacenamos ningún objeto inicial en ellas. El compilador dará un valor por defecto a los campos para los que no se indique explícitamente ningún valor según se explica en el siguiente apartado. Sin embargo, a la variables locales no les da ningún valor inicial, pero detecta cualquier intento de leerlas antes de darles valor y produce erores de compilación en esos casos.

Ya hemos visto que para crear objetos se utiliza el operador **`new`**. Por tanto, una forma de asignar un valor a la variable p del ejemplo anterior sería así:

```
Persona p;
p = new Persona("José", 22, "76543876-A");
```

Sin embargo, C# también proporciona una sintaxis más sencilla con la que podremos asignar un objeto a una variable en el mismo momento se define. Para ello se la ha de definir usando esta otra notación:

```
<tipoVariable> <nombreVariable> = <valorInicial>;
```

Asi por ejemplo, la anterior asignación de valor a la variable `p` podría reescribirse de esta otra forma más compacta:

```
Persona p  = new Persona("José", 22, "76543876-A");
```

La especificación de un valor inicial también combinarse con la definición de múltiples variables separadas por comas en una misma línea. Por ejemplo, las siguientes definiciones son válidas:

```
Persona p1 = new Persona("José", 22, "76543876-A"),
  p2 = new Persona("Juan", 21,"87654212-S");
```

Y son tratadas por el compilador de forma completamente equivalentes a haberlas declarado como:

```
Persona p1 = new Persona("José", 22, "76543876-A");
Persona p2 = new Persona("Juan", 21, "87654212-S");
```

Tipos de datos básicos

Los **tipos de datos básicos** son ciertos tipos de datos tan comúnmente utilizados en la escritura de aplicaciones que en C# se ha incluido una sintaxis especial para tratarlos. Por ejemplo, para representar números enteros de 32 bits con signo se utiliza el tipo de dato **`System.Int32`** definido en la BCL, aunque a la hora de crear un objeto a de este tipo que represente el valor 2 se usa la siguiente sintaxis:

```
System.Int32 a = 2;
```

Como se ve, no se utiliza el operador **new** para crear un objeto **System.Int32**, sino que directamente se indica el literal que representa el valor a crear, con lo que la sintaxis necesaria para crear entero de este tipo se reduce considerablemente. Es más, dado lo frecuente que es el uso de este tipo también se ha predefinido en C# el alias **int** para el mismo, por lo que la definición de variable anterior queda así de compacta:

```
int a = 2;
```

System.Int32 no es el único tipo de dato básico incluido en C#. En el espacio de nombres **System** se han incluido todos estos:

Tipo	Descripción	Bits	Rango de valores	Alias
SByte	*Bytes* con signo	8	-128 - 127	sbyte
Byte	*Bytes* sin signo	8	0 - 255	byte
Int16	Enteros cortos con signo	16	[-32.768, 32.767]	short
UInt16	Enteros cortos sin signo	16	[0, 65.535]	ushort
Int32	Enteros normales	32	[-2.147.483.648, 2.147.483.647]	int
UInt32	Enteros normales sin signo	32	[0, 4.294.967.295]	uint
Int64	Enteros largos	64	[-9.223.372.036.854.775.808, 9.223.372.036.854.775.807]	long
UInt64	Enteros largos sin signo	64	[0-18.446.744.073.709.551.615]	ulong
Single	Reales con 7 dígitos de precisión	32	$[1,5\times10^{-45}$ - $3,4\times10^{38}]$	float
Double	Reales de 15-16 dígitos de precisión	64	$[5,0\times10^{-324}$ - $1,7\times10^{308}]$	double
Decimal	Reales de 28-29 dígitos de precisión	128	$[1,0\times10^{-28}$ - $7,9\times10^{28}]$	decimal
Boolean	Valores lógicos	32	**true**, **false**	bool
Char	Caracteres Unicode	16	['\u0000', '\uFFFF']	char
String	Cadenas de caracteres	Variable	El permitido por la memoria	string
Object	Cualquier objeto	Variable	Cualquier objeto	object

Tabla 5: Tipos de datos básicos

Pese a su sintaxis especial, en C# los tipos básicos son tipos del mismo nivel que cualquier otro tipo del lenguaje. Es decir, heredan de **System.Object** y pueden ser tratados como objetos de dicha clase por cualquier método que espere un **System.Object**, lo que es muy útil para el diseño de rutinas genéricas que admitan parámetros de cualquier tipo y es una ventaja importante de C# frente a lenguajes similares como Java donde los tipos básicos no son considerados objetos.

El valor que por defecto se da a los campos de tipos básicos consiste en poner a cero todo el área de memoria que ocupen. Esto se traduce en que los campos de tipos básicos numéricos se inicializan por defecto con el valor 0, los de tipo **bool** lo hacen con **false**, los de tipo **char** con '\u0000', y los de tipo **string** y **object** con **null**.

Ahora que sabemos cuáles son los tipos básicos, es el momento de comentar cuáles son los sufijos que admiten los literales numéricos para indicar al compilador cuál es el tipo que se ha de considerar que tiene. Por ejemplo, si tenemos en una clase los métodos:

```
public static void F(int x)
```

```
public static void F(long x)
```

Ante una llamada como `F(100)`, ¿a cuál de los métodos se llamara? Pues bien, en principio se considera que el tipo de un literal entero es el correspondiente al primero de estos tipos básicos que permitan almacenarlo: **`int`**, **`uint`**, **`long`**, **`ulong`**, por lo que en el caso anterior se llamaría al primer `F()` Para llamar al otro podría añadirse el sufijo **`L`** al literal y hacer la llamada con `F(100L)` En la **Tabla 6** se resumen los posibles sufijos válidos:

Sufijo	Tipo del literal entero
ninguno	Primero de: **int, uint, long**,**ulong**
L ó **l**	Primero de: **long, ulong**
U ó **u**	Primero de: **int, uint**
UL, Ul, uL, ul, LU, Lu, lU ó **lu**	**ulong**

Tabla 6: Sufijos de literales enteros

Por su parte, en la **Tabla 7** se indican los sufijos que admiten los literales reales son:

Sufijo	Tipo del literal real
F ó **f**	**float**
ninguno,**D** ó**d**	**double**
M ó **m**	**decimal**

Tabla 7: Sufijos de literales reales

Tablas

Tablas unidimensionales

Una **tabla unidimensional** es un tipo especial de variable que es capaz de almacenar en su interior y de manera ordenada uno o varios datos de un determinado tipo. Para declarar variables de este tipo especial se usa la siguiente sintaxis:

```
<tipoDatos>[] <nombreTabla>;
```

Por ejemplo, una tabla que pueda almacenar objetos de tipo **`int`** se declara así:

```
int[] tabla;
```

Con esto la tabla creada no almacenaría ningún objeto, sino que valdría **`null`**. Si se desea que verdaderamente almacene objetos hay que indicar cuál es el número de objetos que podrá almacenar, lo que puede hacerse usando la siguiente sintaxis al declararla:

```
<tipoDatos>[] <nombreTabla> = new <tipoDatos>[<númeroDatos>];
```

Por ejemplo, una tabla que pueda almacenar 100 objetos de tipo **`int`** se declara así:

```
int[] tabla = new int[100];
```

Aunque también sería posible definir el tamaño de la tabla de forma separada a su declaración de este modo:

```
int[] tabla;
tabla = new int[100];
```

Con esta última sintaxis es posible cambiar dinámicamente el número de elementos de una variable tabla sin más que irle asignando nuevas tablas. Ello no significa que una tabla se pueda redimensionar conservando los elementos que tuviese antes del cambio de tamaño, sino que ocurre todo lo contrario: cuando a una variable tabla se le asigna una tabla de otro tamaño, sus elementos antiguos son sobreescritos por los nuevos.

Si se crea una tabla con la sintaxis hasta ahora explicada todos sus elementos tendrían el valor por defecto de su tipo de dato. Si queremos darles otros valores al declarar la tabla, hemos de indicarlos entre llaves usando esta sintaxis:

```
<tipoDatos>[] <nombreTabla> = new <tipoDatos>[] ;
```

Ha de especificarse tantos `<valores>` como número de elementos se desee que tenga la tabla, y si son más de uno se han de separar entre sí mediante comas (`,`) Nótese que ahora no es necesario indicar el número de elementos de la tabla (aunque puede hacerse) si se desea), pues el compilador puede deducirlo del número de valores especificados. Por ejemplo, para declarar una tabla de cuatro elementos de tipo **`int`** con valores 5,1,4,0 se podría hacer lo siguiente:

```
int[] tabla = new int[] ;
```

Incluso se puede compactar aún más la sintaxis declarando la tabla así:

```
int[] tabla = ;
```

También podemos crear tablas cuyo tamaño se pueda establecer dinámicamente a partir del valor de cualquier expresión que produzca un valor de tipo entero. Por ejemplo, para crear una tabla cuyo tamaño sea el valor indicado por una variable de tipo **`int`** (luego su valor será de tipo entero) se haría:

```
int i = 5;
...
int[] tablaDinámica =  new int[i];
```

A la hora de acceder a los elementos almacenados en una tabla basta indicar entre corchetes, y a continuación de la referencia a la misma, la posición que ocupe en la tabla el elemento al que acceder. Cuando se haga hay que tener en cuenta que en C# las tablas se indexan desde 0, lo que significa que el primer elemento de la tabla ocupará su posición 0, el segundo ocupará la posición 1, y así sucesivamente para el resto de elementos. Por ejemplo, aunque es más ineficiente, la tabla declarada en el último fragmento de código de ejemplo también podría haberse definido así:

```
int[] tabla = new int[4];

tabla[0] = 5;
tabla[1]++;  // Por defecto se inicializó a 0, luego
             // ahora el valor de tabla[1] pasa a ser 1
tabla[2] = tabla[0] - tabla[1];  // tabla[2] pasa a
                                 // valer 4, pues 5-4 = 1
// El contenido de la tabla será , pues
// tabla[3] se inicializó por defecto a 0.
```

Hay que tener cuidado a la hora de acceder a los elementos de una tabla ya que si se especifica una posición superior al número de elementos que pueda almacenar la tabla se producirá una excepción de tipo **`System.OutOfBoundsException`**. En el *Tema 16: Instrucciones* se explica qué son las excepciones, pero por ahora basta considerar que son objetos que informan de situaciones excepcionales (generalmente errores) producidas durante la ejecución de una aplicación. Para evitar este tipo de excepciones puede consultar el

valor del campo de sólo lectura **Length** que está asociado a toda tabla y contiene el número de elementos de la misma. Por ejemplo, para asignar un 7 al último elemento de la tabla anterior se haría:

```
tabla[tabla.Length - 1] = 7;  // Se resta 1 porque tabla.Length
                              // devuelve 4 pero el último elemento
                              // de la tabla es tabla[3]
```

Tablas dentadas

Una **tabla dentada** no es más que una tabla cuyos elementos son a su vez tablas, pudiéndose así anidar cualquier número de tablas. Para declarar tablas de este tipo se usa una sintaxis muy similar a la explicada para las tablas unidimensionales solo que ahora se indican tantos corchetes como nivel de anidación se desee. Por ejemplo, para crear una tabla de tablas de elementos de tipo **int** formada por dos elementos, uno de los cuales fuese una tabla de elementos de tipo **int** formada por los elementos de valores 1,2 y el otro fuese una tabla de elementos de tipo **int** y valores 3,4,5, se puede hacer:

```
int[][] tablaDentada = new int[2][] {new int[] , new int[] };
```

Como se indica explícitamente cuáles son los elementos de la tabla declarada no hace falta indicar el tamaño de la tabla, por lo que la declaración anterior es equivalente a:

```
int[][] tablaDentada = new int[][] {new int[] , new int[] };
```

Es más, igual que como se vió con las tablas unidimensionales también es válido hacer:

```
int[][] tablaDentada = {new int[] , new int[] };
```

Si no quisiésemos indicar cuáles son los elementos de las tablas componentes, entonces tendríamos que indicar al menos cuál es el número de elementos que podrán almacenar (se inicializarán con valores por defecto) quedando:

```
int[][] tablaDentada = {new int[2], new int[3]};
```

Si no queremos crear las tablas componentes en el momento de crear la tabla dentada, entonces tendremos que indicar por lo menos cuál es el número de tablas componentes posibles (cada una valdría **null**), con lo que quedaría:

```
int[][] tablaDentada = new int[2][];
```

Es importante señalar que no es posible especificar todas las dimensiones de una tabla dentada en su definición si no se indica explícitamente el valor inicial de éstas entre llaves. Es decir, esta declaración es incorrecta:

```
int[][] tablaDentada = new int[2][5];
```

Esto se debe a que el tamaño de cada tabla componente puede ser distinto y con la sintaxis anterior no se puede decir cuál es el tamaño de cada una. Una opción hubiese sido considerar que es 5 para todas como se hace en Java, pero ello no se ha implementado en C# y habría que declarar la tabla de, por ejemplo, esta manera:

```
int[][] tablaDentada = {new int[5], new int[5]);
```

Finalmente, si sólo queremos declarar una variable tabla dentada pero no queremos indicar su número de elementos, (luego la variable valdría **null**), entonces basta poner:

```
int[][] tablaDentada;
```

Hay que precisar que aunque en los ejemplos hasta ahora presentes se han escrito ejemplos basados en tablas dentadas de sólo dos niveles de anidación, también es posible crear tablas dentadas de cualquier número de niveles de anidación. Por ejemplo, para una tabla de tablas de tablas de enteros de 2 elementos en la que el primero fuese una tabla dentada formada por dos tablas de 5 enteros y el segundo elemento fuese una tabla dentada formada por una tabla de 4 enteros y otra de 3 se podría definir así:

```
int[][][] tablaDentada = new int[][][]
                         { new int[][] {new int[5], new int[5]},
                           new int[][] {new int[4], new int[3]}};
```

A la hora de acceder a los elementos de una tabla dentada lo único que hay que hacer es indicar entre corchetes cuál es el elemento exacto de las tablas componentes al que se desea acceder, indicándose un elemento de cada nivel de anidación entre unos corchetes diferentes pero colocándose todas las parejas de corchetes juntas y ordenadas de la tabla más externa a la más interna. Por ejemplo, para asignar el valor 10 al elemento cuarto de la tabla que es elemento primero de la tabla que es elemento segundo de la tabla dentada declarada en último lugar se haría:

```
tablaDentada[1][0][3] = 10;
```

Tablas multidimensionales

Una **tabla multidimensional** es una tabla cuyos elementos se encuentran organizando una estructura de varias dimensiones. Para definir este tipo de tablas se usa una sintaxis similar a la usada para declarar tablas unidimensionales pero separando las diferentes dimensiones mediante comas (`,`) Por ejemplo, una tabla multidimensional de elementos de tipo `int` que conste de 12 elementos puede tener sus elementos distribuidos en dos dimensiones formando una estructura 3x4 similar a una matriz de la forma:

```
1   2   3   4
5   6   7   8
9 10 11 12
```

Esta tabla se podría declarar así:

```
int[,] tablaMultidimensional = new int[3,4] {, , };
```

En realidad no es necesario indicar el número de elementos de cada dimensión de la tabla ya que pueden deducirse de los valores explícitamente indicados entre llaves, por lo que la definición anterior es similar a esta:

```
int[,] tablaMultidimensional = new int[,] {, , };
```

Incluso puede reducirse aún más la sintaxis necesaria quedando tan sólo:

```
int[,] tablaMultidimensional =  {, , };
```

Si no queremos indicar explícitamente los elementos de la tabla al declararla, podemos obviarlos pero aún así indicar el tamaño de cada dimensión de la tabla (a los elementos se les daría el valor por defecto de su tipo de dato) así:

```
int[,] tablaMultidimensional = new int[3,4];
```

También podemos no especificar ni siquiera el número de elementos de la tabla de esta forma (`tablaMultidimensional` contendría ahora `null`):

```
int[,] tablaMultidimensional;
```

Aunque los ejemplos de tablas multidimensionales hasta ahora mostrados son de tablas de dos dimensiones, en general también es posible crear tablas de cualquier número de dimensiones. Por ejemplo, una tabla que almacene 24 elementos de tipo **int** y valor 0 en una estructura tridimensional 3x4x2 se declararía así:

```
int[,,] tablaMultidimensional = new int[3,4,2];
```

El acceso a los elementos de una tabla multidimensional es muy sencillo: sólo hay que indicar los índices de la posición que ocupe en la estructura multidimensional el elemento al que se desee acceder. Por ejemplo, para incrementar en una unidad el elemento que ocupe la posición (1,3,2) de la tabla anterior se haría (se indiza desde 0):

```
tablaMultidimensional[0,2,1]++;
```

Nótese que tanto las tablas dentadas como las tablas multidimensionales pueden ser utilizadas tanto para representar estructuras matriciales como para, en general, representar cualquier estructura de varias dimensiones. La diferencia entre ambas son:

- Como las tablas dentadas son tablas de tablas, cada uno de sus elementos puede ser una tabla de un tamaño diferente. Así, con las tablas dentadas podemos representar matrices en las que cada columna tenga un tamaño distinto (por el aspecto "aserrado" de este tipo de matrices es por lo que se les llama tablas dentadas), mientras que usando tablas multidimensionales sólo es posible crear matrices rectangulares o cuadradas. Las estructuras aserradas pueden simularse usando matrices multidimensionales con todas sus columnas del tamaño de la columna más grande necesaria, aunque ello implica desperdiciar mucha memoria sobre todo si los tamaños de cada columna son muy diferentes y la tabla es grande De todos modos, las estructuras más comunes que se usan en la mayoría de aplicaciones suelen ser rectangulares o cuadradas.
- Los tiempos que se tardan en crear y destruir tablas dentadas son superiores a los que se tardan en crear y destruir tablas multidimensionales. Esto se debe a que las primeras son tablas de tablas mientras que las segundas son una única tabla, Por ejemplo, para crear una tabla dentada [100][100] hay que crear 101 tablas (la tabla dentada más las 100 tablas que contiene), mientras que para crear una crear una tabla bidimensional [100,100] hay que crear una única tabla.
- Las tablas dentadas no forman parte del CLS, por lo que no todos los lenguajes gestionados los tienen porqué admitir. Por ejemplo Visual Basic.NET no las admite, por lo que al usarlas en miembros públicos equivale a perder interoperabilidad con estos lenguajes.

Tablas mixtas

Una **tabla mixta** es simplemente una tabla formada por tablas multidimensionales y dentadas combinadas entre sí de cualquier manera. Para declarar una tabla de este tipo basta con tan solo combinar las notaciones ya vistas para las multidimensionales y dentadas. Por ejemplo, para declarar una tabla de tablas multidimensionales cuyos elementos sean tablas unidimensionales de enteros se haría lo siguiente:

```
int[][,][] tablaMixta;
```

Covarianza de tablas

La **covarianza de tablas** es el resultado de llevar el polimorfismo al mundo de las tablas. Es decir, es la capacidad de toda tabla de poder almacenar elementos de clases hijas de la clase de elementos que pueda almacenar. Por ejemplo, en tanto que todas clases son hijas de **System.Object**, la siguiente asignación es válida:

```
string[] tablaCadenas = {"Manolo", "Paco", "Pepe"};
```

```
object[] tablaObjetos = tablaCadenas;
```

Hay que tener en cuenta que la covarianza de tablas sólo se aplica a objetos de tipos referencia y no a objetos de tipos valor Por ejemplo, la siguiente asignación no sería válida en tanto que **int** es un tipo por valor:

```
int[] tablaEnteros = {1, 2, 3};
object[] tablaObjetos = tablaEnteros;
```

La clase System.Array

En realidad, todas las tablas que definamos, sea cual sea el tipo de elementos que contengan, son objetos que derivan de **System.Array**. Es decir, van a disponer de todos los miembros que se han definido para esta clase, entre los que son destacables:

- **Length**: Campo de sólo lectura que informa del número total de elementos que contiene la tabla. Si la tabla tiene más de una dimensión o nivel de anidación indica el número de elementos de todas sus dimensiones y niveles. Por ejemplo:

```
int[] tabla = ;
int[][] tabla2 = {new int[] , new int[] };
int[,] tabla3 = {,};

Console.WriteLine(tabla.Length);   //Imprime 4
Console.WriteLine(tabla2.Length);  //Imprime 5
Console.WriteLine(tabla3.Length);  //Imprime 6
```

- **Rank**: Campo de sólo lectura que almacena el número de dimensiones de la tabla. Obviamente si la tabla no es multidimensional valdrá 1. Por ejemplo:

```
int[] tabla = ;
int[][] tabla2 = {new int[] , new int[] };
int[,] tabla3 = {,};

Console.WriteLine(tabla.Rank);   //Imprime 1
Console.WriteLine(tabla2.Rank);  //Imprime 1
Console.WriteLine(tabla3.Rank);  //Imprime 2
```

- **int GetLength(int dimensión)**: Método que devuelve el número de elementos de la dimensión especificada. Las dimensiones se indican empezando a contar desde cero, por lo que si quiere obtenerse el número de elementos de la primera dimensión habrá que usar GetLength(0), si se quiere obtener los de la segunda habrá que usar GetLength(1), etc. Por ejemplo:

```
int[,] tabla = {, };

Console.WriteLine(tabla.GetLength(0));    // Imprime 2
Console.WriteLine(tabla.GetLength(1));    // Imprime 4
```

- **void CopyTo(Array destino, int posición)**: Copia todos los elementos de la tabla sobre la que es aplica en la que se le pasa como primer parámetro a partir de la posición de la misma indicada como segundo parámetro. Por ejemplo:

```
int[] tabla1 = ;
int[] tabla2 = ;
```

```
tabla1.CopyTo(tabla2,0);  // A partir de ahora, ambas tablas contienen
```

Ambas tablas han de ser unidimensionales. Por otro lado, y como es obvio, la tabla de destino ha de ser de un tipo que pueda almacenar los objetos de la tabla fuente, el índice especificado ha de ser válido (mayor o igual que cero y menor que el tamaño de la tabla de destino) y no ha de valer **null** ninguna. Si no fuese así, saltarían excepciones de diversos tipos informando del error cometido (en la documentación del SDK puede ver cuáles son en concreto).

Aparte de los miembros aquí señalados, de **System.Array** cuenta con muchos más que permiten realizar tareas tan frecuentes como búsquedas de elementos, ordenaciones, etc.

Cadenas de texto

Una **cadena de texto** no es más que una secuencia de caracteres Unicode. En C# se representan mediante objetos del tipo tipo de dato llamado **string**, que no es más que un alias del tipo **System.String** incluido en la BCL.

Las cadenas de texto suelen crearse a partir literales de cadena o de otras cadenas previamente creadas. Ejemplos de ambos casos se muestran a continuación:

```
string cadena1 = "José Antonio";
string cadena2 = cadena;
```

En el primer caso se ha creado un objeto **string** que representa a la cadena formada por la secuencia de caracteres José Antonio indicada literalmente (nótese que las comillas dobles entre las que se encierran los literales de cadena no forman parte del contenido de la cadena que representan sino que sólo se usan como delimitadores de la misma) En el segundo caso la variable cadena2 creada se genera a partir de la variable cadena1 ya existente, por lo que ambas variables apuntarán al mismo objeto en memoria.

Hay que tener en cuenta que el tipo **string** es un tipo referencia, por lo que en principio la comparación entre objetos de este tipo debería comparar sus direcciones de memoria como pasa con cualquier tipo referencia. Sin embargo, si ejecutamos el siguiente código veremos que esto no ocurre en el caso de las cadenas:

```
using System;

public class IgualdadCadenas
{
  public static void Main()
  {
    string cadena1 = "José Antonio";
    string cadena2 = String.Copy(cadena1);

    Console.WriteLine(cadena1==cadena2);
  }
}
```

El método **Copy()** de la clase **String** usado devuelve una copia del objeto que se le pasa como parámetro. Por tanto, al ser objetos diferentes se almacenarán en posiciones distintas de memoria y al compararlos debería devolverse **false** como pasa con cualquier tipo referencia. Sin embargo, si ejecuta el programa verá que lo que se obtiene es precisamente lo contrario: **true**. Esto se debe a que para hacer para hacer más intuitivo el trabajo con cadenas, en C# se ha modificado el operador de igualdad para que cuando se aplique entre cadenas se considere que sus operandos son iguales sólo si son lexicográficamente equivalentes y no si referencian al mismo objeto en memoria. Además, esta comparación se hace teniendo en cuenta la capitalización usada, por lo que `"Hola"=="HOLA"` ó `"Hola"=="hola"` devolverán **false** ya que contienen las mismas letras pero con distinta capitalización.

Si se quisiese comparar cadenas por referencia habría que optar por una de estas dos opciones: compararlas con **Object.ReferenceEquals()** o convertirlas en **object**s y luego compararlas con **==** Por ejemplo:

```
Console.WriteLine(Object.ReferecenceEquals(cadena1, cadena2));
Console.WriteLine( (object) cadena1 == (object) cadena2);
```

Ahora sí que lo que se comparan son las direcciones de los objetos que representan a las cadenas en memoria, por lo que la salida que se mostrará por pantalla es:

```
False
False
```

Hay que señalar una cosa, y es que aunque en principio el siguiente código debería mostrar la misma salida por pantalla que el anterior ya que las cadenas comparadas se deberían corresponder a objetos que aunque sean lexicográficamente equivalentes se almacenan en posiciones diferentes en memoria:

```
using System;

public class IgualdadCadenas2
{
  public static void Main()
  {
    string cadena1 = "José Antonio";
    string cadena2 = "José Antonio";

    Console.WriteLine(Object.ReferenceEquals(cadena1, cadena2));
    Console.WriteLine( ((object) cadena1) == ((object) cadena2));
  }
}
```

Si lo ejecutamos veremos que la salida obtenida es justamente la contraria:

```
True
True
```

Esto se debe a que el compilador ha detectado que ambos literales de cadena son lexicográficamente equivalentes y ha decidido que para ahorra memoria lo mejor es almacenar en memoria una única copia de la cadena que representan y hacer que ambas variables apunten a esa copia común. Esto va a afectar a la forma en que es posible manipular las cadenas como se explicará más adelante.

Al igual que el significado del operador **==** ha sido especialmente modificado para trabajar con cadenas, lo mismo ocurre con el operador binario **+**. En este caso, cuando se aplica entre dos cadenas o una cadena y un carácter lo que hace es devolver una nueva cadena con el resultado de concatenar sus operandos. Así por ejemplo, en el siguiente código las dos variables creadas almacenarán la cadena `Hola Mundo`:

```
public class Concatenación
{
  public static void Main()
  {
    string cadena = "Hola" + " Mundo";
    string cadena2 = "Hola Mund" + 'o';
  }
}
```

Por otro lado, el acceso a las cadenas se hace de manera similar a como si de tablas de caracteres se tratase: su "campo" **Length** almacenará el número de caracteres que la forman y para acceder a sus elementos se

utiliza el operador **[]**. Por ejemplo, el siguiente código muestra por pantalla cada carácter de la cadena Hola en una línea diferente:

```
using System;

public class AccesoCadenas
{
  public static void Main()
  {
    string cadena = "Hola";

    Console.WriteLine(cadena[0]);
    Console.WriteLine(cadena[1]);
    Console.WriteLine(cadena[2]);
    Console.WriteLine(cadena[3]);
  }
}
```

Sin embargo, hay que señalar una diferencia importante respecto a la forma en que se accede a las tablas: las cadenas son inmutables, lo que significa que no es posible modificar los caracteres que las forman. Esto se debe a que el compilador comparte en memoria las referencias a literales de cadena lexicográficamente equivalentes para así ahorrar memoria, y si se permitiese modificarlos los cambios que se hiciesen a través de una variable a una cadena compartida afectarían al resto de variables que la compartan, lo que podría causar errores difíciles de detectar. Por tanto, hacer esto es incorrecto:

```
string cadena = "Hola";
cadena[0]="A"; //Error: No se pueden modificar las cadenas
```

Sin embargo, el hecho de que no se puedan modificar las cadenas no significa que no se puedan cambiar los objetos almacenados en las variables de tipo **string**. Por ejemplo, el siguiente código es válido:

```
String cad = "Hola";
cad = "Adios";  // Correcto, pues no se modifica la cadena
                // almacenada en cad sino que se hace que cad
                // pase a almacenar otra cadena distinta..
```

Si se desea trabajar con cadenas modificables puede usarse **Sytem.Text.StringBuilder**, que funciona de manera similar a **string** pero permite la modificación de sus cadenas en tanto que estas no se comparten en memoria. Para crear objetos de este tipo basta pasar como parámetro de su constructor el objeto **string** que contiene la cadena a representar mediante un **StringBuilder**, y para convertir un **StringBuilder** en **String** siempre puede usarse su método **ToString()** heredado de **System.Object**. Por ejemplo:

```
using System.Text;
using System;

public class ModificaciónCadenas
{
  public static void Main()
  {
    StringBuilder cadena = new StringBuilder("Pelas");
    String cadenaInmutable;

    cadena[0] = 'V';
    Console.WriteLine(cadena); // Muestra Velas
    cadenaInmutable = cadena.ToString();
    Console.WriteLine(cadenaInmutable); // Muestra Velas
```

```
  }
}
```

Aparte de los métodos ya vistos, en la clase **System.String** se definen muchos otros métodos aplicables a cualquier cadena y que permiten manipularla. Los principales son:

- **int IndexOf(string subcadena)**: Indica cuál es el índice de la primera aparición de la subcadena indicada dentro de la cadena sobre la que se aplica. La búsqueda de dicha subcadena se realiza desde el principio de la cadena, pero es posible indicar en un segundo parámetro opcional de tipo **int** cuál es el índice de la misma a partir del que se desea empezar a buscar. Del mismo modo, la búsqueda acaba al llegar al final de la cadena sobre la que se busca, pero pasando un tercer parámetro opcional de tipo **int** es posible indicar algún índice anterior donde terminarla.

  Nótese que es un método muy útil para saber si una cadena contiene o no alguna subcadena determinada, pues sólo si no la encuentra devuelve un **-1**.

- **int LastIndexOf(string subcadena)**: Funciona de forma similar a **IndexOf()** sólo que devuelve la posición de la última aparición de la subcadena buscada en lugar de devolver la de la primera.
- **string Insert(int posición, string subcadena)**: Devuelve la cadena resultante de insertar la subcadena indicada en la posición especificada de la cadena sobre la que se aplica.
- **string Remove(int posición, int número)**: Devuelve la cadena resultante de eliminar el número de caracteres indicado que hubiese en la cadena sobre al que se aplica a partir de la posición especificada.
- **string Replace(string aSustituir, string sustituta)**: Devuelve la cadena resultante de sustituir en la cadena sobre la que se aplica toda aparición de la cadena aSustituir indicada por la cadena sustituta especificada como segundo parámetro.
- **string Substring(int posición, int número)**: Devuelve la subcadena de la cadena sobre la que se aplica que comienza en la posición indicada y tiene el número de caracteres especificados. Si no se indica dicho número se devuelve la subcadena que va desde la posición indicada hasta el final de la cadena.
- **string ToUpper()** y **string ToLower()**: Devuelven, respectivamente, la cadena que resulte de convertir a mayúsculas o minúsculas la cadena sobre la que se aplican.

Es preciso incidir en que aunque hayan métodos de inserción, reemplazo o eliminación de caracteres que puedan dar la sensación de que es posible modificar el contenido de una cadena, en realidad las cadenas son inmutables y dicho métodos lo que hacen es devolver una nueva cadena con el contenido correspondiente a haber efectuado las operaciones de modifiación solicitadas sobre la cadena a la que se aplican. Por ello, las cadenas sobre las que se aplican quedan intactas como muestra el siguiente ejemplo:

```
Using System;

public class EjemploInmutabilidad
{
  public static void Main()
  {
    string cadena1="Hola";
    string cadena2=cadena1.Remove(0,1);

    Console.WriteLine(cadena1);
    Console.WriteLine(cadena2);
  }
}
```

La salida por pantalla de este ejemplo demuestra lo antes dicho, pues es:

```
Hola
ola
```

Como se ve, tras el **Remove()** la cadena1 permance intacta y el contenido de cadena2 es el que debería tener cadena1 si se le hubiese eliminado su primer carácter.

Constantes

Una **constante** es una variable cuyo valor puede determinar el compilador durante la compilación y puede aplicar optimizaciones derivadas de ello. Para que esto sea posible se ha de cumplir que el valor de una constante no pueda cambiar durante la ejecución, por lo que el compilador informará con un error de todo intento de modificar el valor inicial de una constante. Las constantes se definen como variables normales pero precediendo el nombre de su tipo del modificador **const** y dándoles siempre un valor inicial al declararlas. O sea, con esta sintaxis:

```
const <tipoConstante> <nombreConstante> = <valor>;
```

Así, ejemplos de definición de constantes es el siguiente:

```
const int a = 123;
const int b = a + 125;
```

Dadas estas definiciones de constantes, lo que hará el compilador será sustituir en el código generado todas las referencias a las constantes a y b por los valores 123 y 248 respectivamente, por lo que el código generado será más eficiente ya que no incluirá el acceso y cálculo de los valores de a y b. Nótese que puede hacer esto porque en el código se indica explícitamente cual es el valor que siempre tendrá a y, al ser este un valor fijo, puede deducir cuál será el valor que siempre tendrá b. Para que el compilador pueda hacer estos cálculos se ha de cumplir que el valor que se asigne a las constantes en su declaración sea una expresión constante. Por ejemplo, el siguiente código no es válido en tanto que el valor de x no es constante:

```
int x = 123;           // x es una variable normal, no una constante
const int y = x +123;  // Error: x no tiene porqué tener valor
                       // constante (aunque aquí lo tenga)
```

Debido a la necesidad de que el valor dado a una constante sea precisamente constante, no tiene mucho sentido crear constantes de tipos de datos no básicos, pues a no ser que valgan **null** sus valores no se pueden determinar durante la compilación sino únicamente tras la ejecución de su constructor. La única excepción a esta regla son los tipos enumerados, cuyos valores se pueden determinar al compilar como se explicará cuando los veamos en el *Tema 14: Enumeraciones*

Todas las constantes son implícitamente estáticas, por lo se considera erróneo incluir el modificador **static** en su definición al no tener sentido hacerlo. De hecho, para leer su valor desde códigos externos a la definición de la clase donde esté definida la constante, habrá que usar la sintaxis <nombreClase>.<nombreConstante> típica de los campos **static**.

Por último, hay que tener en cuenta que una variable sólo puede ser definida como constante si se es una variable local o un campo, pero no si es un parámetro.

Variables de sólo lectura

Dado que hay ciertos casos en los que resulta interesante disponer de la capacidad de sólo lectura que tienen las constantes pero no es posible usarlas debido a las restricciones que hay impuestas sobre su uso, en C# también se da la posibilidad de definir variables que sólo puedan ser leídas. Para ello se usa la siguiente sintaxis:

```
readonly <tipoConstante> <nombreConstante> = <valor>;
```

Estas variables superan la mayoría de las limitaciones de las constantes. Por ejemplo:

- No es obligatorio darles un valor al definirlas, sino que puede dárseles en el constructor. Ahora bien, una vez dado un valor a una variable **readonly** ya no es posible volverlo a modificar. Si no se le da ningún valor ni en su constructor ni en su definición tomará el valor por defecto correspondiente a su tipo de dato.

- No tienen porqué almacenar valores constantes, sino que el valor que almacenen puede calcularse durante la ejecución de la aplicación.

- No tienen porqué definirse como estáticas, aunque si se desea puede hacerse.

- Su valor se determina durante la ejecución de la aplicación, lo que permite la actualización de códigos cliente sin necesidad de recompilar. Por ejemplo, dado:

```
namespace Programa1
{
  public class Utilidad
  {
    public static readonly int X = 1;
  }
}
namespace Programa2
{
  class Test
  {
    public static void Main() {
      System.Console.WriteLine(Programa1.Utilidad.X);

    }
  }
}
```

En principio, la ejecución de este programa producirá el valor 1. Sin embargo, si cada espacio de nombres se compilan en módulos de código separados que luego se enlazan dinámicamete y cambiamos el valor de `X`, sólo tendremos que recompilar el módulo donde esté definido `Programa1.Utilidad` y `Programa2.Test` podrá ejecutarse usando el nuevo valor de X sin necesidad de recompilalo.

Sin embargo, pese a las ventajas que las variables de sólo lectura ofrecen respecto a las constantes, tienen dos inconvenientes respecto a éstas: sólo pueden definirse como campos (no como variables locales) y con ellas no es posible realizar las optimizaciones de código comentadas para las constantes.

Orden de inicialización de variables

Para deducir el orden en que se inicializarán las variables de un tipo de dato basta saber cuál es el momento en que se inicializa cada una y cuando se llama a los constructores:

- Los **campos estáticos** sólo se inicializan la primera vez que se accede al tipo al que pertenecen, pero no en sucesivos accesos. Estos accesos pueden ser tanto para crear objetos de dicho tipo como para acceder a sus miembros estáticos. La inicialización se hace de modo que en primer lugar se dé a cada variable el valor por defecto correspondiente a su tipo, luego se dé a cada una el valor inicial especificado al definirlas, y por último se llame al constructor del tipo. Un constructor de tipo es similar a un constructor normal sólo que en

su código únicamente puede accederse a miembros **static** (se verá en el *Tema 8: Métodos*)

- Los **campos no estáticos** se inicializan cada vez que se crea un objeto del tipo de dato al que pertenecen. La inicialización se hace del mismo modo que en el caso de los campos estáticos, y una vez terminada se pasa a ejecutar el código del constructor especificado al crear el objeto. En caso de que la creación del objeto sea el primer acceso que se haga al tipo de dato del mismo, entonces primero se inicializarán los campos estáticos y luego los no estáticos.

- Los **parámetros** se inicializan en cada llamada al método al que pertenecen con los valores especificados al llamarlo.

- Las **variables locales** se inicializan en cada llamada al método al cual pertenecen pero tras haberse inicializado los parámetros definidos para el mismo. Si no se les da valor inicial no toman ninguno por defecto, considerándose erróneo todo acceso de lectura que se haga a las mismas mientras no se les escriba algún valor.

Hay que tener en cuenta que al definirse campos estáticos pueden hacerse definiciones cíclicas en las que el valor de unos campos dependa del de otros y el valor de los segundos dependa del de los primeros. Por ejemplo:

```
class ReferenciasCruzadas
{
  static int a = b + 1;
  static int b = a + 1;

  public static void Main()
  {
    System.Console.WriteLine("a = {0}, b = {1}", a, b);
  }
}
```

Esto sólo es posible hacerlo al definir campos estáticos y no entre campos no estáticas o variables locales, ya que no se puede inicializar campos no estáticos en función del valor de otros miembros no estáticos del mismo objeto porque el objeto aún no estaría incializados, y no se puede inicializar variables locales en función del valor de otras variables locales definidas más adelante porque no se pueden leer variables no inicializadas. Además, aunque las constantes sean implícitamente estáticas tampoco puede hacerse definiciones cíclicas entre constantes.

En primer lugar, hay que señalar que escribir un código como el del ejemplo anterior no es un buen hábito de programación ya que dificulta innecesariamente la legibilidad del programa. Aún así, C# admite este tipo de códigos y para determinar el valor con que se inicializarán basta tener en cuenta que siempre se inicializan primero todos los campos con sus valores por defecto y luego se inicializan aquellos que tengan valores iniciales con dichos valores iniciales y en el mismo orden en que aparezcan en el código fuente. De este modo, la salida del programa de ejemplo anterior será:

```
a = 1, b = 2
```

Nótese que lo que se ha hecho es inicializar primero `a` y `b` con sus valores por defecto (0 en este caso), luego calcular el valor final de `a` y luego calcular el valor final de `b`. Como `b` vale 0 cuando se calcula el valor final de a, entonces el valor final de `a` es 1; y como `a` vale 1 cuando se calcula el valor final de `b`, entonces el valor final de `b` es 2.

Principio Página

El lenguaje de programación C#

En esta página:

Tema 8: Métodos

Concepto de método

Un **método** es un conjunto de instrucciones a las que se les da un determinado nombre de tal manera que sea posible ejecutarlas en cualquier momento sin tenerlas que reescribir sino usando sólo su nombre. A estas instrucciones se les denomina **cuerpo** del método, y a su ejecución a través de su nombre se le denomina **llamada** al método.

La ejecución de las instrucciones de un método puede producir como resultado un objeto de cualquier tipo. A este objeto se le llama **valor de retorno** del método y es completamente opcional, pudiéndose escribir métodos que no devuelvan ninguno.

La ejecución de las instrucciones de un método puede depender del valor de unas variables especiales denominadas **parámetros** del método, de manera que en función del valor que se dé a estas variables en cada llamada la ejecución del método se pueda realizar de una u otra forma y podrá producir uno u otro valor de retorno.

Al conjunto formado por el nombre de un método y el número y tipo de sus parámetros se le conoce como **signatura** del método. La signatura de un método es lo que verdaderamente lo identifica, de modo que es posible definir en un mismo tipo varios métodos con idéntico nombre siempre y cuando tengan distintos parámetros. Cuando esto ocurre se dice que el método que tiene ese nombre está **sobrecargado**.

Definición de métodos

Para definir un método hay que indicar tanto cuáles son las instrucciones que forman su cuerpo como cuál es el nombre que se le dará, cuál es el tipo de objeto que puede devolver y cuáles son los parámetros que puede tomar. Esto se indican definiéndolo así:

```
<tipoRetorno> <nombreMétodo>(<parámetros>)
{
  <cuerpo>
}
```

En <tipoRetorno> se indica cuál es el tipo de dato del objeto que el método devuelve, y si no devuelve ninguno se ha de escribir **void** en su lugar.

Como nombre del método se puede poner en <nombreMétodo> cualquier identificador válido. Como se verá más adelante en el *Tema 15: Interfaces*, también es posible incluir en <nombreMétodo> información de explicitación de implementación de interfaz, pero por ahora podemos considerar que siempre será un identificador.

Aunque es posible escribir métodos que no tomen parámetros, si un método los toma se ha de indicar en <*parámetros*> cuál es el nombre y tipo de cada uno de ellos, separándolos con comas si son más de uno y siguiendo la sintaxis que más adelante en este mismo tema es explica.

El <*cuerpo*> del método también es opcional, pero si el método retorna algún tipo de objeto entonces ha de incluir al menos una instrucción **return** que indique cuál es el objeto a devolver.

La sintaxis anteriormente vista no es la que se usa para definir **métodos abstractos**. Como ya se vio en el *Tema 5: Clases*, en esos casos lo que se hace es sustituir el cuerpo del método y las llaves que lo encierran por un simple punto y coma (;) Más adelante en este tema veremos que eso es también lo que se hace para definir **métodos externos**.

A continuación se muestra un ejemplo de cómo definir un método de nombre Saluda cuyo cuerpo consista en escribir en la consola el mensaje "Hola Mundo" y que devuelva un objeto **int** de valor 1:

```
int Saluda()
{
  Console.WriteLine("Hola Mundo");
  return 1;
}
```

Llamada a métodos

La forma en que se puede llamar a un método depende del tipo de método del que se trate. Si es un **método de objeto** (método no estático) se ha de usar la notación:

```
<objeto>.<nombreMétodo>(<valoresParámetros>)
```

El <objeto> indicado puede ser directamente una variable del tipo de datos al que pertenezca el método o puede ser una expresión que produzca como resultado una variable de ese tipo

(recordemos que, debido a la herencia, el tipo del <objeto> puede ser un subtipo del tipo donde realmente se haya definido el método); pero si desde código de algún método de un objeto se desea llamar a otro método de ese mismo objeto, entonces se ha de dar el valor **this** a <objeto>.

En caso de que sea un **método de tipo** (método estático), entones se ha de usar:

```
<tipo>.<nombreMétodo>(<valoresParámetros>)
```

Ahora en <tipo> ha de indicarse el tipo donde se haya definido el método o algún subtipo suyo. Sin embargo, si el método pertenece al mismo tipo que el código que lo llama entonces se puede usar la notación abreviada:

```
<nombreMétodo>(<valoresParámetros>)
```

El formato en que se pasen los valores a cada parámetro en <valoresParámetros> a aquellos métodos que tomen parámetros depende del tipo de parámetro que sea. Esto se explica en el siguiente apartado.

Tipos de parámetros. Sintaxis de definición

La forma en que se define cada parámetro de un método depende del tipo de parámetro del que se trate. En C# se admiten cuatro tipos de parámetros: parámetros de entrada, parámetros de salida, parámetros por referencia y parámetros de número indefinido.

Parámetros de

Un **parámetro de entrada** recibe una copia del valor que almacenaría una variable del tipo del objeto que se le pase. Por tanto, si el objeto es de un tipo valor se le pasará una copia del objeto y cualquier modificación que se haga al parámetro dentro del cuerpo del método no afectará al objeto original sino a su copia; mientras que si el objeto es de un tipo referencia entonces se le pasará una copia de la referencia al mismo y cualquier modificación que se haga al parámetro dentro del método también afectará al objeto original ya que en realidad el parámetro referencia a ese mismo objeto original.

Para definir un parámetro de entrada basta indicar cuál el nombre que se le desea dar y el cuál es tipo de dato que podrá almacenar. Para ello se sigue la siguiente sintaxis:

```
<tipoParámetro> <nombreParámetro>
```

Por ejemplo, el siguiente código define un método llamado Suma que toma dos parámetros de entrada de tipo **int** llamados par1 y par2 y devuelve un **int** con su suma:

```
int Suma(int par1, int par2)
{
  return par1+par2;
}
```

Como se ve, se usa la instrucción **return** para indicar cuál es el valor que ha de devolver el método. Este valor es el resultado de ejecutar la expresión par1+par2; es decir, es la suma de los valores pasados a sus parámetros par1 y par2 al llamarlo.

En las llamadas a métodos se expresan los valores que se deseen dar a este tipo de parámetros indicando simplemente el valor deseado. Por ejemplo, para llamar al método anterior con los valores 2 y 5 se haría `<objeto>.Suma(2,5)`, lo que devolvería el valor 7.

Todo esto se resume con el siguiente ejemplo:

```
using System;

class ParámetrosEntrada
{
  public int a = 1;

  public static void F(ParámetrosEntrada p)
  {
    p.a++;
  }

  public static void G(int p)
  {
    p++;
  }

  public static void Main()
  {
    int obj1 = 0;
    ParámetrosEntrada obj2 = new ParámetrosEntrada();

    G(obj1);
    F(obj2);

    Console.WriteLine("{0}, {1}", obj1, obj2.a);
  }
}
```

Este programa muestra la siguiente salida por pantalla:

```
0, 2
```

Como se ve, la llamada al método `G()` no modifica el valor que tenía `obj1` antes de llamarlo ya que `obj1` es de un tipo valor (`int`) Sin embargo, como `obj2` es de un tipo referencia (`ParámetrosLlamadas`) los cambios que se le hacen dentro de `F()` al pasárselo como parámetro sí que le afectan.

Parámetros de salida

Un **parámetro de salida** se diferencia de uno de entrada en que todo cambio que se le realice en el código del método al que pertenece afectará al objeto que se le pase al llamar dicho método tanto si éste es de un tipo por como si es de un tipo referencia. Esto se debe a que lo que a estos parámetros se les pasa es siempre una referencia al valor que almacenaría una variable del tipo del objeto que se les pase.

Cualquier parámetro de salida de un método siempre ha de modificarse dentro del cuerpo del método

y además dicha modificación ha de hacerse antes que cualquier lectura de su valor. Si esto no se hiciese así el compilador lo detectaría e informaría de ello con un error. Por esta razón es posible pasar parámetros de salida que sean variables no inicializadas, pues se garantiza que en el método se inicializarán antes de leerlas. Además, tras la llamada a un método se considera que las variables que se le pasaron como parámetros de salida ya estarán inicializadas, pues dentro del método seguro que se las inicializació.

Nótese que este tipo de parámetros permiten diseñar métodos que devuelvan múltiples objetos: un objeto se devolvería como valor de retorno y los demás se devolverían escribiendos en los parámetros de salida.

Los parámetros de salida se definen de forma parecida a los parámetros de entrada pero se les ha de añadir la palabra reservada **out**. O sea, se definen así:

```
out <tipoParámetro> <nombreParámetro>
```

Al llamar a un método que tome parámetros de este tipo también se ha preceder el valor especificado para estos parámetros del modificador **out**. Una utilidad de esto es facilitar la legibilidad de las llamadas a métodos. Por ejemplo, dada una llamada de la forma:

```
a.f(x, out z)
```

Es fácil determinar que lo que se hace es llamar al método `f()` del objeto a pasándole `x` como parámetro de entrada y `z` como parámetro de salida. Además, también se puede deducir que el valor de `z` cambiará tras la llamada.

Sin embargo, la verdadera utilidad de forzar a explicitar en las llamadas el tipo de paso de cada parámetro es que permite evitar errores derivados de que un programador pase una variable a un método y no sepa que el método la puede modificar. Teniéndola que explicitar se asegura que el programador sea consciente de lo que hace.

Parámetros por referencia

Un **parámetro por referencia** es similar a un parámetro de salida sólo que no es obligatorio modificarlo dentro del método al que pertenece, por lo que será obligatorio pasarle una variable inicializada ya que no se garantiza su inicialización en el método.

Los parámetros por referencia se definen igual que los parámetros de salida pero sustituyendo el modificador **out** por el modificador **ref**. Del mismo modo, al pasar valores a parámetros por referencia también hay que precederlos del **ref**.

Parámetros de número indefinido

C# permite diseñar métodos que puedan tomar cualquier número de parámetros. Para ello hay que indicar como último parámetro del método un parámetro de algún tipo de tabla unidimensional o dentada precedido de la palabra reservada **params**. Por ejemplo:

```
static void F(int x, params object[] extras)
{}
```

Todos los parámetros de número indefinido que se pasan al método al llamarlo han de ser del mismo

tipo que la tabla. Nótese que en el ejemplo ese tipo es la clase primigenia **object**, con lo que se consigue que gracias al polimorfismo el método pueda tomar cualquier número de parámetros de cualquier tipo. Ejemplos de llamadas válidas serían:

```
F(4);    // Pueden pasarse 0 parámetros indefinidos
F(3,2);
F(1, 2, "Hola", 3.0, new Persona());
F(1, new object[] {2,"Hola", 3.0, new Persona});
```

El primer ejemplo demuestra que el número de parámetros indefinidos que se pasen también puede ser 0. Por su parte, los dos últimos ejemplos son totalmente equivalentes, pues precisamente la utilidad de palabra reservada **params** es indicar que se desea que la creación de la tabla **object[]** se haga implícitamente.

Es importante señalar que la prioridad de un método que incluya el **params** es inferior a la de cualquier otra sobrecarga del mismo. Es decir, si se hubiese definido una sobrecarga del método anterior como la siguiente:

```
static void F(int x, int y)
{}
```

Cuando se hiciese una llamada como `F(3,2)` se llamaría a esta última versión del método, ya que aunque la del **params** es también aplicable, se considera que es menos prioritaria.

Sobrecarga de tipos de parámetros

En realidad los modificadores **ref** y **out** de los parámetros de un método también forman parte de lo que se conoce como signatura del método, por lo que esta clase es válida:

```
class Sobrecarga
{
  public void f(int x)
  {}

  public void f(out int x)
  {}
}
```

Nótese que esta clase es correcta porque cada uno de sus métodos tiene una signatura distinta: el parámetro es de entrada en el primero y de salida en el segundo.

Sin embargo, hay una restricción: no puede ocurrir que la única diferencia entre la signatura de dos métodos sea que en uno un determinado parámetro lleve el modificador **ref** y en el otro lleve el modificador **out**. Por ejemplo, no es válido:

```
class SobrecargaInválida
{
  public void f(ref int x)
  {}

  public void f(out int x)
```

```
  {}
}
```

Métodos externos

Un **método externo** es aquél cuya implementación no se da en el fichero fuente en que es declarado. Estos métodos se declaran precediendo su declaración del modificador **extern**. Como su código se da externamente, en el fuente se sustituyen las llaves donde debería escribirse su cuerpo por un punto y coma (**;**), quedando una sintaxis de la forma:

```
extern <nombreMétodo>(<parámetros>);
```

La forma en que se asocie el código externo al método no está definida en la especificación de C# sino que depende de la implementación que se haga del lenguaje. El único requisito es que no pueda definirse un método como abstracto y externo a la vez, pero por todo lo demás puede combinarse con los demás modificadores, incluso pudiéndose definir métodos virtuales externos.

La forma más habitual de asociar código externo consiste en preceder la declaración del método de un **atributo** de tipo **System.Runtime.InteropServices.DllImport** que indique en cuál librería de enlace dinámico (DLL) se ha implementado. Este atributo requiere que el método externo que le siga sea estático, y un ejemplo de su uso es:

```
using System.Runtime.InteropServices;  // Aquí está definido DllImport

public class Externo
{
  [DllImport("kernel32")]
  public static extern void CopyFile(string fuente, string destino);

  public static void Main()
  {
    CopyFile("fuente.dat", "destino.dat");
  }
}
```

El concepto de atributo se explica detalladamente en el *Tema 14:Atributos*. Por ahora basta saber que los atributos se usan de forman similar a los métodos sólo que no están asociados a ningún objeto ni tipo y se indican entre corchetes (**[]**) antes de declaraciones de elementos del lenguaje. En el caso concreto de **DllImport** lo que indica el parámetro que se le pasa es cuál es el fichero (por defecto se considera que su extensión es **.dll**) donde se encuentra la implementación del método externo a continuación definido.

Lo que el código del ejemplo anterior hace es simplemente definir un método de nombre `CopyFile()` cuyo código se corresponda con el de la función **CopyFile()** del fichero kernel32.dll del **API Win32**. Este método es llamado en `Main()` para copiar el fichero de nombre fuente.dat en otro de nombre destino.dat. Nótese que dado que `CopyFile()` se ha declarado como **static** y se le llama desde la misma clase donde se ha declarado, no es necesario precederlo de la notación `<nombreClase>.` para llamarlo.

Como se ve, la utilidad principal de los métodos externos es permitir hacer **llamadas a código nativo** desde código gestionado, lo que puede ser útil por razones de eficiencia o para reutilizar código antiguamente escrito pero reduce la portabilidad de la aplicación.

Constructores

Concepto de constructores

Los **constructores** de un tipo de datos son métodos especiales que se definen como miembros de éste y que contienen código a ejecutar cada vez que se cree un objeto de ese tipo. Éste código suele usarse para labores de inicialización de los campos del objeto a crear, sobre todo cuando el valor de éstos no es constante o incluye acciones más allá de una asignación de valor (aperturas de ficheros, accesos a redes, etc.)

Hay que tener en cuenta que la ejecución del constructor siempre se realiza después de haberse inicializado todos los campos del objeto, ya sea con los valores iniciales que se hubiesen especificado en su definición o dejándolos con el valor por defecto de su tipo.

Aparte de su especial sintaxis de definición, los constructores y los métodos normales tienen una diferencia muy importante: **los constructores no se heredan**.

Definición de constructores

La sintaxis básica de definición de constructores consiste en definirlos como cualquier otro método pero dándoles el mismo nombre que el tipo de dato al que pertenecen y no indicando el tipo de valor de retorno debido a que nunca pueden devolver nada. Es decir, se usa la sintaxis:

```
<modificadores> <nombreTipo>(<parámetros>)
{
  <código>
}
```

Un constructor nunca puede devolver ningún tipo de objeto porque, como ya se ha visto, sólo se usa junto al operador **new**, que devuelve una referencia al objeto recién creado. Por ello, es absurdo que devuelva algún valor ya que nunca podría ser capturado en tanto que **new** nunca lo devolvería. Por esta razón el compilador considera erróneo indicar algún tipo de retorno en su definición, incluso aunque se indique **void**.

Llamada al constructor

Al constructor de una clase se le llama en el momento en que se crea algún objeto de la misma usando el operador **new**. De hecho, la forma de uso de este operador es:

```
new <llamadaConstructor>
```

Por ejemplo, el siguiente programa demuestra cómo al crearse un objeto se ejecutan las instrucciones de su constructor:

```
class Prueba
{
  Prueba(int x)
  {
    System.Console.Write("Creado objeto Prueba con x={0}",x);
  }
```

```
  public static void Main()
  {
    Prueba p = new Prueba(5);
  }
}
```

La salida por pantalla de este programa demuestra que se ha llamado al constructor del objeto de clase `Prueba` creado en `Main()`, pues es:

```
Creado objeto Prueba con x=5;
```

Llamadas entre constructores

Al igual que ocurre con cualquier otro método, también es posible sobrecargar los constructores. Es decir, se pueden definir varios constructores siempre y cuando estos tomen diferentes números o tipos de parámetros. Además, desde el código de un constructor puede llamarse a otros constructores del mismo tipo de dato antes de ejecutar las instrucciones del cuerpo del primero. Para ello se añade un **inicializador this** al constructor, que es estructura que precede a la llave de apertura de su cuerpo tal y como se muestra en el siguiente ejemplo:

```
class A
{
  int total;

  A(int valor): this(valor, 2);  // (1)
  {
  }

  A(int valor, int peso) // (2)
  {
    total = valor*peso;
  }
}
```

El **`this`** incluido hace que la llamada al constructor (1) de la clase A provoque una llamada al constructor (2) de esa misma clase en la que se le pase como primer parámetro el valor originalmente pasado al constructor (1) y como segundo parámetro el valor 2. Es importante señalar que la llamada al constructor (2) en (1) se hace antes de ejecutar cualquier instrucción de (1)

Nótese que la sobrecarga de constructores -y de cualquier método en general- es un buen modo de definir versiones más compactas de métodos de uso frecuente en las que se tomen valores por defecto para parámetros de otras versiones menos compactas del mismo método. La implementación de estas versiones compactas consistiría en hacer una llamada a la versión menos compacta del método en la que se le pasen esos valores por defecto (a través del **`this`** en el caso de los constructores) y si acaso luego (y/o antes, si no es un constructor) se hagan labores específicas en el cuerpo del método compacto.

Del mismo modo que en la definición de un constructor de un tipo de datos es posible llamar a otros constructores del mismo tipo de datos, también es posible hacer llamadas a constructores de su tipo padre sustituyendo en su inicializador la palabra reservada **`this`** por **`base`**. Por ejemplo:

```
class A
{
  int total;

  A(int valor, int peso)
  {
    total = valor*peso;
  }
}

class B:A
{
  B(int valor):base(valor,2)
  {}
}
```

En ambos casos, los valores pasados como parámetros en el inicializador no pueden contener referencias a campos del objeto que se esté creando, ya que se considera que un objeto no está creado hasta que no se ejecute su constructor y, por tanto, al llamar al inicializador aún no está creado. Sin embargo, lo que sí pueden incluir son referencias a los parámetros con los que se llamó al constructor. Por ejemplo, sería válido hacer:

```
A(int x, int y): this(x+y)
{}
```

Constructor por defecto

Todo tipo de datos ha de disponer de al menos un constructor. Cuando se define un tipo sin especificar ninguno el compilador considera que implícitamente se ha definido uno sin cuerpo ni parámetros de la siguiente forma:

```
public <nombreClase>(): base()
{}
```

En el caso de que el tipo sea una clase abstracta, entonces el constructor por defecto introducido es el que se muestra a continuación, ya que el anterior no sería válido porque permitiría crear objetos de la clase a la que pertenece:

```
protected <nombreClase>(): base()
{}
```

En el momento en se defina explícitamente algún constructor el compilador dejará de introducir implícitamente el anterior. Hay que tener especial cuidado con la llamada que este constructor por defecto realiza en su inicializador, pues pueden producirse errores como el del siguiente ejemplo:

```
class A
{
  public A(int x)
  {}
}

class B:A
```

```
{
  public static void Main()
  {
    B b = new B();  // Error: No hay constructor base
  }
}
```

En este caso, la creación del objeto de clase `B` en `Main()` no es posible debido a que el constructor que por defecto el compilador crea para la clase `B` llama al constructor sin parámetros de su clase base `A`, pero `A` carece de dicho constructor porque no se le ha definido explícitamente ninguno con esas características pero se le ha definido otro que ha hecho que el compilador no le defina implícitamente el primero.

Otro error que podría darse consistiría en que aunque el tipo padre tuviese un constructor sin parámetros, éste fuese privado y por tanto inaccesible para el tipo hijo.

También es importante señalar que aún en el caso de que definamos nuestras propios constructores, si no especificamos un inicializador el compilador introducirá por nosotros uno de la forma `:base()` Por tanto, en estos casos también hay que asegurarse de que el tipo donde se haya definido el constructor herede de otro que tenga un constructor sin parámetros no privado.

Llamadas polimórficas en constructores

Es conveniente evitar en la medida de lo posible la realización de llamadas a métodos virtuales dentro de los constructores, ya que ello puede provocar errores muy difíciles de detectar debido a que se ejecuten métodos cuando la parte del objeto que manipulan aún no se ha sido inicializado. Un ejemplo de esto es el siguiente:

```
using System;

public class Base
{
  public Base()
  {
    Console.WriteLine("Constructor de Base");
    this.F();
  }

  public virtual void F()
  {
    Console.WriteLine("Base.F");
  }
}

public class Derivada:Base
{
  Derivada()
  {
    Console.WriteLine("Constructor de Derivada");
  }

  public override void F()
```

```
    {
      Console.WriteLine("Derivada.F()");
    }

    public static void Main()
    {
      Base b = new
      Derivada();
    }
  }
```

La salida por pantalla mostrada por este programa al ejecutarse es la siguiente:

```
Constructor de Base
Derivada.F()
Constructor de Derivada
```

Lo que ha ocurrido es lo siguiente: Al crearse el objeto `Derivada` se ha llamado a su constructor sin parámetros, que como no tiene inicializador implícitamente llama al constructor sin parámetros de su clase base. El constructor de Base realiza una llamada al método virtual `F()`, y como el verdadero tipo del objeto que se está contruyendo es `Derivada`, entonces versión del método virtual ejecutada es la redefinición del mismo incluida en dicha clase. Por último, se termina llamando al constructor de `Derivada` y finaliza la construcción del objeto.

Nótese que se ha ejecutado el método `F()` de `Derivada` antes que el código del constructor de dicha clase, por lo que si ese método manipulase campos definidos en Derivada que se inicializasen a través de constructor, se habría accedido a ellos antes de inicializarlos y ello seguramente provocaría errores de causas difíciles de averiguar.

Constructor de tipo

Todo tipo puede tener opcionalmente un **constructor de tipo**, que es un método especial que funciona de forma similar a los constructores ordinarios sólo que para lo que se usa es para inicializar los campos `static` del tipo donde se ha definido.

Cada tipo de dato sólo puede tener un constructor de tipo. Éste constructor es llamado automáticamente por el compilador la primera vez que se accede al tipo, ya sea para crear objetos del mismo o para acceder a sus campos estáticos. Esta llamada se hace justo después de inicializar los campos estáticos del tipo con los valores iniciales especificados al definirlos (o, en su ausencia, con los valores por defecto de sus tipos de dato), por lo que el programador no tiene forma de controlar la forma en que se le llama y, por tanto, no puede pasarle parámetros que condicionen su ejecución.

Como cada tipo sólo puede tener un constructor de tipo no tiene sentido poderse usar `this` en su inicializador para llamar a otro. Y además, tampoco tiene sentido usar `base` debido a que éste siempre hará referencia al constructor de tipo sin parámetros de su clase base. O sea, **un constructor de tipo no puede tener inicializador**.

Además, no tiene sentido darle modificadores de acceso ya que el programador nunca lo podrá llamar sino que sólo será llamado automáticamente y sólo al accederse al tipo por primera vez. Como es absurdo, el compilador considera un error dárselos.

La forma en que se define el constructor de tipo es similar a la de los constructores normales, sólo que ahora la definición ha de ir prefijada del modificador **static** y no puede contar con parámetros ni inicializador. O sea, se define de la siguiente manera:

```
static <nombreTipo>()
{
  <código>
}
```

En la especificicación de C# no se ha recogido cuál ha de ser el orden exacto de las llamadas a los constructores de tipos cuando se combinan con herencia, aunque lo que sí se indica es que se ha de asegurar de que no se accede a un campo estático sin haberse ejecutado antes su constructor de tipo. Todo esto puede verse más claro con un ejemplo:

```
using System;

class A
{
  public static X;

  static A()
  {
    Console.WriteLine("Constructor de A");
    X=1;
  }
}

class B:A
{
  static B()
  {
    Console.WriteLine("Constructor de B");
    X=2;
  }

  public static void Main()
  {
    B b = new B();
    Console.WriteLine(B.X);
  }
}
```

La salida que muestra por pantalla la ejecución de este programa es la siguiente:

```
Inicializada clase B
Inicializada clase A
2
```

En principio la salida de este programa puede resultar confusa debido a que los primeros dos mensajes parecen dar la sensación de que la creación del objeto `b` provocó que se ejecutase el constructor de la clase hija antes que al de la clase padre, pero el último mensaje se corresponde con una ejecución en el orden opuesto. Pues bien, lo que ha ocurrido es lo siguiente: como el orden de

llamada a constructores de tipo no está establecido, el compilador de Microsoft ha llamado antes al de la clase hija y por ello el primer mensaje mostrado es `Inicializada clase B`. Sin embargo, cuando en este constructor se va a acceder al campo X se detecta que la clase donde se definió aún no está inicializada y entonces se llama a su constructor de tipo, lo que hace que se muestre el mensaje `Inicializada clase A`. Tras esta llamada se machaca el valor que el constructor de `A` dio a `X` (valor 1) por el valor que el constructor de `B` le da (valor 2) Finalmente, el último `WriteLine()` muestra un `2`, que es el último valor escrito en `X`.

Destructores

Al igual que es posible definir métodos constructores que incluyan código que gestione la creación de objetos de un tipo de dato, también es posible definir un **destructor** que gestione cómo se destruyen los objetos de ese tipo de dato. Este método suele ser útil para liberar recursos tales como los ficheros o las conexiones de redes abiertas que el objeto a destruir estuviese acaparando en el momento en que se fuese a destruir.

La destrucción de un objeto es realizada por el recolector de basura cuando realiza una recolección de basura y detecta que no existen referencias a ese objeto ni en pila, ni en registros ni desde otros objetos sí referenciados. Las recolecciones se inician automáticamente cuando el recolector detecta que queda poca memoria libre o que se va a finalizar la ejecución de la aplicación, aunque también puede forzarse llamando al método **`Collect()`** de la clase **`System.GC`**

La sintaxis que se usa para definir un destructor es la siguiente:

```
~<nombreTipo>()
{
  <código>
}
```

Tras la ejecución del destructor de un objeto de un determinado tipo siempre se llama al destructor de su tipo padre, formándose así una cadena de llamadas a destructores que acaba al llegarse al destructor de **`object`**. Éste último destructor no contiene código alguno, y dado que **`object`** no tiene padre, tampoco llama a ningún otro destructor.

Los destructores no se heredan. Sin embargo, para asegurar que la cadena de llamadas a destructores funcione correctamente si no incluimos ninguna definición de destructor en un tipo, el compilador introducirá en esos casos una por nosotros de la siguiente forma:

```
~<nombreTipo>()

{}
```

El siguiente ejemplo muestra como se definen destructores y cómo funciona la cadena de llamada a destructores:

```
using System;

class A
{
  ~A()
  {
    Console.WriteLine("Destruido objeto de clase A");
```

```
  }
}

class B:A
{
  ~B()
  {
    Console.WriteLine("Destruido objeto de clase B");
  }
  public static void Main()
  {
    new B();
  }
}
```

El código del método `Main()` de este programa crea un objeto de clase `B` pero no almacena ninguna referencia al mismo. Luego finaliza la ejecución del programa, lo que provoca la actuación del recolector de basura y la destrucción del objeto creado llamando antes a su destructor. La salida que ofrece por pantalla el programa demuestra que tras llamar al destructor de `B` se llama al de su clase padre, ya que es:

```
Destruido objeto de clase B
Destruido objeto de clase A
```

Nótese que aunque no se haya guardado ninguna referencia al objeto de tipo `B` creado y por tanto sea innacesible para el programador, al recolector de basura no le pasa lo mismo y siempre tiene acceso a los objetos, aunque sean inútiles para el programador.

Es importante recalcar que no es válido incluir ningún modificador en la definición de un destructor, ni siquiera modificadores de acceso, ya que como nunca se le puede llamar explícitamente no tiene ningún nivel de acceso para el programador. Sin embargo, ello no implica que cuando se les llame no se tenga en cuenta el verdadero tipo de los objetos a destruir, como demuestra el siguiente ejemplo:

```
using System;

public class Base
{
  public virtual void F()
  {
    Console.WriteLine("Base.F");
  }

  ~Base()
  {
    Console.WriteLine("Destructor de Base");
    this.F();
  }
}

public class Derivada:Base
{
  ~Derivada()
```

```
    {
      Console.WriteLine("Destructor de Derivada");
    }

    public override void F()
    {
      Console.WriteLine("Derivada.F()");
    }

    public static void Main()
    {
      Base b = new Derivada();
    }
  }
```

La salida mostrada que muestra por pantalla este programa al ejecutarlo es:

```
Destructor de Derivada
Destructor de Base
Derivada.F()
```

Como se ve, aunque el objeto creado se almacene en una variable de tipo `Base`, su verdadero tipo es `Derivada` y por ello se llama al destructor de esta clase al destruirlo. Tras ejecutarse dicho contructor se llama al constructor de su clase padre siguiéndose la cadena de llamadas a destructores. En este constructor padre hay una llamada al método virtual `F()`, que como nuevamente el objeto que se está destruyendo es de tipo `Derivada`, la versión de `F()` a la que se llamará es a la de la dicha clase.

Nótese que una llamada a un método virtual dentro de un destructor como la que se hace en el ejemplo anterior puede dar lugar a errores difíciles de detectar, pues cuando se llama al método virtual ya se ha destruido la parte del objeto correspondiente al tipo donde se definió el método ejecutado. Así, en el ejemplo anterior se ha ejecutado `Derivada.F()` tras `Derivada.~F()`, por lo que si en `Derivada.F()` se usase algún campo destruido en `Derivada.~F()` podrían producirse errores difíciles de detectar.

Principio Página

El lenguaje de programación C#

En esta página:

Tema 9: Propiedades

Concepto de propiedad

Una **propiedad** es una mezcla entre el concepto de campo y el concepto de método. Externamente es accedida como si de un campo normal se tratase, pero internamente es posible asociar código a ejecutar en cada asignación o lectura de su valor. Éste código puede usarse para comprobar que no se asignen valores inválidos, para calcular su valor sólo al solicitar su lectura, etc.

Una propiedad no almacena datos, sino sólo se utiliza como si los almacenase. En la práctica lo que se suele hacer escribir como código a ejecutar cuando se le asigne un valor, código que controle que ese valor sea correcto y que lo almacene en un campo privado si lo es; y como código a ejecutar cuando se lea su valor, código que devuelva el valor almacenado en ese campo público. Así se simula que se tiene un campo público sin los inconvenientes que estos presentan por no poderse controlar el acceso a ellos.

Definición de propiedades

Para definir una propiedad se usa la siguiente sintaxis:

```
< tipoPropiedad> <nombrePropiedad>
{
  set
  {
```

```
    <códigoEscritura>
  }

  get
  {
    <códigoLectura>
  }
}
```

Una propiedad así definida sería accedida como si de un campo de tipo <tipoPropiedad> se tratase, pero en cada lectura de su valor se ejecutaría el <códigoLectura> y en cada escritura de un valor en ella se ejecutaría <códigoEscritura>

Al escribir los bloques de código **get** y **set** hay que tener en cuenta que dentro del código **set** se puede hacer referencia al valor que se solicita asignar a través de un parámetro especial del mismo tipo de dato que la propiedad llamado **value** (luego nosotros no podemos definir uno con ese nombre en <códigoEscritura>); y que dentro del código **get** se ha de devolver siempre un objeto del tipo de dato de la propiedad.

En realidad el orden en que aparezcan los bloques de código **set** y **get** es irrelevante. Además, es posible definir propiedades que sólo tengan el bloque **get** (**propiedades de sólo lectura**) o que sólo tengan el bloque **set** (**propiedades de sólo escritura**) Lo que no es válido es definir propiedades que no incluyan ninguno de los dos bloques.

Las propiedades participan del mecanismo de polimorfismo igual que los métodos, siendo incluso posible definir propiedades cuyos bloques de código **get** o **set** sean abstractos. Esto se haría prefijando el bloque apropiado con un modificador **abstract** y sustituyendo la definición de su código por un punto y coma. Por ejemplo:

```
using System;

abstract class A
{
  public abstract int PropiedadEjemplo
  {
    set;
    get;
  }
}

class B:A
{
  private int valor;

  public override int PropiedadEjemplo
  {
    get
    {
      Console.WriteLine("Leído {0} de PropiedadEjemplo", valor);
      return valor;
    }
```

```
      set
      {
        valor = value;
        Console.WriteLine("Escrito {0} en PropiedadEjemplo", valor);
      }
    }
  }
```

En este ejemplo se ve cómo se definen y redefinen propiedades abstractas. Al igual que **abstract** y **override**, también es posible usar cualquiera de los modificadores relativos a herencia y polimorfismo ya vistos: **virtual**, **new** y **sealed**.

Nótese que aunque en el ejemplo se ha optado por asociar un campo privado valor a la propiedad PropiedadEjemplo, en realidad nada obliga a que ello se haga y es posible definir propiedades que no tenga campos asociados. Es decir, una propiedad no se tiene porqué corresponder con un almacén de datos.

Acceso a propiedades

La forma de acceder a una propiedad, ya sea para lectura o escritura, es exactamente la misma que la que se usaría para acceder a un campo de su mismo tipo. Por ejemplo, se podría acceder a la propiedad de un objeto de la clase B del ejemplo anterior con:

```
B obj = new B();
obj.PropiedadEjemplo++;
```

El resultado que por pantalla se mostraría al hacer una asignación como la anterior sería:

```
Leído 0 de PropiedadEjemplo;
Escrito 1 en PropiedadEjemplo;
```

Nótese que en el primer mensaje se muestra que el valor leído es 0 porque lo que devuelve el bloque **get** de la propiedad es el valor por defecto del campo privado valor, que como es de tipo **int** tiene como valor por defecto 0.

Implementación interna de propiedades

En realidad la definición de una propiedad con la sintaxis antes vista es convertida por el compilador en la definición de un par de métodos de la siguiente forma:

```
<tipoPropiedad> get_<nombrePropiedad>()
{                       // Método en que se convierte el bloque get
  <códigoLectura>
}

void set_<nombrePropiedad> (<tipoPropiedad> value)
{                       // Método en que se convierte el bloque set
  <códigoEscritura>
}
```

Esto se hace para que desde lenguajes que no soporten las propiedades se pueda acceder también a ellas. Si una propiedad es de sólo lectura sólo se generará el método **get_X()**, y si es de sólo escritura sólo se generará el **set_X()** Ahora bien, en cualquier caso hay que tener cuidado con no definir en un mismo tipo de dato métodos con signaturas como estas si se van a generar internamente debido a la definición de una propiedad, ya que ello provocaría un error de definición múltiple de método.

Teniendo en cuenta la implementación interna de las propiedades, es fácil ver que el último ejemplo de acceso a propiedad es equivalente a:

```
B b = new B();
obj.set_PropiedadEjemplo(obj.get_Propiedad_Ejemplo()++);
```

Como se ve, gracias a las propiedades se tiene una sintaxis mucho más compacta y clara para acceder a campos de manera controlada. Se podría pensar que la contrapartida de esto es que el tiempo de acceso al campo aumenta considerablemente por perderse tiempo en hacer las llamada a métodos **set**/**get**. Pues bien, esto no tiene porqué ser así ya que el compilador de C# elimina llamadas haciendo **inlining** (sustitución de la llamada por su cuerpo) en los accesos a bloques **get**/**set** no virtuales y de códigos pequeños, que son los más habituales.

Nótese que de la forma en que se definen los métodos generados por el compilador se puede deducir el porqué del hecho de que en el bloque **set** se pueda acceder a través de **value** al valor asignado y de que el objeto devuelto por el código de un bloque **get** tenga que ser del mismo tipo de dato que la propiedad a la que pertenece.

Principio Página

En esta página:

Tema 10: Indizadores

Concepto de indizador

Un **indizador** es una definición de cómo se puede aplicar el operador de acceso a tablas (**[ ]**) a los objetos de un tipo de dato. Esto es especialmente útil para hacer más clara la sintaxis de acceso a elementos de objetos que puedan contener colecciones de elementos, pues permite tratarlos como si fuesen tablas normales.

Los indizadores permiten definir código a ejecutar cada vez que se acceda a un objeto del tipo del que son miembros usando la sintaxis propia de las tablas, ya sea para leer o escribir. A diferencia de las tablas, los índices que se les pase entre corchetes no tiene porqué ser enteros, pudiéndose definir varios indizadores en un mismo tipo siempre y cuando cada uno tome un número o tipo de índices diferente.

Definición de indizador

A la hora de definir un indizador se usa una sintaxis parecida a la de las propiedades:

```
<tipoIndizador> this[<índices>]
{
  set
  {
    <códigoEscritura>
  }
  get
  {
    <códigoLectura>
  }
}
```

Las únicas diferencias entre esta sintaxis y la de las propiedades son:

- El nombre dado a un indizador siempre ha de ser **this**, pues carece de sentido poder darle cualquiera en tanto que a un indizador no se accede por su nombre sino aplicando el operador **[ ]** a un objeto. Por ello, lo que diferenciará a unos indizadores de otros será el número y tipo de sus <índices>

- En <índices> se indica cuáles son los índices que se pueden usar al acceder al indizador. Para ello la sintaxis usada es casi la misma que la que se usa para especificar los parámetros de un método, sólo que no se admite la inclusión de modificadores **ref**, **out** o **params** y que siempre ha de definirse al menos un parámetro. Obviamente, el nombre que se dé a cada índice será el nombre con el que luego se podrá acceder al mismo en los bloques **set**/**get**.

- No se pueden definir indizadores estáticos, sino sólo indizadores de objetos.

Por todo lo demás, la sintaxis de definición de los indizadores es la misma que la de las propiedades: pueden ser de sólo lectura o de sólo escritura, da igual el orden en que se definan sus bloques **set**/**get**, dentro del bloque **set** se puede acceder al valor a escribir a través del parámetro especial **value** del tipo del indizador, el código del bloque **get** ha de devolver un objeto de dicho tipo, etc.

A continuación se muestra un ejemplo de definición de una clase que consta de dos indizadores: ambos permiten almacenar elementos de tipo entero, pero uno toma como índice un entero y el otro toma dos cadenas:

```
using System;

public class A
{
  public int this[int índice]
  {
    set
    {
      Console.WriteLine("Escrito {0} en posición {1}", value, índice);
    }
    get
    {
      Console.WriteLine("Leído 1 de posición {0}",  índice);
      return 1;
    }
  }

  public int this[string cad1, string cad2]
  {
    set
    {
      Console.WriteLine("Escrito {0} en posición ({1},{2})", value, cad1, cad2);
    }
    get
    {
      Console.WriteLine("Leído prueba de posición ({0},{1})", cad1, cad2);
      return 2;
    }
  }
}
```

Acceso a indizadores

Para acceder a un indizador se utiliza exáctamente la misma sintaxis que para acceder a una tabla, sólo que los índices no tienen porqué ser enteros sino que pueden ser de cualquier tipo de dato que se haya especificado en su definición. Por ejemplo, accesos válidos a los indizadores de un objeto de la clase A definida en el epígrafe anterior

son:

```
A obj = new A();
obj[100] = obj["barco", "coche"];
```

La ejecución de la asignación de este ejemplo producirá esta salida por pantalla:

```
Leído prueba de posición (barco, coche)
Escrito 2 en posición 100
```

Implementación interna de indizadores

Al igual que las propiedades, para facilitar la interoperabilidad entre lenguajes los indizadores son también convertidos por el compilador en llamadas a métodos cuya definición se deduce de la definición del indizador. Ahora los métodos son de la forma:

```
<tipoIndizador> get_Item(<índices>)
{
  <códigoLectura>
}

void set_Item(<índices>, <tipoIndizador> value)
{
  <códigoEscritura>
}
```

Nuevamente, hay que tener cuidado con la signatura de los métodos que se definan en una clase ya que como la de alguno coincida con la generada automáticamente por el compilador para los indizadores se producirá un error de ambigüedad.

Principio Página

El lenguaje de programación C#

En esta página:

Tema 11: Redefinición de operadores

Concepto de redefinición de operador

Un **operador** en C# no es más que un símbolo formado por uno o más caracteres que permite realizar una determinada operación entre uno o más datos y produce un resultado. En el *Tema 4: Aspectos Léxicos* ya hemos visto que C# cuenta con un buen número de operadores que permiten realizar con una sintaxis clara e intuitiva las operaciones comunes a la mayoría de lenguajes (aritmética, lógica, etc) así como otras operaciones más particulares de C# (operador `is`, operador `stackalloc`, etc.)

En C# viene predefinido el comportamiento de sus operadores cuando se aplican a ciertos tipos de datos. Por ejemplo, si se aplica el operador `+` entre dos objetos `int` devuelve su suma, y si se aplica entre dos objetos `string` devuelve su concatenación. Sin embargo, también se permite que el programador pueda definir el significado la mayoría de estos operadores cuando se apliquen a objetos de tipos que él haya definido, y esto es a lo que se le conoce como **redefinición de operador**.

Nótese que en realidad la posibilidad de redefinir un operador no aporta ninguna nueva funcionalidad al lenguaje y sólo se ha incluido en C# para facilitar la legibilidad del código. Por ejemplo, si tenemos una clase Complejo que representa números complejos podríamos definir una función `Sumar()` para sus objetos de modo que a través de ella se pudiese conseguir la suma de dos objetos de esta clase como muestra este ejemplo:

```
Complejo c1 = new Complejo(3,2);          // c1 = 3 + 2i
Complejo c2 = new Complejo(5,2);          // c2 = 5 + 2i
Complejo c3 = c1.Sumar(c2);                 // c3 = 8 + 4i
```

Sin embargo, el código sería mucho más legible e intuitivo si en vez de tenerse que usar el método `Sumar()` se redefiniese el significado del operador `+` para que al aplicarlo entre objetos `Complejo` devolviese su suma. Con ello, el código anterior quedaría así:

```
Complejo c1 = new Complejo(3,2);      // c1 = 3 + 2i
Complejo c2 = new Complejo(5,2);      // c2 = 5 + 2i
Complejo c3 = c1 + c2;             // c3 = 8 + 4i
```

Ésta es precisamente la utilidad de la redefinición de operadores: hacer más claro y legible el código, no hacerlo más corto. Por tanto, cuando se redefina un operador es importante que se le dé un significado intuitivo ya que si no se iría contra de la filosofía de la redefinición de operadores. Por ejemplo, aunque sería posible redefinir el operador `*` para que cuando se aplicase entre objetos de tipo `Complejo` devuelva su suma o imprimiese los valores de sus operandos en la ventana de consola, sería absurdo hacerlo ya que más que clarificar el código lo que haría sería dificultar su comprensión.

De todas formas, suele ser buena idea que cada vez que se redefina un operador en un tipo de dato también se dé una definición de un método que funcione de forma equivalente al operador. Así desde lenguajes que no soporten la redefinición de operadores también podrá realizarse la operación y el tipo será más reutilizable.

Definición de redefiniciones de operadores

Sintaxis general de redefinición de operador

La forma en que se redefine un operador depende del tipo de operador del que se trate, ya que no es lo mismo definir un operador unario que uno binario. Sin embargo, como regla general podemos considerar que se definiendo un método público y estático cuyo nombre sea el símbolo del operador a redefinir y venga precedido de la palabra reservada **`operator`**. Es decir, se sigue una sintaxis de la forma:

```
public static <tipoDevuelto> operator <símbolo>(<operandos>)
{
  <cuerpo>
}
```

Los modificadores **`public`** y **`static`** pueden permutarse si se desea, lo que es importante es que siempre aparezcan en toda redefinición de operador. Se puede redefinir tanto operadores unarios como binarios, y en `<operandos>` se ha de incluir tantos parámetros como operandos pueda tomar el operador a redefinir, ya que cada uno representará a uno de sus operandos. Por último, en `<cuerpo>` se ha de escribir las instrucciones a ejecutar cada vez que se aplique la operación cuyo operador es `<símbolo>` a operandos de los tipos indicados en `<operandos>`

`<tipoDevuelto>` no puede ser **`void`**, pues por definición toda operación tiene un resultado, por lo que todo operador ha de devolver algo. Además, permitirlo complicaría innecesariamente el compilador y éste tendría que admitir instrucciones poco intuitivas (como `a+b;` si el **`+`** estuviese redefinido con valor de retorno **`void`** para los tipos de `a` y `b`)

Además, los operadores no pueden redefinirse con total libertad ya que ello dificultaría inncesariamente la legibilidad del código, por lo que se han introducido las siguientes restricciones al redefinirlos:

Al menos uno de los operandos ha de ser del mismo tipo de dato del que sea miembro la redefinición del operador. Como puede deducirse, ello implica que aunque puedan sobrecargarse los operadores binarios nunca podrá hacerse lo mismo con los binarios ya que su único parámetro sólo puede ser de un único tipo (el tipo dentro del que se defina) Además, ello también provoca que no pueden redefinirse las conversiones ya incluidas en la BCL porque al menos uno de los operandos siempre habrá de ser de algún nuevo tipo definido por el usuario.

No pueden alterarse sus reglas de precedencia, asociatividad, ubicación y número de operandos, pues si

ya de por sí es difícil para muchos recordarlas cuando son fijas, mucho más lo sería si pudiesen modificarse según los tipos de sus operandos.

No pueden definirse nuevos operadores ni combinaciones de los ya existentes con nuevos significados (por ejemplo **** para representar exponenciación), pues ello complicaría innecesariamente el compilador, el lenguaje y la legibilidad del código cuando en realidad es algo que puede simularse definiendo métodos.

No todos los operadores incluidos en el lenguaje pueden redefinirse, pues muchos de ellos (como **.**, **new**, **=**, etc.) son básicos para el lenguaje y su redefinición es inviable, poco útil o dificultaría innecesariamente la legibilidad de los fuentes. Además, no todos los redefinibles se redefinen usando la sintaxis general hasta ahora vista, aunque en su momento se irán explicando cuáles son los redefinibles y cuáles son las peculiaridades de aquellos que requieran una redefinición especial.

A continuación se muestra cómo se redefiniría el significado del operador **+** para los objetos `Complejo` del ejemplo anterior:

```
class Complejo;
{
  public float ParteReal;
  public float ParteImaginaria;

  public Complejo (float parteReal, float parteImaginaria)
  {
    this.ParteReal = parteReal;
    this.ParteImaginaria = parteImaginaria;
  }

  public static Complejo operator +(Complejo op1, Complejo op2)
  {
    Complejo resultado = new Complejo();

    resultado.ParteReal = op1.ParteReal + op2.ParteReal;
    resultado.ParteImaginaria = op1.ParteImaginaria + op2.ParteImaginaria;
    return resultado;
  }
}
```

Es fácil ver que lo que en el ejemplo se ha redefinido es el significado del operador **+** para que cuando se aplique entre dos objetos de clase Complejo devuelva un nuevo objeto Complejo cuyas partes real e imaginaria sea la suma de las de sus operandos.

Se considera erróneo incluir la palabra reservada **new** en la redefinición de un operador, ya que no pueden ocultarse redefiniciones de operadores en tanto que estos no se aplican a usando el nombre del tipo en que estén definidos. Las únicas posibles coincidencias se daría en situaciones como la del siguiente ejemplo:

```
using System;

class A
{
  public static int operator +(A obj1, B obj2)
  {
    Console.WriteLine("Aplicado + de A");
    return 1;
  }
```

```
}

class B:A
{
  public static int operator +(A obj1, B obj2)
  {
    Console.WriteLine("Aplicado + de B");
    return 1;
  }

  public static void Main()
  {
    A o1 = new A();
    B o2 = new B();

    Console.WriteLine("o1+o2={0}", o1+o2);
  }
}
```

Sin embargo, más que una ocultación de operadores lo que se tiene es un problema de ambigüedad en la definición del operador **+** entre objetos de tipos A y B, de la que se informará al compilar ya que el compilador no sabrá cuál versión del operador debe usar para traducir o1+o2 a código binario.

Redefinición de operadores unarios

Los únicos operadores unarios redefinibles son: **!**, **+**, **-**, **~**, **++**, **--**, **true** y **false**, y toda redefinición de un operador unario ha de tomar un único parámetro que ha de ser del mismo tipo que el tipo de dato al que pertenezca la redefinición.

Los operadores **++** y **--** siempre han de redefinirse de manera que el tipo de dato del objeto devuelto sea el mismo que el tipo de dato donde se definen. Cuando se usen de forma prefija se devolverá ese objeto, y cuando se usen de forma postifja el compilador lo que hará será devolver el objeto original que se les pasó como parámetro en lugar del indicado en el **return**. Por ello es importante no modificar dicho parámetro si es de un tipo referencia y queremos que estos operadores tengan su significado tradicional. Un ejemplo de cómo hacerlo es la siguiente redefinición de **++** para el tipo Complejo:

```
public static Complejo operator ++ (Complejo op)
{
  Complejo resultado = new Complejo(op.ParteReal + 1, op.ParteImaginaria);

  return resultado;
}
```

Nótese que si hubiésemos redefinido el **++** de esta otra forma:

```
public static Complejo operator ++ (Complejo op)
{
  op.ParteReal++;

  return op;
}
```

entonces el resultado devuelto al aplicárselo a un objeto siempre sería el mismo tanto si fue aplicado de forma prefija como si lo fue de forma postifija, ya que en ambos casos el objeto devuelto sería el mismo.

Sin embargo, eso no ocurriría si Complejo fuese una estructura, ya que entonces op no sería el objeto original sino una copia de éste y los cambios que se le hiciesen en el cuerpo de la redefinición de ++ no afectarían al objeto original, que es el que se devuelve cuando se usa ++ de manera postfija.

Respecto a los operadores **true** y **false**, estos indican respectivamente, cuando se ha de considerar que un objeto representa el valor lógico cierto y cuando se ha de considerar que representa el valor lógico falso, por lo que su redefiniciones siempre han de devolver un objeto de tipo **bool** que indique dicha situación. Además, si se redefine uno de estos operadores, entonces es obligatorio redefinir también el otro, en tanto que siempre es posible usar indistintamente uno u otro para determinar el valor lógico que un objeto de ese tipo represente.

En realidad los operadores **true** y **false** no pueden usarse directamente en el código fuente, sino que redefinirlos para un tipo de dato es útil porque permitir usar objetos de ese tipo en expresiones condicionales tal y como si de un valor lógico se tratase. Por ejemplo, podemos redefinir estos operadores en el tipo Complejo de modo que consideren cierto a todo complejo distinto de 0 + 0i y falso a 0 + 0i:

```
public static bool operator true(Complejo op)
{
  return (op.ParteReal != 0 || op.ParteImaginaria != 0);
}

public static bool operator false(Complejo op)
{
  return (op.ParteReal == 0 && op.ParteImaginaria == 0);
}
```

Con estas redefiniciones, un código como el que sigue mostraría por pantalla el mensaje Es cierto:

```
Complejo c1 = new Complejo(1, 0);       // c1 = 1 + 0i
if (c1)
  System.Console.WriteLine("Es cierto");
```

Redefinición de operadores binarios

Los operadores binarios redefinibles son **+**, **-**, *****, **/**, **%**, **&**, **|**, **^**, **<<**, **>>**, **==**, **!=**, **>**, **<**, **>=** y **<=** Toda redefinición que se haga de ellos ha de tomar dos parámetros tales que al menos uno de ellos sea del mismo tipo que el tipo de dato del que es miembro la redefinición.

Hay que tener en cuenta que aquellos de estos operadores que tengan complementario siempre han de redefinirse junto con éste. Es decir, siempre que se redefina en un tipo el operador **>** también ha de redefinirse en él el operador **<**, siempre que se redefina **>=** ha de redefinirse **<=**, y siempre que se redefina **==** ha de redefinirse **!=**.

También hay que señalar que, como puede deducirse de la lista de operadores binarios redefinibles dada, no es posible redefinir directamente ni el operador de asignación = ni los operadores compuestos (**+=**, **-=**, etc.) Sin embargo, en el caso de estos últimos dicha redefinición ocurre de manera automática al redefinir su parte "no **=**" Es decir, al redefinir **+** quedará redefinido consecuentemente **+=**, al redefinir ***** lo hará ***=**, etc.

Por otra parte, también cabe señalar que no es posible redefinir diréctamente los operadores **&&** y **||**. Esto se debe a que el compilador los trata de una manera especial que consiste en evaluarlos perezosamente. Sin embargo, es posible simular su redefinición redefiniendo los operadores unarios **true** y **false**, los operadores binarios **&** y **|** y teniendo en cuenta que **&&** y **||** se evalúan así:

- **&&**: Si tenemos una expresión de la forma `x && y`, se aplica primero el operador **false** a `x`. Si devuelve **false**, entonces `x && y` devuelve el resultado de evaluar `x`; y si no, entonces devuelve el resultado de evaluar `x & y`

- **||**: Si tenemos una expresión de la forma `x || y`, se aplica primero el operador **true** a `x`. Si devuelve **true**, se devuelve el resultado de evaluar `x`; y si no, se devuelve el de evaluar `x | y`.

Redefiniciones de operadores de conversión

En el *Tema 4: Aspectos Léxicos* ya vimos que para convertir objetos de un tipo de dato en otro se puede usar un operador de conversión que tiene la siguiente sintaxis:

```
(<tipoDestino>) <expresión>
```

Lo que este operador hace es devolver el objeto resultante de convertir al tipo de dato de nombre `<tipoDestino>` el objeto resultante de evaluar `<expresión>` Para que la conversión pueda aplicarse es preciso que exista alguna definición de cómo se ha de convertir a `<tipoDestino>` los objetos del tipo resultante de evaluar `<expresión>` Esto puede indicarse introduciendo como miembro del tipo de esos objetos o del tipo `<tipoDestino>` una redefinición del operador de conversión que indique cómo hacer la conversión del tipo del resultado de evaluar `<expresión>` a `<tipoDestino>`

Las redefiniciones de operadores de conversión puede ser de dos tipos:

- **Explícitas:** La conversión sólo se realiza cuando se usen explícitamente los operadores de conversión antes comentado.

- **Implícitas:** La conversión también se realiza automáticamente cada vez que se asigne un objeto de ese tipo de dato a un objeto del tipo `<tipoDestino>`. Estas conversiones son más cómodas que las explícitas pero también más peligrosas ya que pueden ocurrir sin que el programador se dé cuenta. Por ello, sólo deberían definirse como implícitas las conversión seguras en las que no se puedan producir excepciones ni perderse información al realizarlas.

En un mismo tipo de dato pueden definirse múltiples conversiones siempre y cuando el tipo origen de las mismas sea diferente. Por tanto, no es válido definir a la vez en un mismo tipo una versión implícita de una cierta conversión y otra explícita.

La sintaxis que se usa para hacer redefinir una operador de conversión es parecida a la usada para cualquier otro operador sólo que no hay que darle nombre, toma un único parámetro y hay que preceder la palabra reservada **operator** con las palabras reservadas **explicit** o **implicit** según se defina la conversión como explícita o implícita. Por ejemplo, para definir una conversión implícita de `Complejo` a **float** podría hacerse:

```
public static implicit operator float(Complejo op)
{
  return op.ParteReal;
}
```

Nótese que el tipo del parámetro usado al definir la conversión se corresponde con el tipo de dato del objeto al que se puede aplicar la conversión (**tipo origen**), mientras que el tipo del valor devuelto será el tipo al que se realice la conversión (**tipo destino**) Con esta definición podrían escribirse códigos como el siguiente:

```
Complejo c1 = new Complejo(5,2);            // c1 = 5 + 2i
float f = c1;                                          // f = 5
```

Nótese que en la conversión de Complejo a **float** se pierde información (la parte imaginaria), por lo que sería mejor definir la conversión como explícita sustituyendo en su definición la palabra reservada **implicit** por **explicit**. En ese caso, el código anterior habría de cambiarse por:

```
Complejo c1 = new Complejo(5,2);            // c1 =  5 + 2i
float f = (float) c1;                                  // f = 5
```

Por otro lado, si lo que hacemos es redefinir la conversión de **float** a Complejo con:

```
public static implicit operator Complejo(float op)
{
  return (new Complejo(op, 0));
}
```

Entonces se podría crear objetos Complejo así:

```
Complejo c2 = 5;   // c2  = 5  + 0i
```

Véase que en este caso nunca se perderá información y la conversión nunca fallará, por lo que es perfectamente válido definirla como implícita. Además, nótese como redefiniendo conversiones implícitas puede conseguirse que los tipos definidos por el usuario puedan inicializarse directamente a partir de valores literales tal y como si fuesen tipos básicos del lenguaje.

En realidad, cuando se definan conversiones no tiene porqués siempre ocurrir que eltipo destino indicado sea el tipo del que sea miembro la redefinición, sino que sólo ha de cumplirse que o el tipo destino o el tipo origen sean de dicho tipo. O sea, dentro de un tipo de dato sólo pueden definirse conversiones de ese tipo a otro o de otro tipo a ese. Sin embargo, al permitirse conversiones en ambos sentidos hay que tener cuidado porque ello puede producir problemas si se solicitan conversiones para las que exista una definición de cómo realizarlas en el tipo fuente y otra en el tipo destino. Por ejemplo, el siguiente código provoca un error al compilar debido a ello:

```
class A
{
  static void Main(string[] args)
  {
    A obj = new B();  // Error: Conversión de B en A ambigua
  }

  public static  implicit operator A(B obj)
  {
    return new A();
  }
}

class B
{
  public static implicit operator A(B obj)
  {
    return new A();
  }
```

```
}
```

El problema de este tipo de errores es que puede resulta difícil descubrir sus causas en tanto que el mensaje que el compilador emite indica que no se pueden convertir los objetos A en objetos B pero no aclara que ello se deba a una ambigüedad.

Otro error con el que hay que tener cuidado es con el hecho de que puede ocurrir que al mezclar redefiniciones implícitas con métodos sobrecargados puedan haber ambiguedades al determinar a qué versión del método se ha de llamar. Por ejemplo, dado el código:

```
using System;

class A
{

  public static  implicit operator A(B obj)
  {
    return new A();
  }

  public static void MétodoSobrecargado(A o)
  {
    Console.WriteLine("Versión que toma A");
  }

  public static void MétodoSobrecargado(C o)
  {
    Console.WriteLine("Versión que toma C");
  }

  static void Main(string[] args)
  {
    MétodoSobrecargado(new B());
  }
}

class B
{
  public static implicit operator C(B obj)
  {
    return new C();
  }
}

class C
{}
```

Al compilarlo se producirá un error debido a que en la llamada a `MétodoSobrecargado()` el compilador no puede deducir a qué versión del método se desea llamar ya que existen conversiones implíctas de objetos de tipo B en cualquiera de los tipos admitidos por sus distintas versiones. Para resolverlo lo mejor especificar explícitamente en la llamada la conversión a aplicar usando el operador **()** Por ejemplo, para usar usar la versión del método que toma como parámetro un objeto de tipo A se podría hacer:

```
MétodoSobrecargado ( (A) new B());
```

Sin embargo, hay que tener cuidado ya que si en vez del código anterior se tuviese:

```
class A
{
  public static  implicit operator A(B obj)
  {
    return new A();
  }

  public static void MétodoSobrecargado(A o)
  {
    Console.WriteLine("Versión que toma A");
  }

  public static void MétodoSobrecargado(C o)
  {
    Console.WriteLine("Versión que toma C");
  }

  static void Main(string[] args)
  {
    MétodoSobrecargado(new B());
  }
}

class B
{
  public static implicit operator A(B obj)
  {
    return new A();
  }

  public static implicit operator C(B obj)
  {
    return new C();
  }
}

class C
{}
```

Entonces el fuente compilaría con normalidad y al ejecutarlo se mostraría el siguiente mensaje que demuestra que se ha usado la versión del método que toma un objeto C.

```
Versión que toma C
```

Finalmente, hay que señalar que no es posible definir cualquier tipo de conversión, sino que aquellas para los que ya exista un mecanismo predefinido en el lenguaje no son válidas. Es decir, no pueden definirse conversiones entre un tipo y sus antecesores (por el polimorfismo ya existen), ni entre un tipo y él mismo, ni entre tipos e interfaces por ellos implementadas (las interfaces se explicarán en el *Tema 16: Interfaces*)

Principio Página

El lenguaje de programación C#

En esta página:

Tema 12: Delegados y eventos

Concepto de delegado

Un **delegado** es un tipo especial de clase cuyos objetos pueden almacenar referencias a uno o más métodos de tal manera que a través del objeto sea posible solicitar la ejecución en cadena de todos ellos.

Los delegados son muy útiles ya que permiten disponer de objetos cuyos métodos puedan ser modificados dinámicamente durante la ejecución de un programa. De hecho, son el mecanismo básico en el que se basa la escritura de aplicaciones de ventanas en la plataforma .NET. Por ejemplo, si en los objetos de una clase `Button` que represente a los botones estándar de Windows definimos un campo de tipo delegado, podemos conseguir que cada botón que se cree ejecute un código diferente al ser pulsado sin más que almacenar el código a ejecutar por cada botón en su campo de tipo delegado y luego solicitar la ejecución todo este código almacenado cada vez que se pulse el botón.

Sin embargo, también son útiles para muchísimas otras cosas tales como asociación de código a la carga y descarga de ensamblados, a cambios en bases de datos, a cambios en el sistema de archivos, a la finalización de operaciones asíncronas, la ordenación de conjuntos de elementos, etc. En general, son útiles en todos aquellos casos en que interese pasar métodos como parámetros de otros métodos.

Además, los delegados proporcionan un mecanismos mediante el cual unos objetos pueden solicitar a otros que se les notifique cuando ocurran ciertos sucesos. Para ello, bastaría seguir el patrón consistente en hacer que los objetos notificadores dispongan de algún campo de tipo delegado y hacer que los objetos interesados almacenen métodos suyos en dichos campos de modo que cuando ocurra el suceso apropiado el objeto notificador simule la notificación ejecutando todos los métodos así asociados a él.

.

Definición de delegados

Un delegado no es más que un tipo especial de subclase **System.MulticastDelegate** . Sin embargo, para definir estas clases no se puede utilizar el mecanismo de herencia normal sino que ha de seguirse la siguiente sintaxis especial:

```
<modificadores> delegate <tipoRetorno> <nombreDelegado> (<parámetros>);
```

<nombreDelegado> será el nombre de la clase delegado que se define, mientras que <tipoRetorno> y <parámetros> se corresponderán, respectivamente, con el tipo del valor de retorno y la lista de parámetros de los métodos cuyos códigos puede almacenar en su interior los objetos de ese tipo delegado (**objetos delegados**)

Un ejemplo de cómo definir un delegado de nombre Deleg cuyos objetos puedan almacenar métodos que devuelvan un **string** y tomen como parámetro un **int** es:

```
delegate string Deleg(int valor);
```

Los objetos delegados de este tipo sólo podrán almacenar códigos de métodos que no devuelvan nada y tomen un único parámetro de tipo int. Cualquier intento de almacenar métodos con otras características producirá un error de compilación o, si no puede detectarse al compilar, una excepción de tipo **System.ArgumentNullException** en tiempo de ejecución tal y como muestra el siguiente programa de ejemplo:

```
using System;
using System.Reflection;

public delegate void D();

public class ComprobaciónDelegados
{
  public static void Main()
  {
    Type t = typeof(ComprobaciónDelegados);
    MethodInfo m = t.GetMethod("Método1");
    D obj  = (D) Delegate.CreateDelegate(typeof(D), m);
    obj();
  }

  public static void Método1()
  { Console.WriteLine("Ejecutado Método1"); }

  public static void Método2(string s)
  { Console.WriteLine("Ejecutado Método2"); }
}
```

Lo que se hace en el método Main() de este programa es crear a partir del objeto **Type** que representa al tipo ComprobaciónDelegados un objeto **System.Reflection.MethodInfo** que representa a su método Método1. Como se ve, para crear el objeto **Type** se utiliza el operador **typeof** ya estudiado, y para obtener el objeto **MethodInfo** se usa su método **GetMethod()** que toma como parámetro una cadena con el nombre del método cuyo **MethodInfo** desee obtenerse. Una vez conseguido, se crea un objeto delegado de tipo D que almacene una referencia al método por él representado a través del método **CreateDelegate()** de la clase **Delegate** y se llama dicho objeto, lo que muestra el mensaje:

```
Ejecutado Método1
```

Aunque en vez de obtener el **MethodInfo** que representa al Método1 se hubiese obtenido el que representa al Método2 el compilador no detectaría nada raro al compilar ya que no es lo bastante inteligente como para saber que dicho objeto no representa a un método almacenable en objetos delegados de tipo D. Sin embargo, al ejecutarse la aplicación el CLR sí que lo detectaría y ello provocaría una **ArgumentNullException**

Ésto es un diferencia importante de los delegados respecto a los punteros a función de C/C++ (que también pueden almacenar referencias a métodos), ya que con estos últimos no se realizan dichas comprobaciones en tiempo de ejecución y puede terminar ocurriendo que un puntero a función apunte a un método cuya signatura o valor de retorno no se correspondan con los indicados en su definición, lo que puede ocasionar que el programa falle por causas difíciles de detectar.

Las definiciones de delegados también pueden incluir cualquiera de los modificadores de accesibilidad válidos para una clase, ya que al fin y al cabo los delegados son clases. Es decir, todos pueden incluir los modificadores **public** e **internal**, y los se definan dentro de otro tipo también pueden incluir **protected**, **private** y **protected internal**.

Manipulación de objetos delegados

Un objeto de un tipo delegado se crea exactamente igual que un objeto de cualquier clase sólo que en su constructor ha de pasársele el nombre del método cuyo código almacenará. Este método puede tanto ser un método estático como uno no estático. En el primer caso se indicaría su nombre con la sintaxis <nombreTipo>.<nombreMétodo>, y en el segundo se indicaría con <objeto>.<nombreMétodo>

Para llamar al código almacenado en el delegado se usa una sintaxis similar a la de las llamadas a métodos, sólo que no hay que prefijar el objeto delegado de ningún nombre de tipo o de objeto y se usa simplemente <objetoDelegado>(<valoresParámetros>)

El siguiente ejemplo muestra cómo crear un objeto delegado de tipo D, asociarle el código de un método llamado F y ejecutar dicho código a través del objeto delegado:

```
using System;

delegate void D(int valor);

class EjemploDelegado
{
  public static void Main()
  {
    D objDelegado = new D(F);
    objDelegado(3);
  }

  public static void F(int x)
  {
    Console.WriteLine( "Pasado valor {0} a F()");
  }
}
```

La ejecución de este programa producirá la siguiente salida por pantalla:

```
Pasado valor 3 a F()
```

Nótese que para asociar el código de F() al delegado no se ha indicado el nombre de este método estático con la sintaxis <nombreTipo>.<nombreMétodo> antes comentada. Esto se debe a que no es necesario incluir el <nombreTipo>. cuando el método a asociar a un delegado es estático y está definido en el mismo tipo que el código donde es asociado

En realidad un objeto delegado puede almacenar códigos de múltiples métodos tanto estáticos como no estáticos de manera que una llamada a través suya produzca la ejecución en cadena de todos ellos en el mismo orden en que se almacenaron en él. Nótese que si los métodos devuelven algún valor, tras la ejecución de la cadena de llamadas sólo se devolverá el valor de retorno de la última llamada.

Además, cuando se realiza una llamada a través de un objeto delegado no se tienen en cuenta los modificadores de visibilidad de los métodos que se ejecutarán, lo que permite llamar desde un tipo a métodos privados de otros tipos que estén almacenados en un delegado por accesible desde el primero tal y como muestra el siguiente ejemplo:

```
using System;

public delegate void D();

class A
{
  public static D obj;

  public static void Main()
  {
    B.AlmacenaPrivado();
    obj();
  }
}

class B
{
  private static void Privado()
  { Console.WriteLine("Llamado a método privado"); }

  public static void AlmacenaPrivado()
  { A.obj += new D(Privado); }
}
```

La llamada a `AlmacenaPrivado` en el método `Main()` de la clase `A` provoca que en el campo delegado obj de dicha clase se almacene una referencia al método privado `Privado()` de la clase `B`, y la instrucción siguiente provoca la llamada a dicho método privado desde una clase externa a la de su definición como demuestra la salida del programa:

```
Llamado a método privado
```

Para añadir nuevos métodos a un objeto delegado se le aplica el operador **+=** pasándole como operando derecho un objeto delegado de su mismo tipo (no vale de otro aunque admita los mismos tipos de parámetros y valor de retorno) que contenga los métodos a añadirle, y para quitárselos se hace lo mismo pero con el operador **-=**. Por ejemplo, el siguiente código muestra los efectos de ambos operadores:

```
using System;

delegate void D(int valor);

class EjemploDelegado
{
  public string Nombre;

  EjemploDelegado(string nombre)
  {
    Nombre = nombre;
  }
```

```
  public static void Main()
  {
    EjemploDelegado obj1 += new EjemploDelegado("obj1");
    D objDelegado = new D(f);
    objDelegado +=  new D(obj1.g);
    objDelegado(3);
    objDelegado -= new D(obj1.g);
    objDelegado(5);
  }

  public void g(int x)
  {
    Console.WriteLine("Pasado valor {0} a g() en objeto {1}", x, Nombre);
  }

  public static void f(int x)
  {
    Console.WriteLine( "Pasado valor {0} a f()", x);
  }
}
```

La salida producida por pantalla por este programa será:

```
Pasado valor 3 a f()
Pasado valor 3 a g() en objeto obj1
Pasado valor 5 a f()
```

Como se ve, cuando ahora se hace la llamada `objDelegado(3)` se ejecutan los códigos de los dos métodos almacenados en `objDelegado`, y al quitársele luego uno de estos códigos la siguiente llamada sólo ejecuta el código del que queda. Nótese además en el ejemplo como la redefinición de **+** realizada para los delegados permite que se pueda inicializar `objDelegado` usando **+=** en vez de **=**. Es decir, si uno de los operandos de **+** vale **null** no se produce ninguna excepción, sino que tan sólo no se añade ningún método al otro.

Hay que señalar que un objeto delegado vale **null** si no tiene ningún método asociado, ya sea porque no se ha llamado aún a su constructor o porque los que tuviese asociado se le hayan quitado con **-=**. Así, si al `Main()` del ejemplo anterior le añadimos al final:

```
objDelegado -= new D(f);
objDelegado(6);
```

Se producirá al ejecutarlo una excepción de tipo **System.NullReferenceException** indicando que se ha intentado acceder a una referencia nula.

También hay que señalar que para que el operador **-=** funcione se le ha de pasar como operador derecho un objeto delegado que almacene algún método exactamente igual al método que se le quiera quitar al objeto delegado de su lado izquierdo. Por ejemplo, si se le quiere quitar un método de un cierto objeto, se le ha de pasar un objeto delegado que almacene ese método de ese mismo objeto, y no vale que almacene ese método pero de otro objeto de su mismo tipo. Por ejemplo, si al Main() anterior le añadimos al final:

```
objDelegado -= new g(obj1.g);
objDelegado(6);
```

Entonces no se producirá ninguna excepción ya que el **-=** no eliminará ningún método de `objDelegado` debido a que ese objeto delegado no contiene ningún método `g()` procedente del objeto `obj1`. Es más, la salida que se producirá por pantalla será:

```
Pasado valor 3 a f()
Pasado valor 3 a g() en objeto obj1
Pasado valor 5 a f()
Pasado valor 6 a f()
```

La clase System.MulticastDelegate

Ya se ha dicho que la sintaxis especial de definición de delegados no es más que una forma especial definir subclases de **System.MulticastDelegate**. Esta clase a su vez deriva de **System.Delegate**, que representa a objetos delegados que sólo puede almacenar un único método. Por tanto, todos los objetos delegado que se definan contarán con los siguientes miembros comunes heredados de estas clases:

- **object Target**: Propiedad de sólo lectura que almacena el objeto al que pertenece el último método añadido al objeto delegado. Si es un método de clase vale **null**.

- **MethodInfo Method**: Propiedad de sólo lectura que almacena un objeto **System.Reflection.MethodInfo** con información sobre el último método añadido al objeto (nombre, modificadores, etc.) Para saber cómo acceder a estos datos puede consultar la documentación incluida en el SDK sobre la clase **MethodInfo**

- **Delegate[] getInvocationList()**: Permite acceder a todos los métodos almacenados en un delegado, ya que devuelve una tabla cuyos elementos son delegados cada uno de los cuales almacenan uno, y sólo uno, de los métodos del original. Estos delegados se encuentran ordenados en la tabla en el mismo orden en que sus métodos fueron fue almacenados en el objeto delegado original.

  Este método es especialmente útil porque a través de la tabla que retorna se pueden hacer cosas tales como ejecutar los métodos del delegado en un orden diferente al de su almacenamiento, procesar los valores de retorno de todas las llamadas a los métodos del delegado original, evitar que una excepción en la ejecución de uno de los métodos impida la ejecución de los demás, etc.

Aparte de estos métodos de objeto, la clase **System.MulticastDelegate** también cuenta con los siguientes métodos de tipo de uso frecuente:

- **static Delegate Combine(Delegate fuente, Delegate destino)**: Devuelve un nuevo objeto delegado que almacena la concatenación de los métodos de fuente con los de destino. Por tanto, nótese que estas tres instrucciones son equivalentes:

  ```
  objDelegado += new D(obj1.g);
  objDelegado = objDelegado + new D(obj1.g);
  objDelegado = (D) MulticastDelegate.Combine(objDelegado, new D(obj1.g);
  ```

  Es más, en realidad el compilador de C# lo que hace es convertir toda aplicación del operador **+** entre delegados en una llamada a **Combine()** como la mostrada.

  Hay que tener cuidado con los tipos de los delegados a combinar ya que han de ser exáctamente los mismos o si no se lanza una **System.ArgumentException**, y ello ocurre aún en el caso de que dichos sólo se diferencien en su nombre y no en sus tipos de parámetros y valor de retorno.

- **static Delegate Combine(Delegate[] tabla)**: Devuelve un nuevo delegado cuyos métodos almacenados son la concatenación de todos los de la lista que se le pasa como parámetro y en el orden en que apareciesen en ella. Es una buena forma de crear delegados con muchos métodos sin tener que aplicar **+=** varias veces.

Todos los objetos delegados de la tabla han de ser del mismo tipo, pues si no se produciría una

System.ArgumentException.

- **static Delegate Remove(Delegate original, Delegate aBorrar)**: Devuelve un nuevo delegado cuyos métodos almacenados son el resultado de eliminar de original los que tuviese aBorrar. Por tanto, estas instrucciones son equivalentes:

```
objDelegado -= new D(obj1.g);
objDelegado - objDelegado - new D(obj1.g);
objDelegado = (D) MulticastDelegate.Remove(objDelegado, new D(obj1.g);
```

  Nuevamente, lo que hace el compilador de C# es convertir toda aplicación del operador **-** entre delegados en una llamada a **Remove()** como la mostrada. Por tanto, al igual que con **-=**, para borrar métodos de objeto se ha de especificar en aBorrar un objeto delegado que contenga referencias a métodos asociados a exáctamente los mismos objetos que los almacenados en original.

- **static Delegate CreateDelegate (Type tipo, MehodInfo método)**: Ya se usó este método en el ejemplo de comprobación de tipos del epígrafe *"Definición de delegados"* de este mismo tema. Como recordará pemrite crear dinámicamente objetos delegados, ya que devuelve un objeto delegado del tipo indicado que almacena una referencia al método representado por su segundo parámetro.

Llamadas asíncronas

La forma de llamar a métodos que hasta ahora se ha explicado realiza la llamada de manera **síncrona**, lo que significa que la instrucción siguiente a la llamada no se ejecuta hasta que no finalice el método llamado. Sin embargo, a todo método almacenado en un objeto delegado también es posible llamarde manera **asíncrona** a través de los métodos del mismo, lo que consiste en que no se espera a que acabe de ejecutarse para pasar a la instrucción siguiente a su llamada sino que su ejecución se deja en manos de un hilo aparte que se irá ejecutándolo en paralelo con el hilo llamante.

Por tanto los delegados proporcionan un cómodo mecanismo para ejecutar cualquier método asíncronamente, pues para ello basta introducirlo en un objeto delegado del tipo apropiado. Sin embargo, este mecanismo de llamada asíncrona tiene una limitación, y es que sólo es válido para objetos delegados que almacenen un único método.

Para hacer posible la llamadas asíncronas, aparte de los métodos heredados de **System.MulticastDelegate** todo objeto delegado cuenta con estos otros dos métodos que el compilador define a su medida en al clase en que traduce la definición de su tipo:

```
IAsyncResult BeginInvoke( <parámetros> , AsyncCallback cb,  Object o)
<tipoRetorno> EndInvoke(<parámetrosRefOut>, IASyncResult ar)
```

BeginInvoke() crea un hilo que ejecutará los métodos almacenados en el objeto delegado sobre el que se aplica con los parámetros indicados en *<parámetros>* y devuelve un objeto **IAsyncResult** que almacenará información relativa a ese hilo (por ejemplo, a través de su propiedad de sólo lectura **bool IsComplete** puede consultarse si ha terminado su labor) Sólo tiene sentido llamarlo si el objeto delegado sobre el que se aplica almacena un único método, pues si no se lanza una **System.ArgumentException**.

El parámetro cb de **BeginInvoke()** es un objeto de tipo delegado que puede almacenar métodos a ejecutar cuando el hilo antes comentado finalice su trabajo. A estos métodos el CLR les pasará automáticamente como parámetro el **IAsyncResult** devuelto por **BeginInvoke()**, estándo así definido el delegado destinado a almacenarlos:

```
public delegate void ASyncCallback(IASyncResult obj);
```

Por su parte, el parámetro o de **BeginInvoke** puede usarse para almacenar cualquier información adicional

que se considere oportuna. Es posible acceder a él a través de la propiedad **object AsyncState** del objeto **IAsyncResult** devuelto por **BeginInvoke()**

En caso de que no se desee ejecutar ningún código especial al finalizar el hilo de ejecución asíncrona o no desee usar información adicional, puede darse sin ningún tipo de problema el valor **null** a los últimos parámetros de **BeginInvoke()** según corresponda.

Finalmente, **EndInvoke()** se usa para recoger los resultados de la ejecución asíncrona de los métodos iniciada a través **BeginInvoke()** Por ello, su valor de retorno es del mismo tipo que los métodos almacenables en el objeto delegado al que pertenece y en <parámetrosRefOut> se indican los parámetros de salida y por referencia de dichos métodos. Su tercer parámetro es el **IAsyncResult** devuelto por el **BeginInvoke()** que creó el hilo cuyos se solicita recoger y se usa precisamente para identificarlo. En caso de que ese hilo no haya terminado aún de hacer las llamadas se esperará a que lo haga.

Para ilustrar mejor el concepto de llamadas asíncronas, el siguiente ejemplo muestra cómo encapsular en un objeto delegado un método F() para ejecutarlo asíncronamente:

```
D objDelegado =  new  D (F);
IAsyncResult hilo = objDelegado.BeginInvoke(3, new AsyncCallback(M),
                                            "prueba");
// ... Hacer cosas
objDelegado.EndInvoke(hilo);
```

Donde el método M ha sido definido en la misma clase que este código así:

```
public static void M(IAsyncResult obj)
{
  Console.WriteLine("Llamado a M() con {0}", obj.AsyncState);
}
```

Si entre el **BeginInvoke()** y el **EndInvoke()** no hubiese habido ninguna escritura en pantalla, la salida del fragmento de código anterior sería:

```
Pasado valor 3 a F()
Llamado a M() con prueba
```

La llamada a **BeginInvoke()** lanzará un hilo que ejecutará el método F() almacenado en objDelegado, pero mientras tanto también seguirá ejecutándose el código del hilo desde donde se llamó a **BeginInvoke()** Sólo tras llamar a **EndInvoke()** se puede asegurar que se habrá ejecutado el código de F(), pues mientras tanto la evolución de ambos hilos es prácticamente indeterminable ya que depende del cómo actúe el planificador de hilos.

Aún si el hilo llamador modifica el valor de alguno de los parámetros de salida o por referencia de tipos valor, el valor actualizado de éstos no será visible para el hilo llamante hasta no llamar a **EndInvoke()** Sin embargo, el valor de los parámetros de tipos referencia sí que podría serlo. Por ejemplo, dado un código como:

```
int x=0;
Persona p = new Persona("Josan", "7361928-E", 22);

IAsyncResult res  = objetoDelegado.BeginInvoke(ref x, p, null, null);
// Hacer cosas...
objetoDelegado.EndInvoke(ref x, res);
```

Si en un punto del código comentado con // Hacer cosas... donde el hilo asíncrono ya hubiese modificado los contenidos de x y p se intentase leer los valores de estas variables sólo se leería el valor actualizado de p, mientras el de x no se vería hasta después de la llamada a **EndInvoke()**

Por otro lado, hay que señalar que si durante la ejecución asíncrona de un método se produce alguna excepción, ésta no sería notificada pero provocaría que el hilo asíncrono abortase. Si posteriormente se llamase a **EndInvoke()** con el **IAsyncResult** asociado a dicho hilo, se relanzaría la excepción que produjo el aborto y entonces podría tratarse.

Para optimizar las llamadas asíncronas es recomendable marcar con el atributo **OneWay** definido en **System.Runtime.Remoting.Messaging** los métodos cuyo valor de retorno y valores de parámetros de salida no nos importen, pues ello indica a la infraestructura encargada de hacer las llamadas asíncronas que no ha de considerar. Por ejemplo:

```
[OneWay] public void Método()
{}
```

Ahora bien, hay que tener en cuenta que hacer esto implica perder toda posibilidad de tratar las excepciones que pudiese producirse al ejecutar asíncronamente el método atribuido, pues con ello llamar a **EndInvoke()** dejaría de relanzar la excepción producida.

Por último, a modo de resumen a continuación se indican cuáles son los patrones que pueden seguirse para recoger los resultados de una llamada asíncrona:

- Detectar si la llamada asíncrona ha finalizado mirando el valor de la propiedad **IsComplete** del objeto **IAsyncResult** devuelto por **BeginInvoke()** Cuando sea así, con **EndInvoke()** puede recogerse sus resultados.
- Pasar un objeto delegado en el penúltimo parámetro de **BeginInvoke()** con el método a ejecutar cuando finalice el hilo asíncrono, lo que liberaría al hilo llamante de la tarea de tener que andar mirando si ha finalizado o no.

  Si desde dicho método se necesitase acceder a los resultados del método llamado podría accederse a ellos a través de la propiedad **AsyncDelegate** del objeto **IAsyncResult** que recibe. Esta propiedad contiene el objeto delegado al que se llamó, aunque se muestra a continuación antes de acceder a ella hay que convertir el parámetro **IAsyncResult** de ese método en un **AsyncResult**:

```
public static void M(IAsyncResult iar)
{
  D objetoDelegado = (D) ((AsyncResult iar)).AsyncDelegate;

  // A partir de aquí podría llamarse a EndInvoke() a
  // través de objetoDelegado
}
```

Implementación interna de los delegados

Cuando hacemos una definición de delegado de la forma:

```
<modificadores> delegate <tipoRetorno> <nombre>(<parámetros>);
```

El compilador internamente la transforma en una definición de clase de la forma:

```
<modificadores> class <nombre>:System.MulticastDelegate
{
  private object _target;
  private int _methodPtr;
  private MulticastDelegate _prev;

  public <nombre>(object objetivo, int punteroMétodo)
```

```
    public virtual <tipoRetorno> Invoke(<parámetros>)

    public virtual IAsyncResult BeginInvoke(<parámetros>, AsyncCallback cb,
                                            Object o)

    public virtual <tipoRetorno> EndInvoke(<parámetrosRefOut>,IASyncResult ar)
  }
```

Lo primero que llama la atención al leer la definición de esta clase es que su constructor no se parece en absoluto al que hemos estado usando hasta ahora para crear objetos delegado. Esto se debe a que en realidad, a partir de los datos especificados en la forma de usar el constructor que el programador utiliza, el compilador es capaz de determinar los valores apropiados para los parámetros del verdadero constructor, que son:

- **object objetivo** contiene el objeto al cual pertenece el método especificado, y su valor se guarda en el campo `_target`. Si es un método estático almacena `null`.
- **int punteroMétodo** contiene un entero que permite al compilador determinar cuál es el método del objeto al que se desea llamar, y su valor se guarda en el campo `_methodPtr`. Según donde se haya definido dicho método, el valor de este parámetro procederá de las tablas `MethodDef` o `MethodRef` de los metadatos.

El campo privado `_prev` de un delegado almacena una referencia al delegado previo al mismo en la cadena de métodos. En realidad, en un objeto delegado con múltiples métodos lo que se tiene es una cadena de objetos delegados cada uno de los cuales contiene uno de los métodos y una referencia (en `_prev`) a otro objeto delegado que contendrá otro de los métodos de la cadena.

Cuando se crea un objeto delegado con `new` se da el valor `null` a su campo `_prev` para así indicar que no pertenece a una cadena sino que sólo contiene un método. Cuando se combinen dos objetos delegados (con `+` o `Delegate.Combine()`) el campo `_prev` del nuevo objeto delegado creado enlazará a los dos originales; y cuando se eliminen métodos de la cadena (con `-` o `Delegate.Remove()`) se actualizarán los campos `_prev` de la cadena para que salten a los objetos delegados que contenían los métodos eliminados.

Cuando se solicita la ejecución de los métodos almacenados en un delegado de manera asíncrona lo que se hace es llamar al método `Invoke()` del mismo. Por ejemplo, una llamada como esta:

```
objDelegado(49);
```

Es convertida por el compilador en:

```
objDelegado.Invoke(49);
```

Aunque `Invoke()` es un método público, C# no permite que el programador lo llame explícitamente. Sin embargo, otros lenguajes gestionados sí que podrían permitirlo.

El método `Invoke()` se sirve de la información almacenada en `_target`, `_methodPtr` y `_prev`, para determinar a cuál método se ha de llamar y en qué orden se le ha de llamar. Así, la implementación de `Invoke()` será de la forma:

```
public virtual <tipoRetorno> Invoke(<parámetros>)
{
  if (_prev!=null)
    _prev.Invoke(<parámetros>);
```

```
    return _target._methodPtr(<parámetros>);
}
```

Obviamente la sintaxis **_target.methodPtr** no es válida en C#, ya que **_methodPtr** no es un método sino un campo. Sin embargo, se ha escrito así para poner de manifiesto que lo que el compilador hace es generar el código apropiado para llamar al método perteneciente al objeto indicado en **_target** e identificado con el valor de **_methodPtr**

Nótese que la instrucción **if** incluida se usa para asegurar que las llamadas a los métodos de la cadena se hagan en orden: si el objeto delegado no es el último de la cadena. (**_prev!=null**) se llamará antes al método **Invoke()** de su predecesor.

Por último, sólo señalar que, como es lógico, en caso de que los métodos que el objeto delegado pueda almacenar no tengan valor de retorno (éste sea **void**), el cuerpo de **Invoke()** sólo varía en que la palabra reservada **return** es eliminada del mismo.

Eventos

Concepto de evento

Un **evento** es una variante de las propiedades para los campos cuyos tipos sean delegados. Es decir, permiten controlar la forman en que se accede a los campos delegados y dan la posibilidad de asociar código a ejecutar cada vez que se añada o elimine un método de un campo delegado.

Sintaxis básica de definición de delegados

La sintaxis básica de definición de un evento consiste en definirlo como cualquier otro campo con la única peculiaridad de que se le ha de anteponer la palabra reservada **event** al nombre de su tipo (que será un delegado) O sea, se sigue la sintaxis:

```
<modificadores> event <tipoDelegado> <nombreEvento>;
```

Por ejemplo, para definir un evento de nombre Prueba y tipo delegado D se haría:

```
public event D Prueba;
```

También pueden definirse múltiples eventos en una misma línea separando sus nombres mediante comas. Por ejemplo:

```
public event D Prueba1, Prueba2;
```

Desde código ubicado dentro del mismo tipo de dato donde se haya definido el evento se puede usar el evento tal y como si de un campo delegado normal se tratase. Sin embargo, desde código ubicado externamente se imponen una serie de restricciones que permite controlar la forma en que se accede al mismo. Éstas son:

- No se le puede aplicar los métodos heredados de **System.MulticastDelegate**.
- Sólo se le puede aplicar dos operaciones: añadido de métodos con **+=** y eliminación de métodos con **-=**. De este modo se evita que se use sin querer **=** en vez de **+=** ó **-=** y se sustituyan todos los métodos de la lista de métodos del campo delegado por otro que en realidad se le quería añadir o quitar (si ese otro valiese **null**, ello incluso podría provocar una **System.NullReferenceException**)
- No es posible llamar a los métodos almacenados en un campo delegado a través del mismo.

Esto permite controlar la forma en que se les llama, ya que obliga a que la llamada tenga que hacerse a través de algún método público definido en la definición del tipo de dato donde el evento fue definido.

Sintaxis completa de definición de delegados

La verdadera utilidad de un evento es que permite controlar la forma en que se asocian y quitan métodos de los objetos delegados con **+=** y **-=**. Para ello se han de definir con la siguiente sintaxis avanzada:

```
<modificadores> event <tipoDelegado> <nombreEvento>
{
  add
  {
    <códigoAdd>
  }
  remove
  {
    <códigoRemove>
  }
}
```

Con esta sintaxis no pueden definirse varios eventos en una misma línea como ocurría con la básica. Su significado es el siguiente: cuando se asocie un método con **+=** al evento se ejecutará el <códigoAdd>, y cuando se le quite alguno con **-=** se ejecutará el <códigoRemove>. Esta sintaxis es similar a la de los bloques **set**/**get** de las propiedades pero con una importante diferencia: aunque pueden permutarse las secciones **add** y **remove**, es obligatorio incluir siempre a ambas.

La sintaxis básica es en realidad una forma abreviada de usar la avanzada. Así, la definición `public event D Prueba(int valor);` la interpretaría el compilador como:

```
private D prueba

public event D  Prueba
{
  [MethodImpl(MethodImlOptions.Synchronized)]
  add
  {
    prueba = (D) Delegate.Combine(prueba, value);
  }

  [MethodImpl(MethodImlOptions.Synchronized)]
  remove
  {
    prueba = (D) Delegate.Remove(prueba, value);
  }
}
```

Es decir, el compilador definirá un campo delegado privado y códigos para **add** y **remove** que hagan que el uso de **+=** y **-=** sobre el evento tenga el efecto que normalmente tendrían si se aplicasen directamente sobre el campo privado. Como se ve, dentro de estos métodos se puede usar **value** para hacer referencia al operando derecho de los operadores **+=** y **-=**. El atributo **System.Runtime.InteropServices.MethodImpl** que precede a los bloques **add** y **remove** sólo se incluye para asegurar que un cambio de hilo no pueda interrumpir la ejecución de sus códigos asociados.

Las restricciones de uso de eventos desde códigos externos al tipo donde se han definido se deben a que en realidad éstos no son objetos delegados sino que el objeto delegado es el campo privado que internamente

define el compilador. El compilador traduce toda llamada al evento en una llamada al campo delegado. Como este es privado, por eso sólo pueda accederse a él desde código de su propio tipo de dato.

En realidad, el compilador internamente traduce las secciones **add** y **remove** de la definición de un evento en métodos de la forma:

```
void add_<nombreEvento>(<tipoDelegado> value)
void remove_<nombreEvento>(<tipoDelegado> value)
```

Toda aplicación de **+=** y **-=** a un evento no es convertida en una llamada al campo privado sino en una llamada al método **add**/**remove** apropiado, como se puede observar analizando el MSIL de cualquier fuente donde se usen **+=** y **-=** sobre eventos. Además, como estos métodos devuelven **void** ése será el tipo del valor devuelto al aplicar **+=** ó **-=** (y no el objeto asignado), lo que evitará que código externo al tipo donde se haya definido el evento pueda acceder directamente al campo delegado privado.

Si en vez de la sintaxis básica usamos la completa no se definirá automáticamente un campo delegado por cada evento que se defina, por lo que tampoco será posible hacer referencia al mismo desde código ubicado en la misma clase donde se ha definido. Sin embargo ello permite que el programador pueda determinar, a través de secciones **add** y **remove**, cómo se almacenarán los métodos. Por ejemplo, para ahorrar memoria se puede optar por usar un diccionario donde almacenar los métodos asociados a varios eventos de un mismo objeto en lugar de usar un objeto delegado por cada uno.

Dado que las secciones **add** y **remove** se traducen como métodos, los eventos también podrán participar en el mecanismo de herencia y redefiniciones típico de los métodos. Es decir, en <modificadores> aparte de modificadores de acceso y el modificador **static**, también se podrán incluir los modificadores relativos a herencia. En este sentido hay que precisar algo: un evento definido como **abstract** ha de definirse siempre con la sintaxis básica (no incluirá secciones **add** o **remove**)

Principio Página

El lenguaje de programación C#

En esta página:

Tema 13: Estructuras

Concepto de estructura

Una **estructura** es un tipo especial de clase pensada para representar objetos ligeros. Es decir, que ocupen poca memoria y deban ser manipulados con velocidad, como objetos que representen puntos, fechas, etc. Ejemplos de estructuras incluidas en la BCL son la mayoría de los tipos básicos (excepto `string` y `object`), y de hecho las estructuras junto con la redefinición de operadores son la forma ideal de definir nuevos tipos básicos a los que se apliquen las misma optimizaciones que a los predefinidos.

Diferencias entre clases y estructuras

A diferencia de una clase y fielmente a su espíritu de "ligereza", una estructura no puede derivar de ningún tipo y ningún tipo puede derivar de ella Por estas razones sus miembros no pueden incluir modificadores relativos a herencia, aunque con una excepción: pueden incluir `override` para redefinir los miembros de `System.Object`.

Otra diferencia entre las estructuras y las clases es que sus variables no almacenan referencias a zonas de memoria dinámica donde se encuentran almacenados objetos sino directamente referencian a objetos. Por ello se dice que las clases son **tipos referencia** y las estructuras son **tipos valor**, siendo posible tanto encontrar objetos de estructuras en pila (no son campos de clases) como en memoria dinámica (son campos de clases)

Una primera consecuencia de esto es que los accesos a miembros de objetos de tipos valor son

mucho más rápidos que los accesos a miembros de pilas, ya que es necesario pasar por una referencia menos a la hora de acceder a ellos. Además, el tiempo de creación y destrucción de estructuras también es inferior. De hecho, la destrucción de los objetos almacenados en pila es prácticamente inapreciable ya que se realiza con un simple decremento del puntero de pila y no interviene en ella el recolector de basura.

Otra consecuencia de lo anterior es que cuando se realicen asignaciones entre variables de tipos valor, lo que se va a copiar en la variable destino es el objeto almacenado por la variable fuente y no la dirección de memoria dinámica a la que apuntaba ésta. Por ejemplo, dado el siguiente tipo (nótese que las estructuras se definen igual que las clases pero usando la palabra reservada **struct** en vez de **class**):

```
struct Point
{
  public int x, y;

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

Si usamos este tipo en un código como el siguiente:

```
Punto p = new Punto(10,10);
Punto p2 = p;
p2.x = 100;
Console.WriteLine(p.x);
```

Lo que se mostrará por pantalla será 10. Esto se debe a que el valor de x modificado es el de p2, que es como es una copia de p los cambios que se le hagan no afectarán a p. Sin embargo, si Punto hubiese sido definido como una clase entonces sí que se hubiese mostrado por pantalla 100, ya que en ese caso lo que se habría copiado en p2 habría sido una referencia a la misma dirección de memoria dinámica referenciada por p, por lo que cualquier cambio que se haga en esa zona a través de p2 también afectará a p.

De lo anterior se deduce que la asignación entre objetos de tipos estructuras es mucho más lenta que la asignación entre objetos de clases, ya que se ha de copiar un objeto completo y no solo una referencia. Para aliviar esto al pasar objetos de tipos estructura como parámetros, se da la posibilidad de pasarlos como parámetros por referencia (modificador **ref**) o parámetros de salida (**out**) en vez de como parámetros de entrada.

Todas las estructuras derivan implícitamente del tipo **System.ValueType**, que a su vez deriva de la clase primigenia **System.Object**. **ValueType** tiene los mismos miembros que su padre, y la única diferencia señalable entre ambos es que en **ValueType** se ha redefinido **Equals()** de modo que devuelva **true** si los objetos comparados tienen el mismo valor en todos sus campos y **false** si no. Es decir, la comparación entre estructuras con **Equals()** se realiza por valor.

Respecto a la implementación de la igualdad en los tipos definidos como estructuras, también es importante tener muy en cuenta que el operador **==** no es en principio aplicable a las estructuras que defina el programador. Si se desea que lo tenga ha de dársele explícitamente una redefinición al

definir dichas estructuras.

Boxing y unboxing

Dado que toda estructura deriva de **System.Object**, ha de ser posible a través del polimorfismo almacenar objetos de estos tipos en objetos **object**. Sin embargo, esto no puede hacerse directamente debido a las diferencias semánticas y de almacenamiento que existen entre clases y estructuras: un **object** siempre ha de almacenar una referencia a un objeto en memoria dinámica y una estructura no tiene porqué estarlo. Por ello ha de realizársele antes al objeto de tipo valor una conversión conocida como **boxing**. Recíprocamente, al proceso de conversión de un **object** que contenga un objeto de un tipo valor al tipo valor original se le denomina **unboxing**.

El proceso de boxing es muy sencillo. Consiste en envolver el objeto de tipo valor en un objeto de un tipo referencia creado específicamente para ello. Por ejemplo, para un objeto de un tipo valor T, el tipo referencia creado sería de la forma:

```
class T_Box
{
  T value;
  T_Box(T t)
  {
    value = t;
  }
}
```

En realidad todo esto ocurre de forma transparente al programador, el cual simplemente asigna el objeto de tipo valor a un objeto de tipo referencia como si de cualquier asignación polimórfica se tratase. Por ejemplo:

```
int p = new Punto(10,10);
object o = p; // boxing. Es equivalente a object o = new Punto_Box(p);
```

En realidad la clase envoltorio arriba escrita no se crea nunca, pero conceptualmente es como si se crease. Esto se puede comprobar viendo a través del siguiente código que el verdadero tipo del objeto o del ejemplo anterior sigue siendo `Punto` (y no `Punto_Box`):

```
Console.WriteLine((p is Punto));
```

La salida por pantalla de este código es True, lo que confirma que se sigue considerando que en realidad o almacena un Punto (recuérdese que el operador **is** sólo devuelve **true** si el objeto que se le pasa como operando izquierdo es del tipo que se le indica como operando derecho)

El proceso de unboxing es también transparente al programador. Por ejemplo, para recuperar como `Punto` el valor de tipo `Punto` almacenado en el objeto o anterior se haría:

```
p = (Punto) o;  // Es equivalente a ((Punto_Box) o).value
```

Obviamente durante el unboxing se hará una comprobación de tipo para asegurar que el objeto almacenado en o es realmente de tipo `Punto`. Esta comprobación es tan estricta que se ha de cumplir que el tipo especificado sea exactamente el mismo que el tipo original del objeto, no vale que sea un compatible. Por tanto, este código es inválido:

```
int i = 123;
object o = i;
long l = (long) o // Error: o contiene un int, no un long
```

Sin embargo, lo que si sería válido es hacer:

```
long l = (long) (int) o;
```

Como se puede apreciar en el constructor del tipo envoltorio creado, durante el boxing el envoltorio que se crea recibe una copia del valor del objeto a convertir, por lo que los cambios que se le hagan no afectarán al objeto original. Por ello, la salida del siguiente código será 10:

```
Punto p = new Punto(10,10);
object o = p;      // boxing
p.X = 100;
Console.WriteLine( ((Punto) o).X); // unboxing
```

Sin embargo, si `Punto` se hubiese definido como una clase entonces sí que se mostraría por pantalla un 100 ya que entonces no se haría boxing en la asignación de `p` a `o` sino que se aplicaría el mecanismo de polimorfismo normal, que consiste en tratar `p` a través de `o` como si fuese de tipo `object` pero sin realizarse ninguna conversión.

El problema del boxing y el unboxing es que son procesos lentos, ya que implican la creación y destrucción de objetos envoltorio. Por ello puede interesar evitarlos en aquellas situaciones donde la velocidad de ejecución de la aplicación sea crítica, y para ello se proponen varias técnicas:

- Si el problema se debe al paso de estructuras como parámetros de métodos genéricos que tomen parámetros de tipo **`object`**, puede convenir definir sobrecargas de esos métodos que en lugar de tomar **`object`**s tomen objetos de los tipos estructura que en concreto la aplicación utiliza

- Siguiendo en la línea de lo anterior, puede que interese usar **plantillas** en lugar de tipos genéricos. Éstas no son más que definiciones de tipos de datos en las que no se indica cuál es el tipo exacto de ciertas variables sino que se deja en función de parámetros a los que puede dárseles distintos valores al crear cada objeto de ese tipo. Así, en vez de crearse siempre objetos con métodos que tomen parámetros **`object`**, se podrían ir creando diferentes versiones del tipo según el tipo de estructura con la que se vaya a trabajar.

Actualmente el CLR puede trabajar con plantillas, pero se espera que en versiones futuras del .NET Framework lo haga y el lenguaje C# las incluya.

- Muchas veces conviene hacer unboxing para poder acceder a miembros específicos de ciertas estructuras almacenadas en **`object`**s, aunque a continuación vuelva a necesitarse realmacenar la estructura en un **`object`**. Para evitar esto una posibilidad sería almacenar en el objecto no directamente la estructura sino un objeto de una clase envolvente creada a medida por el programador y que incluya los miembros necesarios para hacer las operaciones anteriores. Así se evitaría tener que hacer unboxing, pues se convertiría de **`object`** a esa clase, que no es un tipo valor y por

tanto no implica unboxing.

- Con la misma idea, otra posibilidad sería que el tipo estructura implementase ciertas interfaces mediante las que se pudiese hacer las operaciones antes comentadas. Aunque las interfaces no se tratarán hasta el *Tema 15: Interfaces*, por ahora basta saber que las interfaces son también tipos referencia y por tanto convertir de **object** a un tipo interfaz no implica unboxing.

Constructores

Los constructores de las estructuras se comportan de una forma distinta a los de las clases. Por un lado, no pueden incluir ningún inicializador base debido a que como no puede haber herencia el compilador siempre sabe que ha de llamar al constructor sin parámetros de **System.ValueType**. Por otro, dentro de su cuerpo no se puede acceder a sus miembros hasta inicializarlos, pues para ahorrar tiempo no se les da ningún valor inicial antes de llamar al constructor.

Sin embargo, la diferencia más importante entre los constructores de ambos tipos se encuentra en la implementación del constructor sin parámetros: como los objetos estructura no puede almacenar el valor por defecto **null** cuando se declaran sin usar constructor ya que ese valor indica referencia a posición de memoria dinámica indeterminada y los objetos estructura no almacenan referencias, toda estructura siempre tiene definido un constructor sin parámetros que lo que hace es darle en esos casos un valor por defecto a los objetos declarados. Ese valor consiste en poner a cero toda la memoria ocupada por el objeto, lo que tiene el efecto de dar como valor a cada campo el cero de su tipo. Por ejemplo, el siguiente código imprime un 0 en pantalla:

```
Punto p = new Punto();
Console.WriteLine(p.X);
```

Y el siguiente también:

```
using System;

struct Punto
{
  public int X,Y;
}

class EjemploConstructorDefecto
{
  Punto p;

  public static void Main()
  {
    Console.WriteLine(p.X);
  }
}
```

Sin embargo, el hecho de que este constructor por defecto se aplique no implica que se pueda acceder a las variables locales sin antes inicializarlas con otro valor. Por ejemplo, el siguiente fragmente de código de un método sería incorrecto:

```
Punto p;
```

```
Console.WriteLine(p.X);  // X no inicializada
```

Sin embrago, como a las estructuras declaradas sin constructor no se les da el valor por defecto **null**, sí que sería válido:

```
Punto p;
p.X = 2;
Console.WriteLine(p.X);
```

Para asegurar un valor por defecto común a todos los objetos estructura, se prohíbe al programador dar una definición propia de su constructor sin parámetros. Mientras que en las clases es opcional implementarlo y si no se hace el compilador introduce uno por defecto, en las estructuras no es válido hacerlo. Además, aún en el caso de que se definan otros constructores, el constructor sin parámetros seguirá siendo introducido automáticamente por el compilador a diferencia de cómo ocurría con las clases donde en ese caso el compilador no lo introducía.

Por otro lado, para conseguir que el valor por defecto de todos los objetos estructuras sea el mismo, se prohíbe darles una valor inicial a sus campos en el momento de declararlos, pues si no el constructor por defecto habría de tenerlos en cuenta y su ejecución sería más ineficiente. Por esta razón, los constructores definidos por el programador para una estructura han de inicializar todos sus miembros no estáticos en tanto que antes de llamarlos no se les da ningún valor inicial.

Nótese que debido a la existencia de un constructor por defecto cuya implementación escapa de manos del programador, el código de los métodos de una estructura puede tener que considerar la posibilidad de que se acceda a ellos con los valores resultantes de una inicialización con ese constructor. Por ejemplo, dado:

```
struct A
{
  public readonly string S;

  public A(string s)
  {
    if (s==null)
      throw (new ArgumentNullException());
    this.s = S;
  }
}
```

Nada asegura que en este código los objetos de clase A siempre se inicialicen con un valor distinto de **null** en su campo S, pues aunque el constructor definido para A comprueba que eso no ocurra lanzando una excepción en caso de que se le pase una cadena que valga **null**, si el programador usa el constructor por defecto creará un objeto en el que S valga **null**. Además, ni siquiera es válido especificar un valor inicial a S en su definición, ya que para inicializar rápidamente las estructuras sus campos no estáticos no pueden tener valores iniciales.

Principio Página

El lenguaje de programación C#

En esta página:

Tema 14: Enumeraciones

Concepto de enumeración

Una **enumeración** o **tipo enumerado** es un tipo especial de estructura en la que los literales de los valores que pueden tomar sus objetos se indican explícitamente al definirla. Por ejemplo, una enumeración de nombre Tamaño cuyos objetos pudiesen tomar los valores literales Pequeño, Mediano o Grande se definiría así:

```
enum Tamaño
{
  Pequeño,
  Mediano,
  Grande
}
```

Para entender bien la principal utilidad de las enumeraciones vamos a ver antes un problema muy típico en programación: si queremos definir un método que pueda imprimir por pantalla un cierto texto con diferentes tamaños, una primera posibilidad sería dotarlo de un parámetro de algún tipo entero que indique el tamaño con el que se desea mostrar el texto. A estos números que los métodos interpretan con significados específicos se les suele denominar **números mágicos**, y su utilización tiene los inconvenientes de que dificulta la legibilidad del código (hay que recordar que significa para el método cada valor del número) y su escritura (hay que recordar qué número ha pasársele al método para que que funcione de una cierta forma)

Una alternativa mejor para el método anterior consiste en definirlo de modo que tome un parámetro de tipo Tamaño para que así el programador usuario no tenga que recordar la correspondencia entre tamaños y números. Véase así como la llamada (2) del ejemplo que sigue es mucho más legible que la (1):

```
obj.MuestraTexto(2);                  // (1)
obj.MuestraTexto(Tamaño.Mediano);     // (2)
```

Además, estos literales no sólo facilitan la escritura y lectura del código sino que también pueden ser usados por

herramientas de documentación, depuradores u otras aplicaciones para sustituir números mágicos y mostrar textos muchos más legibles.

Por otro lado, usar enumeraciones también facilita el mantenimiento del código. Por ejemplo, si el método (1) anterior se hubiese definido de forma que 1 significase tamaño pequeño, 2 mediano y 3 grande, cuando se quisiese incluir un nuevo tamaño intermedio entre pequeño y mediano habría que darle un valor superior a 3 o inferior a 1 ya que los demás estarían cogidos, lo que rompería el orden de menor a mayor entre números y tamaños asociados. Sin embargo, usando una enumeración no importaría mantener el orden relativo y bastaría añadirle un nuevo literal.

Otra ventaja de usar enumeraciones frente a números mágicos es que éstas participan en el mecanismo de comprobación de tipos de C# y el CLR. Así, si un método espera un objeto Tamaño y se le pasa uno de otro tipo enumerado se producirá, según cuando se detecte la incoherencia, un error en compilación o una excepción en ejecución. Sin embargo, si se hubiesen usado números mágicos del mismo tipo en vez de enumeraciones no se habría detectado nada, pues en ambos casos para el compilador y el CLR serían simples números sin ningún significado especial asociado.

Definición de enumeraciones

Ya hemos visto un ejemplo de cómo definir una enumeración. Sin embargo, la sintaxis completa que se puede usar para definirlas es:

```
enum <nombreEnumeración> : <tipoBase>
{
  <literales>
}
```

En realidad una enumeración es un tipo especial de estructura (luego **System.ValueType** será tipo padre de ella) que sólo puede tener como miembros campos públicos constantes y estáticos. Esos campos se indican en <literales>, y como sus modificadores son siempre los mismos no hay que especificarlos (de hecho, es erróneo hacerlo)

El tipo por defecto de las constantes que forman una enumeración es **int**, aunque puede dárseles cualquier otro tipo básico entero (**byte**, **sbyte**, **short**, **ushort**, **uint**, **int**, **long** o **ulong**) indicándolo en <tipoBase> Cuando se haga esto hay que tener muy presente que el compilador de C# sólo admite que se indiquen así los alias de estos tipos básicos, pero no sus nombres reales (**System.Byte**, **System.SByte**, etc.)

Si no se especifica valor inicial para cada constante, el compilador les dará por defecto valores que empiecen desde 0 y se incrementen en una unidad para cada constante según su orden de aparición en la definición de la enumeración. Así, el ejemplo del principio del tema es equivalente ha:

```
enum Tamaño:int
{
  Pequeño = 0,
  Mediano = 1,
  Grande = 2
}
```

Es posible alterar los valores iniciales de cada constante indicándolos explícitamente como en el código recién mostrado. Otra posibilidad es alterar el valor base a partir del cual se va calculando el valor de las siguientes constantes como en este otro ejemplo:

```
enum Tamaño
{
  Pequeño,
  Mediano = 5,
  Grande
}
```

En este último ejemplo el valor asociado a Pequeño será 0, el asociado a Mediano será 5, y el asociado a Grande será 6 ya que como no se le indica explícitamente ningún otro se considera que este valor es el de la constante anterior más 1.

Obviamente, el nombre que se de a cada constante ha de ser diferente al de las demás de su misma enumeración y el valor que se de a cada una ha de estar incluido en el rango de valores admitidos por su tipo base. Sin embargo, nada obliga a que el valor que se de a cada constante tenga que ser diferente al de las demás, y de hecho puede especificarse el valor de una constante en función del valor de otra como muestra este ejemplo:

```
enum Tamaño
{
  Pequeño,
  Mediano = Pequeño,
  Grande = Pequeño + Mediano
}
```

En realidad, lo único que importa es que el valor que se dé a cada literal, si es que se le da alguno explícitamente, sea una expresión constante cuyo resultado se encuentre en el rango admitido por el tipo base de la enumeración y no provoque definiciones circulares. Por ejemplo, la siguiente definición de enumeración es incorrecta ya que en ella los literales Pequeño y Mediano se han definido circularmente:

```
enum TamañoMal
{
  Pequeño = Mediano,
  Mediano = Pequeño,
  Grande
}
```

Nótese que la siguiente definición de enumeración también sería incorrecta ya que en ella el valor de B depende del de A implícitamente (sería el de A más 1):

```
enum EnumMal
{
  A = B,
  B
}
```

Uso de enumeraciones

Las variables de tipos enumerados se definen como cualquier otra variable (sintaxis <nombreTipo> <nombreVariable>) Por ejemplo:

```
Tamaño t;
```

El valor por defecto para un objeto de una enumeración es 0, que puede o no corresponderse con alguno de los literales definidos para ésta. Así, si la t del ejemplo fuese un campo su valor sería Tamaño.Pequeño. También puede dársele otro valor al definirla, como muestra el siguiente ejemplo donde se le da el valor Tamaño.Grande:

```
Tamaño t = Tamaño.Grande;   // Ahora t vale Tamaño.Grande
```

Nótese que a la hora de hacer referencia a los literales de una enumeración se usa la sintaxis <nombreEnumeración>.<nombreLiteral>, como es lógico si tenemos en cuenta que en realidad los literales de una enumeración son constantes publicas y estáticas, pues es la sintaxis que se usa para acceder a ese tipo de miembros. El único sitio donde no es necesario preceder el nombre del literal de <nombreEnumeración>. es en la propia definición de la enumeración, como también ocurre con cualquier constante estática.

En realidad los literales de una enumeración son constantes de tipos enteros y las variables de tipo enumerado

son variables del tipo entero base de la enumeración. Por eso es posible almacenar valores de enumeraciones en variables de tipos enteros y valores de tipos enteros en variables de enumeraciones. Por ejemplo:

```
int i = Tamaño.Pequeño;   // Ahora i vale 0
Tamaño t = (Tamaño) 0;    // Ahora t vale Tamaño.Pequeño (=0)
t = (Tamaño)100;          // Ahora t vale 100, que no se
                          // corresponde con ningún literal
```

Como se ve en el último ejemplo, también es posible darle a una enumeración valores enteros que no se correspondan con ninguno de sus literales.

Dado que los valores de una enumeración son enteros, es posible aplicarles muchos de las operaciones que se pueden aplicar a los mismos: **==**, **!=**, **<**, **>**, **<=**, **>=**, **+**, **•**, **^**, **&**, **|**, **~**, **++**, **--** y **sizeof**. Sin embargo, hay que concretar que los operadores binarios **+** y **-** no pueden aplicarse entre dos operandos de enumeraciones, sino que al menos uno de ellos ha de ser un tipo entero; y que **|**, **&** y **^** sólo pueden aplicarse entre enumeraciones.

La clase System.Enum

Todos los tipos enumerados derivan de **System.Enum**, que deriva de **System.ValueType** y ésta a su vez deriva de la clase primigenia **System.Object**. Aparte de los métodos heredados de estas clases padres y ya estudiados, toda enumeración también dispone de otros métodos heredados de **System.Enum**, los principales de los cuales son:

- **static Type getUnderlyingType(Type enum)**: Devuelve un objeto **System.Type** con información sobre el tipo base de la enumeración representada por el objeto **System.Type** que se le pasa como parámetro.

- **string ToString(string formato)**: Cuando a un objeto de un tipo enumerado se le aplica el método **ToString()** heredado de object lo que se muestra es una cadena con el nombre del literal almacenado en ese objeto. Por ejemplo:

  ```
  Tamaño t = Color.Pequeño;
  Console.WriteLine(t); // Muestra por pantalla la cadena "Pequeño"
  ```

  Como también puede resultar interasante obtener el valor numérico del literal, se ha sobrecargado **System.Enum** el método anterior para que tome como parámetro una cadena que indica cómo se desea mostrar el literal almacenado en el objeto. Si esta cadena es nula, vacía o vale `"G"` muestra el literal como si del método **ToString()** estándar se tratase, pero si vale `"D"` o `"X"` lo que muestra es su valor numérico (en decimal si vale `"D"` y en hexadecimal si vale `"X"`) Por ejemplo:

  ```
  Console.WriteLine(t.ToString("X")); // Muestra 0
  Console.WriteLine(t.ToString("G")); // Muestra Pequeño
  ```

  En realidad, los valores de formato son insensibles a la capitalización y da igual si en vez de `"G"` se usa `"g"` o si en vez de `"X"` se usa `"x"`.

- **static string Format(Type enum, object valorLiteral, string formato)**: Funciona de forma parecida a la sobrecarga de **ToString()** recien vista, sólo que ahora no es necesario disponer de ningún objeto del tipo enumerado cuya representación de literal se desea obtener sino que basta indicar el objeto **Type** que lo representa y el número del literal a obtener. Por ejemplo:

  ```
  Console.Write(Enum.Format(typeof(Tamaño), 0, "G"); // Muestra Pequeño
  ```

  Si el `valorLiteral` indicado no estuviese asociado a ningún literal del tipo enumerador representado por enum, se devolvería una cadena con dicho número. Por el contrario, si hubiesen varios literales en la enumeración con el mismo valor numérico asociado, lo que se devolvería sería el nombre del declarado en

último lugar al definir la enumeración.

static object Parse(Type enum, string nombre, *bool mayusculas?*): Crea un objeto de un tipo enumerado cuyo valor es el correspondiente al literal de nombre asociado nombre. Si la enumeración no tuviese ningún literal con ese nombre se lanzaría una **ArgumentException**, y para determinar cómo se ha de buscar el nombre entre los literales de la enumeración se utiliza el tercer parámetro (es opcional y por defecto vale **false**) que indica si se ha de ignorar la capitalización al buscarlo. Un ejemplo del uso de este método es:

```
Tamaño t = (Tamaño) Enum.Parse(typeof(Tamaño), "Pequeño");
Console.WriteLine(t) // Muestra Pequeño
```

Aparte de crear objetos a partir del nombre del literal que almacenarán, **Parse()** también permite crearlos a partir del valor numérico del mismo. Por ejemplo:

```
Tamaño t = (Tamaño) Enum.Parse(typeof(Tamaño), "0");
Console.WriteLine(t) // Muestra Pequeño
```

En este caso, si el valor indicado no se correspondiese con el de ninguno de los literales de la enumeración no saltaría ninguna excepción, pero el objeto creado no almacenaría ningún literal válido. Por ejemplo:

```
Tamaño t = (Tamaño) Enum.Parse(typeof(Tamaño), "255");
Console.WriteLine(t) // Muestra 255
```

- **static object[] GetValues(Type enum)**: Devuelve una tabla con los valores de todos los literales de la enumeración representada por el objeto **System.Type** que se le pasa como parámetro. Por ejemplo:

```
object[] tabla = Enum.GetValues(typeof(Tamaño));
Console.WriteLine(tabla[0]);  // Muestra 0, pues Pequeño =  0
Console.WriteLine(tabla[1]);  // Muestra 1, pues Mediano = 1
Console.WriteLine(tabla[2]);  // Muestra 1, pues Grande = Pequeño+Mediano
```

- **static string GetName(Type enum, object valor)**: Devuelve una cadena con el nombre del literal de la enumeración representada por enum que tenga el valor especificado en valor. Por ejemplo, este código muestra Pequeño por pantalla:

```
Console.WriteLine(Enum.GetName(typeof(Tamaño), 0)); //Imprime Pequeño
```

Si la enumeración no contiene ningún literal con ese valor devuelve **null**, y si tuviese varios con ese mismo valor devolvería sólo el nombre del último. Si se quiere obtener el de todos es mejor usar **GetNames()**, que se usa como **GetName()** pero devuelve un **string[]** con los nombres de todos los literales que tengan el valor indicado ordenados según su orden de definición en la enumeración.

- **static bool isDefined (Type enum, object valor)**: Devuelve un booleano que indica si algún literal de la enumeración indicada tiene el valor indicado.

Enumeraciones de flags

Muchas veces interesa dar como valores de los literales de una enumeración únicamente valores que sean potencias de dos, pues ello permite que mediante operaciones de bits **&** y **|** se puede tratar los objetos del tipo enumerado como si almacenasen simultáneamente varios literales de su tipo. A este tipo de enumeraciones las llamaremos **enumeraciones de flags**, y un ejemplo de ellas es el siguiente:

```
enum ModificadorArchivo
{
  Lectura = 1,
```

```
    Escritura = 2,
    Oculto = 4,
    Sistema = 8
}
```

Si queremos crear un objeto de este tipo que represente los modificadores de un archivo de lectura-escritura podríamos hacer:

```
ModificadorArchivo obj = ModificadorArchivo.Lectura |
ModificadorArchivo.Escritura
```

El valor del tipo base de la enumeración que se habrá almacenado en obj es 3, que es el resultado de hacer la operación OR entre los bits de los valores de los literales `Lectura` y `Escritura`. Al ser los literales de `ModificadorArchivo` potencias de dos sólo tendrán un único bit a 1 y dicho bit será diferente en cada uno de ellos, por lo que la única forma de generar un 3 (últimos dos bits a 1) combinando literales de `ModificadorArchivo` es combinando los literales `Lectura` (último bit a 1) y `Escritura` (penúltimo bit a 1) Por tanto, el valor de `obj` identificará unívocamente la combinación de dichos literales.

Debido a esta combinabilidad no se debe determinar el valor literal de los objetos `ModificadorArchivo` tal y como si sólo pudiesen almacenar un único literal, pues su valor numérico no tendría porqué corresponderse con el de ningún literal de la enumeración Por ejemplo:

```
bool permisoLectura = (obj == ModificadorArchivo.Lectura); // Almacena false
```

Aunque los permisos representados por obj incluían permiso de lectura, se devuelve **`false`** porque el valor numérico de `obj` es 3 y el del `ModificadorArchivo.Lectura` es 1. Si lo que queremos es comprobar si `obj` contiene permiso de lectura, entonces habrá que usar el operador de bits **`&`** para aislarlo del resto de literales combinados que contiene:

```
bool permisoLectura = (ModificadorArchivo.Lectura ==
                       (obj & ModificadorArchivo.Lectura)); // Almacena true
```

O lo que es lo mismo:

```
bool permisoLectura = ( (obj & ModificadorArchivo.Lectura) != 0);
                                    // Almacena true
```

Del mismo, si directamente se intenta mostrar por pantalla el valor de un objeto de una enumeración que almacene un valor que sea combinación de literales, no se obtendrán el resultado esperado (nombre del literal correspondiente a su valor) Por ejemplo, dado:

```
Console.Write(obj);  // Muestra 3
```

Se mostrará un 3 por pantalla ya que en realidad ningún literal de ModificadorArchivo tiene asociado dicho valor. Como lo natural sería que se desease obtener un mensaje de la forma Lectura, Escritura, los métodos **`ToString()`** y **`Format()`** de las enumeraciones ya vistos admiten un cuarto valor `"F"` para su parámetro formato (su nombre viene de flags) con el que se consigue lo anterior. Por tanto:

```
Console.Write(obj.ToString("F")); // Muestra Lectura, Escritura
```

Esto se debe a que cuando **`Format()`** detecta este indicador (**`ToString()`** también, pues para generar la cadena llama internamente a **`Format()`**) y el literal almacenado en el objeto no se corresponde con ninguno de los de su tipo enumerado, entonces lo que hace es mirar uno por uno los bits a uno del valor numérico asociado de dicho literal y añadirle a la cadena a devolver el nombre de cada literal de la enumeración cuyo valor asociado sólo tenga ese bit a uno, usándo como separador entre nombres un carácter de coma.

Nótese que nada obliga a que los literales del tipo enumerado tengan porqué haberse definido como potencias de dos, aunque es lo más conveniente para que `"F"` sea útil, pues si la enumeración tuviese algún literal con el valor

del objeto de tipo enumerado no se realizaría el proceso anterior y se devolvería sólo el nombre de ese literal.

Por otro lado, si alguno de los bits a 1 del valor numérico del objeto no tuviese el correspondiente literal con sólo ese bit a 1 en la enumeración no se realizaría tampoco el proceso anterior y se devolvería una cadena con dicho valor numérico.

Una posibilidad más cómoda para obtener el mismo efecto que con `"F"` es marcar la definición de la enumeración con el atributo **`Flags`**, con lo que ni siquiera sería necesario indicar formato al llamar a **`ToString()`** O sea, si se define `ModificadorArchivo` así:

```
[Flags]
enum ModificadorArchivo
{
  Lectura = 1,
  Escritura = 2,
  Oculto = 4,
  Sistema = 8
}
```

Entonces la siguiente llamada producirá como salida `Lectura, Escritura`:

```
Console.Write(obj); // Muestra Lectura, Escritura
```

Esto se debe a que en ausencia del modificador `"F"`, **`Format()`** mira dentro de los metadatos del tipo enumerado al que pertenece el valor numérico a mostrar si éste dispone del atributo **`Flags`**. Si es así funciona como si se le hubiese pasado `"F"`.

También cabe destacar que, para crear objetos de enumeraciones cuyo valor sea una combinación de valores de literales de su tipo enumerado, el método método **`Parse()`** de **`Enum`** permite que la cadena que se le especifica como segundo parámetro cuente con múltiples literales separados por comas. Por ejemplo, un objeto `ModificadorArchivo` que represente modificadores de lectura y ocultación puede crearse con:

```
ModificadorArchivo obj = (ModificadorArchivo)
        Enum.Parse(typeof(ModificadorArchivo),"Lectura,Oculto"));
```

Hay que señalar que esta capacidad de crear objetos de enumeraciones cuyo valor almacenado sea una combinación de los literales definidos en dicha enumeración es totalmente independiente de si al definirla se utilizó el atributo **`Flags`** o no.

Principio Página

El lenguaje de programación C#

En esta página:

Tema 15: Interfaces

Concepto de interfaz

Una **interfaz** es la definición de un conjunto de métodos para los que no se da implementación, sino que se les define de manera similar a como se definen los métodos abstractos. Es más, una interfaz puede verse como una forma especial de definir clases que sólo cuenten con miembros abstractos.

Como las clases abstractas, las interfaces son tipos referencia, no puede crearse objetos de ellas sino sólo de tipos que deriven de ellas, y participan del polimorfismo. Sin embargo, también tiene muchas diferencias con éstas:

- **Es posible definir tipos que deriven de más de una interfaz**. Esto se debe a que los problemas que se pueden presentar a la hora de crear tipos que hereden de varios tipos son debidos a que pueden haber conflictos difíciles de resolver y si un tipo hereda más de una versión de un mismo método procedentes de padres diferentes y con códigos distintos. Sin embargo, con las interfaces las interfaces se permite la herencia múltiple porque esto nunca puede ocurrir debido a que las interfaces no incluyen código.

- Aunque las estructuras no pueden heredar clases, sí pueden hacerlo de interfaces

- Todo tipo que derive de una interfaz ha de dar una implementación de todos los miembros que hereda de esta, y no como ocurre con las clases abstractas donde es posible no darla si se define como abstracta también la clase hija. De esta manera queda definido un contrato en la clase que la hereda que va a permitir poder usarla con seguridad en situaciones polimórficas: toda clase que herede una interfaz implementará todos los métodos de la misma. Por esta razón se suele denominar **implementar** una interfaz al hecho de heredar de ella.

- Las interfaces sólo pueden tener como miembros métodos normales, eventos, propiedades e

indizadores; pero no pueden incluir definiciones de campos, operadores, constructores o destructores.

Definición de interfaces

La sintaxis general que se sigue a la hora de definir una interfaz es:

```
<modificadores> interface <nombre>:<interfacesBase>
{
  <miembros>
}
```

Los <modificadores> admitidos por las interfaces son los mismos que los de las clases Es decir, **public**, **internal, private**, **protected**, **protected internal** o **new** (e igualmente, los cuatro último sólo son aplicables a interfaces definidas dentro de otros tipos)

El <nombre> de una interfaz puede ser cualquier identificador válido, aunque por convenio se suele usar I como primer carácter del mismo (IComparable, IA, etc)

Los <miembros> de las interfaces pueden ser definiciones de métodos, propiedades, indizadores o eventos, pero no campos, operadores, constructores o destructores. La sintaxis que se sigue para definir cada tipo de miembro es la misma que para definirlos como abstractos en una clase pero sin incluir **abstract** por suponerse impícitamente:

- **Métodos:** <tipoRetorno> <nombreMétodo>**(**<*parámetros*>**);**
- **Propiedades:** <tipo> <nombrePropiedad> **{***set; get;***}**

  Los bloques **get** y **set** pueden intercambiarse y puede no incluirse uno de ellos (propiedad de sólo lectura o de sólo escritura según el caso), pero no los dos.

- **Indizadores:** <tipo> **this[**<índices>**] {***set; get;***}**

  Al igual que las propiedades, los bloques **set** y **get** pueden intercambiarse y obviarse uno de ellos al definirlos.

- **Eventos:** **event** <delegado> <nombreEvento>**;**

  Nótese que a diferencia de las propiedades e indizadores, no es necesario indicar nada sobre sus bloques **add** y **remove**. Esto se debe a que siempre se han de implementar ambos, aunque si se usa la sintaxis básica el compilador les da una implementación por defecto automáticamente.

Cualquier definición de un miembro de una interfaz puede incluir el modificador **new** para indicar que pretende ocultar otra heredada de alguna interfaz padre. Sin embargo, el resto de modificadores no son válidos ya que implícitamente siempre se considera que son **public** y **abstract**. Además, una interfaz tampoco puede incluir miembros de tipo, por lo que es incorrecto incluir el modificador **static** al definir sus miembros.

Cada interfaz puede heredar de varias interfaces, que se indicarían en <interfacesBase> separadas por comas. Esta lista sólo puede incluir interfaces, pero no clases o estructuras; y a continuación se muestra un ejemplo de cómo definir una interfaz IC que hereda de otras dos interfaces IA y IB:

```
public delegate void D (int x);

interface IA
{
  int PropiedadA
```

```
  void Común(int x);
}

interface IB
{
  int this [int índice] {get; set;}
  void Común(int x);
}

interface IC: IA, IB
{
  event D EventoC;
}
```

Nótese que aunque las interfaces padres de IC contienen un método común no hay problema alguno a la hora de definirlas. En el siguiente epígrafe veremos cómo se resuelven las ambigüedades que por esto pudiesen darse al implementar IC.

Implementación de interfaces

Para definir una clase o estructura que implemente una o más interfaces basta incluir los nombres de las mismas como si de una clase base se tratase -separándolas con comas si son varias o si la clase definida hereda de otra clase- y asegurar que la clase cuente con definiciones para todos los miembros de las interfaces de las que hereda -lo que se puede conseguir definiéndolos en ella o heredándolos de su clase padre.

Las definiciones que se den de miembros de interfaces han de ser siempre públicas y no pueden incluir **override**, pues como sus miembros son implícitamente **abstract** se sobreentiende. Sin embargo, sí pueden dársele los modificadores como **virtual** ó **abstract** y usar **override** en redefiniciones que se les den en clases hijas de la clase que implemente la interfaz.

Cuando una clase deriva de más de una interfaz que incluye un mismo miembro, la implementación que se le dé servirá para todas las interfaces que cuenten con ese miembro. Sin embargo, también es posible dar una implementación diferente para cada una usando una **implementación explícita**, lo que consiste en implementar el miembro sin el modificador **public** y anteponiendo a su nombre el nombre de la interfaz a la que pertenece seguido de un punto (carácter **.**)

Cuando un miembro se implementa explícitamente no puede dársele modificadores como en las implementaciones implícitas, ni siquiera **virtual** o **abstract**. Una forma de simular los modificadores que se necesiten consiste en darles un cuerpo que lo que haga sea llamar a otra función que sí cuente con esos modificadores.

El siguiente ejemplo muestra cómo definir una clase CL que implemente la interfaz IC:

```
class CL:IC
{
  public int PropiedadA
  {
    get {return  5;}
    set {Console.WriteLine("Asignado {0} a PropiedadA", value);}
  }

  void IA.Común(int x)
  {
    Console.WriteLine("Ejecutado Común() de IA");
  }

  public int this[int índice]
  {
```

```
    get { return 1;}
    set { Console.WriteLine("Asignado {0} a indizador", value); }
  }

  void IB.Común(int x)
  {
    Console.WriteLine("Ejecutado Común() de IB");
  }

  public event D EventoC;
}
```

Como se ve, para implementar la interfaz IC ha sido necesario implementar todos sus miembros, incluso los heredados de IA y IB, de la siguiente manera:

- Al EventoC se le ha dado la implementación por defecto, aunque si se quisiese se podría haber dado una implementación específica a sus bloques **add** y **remove**.
- Al método Común() se le ha dado una implementación para cada versión heredada de una de las clases padre de IC, usándose para ello la sintaxis de implementación explícita antes comentada. Nótese que no se ha incluido el modificador **public** en la implementación de estos miembros.
- A la PropiedadA se le ha dado una implementación con un bloque set que no aparecía en la definición de PropiedadA en la interfaz IA. Esto es válido hacerlo siempre y cuando la propiedad no se haya implementado explícitamente, y lo mismo ocurre con los indizadores y en los casos en que en vez de **set** sea **get** el bloque extra implementado.

Otra utilidad de las implementaciones explícitas es que son la única manera de conseguir poder dar implementación a métodos ocultados en las definiciones de interfaces. Por ejemplo, si tenemos:

```
interface IPadre
{
  int P
}

interface IHija:Padre
{
  new int P();
}
```

La única forma de poder definir una clase donde se dé una implementación tanto para el método P() como para la propiedad P, es usando implementación explícita así:

```
class C: IHija
{
  void IPadre.P {}
  public int P()
}
```

O así:

```
class C: IHija
{
  public void P () {}
  int IHija.P() {}
```

```
}
```

O así:

```
class C: IHija
{
  void IPadre.P() {}
  int IHija.P() {}
}
```

Pero como no se puede implementar es sin ninguna implementación explícita, pues se produciría un error al tener ambos miembros las misma signatura. Es decir, la siguiente definición no es correcta:

```
class C: IHija
{
  public int P() {}     // ERROR: Ambos miembros tienen la misma signatura
  public void P() {}
}
```

Es posible reimplementar en una clase hija las definiciones que su clase padre diese para los métodos que heredó de una interfaz. Para hacer eso basta hacer que la clase hija también herede de esa interfaz y dar en ella las definiciones explícitas de miembros de la interfaz que se estimen convenientes, considerándose que las implementaciones para los demás serán las heredadas de su clase padre. Por ejemplo:

```
using System;

interface IA
{
  void F();
}

class C1: IA
{
  public void F()
  {
    Console.WriteLine("El F() de C1");
  }
}

class C2: C1, IA
{
  void IA.F()    // Sin implementación explícita no redefiniría, sino ocultaría
  {
    Console.WriteLine("El F() de C2");
  }

  public static void Main()
  {
    IA obj = new  C1();
    IA obj2 = new C2();

    obj.F();
    obj2.F();
  }
}
```

Reimplementar un miembro de una interfaz de esta manera es parecido a redefinir los miembros reimplementados, sólo que ahora la redefinición sería sólamente accesible a través de variables del tipo de la

interfaz Así, la salida del ejemplo anterior sería:

```
El F() de C1
El F() de C2
```

Hay que tener en cuenta que de esta manera sólo pueden hacerse reimplementaciones de miembros si la clase donde se reimplementa hereda directamente de la interfaz implementada explícitamente o de alguna interfaz derivada de ésta. Así, en el ejemplo anterior sería incorrecto haber hecho:

```
class C2:C1     // La lista de herencias e interfaces implementadas por C2
                // sólo incluye a C1
{
  void IA.f();  // ERROR: Aunque C1 herede de IA, IA no se incluye directamente
                // en la lista de interfaces implementadas por C2
}
```

Es importante señalar que el nombre de interfaz especificado en una implementación explícita ha de ser exactamente el nombre de la interfaz donde se definió el miembro implementado, no el de alguna subclase de la misma. Por ejemplo:

```
interface I1
{
  void F()
}

interface I2
{}

class C1:I2
{
  public void I2.F();  //ERROR: habría que usar I1.F()
}
```

En el ejemplo anterior, la línea comentada contiene un error debido a que `F()` se definió dentro de la interfaz `I1`, y aunque también pertenezca a `I2` porque ésta lo hereda de `I1`, a la hora de implementarlo explícitamente hay que prefijar su nombre de `I1`, no de `I2`.

Acceso a miembros de una interfaz

Se puede acceder a los miembros de una interfaz implementados en una clase de manera no explícita a través de variables de esa clase como si de miembros normales de la misma se tratase. Por ejemplo, este código mostraría un cinco por pantalla:

```
CL c = new CL();
Console.WriteLine(c.PropiedadA);
```

Sin embargo, también es posible definir variables cuyo tipo sea una interfaz. Aunque no existen constructores de interfaces, estas variables pueden inicializarse gracias al polimorfismo asignándoles objetos de clases que implementen esa interfaz. Así, el siguiente código también mostraría un cinco por pantalla:

```
IA a = new CL();
Console.WriteLine(a.PropiedadA);
```

Nótese que a través de una variable de un tipo interfaz sólo se puede acceder a miembros del objeto almacenado en ella que estén definidos en esa interfaz. Es decir, los únicos miembros válidos para el objeto a anterior serían `PropiedadA` y `Común()`

En caso de que el miembro al que se pretenda acceder haya sido implementado explícitamente, sólo puede accederse a él a través de variables del tipo interfaz al que pertenece y no a través de variables de tipos que hereden de ella, ya que la definición de estos miembros es privada al no llevar modificador de acceso. Por ejemplo:

```
CL cl = new CL();
IA a = cl;
IB b = cl;
// Console.WriteLine(cl.Común()); // Error: Común() fue
                                  // implementado explícitamente
Console.WriteLine(a.Común());
Console.WriteLine(b.Común());
Console.WriteLine(((IA) cl).Común());
Console.WriteLine(((IB) cl).Común());
```

Cada vez que se llame a un método implementado explícitamente se llamará a la versión del mismo definida para la interfaz a través de la que se accede. Por ello, la salida del código anterior será:

```
Ejecutado Común() de IA
Ejecutado Común() de IB
Ejecutado Común() de IA
Ejecutado Común() de IB
```

Se puede dar tanto una implementación implícita como una explícita de cada miembro de una interfaz. La explícita se usará cuando se acceda a un objeto que implemente esa interfaz a través de una referencia a la interfaz, mientras que la implícita se usará cuando el acceso se haga a través de una referencia del tipo que implementa la interfaz. Por ejemplo, dado el siguiente código:

```
interface I
{
  object Clone();
}

class Clase:I
{
  public object Clone()
  {
    Console.WriteLine("Implementación implícita");
  }

  public object IClonable.Clone()
  {
    Console.WriteLine("Implementación explícita");
  }

  public static void Main()
  {
    Clase obj = new Clase();
    ((I) obj).Clone();
    obj.Clone();
  }
}
```

El resultado que por pantalla se mostrará tras ejecutarlo es:

```
Implementación explícita
Implementación implícita
```

Principio Página

El lenguaje de programación C#

En esta página:

Tema 16: Instrucciones

Concepto de instrucción

Toda acción que se pueda realizar en el cuerpo de un método, como definir variables locales, llamar a métodos, asignaciones y muchas cosas más que veremos a lo largo de este tema, son **instrucciones**.

Las instrucciones se agrupan formando **bloques de instrucciones**, que son listas de instrucciones encerradas entre llaves que se ejecutan una tras otra. Es decir, la sintaxis que se sigue para definir un bloque de instrucciones es:

```
{ <listaInstrucciones> }
```

Toda variable que se defina dentro de un bloque de instrucciones sólo existirá dentro de dicho bloque. Tras él será inaccesible y podrá ser destruida por el recolector de basura. Por ejemplo, este código no es válido:

```
public void f();
{
  { int b; }
  b = 1;       // ERROR: b no existe fuera del bloque donde se declaró.
}
```

Los bloques de instrucciones pueden anidarse, aunque si dentro de un bloque interno definimos una variable con el mismo nombre que otra definida en un bloque externo se considerará que se ha producido un error, ya que no se podrá determinar a cuál de las dos se estará haciendo referencia cada vez que se utilice su nombre en el bloque interno.

Instrucciones básicas

Definiciones de variables locales

En el *Tema 7:Variables y tipos de datos* ya se vió que las **variables locales** son variables que se definen en el cuerpo de los métodos y sólo son accesibles desde dichos cuerpos. Recuérdese que la sintaxis explicada para definirlas era la siguiente:

```
<modificadores> <tipoVariable> <nombreVariable>  = <valor>;
```

También se vió que podían definirse varias variables en una misma instrucción separando sus pares nombre-valor mediante comas. Por ejemplo:

```
int a=5, b, c=-1;
```

Asignaciones

Una **asignación** es simplemente una instrucción mediante la que se indica un valor a almacenar en un dato. La sintaxis usada para ello es:

```
<destino> = <origen>;
```

En temas previos ya se han dado numerosos ejemplos de cómo hacer esto, por lo que no es necesario hacer ahora mayor hincapié en ello.

Llamadas a métodos

En el *Tema 8: Métodos* ya se explicó que una **llamada a un método** consiste en solicitar la ejecución de sus instrucciones asociadas dando a sus parámetros ciertos valores. Si el método a llamar es una método de objeto, la sintaxis usada para ello es:

```
<objeto>.<nombreMétodo>(<valoresParámetros>);
```

Y si el método a llamar es un método de tipo, entonces la llamada se realiza con:

```
<nombreTipo>.<nombreMétodo>(<valoresParámetros>);
```

Recuérdese que si la llamada al método de tipo se hace dentro de la misma definición de tipo donde el método fue definido, la sección *<nombreTipo>.* de la sintaxis es opcional.

Instrucción nula

La **instrucción nula** es una instrucción que no realiza nada en absoluto. Su sintaxis consiste en escribir un simple punto y coma para representarla. O sea, es:

```
;
```

Suele usarse cuando se desea indicar explícitamente que no se desea ejecutar nada. Usarla es útil para facilitar la legibilidad del código o, como veremos más adelante en el tema, porque otras instrucciones la necesitan para indicar cuándo en algunos de sus bloques de instrucciones componentes no se ha de realizar ninguna acción.

Instrucciones condicionales

Las **instrucciones condicionales** son instrucciones que permiten ejecutar bloques de instrucciones sólo si se da

una determinada condición. En los siguientes subapartados de este epígrafe se describen cuáles son las instrucciones condicionales disponibles en C#

Instrucción if

La **instrucción if** permite ejecutar ciertas instrucciones sólo si de da una determinada condición. Su sintaxis de uso es la sintaxis:

```
if (<condición>)
  <instruccionesIf>
else
  <instruccionesElse>
```

El significado de esta instrucción es el siguiente: se evalúa la expresión <condición>, que ha de devolver un valor lógico. Si es cierta (devuelve **true**) se ejecutan las <instruccionesIf>, y si es falsa (**false**) se ejecutan las <instruccionesElse> La rama **else** es opcional, y si se omite y la condición es falsa se seguiría ejecutando a partir de la instrucción siguiente al **if**. En realidad, tanto <instruccionesIf> como <instruccionesElse> pueden ser una única instrucción o un bloque de instrucciones.

Un ejemplo de aplicación de esta instrucción es esta variante del HolaMundo:

```
using System;

class HolaMundoIf
{
  public static void Main(String[] args)
  {
    if (args.Length > 0)
      Console.WriteLine("¡Hola {0}!", args[0]);
    else
      Console.WriteLine("¡Hola mundo!");
  }
}
```

Si ejecutamos este programa sin ningún argumento veremos que el mensaje que se muestra es ¡Hola Mundo!, mientras que si lo ejecutamos con algún argumento se mostrará un mensaje de bienvenida personalizado con el primer argumento indicado.

Instrucción switch

La **instrucción switch** permite ejecutar unos u otros bloques de instrucciones según el valor de una cierta expresión. Su estructura es:

```
switch (<expresión>)
{
  case <valor1>:  <bloque1>
                                        <siguienteAcción>
  case <valor2>:  <bloque2>
                                        <siguienteAcción>
                        ...
  default:  <bloqueDefault>
                                <siguienteAcción>
}
```

El significado de esta instrucción es el siguiente: se evalúa <expresión>. Si su valor es <valor1> se ejecuta el <bloque1>, si es <valor2> se ejecuta <bloque2>, y así para el resto de valores especificados. Si no es igual a

ninguno de esos valores y se incluye la rama **default**, se ejecuta el <bloqueDefault>; pero si no se incluye se pasa directamente a ejecutar la instrucción siguiente al **switch** .

Los valores indicados en cada rama del **switch** han de ser expresiones constantes que produzcan valores de algún tipo básico entero, de una enumeración, de tipo **char** o de tipo **string**. Además, no puede haber más de una rama con el mismo valor.

En realidad, aunque todas las ramas de un **switch** son opcionales siempre se ha de incluir al menos una. Además, la rama **default** no tiene porqué aparecer la última si se usa, aunque es recomendable que lo haga para facilitar la legibilidad del código.

El elemento marcado como <siguienteAcción> colocado tras cada bloque de instrucciones indica qué es lo que ha de hacerse tras ejecutar las instrucciones del bloque que lo preceden. Puede ser uno de estos tres tipos de instrucciones:

```
goto case <valori>;
goto default;
break;
```

Si es un **goto case** indica que se ha de seguir ejecutando el bloque de instrucciones asociado en el **switch** a la rama del <valori> indicado, si es un **goto default** indica que se ha de seguir ejecutando el bloque de instrucciones de la rama **default**, y si es un **break** indica que se ha de seguir ejecutando la instrucción siguiente al switch.

El siguiente ejemplo muestra cómo se utiliza **switch**:

```
using System;

class HolaMundoSwitch
{
  public static void Main(String[] args)
  {
    if (args.Length > 0)
      switch(args[0])
      {
        case "José": Console.WriteLine("Hola José. Buenos días");
                     break;
        case "Paco": Console.WriteLine("Hola Paco. Me alegro de verte");
                     break;
        default: Console.WriteLine("Hola {0}", args[0]);
                 break;
      }
    else
      Console.WriteLine("Hola Mundo");
  }
}
```

Este programa reconoce ciertos nombres de personas que se le pueden pasar como argumentos al lanzarlo y les saluda de forma especial. La rama **default** se incluye para dar un saludo por defecto a las personas no reconocidas.

Para los programadores habituados a lenguajes como C++ es importante resaltarles el hecho de que, a diferencia de dichos lenguajes, C# obliga a incluir una sentencia **break** o una sentencia **goto case** al final de cada rama del **switch** para evitar errores comunes y difíciles de detectar causados por olvidar incluir **break;** al final de alguno de estos bloques y ello provocar que tras ejecutarse ese bloque se ejecute también el siguiente.

Instrucciones iterativas

Las **instrucciones iterativas** son instrucciones que permiten ejecutar repetidas veces una instrucción o un bloque de instrucciones mientras se cumpla una condición. Es decir, permiten definir bucles donde ciertas instrucciones se ejecuten varias veces. A continuación se describen cuáles son las instrucciones de este tipo incluidas en C#.

Instrucción while

La **instrucción** `while` permite ejecutar un bloque de instrucciones mientras se de una cierta instrucción. Su sintaxis de uso es:

```
while (<condición>)
  <instrucciones>
```

Su significado es el siguiente: Se evalúa la `<condición>` indicada, que ha de producir un valor lógico. Si es cierta (valor lógico `true`) se ejecutan las `<instrucciones>` y se repite el proceso de evaluación de `<condición>` y ejecución de `<instrucciones>` hasta que deje de serlo. Cuando sea falsa (`false`) se pasará a ejecutar la instrucción siguiente al `while`. En realidad `<instrucciones>` puede ser una única instrucción o un bloque de instrucciones.

Un ejemplo cómo utilizar esta instrucción es el siguiente:

```
using System;

class HolaMundoWhile
{
  public static void Main(String[] args)
  {
    int actual = 0;

    if (args.Length > 0)
      while (actual < args.Length)
      {
        Console.WriteLine("¡Hola {0}!", args[actual]);
        actual  = actual + 1;
      }
    else
      Console.WriteLine("¡Hola mundo!");
  }
}
```

En este caso, si se indica más de un argumento al llamar al programa se mostrará por pantalla un mensaje de saludo para cada uno de ellos. Para ello se usa una variable actual que almacena cuál es el número de argumento a mostrar en cada ejecución del `while`. Para mantenerla siempre actualizada lo que se hace es aumentar en una unidad su valor tras cada ejecución de las `<instrucciones>` del bucle.

Por otro lado, dentro de las `<instrucciones>` de un `while` pueden usarse dos instrucciones especiales:

- **break;**: Indica que se ha de abortar la ejecución del bucle y continuarse ejecutando por la instrucción siguiente al **while**.
- **continue;**: Indica que se ha de abortar la ejecución de las <instrucciones> y reevaluarse la <condición> del bucle, volviéndose a ejecutar la <instrucciones> si es cierta o pasándose a ejecutar la instrucción siguiente al**while** si es falsa.

Instrucción do...while

La instrucción **do...while** es una variante del **while** que se usa así:

```
do
  <instrucciones>
while(<condición>);
```

La única diferencia del significado de **do...while** respecto al de **while** es que en vez de evaluar primero la condición y ejecutar <instrucciones> sólo si es cierta, **do...while** primero ejecuta las <instrucciones> y luego mira la <condición> para ver si se ha de repetir la ejecución de las mismas. Por lo demás ambas instrucciones son iguales, e incluso también puede incluirse **break;** y **continue;** entre las <instrucciones> del **do...while**.

do ... while está especialmente destinado para los casos en los que haya que ejecutar las <instrucciones> al menos una vez aún cuando la condición sea falsa desde el principio., como ocurre en el siguiente ejemplo:

```
using System;

 class HolaMundoDoWhile
 {
   public static void Main()
   {
    String leído;

    do
    {
      Console.WriteLine("Clave: ");
      leído = Console.ReadLine();
    }
    while (leído != "José");
    Console.WriteLine("Hola José");
  }
}
```

Este programa pregunta al usuario una clave y mientras no introduzca la correcta (José) no continuará ejecutándose. Una vez que introducida correctamente dará un mensaje de bienvenida al usuario.

Instrucción for

La **instrucción** **for** es una variante de **while** que permite reducir el código necesario para escribir los tipos de bucles más comúnmente usados en programación. Su sintaxis es:

```
for (<inicialización>; <condición>; <modificación>)
  <instrucciones>
```

El significado de esta instrucción es el siguiente: se ejecutan las instrucciones de <inicialización>, que suelen usarse para definir e inicializar variables que luego se usarán en <instrucciones>. Luego se evalúa <condición>, y si es falsa se continúa ejecutando por la instrucción siguiente al **for;** mientras que si es cierta se ejecutan las <instrucciones> indicadas, luego se ejecutan las instrucciones de <modificación> -que como su nombre indica suelen usarse para modificar los valores de variables que se usen en <instrucciones>- y luego se reevalúa <condición> repitiéndose el proceso hasta que ésta última deje de ser cierta.

En <inicialización> puede en realidad incluirse cualquier número de instrucciones que no tienen porqué ser

relativas a inicializar variables o modificarlas, aunque lo anteriro sea su uso más habitual. En caso de ser varias se han de separar mediante comas (**,**), ya que el carácter de punto y coma (**;**) habitualmente usado para estos menesteres se usa en el **for** para separar los bloques de <inicialización>, <condición> y <modificación>. Además, la instrucción nula no se puede usar en este caso y tampoco pueden combinarse definiciones de variables con instrucciones de otros tipos.

Con <modificación> pasa algo similar, ya que puede incluirse código que nada tenga que ver con modificaciones pero en este caso no se pueden incluir definiciones de variables.

Como en el resto de instrucciones hasta ahora vistas, en <instrucciones> puede ser tanto una única instrucción como un bloque de instrucciones. Además, las variables que se definan en <inicialización> serán visibles sólo dentro de esas <instrucciones>

La siguiente clase es equivalente a la clase HolaMundoWhile ya vista solo que hace uso del **for** para compactar más su código:

```
using System;

class HolaMundoFor
{
  public static void Main(String[] args)
  {
    if (args.Length > 0)
      for (int actual = 0; actual < args.Length; actual++)
        Console.WriteLine("¡Hola {0}!", args[actual]);
    else
      Console.WriteLine("¡Hola mundo!");
  }
}
```

Al igual que con **while**, dentro de las <instrucciones> del **for** también pueden incluirse instrucciones **continue;** y **break;** que puedan alterar el funcionamiento normal del bucle.

Instrucción foreach

La **instrucción** **foreach** es una variante del **for** pensada especialmente para compactar la escritura de códigos donde se realice algún tratamiento a todos los elementos de una colección, que suele un uso muy habitual de **for** en los lenguajes de programación que lo incluyen. La sintaxis que se sigue a la hora de escribir esta instrucción **foreach** es:

```
foreach (<tipoElemento> <elemento> in <colección>)
  <instrucciones>
```

El significado de esta instrucción es muy sencillo: se ejecutan **<instrucciones>** para cada uno de los elementos de la <colección> indicada. <elemento> es una variable de sólo lectura de tipo <tipoElemento> que almacenará en cada momento el elemento de la colección que se esté procesando y que podrá ser accedida desde <instrucciones>.

Es importante señalar que <colección> no puede valer **null** porque entonces saltaría una excepción de tipo System.NullReferenceException, y que <tipoElemento> ha de ser un tipo cuyos objetos puedan almacenar los valores de los elementos de <colección>

En tanto que una tabla se considera que es una colección, el siguiente código muestra cómo usar **for** para compactar aún más el código de la clase HolaMundoFor anterior:

```
using System;
```

```
class HolaMundoFoeach
{
  public static void Main(String[] args)
  {
    if (args.Length > 0)
      foreach(String arg in args)
        Console.WriteLine("¡Hola {0}!", arg);
    else
      Console.WriteLine("¡Hola mundo!");
  }
}
```

En general, se considera que una colección es todo aquel objeto que implemente la interfaz **System.Collections.IEnumerable**. Esta interfaz está definida en la BCL así:

```
interface IEnumerable
{
   IEnumerator GetEnumerator();
}
```

El objeto de interfaz **System.Collections.IEnumerator** devuelto ha de ser un enumerador que permita recorrer los elementos de la <colección>. Dicha interfaz está así predefinida:

```
interface IEnumerator
{
   object Current
   bool MoveNext();
   void Reset();
}
```

El método **Reset()** ha de implementarse de modo que devuelva el enumerador reiniciado a un estado inicial donde aún no referencie ni siquiera al primer elemento de la colección sino que sea necesario llamar a **MoveNext()** para que lo haga.

El método **MoveNext()** se ha de implementar de modo que haga que el enumerador pase a apuntar al siguiente elemento de la colección y devuelva un booleano que indique si tras avanzar se ha alcanzado el final de la colección.

La propiedad **Current** se ha de implementar de modo que devuelva siempre el elemento de la colección al que el enumerador esté referenciando. Si se intenta leer **Current** habiéndose ya recorrido toda la colección o habiéndose reiniciado la colección y no habiéndose colocado en su primer elemento con **MoveNext()**, se ha de producir una excepción de tipo **System.Exception.SystemException.InvalidOperationException**

Otra forma de conseguir que **foreach** considere que un objeto es una colección válida consiste en hacer que dicho objeto siga el **patrón de colección**. Este patrón consiste en definir el tipo del objeto de modo que sus objetos cuenten con un método público **GetEnumerator()** que devuelva un objeto no nulo que cuente con una propiedad pública llamada **Current** que permita leer el elemento actual y con un método público **bool MoveNext()** que permita cambiar el elemento actual por el siguiente y devuelva **false** sólo cuando se haya llegado al final de la colección.

El siguiente ejemplo muestra ambos tipos de implementaciones:

```
using System;
using System.Collections;

class Patron
```

```
{
  private int actual = -1;

  public Patron GetEnumerator()
  {
    return this;
  }

  public int Current
  {
    get {return actual;}
  }

  public bool MoveNext()
  {
    bool resultado = true;

    actual++;
    if (actual==10)
      resultado = false;
    return resultado;
  }
}

class Interfaz:IEnumerable,IEnumerator
{
  private int actual = -1;

  public object Current
  {
    get {return actual;}
  }

  public bool MoveNext()
  {
    bool resultado = true;

    actual++;
    if (actual==10)
      resultado = false;
    return resultado;
  }

  public IEnumerator GetEnumerator()
  {
    return this;
  }

  public void Reset()
  {
    actual = -1;
  }
}

class Principal
{
  public static void Main()
  {
    Patron obj = new Patron();
```

```
      Interfaz obj2 = new Interfaz();

      foreach (int elem in obj)
        Console.WriteLine(elem);
      foreach (int elem in obj2)
        Console.WriteLine(elem);
    }
  }
```

El tipo **System.Array** implementa la interfaz **System.Collectiones.IEnumerator**, por lo que todas las tablas podrán ser usadas recorridas con **foreach**. Si la tabla a recorrer es multidimensional, sus elementos se recorrerán en orden como muestra este ejemplo:

```
int[,] tabla = { ,  };

foreach (int elemento in tabla)
  Console.WriteLine(elemento);
```

La salida por pantalla del fragmento de código anterior será:

```
1
2
3
4
```

La utilidad de implementar el patrón colección en lugar de la interfaz **IEnumerable** es que así no es necesario que **Current** devuelva siempre un **object**, sino que puede devolver objetos de tipos más concretos y gracias a ello puede detectarse al compilar si el <tipoElemento> indicado puede o no almacenar los objetos de la colección.

Por ejemplo, si en el ejemplo anterior sustituimos en el último **foreach** el <tipoElemento> indicado por Patrón, el código seguirá compilando pero al ejecutarlo saltará una excepción **System.InvalidCastException**. Sin embargo, si la sustitución se hubiese hecho en el penúltimo **foreach**, entonces el código directamente no compilaría y se nos informaría de un error debido a que los objetos **int** no son convertibles en objetos Patrón.

También hay que tener en cuenta que la comprobación de tipos que se realiza en tiempo de ejecución si el objeto sólo implementó la interfaz **IEnumerable** es muy estricta, en el sentido de que si en el ejemplo anterior sustituimos el <tipoElemento> del último **foreach** por **byte** también se lanzará la excepción al no ser los objetos de tipo **int** implícitamente convertibles en **byte**s sino sólo a través del operador **()** Sin embargo, cuando se sigue el patrón de colección las comprobaciones de tipo no son tan estrictas y entonces sí que sería válido sustituir **int** por **byte** en <tipoElemento>.

El problema de sólo implementar el patrón colección es que este es una característica propia de C# y con las instrucciones **foreach** (o equivalentes) de lenguajes que no lo soporten no se podría recorrer colecciones que sólo siguiesen este patrón. Una solución en estos casos puede ser hacer que el tipo del objeto colección implemente tanto la interfaz **IEnumerable** como el patrón colección. Obviamente esta interfaz debería implementarse explícitamente para evitarse conflictos derivados de que sus miembros tengan signaturas coincidentes con las de los miembros propios del patrón colección.

Si un objeto de un tipo colección implementa tanto la interfaz **IEnumerable** como el patrón de colección, entonces en C# **foreach** usará el patrón colección para recorrerlo.

Instrucciones de excepciones

Concepto de excepción.

Las **excepciones** son el mecanismo recomendado en la plataforma .NET para la propagación de errores que se produzcan durante la ejecución de las aplicaciones (divisiones por cero, intentos de lectura de archivos dañados, etc.) Básicamente una excepción es un objeto derivado de **System.Exception** que se genera cuando en tiempo de ejecución se produce algún error y que contiene información sobre el mismo.

Tradicionalmente, el sistema que en otros lenguajes y plataformas se ha venido usando para informar estos errores consistía simplemente en hacer que los métodos en cuya ejecución pudiesen producirse devolvieran códigos que informasen sobre si se han ejecutado correctamente o, en caso contrario, sobre cuál fue el error producido. Sin embargo, las excepciones proporcionan las siguientes ventajas frente a dicho sistema:

- **Claridad:** El uso de códigos especiales para informar de error suele dificultar la legibilidad del fuente en tanto que se mezclan las instrucciones propias de la lógica del mismo con las instrucciones propias del tratamiento de los errores que pudiesen producirse durante su ejecución. Por ejemplo:

```
int resultado = obj.Método();
if (resultado == 0)  // Sin errores al ejecutar obj.Método();

else if (resultado == 1)  // Tratamiento de error de código 1

else if (resultado == 2) // Tratamiento de error de código 2
...
```

  Como se verá, utilizando excepciones es posible escribir el código como si nunca se fuesen a producir errores y dejar en una zona aparte todo el código de tratamiento de errores, lo que contribuye a facilitar la legibilidad de los fuentes.

- **Más información:** A partir del valor de un código de error puede ser difícil deducir las causas del mismo y conseguirlo muchas veces implica tenerse que consultar la documentación que proporcionada sobre el método que lo provocó, que puede incluso que no especifique claramente su causa.

  Por el contrario, una excepción es un objeto que cuenta con campos que describen las causas del error y a cuyo tipo suele dársele un nombre que resuma claramente su causa. Por ejemplo, para informar errores de división por cero se suele utilizar una excepción predefinida de tipo **DivideByZeroException** en cuyo campo **Message** se detallan las causas del error producido

- **Tratamiento asegurado:** Cuando se utilizan códigos de error nada obliga a tratarlos en cada llamada al método que los pueda producir, e ignorarlos puede provocar más adelante en el código comportamientos inesperados de causas difíciles de descubrir.

  Cuando se usan excepciones siempre se asegura que el programador trate toda excepción que pueda producirse o que, si no lo hace, se aborte la ejecución de la aplicación mostrándose un mensaje indicando dónde se ha producido el error.

Ahora bien, tradicionalmente en lenguajes como C++ el uso de excepciones siempre ha tenido las desventajas respecto al uso de códigos de error de complicar el compilador y dar lugar a códigos más lentos y difíciles de optimizar en los que tras cada instrucción que pudiese producir excepciones el compilador debe introducir las comprobaciones necesarias para detectarlas y tratarlas así como para comprobar que los objetos creados sean correctamente destruidos si se producen.

Sin embargo, en la plataforma .NET desaparacen los problemas de complicar el compilador y dificultar las optimizaciones ya que que es el CLR quien se encarga de detectar y tratar las excepciones y es su recolector de basura quien se encarga asegurar la correcta destrucción de los objetos. Obviamente el código seguirá siendo algo más lento, pero es un pequeño sacrificio que merece la pena hacer en tanto que ello asegura que nunca se producirán problemas difíciles de detectar derivados de errores ignorados.

La clase System.Exception

Como ya se ha dicho, todas las excepciones derivan de un tipo predefinido en la BCL llamado `System.Exception`. Los principales miembros que heredan de éste son:

- **`string Message {virtual get;}`**: Contiene un mensaje descriptivo de las causas de la excepción. Por defecto este mensaje es una cadena vacía (`""`)
- **`Exception InnerException {virtual get;}`**: Si una excepción fue causada como consecuencia de otra, esta propiedad contiene el objeto `System.Exception` que representa a la excepción que la causó. Así se pueden formar cadenas de excepciones de cualquier longitud. Si se desea obtener la última excepción de la cadena es mejor usar el método `virtual Exception GetBaseException()`
- **`string StackTrace {virtual get;}`:** Contiene la pila de llamadas a métodos que se tenía en el momento en que se produjo la excepción. Esta pila es una cadena con información sobre cuál es el método en que se produjo la excepción, cuál es el método que llamó a este, cuál es el que llamó a ese otro, etc.
- **`string Source {virtual get; virtual set;}`**: Almacena información sobre cuál fue la aplicación u objeto que causó la excepción.
- **`MethodBase TargetSite {virtual get;}`**: Almacena cuál fue el método donde se produjo la excepción en forma de objeto `System.Reflection.MethodBase`. Puede consultar la documentación del SDK si desea cómo obtener información sobre las características del método a través del objeto `MethodBase`.
- **`string HelpLink {virtual get;}`:** Contiene una cadena con información sobre cuál es la URI donde se puede encontrar información sobre la excepción. El valor de esta cadena puede establecerse con `virtual Exception SetHelpLink (string URI)`, que devuelve la excepción sobre la que se aplica pero con la URI ya actualizada.

Para crear objetos de clase `System.Exception` se puede usar los constructores:

```
    Exception()
Exception(string msg)
Exception(string msg, Exception causante)
```

El primer constructor crea una excepción cuyo valor para Message será `""` y no causada por ninguna otra excepción (`InnerException` valdrá `null`) El segundo la crea con el valor indicado para `Message`, y el último la crea con además la excepción causante indicada.

En la práctica, cuando se crean nuevos tipos derivados de `System.Exception` no se suele redefinir sus miembros ni añadirles nuevos, sino que sólo se hace la derivación para distinguir una excepciones de otra por el nombre del tipo al que pertenecen. Ahora bien, es conveniente respetar el convenio de darles un nombre acabado en `Exception` y redefinir los tres constructores antes comentados.

Excepciones predefinidas comunes

En el espacio de nombres **System** de la BCL hay predefinidas múltiples excepciones derivadas de **System.Exception** que se corresponden con los errores más comunes que pueden surgir durante la ejecución de una aplicación. En la **Tabla 8** se recogen algunas:

Tipo de la excepción	Causa de que se produzca la excepción
ArgumentException	Pasado argumento no válido (base de excepciones de argumentos)

ArgumentNullException	Pasado argumento nulo
ArgumentOutOfRangeException	Pasado argumento fuera de rango
ArrayTypeMistmatchException	Asignación a tabla de elemento que no es de su tipo
COMException	Excepción de objeto COM
DivideByZeroException	División por cero
IndexOutOfRangeException	Índice de acceso a elemento de tabla fuera del rango válido (menor que cero o mayor que el tamaño de la tabla)
InvalidCastException	Conversión explícita entre tipos no válida
InvalidOperationException	Operación inválida en estado actual del objeto
InteropException	Base de excepciones producidas en comunicación con código inseguro
NullReferenceException	Acceso a miembro de objeto que vale **null**
OverflowException	Desbordamiento dentro de contexto donde se ha de comprobar los desbordamientos (expresión constante, instrucción checked, operanción checked u opción del compilador /checked)
OutOfMemoryException	Falta de memoria para crear un objeto con new
SEHException	Excepción SHE del API Win32
StackOverflowException	Desbordamiento de la pila, generalmente debido a un excesivo número de llamadas recurrentes.
TypeInizializationException	Ha ocurrido alguna excepción al inicializar los campos estáticos o el constructor estático de un tipo. En InnerException se indica cuál es.

Tabla 8: Excepciones predefinidas de uso frecuente

Obviamente, es conveniente que si las aplicaciones que escribamos necesiten lanzar excepciones relativas a errores de los tipos especificados en la **Tabla 8**, lancen precisamente las excepciones indicadas en esa tabla y no cualquier otra - ya sea definida por nosotros mismos o predefinida en la BCL con otro significado.

Lanzamiento de excepciones. Instrucción throw

Para informar de un error no basta con crear un objeto del tipo de excepción apropiado, sino que también hay pasárselo al mecanismo de propagación de excepciones del CLR. A esto se le llama **lanzar la excepción**, y para hacerlo se usa la siguiente instrucción:

```
throw <objetoExcepciónALanzar>;
```

Por ejemplo, para lanzar una excepción de tipo **DivideByZeroException** se podría hacer:

```
throw new DivideByZeroException();
```

Si el objeto a lanzar vale **null**, entonces se producirá una **NullReferenceException** que será lanzada en vez de la excepción indicada en la instrucción **throw**.

Captura de excepciones. Instrucción try

Una vez lanzada una excepción es posible escribir código que es encarge de tratarla. Por defecto, si este código no se escribe la excepción provoca que la aplicación aborte mostrando un mensaje de error en el que se describe la excepción producida (información de su propiedad **Message**) y dónde se ha producido (información de su propiedad **StackTrace**) Así, dado el siguiente código fuente de ejemplo:

```
using System;
```

```
class PruebaExcepciones
{
  static void Main()
  {
    A obj1 = new A();
    obj1.F();
  }
}

class A
{
  public void F()
  {
    G();
  }

  static public void G()
  {
    int c = 0;
    int d = 2/c;
  }
}
```

Al compilarlo no se detectará ningún error ya que al compilador no le merece la pena calcular el valor de `c` en tanto que es una variable, por lo que no detectará que dividir `2/c` no es válido. Sin embargo, al ejecutarlo se intentará dividir por cero en esa instrucción y ello provocará que aborte la aplicación mostrando el siguiente mensaje:

```
Unhandled Exception: System.DivideByZeroException: Attempted to divide by zero.
   at PruebaExcepciones.Main()
```

Como se ve, en este mensaje se indica que no se ha tratado una excepción de división por cero (tipo **`DivideByZeroException`**) dentro del código del método `Main()` del tipo `PruebaExcepciones`. Si al compilar el fuente hubiésemos utilizado la opción **`/debug`**, el compilador habría creado un fichero **`.pdb`** con información extra sobre las instrucciones del ejecutable generado que permitiría que al ejecutarlo se mostrase un mensaje mucho más detallado con información sobre la instrucción exacta que provocó la excepción, la cadena de llamadas a métodos que llevaron a su ejecución y el número de línea que cada una ocupa en el fuente:

```
Unhandled Exception: System.DivideByZeroException: Attempted to divide by zero.
   at A.G() in E:\c#\Ej\ej.cs:line 22
   at A.F() in E:\c#\Ej\ej.cs:line 16
   at PruebaExcepciones.Main() in E:\c#\Ej\ej.cs:line 8
```

Si se desea tratar la excepción hay que encerrar la división dentro de una **instrucción `try`** con la siguiente sintaxis:

```
try
  <instrucciones>
catch (<excepción1>)
  <tratamiento1>
catch (<excepción2>)
  <tratamiento2>
...
finally
  <instruccionesFinally>
```

El significado de **`try`** es el siguiente: si durante la ejecución de las `<instrucciones>` se lanza una excepción

de tipo <excepción1> (o alguna subclase suya) se ejecutan las instrucciones <tratamiento1>, si fuese de tipo <excepción2> se ejecutaría <tratamiento2>, y así hasta que se encuentre una cláusula **catch** que pueda tratar la excepción producida. Si no se encontrase ninguna y la instrucción **try** estuviese anidada dentro de otra, se miraría en los **catch** de su **try** padre y se repetiría el proceso. Si al final se recorren todos los **try**s padres y no se encuentra ningún **catch** compatible, entonces se buscaría en el código desde el que se llamó al método que produjo la excepción. Si así se termina llegando al método que inició el hilo donde se produjo la excepción y tampoco allí se encuentra un tratamiento apropiado se aborta dicho hilo; y si ese hilo es el principal (el que contiene el punto de entrada) se aborta el programa y se muestra el mensaje de error con información sobre la excepción lanzada ya visto.

Así, para tratar la excepción del ejemplo anterior de modo que una división por cero provoque que a d se le asigne el valor 0, se podría reescribir G() de esta otra forma:

```
static public void G()
{
  try
  {
    int c = 0;
    int d = 2/c;
  }
  catch (DivideByZeroException)
  { d=0; }
}
```

Para simplificar tanto el compilador como el código generado y favorecer la legibilidad del fuente, en los **catch**s se busca siempre orden de aparación textual, por lo que para evitar **catch**s absurdos no se permite definir **catch**s que puedan capturar excepciones capturables por **catch**s posteriores a ellos en su misma instrucción **try**.

También hay que señalar que cuando en <instrucciones> se lance una excepción que sea tratada por un **catch** de algún **try** -ya sea de la que contiene las <instrucciones>, de algún **try** padre suyo o de alguno de los métodos que provocaron la llamada al que produjo la excepción- se seguirá ejecutando a partir de las instrucciones siguientes a ese **try**.

El bloque **finally** es opcional, y si se incluye ha de hacerlo tras todas los bloques **catch**. Las <instruccionesFinally> de este bloque se ejecutarán tanto si se producen excepciones en <instrucciones> como si no. En el segundo caso sus instrucciones se ejecutarán tras las <instrucciones>, mientras que en el primero lo harán después de tratar la excepción pero antes de seguirse ejecutando por la instrucción siguiente al **try** que la trató. Si en un **try** no se encuentra un **catch** compatible, antes de pasar a buscar en su **try** padre o en su método llamante padre se ejecutarán las <instruccionesFinally>.

Sólo si dentro de un bloque **finally** se lanzase una excepción se aborta la ejecución del mismo. Dicha excepción sería propagada al **try** padre o al método llamante padre del **try** que contuviese el **finally**.

Aunque los bloques **catch** y **finally** son opcionales, toda instrucción **try** ha de incluir al menos un bloque **catch** o un bloque **finally**.

El siguiente ejemplo resume cómo funciona la propagación de excepciones:

```
using System;

class MiException:Exception {}

class Excepciones
{
  public static void Main()
  {
```

```
      try
      {
        Console.WriteLine("En el try de Main()");
        Método();
        Console.WriteLine("Al final del try de Main()");
      }
      catch (MiException)
      {
        Console.WriteLine("En el catch de Main()");
      }
      finally
      {
        Console.WriteLine("finally de Main()");
      }
    }

    public static void Método()
    {
      try
      {
        Console.WriteLine("En el try de Método()");
        Método2();
        Console.WriteLine("Al final del try de Método()");
      }
      catch (OverflowException)
      {
        Console.WriteLine("En el catch de Método()");
      }
      finally
      {
        Console.WriteLine("finally de Método()");
      }
    }

    public static void Método2()
    {
      try
      {
        Console.WriteLine("En el try de Método2()");
        throw new MiException();
        Console.WriteLine("Al final del try de Método2()");
      }
      catch (DivideByZeroException)
      { Console.WriteLine("En el catch de Método2()"); }
      finally
      { Console.WriteLine("finally de Método2()"); }
    }
  }
```

Nótese que en este código lo único que se hace es definir un tipo nuevo de excepción llamado `MiException` y llamarse en el **`Main()`** a un método llamado `Método()` que llama a otro de nombre `Método2()` que lanza una excepción de ese tipo. Viendo la salida de este código es fácil ver el recorrido seguido durante la propagación de la excepción:

```
En try de Main()
En try de Método()
En try de Método2()
finally de Método2
```

```
finally de Método
En catch de Main()
finally de Main()
```

Como se puede observar, hay muchos **WriteLine()** que nunca se ejecutan ya que en cuento se lanza una excepción se sigue ejecutando tras la instrucción siguiente al **try** que la trató (aunque ejecutando antes los **finally** pendientes, como se deduce de la salida del ejemplo) De hecho, el compilador se dará cuenta que la instrucción siguiente al **throw** nunca se ejecutará e informará de ello con un mensaje de aviso.

La idea de todo este mecanismo de excepciones es evitar mezclar el código normal con el código de tratamiento de errores. Así, en <instrucciones> se escibiría el código como si no se pudiesen producir errores, en las cláusulas **catch** se tratarían los posibles errores, y en la cláusula **finally** se incluiría código a ejecutar tanto si produjesen errores como si no (suele usarse para liberar recursos ocupados, como fichero o conexiones de red abiertas)

En realidad, también es posible escribir cada cláusula **catch** definiendo una variable que se podrá usar dentro del código de tratamiento de la misma para hacer referencia a la excepción capturada. Esto se hace con la sintaxis:

```
catch (<tipoExcepción> <nombreVariable>)
{
  <tratamiento>
}
```

Nótese que en tanto que todas las excepciones derivan de **System.Exception**, para definir una cláusula **catch** que pueda capturar cualquier tipo de excepción basta usar:

```
catch(System.Exception <nombreObjecto>)
{
  <tratamiento>
}
```

En realidad la sintaxis anterior sólo permite capturar las excepciones propias de la plataforma .NET, que derivan de **System.Exception**. Sin embargo, hay lenguajes como C++ que permiten lanzar excepciones no derivadas de dicha clase, y para esos casos se ha incluido en C# una variante de **catch** sí que realmente puede capturar excepciones de cualquier tipo, tanto si derivan de **System.Exception** como si no. Su sintaxis es:

```
catch
{
 <tratamiento>
}
```

Como puede deducirse de su sintaxis, el problema que presenta esta última variante de **catch** es que no proporciona información sobre cuál es la excepción capturada, por lo que a veces puede resultar poco útil y si sólo se desea capturar cualquier excepción derivada de **System.Exception** es mejor usar la sintaxis explicada previamente a ella.

En cualquier casos, ambos tipos de cláusulas **catch** sólo pueden ser escritas como la última cláusula **catch** del **try**, ya que si no las cláusulas **catch** que le siguiesen nunca llegarían a ejecutarse debido a que las primeras capturarían antes cualquier excepción derivada de **System.Exception**.

Respecto al uso de **throw**, hay que señalar que hay una forma extra de usarlo que sólo es válida dentro de códigos de tratamiento de excepciones (códigos <tratamientoi> de las cláusulas **catch**) Esta forma de uso consiste en seguir simplemente esta sintaxis:

```
throw;
```

En este caso lo que se hace es relanzar la misma excepción que se capturó en el bloque **catch** dentro de cuyo de código de tratamiento se usa el **throw;** Hay que precisar que la excepción relanzada es precisamente la capturada, y aunque en el bloque **catch** se la modifique a través de la variable que la repreesnta, la versión relanzada será la versión original de la misma y no la modificada.

Además, cuando se relance una excepción en un **try** con cláusula **finally**, antes de pasar a reprocesar la excepción en el **try** padre del que la relanzó se ejecutará dicha cláusula.

Instrucciones de salto

Las **instrucciones de salto** permiten ejecutar variar el orden normal en que se ejecutan las instrucciones de un programa, que consiste en ejecutarlas una tras otra en el mismo orden en que se hubiesen escrito en el fuente. En los subapartados de este epígrafe se describirán cuáles son las instrucciones de salto incluidas en C#:

Instrucción break

Ya se ha visto que la **instrucción** **break** sólo puede incluirse dentro de bloques de instrucciones asociados a instrucciones iterativas o instrucciones **switch** e indica que se desea abortar la ejecución de las mismas y seguir ejecutando a partir de la instrucción siguiente a ellas. Se usa así:

```
break;
```

Cuando esta sentencia se usa dentro de un **try** con cláusula **finally**, antes de abortarse la ejecución de la instrucción iterativa o del **switch** que la contiene y seguirse ejecutando por la instrucción que le siga, se ejecutarán las instrucciones de la cláusula **finally** del **try**. Esto se hace para asegurar que el bloque **finally** se ejecute aún en caso de salto.

Además, si dentro una cláusula **finally** incluida en de un **switch** o de una instrucción iterativa se usa **break**, no se permite que como resultado del **break** se salga del **finally**.

Instrucción continue

Ya se ha visto que la **instrucción** **continue** sólo puede usarse dentro del bloque de instrucciones de una instrucción iterativa e indica que se desea pasar a reevaluar directamente la condición de la misma sin ejecutar el resto de instrucciones que contuviese. La evaluación de la condición se haría de la forma habitual: si es cierta se repite el bucle y si es falsa se continúa ejecutando por la instrucción que le sigue. Su sintaxis de uso es así de sencilla:

```
continue;
```

En cuanto a sus usos dentro de sentencias **try**, tiene las mismas restricciones que **break**: antes de salir de un **try** se ejecutará siempre su bloque **finally** y no es posible salir de un **finally** incluido dentro de una instrucción iterativa como consecuencia de un **continue**.

Instrucción return

Esta instrucción se usa para indicar cuál es el objeto que ha de devolver un método, y se usa así:

```
return <objetoRetorno>;
```

La ejecución de esta instrucción provoca que se aborte la ejecución del método dentro del que aparece y que se devuelva el <objetoRetorno> al método que lo llamó. Como es lógico, este objeto ha de ser del tipo de retorno del método en que aparece el **return** o de alguno compatible con él, por lo que esta instrucción sólo podrá incluirse en métodos cuyo tipo de retorno no sea **void**, o en los bloques **get** de las propiedades o indizadores. De hecho, es obligatorio que todo método con tipo de retorno termine por un **return**.

Los métodos que devuelvan **void** pueden tener un **return** con una sintaxis espacial en la que no se indica ningún valor a devolver sino que simplemente se usa **return** para indicar que se desea terminar la ejecución del método:

```
return;
```

Nuevamente, como con el resto de instrucciones de salto hasta ahora vistas, si se incluyese un **return** dentro de un bloque **try** con cláusula **finally**, antes de devolverse el objeto especificado se ejecutarían las instrucciones de la cláusula **finally**. Si hubiesen varios bloques **finally** anidados, las instrucciones de cada uno es ejecutarían de manera ordenada (o sea, del más interno al más externo) Ahora bien, lo que no es posible es incluir un **return** dentro de una cláusula **finally**.

Instrucción goto

La **instrucción** **goto** permite pasar a ejecutar el código a partir de una instrucción cuya etiqueta se indica en el **goto**. La sintaxis de uso de esta instrucción es:

```
goto <etiqueta>;
```

Como en la mayoría de los lenguajes, **goto** es una **instrucción maldita** cuyo uso no se recomienda porque dificulta innecesariamente la legibilidad del código y suele ser fácil simularla usando instrucciones iterativas y selectivas con las condiciones apropiadas. Sin embargo, en C# se incluye porque puede ser eficiente usarla si se anidan muchas instrucciones y para reducir sus efectos negativos se le han impuesto unas restricciones:

Sólo se pueden etiquetar instrucciones, y no a directivas **using**, directivas de preprocesado, definiciones de miembros, de tipos o de espacios de nombres.

- La etiqueta indicada no pueda pertenecer a un bloque de instrucciones anidado dentro del bloque desde el que se usa el **goto** ni que etiquete a instrucciones de otro método diferente a aquél en el cual se encuentra el **goto** que la referencia.
- Para etiquetar una instrucción de modo que pueda ser destino de un salto con goto basta precederla del nombre con el que se la quiera etiquetar seguido de dos puntos (**:**) Por ejemplo, el siguiente código demuestra cómo usar **goto** y definir una etiqueta:

```
using System;

class HolaMundoGoto
{
  public static void Main(string[] args)
  {
    for (int i=0; i<args.Length; i++)
    {
      if (args[i] != "salir")
        Console.WriteLine(args[i]);
      else
        goto fin:
    }
    fin: ;
  }
}
```

Este programa de ejemplo lo que hace es mostrar por pantalla todos los argumentos que se le pasen como parámetros, aunque si alguno fuese salir entonces se dejaría de mostrar argumentos y se aborta la ejecución de la aplicación. Véase además que este ejemplo pone de manifiesto una de las utilidades de la instrucción nula, ya que si no se hubiese escrito tras la etiqueta fin el programa no compilaría en tanto que toda etiqueta ha de preceder a alguna instrucción (aunque sea la instrucción nula)

Nótese que al fin y al cabo los usos de **goto** dentro de instrucciones **switch** que se vieron al estudiar dicha instrucción no son más que variantes del uso general de **goto**, ya que **default:** no es más que una etiqueta y **case** <valor>**:** puede verse como una etiqueta un tanto especial cuyo nombre es **case** seguido de espacios en blanco y un valor. En ambos casos, la etiqueta indicada ha de pertenecer al mismo switch que el **goto** usado y no vale que éste no la contenga pero la contenga algún **switch** que contenga al **switch** del **goto**.

El uso de **goto** dentro de sentencias **try**, tiene las mismas restricciones que **break**, **continue** y **return**: antes de salir con un **goto** de un **try** se ejecutará siempre su bloque **finally** y no es posible forzar a saltar fuera de un **finally**.

Instrucción throw

La **instrucción** **throw** ya se ha visto que se usa para lanzar excepciones de este modo:

```
throw <objetoExcepciónALanzar>;
```

En caso de que no se indique ningún <objetoExcepciónALanzar> se relanzará el que se estuviese tratando en ese moment, aunque esto sólo es posible si el **throw** se ha escrito dentro del código de tratamiento asociado a alguna cláusula **catch**.

Como ya se ha explicado a fondo esta instrucción en este mismo tema, para más información sobre basta remitirse al epígrafe *Excepciones* de este tema.

Otras instrucciones

Las instrucciones vistas hasta ahora son comunes a muchos lenguajes de programación. Sin embargo, en C# también se ha incluido un buen número de nuevas instrucciones propias de este lenguaje. Estas instrucciones se describen en los siguientes apartados:

Instrucciones checked y unchecked

Las instrucciones checked y unchecked permiten controlar la forma en que tratarán los desbordamientos que ocurran durante la realización de operaciones aritméticas con tipos básico enteros. Funcionan de forma similar a los operadores **checked** y **unchecked** ya vistos en el *Tema 4: Aspectos léxicos*, aunque a diferencia de éstos son aplicables a bloques enteros de instrucciones y no a una única expresión. Así, la **instrucción** **checked** se usa de este modo:

```
checked
  <instrucciones>
```

Todo desbordamiento que se produzca al realizar operaciones aritméticas con enteros en <instrucciones> provocará que se lance una excepción **System.OverflowException**. Por su parte, la **instrucción** **unchecked** se usa así:

```
unchecked
  <instrucciones>
```

En este caso, todo desbordamiento que se produzca al realizar operaciones aritméticas con tipos básicos enteros en <instrucciones> será ignorado y lo que se hará será tomar el valor resultante de quedarse con los bits menos significativos necesarios.

Por defecto, en ausencia de estas instrucciones las expresiones constantes se evalúan como si se incluyesen dentro de una instrucción **checked** y las que no constantes como si se incluyesen dentro de una instrucción **unchecked**. Sin embargo, a través de la opción **/checked** del compilador es posible tanto hacer que por defecto se comprueben los desbordamiento en todos los casos para así siempre poder detectarlos y tratarlos

Desde Visual Studio.NET, la forma de controlar el tipo de comprobaciones que por defecto se harán es a través de **View -> Propety Pages -> Configuration Settings -> Build -> Check for overflow underflow**.

El siguiente código muestra un ejemplo de cómo usar ambas instrucciones:

```
using System;

class Unchecked
{
  static short x = 32767;   // Valor maximo del tipo short

  public static void Main()
  {
    unchecked
    {
      Console.WriteLine((short) (x+1));  // (1)
      Console.WriteLine((short) 32768); // (2)
    }
  }
}
```

En un principio este código compilaría, pero los desbordamientos producidos por el hecho de que 32768 no es un valor que se pueda representar con un **short** (16 bits con signo) provocaría que apareciese por pantalla dicho valor truncado, mostrándose:

```
-32768
-32678
```

Sin embargo, si sustituyésemos la instrucción **unchecked** por **checked**, el código anterior ni siquiera compilaría ya que el compilador detectaría que se va a producir un desbordamiento en `(2)` debido a que 32768 es constante y no representable con un **short**.

Si eliminamos la instrucción `(2)` el código compilaría ya que `(x+1)` no es una expresión constante y por tanto el compilador no podría detectar desbordamiento al compilar. Sin embargo, cuando se ejecutase la aplicación se lanzaría una **System.OverflowException**.

Instrucción lock

La **instrucción** **lock** es útil en aplicaciones concurrentes donde múltiples hilos pueden estar accediendo simultáneamente a un mismo recurso, ya que lo que hace es garantizar que un hilo no pueda acceder a un recurso mientras otro también lo esté haciendo. Su sintaxis es la siguiente:

```
lock (<objeto>)
  <instrucciones>
```

Su significado es el siguiente: ningún hilo puede ejecutar las `<instrucciones>` del bloque indicado si otro las está ejecutando, y si alguno lo intenta se quedará esperando hasta que acabe el primero. Esto también afecta a bloques de `<instrucciones>` de cualquier otro **lock** cuyo `<objeto>` sea el mismo. Este `<objeto>` ha de ser de algún tipo referencia.

En realidad, la instrucción anterior es equivalente a hacer:

```
System.Threading.Monitor.Enter(<objeto>);
try
  <instrucciones>
finally
```

```
{
  System.Threading.Monitor.Exit(<objeto>);
}
```

Sin embargo, usar **lock** tiene dos ventajas: es más compacto y eficiente (**<objeto>** sólo se evalúa una vez)

Una buena forma de garantizar la exclusión mutua durante la ejecución de un método de un cierto objeto es usando **this** como <objeto>. En el caso de que se tratase de un método de tipo, en tanto que **this** no tiene sentido dentro de estos métodos estáticos una buena alternativa sería usar el objeto **System.Type** que representase a ese tipo. Por ejemplo:

```
class C
{
  public static void F()
  {
    lock(typeof(C))
    {
      // ... Código al que se accede exclusivamente
    }
  }
}
```

Instrucción using

La **instrucción** **using** facilita el trabajo con objetos que tengan que ejecutar alguna tarea de limpieza o liberación de recursos una vez que termine de ser útiles. Aunque para estos menesteres ya están los destructores, dado su carácter indeterminista puede que en determinadas ocasiones no sea conveniente confiar en ellos para realizar este tipo de tareas. La sintaxis de uso de esta instrucción es la siguiente:

```
using (<tipo> <declaraciones>)
  <instrucciones>
```

En <declaraciones> se puede indicar uno o varios objetos de tipo <tipo> separados por comas. Estos objetos serán de sólo lectura y sólo serán accesibles desde <instrucciones>. Además, han de implementar la interfaz **System.IDisposable** definida como sigue:

```
interface IDisposable
{
  void Dispose()
}
```

En la implementación de **Dispose()** se escribiría el código de limpieza necesario, pues el significado de **using** consiste en que al acabar la ejecución de <instrucciones>, se llama automáticamente al método **Dispose()** de los objetos definidos en <declaraciones>

Hay que tener en cuenta que la llamada a Dispose() se hace sea cual sea la razón de que se deje de ejecutar las <instrucciones> Es decir, tanto si se ha producido una excepción como si se ha acabado su ejecución normalmente o con una instrucción de salto, **Dispose()** es siempre llamado. En realidad una instrucción **using** como:

```
using (R1 r1 = new R1())
{
  r1.F();
}
```

Es tratada por el compilador como:

```
{
  R1 r1 = new R1()
  try
  {
    r1.F();
  }
  finally
  {
    if (r1!=null)
      ((IDisposable) r1).Dispose();
  }
}
```

Si se declarasen varios objetos en <declaraciones>, a **Dispose()** se le llamaría en el orden inverso a como fueron declarados. Lo mismo ocurre si se anidasen varias instrucciones **using**: primero se llamaría al **Dispose()** de las variables declaradas en los **using** internos y luego a las de los externos. Así, estas dos instrucciones son equivalentes:

```
using (Recurso obj = new Recurso(), obj2= new Recurso())
{
  r1.F();
  r2.F();
}

using (Recurso obj = new Recurso())
{
  using (Recurso obj2= new Recurso())
  {
    r1.F();
    r2.F();
  }
}
```

El siguiente ejemplo resume cómo funciona la sentencia **using**:

```
using System;

class A:IDisposable
{
  public void Dispose()
  {
    Console.WriteLine("Llamado a Dispose() de {0}", Nombre);
  }

  public A(string nombre)
  {
    Nombre = nombre;
  }

  string Nombre;
}

class Using
{
  public static void Main()
  {
    A objk = new A("objk");
```

```
    using (A obj1 = new A("obj1"), obj2 = new A("objy"))
    {
      Console.WriteLine("Dentro del using");
    }
    Console.WriteLine("Fuera del using");
  }
}
```

La salida por pantalla resultante de ejecutar este ejemplo será:

```
Dentro del using
Llamando a Dispose() de objy
Llamando a Dispose() de obj1
Fuera del using
```

Como se deduce de los mensajes de salida obtenidos, justo antes de salirse del **using** se llama a los métodos **Dispose()** de los objetos declarados en la sección **<declaraciones>** de dicha instrucción y en el mismo orden en que fueron declarados.

Instrucción fixed

La **instrucción** **fixed** se utiliza para fijar objetos en memoria de modo que el recolector de basura no pueda moverlos durante la ejecución de un cierto bloque de instrucciones.

Esta instrucción sólo tiene sentido dentro de regiones de código inseguro, concepto que se trata en el *Tema 18: Código inseguro*, por lo que será allí es donde se explique a fondo cómo utilizarla. Aquí sólo diremos que su sintaxis de uso es:

```
fixed(<tipoPunteros> <declaracionesPunterosAFijar>)
  <instrucciones >
```

Principio Página

El lenguaje de programación C#

En esta página:

Tema 17: Atributos

Concepto de atributo

Un **atributo** es información que se puede añadir a los metadatos de un módulo de código. Esta información puede ser referente tanto al propio módulo o el ensamblado al que peretenezca como a los tipos de datos en definidos en él, sus miembros, los parámetros de sus métodos, los bloques `set` y `get` de sus propiedades e indizadores o los bloques `add` y `remove` de sus eventos.

En C# se incluyen numerosos modificadores que nos permiten asociar información a los metadatos de un módulo. Por ejemplo, con los modificadores `public`, `protected`, `private`, `internal` o `protected internal` podemos añadir información sobre la visibilidad de los tipos del módulo y de sus miembros. Pues bien, los atributos pueden verse como un mecanismo mediante el cual el programador puede crear sus propios modificadores.

Un ejemplo de atributo podría ser uno llamado Ayuda que pudiese prefijar las definiciones de miembros de tipos e indicase cuál es la URL donde se pudiese encontrar información detallada con ayuda sobre el significado del miembro prefijado.

Utilización de atributos

Para colocar un atributo a un elemento basta prefijar la definición de dicho elemento con una estructura de esta forma:

```
[<nombreAtributo>(<parámetros>)]
```

Esta estructura ha de colocarse incluso antes que cualquier modificador que pudiese acompañar la definición del elemento a atribuir.

Los parámetros de una atributo pueden ser opcionales, y si se usa sin especificar valores para sus parámetros no hay porqué que usar paréntesis vacíos como en las llamadas a métodos, sino que basta usar el atributo indicando sólo la sintaxis `[<nombreAtributo>]`

Los parámetros de un atributo pueden ser de dos tipos:

- **Parámetros sin nombre:** Se usan de forma similar a los parámetros de los métodos, sólo que no pueden contar con modificadores **ref** u **out**.

- **Parámetros con nombre:** Son opcionales y pueden colocarse en cualquier posición en la lista de <parámetros> del atributo. Lo último se debe a que a la hora de darles valor se usa la sintaxis <nombreParámetro>=<valor>, por lo que el compilador no dependerá de su posición a la hora de determinar a qué parámetro se le está dando cada valor.

Para evitar conflictos entre parámetros con nombre y parámetros sin nombre, los primeros siempre se han de incluir después de los segundos, no siendo posible mezclarlos indiscriminadamente.

Si se desean especificar varios atributos para un mismo elemento se pueden indicar todos ellos entre unos mismos corchetes serapados por comas. Es decir, de la forma:

```
[<atributo1>(<parametros1>), <atributo2>(<parámetros>), ...]
```

Aunque también sería posible especificarlos por separado. O sea, de esta otra forma:

```
[<atributo1>(<parametros1>)] [<atributo2>(parámetros>)] ...
```

Hay casos en los que por la ubicación del atributo no se puede determinar de manera unívoca a cuál elemento se le desea aplicar, ya que podría ser aplicable a varios. En esos casos, para evitar ambigüedades lo que se hace es usar el atributo prefijando su nombre de un **indicador de tipo de elemento**, quedando así la sintaxis a usar:

```
[<indicadorElemento>:<nombreAtributo> (<parámetros>)]
```

Aunque cada implementación de C# puede incluir sus propios indicadores de tipo de elemento, todas ellas incluirán al menos los siguientes:

- **assembly:** Indica que el atributo se aplica al ensamblado en que se compile el código fuente que lo contenga. Al definir atributos de ensamblado es obligatorio incluir este indicador, ya que estos atributos se colocan precediendo cualquier definición de clase o espacio de nombres y si no se incluyesen se confundiría con atributos de tipo, que se colocan en el mismo sitio.

- **module:** Indica que el atributo se aplica al módulo en que se compile el código fuente que lo contenga. Al igual que el indicador **assembly**, hay que incluirlo siempre para definir este tipo de atributos porque si no se confundirían con atributos de tipo, ya que también se han de ubicar precediendo las definiciones de clases y espacios de nombres.

- **type:** Indica que el atributo se aplica al tipo cuya definición precede. En realidad no hace falta utilizarlo, pues es lo que por defecto se considera para todo atributo que preceda a una definición de tipo. Sin embargo, se ha incluido por consistencia con el resto de indicadores de tipo de atributo y porque puede resultar conveniente incluirlo ya que explicitarlo facilita la lectura del código.

- **return:** Indica que el atributo se aplica a un valor de retorno de un método, operador, bloque **get**, o definición de delegado. Si no se incluyese se consideraría que se aplica a la definición del método, operador, bloque **get** o delegado, ya que estos atributos se colocan antes de la misma al igual que los atributos de valores de retorno.

- **param:** Indica que el atributo se aplica a un parámetro de un método. Si no se incluyese al definir bloques **set**, **add** o **remove** se consideraría que el atributo se refiere a los bloques en sí y no al parámetro **value** en ellos implícito.

- **method:** Indica que el atributo se aplica al método al que precede. En realidad no es necesario usarlo porque, como se dice en la explicación de los indicadores **param** y **return**, es lo que se considera por defecto. Sin embrago, y como pasaba con **type**, se incluye por consistencia y porque puede ser buena idea incluirlo para facilitar la legibilidad del código con su explicitación.

- **event:** Indica que el atributo se aplica al evento a cuya definición precede. En realidad no es necesario incluirlo porque es lo que se considera por defecto, pero nuevamente se ha incluido por consistencia y para facilitar la lectura del código.

- **property:** Indica que el atributo se aplica a la propiedad a cuya definición precede. Éste también es es un indicador innecesario e incluido tan sólo por consistencia y para facilitar la legibilidad del código.

- **field:** Indica que el atributo se aplica al cuya definición precede. Como otros indicadores, sólo se incluye por consistencia y para hacer más legible el código.

Definición de nuevos atributos

Especificación del nombre del atributo

Se considera que un atributo es toda aquella clase que derive de **System.Attribute**. Por tanto, para definir un nuevo tipo de atributo hay que crear una clase que derive de ella. Por convenio, a este tipo de clases suele dárseles nombres acabados en **Attribute**, aunque a la hora de usarlas desde C# es posible obviar dicho sufijo. Un ejemplo de cómo definir una atributo llamado Ayuda es:

```
using System;

class AyudaAttribute:Attribute
{}
```

Y ejemplos de cómo usarlo prefijando la definición de clases son:

```
[Ayuda] class A
{}

[AyudaAttribute] class B
{}
```

Puede darse la circunstancia de que se haya definido un atributo con un cierto nombre sin sufijo `Attribute` y otro que si lo tenga. Como es lógico, en ese caso cuando se use el atributo sin especificar el sufijo se hará referencia a la versión sin sufijo y cuando se use con sufijo se hará referencia a la versión con sufijo.

Especificación del uso de un atributo

Por defecto cualquier atributo que se defina puede preceder la definición de cualquier elemento del lenguaje. Si se desea limitar a qué definiciones puede preceder es necesario prefijar la clase que lo define con un atributo especial llamado **System.AttributeUsage**. Este atributo consta de los siguientes parámetros con nombre:

- **AllowMultiple:** Por defecto cada atributo sólo puede aparecer una vez prefijando a cada elemento. Dándole el valor **true** a este parámetro se considerará que puede aparecer múltiples veces.

- **Inherited:** Por defecto los atributos aplicados a una clase no son heredados en sus clases hijas. Dándole el valor **true** a este parámetros se consigue que sí lo sean.

Aparte de estos dos parámetros, **AttributeUsage** también puede contar con un parámetro opcional sin nombre que indique a qué tipos de definiciones puede preceder. Por defecto se considera que un atributo puede preceder a

cualquier elemento, lo que es equivalente a darle el valor **AttributeTargets.All** a este parámetro. Sin embrago es posible especificar otras posibilidades dándole valores de la enumeración **System.AttributeTargets**, que son los que se recogen en la **Tabla 9**:

Valor de AttributeTargets	Significa que el atributo puede preceder a...
All	Cualquier definición
Assembly	Definiciones de espacio de nombres, considerándose que el atributo se refiere al ensamblado en general.
Module	Definiciones de espacio de nombres, considerándose que el atributo se refiere al módulo en su conjunto.
Class	Definiciones de clases
Delegate	Definiciones de delegados
Interface	Definiciones de interfaces
Struct	Definiciones de estructuras
Enum	Definiciones de enumeraciones
Field	Definiciones de campos
Method	Definiciones de métodos
Constructor	Definiciones de constructores
Property	Definiciones de propiedades o indizadores
Event	Definiciones de eventos
Parameter	Definiciones de parámetros de métodos
ReturnValue	Definiciones de valores de retorno de métodos

Tabla 9: Valores de **AttributeTargets**

Es posible combinar varios de estos valores mediante operaciones lógicas "or" (carácter |) Por ejemplo, si queremos definir el atributo Ayuda anterior de modo que sólo pueda ser usado para prefijar definiciones de enumeraciones o de clases se haría:

```
[AttributeUsage(AttributeTargets.Class | AttributeTargetes.Enum)]
class Ayuda:Attribute
{}
```

Es importante resaltar que **AttributeUsage** sólo puede incluirse precediendo definiciones de otros atributos (o sea, de clases derivadas de **System.Attribute**)

Especificación de parámetros válidos

Se considera que los parámetros sin nombre que puede tomar un atributo son aquellos que se especifiquen como parámetros en el constructor del tipo que lo define, y que sus parámetros con nombre serán las propiedades y campos públicos, no estáticos y de lectura/escritura definidos en dicho tipo.

Un ejemplo de cómo definir el atributo Ayuda anterior de modo que tome un parámetro sin nombre con la URL que indique dónde encontrar la ayuda sobre el miembro o clase al que precede y un parámetro con nombre llamado Autor que indique quién es el autor de esa documentación es:

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Enum)]
class Ayuda:Attribute
{
  private string autor;
  private string url;

  public Ayuda(string URL)
```

```
    { url=URL; }

    public string Autor
    {
      set {autor = value;}
      get {return autor;}
    }
  }
```

Ejemplos de usos válidos de este atributo son:

```
[Ayuda("http://www.josan.com/Clases/A.html")]
class A {}

[Ayuda("http://www.josan.com/Clases/B.html", Autor="José Antonio González Seco")]
class B {}
```

Los tipos válidos de parámetros, tanto con nombre como sin él, que puede tomar un atributo son: cualquier tipo básico excepto **decimal** y los tipos enteros sin signo, cualquier enumeración pública, **System.Type** o tablas unidimensionales de elementos de cualquiera de los anteriores tipos válidos.

Lectura de atributos en tiempo de ejecución

Para acceder a los metadatos de cualquier ensamblado se utilizan las clases del espacio de nombres **System.Reflection**. Este espacio de nombres es inmenso y explicar cómo utilizarlo queda fuera del alcance de este libro, aunque de todos modos a continuación se darán unas ideas básicas sobre cómo acceder a través de sus tipos a los atributos incluidos en los ensamblados.

La clave para acceder a los atributos se encuentra en el método estático de la clase **System.Attribute** llamado **Attribute[] GetCustomAttributes(<x> objetoReflexivo)**, donde <x> es el tipo de **System.Reflection** que representa a los elementos cuyos atributos se desea obtener. Los posibles tipos son: **Assembly**, que representa ensamblados, **Module** que representa módulos, **MemberInfo** que representa miembros (incluidos tipos, que al fin y al cabo son miembros de espacios de nombres), y **ParameterInfo** que representa parámetros. El parámetros tomado por este método será el objeto que represente al elemento en concreto cuyos metadatos se quieren obtener.

Como se ve, **GetCustomAttributes()** devuelve una tabla con los atributos en forma de objetos **Attribute**, que es la clase base de todos los atributos, por lo que si a partir de ellos se desease acceder a características específica de cada tipo de atributo habría que aplicar downcasting como se comentó en el *Tema 5: Clases* (para asegurase de que las conversiones se realicen con éxito recuérdese que se puede usar el operador **is** para determinar cuál es el verdadero tipo de cada atributo de esta tabla)

Para obtener el objeto **Assembly** que representa al ensamblado al que pertenezca el código que se esté ejecutando se usa el método **Assembly GetExecutingAssembly()** de la clase **Assembly**, que se usa tal y como se muestra:

```
Assembly ensamblado = Assembly.GetExecutingAssembly();
```

Otra posibilidad sería obtener ese objeto **Assembly** a partir del nombre del fichero donde se encuentre almacenado el ensamblado. Para ello se usa el método **Assembly LoadFrom(string rutaEnsamblado)** de la clase **Assembly** como se muestra:

```
Assembly ensamblado = Assembly.LoadFrom("josan.dll");
```

Una vez obtenido el objeto que representa a un ensamblado pueden obtenerse los objetos **Module** que representan a los módulos que lo forman a través de su método **Module[] GetModules()**.

A partir del objeto **Module** que representa a un módulo puede obtenerse los objetos **Type** que representan a sus

tipos a través de su método **Type[] GetTypes()** Otra posibilidad sería usar el operador **typeof** ya visto para obtener el **Type** que representa a un tipo en concreto sin necesidad de crear objetos **Module** o **Assembly**.

En cualquier caso, una vez obtenido un objeto **Type**, a través de sus métodos **FieldInfo[] GetFields()**, **MethodInfo[] GetMethods()**, **ConstructorInfo[] GetConstructors()**, **EventInfo[] GetEvents[]** y **PropertyInfo[] GetProperties()** pueden obtenerse los objetos reflexivos que representan, de manera respectiva, a sus campos, métodos, constructores, eventos y propiedades o indizadores. Tanto todos estos objetos como los objetos **Type** derivan de **MemberInfo**, por lo que pueden ser pasados como parámetros de **GetCustomAttributes()** para obtener los atributos de los elementos que representan.

Por otro lado, a través de los objetos **MethodInfo** y **ConstructorInfo**, es posible obtener los tipos reflexivos que representan a los parámetros de métodos y constructores llamando a su método **ParameterInfo[] GetParameters()** Además, en el caso de los objetos **MethodInfo** también es posible obtener el objeto que representa al tipo de retorno del método que representan mediante su propiedad **Type ReturnType** .

En lo referente a las propiedades, es posible obtener los objetos **MethodInfo** que representan a sus bloques **get** y **set** a través de los métodos **MethodInfo GetSetMethod()** y **MethodInfo GetSetMethod()** de los objetos **PropertyInfo** que las representan. Además, para obtener los objetos reflexivos que representen a los índices de los indizadores también se dispone de un método **ParamterInfo[] GetIndexParameters()**

Y en cuanto a los eventos, los objetos **EventInfo** disponen de métodos **MethodInfo GetAddMethod()** y **MethodInfo GetRemoveMethod()** mediante los que es posible obtener los objetos reflexivos que representan a sus bloques **add** y **remove**.

A continuación se muestra un programa de ejemplo que lo que hace es mostrar por pantalla el nombre de todos los atributos que en él se hayan definido:

```
using System.Reflection;
using System;

[assembly: EjemploEnsamblado]
[module: EjemploModulo]
[AttributeUsage(AttributeTargets.Method)]
class EjemploMétodo:Attribute
{}

[AttributeUsage(AttributeTargets.Assembly)]
class EjemploEnsamblado:Attribute
{}

[AttributeUsage(AttributeTargets.Module)]
class EjemploModulo:Attribute
{}

[AttributeUsage(AttributeTargets.Class)]
class EjemploTipo:Attribute
{}

[AttributeUsage(AttributeTargets.Field)]
class EjemploCampo:Attribute
{}

[EjemploTipo]
class A
{
  public static void Main()
  {
    Assembly ensamblado = Assembly.GetExecutingAssembly();
```

```
    foreach (Attribute atributo in Attribute.GetCustomAttributes(ensamblado))
      Console.WriteLine("ENSAMBLADO: {0}",atributo);
    foreach (Module modulo in ensamblado.GetModules())
    {
      foreach(Attribute atributo in Attribute.GetCustomAttributes(modulo))
        Console.WriteLine("MODULO: {0}", atributo);
      foreach (Type tipo in modulo.GetTypes())
      {
        foreach(Attribute atributo in Attribute.GetCustomAttributes(tipo))
          Console.WriteLine("TIPO: {0}", atributo);

        foreach (FieldInfo campo in tipo.GetFields())
          muestra("CAMPO", campo);

        foreach (MethodInfo metodo in tipo.GetMethods())
          muestra("METODO", metodo);

        foreach (EventInfo evento in tipo.GetEvents())
          muestra("EVENTO", evento);

        foreach (PropertyInfo propiedad in tipo.GetProperties())
          muestra("PROPIEDAD", propiedad);

        foreach (ConstructorInfo constructor in tipo.GetConstructors())
          muestra("CONSTRUCTOR",constructor);
      }
    }

  }

  static private void muestra(string nombre, MemberInfo miembro)
  {
    foreach (Attribute atributo in Attribute.GetCustomAttributes(miembro))
      Console.WriteLine("{0}: {1}", nombre, atributo);
  }

}
```

Lo único que hace el **Main()** de este programa es obtener el **Assembly** que representa el ensamblado actual y mostrar todos sus atributos de ensamblado. Luego obtiene todos los **Module**s que representa a los módulos de dicho ensamblado y muestra todos los atributos de módulo de cada uno. Además, de cada módulo se obtienen todos los **Type**s que representan a los tipos en él definidos y se muestran todos sus atributos; y de cada tipo se obtienen los objetos reflexivos que representan a sus diferentes tipos de miembros y se muestran los atributos atributos de cada miembro.

Aparte del método **Main()** en el ejemplo se han incluido definiciones de numerosos atributos de ejemplo aplicables a diferentes tipos de elemento y se han diseminado a lo largo del fuente varios usos de estos atributos. Por ello, la salida del programa es:

```
ENSAMBLADO: EjemploEnsamblado
ENSAMBLADO: System.Diagnostics.DebuggableAttribute
MODULO EjemploModulo
TIPO: System.AttributeUsageAttribute
TIPO: System.AttributeUsageAttribute
TIPO: System.AttributeUsageAttribute
TIPO: System.AttributeUsageAttribute
TIPO: System.AttributeUsageAttribute
TIPO: EjemploTipo
METODO: EjemploMétodo
```

Nótese que aparte de los atributos utilizados en el código fuente, la salida del programa muestra que el compilador ha asociado a nivel de ensamblado un atributo extra llamado **Debuggable**. Este atributo incluye información sobre si pueden aplicarse optimizaciones al compilar JIT el ensamblado o si se ha de realizar una traza de su ejecución. Sin embargo, no conviene fiarse de su implementación ya que no está documentado por Microsoft y puede cambiar en futuras versiones de la plataforma .NET.

Atributos de compilación

Aunque la mayoría de los atributos son interpretados en tiempo de ejecución por el CLR u otras aplicaciones, hay una serie de atributos que tienen un significado especial en C# y condicionan el proceso de compilación. Estos son los que se explican a continuación.

Atributo System.AttributeUsage

Ya hemos visto en este mismo tema que se utiliza para indicar dónde se pueden colocar los nuevos atributos que el programador defina, por lo que no se hará más hincapié en él.

Atributo System.Obsolete

Se puede usar prefijando cualquier elemento de un fichero de código fuente para indicar que el elemento precedido está obsoleto. Puede tomar dos parámetros sin nombre:

- Un primer parámetro de tipo **string** que contenga una cadena con un mensaje a mostrar cuando al compilar se detecte que se ha usado el elemento obsoleto.
- Un segundo parámetro de tipo **bool** que indique si se ha de producir un aviso o un error cuando se detecte el uso del elemento obsoleto. Por defecto se muestra un aviso, pero si se le da el valor **true** a este parámetro se producirá un error.

El siguiente ejemplo muestra como usar este atributo:

```
using System;

class Obsoleta
{
  [Obsolete("No usar f(), que está obsoleto.", true)]
  public static void f()
  {}

  public static void Main()
  {
    f();
  }
}
```

Cuando se compile este programa el compilador producirá el siguiente mensaje de error:

```
obsolete.cs(11,17): error CS0619: 'Obsoleta.f()' is obsolete: no usr f(), que está
obsoleto.
```

Si se hubiese usado **Obsolete** sin segundo parámetros, entonces se mostraría el aviso:

```
obsolete.cs(11,17): warning CS0618: 'Obsoleta.f()' is obsolete: no usr f(), que está
obsoleto.
```

Atributo System.Diagnostics.Conditional

Este atributo sólo puede prefijar definiciones de métodos, y permite definir si las llamadas al método prefijado se

han de compilar o no. Puede usarse múltiples veces prefijando a un mismo método y toma un parámetro sin nombre de tipo **string**. Sólo se compilarán aquellas llamadas al método tales que en el momento de hacerlas esté definida alguna directiva de preprocesado con el mismo nombre que el parámetro de alguno de los atributos **Conditional** que prefijen la definición de ese método.

Como se ve, este atributo ces una buena forma de simplificar la escritura de código que se deba compilar condicionalmente, ya que evita tener varias directivas **#if** que encierren cada llamada al método cuya ejecución se desea controlar. Sin embargo, **Conditional** no controla la compilación de ese método, sino sólo las llamadas al mismo.

El siguiente ejemplo muestra cómo usar **Conditional**:

```
using System;
using System.Diagnostics;

class Condicional
{
  [Conditional("DEBUG")]
  public static void F()
  { Console.WriteLine("Ff()"); }

  public static void Main()
  {
    F();
  }
}
```

Sólo si compilamos el este código definiendo la constante de preprocesado DEBUG se mostrará por pantalla el mensaje F() En caso contrario, nunca se hará la llamada a F()

Hay que precisar que en realidad **Conditional** no puede preceder a cualquier definición de método, sino que en su colocación hay impuestas ciertas restricciones especiales:

- El método ha de tener un tipo de retorno **void**. Esto se debe a que si tuviese otro se podría usar su valor de retorno como operando en expresiones, y cuando no fuesen compiladas sus llamadas esas expresiones podrían no tener sentido y producir errores de compilación.

- Si se aplica a un método virtual todas sus redefiniciones lo heredan, siendo erróneo aplicárselo explícitamente a una de ellas. Esto debe a que en tiempo de compilación puede no saberse cuál es el verdadero tipo de un objeto, y si unas redefiniciones pudiesen ser condicionales y otras no, no podría determinarse al compilar si es condicional la versión del método a la que en cada caso se llame.

- No puede atribuirse a métodos definidos en interfaces ni a implementaciones de métodos de interfaces, pues son también virtuales y podrían reimplementarse.

Principio Página

En esta página:

Tema 18: Código inseguro

Concepto de código inseguro

Código inseguro es todo aquél fragmento de código en C# dentro del cual es posible hacer uso de punteros.

Un **puntero** en C# es una variable que es capaz de almacenar direcciones de memoria. Generalmente suele usarse para almacenar direcciones que almacenen objetos, por lo en esos casos su significado es similar al de variables normales de tipos referencia. Sin embargo, los punteros no cuentan con muchas de las restricciones de éstas a la hora de acceder al objeto. Por ejemplo, al accederse a los elementos de una tabla mediante un puntero no se pierde tiempo en comprobar que el índice especificado se encuentre dentro de los límites de la tabla, lo que permite que el acceso se haga más rápidamente.

Aparte de su mayor eficiencia, también hay ciertos casos en que es necesario disponer del código inseguro, como cuando se desea hacer llamadas a funciones escritas en lenguajes no gestionados cuyos parámetros tengan que ser punteros.

Es importante señalar que los punteros son una excepción en el sistema de tipos de .NET, ya que no derivan de la clase primigenia `System.Object`, por lo que no dispondrán de los métodos comunes a todos los objetos y una variable `object` no podrá almacenarlos (tampoco existen procesos similares al boxing y unboxing que permitan simularlo)

Compilación de códigos inseguros

El uso de punteros hace el código más proclive a fallos en tanto que se salta muchas de las medidas incluidas en el acceso normal a objetos, por lo que es necesario incluir ciertas medidas de seguridad que eviente la instroducción accidental de esta inseguridad

La primera medida tomada consiste en que explícitamente hay que indicar al compilador que deseamos compilar código inseguro. Para ello, al compilador de línea de comandos hemos de pasarle la opción `/unsafe`, como se muestra el ejemplo:

```
csc códigoInseguro.cs /unsafe
```

Si no se indica la opción unsafe, cuando el compilador detecte algún fuente con código inseguro producirá un mensaje de error como el siguiente:

```
códigoInseguro(5,23): error CS0277: unsafe code may only appear if compiling with /unsafe
```

En caso de que la compilación se vaya a realizar a través de Visual Studio.NET, la forma de indicar que se desea compilar código inseguro es activando la casilla **View -> Property Pages -> Configuration Properties -> Build -> Allow unsafe code blocks**

Marcación de códigos inseguros

Aparte de forzarse a indicar explícitamente que se desea compilar código inseguro, C# también obliga a que todo uso de código inseguro que se haga en un fichero fuente tenga que ser explícitamente indicado como tal. A las zonas de código donde se usa código inseguro se les denomina **contextos inseguros**, y C# ofrece varios mecanismos para marcar este tipo de contextos.

Una primera posibilidad consiste en preceder un bloque de instrucciones de la palabra reservada **unsafe** siguiendo la siguiente sintaxis:

```
unsafe <instrucciones>
```

En el código incluido en <instrucciones> podrá definirse variables de tipos puntero y podrá hacerse uso de las mismas. Por ejemplo:

```
public void f()
{
  unsafe
  {
    int *x;
  }
}
```

Otra forma de definir contextos inseguros consiste en añadir el modificador **unsafe** a la definición de un miembro, caso en que dentro de su definición se podrá hacer uso de punteros. Así es posible definir campos de tipo puntero, métodos con parámetros de tipos puntero, etc. El siguiente ejemplo muestra cómo definir dos campos de tipo puntero. Nótese sin embargo que no es posible definir los dos en una misma línea:

```
struct PuntoInseguro
{
  public unsafe int *X;   //   No es válido hacer public unsafe int *X, Y;
  public unsafe int *Y;   //  Tampoco lo es hacer public unsafe int *X, *Y;
}
```

Obviamente, en un método que incluya el modificador **unsafe** no es necesario preceder con dicha palabra sus bloques de instrucciones inseguros.

Hay que tener en cuenta que el añadido de modificadores unsafe es completamente inocuo. Es decir, no influye para nada en cómo se haya de redefinir y si un método **Main()** lo tiene sigue siendo un punto de entrada válido.

Una tercera forma consiste en añadir el modificador **unsafe** en el definición de un tipo, caso en que todas las definiciones de miembros del mismo podrán incluir código inseguro sin necesidad de añadir a cada una el modificador **unsafe** o preceder sus bloques de instrucciones inseguras de la palabra reservada **unsafe**. Por ejemplo:

```
unsafe struct PuntoInseguro
{
  public int * X, *Y;
```

```
}
```

Definición de punteros

Para definir una variable puntero de un determinado tipo se sigue una sintaxis parecida a la usada para definir variables normales sólo que al nombre del tipo se le postpone un símbolo de asterisco (*) O sea, un puntero se define así:

```
<tipo> * <nombrePuntero>;
```

Por ejemplo, una variable puntero llamada a que pueda almacenar referencias a posiciones de memoria donde se almacenen objetos de tipo **int** se declara así:

```
int * a;
```

En caso de quererse declarar una tabla de punteros, entonces el asterisco hay que incluirlo tras el nombre del tipo pero antes de los corchetes. Por ejemplo, una tabla de nombre t que pueda almacenar punteros a objetos de tipo **int** se declara así:

```
int*[]  t;
```

Hay un tipo especial de puntero que es capaz de almacenar referencias a objetos de cualquier tipo. Éstos punteros de declara indicando **void** como **<tipo>**. Por ejemplo:

```
void * punteroACualquierCosa;
```

Hay que tener en cuenta que en realidad lo que indica el tipo que se dé a un puntero es cuál es el tipo de objetos que se ha de considerar que se almacenan en la dirección de memoria almacenada por el puntero. Si se le da el valor **void** lo que se está diciendo es que no se desea que se considere que el puntero apunta a ningún tipo específico de objeto. Es decir, no se está dando información sobre el tipo apuntado.

Se pueden declarar múltiples variables locales de tipo puntero en una misma línea. En ese caso el asterisco sólo hay que incluirlo antes del nombre de la primera. Por ejemplo:

```
int * a, b; // a y b son de tipo int *
            // No sería válido haberlas definido como int *a, *b;
```

Hay que tener en cuenta que esta sintaxis especial para definir en una misma definición varios punteros de un mismo tipo sólo es válida en definiciones de variables locales. Al definir campos no sirve y hay que dar para cada campo una definición independiente.

El recolector de basura no tiene en cuenta los datos a los que se referencie con punteros, pues ha de conocer cuál es el objeto al referenciado por cada variable y un puntero en realidad no tiene porqué almacenar referencias a objetos de ningún tipo en concreto. Por ejemplo, pueden tenerse punteros **int *** que en realidad apunten a objeto **char**, o punteros **void *** que no almacenen información sobre el tipo de objeto al que debería considerarse que apuntan, o punteros que apunte a direcciones donde no hayan objetos, etc.

Como el recolector de basura no trabaja con punteros, no es posible definir punteros de tipos que se almacenen en memoria dinámica o contengan miembros que se almacenen en memoria dinámica, ya que entonces podría ocurrir que un objeto sólo referenciado a través de punteros sea destruido por considear el recolector que nadie le referenciaba. Por ello, sólo es válido definir punteros de tipos cuyos objetos se puedan almacenar completamente en pila, pues la vida de estos objetos no está controlada por el recolector de basura sino que se destruyen cuando se abandona el ámbito donde fueron definidos.

En concreto, los únicos punteros válidos son aquellos cuyo tipos sean tipos valor básicos, enumeraciones o estructuras que no contengan campos de tipos referencias. También pueden definirse punteros de tipos puntero, como muestra este ejemplo de declaración de un puntero a puntero de tipo **int** llamando punteroApuntero:

```
int ** punteroApuntero;
```

Obviamente la anidación puede hacerse a cualquier nivel de profundidad, pudiéndose definir punteros a punteros a punteros, o punteros a punteros a punteros a punteros, etc.

Manipulación de punteros

Obtención de dirección de memoria. Operador &

Para almacenar una referencia a un objeto en un puntero se puede aplicar al objeto el operador prefijo **&**, que lo que hace es devuelver la dirección que en memoria ocupa el objeto sobre el que se aplica. Un ejemplo de su uso para inicializar un puntero es:

```
int x =10;
int * px = &x;
```

Este operador no es aplicable a expresiones constantes, pues éstas no se almacenan en ninguna dirección de memoria específica sino que se incrustan en las instrucciones. Por ello, no es válido hacer directamente:

```
int px = &10;  // Error 10 no es una variable con dirección propia
```

Tampoco es válido aplicar **&** a campos **readonly**, pues si estos pudiesen ser apuntados por punteros se correría el riesgo de poderlos modificar ya que a través de un puntero se accede a memoria directamente, sin tenerse en cuenta si en la posición accedida hay algún objeto, por lo que mucho menos se considerará si éste es de sólo lectura.

Lo que es sí válido almacenar en un puntero es la dirección de memoria apuntada por otro puntero. En ese caso ambos punteros apuntarían al mismo objeto y las modificaciones a éste realizadas a través de un puntero también afectarían al objeto visto por el otro, de forma similar a como ocurre con las variables normales de tipos referencia. Es más, los operadores relacionales típicos (**==**, **!=**, **<**, **>** ,**<=** y **>=**) se han redefinido para que cuando se apliquen entre dos punteros de cualesquiera dos tipos lo que se compare sean las direcciones de memoria que estos almacenan. Por ejemplo:

```
int x = 10;
int px = &x;
int px2 = px;                    // px y px2 apuntan al objeto almacenado en x
Console.WriteLine( px == px2);  // Imprime por pantalla True
```

En realidad las variables sobre las que se aplique **&** no tienen porqué estar inicializadas. Por ejemplo, es válido hacer:

```
private void f()
{
  int x;
  unsafe
  { int px = &x;}
}
```

Esto se debe a que uno de los principales usos de los punteros en C# es poderlos pasar como parámetros de funciones no gestionadas que esperen recibir punteros. Como muchas de esas funciones han sido programadas para inicializar los contenidos de los punteros que se les pasan, pasarles punteros inicializados implicaría perder tiempo innecesariamente en inicializarlos.

Acceso a contenido de puntero. Operador *

Un puntero no almacena directamente un objeto sino que suele almacenar la dirección de memoria de un objeto (o sea, apunta a un objeto) Para obtener a partir de un puntero el objeto al que apunta hay que aplicarle al mismo el operador prefijo *****, que devuelve el objeto apuntado. Por ejemplo, el siguiente código imprime en pantalla un 10:

```
int x = 10;
int * px= &x;
Console.WriteLine(*px);
```

Es posible en un puntero almacenar **null** para indicar que no apunta a ninguna dirección válida. Sin embargo, si luego se intenta acceder al contenido del mismo a través del operador * se producirá generalmente una excepción de tipo NullReferenceException (aunque realmente esto depende de la implementación del lenguaje) Por ejemplo:

```
int * px = null;
Console.WriteLine(*px); // Produce una NullReferenceException
```

No tiene sentido aplicar * a un puntero de tipo **void *** ya que estos punteros no almacenan información sobre el tipo de objetos a los que apuntan y por tanto no es posible recurperarlos a través de los mismos ya que no se sabe cuanto espacio en memoria a partir de la dirección almacenada en el puntero ocupa el objeto apuntado y, por tanto, no se sabe cuanta memoria hay que leer para obtenerlo.

Acceso a miembro de contenido de puntero. Operador ->

Si un puntero apunta a un objeto estructura que tiene un método **F()** sería posible llamarlo a través del puntero con:

```
(*objeto).F();
```

Sin embargo, como llamar a objetos apuntados por punteros es algo bastante habitual, para facilitar la sintaxis con la que hacer esto se ha incluido en C# el operador **->**, con el que la instrucción anterior se escribiría así:

```
objeto->f();
```

Es decir, del mismo modo que el operador **.** permite acceder a los miembros de un objeto referenciado por una variable normal, **->** permite acceder a los miembros de un objeto referenciado por un puntero. En general, un acceso de la forma `O -> M` es equivalente a hacer **(*O).**M. Por tanto, al igual que es incorrecto aplicar * sobre punteros de tipo **void ***, también lo es aplicar **->**

Conversiones de punteros

De todo lo visto hasta ahora parece que no tiene mucho sentido el uso de punteros de tipo **void *** Pues bien, una utilidad de este tipo de punteros es que pueden usarse como almacén de punteros de cualquier otro tipo que luego podrán ser recuperados a su tipo original usando el operador de conversión explícita. Es decir, igual que los objetos de tipo **object** pueden almacenar implícitamente objetos de cualquier tipo, los punteros **void *** pueden almacenar punteros de cualquier tipo y son útiles para la escritura de métodos que puedan aceptar parámetros de cualquier tipo de puntero.

A diferencia de lo que ocurre entre variables normales, las conversiones entre punteros siempre se permiten, al realizarlas nunca no se comprueba si son válidas. Por ejemplo, el siguiente código es válido:

```
char c = 'A';
char* pc = &c;
void* pv = pc;
int* pi = (int*)pv;
int i = *pi;            // Almacena en 16 bits del char de pv
                        // + otros 16 indeterminados
Console.WriteLine(i);
*pi = 123456;           // Machaca los 32 bits apuntados por pi
```

En este código pi es un puntero a un objeto de tipo **int** (32 bits), pero en realidad el objeto al que apunta es de tipo **char** (16 bits), que es más pequeño. El valor que se almacene en **i** es en principio indefinido, pues depende de lo que hubiese en los 16 bits extras resultantes de tratar **pv** como puntero a **int** cuando en realidad apuntaba a un **char**.

Del mismo modo, conversiones entre punteros pueden terminar produciendo que un puntero apunte a un objeto de mayor tamaño que los objetos del tipo del puntero. En estos casos, el puntero apuntaría a los bits menos significativos del objeto apuntado.

También es posible realizar conversiones entre punteros y tipos básicos enteros. La conversión de un puntero en un tipo entero devuelve la dirección de memoria apuntada por el mismo. Por ejemplo, el siguiente código muestra por

pantalla la dirección de memoria apuntada por **px**:

```
int x = 10;
int *px = &10;
Console.WriteLine((int) px);
```

Por su parte, convertir cualquier valor entero en un puntero tiene el efecto de devolver un puntero que apunte a la dirección de memoria indicada por ese número. Por ejemplo, el siguiente código hace que **px** apunte a la dirección 1029 y luego imprime por pantalla la dirección de memoria apuntada por **px** (que será 1029):

```
int *px = (int *) 10;
Console.WriteLine((int) px);
```

Nótese que aunque en un principio es posible hacer que un puntero almacene cualquier dirección de memoria, si dicha dirección no pertenece al mismo proceso que el código en que se use el puntero se producirá un error al leer el contenido de dicha dirección. El tipo de error ha producir no se indica en principio en la especifricación del lenguaje, pero la implementación de Microsoft lanza una referencia **NullReferenceException**. Por ejemplo, el siguiente código produce una excepción de dicho tipo al ejecturase:

```
using System;

class AccesoInválido
{
  public unsafe static void Main()
  {
    int * px = (int *) 100;
    Console.Write(*px); // Se lanza NullReferenceException
  }
}
```

Aritmética de punteros

Los punteros se suelen usar para recorrer tablas de elementos sin necesidad de tener que comprobarse que el índice al que se accede en cada momento se encuentra dentro de los límites de la tabla. Por ello, los operadores aritméticos definidos para los punteros están orientados a facilitar este tipo de recorridos.

Hay que tener en cuenta que todos los operadores aritméticos aplicables a punteros dependen del tamaño del tipo de dato apuntado, por lo que no son aplicables a punteros **void *** ya que estos no almacenan información sobre dicho tipo. Esos operadores son:

- **++** y **--**: El operador **++** no suma uno a la dirección almacenada en un puntero, sino que le suma el tamaño del tipo de dato al que apunta. Así, si el puntero apuntaba a un elemento de una tabla pasará a apuntar al siguiente (los elementos de las tablas se almacenan en memoria consecutivamente) Del mismo modo, **--** resta a la dirección almacenada en el puntero el tamaño de su tipo de dato. Por ejemplo, una tabla de 100 elementos a cuyo primer elemento inicialmente apuntase **pt** podría recorrerse así:

  ```
  for (int i=0; i<100; i++)
  Console.WriteLine("Elemento {0}={1}", i, (*p)++);
  ```

  El problema que puede plantear en ciertos casos el uso de **++** y **--** es que hacen que al final del recorrido el puntero deje de apuntar al primer elemento de la tabla. Ello podría solucionarse almacenando su dirección en otro puntero antes de iniciar el recorrido y restaurándola a partir de él tras finalizarlo.

- **+** y **-**: Permiten solucionar el problema de **++** y **--** antes comentado de una forma más cómoda basada en sumar o restar un cierto entero a los punteros. **+** devuelve la dirección resultante de sumar a la dirección almacenada en el puntero sobre el que se aplica el tamaño del tipo de dicho puntero tantas veces como indique el entero sumado. **-** tiene el mismo significado pero restando dicha cantidad en vez de sumarla. Por ejemplo, usando **+** el bucle anterior podría reescribrise así:

```
for (int i=0; i<100; i++)
  Console.WriteLine("Elemento {0}={1}", i, *(p+i));
```

El operador **-** también puede aplicarse entre dos punteros de un mismo tipo, caso en que devuelve un **long** que indica cuántos elementos del tipo del puntero pueden almacenarse entre las direcciones de los punteros indicados.

- **[]**: Dado que es frecuente usar **+** para acceder a elementos de tablas, también se ha redefinido el operador **[]** para que cuando se aplique a una tabla haga lo mismo y devuelva el objeto contenido en la dirección resultante. O sea ***(p+i)** es equivalente a **p[i]**, con lo que el código anterior equivale a:

```
for (int i=0; i<100; i++)
  Console.WriteLine("Elemento {0}={1}", i, p[i]);
```

No hay que confundir el acceso a los elementos de una tabla aplicando **[]** sobre una variable de tipo tabla normal con el acceso a través de un puntero que apunte a su primer elemento. En el segundo caso no se comprueba si el índice indicado se encuentra dentro del rango de la tabla, con lo que el acceso es más rápido pero también más proclive a errores difíciles de detectar.

Finalmente, respecto a la aritmética de punteros, hay que tener en cuenta que por eficiencia, en las operaciones con punteros nunca se comprueba si se producen desbordamientos, y en caso de producirse se truncan los resultados sin avisarse de ello mediante excepciones. Por eso hay que tener especial cuidado al operar con punteros no sea que un desbordamiento no detectado cause errores de causas difíciles de encontrar.

Operadores relacionados con código inseguro

Operador sizeof. Obtención de tamaño de tipo

El operador unario y prefijo **sizeof** devuelve un objeto **int** con el tamaño en bytes del tipo de dato sobre el que se aplica. Sólo puede aplicarse en contextos inseguros y sólo a tipos de datos para los que sea posible definir punteros, siendo su sintaxis de uso:

```
sizeof(<tipo>)
```

Cuando se aplica a tipos de datos básicos su resultado es siempre constante. Por ello, el compilador optimiza dichos usos de **sizeof** sustituyéndolos internamente por su valor (inlining) y considerando que el uso del operador es una expresión constante. Estas constantes correspondientes a los tipos básicos son las indicadas en la **Tabla 10**:

Tipos	Resultado
sbyte, byte, bool	1
short, ushort, char	2
int, uint, float	4
long, ulong, double	8

Tabla 10: Resultados de sizeof para tipos básicos

Para el resto de tipos a los que se les puede aplicar, **sizeof** no tiene porqué devuelver un resultado constante sino que los compiladores pueden alinear en memoria las estructuras incluyendo bits de relleno cuyo número y valores sean en principio indeterminado. Sin embargo, el valor devuelto por **sizeof** siempre devolverá el tamaño en memoria exacto del tipo de dato sobre el que se aplique, incluyendo bits de relleno si los tuviese.

Nótese que es fácil implementar los operadores de aritmética de punteros usando **sizeof**. Para ello, **++** se definiría como añadir a la dirección almacenada en el puntero el resultado de aplicar **sizeof** a su tipo de dato, y **--** consistiría en restarle dicho valor. Por su parte, el operador **+** usado de la forma `P + N` (`P` es un puntero de tipo `T` y `N` un entero) lo que devuelve es el resultado de añadir al puntero `sizeof(T)*N`, y `P - N` devuelve el resultado de restarle

`sizeof(T)*N`. Por último, si se usa `-` para restar dos punteros `P1` y `P2` de tipo `T`, ello es equivalente a calcular `(((long)P1) - ((long)P2)))/sizeof(T)`

Operador stackalloc. Creación de tablas en pila.

Cuando se trabaja con punteros puede resultar interesante reservar una zona de memoria en la pila donde posteriormente se puedan ir almacenando objetos. Precisamente para eso está el operador **`stackalloc`**, que se usa siguiéndose la siguiente sintaxis:

```
stackalloc <tipo>[<número>]
```

`stackalloc` reserva en pila el espacio necesario para almacenar contiguamente el número de objetos de tipo `<tipo>` indicado en `<número>` (reserva `sizeof(<tipo>)*<número>` bytes) y devuelve un puntero a la dirección de inicio de ese espacio. Si no quedase memoria libre suficiente para reservarlo se produciría una excepción **`System.StackOverflowException`**.

`stackalloc` sólo puede usarse para inicializar punteros declarados como variables locales y sólo en el momento de su declaración.. Por ejemplo, un puntero pt que apuntase al principio de una región con capacidad para 100 objetos de tipo **`int`** se declararía con:

```
int * pt = stackalloc int[100];
```

Sin embargo, no sería válido hacer:

```
int * pt;
pt = stackalloc int[100];   // ERROR: Sólo puede usarse stackalloc en declaraciones
```

Aunque pueda parecer que `stackalloc` se usa como sustituto de **`new`** para crear tablas en pila en lugar de en memoria dinámica, no hay que confundirse: **`stackalloc`** sólo reserva un espacio contiguo en pila para objetos de un cierto tipo, pero ello no significa que se cree una tabla en pila. Las tablas son objetos que heredan de **`System.Array`** y cuentan con los miembros heredados de esta clase y de **`object`**, pero regiones de memoria en pila reservadas por **`stackalloc`** no. Por ejemplo, el siguiente código es inválido.

```
int[] tabla;
int * pt = stackalloc int[100];
tabla = *pt;  // ERROR: El contenido de pt es un int, no una tabla (int[])
Console.WriteLine(pt->Length); // ERROR: pt no apunta a una tabla
```

Sin embargo, gracias a que como ya se ha comentado en este tema el operador **`[]`** está redefinido para trabajar con punteros, podemos usarlo para acceder a los diferentes objetos almacenados en las regiones reservadas con **`stackalloc`** como si fuesen tablas. Por ejemplo, este código guarda en pila los 100 primeros enteros y luego los imprime:

```
class Stackalloc
{
  public unsafe static void Main()
  {
    int * pt = stackalloc int[100];
    for (int i=0; i<100; i++)
      pt[i] = i;
    for(int i=0; i<100; i++)
      System.Console.WriteLine(pt[i]);
  }
}
```

Nótese que, a diferencia de lo que ocurriría si pt fuese una tabla, en los accesos con `pt[i]` no se comprueba que `i` no supere el número de objetos para los que se ha reservado memoria. Como contrapartida, se tiene el inconveniente de que al no ser `pt` una tabla no cuenta con los métodos típicos de éstas y no puede usarse **`foreach`** para recorrerla.

Otra ventaja de la simulación de tablas con **stackalloc** es que se reserva la memoria mucho más rápido que el tiempo que se tardaría en crear una tabla. Esto se debe a que reservar la memoria necesaria en pila tan sencillo como incrementar el puntero de pila en la cantidad correspondiente al tamaño a reservar, y no hay que perder tiempo en solicitar memoria dinámica. Además, **stackalloc** no pierde tiempo en inicializar con algún valor el contenido de la memoria, por lo que la "tabla" se crea antes pero a costa de que luego sea más inseguro usarla ya que hay que tener cuidado con no leer trozos de ella antes de asignarles valores válidos.

Fijación de variables apuntadas

Aunque un puntero sólo puede apuntar a datos de tipos que puedan almacenarse completamente en pila (o sea, que no sean ni objetos de tipos referencia ni estructuras con miembros de tipos referencia), nada garantiza que los objetos apuntado en cada momento estén almacenados en pila. Por ejemplo, las variables estáticas de tipo **int** o los elementos de una tabla de tipo **int** se almacenan en memoria dinámica aún cuando son objetos a los que se les puede apuntar con punteros.

Si un puntero almacena la dirección de un objeto almacenado en memoria dinámica y el recolector de basura cambia al objeto de posición tras una compactación de memoria resultante de una recolección, el valor almacenado en el puntero dejará de ser válido. Para evitar que esto ocurra se puede usar la instrucción **fixed**, cuya sintaxis de uso es:

```
fixed(<tipo> <declaraciones>)
  <instrucciones>
```

El significado de esta instrucción es el siguiente: se asegura que durante la ejecución del bloque de `<instrucciones>` indicado el recolector de basura nunca cambie la dirección de ninguno de los objetos apuntados por los punteros de tipo `<tipo>` declarados. Estas `<declaraciones>` siempre han de incluir una especificación de valor inicial para cada puntero declarado, y si se declaran varios se han de separar con comas.

Los punteros declarados en `<declaraciones>` sólo existirán dentro de `<instrucciones>`, y al salir de dicho bloque se destruirán. Además, si se les indica como valor inicial una tabla o cadena que valga **null** saltará una **NullReferenceException**. También hay que señalar que aunque sólo pueden declarase punteros de un mismo tipo en cada **fixed**, se puede simular facilmente la declaración de punteros de distintos tipos anidando varios **fixed**.

Por otro lado, los punteros declarados en `<declaraciones>` son de sólo lectura, ya que si no podría cambiárseles su valor por el de una dirección de memoria no fijada y conducir ello a errores difíciles de detectar.

Un uso frecuente de **fixed** consiste en apuntar a objetos de tipos para los que se puedan declarar punteros pero que estén almacenados en tablas, ya que ello no se puede hacer directamente debido a que las tablas se almacenan en memoria dinámica. Por ejemplo, copiar usando punteros una tabla de 100 elementos de tipo **int** en otra se haría así:

```
class CopiaInsegura
{
  public unsafe static void Main()
  {
    int[] tOrigen = new int[100];
    int[] tDestino = new int[100];
    fixed (int * pOrigen=tOrigen, pDestino=tDestino)
    {
      for (int i=0; i<100; i++)
        pOrigen[i] = pDestino[i];
    }
  }
}
```

Como puede deducirse del ejemplo, cuando se inicializa un puntero con una tabla, la dirección almacenada en el puntero en la zona `<declaraciones>` del **fixed** es la del primer elemento de la tabla (también podría haberse hecho `pOrigen = &tOrigen[0]`), y luego es posible usar la aritmética de punteros para acceder al resto de elementos a partir de la dirección del primero ya que éstos se almacenan consecutivamente.

Al igual que tablas, también puede usarse **fixed** para recorrer cadenas. En este caso lo que hay que hacer es

inicializar un puntero de tipo **char *** con la dirección del primer carácter de la cadena a la que se desee que apunte tal y como muestra este ejemplo en el que se cambia el contenido de una cadena "Hola" por "XXXX":

```
class CadenaInsegura
{
  public unsafe static void Main()
  {
    string s="Hola";

    Console.WriteLine("Cadena inicial: {0}", s);
    fixed (char * ps=s)
    {
      for (int i=0;i<s.Length;i++)
        ps[i] = 'A';
    }
    Console.WriteLine("Cadena final: {0}", s);
  }
}
```

La salida por pantalla de este último programa es:

```
Hola
AAAA
```

La ventaja de modificar la cadena mediante punteros es sin ellos no sería posible hacerlo ya que el indizador definido para los objetos **string** es de sólo lectura.

Cuando se modifiquen cadenas mediante punteros hay que tener en cuenta que, aunque para facilitar la comunicación con código no gestionado escrito en C o C++ las cadenas en C# también acaban en el carácter '\0', no se recomienda confiar en ello al recorrerlas con punteros porque '\0' también puede usarse como carácter de la cadena. Por ello, es mejor hacer como en el ejemplo y detectar su final a través de su propiedad **Length**.

Hay que señalar que como fixed provoca que no pueda cambiarse de dirección a ciertos objetos almacenados en memoria dinámica, ello puede producir la generación de huecos en memoria dinámica, lo que tiene dos efectos muy negativos:

- El recolector de basura está optimizado para trabajar con memoria compactada, pues si todos los objetos se almacenan consecutivamente en memoria dinámica crear uno nuevo es tan sencillo como añadirlo tras el último. Sin embargo, **fixed** rompe esta consecutividad y la creación de objetos en memoria dinámica dentro de este tipo de instrucciones es más lenta porque hay que buscar huecos libres.

- Por defecto, al eliminarse objetos de memoria durante una recolección de basura se compacta la memoria que queda ocupada para que todos los objetos se almacenen en memoria dinámica. Hacer esto dentro de sentencias **fixed** es más lento porque hay que tener en cuenta si cada objeto se puede o no mover.

Por estas razones es conveniente que el contenido del bloque de instrucciones de una sentencia **fixed** sea el mínimo posible, para que así el **fixed** se ejecute lo antes posible.

Principio Página

El lenguaje de programación C#

En esta página:

Tema 19: Documentación XML

Concepto y utilidad de la documentación XML

La documentación de los tipos de datos creados siempre ha sido una de las tareas más pesadas y aburridas a las que un programador se ha tenido que enfrentar durante un proyecto, lo que ha hecho que muchas veces se escriba de manera descuidada y poco concisa o que incluso que no se escriba en absoluto. Sin embargo, escribirla es una tarea muy importante sobre todo en un enfoque de programación orientada a componentes en tanto que los componentes desarrollados muchas veces van a reutilizados por otros. E incluso para el propio creador del componente puede resultar de inestimable ayuda si en el futuro tiene que modificarlo o usarlo y no recuerda exáctamente cómo lo implementó.

Para facilitar la pesada tarea de escribir la documentación, el compilador de C# es capaz de generarla automáticamente a partir de los comentarios que el progamador escriba en los ficheros de código fuente.

El hecho de que la documentación se genere a partir de los fuentes permite evitar que se tenga que trabajar con dos tipos de documentos por separado (fuentes y documentación) que deban actualizarse simultáneamente para evitar incosistencias entre ellos derivadas de que evolucionen de manera separada ya sea por pereza o por error.

El compilador genera la documentación en XML con la idea de que sea fácilmente legible para cualquier aplicación. Para facilitar su legibilidad a humanos bastaría añaderle una hoja de estilo XSL o usar alguna aplicación específica encargada de leerla y mostrarla de una forma más cómoda para humanos.

Aunque explicar XML y XSL queda fuera del alcance del libro, en este tema se resumirán brevemente tanto XML como la forma de aplicar hojas XSL a ficheros XML.

Introducción a XML

Antes de continuar es necesario hacer una pequeña introducción a XML ya que que es el lenguaje en que se han de escribir los comentarios especiales de documentación. Si ya conoce este lenguaje puede saltarse este epígrafe.

XML (Extensible Markup Language) es un **metalenguaje de etiquetas**, lo que significa que es un lenguaje que se utiliza para definir lenguajes de etiquetas. A cada lenguaje creado con XML se le denomina **vocabulario XML**, y la documentación generada por el compilador de C# está escrita en un vocabulario de este tipo.

Los comentarios a partir de los que el compilador generará la documentación han de escribirse en XML, por lo que han de respetar las siguientes reglas comunes a todo documento XML bien formado:

- La información ha de incluirse dentro de **etiquetas**, que son estructuras de la forma:

  ```
  <<etiqueta>> <contenido> </<etiqueta>
  ```

  En <etiqueta> se indica cuál es el nombre de la etiqueta a usar. Por ejemplo:

  ```
  <EtiquetaEjemplo> Esto es una etiqueta de ejemplo </EtiquetaEjemplo>
  ```

  Como <contenido> de una etiqueta puede incluirse tanto texto plano (es el caso del ejemplo) como otras etiquetas. Lo que es importante es que toda etiqueta cuyo uso comience dentro de otra también ha de terminar dentro de ella. O sea, no es válido:

  ```
  <Etiqueta1> <Etiqueta2> </Etiqueta1></Eetiqueta2>
  ```

  Pero lo que sí sería válido es:

  ```
  <Etiqueta1> <Etiqueta2> </Etiqueta2></Etiqueta1>
  ```

  También es posible mezclar texto y otras etiquetas en <contenido>. Por ejemplo:

  ```
  <Etiqueta1> Hola <Etiqueta2> a </Etiqueta2> todos </Etiqueta1>
  ```

- XML es un lenguaje sensible a mayúsculas, por lo que si una etiqueta se abre con una cierta capitalización, a la hora de cerrarla habrá que usar exactamente la misma.
- Es posible usar la siguiente sintaxis abreviada para escribir etiquetas sin <contenido>:

  ```
  <<etiqueta>/>
  ```

  Por ejemplo:

  ```
  <<EtiquetaSinContenidoDeEjemplo>/>
  ```

- En realidad en la <etiqueta> inicial no tiene porqué indicarse sólo un identificador que sirva de nombre para la etiqueta usada, sino que también pueden indicarse **atributos** que permitan configurar su significado. Estos atributos se escriben de la forma <nombreAtributo> = "<valor>" y separados mediante espacios. Por ejemplo:

  ```
  <EtiquetaConAtributo AtributoEjemplo="valor1" >
      Etiqueta de ejemplo que incluye un atributo
  </EtiquetaConAtributo>

  <EtiquetaSinContenidoYConAtributo AtributoEjemplo="valor2" />
  ```

- Sólo puede utilizarse caracteres ASCII, y los caracteres no ASCII (acentos, eñes, ...) o caracteres con algún significado especial en XML han de ser sustituidos por secuencias de escape de la forma &#<códigoUnicode>;. Para los caracteres más habituales también se han definido las siguientes secuencias de escape especiales:

Carácter	Secuencia de escape Unicode	Secuencia de escape especial

<	&#60;	&lt;
>	&#62;	&gt;
&	&#38;	&amp;
'	&#39;	&apos;
"	&#34;	&quot;

Tabla 11: Secuencias de espace XML de uso frecuente

Comentarios de documentación XML

Sintaxis general

Los comentarios de documentación XML se escriben como comentarios normales de una línea pero con las peculiaridades de que su primer carácter ha de ser siempre `/` y de que su contenido ha de estar escrito en XML ya que será insertado por el compilador en el fichero XML de documentación que genera. Por tanto, son comentarios de la forma:

```
/// <textoXML>
```

Estos comentarios han preceder las definiciones de los elementos a documentar. Estos elementos sólo pueden ser definiciones de miembros, ya sean tipos de datos (que son miembros de espacios de nombres) o miembros de tipos datos, y han de colocarse incluso incluso antes que sus atributos.

En `<textoXML>` el programador puede incluir cualesquiera etiquetas con el significado, contenido y atributos que considere oportunos, ya que en principio el compilador no las procesa sino que las incluye tal cual en la documentación que genera dejándo en manos de las herramientas encargadas de procesar dicha documentación la determinación de si se han usado correctamente.

Sin embargo, el compilador comprueba que los comentarios de documentación se coloquen donde deberían y que contegan XML bien formado. Si no fuese así generaría un mensaje de aviso y en la documentación generada los sustituiría por un comentario XML que explicase el tipo de error cometido.

El atributo cref

Aunque en principio los atributos de las etiquetas no tienen ningún significado predeterminado para el compilador, hay una excepción: el atributo **`cref`** siempre va a tener un significado concreto consistente en forzarlo a comprobar cuando vaya a generar la documentación si existe el elemento cuyo nombre indique y, si no es así, hacerle producir un mensaje de aviso (su nombre viene de "check reference")

Los elementos especificados en **`cref`** suelen indicarse mediante calificación completa, y pueden ser tanto nombres de miembros como de espacios de nombres. En el *Tema 6: Espacios de Nombres* ya se explicó como indicar así nombres de tipos y de espacios de nombres, mientras que para indicar el de miembros de tipos basta escribir el nombre completo del tipo donde estén definidos seguido de un punto tras el cual, dependiendo del tipo de miembro del que se trate, se escribiría :

- Si es un **campo**, **propiedad, evento** o **tipo** interno, su nombre.
- Si es un **método**, su nombre seguido de los nombres completos de los tipos de sus parámetros separados mediante comas y entre paréntesis. Estos nombres de tipos de parámetros llevan un carácter `@` concatendo al final en los parámetros **`ref`** u **`out`**, un carácter `*` al final en los que sean de tipos punteros, un símbolo `[]` por cada nivel de anidación al final de los que sean tablas unidimensionales, y una estructura de la forma `[0:,0:]` al final de los que sean tablas bidimensionales (para tablas de más dimensiones simplemente se irían añadiendo los bloques `,0:` apropiados).
- Si es un **indizador**, el identificador **`Item`** seguido de la lista de tipos de sus índices como si de los parámetros de un método se tratase
- Si es un **constructor** de objeto, el identificador **`#ctor`** seguido de la lista de tipos de sus parámetros como si de un método normal se tratase. Si el constructor fuese de tipos entonces el

identificador usado sería `#cctor`

- Si es un **destructor**, el identificador `Finalize`.
- Si es un **operador**, el identificador que represente a ese operador segudio de la lista de los tipos de sus operandos como si fuesen los parámetros de un método normal. En la **Tabla 12** se resumen los identificador que se dan a cada operador:

Operador	Identificador
+	op_Addition
-	op_Substraction
*	op_Multiply
/	op_Division
%	op_Modulus
<	op_LessThan
>	op_GreaterThan
>=	op_GreaterThanOrEqual
<=	op_LowerThanOrEqual
==	op_Equality
!=	op_Inequality
!	op_LogicalNot
&	op_BitwiseAnd
|	op_BitwiseOr
^	op_ExclusiveOr
~	op_OnesComplement
<<	op_LeftShift
>>	op_RightShift
true	op_True
false	op_False
++	op_Increment
--	op_Decrement
Conversión explícita	Op_Explict
Conversión implícita	Op_Implicit

Tabla 12: Nombres dados a operadores en documentación XML

En el caso de los operadores de conversión, tras la lista de parámetros se incluye adicionalmente un carácter `~` seguido del tipo de retorno del operador.

Para que se entienda mejor la forma en que se han de dar valores a `cref`, a continuación se muestra un fragmento de código de ejemplo en el que junto a cada definición se ha escrito un comentario con el valor que habría que darle a `cref` para referenciarla:

```
// cref="Espacio"
namespace Espacio
{
  // cref="Espacio.Clase"
  class Clase
  {
    // cref="Espacio.Clase.Campo"
    int Campo;
```

```
    // cref="Espacio.Clase.Propiedad"
    int Propiedad
    { set {} }

    // cref="Espacio.Clase.EstructuraInterna"
    struct EstructuraInterna
    {}

    // cref="Espacio.Clase.DelegadoInterno"
    public delegate int DelegadoInterno(string s, float f);

    // cref ="Espacio.Clase.Evento"
    public event DelegadoInterno Evento;

    // cref="Espacio.Clase.Metodo(System.Int32, System.Int32@,
    //       System.Int32*, System.Int32@,
    //       System.Int32[][], System.Int32[0:, 0:, 0:])"
    int Metodo(int a,  out int b, int * c, ref d, int[][] e, int[,,] f)
    {return 1;}

    // cref="Espacio.Clase.Item(System.String)"
    int this[string s]
    { set {} }

    // cref="Espacio.Clase.#ctor"
    Clase(int a)
    {}

    // cref="Espacio.Clase.#cctor"
    static Clase(int a)
    {}

    // cref="Espacio.Clase.Finalize"
    ~X()
    {}

    // cref="Espacio.Clase.op_Addition(Espacio.Clase, Espacio.Clase)"
    public static int operator +(Clase operando1, Clase operando2)
    { return 1; }

    // cref="Espacio.Clase.op_Explicit (Espacio.Clase)~System.Int32"
    public static explicit operator int(Clase fuente)
    { return 1; }
  }
}
```

En realidad no es siempre necesario usar calificación completa en el valor de **cref**. Si se referencia a un tipo desde la misma definición de espacio de nombres desde donde se le definió o que importa su espacio de nombres, no es necesario incluir dicho espacio en la referencia; y si se referencia a un miembro desde el el mismo tipo donde se definió, no es necesario incluir ni el nombre del tipo ni el de su espacio de nombres.

Etiquetas recomendadas para documentación XML

Aunque el programador puede utilizar las etiquetas estime oportunas en sus comentarios de documentación y darles el significado que quiera, Microsoft recomienda usar un juego de etiquetas concreto con significados concretos para escribir ciertos tipos de información común. Con ello se obtendría un conjunto básico de etiquetas que cualquier herramienta que trabaje con documentación XML pueda estar preparada para procesar (como veremos más adelante, el propio Visual Studio.NET da ciertos usos específicos a la información así documentada)

En los siguientes epígrafes se explican estas etiquetas recomendadas agrupándolas según su utilidad. Todas son opcionales, y no incluirlas sólo tiene el efecto de que no en la documentación resultante no se generarían las

secciones correspondientes a ellas.

Etiquetas de uso genérico

Hay una serie de etiquetas predefinidas que pueden colocarse, en cualquier orden, precediendo las definiciones de miembros en los ficheros fuente. Estas etiquetas, junto al significado recomendado para su contenido, son las explicadas a continuación:

- **<summary>**: Su contenido se utiliza para indicar un resumen sobre el significado del elemento al que precede. Cada vez que en VS.NET se use el operador **.** para acceder a algún miembro de un objeto o tipo se usará esta información para mostrar sobre la pantalla del editor de texto un resumen acerca de su utilidad.
- **<remarks>**: Su contenido indica una explicación detallada sobre el elemento al que precede. Se recomienda usar **<remarks>** para dar una explicación detallada de los tipos de datos y **<summary>** para dar una resumida de cada uno de sus miembros.
- **<example>**: Su contenido es un ejemplo sobre cómo usar el elemento al que precede.
- **<seealso>**: Se usa para indicar un elemento cuya documentación guarda alguna relación con la del elemento al que precede. No tiene contenido y el nombre del elemento al que se remite se indicá en su atributo **cref**, por lo que el compilador comprobará si existe. Para indicar múltiples documentaciones relativas a un cierto elemento basta usar una etiqueta **<seealso>** por cada una.
- **<permission>**: Se utiliza para indicar qué permiso necesita un elemento para poder funcionar. En su contenido se indica una descripción del mismo, y su atributo **cref** suele usarse para indicar el tipo que representa a ese permiso Por ejemplo:

```
/// <permission cref="System.Security.Permissions.FileIOPermission">
///     Necesita permiso de lectura/escritura en  el directorio C:\Datos
/// </permission>
```

Como con **<seealso>**, si un miembro ha de disponer varios tipos de permisos puede documentarse su definición con tantas etiquetas **<permission>** como sea necesario.

Etiquetas relativas a métodos

Además de las etiquetas uso general ya vistas, en las definiciones de métodos se pueden usar las siguientes etiquetas recomendadas adicionales para describir sus parámetros y valor de retorno:

- **<param>**: Permite documentar el significado de un parámetro de un método. En su propiedad **name** se indica el nombre del parámetro a documentar y en su contenido se describe su utilidad. Por ejemplo:

```
/// <summary> Método que muestra un texto por pantalla </summary>
/// <param name="texto">  Texto a mostrar  </param>

bool MuestraTexto(string texto)
```

Al generarse la documentación se comprueba si el método documentado dispone de algún parámetro con el nombre indicado en **name** y, como ocurre con **cref**, si no fuese así se generaría un mensaje de aviso informando de ello.

- **<paramref>**: Se usa para referenciar a parámetros de métodos. No tiene contenido y el nombre del parámetro referenciado se indica en su atributo **name**. Por ejemplo:

```
/// <summary>
///     Método que muestra por pantalla un texto con un determinado color
/// </summary>
/// <param name="texto">  Texto a mostrar </param>
/// <param  name="color">
///     Color con el que mostrar el <paramref name="texto"/> indicado
```

```
/// </param>

bool MuestraTexto(string texto, Color color)
```

Nuevamente, al generarse la documentación se comprobará si realmente el parámetro referenciado existe en la definición del método documentado y si no es así se generá un mensaje de aviso informando de ello.

- **<returns>**: Permite documentar el significado del valor de retorno de un método, indicando como contenido suyo una descripción sobre el mismo. Por ejemplo:

```
/// <summary>
///     Método que muestra por pantalla un texto con un determinado color
/// </summary>
/// <param name="texto">  Texto a mostrar </param>
/// <param  name="color">
///     Color con el que mostrar el <paramref name="texto"/> indicado
/// </param>
/// <returns> Indica si el método se ha ejecutado con éxito o no </summary>

bool MuestraTexto(string texto, Color color)
```

Etiquetas relativas a propiedades

El uso más habitual de una propiedad consiste en controlar la forma en que se accede a un campo privado, por lo que esta se comporta como si almacenase un valor. Mediante el contenido de la etiqueta **<value>** es posible describir el significado de ese valor:

```
private int edad;
/// <summary>
///     Almacena la edad de una persona. Si se le asigna una edad menor
///     que 0 la sustituye por 0.
/// </summary>
/// <value> Edad de la persona representada </value>
public int Edad
{
  set { edad  = (value<0)? 0:value }
  get { return edad; }
}
```

Etiquetas relativas a excepciones

Para documentar el significado de un tipo defindo como excepción puede incluirse un resumen sobre el mismo como contenido de una etiqueta de documentación **<exception>** que preceda a su definición. El atributo **cref** de ésta suele usarse para indicar la clase de la que deriva la excepción definida. Por ejemplo:

```
/// <exception cref="System.Exception">
///        Excepción de ejemplo creada por Josan
/// </exception>

class JosanExcepción: Exception
{}
```

Etiquetas relativas a formato

Para mejorar la forma de expresar el contenido de las etiquetas de documentación que se utilicen es posible incluir en ellas las siguientes etiquetas de formato:

- **<see>**:Se utiliza para indicar hipervínculos a otros elementos de la documentación generada. Es

una etiqueta sin contenido en la que el destino del enlace es la documentación del miembro cuyo nombre completo se indica en su atributo **cref**. Ese nombre es también el texto que las hojas de estilo sueln mostrar para representar por pantalla el enlace, por lo que los usos de esta etiqueta suelen ser de la forma:

```
/// <summary>
///    Muestra por la salida estándar el mensaje ¡Hola!.
///    Si no sabe como se escribe en pantalla puede consultar la
///    documentación del método
///    <see cref="System.Console.WriteLine"/>.
/// </summary>
public static void Saluda()
{
  Console.WriteLine("¡Hola!");
}
```

Nótese que la diferencia de **<see>** y **<seealso>** es que la primera se usa para indicar enlaces en medio de textos mientras que la otra se usa para indicar enlaces que se deseen incluir en una sección aparte tipo "Véase también".

- **<code>** y **<c>**: Ambas etiquetas se usan para delimitar textos han de ser considerarse fragmentos de código fuente. La diferencia entre ellas es que **<code>** se recomienda usar para fragmentos multilínea y **<c>** para los de una única línea; y que las hojas de estilo mostrarán el contenido de las etiquetas **<code>** respetando su espaciado y el de las etiquetas **<c>** sin respetarlo y trantando cualquier aparición consecutiva de varios caracteres de espaciado como si fuesen un único espacio en blanco.

  En general, **<code>** suele usarse dentro de etiquetas **<example>** para mostrar fragmentos de códigos de ejemplo, mientras que **<c>** suele usarse para hacer referencia a elementos puntales de los códigos fuente. Por ejemplo:

```
/// <example>
///    Este ejemplo muestra cómo llamar al método
///    <c>Cumple()</c> de esta clase:
///    <code>
///        Persona p = new Persona(...);
///        p.Cumple();
///    </code>
/// </example>
```

- **<para>**: Se usa para delimitar párrafos dentro del texto contenido en otras etiquetas, considerándose que el contenido de cada etiqueta **<para>** forma parte de un párrafo distinto. Generalmente se usa dentro de etiquetas **<remarks>**, ya que son las que suelen necesitar párrafos al tener un contenido más largo. Por ejemplo:

```
/// <remarks>
///    <para>
///        Primer párrafo de la descripción del miembro...
///    </para>
///    <para>
///        Segundo párrafo de la descripción del miembro...
///    </para>
/// </remarks>
```

- **<list>**: Se utiliza para incluir listas y tablas como contenido de otras etiquetas. Todo uso de esta etiqueta debería incluir un atributo **type** que indique el tipo de estructura se desea definir según tome uno de los siguientes valores:
  - **bullet**: Indica que se trata de una lista no numerada

- **number**: Indica que se trata de una lista numerada
- **table**: Indica que se trata de una tabla

El contenido de **<list>** dependerá del tipo de estructura representado en cada caso:

- Si se trata de una lista normal -ya sea numerada o no numerada- su contenido será una etiqueta **<item>** por cada elemento de la lista, y cada etiqueta de este tipo contendrá una etiqueta **<description>** con el texto correspondiente a ese elemento. Por ejemplo:

```
/// <list type="bullet">
///         <item>
///                     <description>
///                                 Elemento 1
///                     </description>
///         </item>
///         <item>
///                     <description>
///                                 Elemento 2
///                     </description>
///         </item>
/// </list>
```

- Si tratase de una tabla, su contenido sería similar al de las listas normales sólo que por cada fila se incluiría una etiqueta **<item>** y dentro de ésta se incliría una etiqueta **<description>** por cada columna de esa fila.

  Además, opcionalmente se podría incluir una etiqueta **<listheader>** antes de las etiquetas **<item>** donde se indicaría cuál ha de ser el texto de la cabecera de la tabla. Esta etiqueta se usa igual que las etiquetas **<item>**: incluirá una etiqueta **<description>** por cada columna.

- Por último, si fuese una lista de definiciones cada **<item>** contendría una primera etiqueta **<term>** con el nombre del elemento a definir y otra segunda etiqueta **<description>** con su definición. Opcionalmente también podria incluirse una etiqueta **<listheader>** con la cabecera de la lista. Por ejemplo:

```
/// <list type="bullet">
///         <item>
///                     <term>
///                                 Término 1
///                     </term>
///                     <description>
///                                 Descripción de término 1
///                     </description>
///         </item>
///         <item>
///                     <term>
///                                 Término 2
///                     </term>
///                     <description>
///                                 Descripción de término 2
///                     </description>
///         </item>
/// </list>
```

Generación de documentación XML

Generación a través del compilador en línea de comandos

Usando el compilador en línea de comandos puede generarse documentación sobre los tipos definidos en los fuentes a compilar usando la opción de compilación **/doc:**<fichero>. Por ejemplo, para compilar un fichero de código fuente Persona.cs y generar su documentación en Persona.xml, habría que llamar al compilador con:

```
csc persona.cs /doc:persona.xml
```

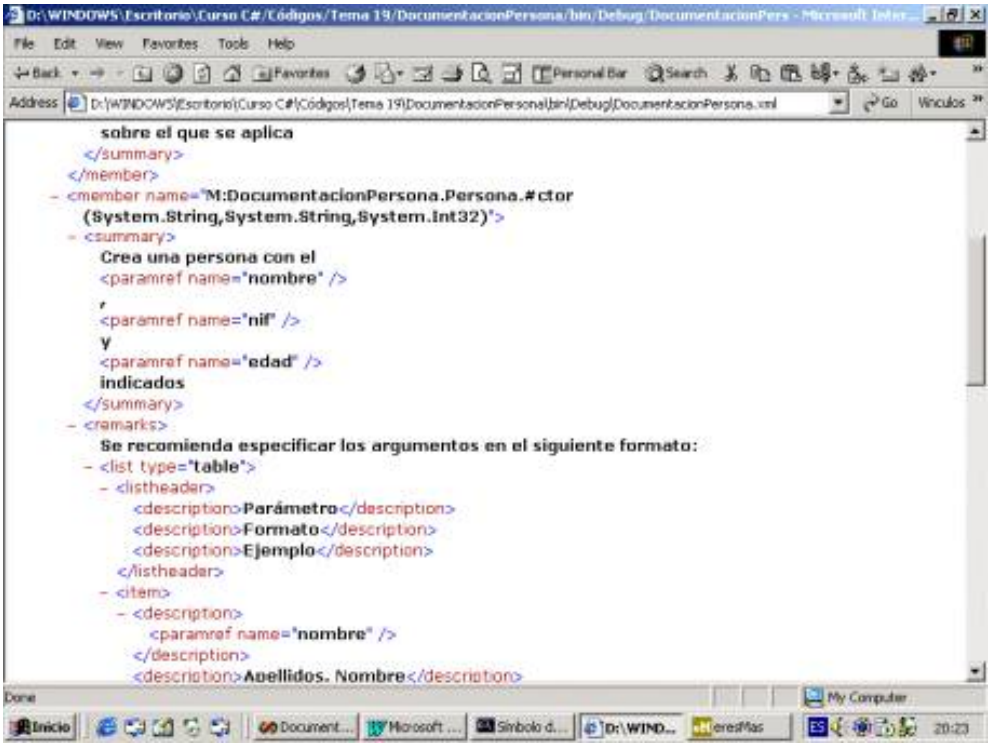
Si se abre con Internet Explorer el fichero XML así generado se verá un conjunto de etiquetas que recogen toda la información ubicada en los comentarios de documentación de los fuentes compilados. Aunque para una persona pueda resultar difícil leer esta información, para una aplicación hacerlo es muy sencillo a través de un analizador XML. Si se dese que también sea legible para humanos basta abrirlo con cualquier editor de textos y añadirle una primera línea de la forma:

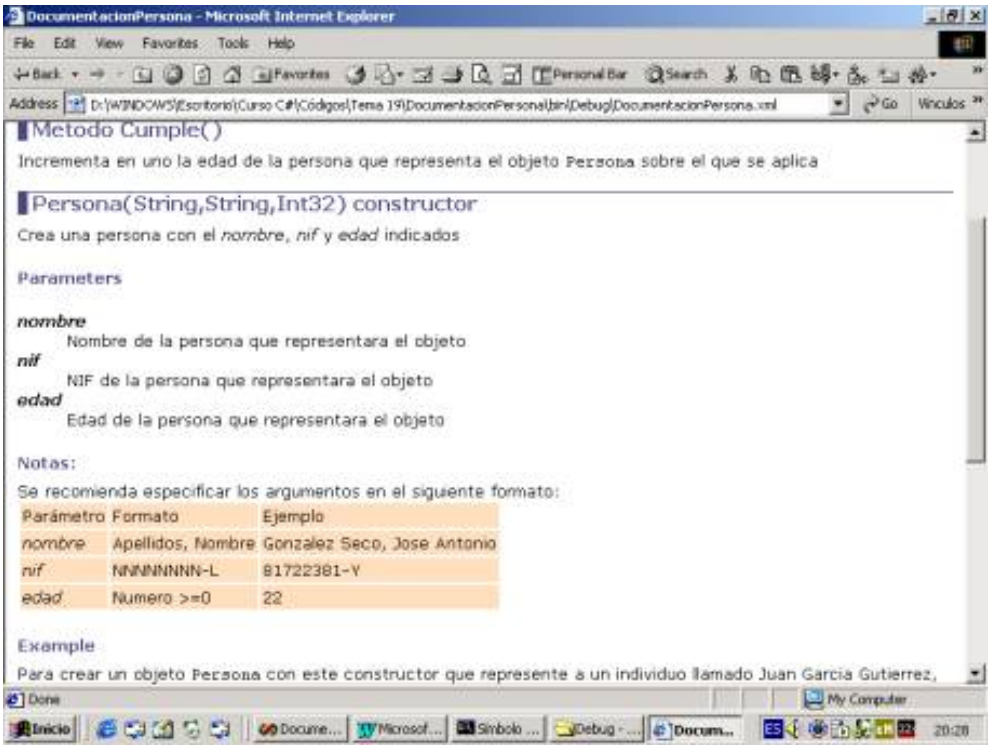
```
<?xml:stylesheet href="<ficheroXSL>" type="text/xsl"?>
```

Con esta línea se indica que se desea utilizar el fichero indicado en <ficheroXSL> como hoja de estilo XSL con la que convertir la documentación XML a algún lenguaje más fácilmente legible por humanos (generalmente, HTML). Por ejemplo, si doc.xsl es el nombre de dicho fichero XSL, bastaría escribir:

```
<?xml:stylesheet href="doc.xsl" type="text/xsl"?>
```

Para hacerse una idea de las diferencias existentes entre abrir con Internet Explorer un fichero de documentación sin hoja XSL asociada y abrir ese mismo fichero pero asociándole una hoja XSL, puede observar las siguientes ilustraciones:





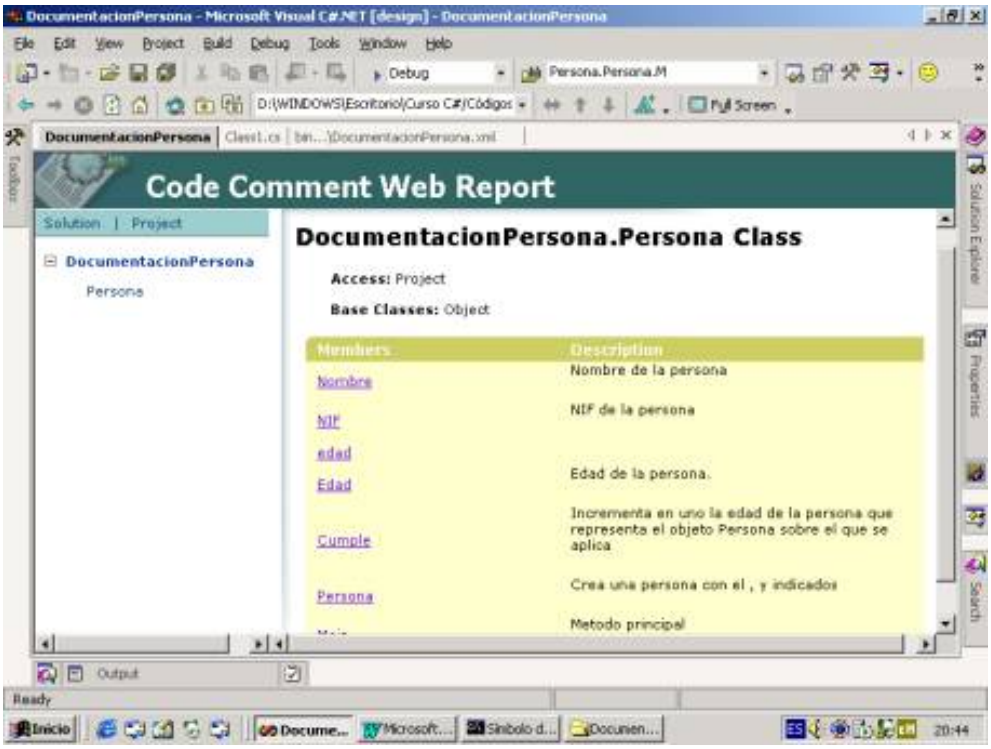
No se preocupe si no sabe escribir hojas de estilo, pues como se explica en el siguiente epígrafe, Visual Studio.NET incluye una herramienta que puede generar directamente la documentación en un HTML fácilmente legible para humanos.

Generación a través de Visual Studio.NET

Si prefiere usar Visual Studio.NET, entonces para la generación de la documentación basta señalar el proyecto a documentar en el **Solution Explorer** y escribir el nombre del fichero XML a generar en el cuadro de texto **View -> Property Pages -> Configuration Properties -> Build -> XML Documentation File**

Cuando se compile el proyecto, la documentación XML sobre el mismo se guardará en el fichero indicado en el cuadro de texto anterior. Este fichero se almacenará dentro de la subcarpeta Bin del directorio del proyecto, y si se desea poder visualizarla desde el **Solution Explorer** hay que activar en éste el botón **Show All Files**.

En principio, para conseguir visualizar esta documentación en un formato más legible para humanos podría asociársele una hoja XSL como se explicó para el caso del compilador en línea de comandos. Sin embargo, Visual Studio.NET proporciona una forma más sencilla de hacerlo a través de la herramienta ubicada en **Tools -> Build Comments Web Pages** Esta utilidad a partir de la información incluida en las etiquetas recomendadas de los comentarios del fuente genera páginas HTML que muestran la documentación del proyecto de una forma vistosa e intuitiva (ver **Ilustración**)



Estructura de la documentación XML

Ahora que ya sabemos cómo escribir comentarios de documentación y generar a partir de ellos un fichero XML con la documentación de los tipos de datos de un fichero, sólo queda estudiar cuál es concretamente la estructura de dicho fichero generado ya que entenderla es fundamental para la escritura de aplicaciones encargadas de procesarlo.

En principio, si compilamos como módulo un fuente sin comentarios de documentación pero solicitando la generación de documentación, se obtendrá el siguiente fichero XML:

```
<?xml version="1.0"?>
<doc>
  <members>
  </members>
</doc>
```

Como se ve, la primera línea del fichero es la cabecera típica de todo fichero XML en la que se indica cuál es la versión del lenguaje que utiliza. Tras ella se coloca una etiqueta **<doc>** que contendrá toda la documentación generada, y los comentarios de documentación de los miembros del fuente compilado se irían incluyendo dentro de la etiqueta **<members>** que contiene (en este caso dicha etiqueta está vacía ya que el fuente compilado carecía de comentarios de documentación)

Si hubiésemos compilado el fuente como librería o como ejecutable se habría generado un ensamblado, y a la estructura anterior se le añadiría una etiqueta adicional dentro de **<doc>** con información sobre el mismo, quedando:

```
<?xml version="1.0"?>
<doc>
  <assembly>
    <name>Persona</name>
  </assembly>
  <members>
  </members>
</doc>
```

Como se ve, dentro de la etiqueta **<assembly>** contenida en **<doc>** se indican las características del ensamblado generado. En concreto, su nombre se indica en la etiqueta **<name>** que contiene (se supone que el ensamblado se compiló con el nombre `Persona`)

Si ahora le añadimos comentarios de documentación veremos que el contenido de estos se inserta dentro de la etiqueta **<members>**, en una etiqueta **<member>** específica para cada miembro con comentarios de documentación. Por ejemplo, dado el fuente:

```
/// <summary>
///         Clase de ejemplo de cómo escribir documentacion XML
/// </summary>
class A
{
  /// <summary>
  ///         Método principal de ejemplo perteneciente a clase <see cref="A"/>
  /// </summary>
  /// <remarks>
  ///         No hace nada
  /// </remarks>
  static void Main()
  {}
}
```

La documentación XML que generara compilarlo con la opción **/doc** es:

```
<?xml version="1.0"?>
<doc>
  <assembly>
    <name>A</name>
  </assembly>
  <members>
    <member name="T:A">
      <summary>
      Clase de ejemplo de cómo escribir documentacion XML
      </summary>
    </member>
    <member name="M:A.Main">
      <summary>
      Método principal de ejemplo perteneciente a clase <see cref="T:A"/>
      </summary>
      <remarks>
      No hace nada
      </remarks>
    </member>
  </members>
</doc>
```

Como puede verse, dentro de la etiqueta **<members>** no se sigue ninguna estructura jerárquica a la hora de describir los elementos del fuente, sino que todos se describen al mismo nivel y de la misma forma: se incluye una etiqueta **<member>** por cada miembro documentado en cuyo atributo **name** se indica su nombre y en cuyo contenido se inserta el texto de sus comentarios de documentación.

Nótese que a cada elemento se le da en el atributo **name** de su etiqueta **<member>** correspondiente un identificador que lo distingue unívocamente del resto de miembros documentados y que sigue la siguiente sintaxis:

```
<indicadorElemento>:<nombreCompletamenteCalificado>
```

El <indicadorElemento> es simplemente un carácter que indica qué tipo de elemento se documenta dentro de la etiqueta **<member>**. Puede tomar estos valores:

Indicador de tipo de elemento	Tipo de elemento indicado
T	Tipo de dato

F	Campo
P	Propiedad o indizador
M	Método (incluidos operadores y contructores)
E	Evento

Tabla 13: Indicadores de tipos de elementos en documentaciones XML

Como se ve en el ejemplo, en la documentación generada se usa también la sintaxis de los valores del atributo name de las etiquetas **<member>** para representar las referencias mediante atributos **cref**. Además, cuando dicha sintaxis se usa para expresar valores de **cref** pueden usarse dos tipos de indicadores más:

Indicador de tipo de elemento	**Tipo de elemento indicado**
N	Espacio de nombres
!	Ninguno. Se genera cuando el miembro indicado en **cref** no existe.

Tabla 14: Indicadores de tipos de elementos para atributos cref

La idea que hay detrás de usar la sintaxis vista para representar elementos del fuente es proporcinar un mecanismo sencillo mediante el que las herramientas encargadas de procesar las documentaciones XML puedan determinar cuáles son los miembros documentados o referenciados y acceder, con ayuda de los tipos de **System.Reflection**, a sus metadatos asociados.

Separación entre documentación XML y código fuente

A veces puede que interesar incrustar toda la documentación en el mismo fichero que el código fuente, por ejemplo si se desea reusarla en múltiples fuentes o si es muy voluminosa e incluirla en el fuente dificultaría su legibilidad. Para estos casos se da la posiblidad de dejar la documentación en un fichero XML aparte y referenciarla en el código fuente a través de la etiqueta de documentación **<include>**, que su usa así:

```
<include file="<nombreFichero>" path="<rutaDocumentación>"/>
```

Cuando el compilador encuentre esta etiqueta al generar la documentación lo que hará será tratarla como si fuese la etiqueta del fichero <nombreFichero> indicada por la expresión **XPath** <rutaDocumentación> Por ejemplo, si se tiene el código:

```
/// <include file="otro.xml" path="Miembros/Miembro[@nombre="A"]/*"/>
class A
{}
```

En este uso de **<include>** se está indicando que se ha de insertar todo el contenido de la etiqueta <Miembro> contenida en <Miembros> cuyo atributo nombre valga A. Luego, si el contenido del fichero otro.xml es de la forma:

```
<Miembros>
  ...
  <Miembro name="A">
    <remarks>
      Ejemplo de inclusión de documentación XML externa
    </remarks>
    <example>
      Para crear un objeto de esta clase usar:
        <code>
          A obj = new A();
        </code>
    </example>
  </Miembro>
```

```
  ...
</Miembros>
```

Entonces, el compilador generará documentación como si el fuente contuviese::

```
/// <remarks>
///         Ejemplo de inclusión de documentación XML externa
/// </remarks>
/// <example>
///         Para crear un objeto de esta clase usar:
///         <code>
///                     A obj = new A();
///         </code>
/// </example>
class A
{}
```

Principio Página

El lenguaje de programación C#

En esta página:

Tema 20: El compilador de C# de Microsoft

Introducción

A lo largo de los temas anteriores se han explicando muchos aspectos sobre cómo usar el compilador de C# de Microsoft incluido en el .NET Framework SDK. Sin embargo, una vez descrito el lenguaje por completo es el momento adecuado para explicar pormenorizadamente cómo utilizarlo y qué opciones de compilación admite, pues muchas de ellas se basan en conceptos relacionados con características del lenguaje.

Por otro lado, las diferentes explicaciones dadas sobre él se han ido desperdigando a lo largo de muchos de los temas previos, por lo que es también conviene agruparlas todas en un mismo sitio de modo que sea más fácil localizarlas.

Aunque en un principio lo que se va es a explicar cómo usar el compilador en línea de comandos, dado que Visual Studio.NET también hace uso interno de él para compilar, al final del tema se incluirá un epígrafe dedicado a explicar cómo controlar desde dicha herramienta visual las opciones que se utilizarán al llamarlo.

Sintaxis general de uso del compilador

El nombre del ejecutable del compilador de C# incluido en el .NET Framework SDK es `csc.exe` y podrá encontrarlo en la carpeta `Microsoft.NET\Framework\v1.0.2914` incluida dentro del directorio de instalación de su versión de Windows. De todas formas, el programa de instalación del SDK lo añade automáticamente al path, por lo que en principio puede llamársele sin problemas desde cualquier directorio.

La forma más básica de llamar al compilador consiste en pasarle como argumentos los nombre de los fuentes a compilar, caso en que intentaría generar en el directorio desde el que se le llame un ejecutable a partir de ellos con el mismo nombre que el primero de los fuentes indicados y extensión `.exe` Por ejemplo, ante una

llamada como:

```
csc FuenteA.cs FuenteB.cs FuenteC.cs
```

El compilador intentará generar un fuente `FuenteA.exe` en el directorio desde el que se lo llamó cuyo código sea el resultante de compilar FuenteA.cs, FuenteB.cs y FuenteC.cs Obviamente, para que ello sea posible el compilador habrá de disponer de permiso de escritura y espacio suficiente en dicho directorio y además alguno de los fuentes indicados tendrá que disponer de un punto de entrada válido.

Este comportamiento por defecto puede variarse especificando en la llamada a csc opciones de compilación adicionales que sigan la sintaxis:

```
<indicadorOpción><opción>
```

El `<indicadorOpción>` puede ser el carácter **/** o el carácter **-**, aunque en adelante sólo haremos uso de **/**. Respecto a `<opción>`, pueden indicarse dos tipos de opciones:

- **Flags:** Son opciones cuya aparición o ausencia tienen un determinado significado para el compilador. Se indican de esta manera:

  ```
  <nombreFlag><activado?>
  ```

  `<activado>` es opcional e indica si se desea activar el significado del flag. Puede ser el carácter **+** para indicar que sí o el carácter **-** para indicar que no, aunque en realidad darle el valor **+** es innecesario porque es lo que se toma por defecto. También hay algunos flags que no admiten ninguno de los dos caracteres, pues se considera que siempre que aparezcan en la llamada al compilador es porque se desea activar su significado y si no apareciesen se consideraría que se desea desactivarlo.

  A continuación se muestran algunos ejemplos de uso de un flag llamado **/optimize** al compilar . No se preocupe por saber ahora para que sirve, sino simplemente fíjese en cómo se usa y note que los dos primeros ejemplos son equivalentes:

  ```
  csc /optimize   Fuente.cs
  csc /optimize+  Fuente.cs
  csc /optimize-  Fuente.cs
  ```

- **Opciones con valores:** A diferencia de los flags, son opciones cuya aparición no es válida por sí misma sino que siempre que se usen han de incluir la especificación de uno o varios valores. La forma en que se especifican es:

  ```
  <nombreFlag>:<valores>
  ```

  Los `<valores>` indicados pueden ser cualesquiera, aunque si se desea especificar varios hay que separarlos entre sí con caracteres de coma (**,**) ó punto y coma (**;**)

  Como es lógico, en principio los `<valores>` indicados no pueden incluir caracteres de espacio ya que éstos se interpretarían como separadores de argumentos en la llamada a csc. Sin embargo, lo que sí se permite es incluirlos si previamente se les encierra entre comillas dobles (**"**)

  Obviamente, como las comillas dobles también tiene un significado especial en los argumentos de csc tampoco será posible incluirlas directamente como carácter en `<valores>`. En este caso, para solventar esto lo que se hace es interpretarlas como caracteres normales si van precedidas de **** y con su significado especial si no.

De nuevo, esto lleva al problema de que el significado de `\` si precede a `"` también puede ser especial, y para solucionarlo lo ahora que se hace es incluirlo duplicado (`\\`) si aparece precediendo a un `"` pero no se desea que tome su significado especial.

Ejemplos equivalentes de cómo compilar dando valores a una opción `/r` son:

```
csc /r:Lib.dll /r:Lib2.dll Fuente.cs
csc /r:Lib1.dll,Lib2.dll   Fuente.cs
csc /r:Lib1.dll;Lib3.dll   Fuente.cs
```

Aunque en los ejemplos mostrados siempre se han incluido las opciones antes que los nombres de los fuentes a compilar, en realidad ello no tiene porqué ser así y se pueden mezclar libremente y en cualquier orden opciones y nombres de fuentes a compilar (salvo excepciones que en su momento se explicarán)

Opciones de compilación

Una vez explicado cómo utilizar el compilador en líneas generales es el momento propicio para pasar a explicar cuáles son en concreto las opciones que admite. Esto se hará desglosándolas en diferentes categorías según su utilidad.

Antes de empezar es preciso comentar que la mayoría de estas opciones disponen de dos nombres diferentes: un nombre largo que permite deducir con facilidad su utilidad y un nombre corto menos claro pero que permite especificarlas más abreviadamente. Cuando se haga referencia por primera vez a cada opción se utilizará su nombre largo y entre paréntesis se indicará su nombre corto justo a continuación. El resto de referencias a cada opción se harán usando indistintamente uno u otro de sus nombres.

Opciones básicas

En este epígrafe se explicarán todas aquellas opciones que suelen usarse con mayor frecuencia a la hora de compilar aplicaciones. Como la mayoría ya se explicaron en detalle en el *Tema 2: Introducción a C#*, dichas opciones aquí simplemente se resumen:

- **/recurse**: Si en vez de indicar el nombre de cada fichero a compilar como se ha dicho se indica como valor de esta opción se consigue que si el compilador no lo encuentra en la ruta indicada lo busque en los subdirectorios de la misma.

  Por ejemplo, la siguiente llamada indica que se desea compilar el fichero fuente.cs ubicado dentro del directorio c:\Mis Documentos o algún subdirectorio suyo:

  ```
  csc /recurse:"Mis Documentos"\fuente.cs
  ```

- **/target** (**/t**): Por defecto al compilar se genera un ejecutable cuya ejecución provoca la apertura de una ventana de consola si al lanzarlo no hubiese ninguna abierta. Esto puede cambiarse dando uno de los valores indicados en la **Tabla 15** a esta opción:

Valor	Tipo de fichero a generar
exe ó ninguno	Ejecutable con ventana de consola (valor por defecto)
winexe	Ejecutable sin ventana de consola. Útil para escribir aplicaciones de ventanas o sin interfaz
library	Librería
module	Módulo de código no perteneciente a ningún ensamblado

Tabla 15: Valores admitidos por la opción /tde csc

Tanto las librerías como los ejecutables son simples colecciones de tipos de datos compilados. La única diferencia entre ellos es que los segundos disponen de un método especial (**Main()**) que sirve de punto de entrada a partir del que puede ejecutarse código usando los mecanismo ofrecidos por el sistema operativo (escribiendo su nombre en la línea de comandos, seleccionándolo gráficamente, etc.)

La diferencia de un módulo con los anteriores tipos de ficheros es que éste no forma parte de ningún ensamblado mientras que los primeros sí. El CLR no puede trabajar con módulos porque estos carecen de manifiesto, pero crearlos permite disponer de código compilado que pueda añadirse a ensamblados que se generen posteriormente y que podrán acceder a sus miembros **internal**.

- **/main**: Si al compilar un ejecutable hubiese más de un punto de entrada válido entre los tipos definidos en los fuentes a compilar se ha de indicar como valor de esta opción cuál es el nombre del tipo que incluye la definición del **Main()** a utilizar, pues si no el compilador no sabría con cúal de todas quedarse.

  Como es lógico, lo que nunca puede hacerse es definir más de un punto de entrada en un mismo tipo de dato, pues entonces ni siquiera a través de la opción **/main** podría resolverse la ambigüedad.

- **/out** (**/o**): Por defecto el resultado de la compilación de un ejecutable es un fichero **.exe** con el nombre del fuente compilado que contenga el punto de entrada, y el de la compilación de un módulo o librería es un fichero con el nombre del primero de los fuentes a compilar indicados y extensión dependiente del tipo de fichero generado (**.netmodule** para módulos y **.dll** para librerías) Si se desea darle otro nombre basta indicarlo como valor de esta opción.

  El valor que se le dé ha de incluir la extensión del fichero a generar, lo que permite compilar ficheros con extensiones diferentes a las de su tipo. Por ejemplo, para crear un módulo `A.exe` a partir de un fuente `A.cs` puede hacerse:

  ```
  csc /out:A.exe /t:module A.cs
  ```

  Obviamente, aunque tenga extensión **.exe** el fichero generado será un módulo y no un ejecutable, por lo que si se intenta ejecutarlo se producirá un error informando de que no es un ejecutable válido. Como puede deducirse, cambiar la extensión de los ficheros generados no suele ser útil y sólo podría venir bien para dificultar aposta la comprensión del funcionamiento de una aplicación o para identificar ensamblados con algún significado o contenido especial.

- **/reference** (**/r**): Por defecto sólo se buscan definiciones de tipos de datos externas a los fuentes a compilar en la librería **mscorlib.dll** que forma parte de la BCL. Si alguno de los fuentes a compilar hace uso de tipos públicos definidos en otros ensamblados hay que indicar como valores de **/r** cuáles son esos ensamblados para que también se busque en ellos.

  En **mscorlib.dll** se encuentran los tipos de uso más frecuentos incluidos en la BCL. En el poco frecuente caso de que haya definido su propia versión de ellos y no desee que se use la de la BCL, puede puede pasar al compilador el flag **/nostdlib** para indicarle que no desea que busque implícitamente en **mscorlib.dll**.

  Puede que termine descubriendo que en realidad tampoco hace falta referenciar a la mayoría de las restantes librerías que forman la BCL. Pues bien, esto no se debe a que también las referencie implícitamente el compilador, sino a que se incluyen en un fichero de respuesta (más adelante se explica lo que son este tipo de ficheros) usado por defecto por el compilador. Si no desea que utilice este fichero puede pasarle el flag **/noconfig**.

Cuando se den valores a **/r** hay que tener en cuenta que por defecto el compilador interpretará cada ruta así indicada de manera relativa respecto al directorio desde el que se le llame. Si no lo encuentra allí lo hará relativamente respecto al directorio donde esté instalado el CLR, que en los sistemas operativos Windows es el subdirectorio **Microsoft.NET\Framework\v1.0.2914** del directorio de instalación de Windows. Y si tampoco lo encuentra allí la interpretará respecto a los directorios indicados por la variable de entorno **LIB** de su sistema operativo.

Esta política de búsqueda puede modificarse incluyendo opciones **/lib** al llamar al compilador cuyos valores le indiquen en qué directorios ha de buscar antes de pasar a buscar en los indicados por la variable de entorno **LIB**.

- **/addmodule**: Funciona de forma parecida a **/r** pero se utiliza cuando lo que usan los fuentes son tipos definidos externamente en módulos en vez de en ensamblados. Incluso a la hora de buscar módulos se sigue la misma política que al buscar ensamblados y se admite el uso de **/lib** para modificarla.

  Se incluyen opciones **/r** y **/addmodule** separadas porque añadir un módulo a una compilación implica decir que se desea que los tipos que incluye formen parte del ensamblado a generar, por lo que los fuentes a compilar podrán acceder a sus miembros **internal**. Sin embargo, cuando se referencia a otros ensamblados con **/r** esto no ocurre y los fuentes compilados no podrán acceder a sus miembros **internal**.

  Es importante señalar que el CLR espera que todos los módulos que se añadan a un ensamblado se distribuyan dentro del mismo directorio que la librería o ejecutable correspondiente al mismo. Si no se haciese así no los podría localizar y en tiempo de ejecución se produciría una **System.TypeLoadException** si se intentase acceder a los tipos definidos en ellos.

Aunque en principio se ha dicho que no importa cómo se intercalen opciones y nombres de fuentes entre los argumentos pasados a **csc**, hay una excepción que consiste en que **/out** y **/r** siempre han de indicarse antes de algún fuente. Esto permite que en una misma llamada al compilador sea posible solicitar la generación de un ensamblado y múltiples módulos de código, pues se considera que cada aparición de las opciones anteriores hace referencia sólo a los fuentes que le siguen. Por ejemplo, dada:

```
csc /t:library /out:LibA.dll A.cs /t:module /out:ModB.netmodule B.cs
```

Esta llamada provocará la compilación de `A.cs` como librería de nombre `LibA.dll` y la de `B.cs` como módulo llamado `ModB.netmodule`.

Sin embargo, al hacer así compilaciones múltiples hay que tener en cuenta que sólo es válido solicitar que el primer grupo de ficheros indicado se compile como ensamblado. Por tanto, sería incorrecto hacer:

```
csc /t:module /out:ModB.netmodule B.cs /t:library /out:LibA.dll A.cs
```

Esta llamada es incorrecta porque indica que se desea que el segundo grupo de ficheros dé lugar a un ensamblado y ello sólo puede hacerse con el primero.

Por otro lado, también hay que tener en cuenta que no es válido que un mismo tipo de dato se defina en varios de los grupos de ficheros indicados. Por ejemplo, si se quisiese compilar `A.cs` como ejecutable y como módulo podría pensarse en hacer:

```
csc A.cs /t:library A.cs
```

Sin embargo, esta llamada no es válida porque los dos grupos de ficheros indicados contienen el mismo fichero y por tanto definiciones comunes de tipos de datos. La única solución posible sería hacer dos llamadas por separado al compilador como:

```
csc A.cs
csc /t:library A.cs
```

Manipulación de recursos

Los **ficheros de recursos** son archivos que no contienen código sino sólo datos tales como como cadenas de textos, imágenes, vídeos o sonidos. Su utilidad es facilitar el desacople entre las aplicaciones y los datos concretos que usen, de modo que sea fácil reutilizarlos en múltiples aplicaciones, modificarlos sin tener que recompilar los fuentes y desarrollar diferentes versiones de cada aplicación en las que sólo varíen dichos datos.

Estos ficheros son especialmente útiles al hora de internacionalizar aplicaciones, pues si se dejan todos los datos que se utilicen en ficheros de recursos independiente del código, a la hora de crear nuevas versiones en otros idiomas sólo será necesario cambiar los ficheros de recursos y habrá que tocar para nada el código.

El objetivo de este tema no es explicar cómo crear y acceder a ficheros de recursos, sino explicar el significado de las opciones de compilación relacionadas con ellos. Si desea aprender más sobre recursos puede comenzar buscando en el apartado **Visual Studio.NET -> .NET Framework -> .NET Framework Tutorials -> Resources and Localization Using the .NET Framework SDK** de la ayuda del SDK.

Lo que sí es importante es señalar que aunque en la plataforma .NET pueden crearse ficheros de recursos tanto en formato `.txt` como `.resx`, el compilador de C# sólo los admite si están compilados en formato .resources. Para ello, en el SDK se incluye una utilidad llamad `resgen.exe` que permite compilar en dicho formato ficheros de recursos escritos en cualquiera de los formatos anteriores con sólo pasárselos como argumentos. Por ejemplo, si se le llama así:

```
resgen misrecursos.resx
```

Suponiendo que el contenido de `misrecursos.resx` sea el de un fichero `.resx` válido, tras esta llamada se habrá generado en el directorio desde el que se le llamó un fichero misrecursos.resources con el contenido de `misrecursos.resx`.

Para añadir este fichero al ensamblado resultante de una compilación se puede utilizar la opción `/linkresource` (`/linkres`) Así por ejemplo, para crear un ensamblado `fuente1.dll` formado por el código resultante de compilar `fuente1.cs` y los recursos de `misrecursos.resources` podría compilarse con:

```
csc /t:library fuente1.cs /linkres:misrecursos.resources
```

De este modo el fichero de recursos formará parte del ensamblado generado pero permanecerá en un fichero separado de `fuente1.dll`. Si se desease incrustarlo en él habría que haber compilado con la opción `/resource` (`/res`) en vez de `/linkres` tal y como se muestra a continuación:

```
csc /t:library fuente1.cs /res:misrecursos.resources
```

Como un tipo especial de recurso que comúmente suele incrustarase en los ejecutables de los programas es el icono (fichero gráfico en formato `.ico`) con el que desde las interfaces gráficas de los sistemas operativos se les representará, `csc` ofrece una opción específica llamada `/win32icon` en cuyo valor puede indicársele el icono a incrustar:

```
csc programa.cs /win32icon:programa.ico
```

En realidad hay que recordar el uso de ficheros de recursos no es un aspecto introducido en la plataforma .NET sino disponible desde hace tiempo en la plataforma Windows en forma de ficheros `.res`. Por compatibilidad con este antiguo formato de recursos, `csc` incorpora una opción `/win32res` que permite

incrustarlos de igual forma a como **/res** incrusta los novedosos ficheros **.resources**.

En cualquier caso, hay que señalar que siempre que se añada un fichero de recursos a un ensamblado la visibilidad que se considerará para los recursos que incluya es **public**.

Configuración de mensajes de avisos y errores

Cada vez que el compilador detecta algún error en uno de los fuentes a compilar genera un mensaje informando de ello en el que indica en qué fichero de código fuente y en qué posición exacta del mismo (línea y columna) lo ha detectado. Por ejemplo, si en la columna 3 de la línea 7 de un fuente llamado `ej.cs` se llama a un método con nombre completo `A.K()` inexistente, se mostrará un mensaje como:

```
ej.cs(7,3): error CS0117: 'A' does not contain a definition for 'K'
```

Nótese que del fichero sólo se da su nombre y ello podría no identificarlo unívocamente si se compilaron a la vez varios con el mismo nombre pero pertencientes a directorios diferentes. Para solucionar esto puede usarse la opción **/fullpaths**, con lo que de los mensajes de error incluirían siempre la ruta completa de los ficheros defectuosos. Por ejemplo, si el fichero del ejemplo anterior se encontraba en `C:\Ejemplo`, al compilarlo con esta opción se mostraría el mensaje de error así:

```
C:\Ejemplo\ej.cs(7,3): error CS0117: 'A' does not contain a definition for 'K'
```

Hay veces que el compilador detecta que se han escrito en el fuente ciertas secciones de tal manera que sin ser erróneas son cuanto menos sospechosas (ya sea por ser absurdas, por prestarse a confusión, etc), y en esos casos lo que hace es emitir mensajes de aviso. Por ejemplo, si en la definición del tipo A del fuente prueba.cs se hubiese incluido:

```
static void Main(int x)
{}
```

En principio es una definición de método perfectamente válida. Sin embargo, como se parece mucho a una definición de punto de entrada pero no es válida como tal, el compilador generará el mensaje de aviso que sigue para informar de ello al usuario por si acaso éste lo que quería hacer era definir un punto de entrada y se equivocó:

```
prueba.cs(7,14): warning CS0028: 'A.Main(int)' has the wrong signature to be an
entry point
```

Como se ve, la estructura de los mensajes de aviso es muy similar a la de los mensajes de error y sólo se diferencia de ésta en que incluye warning en vez de error tras el indicador de posición en el fuente. Incluso como a estos, la opción **/fullpaths** también les afecta y provoca que se muestren las rutas de los fuentes al completo.

Una diferencia importante entre avisos y errores es que la aparición de mensajes de los segundos durante la compilación aborta la generación del binario, mientras que la aparición de los primeros no (aunque en ambos casos nunca se aborta la compilación sino que tras mostrarlos se sigue analizando los fuentes por si pudiesen detectarse más errores y avisos) Ahora bien, también puede forzarse a que ello ocurra con los de aviso pasando al compilador el flag **/warnaserror**, con lo que se conseguiría que todo mensaje de aviso se muestrase como error. Ello puede resultar útil porque fuerza a escribir los fuentes de la manera más fiable e inteligentemente posible.

En el laod opuesto, puede que haya ciertos tipos de mensajes de aviso de los que no se desea siquiera que se informe en tanto que la información que aportan ya se conoce y se sabe que no afectará negativamente al programa. En esos casos puede usarse la opción **/nowarn** indicando como valores suyos los códigos asociados a los mensaje de aviso que no se desea que se reporten. El código asociado a cada tipo de mensaje de aviso se es la palabra de la forma **CS**`<código>` que se muestra tras warning en el mensaje de

aviso. Así, para compilar el `prueba.cs` del ejemplo anterior sin que se genere el mensaje de aviso arriba mostrado puede hacerse:

```
csc prueba.cs /nowarn:0028
```

En realidad los ceros incluidos a la izquierda del código del aviso en los mensajes de aviso son opcionales, por lo que la compilación anterior es equivalente a:

```
csc prueba.cs /nowarn:28
```

Si desea obtener la lista completa de todos los tipos de mensaje de aviso y error con sus respectivos códigos puede consultar dentro de la documentación del .NET Framework SDK en **Visual Studio.NET -> Visual Basic and Visual C# -> Visual C# Language -> C# Compiler Options -> Compiler Errors CS0001 to CS9999**

Si en lugar de desactivar ciertos tipos de avisos uno por uno desea desactivarlos por grupos según su severidad, entonces puede hacerlo a través de la opción **`/warn`**. Esta opción toma como valor un número comprendido entre 0 y 4 que indica cuál es el nivel de avisos con el que se desea trabajar. Por defecto éste vale 4, lo que significa que se mostrarán todos los avisos, pero puede dársele cualquiera de los de la **Tabla 16**:

Nivel de aviso	Avisos mostrados
0	Ninguno
1	Sólo los más graves
2	Los más graves y algunos menos graves como por ejemplo los relativos a ocultaciones de miembros
3	Los de nivel 2 más algunos poco graves como los relativos al uso de expresiones absurdas que siempre produzcan el mismo resultado
4	Todos

Tabla 16: Niveles de mensajes de aviso

Si está interesado en conocer en concreto el nivel de algún tipo de aviso puede remitirse a la descripción sobre el mismo incluida en la documentación del SDK antes comentada

Ficheros de respuesta

La línea de comandos no es la única forma de pasar información al compilador (tanto ficheros a compilar como opciones de compilación), sino que también es posible almacenar información de este tipo en un fichero y pasársele al compilador como argumento solamente dicho fichero y no toda la información en él contenida. De este modo se facilitaría la labor de pasar como parámetros las opciones de uso más frecuente ya que bastaría sólo indicar cuál es el nombre de un fichero que las especifica.

A este ficheros se les llama **ficheros de respuesta**, ya que al pasárselos al compilador su contenido puede verse como la respuesta a cuáles son los argumentos a usar durante la compilación. La extensión de estos ficheros suele ser **`.rsp`**, y aunque nada obliga a dársela es conveniente hacerlo como ocurre con todo convenio.

Al compilar, por defecto el compilador siempre lee un fichero de respuesta llamado **`csc.rsp`** ubicado en el directorio del CLR, por lo que para entender cuál es la sintaxis a seguir para escribir estos ficheros nada mejor que ver cuál es su contenido y así de paso saber cuáles son las opciones que por defecto se añadiran a toda compilación:

```
# This file contains command-line options that the C#
```

```
# command line compiler (CSC) will process as part
# of every compilation, unless the "/noconfig" option
# is specified.
# Reference the common Framework libraries
/r:Accessibility.dll
/r:Microsoft.Vsa.dll
/r:System.Configuration.Install.dll
/r:System.Data.dll
/r:System.Design.dll
/r:System.DirectoryServices.dll
/r:System.dll
/r:System.Drawing.Design.dll
/r:System.Drawing.dll
/r:System.EnterpriseServices.dll
/r:System.Management.dll
/r:System.Messaging.dll
/r:System.Runtime.Remoting.dll
/r:System.Runtime.Serialization.Formatters.Soap.dll
/r:System.Security.dll
/r:System.ServiceProcess.dll
/r:System.Web.dll
/r:System.Web.RegularExpressions.dll
/r:System.Web.Services.dll
/r:System.Windows.Forms.Dll
/r:System.XML.dll
```

Del contenido de este fichero es fácil deducir que la estructura de los ficheros de respuesta es sencilla: cada opción se incluye en una línea aparte y pueden intercalarse entre ellas comentarios de una línea que comiencen con `#`. Además, como puede verse este fichero de respuesta usado por defecto añade referencias a las librerías de la BCL de uso más común, lo que evita tener que incluirlas constantemente al compilar.

Tras tomar las opciones de este fichero, el compilador mira si en el directorio desde el que se le llama hay otro **`csc.rsp`** y si es así toma sus opciones. Si por alguna razón no nos interesase que se tomasen las opciones de dichos ficheros (por ejemplo, para usar nuevas versiones de tipos incluidos en las librerías que referencian) bastaría pasar el flag **`/noconfig`** al compilar para desactivar esta búsqueda por defecto en ellos, aunque hay que señalar que este flag no admite los sufijos **`+`** y **`-`** admitidos por el resto de flags.

En realidad, la estructura del fichero de respuesta **`csc.rsp`** no es la única posible, pues además de opciones también es válido incluir nombres de fuentes a compilar e incluso puede mezclarse múltiples opciones y nombres de fuentes en cada línea del fichero.

Ahora bien, al escribir ficheros de respuesta hay que tener cuidado con dos cosas: no es posible cortar las opciones o nombres de fichero con retornos de carro que provoquen que ocupen varias líneas; y las opciones son pasadas al compilador en el mismo orden en que aparezcan en el fuente, por lo que hay que tener cuidado con cómo se coloquen las opciones **`/out`** y **`/t`** por lo ya comentado sobre la importacia de su colocación.

Una vez escrito un fichero de respuesta, para indicar al compilador que ha de usarlo basta pasárselo como un nombre de fuente más pero precediendo su nombre del sufijo `@`. Por ejemplo, para compilar A.cs usando las opciones almacenadas en `opc.rsp` habría que llamar al compilador con:

```
csc @opc.rsp A.rsp
```

También sería posible indicar múltiples ficheros de respuesta, caso en que se tomarían las opciones de cada uno en el mismo orden en que apareciesen en la llamada a **`csc`**. Por ejemplo, para compilar `A.rsp` tomando las opciones de `opc1.rsp` y luego las de `opc2.rsp` podría llamarse al compilador con:

```
csc @opc1.rsp @opc2.rsp A.rsp
```

Puede ocurrir que las opciones indicadas en un fichero de respuesta contradigan a opciones indicadas en otro fichero de respuesta indicado a continuación o a opciones dadas al compilador en la línea de comandos. Para resolver estas ambigüedades el compilador siempre va procesando los argumentos que se le pasen de izquierda a derecha y se queda con la última especificación dada a cada opción. Así, en el ejemplo anterior las opciones del **csc.rsp** del directorio desde el que se le llamó -si existiese- tendría preferencia sobre las del **csc.rsp** del directorio del CLR, las de `opc2.rsp` tendrían preferencia sobre las de éste, y las de `opc1.rsp` sobre las de `opc2.rsp`.

También pueden incluirse en los ficheros de respuesta opciones **@** que incluyan a otros ficheros de respuesta, con lo que se tomaría sus opciones antes de continuar tomando las siguientes del fichero que lo incluyó, aunque obviamente nunca se admitirá que un fichero incluido sea el mismo que el que lo incluye o que alguno que incluya a éste, pues entonces se formarían ciclos y nunca acabaría la búsqueda de opciones.

Opciones de depuración

Sin duda la opción de depuración más importante es el flag **/debug**, cuya inclusión indica al compilador que ha de generar un fichero **.pdb** con información sobre la relación entre el fichero binario generado y las líneas de los fuentes a partir de los que se generó. Esta información es muy útil para depurar aplicaciones, pues permite mostrar la instrucción de código fuente que produjo las excepciones en lugar de mostrar las instrucciones de código nativo en que fue traducida.

Para entender mejor la utilidad de este fichero **.pdb** puede escribir el programa:

```
class A
{
  public static void Main()
  {throw new System.Exception();}
}
```

Si lo compila con:

```
csc A.cs
```

Al ejecutarlo se producirá una excepción y surgirá una ventana de selección de depurador. Si pulsa **No** en ella verá en la consola un mensaje como el siguiente:

```
Unhandled Exception: System.Exception: Exception of type System.Exception was
thrown. at A.Main()
```

Sin embargo, si lo compila con:

```
csc A.cs /debug
```

Al ejecutarlo se obtendrá un mensaje mucho más detallado en el que se indicará cuál es la línea exácta del código fuente durante cuya ejecución se produjo la excepción:

```
Unhandled Exception: System.Exception: Exception of type System.Exception was
thrown at A.Main() in E:\c#\Ej\A.cs:line 5
```

Como es fácil deducir, a partir de esta información es fácil crear herramientas de depuración -como el depurador de Visual Studio.NET o el CLR Debugger del SDK- que muestren la línea exácta del código fuente donde se produjo la excepción lanzada; y obviamente estos datos también pueden tener muchos otros usos,

como permitir ejecutar paso a paso los programas mostrando en cada momento cuál es la línea del fuente que se ejecutará a continuación y cosas similares.

También puede usarse **/debug** como opción con argumentos en vez de cómo flag, lo que permite generar una versión recortada de la información de depuración. Si de esta forma se le da el valor **full** funcionará exáctamente igual que al activarla como flag, pero si se le da el valor **pdbonly** entonces la información de depuración generada sólo estará disponible para los depuradores desde los que se haya lanzado la aplicación pero no para los que se le hayan adjuntado dinámicamente una vez lanzada.

Por último, respecto a la depuración de aplicaciones conviene señalar que por defecto el compilador siempre intenta generar el código más compacto y eficiente posible, lo que provoca que compile más lentamente. Sin embargo, como cuando se está depurando suelen realizarse mmuchas recompilaciones de los fuentes puede que en esos casos interese desactivar dichas optimizaciones y así así conseguir recompilar más rápido. Ello puede conseguirse llamando al compilador con **/optimize-** (**/o-**)

Compilación incremental

La **compilación incremental** consiste en sólo recompilar en cada compilación que se haga de un proyecto aquellos métodos cuya definición haya cambiado respecto a la última compilación realizada, con lo que el proyecto podría compilarse más rápido que haciendo una compilación completa normal.

Para que esto sea posible hacerlo hay que llamar al compilador con el flag **/incremental** (**/incr**), lo que provocará la generación de un fichero adicional con el mismo nombre que el binario generado más una extensión **.incr**. Por ejemplo, dado:

```
csc /out:fuente.exe /incremental Fuente.cs
```

Se generará un ejecutable fuente.exe y un fichero adicional `fuente.exe.incr`. Aunque pueda parecer redundante incluir en el ejemplo la opción **/out** al llamar al compilador, es necesaria porque al menos en la versión del compilador incluida en la beta 2 del es obligatorio especificarla siempre que se utilice **/incr**.

El fichero **.incr** generado incluye información sobre la compilación que permitirá que posteriores compilaciones que se realicen con **/incr** activado puedan hacerse de manera incremental. Obviamente, si este fichero se elimina será reconstruido en la siguiente compilación que se haga con **/incr**, pero dicha compilación no se realizará de manera completa por no disponerse del fichero **.incr** durante ella.

Sin embargo, el hecho de que esté disponible un fichero **.incr** al compilar un proyecto no implica que se use, pues el compilador puede ignorarlo y realizar una compilación completa si detecta que han cambiado las opciones de compilación especificadas o si detecta que los fuentes han cambiado tanto que es al menos igual de eficiente hacerla así que de manera incremental.

En realidad no es bueno hacer siempre las compilaciones incrementalmente sino que sólo es útil hacerlo en proyectos formados por mútilples fuentes de pequeño tamaño, minetras que en proyectos con pocos y grandes ficheros se gana poco o nada en tiempo de compilación. Además, los ejecutables generados incrementalmente pueden ocupar más que los generados por compilación completa, por lo sólo es recomendable compilar incrementalmente las versiones de prueba de los proyectos pero no las definitivas.

Opciones relativas al lenguaje

A lo largo de los anteriores temas se ha ido diseminando diversas opciones de compilación relacionadas de manera más o menos directa con el lenguaje C#. En este punto haremos recapitulación de todas ellas mismas y las resumiremos:

- **/define** (**/d**): En el *Tema 3: El preprocesador* ya se introdujo esta opción cuyos valores recordemos que se utilizan para introducir definiciones de símbolos de preprocesado al

principio de todos los fuentes a compilar.

Por ejemplo, si se desea compilar los fuentes `A.cs` y `B.cs` como si al principio de ellos se hubiese incluido las directivas de preprocesado `#define PRUEBA` y `#define VERSION1` podría llamarse al compilador con:

```
csc /d:PRUEBA;VERSION1 A.cs B.cs
```

- **/checked**: En los temas 4 y 16 se explicó que todo desbordamiento que ocurra en operaciones aritméticas entre variables enteras es tratado por defecto truncando el resultado. Pues bien, la utilidad de activar esta opción es precisamente forzar a que se incluyan en el código generado las comprobaciones necesarias para que en caso de desbordamiento se lance en su lugar una **`System.OverflowException`**.

  Obviamente el código compilado con **`/checked`** se ejecutará más lento que el que lo haga sin ella ya que incluirá comprobaciones de desbordamiento adicionales. Sin embago, a cambio con ello se consigue detectar con facilidad errores derivados de desbordamientos que de otra manera podrían pasar inadvertidos.

- **/unsafe**: En el *Tema 18: Código inseguro* ya se explicó que la única utilidad de esta opción es servir al compilador de mecanismo de seguridad gracias al que pueda asegurarse de que el usario sabe lo que hace al compilar código con punteros.
- **/doc**: Esta opción ya se introdujo en el *Tema 19: Documentación XML*, donde se explicó que se usa para indicar al compilador que es desea generar un fichero XML con el contenido de los comentarios de documentación incluidos en los fuentes a compilar. El nombre de ese fichero será el que se dé como valor a esta opción.

  Al usar esta opción hay que tener en cuenta una cosa, y es que para optimizar el tiempo que se tarda en realizar compilaciones incrementales, durante ellas esta opción es ignorada. Por tanto, no tiene mucho sentido combinar **`/doc`** y **`/incr`**.

Otras opciones

Aparte de las opciones comentadas, **`csc`** admite unas cuantas más aún no descritas ya sea porque su uso es muy poco frecuente o porque no encajan correctamente en ninguno de los subepígrafes tratados. Todas estas opciones se recogen finalmente aquí:

- **/filealign**: Los valores dados a esta opción indican el tamaño de las secciones en que se dividirán los ficheros binarios resultantes de la compilación. Puede tomar los valores 512, 1024, 2048, 4096, 8192 ó 16384, y cada sección en los binarios comenzará en un posición que sea múltiplo del valor dado a esta opción.

  Por defecto el valor que se le dé puede variar dependiendo de la implementación que se haga del CLR, aunque darle un valor a medida puede ser útil en el diseño de aplicaciones para dispositivos empotrados con escasa capacidad de almacenamiento ya que puede reducir el tamaño de los ficheros generados.

- **/bugreport**: Dado que es muy difícil diseñar un compilador 100% libre de errores, Microsoft proporciona a través de esta opción un mecanismo que facilita a los usuarios el envío de información sobre los errores que descubran en el mismo y facilita a Microsoft la labor de interpretarla para solucionarlos lo antes posible.

  El valor que se dé a esta opción es el nombre de con el que se desea que se genere el fichero con la información relativa al error descubierto durante la compilación. En dicho fichero csc insertará automáticamente la siguiente información:

- Opciones de compilación utilizadas.
- Versión del compilador, CLR y sistema operativo usado.
- Copia de todos los códigos fuentes compilados. Como es lógico, para facilitar la correción a Microsoft se recomienda enviar el programa más compacto posible en el que se produzca el error descubierto.
- Mensajes de salida mostrados durante la compilación.

Aparte de toda esta información insertada automáticamente por el compilador, durante la generación del fichero de error también se pedirá al usuario que indique una pequeña descripción sobre el error detectado y cómo cree que podría solucionarse. Dicha información también será añadida de manera automática al fichero de error que se cree.

Un ejemplo cómo generar información relativa a un error verídico que se produce al compilar un programa `error.cs` con la Beta 1 del .NET SDK Framework es:

```
csc error.cs /bugreport:ErrorUsing.cs
```

Tras contestar a las preguntas que el compilador hará al usuario sobre el error encontrado, el contenido del fichero generado es el siguiente:

```
### C# Compiler Defect Report, created 07/12/00 20:14:36
### Compiler version: 7.00.9030
### Common Language Runtime version: 1.00.2914.16
### Operating System: Windows NT 5.0.2195   Service Pack 2
### User Name: Administrador
### Compiler command line
csc.exe error.cs /bugreport:ErrorUsing.cs
### Source file: 'e:\c#\ej\error.cs'
using System;

public class R1:IDisposable
{
  public static void Main()
  {
    using (R1 r1 = new R1())
    {
    }
  }

  public void Dispose()
  {}
}
### Compiler output
error.cs(7,3): error CS1513: } expected
error.cs(7,26): error CS1002: ; expected
error.cs(12,9): error CS1518: Expected class, delegate, enum,
interface, or struct
error.cs(14,1): error CS1022: Type or namespace definition, or
end-of-file expected
### User description
No detecta la instruccion using

### User suggested correct behavior
Posiblemente no haya sido implementada en esta version del compilador
```

Nótese que aunque el error detectado en el ejemplo es verídico, en versiones del compilador posteriores a la Beta 1 no se produce porque ya fue corregido.

- **/baseaddress**: Esta opción sólo tiene sentido cuando se solicita la generación de una librería e indica cuál es la dirección de memoria en que se prefiere que ésta se cargue cuando sea enlazada dinámicamente. Nótese que se ha dicho librería, pues si el fichero generado es de cualquier otro tipo será ignorada.

  El valor que se dé a esta opción puede indicarse tanto en hexadecimal como en octal o decimal siguiendo las reglas usadas en C# para la escritura de literales enteros. Sin embargo, hay que tener en cuenta que los bits menos significativos de esta dirección pueden ser redondeados. Por ejemplo, si escribimos:

  ```
  csc fichero.cs /baseaddress:0x11110001
  ```

  El compilador tratará esta llamada tal y como si se le hubiese pasado:

  ```
  csc fichero.cs /baseaddress:0x11110000
  ```

  Si no se da valor a esta opción, las librerías se instalarán en el área de memoria que se estime conveniente en cada implementación del CLR.

- **/codepage**: Por defecto el compilador acepta fuentes escritos en Unicode, UTF-8 o usando la página de códigos por defecto del sistema operativo. Si se desea compilar fuentes escritos en otras páginas de código hay que indicar como valor de esta opción el identificador de ella.

  Un uso típico de esta opción es permitir compilar fuentes escritos en español con un editor de textos de MS-DOS (como **edit.com**), caso en que hay que darle el valor 437 para que acepte los caracteres especiales tales como acentos o eñes.

  /utf8output: Su inclusión indica que el compilador ha de mostrar los mensajes usando el juego de caracteres UTF-8, lo que es útil cuando se utilizan ciertos sistemas operativos internacionales en los que por defecto no se muestren correctamente dichos mensajes por la ventana de consola.

  Para poder leerla en esos casos se recomienda usar este flag al compilar y redirigir la salida a un fichero como muestra el siguiente ejemplo donde se compila A.cs redirigiendo los mensajes de compilación a salida.txt y mostrándolos en UTF-8:

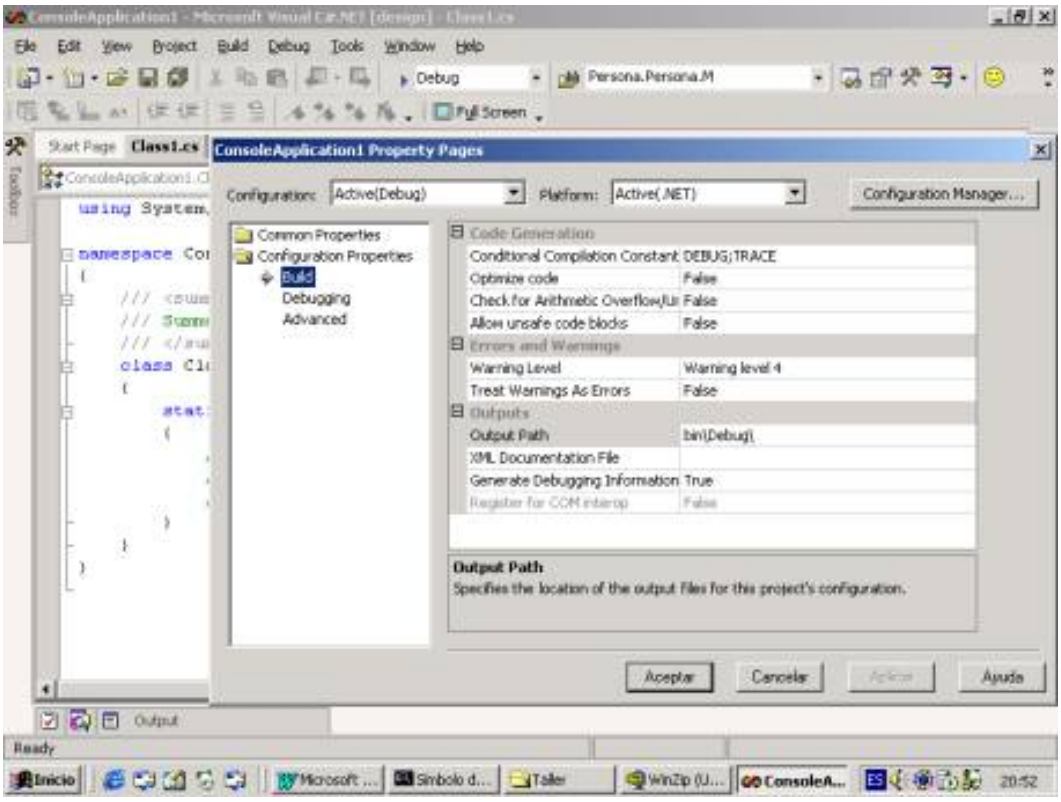
  ```
  csc A.cs /utf8output  > salida.txt
  ```

- **/help** (**/?**): Muestra un mensaje de ayuda resumiendo cuáles son las opciones admitidas por el compilador y para qué sirven. Toda opción o fichero a compilar especificado junto opción son totalmente ignorados.
- **/nologo**: Indica que no se desea que al ejecutar el compilador se genere el mensaje que incluye información sobre la versión del compilador y el copyright de Microsoft sobre el mismo que por defecto se muestra.

  Suele usarse cuando la compilación se solicita desde una aplicación o fichero de procesamiento por lotes, pues oculta la ejecución del compilador al usuario y ello puede venir bien para evitar que éste conozca cómo funciona la aplicación o para conseguir un funcionamiento más elegante y transparente de la misma.

Acceso al compilador desde Visual Studio.NET

Como se explicó en su momento en el *Tema 2: Introducción a C#*, a las opciones de compilación de un proyecto se accede desde VS.NET a través de las páginas de propiedades del mismo, las cuales tiene el aspecto mostrado en la **Ilustración 9** y se obtienen seleccionando el proyecto en el **Solution Explorer** y pulsando sobre **View -> Property Pages** en el menú principal de Visual Studio.



Para la mayoría de opciones admitidas por csc.exe se incluye en estas páginas controles tales como cajas de texto y listas desplegables que permiten configurarlas de una manera visual, cómoda e intuitiva. En la **Tabla 17** se resume en orden alfabético cuál es el control que en concreto se asocia en estas páginas a cada opción:

Opción	Control visual
/baseaddress	Configuration Properties -> Advanced -> Base Address
/checked	Configuration Properties -> Build -> Check for Arithmetic Overflow/Underflow
/debug	Configuration Properties -> Build -> Generate Debugging Information
/define	Configuration Properties -> Build -> Conditional Compilation Constants
/doc	Configuration Properties -> Build -> XML Documentation File
/filealign	Configuration Properties -> Build -> File Alignment
/incremental	Configuration Properties -> Advanced -> Incremental Build
/main	Common Properties -> General -> Startup Object
/optimize	Configuration Properties -> Build -> Optimize code
/out	Common Properties -> General -> Assembly Name
/target	Common Properties -> General -> Output Type
/unsafe	Configuration Properties -> Build -> Allow unsafe code blocks
/warn	Configuration Properties -> Build -> Warning Level blocks
/warnaserror	Configuration Properties -> Build -> Treat Warnings As Errors
/win32icon	Common Properties -> General -> Application Icon

Tabla 17: Controles asociados a opciones de compilación

Como puede observar, desde VS.NET no es posible acceder a muchas de las opciones del compilador en línea de comandos. En los casos de **/codepage**, **/fullpaths**, **/lib**, **/help**, **/nologo**, **/recurse** y **/utf8output** esto es lógico ya que son opciones que pierden su sentido desde dentro en una interfaz gráfica. Hay otros casos en que ello se debe a que se ofrecen desde el menú principal de VS.NET otros mecanismos alternativos para especificarlas, como son los indicados en la **Tabla 18**:

Opción	Mecanismo de acceso
/bugreport	Help -> Customer Feedback
/resource	Seleccionar el recurso en Project -> Add Existing Item
/reference	Seleccionar la referencia en Project -> Add Reference

Tabla 18: Acceso a opciones fuera de las páginas de propiedades

Finalmente, queda un grupo de opciones que no disponibles simplemente porque la implementación de VS.NET (al menos en la Beta 2) no las contempla, y son **@**, **/linkresource**, **/nostdlib**, **/noconfig**, **/nowarn** y **/win32res**. En este sentido, mención aparte merece el valor **module** de **/t**, que tampoco puede usarse en tanto que VS.NET no soporta el trabajo con módulos.

Principio Página

El lenguaje de programación C#

En esta página:

Documentación de referencia

Bibliografía

En el momento de escribir estas líneas no hay disponible ninguna otra bibliografía de calidad sobre C# escrita en castellano, sino que toda la disponible lo está en inglés.

Entre las fuentes de información sobre C# en inglés cabe destacar el documento "C# Language Specification" escrito por Anders Hejlsberg, Scott Wiltamuth y Peter Golde que Microsoft ha remitido al ECMA para la estandarización del lenguaje. Este documento incluye la especificación completa del mismo y Microsoft permite descargarlo gratuitamente desde la dirección http://www.msdn.microsoft.com/net/ecma.

Sin embargo, si lo que busca son libros que expliquen el lenguaje con algo menos de rigurosidad pero de manera mucho más fácil de entender y aplicar, entonces puede consutlar la siguiente bibliografía:

- "A programmer's introduction to C#" escrito por Eric Gunnerson y publicado por Apress en 2000.
- C# and the .NET Framework", escrito por Andrew Troelsen y publicado por Apress en 2001
- "C# Essentials", escrito por Beb Albahari, Peter Drayton y Brand Merril y publicado por O'Reilly en 2000.
- "C# Programming with the Public Beta", escrito por Burton Harvey, Simon

Robinson, Julian Templeman y Karli Watson y publicado por Wrox Press en 2000.
- "Inside C#", escrito por Tom Archer y publicado por Microsoft en 2000
- "Presenting C#", escrito por Christoph Wille y publicado por Sams Publishing en 2000.
- "Professional C#", escrito por Simon Robinson, Burt Harvey, Craig McQueen, Christian Nagel, Morgan Skinner, Jay Glynn, Karli Watson, Ollie Cornes, Jerod Moemeka y publicado por Wrox Press en 2001.
- "Programming C#", escrito por Jesse Liberty y publicado por O'Reilly en 2001

De entre todos estos libros quizás el principalmente recomendable tras leer esta obra pueda ser "Professional C#", pues es el más moderno y abarca numerosos conceptos sobre la aplicación de C# para acceder a la BCL.

Por otra parte, en relación con los libros publicados en 2000 hay que señalar que fueron publicados para el compilador de C# incluido en la Beta 1 del SDK, por lo que no tratan los aspectos nuevos introducidos a partir de la Beta 2 y puede que contengan código de ejemplo que haya quedado obsoleto y actualemente no funcione.

Información en Internet sobre C#

Aunque la bibliografía publicada sobre C# al escribir estas líneas es relativamente escasa, no ocurre lo mismo con la cantidad de material online disponible, que cada vez va inundando más la Red. En esta sección se recogen los principales portales, grupos de noticias y listas de distribución dedicados al lenguaje. Seguramente cuando lea estas líneas habrán surgido muchos más, puede usar la lista ofrecida para encontrar enlaces a los nuevos a partir de los que aquí se recogen.

Portales

Si busca un portal sobre C# escrito en castellano el único que le puedo recomendar es "El Rincón en Español de C#" (http://tdg.lsi.us.es/~csharp), que es el primero dedicado a este lenguaje escrito en castellano. Ha sido desarrollado por profesores de la Facultad de Informática y Estadística de Sevilla, y entre los servicios que ofrece cabe destacar sus aplicaciones de ejemplo, FAQ, seminario "on-line" y lista de distribución de correo.

Si no le importa que el portal esté en inglés, entonces es de obligada visita el ".NET Developers Center" (http://www.msdn.microsoft.com/net) de Microsoft, ya que al ser los creadores del C# y la plataforma .NET su información sobre los mismos suele ser la más amplia, fiable y actualizada. Entre los servicios que ofrece cabe destacar la posibilidad de descargar gratuitamente el .NET Framework SDK y Visual Studio .NET, sus numerosos vídeos y artículos técnicos, y sus ejemplos de desarrollo de software profesional de calidad usando estas tecnologías.

Aparte del portal de Microsoft, otros portales dedicados a C# que pueblan la Red son:

- "C# Corner" (http://www.c-sharpcorner.com)
- "C# Help"(http://www.csharphelp.com)
- "C# Station" (http://www.csharp-station.com)
- "Codehound C#" (http://www.codehound.com/csharp)
- "csharpindex.com" (http://www.csharpindex.com)
- "Developersdex" (http://www.developersdex.com/csharp)
- ".NET Wire" (http://www.dotnetwire.com)

Grupos de noticias y listas de correo

Microsoft ha puesta a disposición de los desarrolladores numerosos grupos de noticias dedicados a resolver dudas sobre C#, .NET y Visual Studio.NET. Los ofrecidos en castellano son:

- microsoft.public.vsnet
- microsoft.public.es.csharp

Respecto a los proporcionados en inglés, señalar que aunque algunos de ellos se recogen en la opción **Online Community** de la página de inicio de VS.NET, la lista completa día a día crece cada vez más y en el momento de escribir estas líneas era:

- microsoft.public.dotnet.academic
- microsoft.public.dotnet.distributed_apps
- microsoft.public.dotnet.faqs
- microsoft.public.dotnet.general
- microsoft.public.dotnet.framework
- microsoft.public.dotnet.framework.adonet
- microsoft.public.dotnet.framework.aspnet
- microsoft.public.dotnet.framework.aspnet.mobile
- microsoft.public.dotnet.framework.aspnet.webservices
- microsoft.public.dotnet.framework.clr
- microsoft.public.dotnet.framework.component_services
- microsoft.public.dotnet.framework.documentation
- microsoft.public.dotnet.framework.interop
- microsoft.public.dotnet.framework.odbcnet
- microsoft.public.dotnet.framework.perfomance
- microsoft.public.dotnet.framework.remoting
- microsoft.public.dotnet.framework.sdk
- microsoft.public.dotnet.framework.setup
- microsoft.public.dotnet.framework.windowsforms
- microsoft.public.dotnet.languages.csharp
- microsoft.public.dotnet.languages.jscript
- microsoft.public.dotnet.languages.vb
- microsoft.public.dotnet.languages.vb.upgrade
- microsoft.public.dotnet.languages.vc
- microsoft.public.dotnet.languages.vc.libraries
- microsoft.public.dotnet.samples
- microsoft.public.dotnet.scripting
- microsoft.public.dotnet.vsa
- microsoft.public.dotnet.xml
- microsoft.public.vsnet.debuggin
- microsoft.public.vsnet.documentation
- microsoft.public.vsnet.enterprise.tools
- microsoft.public.vsnet.faqs
- microsoft.public.vsnet.general
- microsoft.public.vsnet.ide
- microsoft.public.vsnet.samples
- microsoft.public.vsnet.servicepacks
- microsoft.public.vsnet.setup

- microsoft.public.vsnet.visual_studio_modeler
- microsoft.public.vsnet.vsa
- microsoft.public.vsnet.vsip
- microsoft.public.vsnet.vss

En realidad, de entre todos estos grupos de noticias sólo están exclusivamente dedicados a C# **microsoft.public.es** y **csharp microsoft.public.dotnet.languages.csharp**, pero a medida que vaya adentrandose en el lenguaje descubrirá que los dedicados a los diferentes aspectos de .NET y VS.NET también le resultarán de incalculable utililidad.

En lo referente a listas de correo, si busca una lista en castellano la más recomendable es la del "Rincón en Español de C#" (http://tdg.lsi.us.es/csharp) antes mencionada; mientras que si no le importa que estén en inglés, entonces puede consultar las ofrecidas por "DevelopMentor" (http://www.discuss.develop.com)

Principio Página