

USERS

★★★★★
DOMINE EL
FRAMEWORK:
WPF, XAML
Y WCF

C#

AVANZADO

**SOLUCIONES REALES, CON LAS ESTRATEGIAS
DE DESARROLLO MÁS ACTUALES**

PROGRAMACIÓN AVANZADA PARA CLASES

CREACIÓN DE INTERFACES DE USUARIO CON WPF

DELEGADOS, EVENTOS Y FUNCIONES DE CALLBACK

por Nicolás Arrioja Landa Cosío

MANUALES USERS MANUALES USERS MANUALES USERS MANUALES USERS MAN

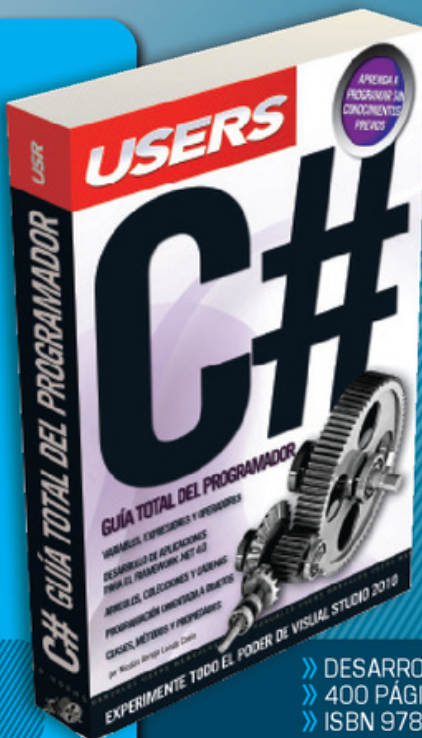
MAXIMICE LA INCREÍBLE POTENCIA DE ESTE LENGUAJE

CONÉCTESE CON LOS MEJORES LIBROS DE COMPUTACIÓN

LLEGAMOS A TODO EL MUNDO
VÍA **OCA** * Y **PHL** **

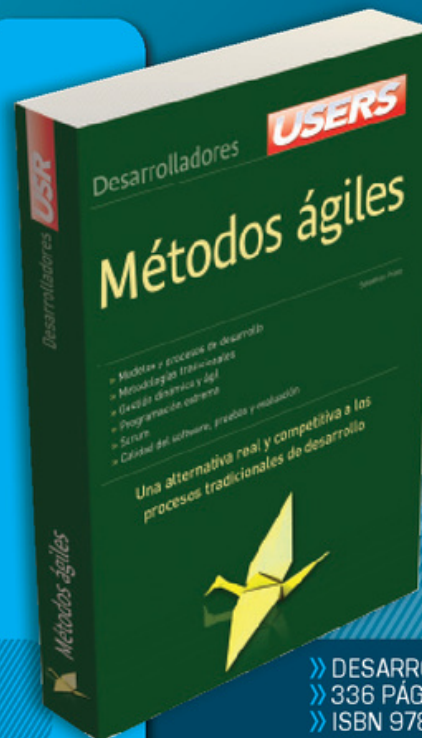
usershop.redusers.com
usershop@redusers.com

* SÓLO VÁLIDO EN LA REPÚBLICA ARGENTINA // ** VÁLIDO EN TODO EL MUNDO EXCEPTO ARGENTINA



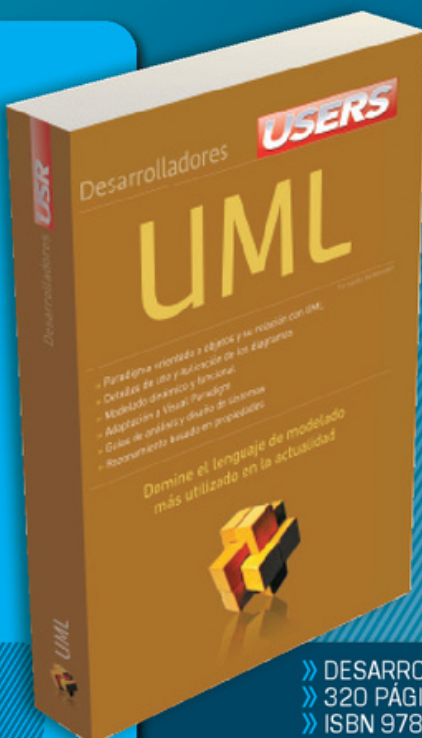
APRENDA A PROGRAMAR DESDE CERO EN C# CON VISUAL STUDIO 2010

» DESARROLLO / .NET
» 400 PÁGINAS
» ISBN 978-987-26013-5-5



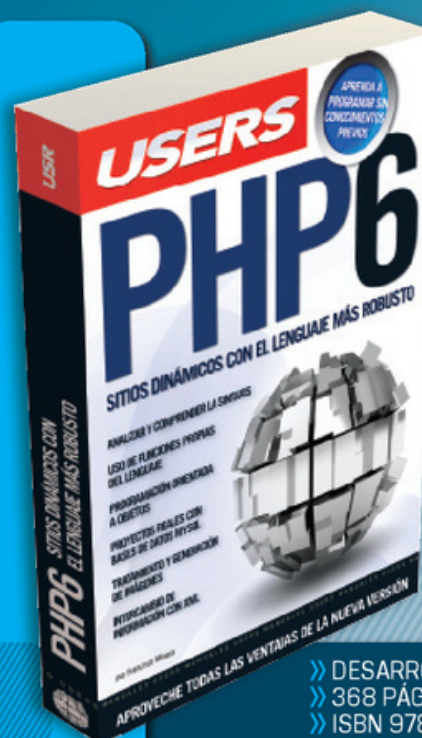
UNA ALTERNATIVA A LOS MÉTODOS TRADICIONALES DE DESARROLLO

» DESARROLLO
» 336 PÁGINAS
» ISBN 978-987-1347-97-1



COMIENCE A DESARROLLAR DESDE LA PERSPECTIVA DE LOS MODELOS

» DESARROLLO
» 320 PÁGINAS
» ISBN 978-987-1347-95-7



APRENDA A CREAR SITIOS DINÁMICOS CON EL LENGUAJE MÁS ROBUSTO

» DESARROLLO / INTERNET
» 368 PÁGINAS
» ISBN 978-987-663-039-9

C# AVANZADO

SOLUCIONES REALES, CON LAS ESTRATEGIAS
DE DESARROLLO MÁS ACTUALES

por Nicolás Arrioja Landa Cosio

RedUSERS



TÍTULO: C# avanzado
AUTOR: Nicolás Cosío
COLECCIÓN: Manuales USERS
FORMATO: 17 x 24 cm
PÁGINAS: 320

Copyright © MMXI. Es una publicación de Fox Andina en coedición con DALAGA S.A. Hecho el depósito que marca la ley 11723. Todos los derechos reservados. Esta publicación no puede ser reproducida ni en todo ni en parte, por ningún medio actual o futuro sin el permiso previo y por escrito de Fox Andina S.A. Su infracción está penada por las leyes 11723 y 25446. La editorial no asume responsabilidad alguna por cualquier consecuencia derivada de la fabricación, funcionamiento y/o utilización de los servicios y productos que se describen y/o analizan. Todas las marcas mencionadas en este libro son propiedad exclusiva de sus respectivos dueños. Impreso en Argentina. Libro de edición argentina. Primera impresión realizada en Sevagraf, Costa Rica 5226, Grand Bourg, Malvinas Argentinas, Pcia. de Buenos Aires en VIII, MMXI.

ISBN 978-987-1773-96-1

Cosío, Nicolás

C# avanzado. - 1a ed. - Buenos Aires : Fox Andina; Dalaga, 2011.

320 p. ; 24x17 cm. - (Manual users; 217)

ISBN 978-987-1773-96-1

1. Informática. I. Título

CDD 005.3



ANTES DE COMPRAR

EN NUESTRO SITIO PUEDE OBTENER, DE FORMA GRATUITA, UN CAPÍTULO DE CADA UNO DE LOS LIBROS EN VERSIÓN PDF Y PREVIEW DIGITAL. ADEMÁS, PODRÁ ACCEDER AL SUMARIO COMPLETO, LIBRO DE UN VISTAZO, IMÁGENES AMPLIADAS DE TAPA Y CONTRATAPA Y MATERIAL ADICIONAL.

RedUSERS
COMUNIDAD DE TECNOLOGÍA



redusers.com

Nuestros libros incluyen guías visuales, explicaciones paso a paso, recuadros complementarios, ejercicios, glosarios, atajos de teclado y todos los elementos necesarios para asegurar un aprendizaje exitoso y estar conectado con el mundo de la tecnología.



LLEGAMOS A TODO EL MUNDO VÍA OCA* Y DHL**

*SÓLO VÁLIDO EN LA REPÚBLICA ARGENTINA // ** VÁLIDO EN TODO EL MUNDO EXCEPTO ARGENTINA

 usershop.redusers.com //  usershop@redusers.com

Nicolás Arrioja Landa Cosio

Catedrático de IUNIVERTECH y de la Universidad Madero, elegido como mejor profesor de los ciclos 2006-2007 y 2007-2008, se dedica principalmente a la investigación, consulta y capacitación en áreas relacionadas con la realidad virtual, la visualización científica y también los videojuegos.

En 1997 desarrolló el primer videojuego de realidad virtual inmersiva en Latinoamérica, conocido como VRaptor. También generó el primer lenguaje de programación para realidad virtual en Latinoamérica, CND-VR, que es usado en el medio académico para enseñar los conceptos de programación de gráficas 3D. Ha sido catedrático en algunas de las universidades más importantes de México por más de diez años. Enseñó desde las bases de la programación en lenguaje C, hasta Inteligencia Artificial. Ha dictado más de 25 conferencias relacionadas con la realidad virtual y los videojuegos en diversas universidades de México. Posee una patente referida a interfaces cinemáticas para videojuegos. Diseñó el plan y el programa de estudios para la Maestría en Realidad Virtual y Videojuegos en IUNIVERTECH. Es autor de los libros *DirectX*, *Inteligencia Artificial* y *Robótica Avanzada*, de esta misma editorial. Actualmente, realiza una investigación sobre Inteligencia Artificial no-Algorítmica y diseño de hardware. Tiene disposición para realizar investigación y dar conferencias en estas áreas en cualquier parte del mundo.

Es posible encontrarlo en nicosio.com, en Facebook como Nicolas Cosio (NICOSIO), y en YouTube, en el canal nicosiored.



Dedicatorias

A Carla, gracias por tu amistad.

Prólogo



Todo evoluciona, especialmente, en el mundo tecnológico, donde ocurre a pasos acelerados. Continuamente salen al mercado nuevos dispositivos, más rápidos, más poderosos y, también, más pequeños. El poder de procesamiento de una computadora actual fácilmente puede ser 1000 veces superior al que tenía un equipo de cómputo hace un par de lustros.

Este avance del hardware ha potenciado una revolución en el mundo del software, que cuenta con capacidades cada vez más flexibles y sorprendentes. De este modo, el software pudo expandir su ámbito de influencia a prácticamente cualquier industria o rama de la actividad humana, desde las aplicaciones más complejas, desarrolladas para el control de bancos, bolsas de valores, aeropuertos, hospitales, etc.; hasta sofisticados y realistas juegos para computadora, consolas o dispositivos móviles.

En todos estos casos, la pregunta es: ¿cómo se hacen estas poderosas aplicaciones? La respuesta implica modernas herramientas de desarrollo, conocidas como frameworks o plataformas de programación, que se caracterizan por ser muy vastas, intuitivas y diseñadas para facilitarle la vida al programador, al permitirle generar mucho con relativamente poco esfuerzo y en menos tiempo que con los lenguajes de programación tradicionales. Así, en este siglo han surgido nuevos frameworks con novedosas funciones, con innumerables librerías de código, con herramientas de productividad como generadores automáticos de código, interfaces inteligentes, etc. Ante todo este panorama, surge un nuevo e interesante reto para los actuales programadores: conocer y dominar la filosofía y el funcionamiento de estos frameworks, para aprovechar sus capacidades y explotarlos en beneficio de la creación de innovadores sistemas de software.

Enmarcado en este contexto, sale a la luz el nuevo libro del Dr. Nicolás Arriola Landa Cosío, con su particular estilo, muy didáctico, claro y con ejemplos de código que ilustran cada uno de los temas abordados, con lo cual brinda al lector un método inmediato para poner en práctica lo expuesto.

El Dr. Nicolás Arriola Landa Cosío es un prolífico escritor en materia de software y hardware; ha publicado libros, artículos,

blogs tecnológicos y, además, es investigador, catedrático universitario con una larga trayectoria, y ponente en conferencias y prestigiosos congresos sobre temas como realidad virtual (disciplina en la que cuenta con una patente), animación y videojuegos, desarrollo de software e inteligencia artificial.

La presente publicación aborda el desarrollo de software con C# avanzado, lenguaje de programación que forma parte de Visual Studio 2010, con el cual explota las tecnologías más relevantes del .NET Framework de Microsoft. A lo largo de los diversos capítulos, se va acrecentando el conocimiento sobre todas las bondades que ofrece el lenguaje C# para los desarrolladores en el marco del .NET Framework.

En esta nueva aventura del autor por el mundo de la programación, se destaca el enfoque hacia la interfaz gráfica en la programación, partiendo desde el uso estándar de los Windows Forms, para entrar en los entornos más avanzados del manejo de interfaces de usuario, como WPF. Para el manejo de la interfaz se explica el funcionamiento de XAML, y en la línea de los dispositivos móviles, se le dedica un capítulo completo a la programación con Windows Phone 7. También se habla sobre la plataforma Silverlight, y, finalmente, para cerrar la obra con broche de oro, se aborda el tema de WCF (*Windows Communication Foundation*), de gran ayuda para la construcción de avanzadas aplicaciones distribuidas, de manera sencilla y eficiente.

De esta forma, el lector tiene en sus manos una valiosa herramienta para aprender a desarrollar avanzadas aplicaciones de software usando C#, uno de los lenguajes más poderosos y empleados en todo el mundo. Así, lo único que resta es poner manos a la obra, aplicando todos los conceptos y teniendo en mente que el único límite es nuestra capacidad creativa.

Jorge Castelán Galicia

*Maestro en Sistemas Computacionales
Catedrático y Representante de Iniciativas Académicas de IT
Universidad Madero
Puebla, México*



**Desarrollos temáticos
en profundidad**

Libros.

Coleccionables.

**Cursos intensivos
con multimedia**



**Capacitación
dinámica**

Revistas.

Sitios Web.

**Noticias al día,
downloads, comunidad**



**Información actualizada
al instante**

Newsletters.

*La red de productos sobre tecnología más
importante del mundo de habla hispana.*



redusers.com

El libro de un vistazo

En este libro encontraremos la ayuda fundamental que nos permitirá apropiarnos de los conocimientos necesarios para utilizar las características avanzadas del desarrollo de aplicaciones. A través de los capítulos que componen esta obra realizaremos tareas relacionadas con el desarrollo en C#.

*01



INTERFACES

En este capítulo nos encargaremos de realizar un breve repaso del concepto programático de interfaz, y de esta forma procederemos a conocer las más importantes características del framework de .NET. También analizaremos la manera en que podemos aprovecharlas para mejorar nuestro propio software. Así estaremos dando los pasos iniciales que necesitamos para comenzar a utilizar las características avanzadas relacionadas con el desarrollo de aplicaciones utilizando el lenguaje de programación C#.

*03



MANEJO AVANZADO DE CLASES

Este capítulo nos brinda el conocimiento para poder hacer un mejor uso de nuestras clases y, de esta forma, simplificar el código que las utilice. Con esto, nuestro código no solamente será más profesional, sino que también se volverá más limpio y fácil de entender en futuras revisiones. Siguiendo cada sección que compone este capítulo, estaremos preparados para realizar un manejo avanzado de las clases, y de esta forma, contar con herramientas que nos ayudarán en la generación de nuestros desarrollos.

*02



CALLBACK, DELEGADOS Y EVENTOS

Este es un tema con el que mucha gente tiene problemas, por lo que procedemos a explicarlo, paso a paso, desde la forma más primitiva hasta la más avanzada, para así, obtener un control completo sobre ellos y poder aprovechar sus opciones. A lo largo de este capítulo pondremos en práctica diversos ejemplos mediante los cuales podremos apropiarnos de las tareas relacionadas con los delegados, entendiendo su alcance y la forma en que podemos utilizarlos.

*04



SERIALIZACIÓN

El hecho de poder guardar el estado de los objetos a disco nos permite tener aplicaciones que almacenen datos y, más tarde, puedan recuperarlos para seguir trabajando. A través de las secciones que encontraremos en este capítulo nos volcaremos a la tarea de aprender que la serialización es sumamente sencilla de implementar en el framework de .NET; así, su uso nos ofrecerá la posibilidad de obtener aplicaciones eficientes con menos esfuerzo.

***05****INTRODUCCIÓN A WPF Y XAML**

WPF es una tecnología que nos permite hacer interfaces de usuario avanzadas, y es importante conocerla si deseamos desarrollar aplicaciones que puedan ser empleadas en dispositivos tales como Windows Phone 7. Este capítulo nos ayudará a realizarlo.

***06****XAML Y LAS INTERFACES DE USUARIO**

Este capítulo se encarga de complementar al anterior, ya que se continúa con la programación de interfaces de una forma didáctica. También es importante si tenemos interés en el desarrollo de aplicaciones para dispositivos móviles basados en Windows.

***07****PROGRAMAR PARA WINDOWS PHONE 7**

Con el conocimiento de los dos capítulos anteriores, ya podemos proceder a realizar

una aplicación sencilla para Windows Phone 7. En este capítulo nos dedicaremos conocer las interesantes características que ofrecen estos dispositivos.

***08****PROGRAMAR EN WCF**

Con WCF podemos realizar aplicaciones distribuidas sin los problemas y la complejidad del pasado. Es muy sencillo usar WCF, y tener aplicaciones que se lleven a cabo en diferentes computadoras y se comuniquen entre sí para resolver el problema.

***Ap****LOS ASSEMBLIES**

Todas las aplicaciones que hacemos se construyen al colocar una serie de assemblies que nos permiten obtener el resultado que estamos buscando. En el apéndice de esta obra nos encargaremos de analizar los assemblies y aprender a utilizarlos en todos nuestros desarrollos.

**INFORMACIÓN COMPLEMENTARIA**

A lo largo de este manual podrá encontrar una serie de recuadros que le brindarán información complementaria: curiosidades, trucos, ideas y consejos sobre los temas tratados. Para que pueda identificarlos en forma más sencilla, cada recuadro está identificado con diferentes iconos:

**CURIOSIDADES
E IDEAS****ATENCIÓN****DATOS ÚTILES
Y NOVEDADES****SITIOS WEB**

Contenido

Sobre el autor	4
Prólogo	5
El libro de un vistazo	10
Información complementaria.....	11
Introducción	12

*01

Interfaces

Entender las interfaces	14
Definir la interfaz	14
Comparación entre interfaces y clases abstractas	19
Creación de una aplicación simple con interfaz	19
El uso de IEnumerable	25
El uso de ICloneable	32
El uso de IComparable	39
Resumen	47
Actividades	48

*02

Callback, delegados y eventos

Conocer las interfaces de callback	50
Creación de la interfaz de callback	50
Entender el sink	51
Uso de más de un sink por evento	58
Cómo utilizar dos o más eventos	61
Los delegados	68
Ejemplo con delegado	69
Delegados con multicasting	78
Entender los eventos	84
Resumen	89
Actividades	90

*03

Manejo avanzado de clases

Comprender los indexers.....	92
Sobrecarga de operadores	98
Sobrecarga de operadores binarios	100
Los vectores	100
Aplicación con sobrecarga de operadores binarios	102
Sobrecarga de operadores unuarios	107
Sobrecarga de operadores de igualdad y comparación	112
Sobrecarga de los operadores < y >	118
Resumen	125
Actividades	126

*04

Serialización

Guardar el estado de los objetos.....	128
Serialización binaria sencilla.....	130
Serialización SOAP sencilla	138
Serialización XML sencilla.....	143
Serialización de clases con composición en formato binario.....	146
Serialización con composición usando XML	153
Serialización de varios objetos	157
Resumen	163
Actividades	164

*05

Introducción a WPF y XAML

Entender WPF	166
Arquitectura de WPF	167
Entender los contenedores	169

Los paneles	169
Primera aplicación WPF	194
Resumen	197
Actividades	198

* 06

XAML y las interfaces de usuario

Efectos y brochas para el fondo de Border	200
Degradado lineal	204
Degradado radial	213
Control de marcos	219
Creación de una aplicación que controle a la interfaz	228
Resumen	233
Actividades	234

* 07

Programar para Windows Phone 7

Entender Windows Phone 7	236
Creación de un proyecto para Windows Phone	237
Entender el hardware	239
Colocar la orientación de la aplicación	242

El sistema multi-touch	244
Crear una aplicación de prueba	245
Resumen	265
Actividades	266

* 08

Programar en WCF

Entender Windows Communication Foundation	268
El modelo de comunicación	272
Creación de aplicaciones WCF	277
Código para un servicio WCF	283
Resumen	297
Actividades	298

* Ap

Los assemblies

Entender los assemblies	300
Entender los namespaces	302
Creación y uso de un assembly	303

*

Servicios al lector

Índice temático	314
-----------------------	-----

Introducción



Hace algunos años, publiqué en esta misma editorial un curso introductorio sobre programación en C#. Sin embargo, mucho del poder de este lenguaje se encuentra en el uso del framework de .NET, por lo que se hace necesario continuar el trabajo iniciado años atrás, con otra obra que permitiera al lector alcanzar niveles más avanzados de programación.

En este libro nos encargamos de cubrir temas que permitirán a los lectores realizar aplicaciones más complejas y, al mismo tiempo, facilitarán la tarea de desarrollo.

Para este libro he seleccionado los temas que considero más importantes, y por medio de diversos ejemplos, trataré de explicarlos de la manera más sencilla posible, para que se logre comprender el poder de la herramienta que estamos utilizando. También he incorporado tecnologías que se usan en otras plataformas, pero que se desarrollan con C#.

El framework de .NET es muy grande y aún quedan suficientes temas para escribir otro libro más. No hay que olvidar que este framework sigue creciendo, por lo que recomiendo al lector que no se quede solo con el conocimiento que puede adquirir en estas páginas, sino que continúe por sí mismo acrecentando lo aprendido sobre los temas tratados y, al mismo tiempo, informándose a partir de diversas fuentes adicionales.

Espero que este libro sea de utilidad a todos los lectores.

Nicolás Arrioja Landa Cosio, Ph. D.



Interfaces

Es muy importante entender que las interfaces son elementos de programación que nos permitirán desarrollar aplicaciones avanzadas. Con ellas podremos crear polimorfismo y, en cierta forma, hasta vencer la limitante de herencia múltiple que tiene el lenguaje de programación C#. También podremos acceder a definir comportamientos comunes para diversas clases, incluso si ellas no están relacionadas entre sí. A lo largo de este capítulo podremos comprender el alcance de las interfaces y también practicar con algunas implementaciones prácticas que nos ayudarán a ejercitar las habilidades que necesitamos para explotar todo su potencial.

▼ Entender las interfaces 14	▼ El uso de ICloneable 32
Definir la interfaz 14	
Comparación entre interfaces	▼ El uso de
y clases abstractas 19	IComparable..... 39
Creación de una aplicación	▼ Resumen..... 47
simple con interfaz..... 19	
▼ El uso de IEnumerable 25	▼ Actividades..... 48



Entender las interfaces

Vamos a utilizar las interfaces como un medio para poder definir comportamientos. Estos comportamientos estarán agrupados dentro de las interfaces. Es natural que, de alguna manera, estén relacionados entre sí, ya sea lógica o funcionalmente. Los comportamientos que tenemos no son otra cosa que métodos, pero cuando definimos la interfaz, solamente usamos **métodos abstractos**.

Los métodos serán implementados por las clases que implementen la interfaz; es decir que solamente indicamos qué comportamientos esperamos de alguien que implementa la interfaz, pero no decimos cómo deben llevarse a cabo esos comportamientos. En otras palabras, la interfaz solo expresa comportamientos.

Definir la interfaz

Debemos tener en cuenta que la interfaz se define directamente en el código de nuestra aplicación, ya sea de forma manual o aprovechando las herramientas que ofrece **Visual Studio 2010**.

Las interfaces deben tener un nombre, que, por regla general, se inicia con la letra "I" mayúscula seguida de una palabra que nos recuerde el comportamiento o concepto relacionado a los métodos de la interfaz. Los nombres de las interfaces tienen que ser únicos. Es necesario que utilicemos la palabra clave **interface**. Los métodos de la interfaz se declaran dentro de un bloque de código.

Las interfaces no precisan una clase base, y algo curioso que debemos recordar con relación a ellas, y que las diferencia de otros tipos dentro del framework de .NET, es que no requieren **Object** como clase base; de esta forma, no será necesario definirla.



MÉTODO ABSTRACTO



Un método abstracto es aquel que solamente es definido pero no implementado; es decir que únicamente indicamos el prototipo del método con su tipo de retorno, nombre y parámetros por utilizar, pero no colocamos el código que lo implementa. Los métodos abstractos deberán ser implementados en alguna clase que descienda de aquella en la que fueron definidos.

Cuando definimos los métodos dentro de la interfaz, debemos notar que no nos hace falta colocar ningún modificador de acceso, ya que se considera que los métodos en las interfaces son, por default, públicos.

Veamos la manera en que podemos definir una interfaz:

```
public interface ImiInterfase
{
    // Aqui definimos los métodos
    int Metodo1( );
    bool Metodo2(int a, int b);
}
```

En primer lugar, le damos un acceso público a la interfaz y declaramos una que se llama **ImiInterfase**. Esta definirá un comportamiento que necesita de dos métodos: el primero es de tipo **int** y no precisa ningún parámetro; el segundo es de tipo **bool** y requiere dos parámetros.

Una vez que tenemos definida la interfaz, hay que implementarla. Una o varias clases pueden llevar a cabo la implementación, y no es necesario que esas clases estén relacionadas entre sí lógicamente o jerárquicamente. Lo único que se precisa es que la clase o las clases realmente necesiten del comportamiento que nos encargamos de declarar en la interfaz.

Para hacer esto, debemos indicar que la clase que deseamos implementará la interfaz; entonces, en su declaración, después del nombre, colocamos el operador **:** y posteriormente el nombre de la interfaz que implementará la clase.

UNA VEZ QUE
TENEMOS DEFINIDA
LA INTERFAZ,
PROCEDEMOS A
IMPLEMENTARLA

”



SYSTEM.OBJECT



Object es el supertipo utilizado en el lenguaje C#; todos los demás tipos que se incorporan dentro del framework de .NET o que son definidos por el programador descienden de él. Esto es importante porque podemos llevar a cabo operaciones polimórficas con nuestras propias clases y definir métodos que puedan trabajar con cualquier clase como parámetro.

En el interior del bloque de código de la clase debemos recordar la necesidad de poner, como siempre, los datos y los métodos propios de ella. Luego, tenemos que colocar las implementaciones de los métodos definidos en la interfaz. Estas no son necesariamente iguales en todas las clases que implementan la interfaz; más bien, debemos implementar ese comportamiento de acuerdo con las necesidades particulares de la clase correspondiente.

```
public class miClase : ImiInterfase
{
    // Datos y métodos propios de la clase
    ...
    // Implementación de los métodos de la interfaz
    public int Metodo1()
    {
        // Aquí el código que implementa este método
        ...
    }
    public bool Metodo2(int a, int b)
    {
        // Aquí el código que implementa al método
        ...
    }
}
```

Recordemos que, si la clase tiene una herencia, inmediatamente después de los dos puntos (:) procedemos a colocar el nombre de la clase padre seguido de una coma (,) y, posteriormente, el nombre de la interfaz que tenemos por implementar:



DESCARGA DE C# EXPRESS



Es interesante saber que si necesitamos contar con el compilador del lenguaje C#, podemos usar la versión denominada Express, que es gratuita y puede descargarse directamente del sitio web de Microsoft. A pesar de no tener costo, se trata de una aplicación que permite realizar una gran cantidad de programas. Para obtenerla, nos dirigimos a www.microsoft.com/express/Windows.


```
public class miClase : clasePadre, ImiInterfase
{
    ...
}
```

Se puede dar el caso de que la clase que tenemos implemente más de una interfaz. Para esto, debemos proceder a listar las interfaces por implementar después de :, separadas por comas:

```
public class miClase : ImiInterfase, ImiOtraInterfase
{
    ...
}
```

En todos los casos, debemos implementar todos los métodos de todas las interfaces; si no lo hacemos así, tendremos un error de compilación.

Adicionar interfaces a una aplicación

Existen dos formas de adicionar una interfaz a nuestra aplicación. La primera consiste en agregar solo el código de la interfaz en cualquier parte válida del **namespace** de la aplicación; esto puede hacerse a mano.

La segunda forma de realizar esta tarea es utilizar las herramientas propias de Visual Studio 2010. Esto es un poco más rápido y cómodo, y presenta la ventaja de que crea el documento donde se guardará el código propio de la interfaz en nuestro proyecto.

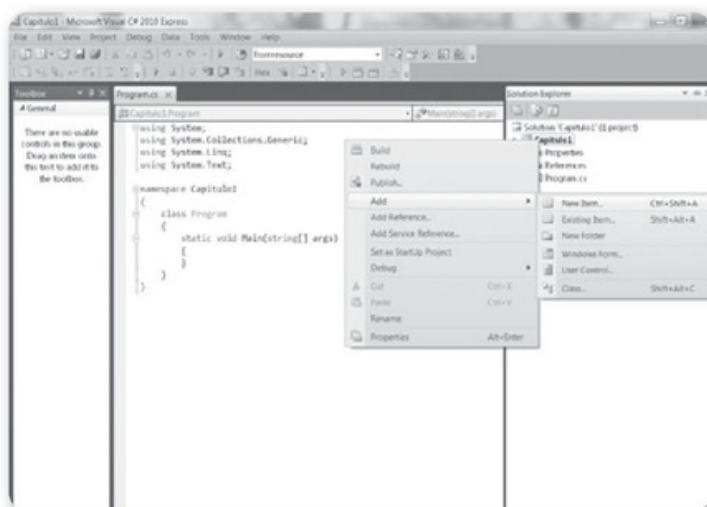
Para esto, nos dirigimos al **Explorador de soluciones** y, haciendo clic derecho del mouse sobre el proyecto, veremos un menú contextual. Allí seleccionamos **Add** o **Adicionar** y, luego, **New Item** o **Elemento Nuevo**.



HERENCIA

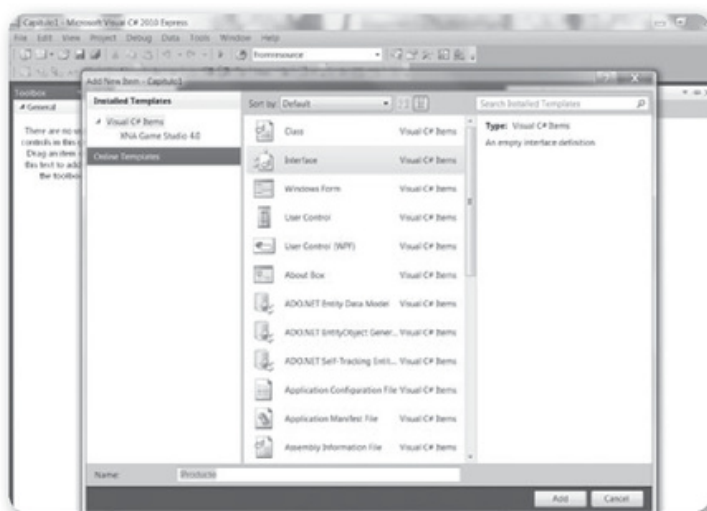


La herencia es una de las bases de la programación orientada a objetos. Esta interesante característica nos permite llevar a cabo la reutilización de código y crear una relación de tipo es-un con otra clase. Debemos recordar que, con la herencia, la clase hija hereda las características del padre, con lo cual ahorramos código y podemos extender las capacidades de nuestra aplicación.



► **Figura 1.** Podemos observar la selección del menú en el **Ex-plorador de Soluciones**, que nos permitirá adicionar la interfaz.

Aparecerá una ventana en la cual podemos elegir el elemento por adicionar; allí buscamos **Interface** o **Interfase**. En la parte inferior colocamos el nombre de la interfaz, por ejemplo, **IProducto**. Después, simplemente oprimimos el botón **Add**.



► **Figura 2.** En este cuadro de diálogo debemos seleccionar el elemento **Interface** y darle nombre.

Debemos tener en cuenta que el entorno de desarrollo Visual Studio crea por nosotros un nuevo documento en el cual se encarga de colocar un esqueleto para la interfaz que hemos generado. A partir de este momento, solo nos resta proceder a colocar los diferentes prototipos de los métodos que necesitamos implementar.

Comparación entre interfaces y clases abstractas

Es posible que no se note la ventaja de usar interfaces, ya que, al parecer, son similares a las clases abstractas. En una clase abstracta, podemos definir métodos abstractos que serán implementados por las clases hijas. Sin embargo, hay una característica importante que marca la diferencia. En la clase abstracta también es posible definir datos e, incluso, métodos concretos, que serán heredados por las clases hijas. Por el contrario, la interfaz solo define el comportamiento; es decir, aísla conceptualmente el comportamiento y no contamina con datos o métodos innecesarios a aquellas clases que la implementan.

Como es sabido, en los lenguajes administrados de .NET no es posible llevar a cabo herencia múltiple. Pero las interfaces nos permiten, de alguna forma, acercarnos un poco a esta funcionalidad. Si bien no es posible hacer la herencia múltiple, sí se puede tener clases que implementen varias interfaces, con diferentes comportamientos. La desventaja es la falta de reutilización de código.

Creación de una aplicación simple con interfaz

Empecemos por crear una aplicación sencilla que utilice una interfaz. Cuando vamos a usar una interfaz, lo primero que debemos hacer es pensar en cuál es el comportamiento que va a representar dicha interfaz. En nuestro caso, crearemos una que represente un producto. Este tendrá como comportamientos calcular el impuesto y mostrar la información correspondiente al producto.



DOCUMENTACIÓN DE ICOMPARABLE



Siempre es importante tener mayor información sobre los elementos con los cuales trabajamos. El hecho de conocer las interfaces de .NET a fondo nos da más poder en nuestros programas. Podemos encontrar la documentación de la interfaz **IComparable** en el siguiente sitio web: <http://msdn.microsoft.com/en-us/library/system.icomparable.aspx>.

También procederemos a crear una clase que implementará la interfaz. Esta clase será para televisores y tendrá la implementación de los métodos definidos en la interfaz.

Empecemos por crear una aplicación nueva de C# usando Visual Studio 2010. El tipo de proyecto es una aplicación de consola. Una vez que la aplicación se haya creado, adicionamos una interfaz, usando cualquiera de los métodos previamente mencionados.

El nombre que le damos a la interfaz es **IProducto** y, en su interior, colocamos el prototipo para los métodos **CalculaImpuesto()** y **MuestraInformacion()**. El código debe lucir de la siguiente manera:

```
// Esta interfase define el comportamiento para el producto
interface IProducto
{
    // Metodos que componen al comportamiento del producto
    // Calcular el impuesto
    void CalculaImpuesto(double imp);
    // Mostrar informacion
    void MuestraInformacion();
}
```

Como podemos ver, en su interior solo declaramos el método, pero no incluimos ningún código que lo implemente. La interfaz únicamente define cuáles son los comportamientos; no, cómo se llevan a cabo.

El método de **CalculaImpuesto()** recibe un parámetro de tipo **double**, que es el porcentaje de impuesto. Debemos saber que el método **MuestraInformacion()** no recibe ningún parámetro.

Continuamos con la tareas de adicionar una clase que se llama **CTelevisor**, cuyo código es el siguiente:



DOCUMENTACIÓN SOBRE ICONEABLE



También encontraremos la documentación de la interfaz denominada **ICloneable** en el sitio web de MSDN que provee Microsoft. Por esta razón, debemos tener en cuenta que se recomienda hacer uso de la clonación para evitar corrupción de información y confusiones entre variables e instancias: <http://msdn.microsoft.com/en-us/library/system.icloneable.aspx>.

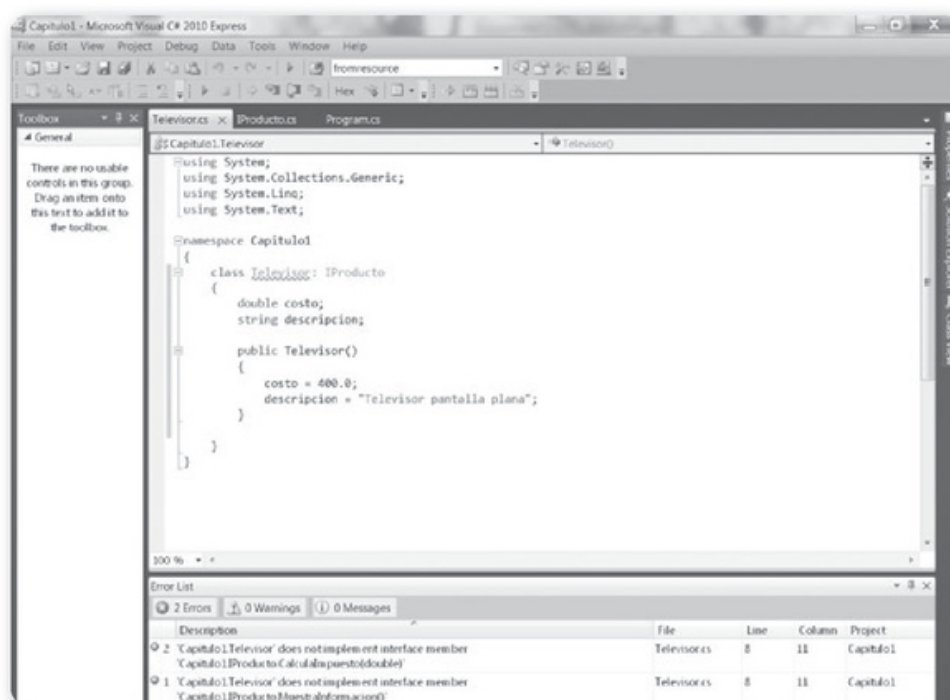
```
class CTelevisor: IProducto
//Indicamos que CTelevisor implementa a IProducto

{
    double costo;
    double impuesto;
    string descripcion;
    // Metodos propios
    // Constructor
    public CTelevisor()
    {
        // Inicializamos los datos
        costo = 400.0;
        descripcion = "Televisor pantalla plana";
    }
    public void ColocarCosto(double nCosto)
    {
        // Asignamos el nuevo costo
        costo = nCosto;
    }
    // Metodos a implementar debido a la interfaz
    public void CalculaImpuesto(double imp)
    {
        // Calculamos el impuesto
        impuesto = costo * imp;
    }
    public void MuestraInformacion()
    {
        // Mostramos la informacion necesaria
        Console.WriteLine("--- Producto televisor");
        Console.WriteLine(descripcion);
        Console.WriteLine("Costo {0}, Impuesto {1}", costo, impuesto);
    }
}
```

Lo primero que notamos es que indicamos que la clase implementa a **IProducto**. Esto se hace utilizando el operador **:** seguido del nombre de la

interfaz. Enseguida colocamos el código de la clase de forma tradicional, y definimos los datos y los métodos propios de la clase. Estos métodos son el constructor y un método para colocar un nuevo valor al costo.

Una vez que hemos finalizado con todos los elementos propios de la clase, continuamos con la implementación del comportamiento de la interfaz; es decir, colocamos el código de los métodos de la interfaz según la forma particular en que **CTelevisor** los lleva a cabo. Es necesario implementar todos los métodos de la interfaz.



► **Figura 3.** Aquí observamos los mensajes de error que nos da el compilador si no implementamos los métodos de la interfaz.



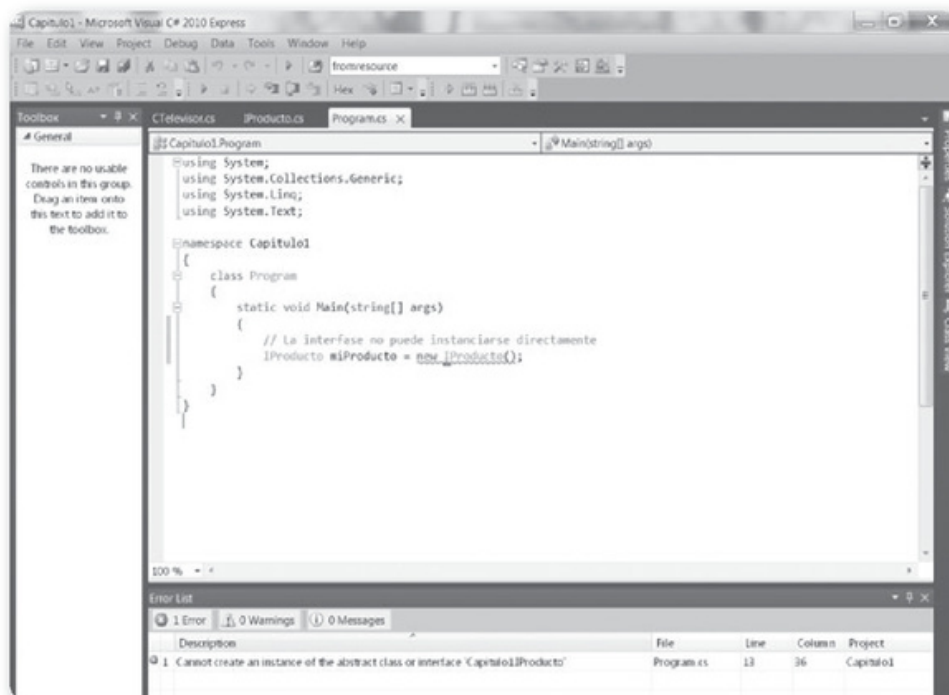
DOCUMENTACIÓN DE IENUMERABLE

Tengamos en cuenta que otra interfaz importante que precisamos conocer es **IEnumerable**, cuya documentación siempre nos ofrece información útil y también nuevos conocimientos que podemos aplicar en nuestros proyectos. Esta interfaz nos brinda una gran flexibilidad en nuestros programas: <http://msdn.microsoft.com/en-us/library/system.collections.ienumerable.aspx>.

Ahora pasamos a crear el código que va dentro de la función denominada **Main()**, pero primero llevaremos a cabo un experimento. Empecemos por colocar el siguiente código:

```
class Program
{
    static void Main(string[] args)
    {
        // La interfaz no puede instanciarse directamente
        IProducto miProducto = new IProducto();
    }
}
```

Si intentamos compilar la aplicación, obtendremos un error. Esto se debe a que no es posible instanciar una interfaz. Lo que debemos hacer es crear instancias de las clases que la implementan. El error que se presenta puede observarse en la **Figura 4**.

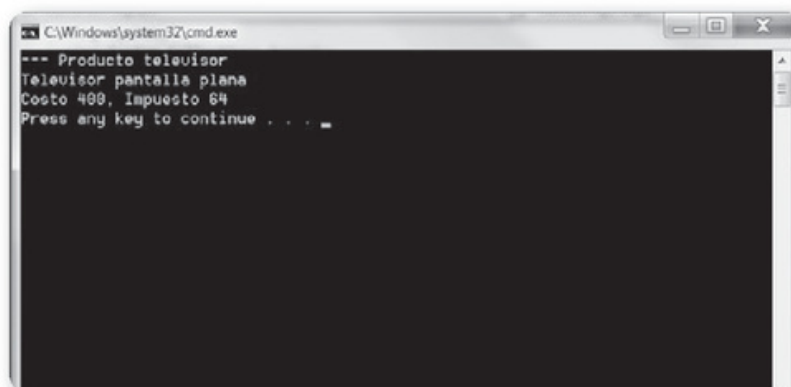


► **Figura 4.** En esta imagen observamos el error obtenido al tratar de instanciar la interfaz en forma directa.

A continuación, modifiquemos otra vez el programa que estamos creando, colocando ahora el código de manera correcta:

```
class Program
{
    static void Main(string[] args)
    {
        // La interfaz no puede instanciarse directamente
        // IProducto miProducto = new IProducto();
        // Instanciamos una clase que implementa a la interfaz
        CTelevisor miTV = new CTelevisor();
        // Invocamos a los comportamientos de la interfaz
        miTV.CalculaImpuesto(0.16);
        miTV.MuestraInformacion();
        // Invocamos un metodo propio de la clase
        miTV.ColocarCosto(567.82);
        miTV.CalculaImpuesto(0.16);
        miTV.MuestraInformacion();
    }
}
```

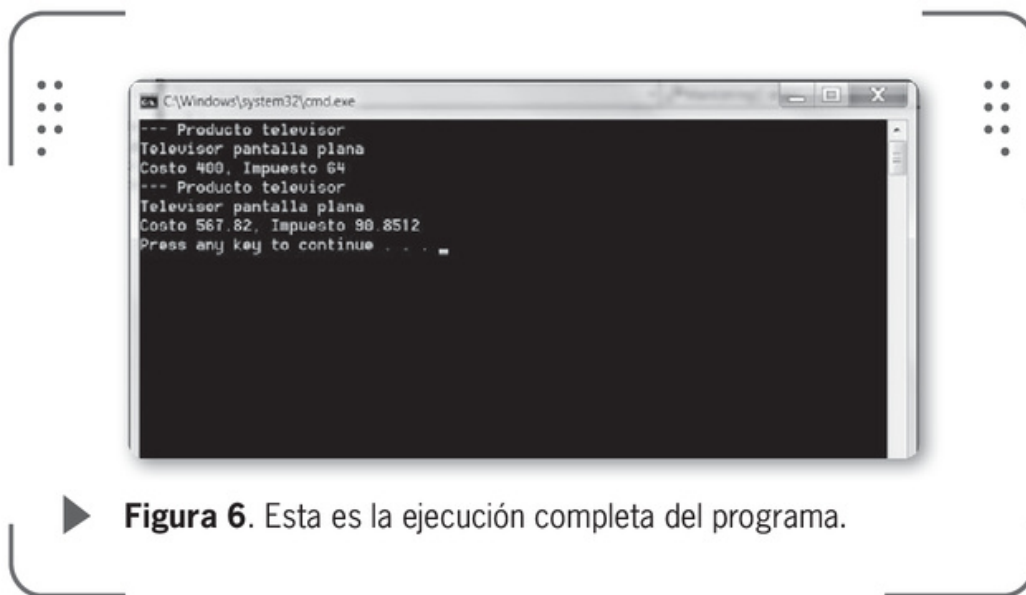
Empezamos por comentar el código erróneo. Enseguida creamos un objeto llamado **miTV**, que tiene como clase a **CTelevisor**; recordemos que esta clase ha implementado la interfaz.



► **Figura 5.** Observamos la ejecución del programa mostrando únicamente la ejecución de los métodos de la interfaz.

Ahora es posible invocar a los métodos de la clase, tanto a los propios como a aquellos que implementan la interfaz. Empezamos invocando los métodos del comportamiento de la interfaz correspondiente. Luego, invocamos el método de la clase y volvemos a invocar los de la interfaz. La invocación de los métodos puede llevarse a cabo en cualquier orden, según lo requiera la lógica del programa.

Compilemos la aplicación otra vez; no debería de haber ningún error, y el programa tendría que ejecutarse correctamente.



► **Figura 6.** Esta es la ejecución completa del programa.

El uso de IEnumerable

El framework de .NET es muy grande y poderoso: debemos tener en cuenta que dentro de él hay muchas clases que brindan apoyo para el desarrollo de aplicaciones, pero también hay varias interfaces definidas que nos permiten llevar a cabo operaciones avanzadas. Una de estas interfaces es la llamada **IEnumerable**. Para continuar, veamos primero dónde existe la necesidad de usar esta interfaz y, así, será más fácil entenderla. Muchas veces tenemos clases en cuyo interior se guardan colecciones de otros objetos, y desde el exterior necesitamos acceder a ellos. Por ejemplo, una forma guarda en su interior una colección de controles. En ocasiones es necesario recorrer todos esos controles, y podemos hacerlo por medio de **foreach**.

Tratemos de usar **foreach** en este contexto en una aplicación que no implemente **IEnumerable**. Para hacerlo, crearemos una clase que represente a una tienda de televisores, y en su interior tendremos un arreglo de la clase **CTelevisor**. Luego, intentaremos recorrer todos los televisores usando un **foreach**. Para empezar, nos encargamos de generar un nuevo proyecto y luego creamos otra vez nuestra interfaz **IProducto** y la clase **CTelevisor**, como vemos a continuación:

```
// Esta interfaz define el comportamiento para el producto
interface IProducto
{
    // Metodos que componen al comportamiento del producto
    // Calcular el impuesto
    void CalculaImpuesto(double imp);
    // Mostrar informacion
    void MuestraInformacion();
}
class CTelevisor: IProducto
//Indicamos que CTelevisor implementa a IProducto

{
    double costo;
    double impuesto;
    string descripcion;
    // Metodos propios
    // Constructor

    public CTelevisor(string desc, double pcosto)
    {
```



ENTENDER FOREACH



El **foreach** es un iterador que nos permite recorrer todos los objetos de un ADT de manera sencilla, como si se tratara de un ciclo. Esto simplifica la cantidad de código necesaria para hacerlo. El iterador recorre elemento por elemento de forma secuencial, según las reglas propias del ADT. En el momento en que no haya más elementos disponibles, finaliza.

```
// Inicializamos los datos
costo = pcosto;
descripcion = desc;
}
public void ColocarCosto(double nCosto)
{
    // Asignamos el nuevo costo
    costo = nCosto;
}
// Metodos a implementar debido a la interfaz
public void CalculaImpuesto(double imp)
{
    // Calculamos el impuesto
    impuesto = costo * imp;
}

public void MuestraInformacion()
{
    // Mostramos la informacion necesaria
    Console.WriteLine("--- Producto televisor");
    Console.WriteLine(descripcion);
    Console.WriteLine("Costo {0}, Impuesto {1}", costo, impuesto);
}
}
```

A continuación, nos dedicamos a la tarea de realizar la creación de la clase que representa a la tienda de televisores:



MSDN



Es muy importante tener en cuenta que uno de los sitios más importantes que nos conviene visitar cuando desarrollamos para .NET es MSDN. Se trata de un sitio web implementado y ofrecido por Microsoft, razón por la cual contiene información relevante para los desarrolladores y entusiastas de la programación. En él encontraremos toda la documentación y también una gran selección de ejemplos para las clases y métodos que componen .NET. Recordemos que es posible hallar información sobre todas las versiones de .NET. El sitio se encuentra en la dirección www.msdn.com.

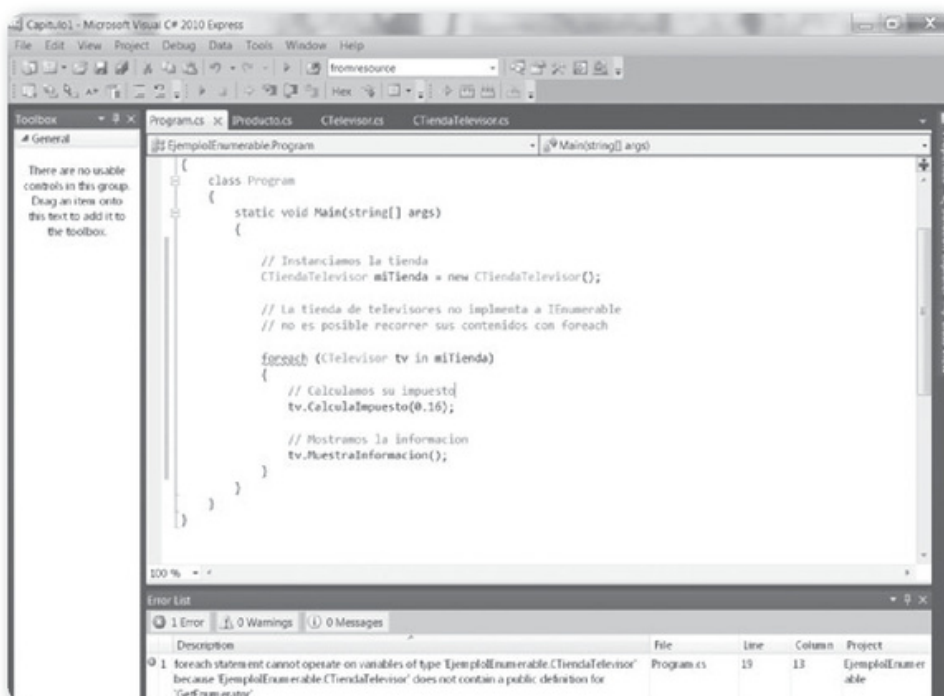
```
class CTiendaTelevisor // No podemos recorrer una clase normal con foreach
{
    // Guardamos los televisores de la tienda
    private CTelevisor[] televisores;
    public CTiendaTelevisor()
    {
        // El arreglo va a contener 3 televisores
        televisores = new CTelevisor[3];
        televisores[0] = new CTelevisor("Televisor LCD", 1500.0);
        televisores[1] = new CTelevisor("Televisor Plasma", 1800.50);
        televisores[2] = new CTelevisor("Televisor LED", 2500.75);
    }
}
```

Esta clase tiene un arreglo de tipo **CTelevisor** llamado **televisores**. En el constructor de la clase, instanciamos el arreglo con un tamaño de **3**. Colocamos los tres televisores dentro del arreglo y tenemos una clase que guarda en su interior un arreglo.

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

        // Instanciamos la tienda
        CTiendaTelevisor miTienda = new CTiendaTelevisor();
        // La tienda de televisores no implementa a IEnumerable
        // no es posible recorrer sus contenidos con foreach
        foreach (CTelevisor tv in miTienda)
        {
            // Calculamos su impuesto
            tv.CalculaImpuesto(0.16);
            // Mostramos la informacion
            tv.MuestraInformacion();
        }
    }
}
```


Como podemos observar, estamos creando una instancia de la clase **CTiendaTelevisor** y, luego, tratamos de recorrer todos los televisores en su interior. El código que tenemos ahí es correcto desde la perspectiva de la sintaxis, no tiene ningún problema. Intentemos compilar el proyecto y veamos qué ocurre: obtenemos un error, ya que la clase que queremos recorrer con **foreach** no implementa a la interfaz **IEnumerable**.



► **Figura 7.** Este error se presenta cuando tratamos de usar **foreach** en una clase que no implementa **IEnumerable**.

Entender IEnumerable

La interfaz **IEnumerable** es pequeña, porque solamente define un método. Está definida de la siguiente forma:

```
public interface IEnumerable
{
    IEnumerator GetEnumerator();
}
```

Cuando usamos **foreach**, este espera al método **GetEnumerator()** para poder funcionar correctamente. Algo que debemos observar es que **GetEnumerator()** regresa un objeto de tipo **IEnumerator**. Esta nueva interfaz está muy ligada con el iterador. Veamos cómo está definida:

```
public interface IEnumerator
{
    bool MoveNext();
    object Current { get; }
    void Reset;
}
```

Como podemos notar en el código anterior, los métodos se encuentran directamente ligados a las operaciones que realiza un iterador. De esta forma, el método denominado **MoveNext()** permite al iterador moverse al siguiente elemento, con la propiedad de **Current** obtenemos el elemento actual, y el método **Reset()** se encarga de colocar la posición actual antes del primer elemento listado, para permitir volver a recorrer la colección correspondiente.

En general, nosotros no necesitamos implementar los métodos de la interfaz **IEnumerator**, pero es posible hacerlo. Esto solamente debemos realizarlo si conocemos la manera en que trabaja el iterador y la colección o ADT que vamos a recorrer.

Creación de una aplicación con IEnumerable

Ahora ya podemos crear la aplicación con el uso de **IEnumerable**, para lo cual vamos a modificar el ejemplo anterior. Para realizar esta acción debemos comenzar con la clase **CTelevisor**:



ENTENDER EL ITERADOR



Recordemos que el iterador es un patrón de diseño que recorre uno por uno todos los elementos de un ADT. De esta forma, cada vez que pasamos por el iterador, recibimos un nuevo elemento, y esto continúa hasta que ya no hay más. El iterador puede ser diseñado por nosotros, y también es posible hacer uso de las clases que ya lo tienen implementado.

```
class CTiendaTelevisor : IEnumerable
{
    // Guardamos los televisores de la tienda
    private CTelevisor[] televisores;
    public CTiendaTelevisor()
    {
        // El arreglo va a contener 3 televisores
        televisores = new CTelevisor[3];
        televisores[0] = new CTelevisor("Televisor LCD", 1500.0);
        televisores[1] = new CTelevisor("Televisor Plasma", 1800.50);
        televisores[2] = new CTelevisor("Televisor LED", 2500.75);
    }
    // Implementamos a IEnumerable
    // El metodo regresa un objeto que implementa a IEnumerator
    // Da el comportamiento de un iterador
    public IEnumerator GetEnumerator()
    {
        return televisores.GetEnumerator();
    }
}
```

El primer cambio que notamos es la indicación de que **IEnumerable** será implementado por la clase **CTiendaTelevisor**. El resto de la clase no sufre modificaciones, pero al final tenemos una adición de código. Como la clase ahora va a implementar **IEnumerable**, posteriormente necesitamos colocar el código del método **GetEnumerator()**.

Este código es sencillo, porque estamos usando un arreglo; la clase **Array** tiene un método propio que se llama **GetEnumerator()** y nos permite obtener un objeto de tipo **IEnumerator** para ser utilizado por el **foreach**.

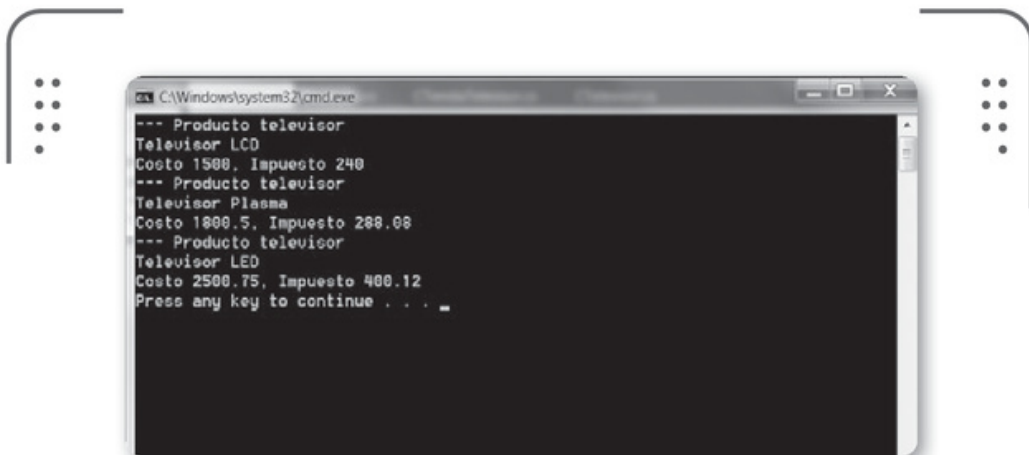


MODIFICADORES DE ACCESO



El modificador de acceso nos permite indicar quién tiene acceso a determinado método o dato. Puede ser público, privado o protegido. El acceso público permite que el exterior del objeto acceda o invoque al dato o método. En el privado, solo el objeto puede acceder al dato; en tanto que en el protegido, únicamente pueden hacerlo las clases en la cadena de herencia.

Es necesario tener en cuenta que si compilamos y ejecutamos la aplicación que estamos desarrollando, podremos darnos cuenta de que ahora es posible acceder a realizar un recorrido a través de los contenidos de la tienda de televisores que implementamos.



► **Figura 8.** En esta figura vemos la ejecución de la aplicación y el recorrido al arreglo de la tienda de televisores.

El uso de ICloneable

Debemos tener en cuenta que algunas veces es necesario obtener una copia de un objeto determinado, pero haciéndolo de tal manera que el objeto original no se modifique si llevamos a cabo cambios en el nuevo. Para efectuar esta acción que se conoce como **clonado** debemos seguir algunas indicaciones. Al principio, podemos pensar que el clonado se realiza directamente haciendo unas asignaciones, pero en la mayor parte de las ocasiones, este no suele ser el caso.

Empecemos por probar qué es lo que sucede si hacemos uso únicamente de las asignaciones para realizar la tarea de clonar un objeto. Para comprobar esto, procederemos a crear una nueva aplicación, que tendrá una clase tradicional no clonable y también poseerá una clase clonable. De este modo, podremos acceder a realizar la observación de la diferencia que se presenta entre las dos.

En este caso colocamos la interfaz **IProducto** y la clase **CTelevisor**, que ya conocemos de ejemplos anteriores:

```
// Esta interfaz define el comportamiento para el producto
interface IProducto
{
    // Metodos que componen al comportamiento del producto
    // Calcular el impuesto
    void CalculaImpuesto(double imp);
    // Mostrar informacion
    void MuestraInformacion();
}

class CTelevisor: IProducto //Indicamos que CTelevisor implemeta a IProducto
{
    double costo;
    double impuesto;
    string descripcion;
    // Metodos propios
    // Constructor
    public CTelevisor()
    {
        // Inicializamos los datos
        costo = 400.0;
        descripcion = "Televisor pantalla plana";
    }
    public void ColocarCosto(double nCosto)
    {
        // Asignamos el nuevo costo
        costo = nCosto;
    }
    // Metodos a implementar debido a la interfaz
    public void CalculaImpuesto(double imp)
    {
        // Calculamos el impuesto

        impuesto = costo * imp;
    }
    public void MuestraInformacion()
    {

```

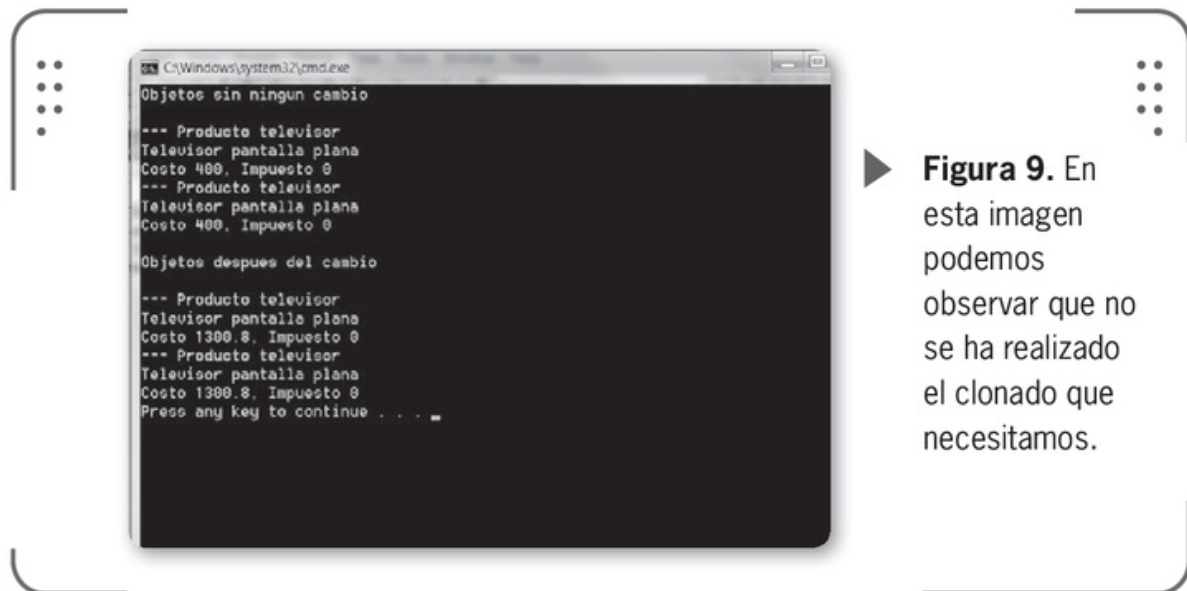
```
// Mostramos la informacion necesaria
Console.WriteLine("--- Producto televisor");
Console.WriteLine(descripcion);
Console.WriteLine("Costo {0}, Impuesto {1}", costo, impuesto);
    }
}
```

La clase **CTelevisor** no implementa **ICloneable**, pero intentemos llevar a cabo una clonación y observemos qué sucede. En la función **Main()** debemos poner el siguiente código:

```
class Program
{
    static void Main(string[] args)
    {
        // Ejemplo con objetos que no se han clonado
        CTelevisor miTV = new CTelevisor();
        CTelevisor tuTV = miTV;
        // Aquí pensamos que hicimos el clonado
        // Vemos los dos objetos
        Console.WriteLine("Objetos sin ningun cambio \n");
        miTV.MuestraInformacion();
        tuTV.MuestraInformacion();
        // Modificamos un objeto
        tuTV.ColocarCosto(1300.80);
        // Vemos los dos objetos
        Console.WriteLine("\nObjetos despues del cambio \n");
        miTV.MuestraInformacion();
        tuTV.MuestraInformacion();
    }
}
```

En primer lugar, creamos una instancia de **CTelevisor** a la cual llamamos **miTV**. Intentamos hacer el clonado al asignar **miTV** a **tuTV**. Para observar el comportamiento, mandamos a mostrar la información de ambos objetos. En este momento lucen iguales, ya que no hemos modificado nada en **tuTV**. Ahora debemos encargarnos de cambiar el objeto **tuTV** colocando el valor de **1300.80** en su costo.

Podemos pensar que **miTV** tiene el valor de default de **400.0** en costo y **tuTV** el de **1300.80**. Comprobemos que esto es cierto al mandar a mostrar la información de ambos objetos. Nos encontraremos con que los dos tienen el mismo valor actualizado, tal como se muestra en la **Figura 9**.



Esto nos demuestra que, con una simple asignación, no llevamos a cabo el clonado de los objetos. Lo que ha sucedido es que se crea una instancia en memoria y estamos usando dos variables de referencia; al tener una asignación de este tipo, ambas variables de referencia apuntan al mismo objeto. El resultado no es una clonación.

Ahora vamos a crear una clase que sea clonable. Generamos una clase para radios que implemente **ICloneable** y, también, **IProducto**. Luego, intentaremos realizar la clonación.

La interfaz **ICloneable** se define de la siguiente manera:

```
public interface ICloneable
{
    object Clone();
}
```

La interfaz solo define un método **Clone()**. Dentro de él, colocamos el código necesario para clonar el objeto, y obtenemos un **object**.

Agreguemos una nueva clase llamada **CRadio**, en cuyo interior colocamos el código que vemos a continuación:

```
// La clase implementa dos interfaces
class CRadio : IProducto, ICloneable
{
    double costo;
    double impuesto;
    string descripcion;

    // Metodos propios
    // Constructor
    public CRadio()
    {
        // Inicializamos los datos
        costo = 150.85;
        descripcion = "Radio de onda corta";
    }
    public void ColocarCosto(double nCosto)
    {
        // Asignamos el nuevo costo
        costo = nCosto;
    }
    // Metodos a implementar debido a la interfaz IProducto
    public void CalculaImpuesto(double imp)
    {
        // Calculamos el impuesto
        impuesto = costo * imp;
    }
    public void MuestraInformacion()
    {
        // Mostramos la informacion necesaria
        Console.WriteLine("--- Producto radio");
        Console.WriteLine(descripcion);
        Console.WriteLine("Costo {0}, Impuesto {1}", costo, impuesto);
    }
    // Metodos a implementar debido a la interfaz ICloneable
    public object Clone()
    {
        CRadio temp = new CRadio();
    }
}
```

```
        temp.costo = costo;  
        temp.descripcion = descripcion;  
        temp.impuesto = impuesto;  
        return temp;  
    }  
}
```

Observemos que, en esta ocasión, la clase está implementando dos interfaces. Cuando esto sucede, las listamos separándolas con comas.

A continuación, definimos los datos propios de la clase. En el constructor inicializamos **costo** y **descripcion**. Luego, simplemente colocamos los métodos propios de la clase y los que debemos implementar de **IProducto** que ya conocemos.

Por último, implementamos el método **Clone()**, en el que creamos una nueva instancia de **CRadio**. Después, copiamos el estado del objeto a la nueva instancia para regresar el nuevo objeto. Tengamos en cuenta que este será una instancia diferente, pero tendrá un estado exactamente igual a nuestro objeto, con lo que producimos el clonado.

Adicionamos código a la función **Main()** para llevar a cabo el clonado:

```
class Program  
{  
    static void Main(string[] args)  
    {  
        // Ejemplo con objetos que no se han clonado  
        CTelevisor miTV = new CTelevisor();  
        CTelevisor tuTV = miTV;  
        // Aquí pensamos que hicimos el clonado  
        // Vemos los dos objetos  
        Console.WriteLine("Objetos sin ningun cambio \n");  
        miTV.MuestraInformacion();  
        tuTV.MuestraInformacion();  
        // Modificamos un objeto  
        tuTV.ColocarCosto(1300.80);  
        // Vemos los dos objetos  
        Console.WriteLine("\nObjetos despues del cambio \n");  
        miTV.MuestraInformacion();  
    }  
}
```



```
tuTV.MuestraInformacion();  
// Lo mismo pero con clases que implementan a ICloneable  
CRadio miRadio = new CRadio();  
// Clonamos el objeto  
CRadio tuRadio = (CRadio)miRadio.Clone();  
// Vemos los dos objetos  
Console.WriteLine("\n --- Radios ---");  
Console.WriteLine("\nObjetos sin ningun cambio \n");  
miRadio.MuestraInformacion();  
tuRadio.MuestraInformacion();  
// Modificamos un objeto  
tuRadio.ColocarCosto(800.50);  
// Vemos los dos objetos  
Console.WriteLine("\nObjetos despues del cambio \n");  
miRadio.MuestraInformacion();  
tuRadio.MuestraInformacion();  
}  
}
```

Creamos la instancia de radio denominada **miRadio**. Luego, usamos otra variable de referencia llamada **tuRadio** y procedemos a clonar el objeto utilizando el método **Clone()**. Cabe notar que tenemos que hacer un **type cast** de tipo **CRadio**; no hay que olvidar que **Clone()** regresa un **object** y, por eso, es necesario hacerlo.

Luego de realizar las tareas mencionadas con anterioridad, procedemos de manera similar al caso anterior, mostrando primero los objetos, cambiando uno y mostrándolos otra vez. En este caso, observamos que, en efecto, tenemos dos instancias diferentes y el clonado por fin se ha llevado a cabo.



EL ESTADO DE UN OBJETO



Es interesante tener en cuenta que se conoce como estado de un objeto a los valores que tienen todos sus datos en un momento particular. De esta forma, cuando copiamos el estado de un objeto a otro, debemos copiar todos los datos que existen en él. Por supuesto, esto incluye, también, datos que sean privados, ya que pueden contener información que sea importante.

```
C:\Windows\system32\cmd.exe
Costo 1300.8, Impuesto 0
--- Producto televisor
Televisor pantalla plana
Costo 1300.8, Impuesto 0

--- Radios ---

Objetos sin ningun cambio

--- Producto radio
Radio de onda corta
Costo 150.8$, Impuesto 0
--- Producto radio
Radio de onda corta
Costo 150.8$, Impuesto 0

Objetos despues del cambio

--- Producto radio
Radio de onda corta
Costo 150.8$, Impuesto 0
--- Producto radio
Radio de onda corta
Costo 300.5, Impuesto 0
Press any key to continue . . . . .
```

► **Figura 10.** Comprobamos que, en el segundo caso, la clonación sí se ha realizado y, en efecto, tenemos dos instancias.

El uso de IComparable

Cuando realizamos un programa, es común tener lógica que lleva a cabo comparaciones, porque nos interesa saber si un valor es igual, mayor o menor que otro. Por lo general, efectuamos esta clase de comparaciones con valores numéricos de tipo entero, flotante o doble. Esto es fácil, pues resulta obvio comparar cantidades. En otras ocasiones, llegamos a comparar cadenas. Lo que muchas veces ignoramos es que es posible hacer comparaciones de todo tipo de clases. Esto nos permitirá codificar más fácilmente nuestros programas y aprovechar algunas herramientas que ya ofrece el framework de .NET.

Es simple comparar enteros, pero ¿cómo podemos comparar objetos, como televisores, autos, etcétera? La respuesta es por medio de la implementación de la interfaz **IComparable**.

Un aspecto que debemos entender es que nosotros somos, como programadores, los responsables de indicar las reglas de comparación. Imaginemos que precisamos comparar televisores. Dependiendo de las necesidades del programa, es posible que el factor importante para comparar sea las dimensiones de la

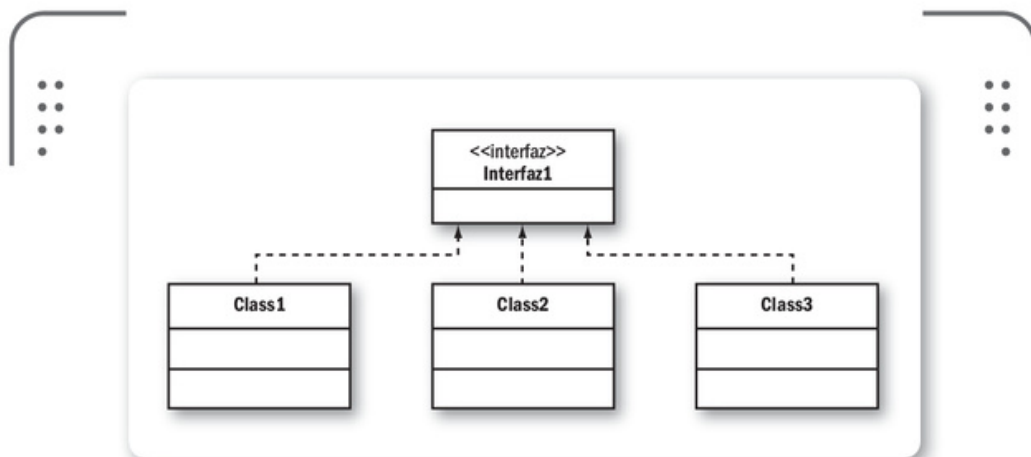
COMO
PROGRAMADORES
SOMOS LOS
RESPONSABLES DE
INDICAR LAS REGLAS
DE COMPARACIÓN

pantalla, el costo o, incluso, el modelo del producto. A veces, realizar las comparaciones es una de las tareas más complicadas, y requiere tener reglas más complejas, por lo que debemos ser cuidadosos.

Antes de implementar **Comparable**, es importante conocer a fondo cómo llevaremos a cabo estas comparaciones. La interfaz **Comparable** está definida de la siguiente manera:

```
public interface Comparable
{
    int CompareTo(object o);
}
```

La interfaz define un método, llamado **CompareTo()**, que es el encargado de hacer la comparación. Para entender este método, debemos saber que lo ejecuta la instancia que deseamos comparar, y que el objeto con el cual la vamos a comparar es **0**. El método regresará un valor entero, que nos indica si la instancia y el objeto son iguales, o si alguno de ellos es mayor que el otro.



► **Figura 11.** Aquí vemos el diagrama de clases que indica cómo las interfaces pueden servirnos para el polimorfismo.

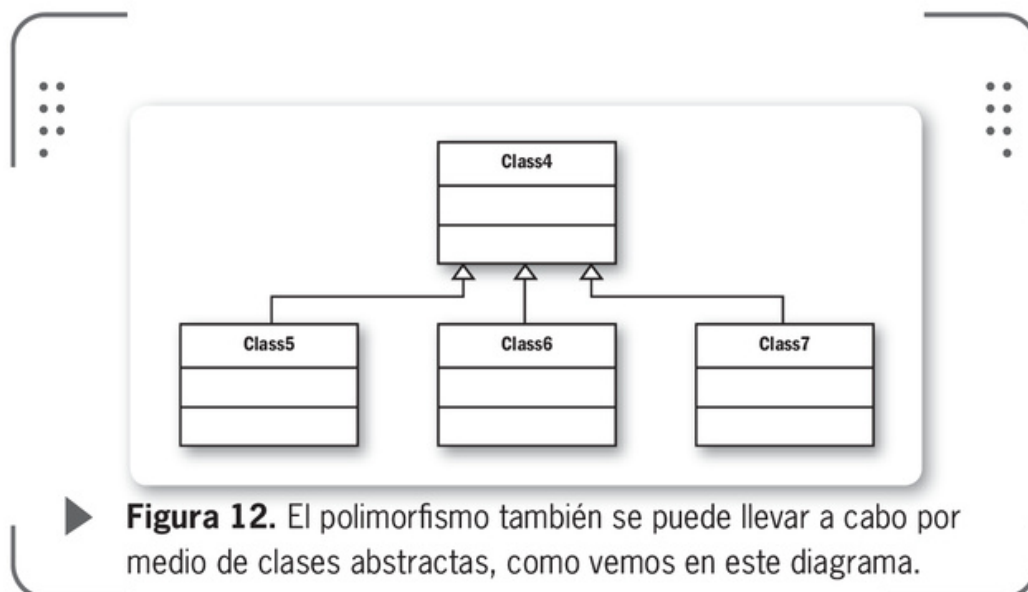
Hay reglas que deben respetarse al dar el valor de retorno de **CompareTo()**, ya que varios algoritmos dependen de estos valores preacordados para funcionar correctamente. Si los dos objetos son iguales, el valor por regresar tiene que ser **0**. En caso de que la instancia sea menor o deba venir anteriormente al objeto por comparar, se debe

regresar un valor menor que **0**, generalmente, **-1**. Cuando la instancia sea mayor o deba estar posteriormente al objeto por comparar, se debe regresar un valor mayor que **0**, generalmente, **1**.

El parámetro que recibimos es el objeto que vamos a comparar con la instancia. Cabe notar que su tipo es **object**; esto significa que, internamente, debemos hacer un type cast a la clase, para poder usarlo y realizar las comparaciones necesarias.

Ahora crearemos una aplicación que haga uso de **IComparable**. Tendremos un arreglo de televisores y, posteriormente, los ordenaremos. Para demostrar cómo una clase que implementa **IComparable** puede aprovechar algunas funcionalidades que nos ofrece el framework de .NET, usaremos, para ordenar, el método **Sort()** de la clase **Array**.

Este método nos permite ordenar un arreglo monodimensional. Simplemente, debemos pasar el arreglo que queremos ordenar como parámetro. Este método tiene 16 sobrecargas, de modo que es conveniente acudir a la documentación para conocerlo mejor.



Cuando trabajamos con **Sort()** es preciso tener cuidado, ya que este método no regresa una copia del arreglo ordenado, sino que trabaja directamente sobre el arreglo original. Si por alguna razón necesitamos mantener también el arreglo desordenado, entonces lo mejor será hacer una copia de él antes de utilizar **Sort()**.

Empecemos por realizar la creación de una nueva aplicación y colocar el código que ya conocemos de la interfaz **IProducto**:

```
// Esta interfaz define el comportamiento para el producto
interface IProducto
{
    // Metodos que componen al comportamiento del producto
    // Calcular el impuesto
    void CalculaImpuesto(double imp);
    // Mostrar información
    void MuestraInformacion();
}
```

Ahora ingresamos el código que corresponde a la clase **CTelevisor**:

```
// La clase implementa dos interfaces
class CTelevisor : IProducto, IComparable
{
    private double costo;
    private double impuesto;
    private string descripcion;
    // Metodos propios
    // Constructor
    public CTelevisor(string desc, double pcosto)
    {
        // Inicializamos los datos
        costo = pcosto;
        descripcion = desc;
    }

    public void ColocarCosto(double nCosto)
    {
```



LA CLASE ARRAY



El framework de .NET nos provee de la clase conocida como **Array**, la cual se encarga de brindarnos las funcionalidades necesarias para trabajar con arreglos. No solo nos permite guardar la información, sino que también nos ofrece métodos para manipularla y ordenarla. Esta clase incluye herramientas adicionales que conviene conocer para nuestro beneficio.

```
// Asignamos el nuevo costo
costo = nCosto;
}

// Metodos a implementar debido a la interfaz IProducto
public void CalculaImpuesto(double imp)

{
    // Calculamos el impuesto
    impuesto = costo * imp;
}

public void MuestraInformacion()
{
    // Mostramos la informacion necesaria
    Console.WriteLine("--- Producto televisor");
    Console.WriteLine(descripcion);
    Console.WriteLine("Costo {0}, Impuesto {1}", costo, impuesto);
}

// Propiedad para el costo, la necesitamos para poder hacer la comparacion
public double Costo
{
    get { return costo; }
}

// Metodo de la interfaz IComparable
// Este método lleva a cabo la comparacion
// Recibe el objeto que vamos a comparar con la instancia ejecutada
// Si el valor de retorno es 0 los objetos son iguales
// Si el retorno es <0 la instancia esta antes que el objeto a comparar
// Si el retorno es >0 la instancia esta despues que el objeto a comparar
int IComparable.CompareTo(object o)
{
    //Vamos a tomar como elemento de comparacion el costo
    // pero podria ser cualquier dato o combinacion de datos
    // Colocamos el objeto como televisor haciendo un type cast
    CTelevisor temp = (CTelevisor)o;

    // Hacemos las comparaciones
```



```
        if (costo > temp.Costo)
            return 1;
        if (costo < temp.Costo)
            return -1;
        // Son iguales
        return 0;
    }
}
```

La clase **CTelevisor** implementará dos interfaces. **IProducto** ya es conocida, y la implementación de los métodos es sencilla. Donde debemos tener más cuidado es con **IComparable**. Hemos visto que, al usarla, es necesario definir las reglas que se van a aplicar para comparar los objetos de esta clase. Para hacer el ejemplo sencillo, usaremos el costo como elemento de comparación, pero bien podría haber sido cualquier otro dato de **CTelevisor** o, incluso, varios de ellos.



► **Figura 13.** Este es el sitio web de Microsoft donde podemos descargar la versión Express de Visual Studio.

En la implementación del método **Compare** de **IComparable** colocamos la lógica de comparación. Recibe como parámetro un objeto de tipo **object**, que necesita un type cast para ser usado como instancia de **CTelevisor**. Creamos un objeto que llamamos **temp**. Como ya tenemos la instancia y el objeto, procedemos a efectuar la comparación.

Empezamos verificando si el costo de la instancia es mayor que el costo del objeto recibido por parámetro. En caso de que esto se

cumpla, regresamos el valor **1**. Si no se cumple, verificamos si el costo de la instancia es menor que el costo del objeto que ha sido recibido por parámetro. Si esta condición es verdadera, el método regresa el valor **-1**. La única opción restante es que los dos costos sean iguales; entonces, ya no necesitamos la comparación y regresamos el valor **0**.

Es importante tener en cuenta que debemos colocar el código de la aplicación que corresponde a la función **Main()**:

```
class Program
{
    static void Main(string[] args)
    {
        // Creamos un arreglo de televisores
        CTelevisor[] misTvs = new CTelevisor[3];
        // Colocamos las instancias en el arreglo
        misTvs[0] = new CTelevisor("Televisor de Plasma", 1500.50);
        misTvs[1] = new CTelevisor("Televisor LCD", 2300.45);
        misTvs[2] = new CTelevisor("Televisor pantalla plana", 575.85);

        Console.WriteLine("--- Antes de ordenar ---\n");
        // Vemos nuestro arreglo
        for(int n=0; n<misTvs.Length; n++)
            misTvs[n].MuestraInformacion();
        // Ordenamos el arreglo aprovechando la implementacion de IComparable
        Array.Sort(misTvs);
        Console.WriteLine("\n--- Despues de ordenar ---\n");
        // Vemos nuestro arreglo
        for (int n = 0; n < misTvs.Length; n++)
            misTvs[n].MuestraInformacion();
    }
}
```

La función empieza declarando un arreglo de tipo **CTelevisor** con un tamaño de **3**. Podemos tener tres instancias de **CTelevisor** para, luego, organizarlas. Posteriormente, creamos las instancias y las colocamos en el arreglo. Cada una de ellas es diferente, y tiene descripciones y costos variados. En este momento, observemos el orden en el que las definimos, para, luego, poder comparar con el arreglo ordenado.

implementation of IComparable to enable object instances to be ordered or sorted.

Examples

The following example illustrates the implementation of IComparable and the requisite CompareTo method.

```

VB  C#  C++  F#  JScript

using System;
using System.Collections;

public class Temperature : IComparable
{
    // The temperature value
    protected double temperatureF;

    public int CompareTo(object obj) {
        if (obj == null) return 1;

        Temperature otherTemperature = obj as Temperature;
        if (otherTemperature != null)
            return this.temperatureF.CompareTo(otherTemperature.temperatureF);
        else
            throw new ArgumentException("Object is not a Temperature");
    }

    public double Fahrenheit
    {
        get
        {
            return this.temperatureF;
        }
    }
}

```

► **Figura 14.** En el sitio web de MSDN podemos encontrar ejemplos de código sobre el uso de la interfaz **IComparable**.

Para ayudarnos en la comparación, mostramos la información guardada en el arreglo. Esto se logra con **for**. Aprovechamos el método **Sort** de **Array**, que es estático y podemos invocarlo sin tener que generar una instancia de la clase **Array**. El método necesita un arreglo por ordenar. Como el arreglo es de una clase que implementa **IComparable**, es posible para **Sort** llevar a cabo el ordenamiento.

El método **Sort** ordena el arreglo, por lo que mandamos a imprimir otra vez para comprobar que el ordenamiento se realizó. Ahora el arreglo está ordenado con relación al costo, que es el dato usado en el método **CompareTo()**. Ejecutamos el código y vemos los resultados.

```

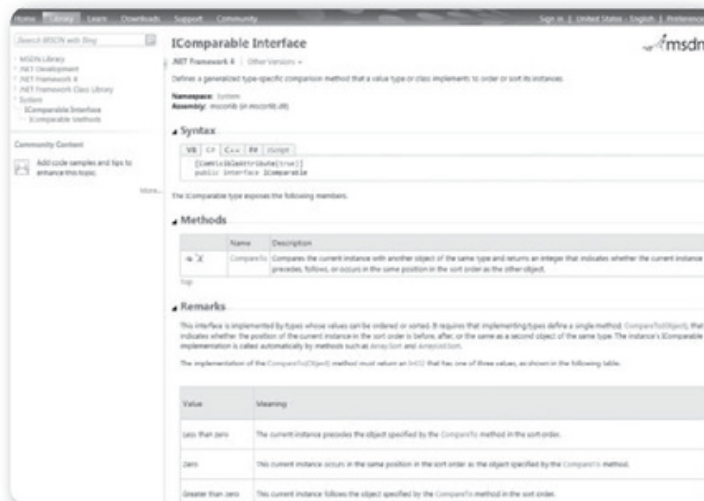
C:\Windows\system32\cmd.exe

--- Antes de ordenar ---
--- Producto televisor
Televisor de Plasma
Costo 1500.5, Impuesto 0
--- Producto televisor
Televisor LCD
Costo 2300.45, Impuesto 0
--- Producto televisor
Televisor pantalla plana
Costo 575.85, Impuesto 0
--- Despues de ordenar ---
--- Producto televisor
Televisor pantalla plana
Costo 575.85, Impuesto 0
--- Producto televisor
Televisor de Plasma
Costo 1500.5, Impuesto 0
--- Producto televisor
Televisor LCD
Costo 2300.45, Impuesto 0
Presione una tecla para c

```

► **Figura 15.** En esta figura podemos observar que el arreglo ha sido ordenado satisfactoriamente.

Con esto terminamos de explicar los conceptos más importantes sobre las interfaces, pero aún quedan temas relacionados con ellas que son sumamente importantes y nos permiten llevar a cabo procesos avanzados en el desarrollo de nuestros programas.



► **Figura 16.** Encontramos toda la documentación necesaria sobre las interfaces que hemos aprendido, en el sitio de MSDN.



RESUMEN

Las interfaces nos permiten definir comportamientos, que son representados por métodos abstractos. Las clases que implementan a las interfaces deben proveer las implementaciones para estos métodos. Una clase puede implementar más de una interfaz. Podemos usar las interfaces para aprovechar al máximo el poder que nos brinda el framework de .NET. Dentro de .NET, las interfaces **IComparable**, **IEnumerable** e **ICloneable** son importantes.

Actividades

TEST DE AUTOEVALUACIÓN

- 1 ¿Qué es una interfaz?
- 2 ¿Cómo definimos los comportamientos de una interfaz?
- 3 ¿De qué forma se indica que una clase implementa la interfaz?
- 4 ¿Es posible tener más de una interfaz implementada en la clase?
- 5 ¿Qué es un método abstracto?
- 6 ¿Para qué podemos utilizar IComparable?
- 7 ¿Cuáles son los valores de retorno por utilizar en la comparación?
- 8 ¿Para qué utilizamos IEnumerable?
- 9 ¿Cómo funciona el método Sort()?
- 10 ¿Para qué se utiliza ICloneable?
- 11 ¿Cuál es la ventaja de tener objetos clonables?
- 12 ¿Con cuántos datos se puede llevar a cabo una comparación?

EJERCICIOS PRÁCTICOS

- 1 Modifique la comparación para llevarla a cabo con otro dato.
- 2 Modifique la comparación para que se tomen dos datos en ella.
- 3 Haga una clase que sea clonable y comparable.
- 4 Cree una aplicación con su propia interfaz que defina un comportamiento.
- 5 Cree una clase que implemente varias interfaces definidas por usted.

Callback, delegados y eventos

Cuando trabajamos en un lenguaje orientado a objetos, los diferentes objetos se comunican entre sí por medio de mensajes. Por ejemplo, creamos la instancia de una clase e invocamos sus métodos, la instancia recibe el mensaje y lleva a cabo una acción. Sin embargo, también debería de ser posible que la instancia pudiera comunicarse con el objeto que la invocó. Este tipo de comunicación de dos vías puede efectuarse por medio de diferentes mecanismos, como las interfaces de callback, los delegados y los eventos.

▼ Conocer las interfaces de callback 50	▼ Los delegados 68
Creación de la interfaz de callback..... 50	Ejemplo con delegado..... 69
Entender el sink..... 51	Delegados con multicasting..... 78
Uso de más de un sink por evento..... 58	▼ Entender los eventos 84
Cómo utilizar dos o más eventos 61	▼ Resumen 89
	▼ Actividades 90



Conocer las interfaces de callback

Hemos aprendido sobre el uso de las interfaces en el capítulo anterior, ahora conoceremos otro de sus usos. Las interfaces pueden servirnos para definir comportamientos callback. Estos comportamientos son conocidos como mecanismos de callback y podemos usarlos para generar comunicación de dos vías entre objetos.

El uso de esta clase de mecanismos no es complicado, pero se entiende mejor al verlo en ejemplos. En este capítulo nos enfocaremos en varios casos que nos permitirán entender cómo utilizarlos.

Pensemos que debemos desarrollar un software para controlar una caldera. Esta estará representada por una clase, y nosotros podemos poner a trabajar esa caldera invocando métodos de esa clase desde **Main()**. Esto no resulta novedoso, pero la caldera puede tener problemas, en cuyo caso sería necesario que el objeto **caldera** también pudiera comunicar con **Main()**. Un problema podría ser la falta de combustible o que, simplemente, la caldera tuviera una temperatura mayor a la máxima permitida por seguridad. Resolveremos este inconveniente a través de diferentes técnicas.

Creación de la interfaz de callback

Lo primero que tenemos que hacer es crear una nueva aplicación de consola en C# y adicionar una interfaz llamada **IEventosCaldera**.

Esta será nuestra interfaz callback, y los comportamientos estarán relacionados con los eventos que pueden suceder con la caldera. La interfaz se crea de la manera habitual:



INTERFACES EN EL AMBIENTE GRÁFICO



Es necesario tener en cuenta que aunque muchos de los ejemplos que se presentan en este libro se muestran en consola por facilidad y rapidez, es posible y deseable usar estas técnicas también en el desarrollo de aplicaciones gráficas. Recordemos que no hay que confundir la interfaz como elemento de programación y el concepto de interfaz gráfica para usuario.

```
// Esta interfaz tiene los eventos propios de la caldera
interface IEventosCaldera
{
    // Este evento se invoca cuando queda poco combustible
    // Envia como parametro la cantidad de combustible en la caldera
    void EPocoCombustible(float c);
    // Aqui podemos adicionar mas eventos
}
```

Como es nuestro primer ejemplo, solamente definiremos un evento, el relacionado con la poca cantidad de combustible. Una parte importante que debemos notar en este punto es el tipo y los parámetros del evento. Su tipo es **void** y recibe un único parámetro que es un flotante. La razón de la importancia de esto reside en que, cuando se genere el evento, únicamente podrán invocarse métodos en el otro objeto que tengan estas mismas características; es decir, que implementen estos métodos de la interfaz.

Entender el sink

Con las interfaces de callback necesitamos un objeto que nos ayude, al que se conoce como **sink**. La clase del objeto **sink** tendrá los métodos que serán invocados cuando la caldera tenga su evento.

La clase para el objeto **sink** se programa del modo tradicional, pero debe implementar la interfaz de callback. Estas implementaciones son los **handlers**. Lo que deseamos que haga el handler cuando se genere el evento de la falta de combustible consiste en un mensaje de advertencia indicando la cantidad de litros disponibles.

Nuestra clase **sink**, entonces, queda de la siguiente manera:



LOS HANDLERS



Un **handler** no es otra cosa más que un método que se invoca cuando sucede algún evento. Estos eventos y la invocación al handler forman parte de un mecanismo de callback. En el handler ponemos el código que deseamos que se ejecute cuando ocurre el evento. Muchas veces, Visual Studio coloca handlers para nosotros de manera automática cuando diseñamos formas.

```
// En esta clase estan los handlers a los eventos producidos por la caldera
// Implementa a IEventosCaldera
public class CEventosCalderaSink : IEventosCaldera
{
    // El codigo del handler para EPocoCombustible
    public void EPocoCombustible(float c)
    {
        // Mandamos el mensaje de que queda poco combustible
        // Aqui podemos adicionar más logica que se lleve a cabo cuando suceda el
        // evento
        Console.WriteLine("----");
        Console.WriteLine("----->Queda poco combustible, {0} litros", c);
        Console.WriteLine("----");
    }
}
```

En este ejemplo solo hemos colocado un handler, que es **EPocoCombustible()**. Dentro de él podemos poner cualquier código necesario para llevar a cabo la lógica del programa. En este caso, simplemente se manda un mensaje en el que se indica que queda poco combustible y la cantidad de litros remanentes. Luego, adicionaremos más handlers en esta clase.

Ahora ya podemos proceder a programar la clase que corresponde a la caldera. Creamos una nueva clase y le llamamos **CCaldera**. Allí tenemos


```
// Esta clase representa a la caldera
class CCaldera
{
    // Datos necesarios
    private float combustible;
    private float temperatura;
    // Este arreglo contiene los sinks de la caldera
    ArrayList misSinks = new ArrayList();
    // Constructor
    public CCaldera(float combInicial)
    {
        // Asignamos el combustible inicial
        combustible = combInicial;
        // Inicializamos la temperatura
        temperatura = 10.0f;
    }
    public float Combustible
    {
        get{return combustible;}
    }
    // Este metodo indica el trabajo de la caldera
    public void Trabajo()
    {
        // Reducimos el combustible
        combustible -= 0.3f;
        // Incrementamos la temperatura
        temperatura += 0.8f;
        // Mostramos la informacion
        Console.WriteLine("Combustible={0}, Temperatura={1}", combustible,
            temperatura);
        // Verificamos si existe la condicion necesaria para hacer una llamada
        // callback y generar el evento
        if (combustible < 5.0f)
        {
            // Recorremos todos los sinks relacionados con la caldera
            // enviandole el mensaje a cada uno de ellos para que actue segun su
            logica
            foreach (IEventosCaldera evento in misSinks)
```

```
    {  
        // Aqui hacemos la llamada callback  
        evento.EPocoCombustible(combustible);  
    }  
}  
  
// Los siguientes metodos proveen la funcionalidad para la administracion de  
// los sink  
// Adicionamos un sink  
public void AdicionaSink(IEventosCaldera sink)  
{  
    // Adicionamos el sink al ArrayList  
    misSinks.Add(sink);  
}  
  
// Eliminamos un sink  
public void BorrarSink(IEventosCaldera sink)  
{  
    // Eliminamos del ArrayList al sink  
    misSinks.Remove(sink);  
}  
}
```

La clase necesita de dos datos propios: uno para guardar la cantidad de combustible y otro para la temperatura. Tenemos un **ArrayList** al cual llamamos **misSinks**. Dentro de este arreglo, guardaremos los sinks que serán invocados como handlers cuando el evento ocurra. Usamos un **ArrayList** para poder tener múltiples handlers.



EL ARRAYLIST



ArrayList es una clase que nos brinda toda la funcionalidad necesaria para crear y administrar una lista dinámica. Es importante señalar que en ella podemos acceder a realizar el aumento de manera dinámica; de esta forma, la cantidad de elementos por guardar será mayor y, por otro lado, podremos usarla tan fácilmente como si se tratara de un arreglo. También debemos tener en cuenta que en su interior, cada uno de los elementos que corresponden se guardan como **object**, por lo que los desarrolladores no deben olvidar los **type cast** que son requeridos.

El constructor es muy sencillo y, simplemente, asigna al dato **combustible** la cantidad que pasamos por el parámetro. La caldera siempre iniciará con una temperatura de **10** grados.

Para apoyar nuestra lógica y facilitar el hecho de mostrar la información, se ha creado una propiedad llamada **Combustible**, que nos permite obtener el estado de este en cualquier momento. En este caso, solo necesitamos la propiedad de salida.

El método **Trabajo()** es el más interesante, ya que aquí se lleva a cabo la lógica principal de la clase y la generación del evento cuando sea necesario.

Lo primero que realiza este método es reducir el combustible, luego incrementa la temperatura y muestra en pantalla la información correspondiente. Ya que realizamos el trabajo propio de la caldera, colocamos la lógica relacionada con la generación del evento. En nuestro caso, hemos decidido que si el combustible es menor a **5**, entonces se debe generar el evento. Al generar el evento, recorreremos con la ayuda de un **foreach** todos los handlers guardados en el interior del **ArrayList misSinks**. A cada uno de ellos le invocamos el método **EPocoCombustible()** pasando como parámetro la cantidad de combustible actual. De esta forma, se invoca el método que se encuentre en la clase **sink**.

La lógica para generar el evento no necesariamente tiene que ir al final del método; debe usarse en el lugar adecuado según lo requiera la lógica propia de la aplicación. Las condiciones en las cuales sucede también dependen de cada aplicación, y hay que seleccionarlas adecuadamente.

EL SINK ES UN OBJETO
DE AYUDA NECESARIO
CUANDO USAMOS
LAS INTERFACES DE
CALLBACK



LOS DELEGADOS



Debemos tener en cuenta que para las personas que vienen de realizar desarrollo con el lenguaje de programación C++, resulta curioso que en C# podamos usar los delegados para apuntar a métodos que sean estáticos o métodos de instancias. Es importante experimentar también con esto. Algunas veces, el hecho de entender a los delegados requiere cierta experimentación. La mejor forma de hacer esto es crear un programa con cada una de las técnicas presentadas en este capítulo, para así poder comparar y contrastar las diferentes tecnologías involucradas en el proceso.

Al final, vamos a tener un par de métodos que nos permiten administrar a los sink. **AdicionaSink()** recibe como parámetro un objeto de alguna clase que implemente **IEventosCaldera**. El objeto, entonces, es adicionado al **ArrayList misSinks**.

Si deseamos que algún sink ya no funcione como handler cuando el evento ocurra, entonces hacemos uso de **BorrarSink()**; del mismo modo, pasamos como parámetro al objeto que deseamos eliminar de **misSinks**.

Con todo esto, ya podemos probar la aplicación, colocando el siguiente código en la función **Main()**:

```
static void Main(string[] args)
{
    // Creamos la instancia de la caldera
    CCaldera miCaldera = new CCaldera(50.0f);
    // Creamos el sink
    CEventosCalderaSink sinkNormal = new CEventosCalderaSink();
    // Le indicamos a la caldera que se va a comunicar con el sink
    miCaldera.AdicionaSink(sinkNormal);
    // Hacemos trabajar la caldera
    while (miCaldera.Combustible > 0)
    {
        miCaldera.Trabajo();
    }
}
```

Lo primero que hacemos es crear una instancia de la clase **CCaldera** llamada **miCaldera**, la cual tendrá un combustible inicial de **50**. Luego, es necesario crear un objeto de la clase **CEventosCalderaSink** al que llamamos **sinkNormal**. Este tiene los métodos que serán invocados cuando el evento



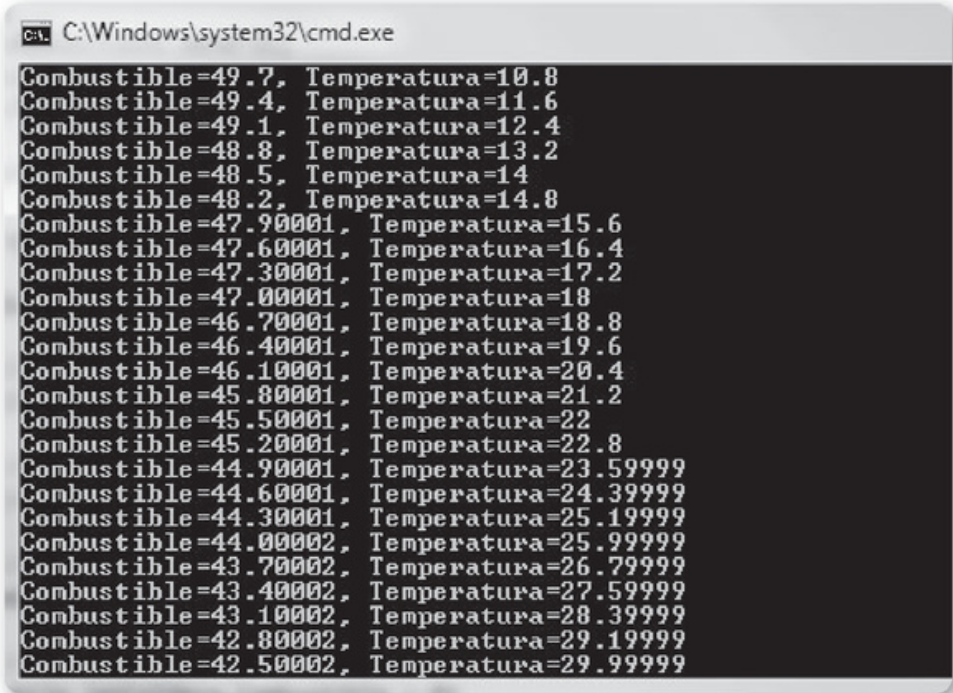
UNA CLASE PARA LOS DELEGADOS



Es importante tener en cuenta que cuando nos encargamos de realizar la creación de un delegado, el compilador de C# en realidad crea por nosotros una clase sellada que se deriva de **System.MulticastDelegate**. Esta no se ve en nuestro código, pero en ella tenemos las funcionalidades necesarias para usarlo, lo cual es un beneficio que nos brinda Microsoft Visual Studio.

ocurra. Cabe notar que estos métodos no están en la caldera y son invocados por un mecanismo de callback.

Ya que tenemos estos objetos, es preciso registrar al sink con la caldera, para que tenga a quien invocar. Simplemente, invocamos el método **AdicionaSink()** pasando como parámetro **sinkNormal**.



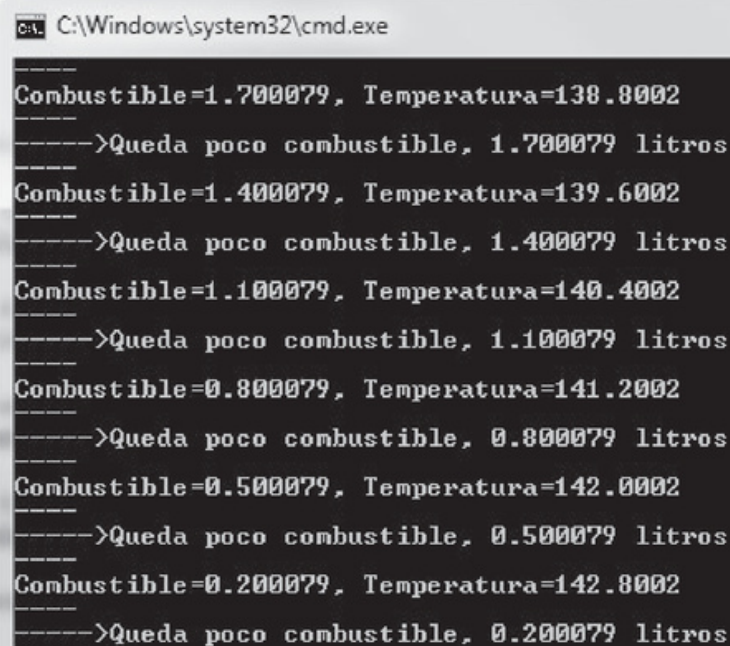
```
C:\Windows\system32\cmd.exe
Combustible=49.7, Temperatura=10.8
Combustible=49.4, Temperatura=11.6
Combustible=49.1, Temperatura=12.4
Combustible=48.8, Temperatura=13.2
Combustible=48.5, Temperatura=14
Combustible=48.2, Temperatura=14.8
Combustible=47.90001, Temperatura=15.6
Combustible=47.60001, Temperatura=16.4
Combustible=47.30001, Temperatura=17.2
Combustible=47.00001, Temperatura=18
Combustible=46.70001, Temperatura=18.8
Combustible=46.40001, Temperatura=19.6
Combustible=46.10001, Temperatura=20.4
Combustible=45.80001, Temperatura=21.2
Combustible=45.50001, Temperatura=22
Combustible=45.20001, Temperatura=22.8
Combustible=44.90001, Temperatura=23.59999
Combustible=44.60001, Temperatura=24.39999
Combustible=44.30001, Temperatura=25.19999
Combustible=44.00002, Temperatura=25.99999
Combustible=43.70002, Temperatura=26.79999
Combustible=43.40002, Temperatura=27.59999
Combustible=43.10002, Temperatura=28.39999
Combustible=42.80002, Temperatura=29.19999
Combustible=42.50002, Temperatura=29.99999
```

► **Figura 1.** En esta figura podemos observar la primera parte de la ejecución; aún no se han generado eventos.

Por último, tenemos un ciclo que se repetirá mientras haya combustible. Tengamos en cuenta que en muchas de las vueltas del ciclo, no se cumplirá la condición necesaria para que se genere el evento, pero en un momento en particular, cuando el combustible sea menor a 5, el evento se empezará a disparar.

A continuación, podemos realizar la compilación y ejecución de la aplicación; de esta forma, veremos qué es lo que sucede.

De esta forma podemos observar la forma como esta funcionando la aplicación, los lugares donde se disparan los eventos y también los mensajes que obtenemos por parte de los handler que se invocan, esto lo verificamos en la siguiente imagen:



```
C:\Windows\system32\cmd.exe

-----
Combustible=1.700079, Temperatura=138.8002
----->Queda poco combustible, 1.700079 litros
-----
Combustible=1.400079, Temperatura=139.6002
----->Queda poco combustible, 1.400079 litros
-----
Combustible=1.100079, Temperatura=140.4002
----->Queda poco combustible, 1.100079 litros
-----
Combustible=0.800079, Temperatura=141.2002
----->Queda poco combustible, 0.800079 litros
-----
Combustible=0.500079, Temperatura=142.0002
----->Queda poco combustible, 0.500079 litros
-----
Combustible=0.200079, Temperatura=142.8002
----->Queda poco combustible, 0.200079 litros
-----
```

► **Figura 2.** En esta parte de la ejecución podemos observar que los eventos se dispararon y el handler ha sido invocado.

Uso de más de un sink por evento

Como hemos comentado anteriormente, un evento puede invocar a más de un handler. Para demostrarlo, llevaremos a cabo una serie de adiciones y modificaciones al programa. Aún seguiremos teniendo un único evento, pero ahora usaremos dos clases de tipo **sink**, cada una de las cuales implementará el handler a su propio estilo. Con esto, veremos que más de un código puede ejecutarse cuando sucede el evento.

Adicionar más sinks y handlers es sencillo. Lo primero que necesitamos hacer es crear otra clase que implemente la interfaz **IEventosCaldera**. No hay que olvidar realizar el registro de un objeto de esta nueva clase en el objeto de **CCaldera**.

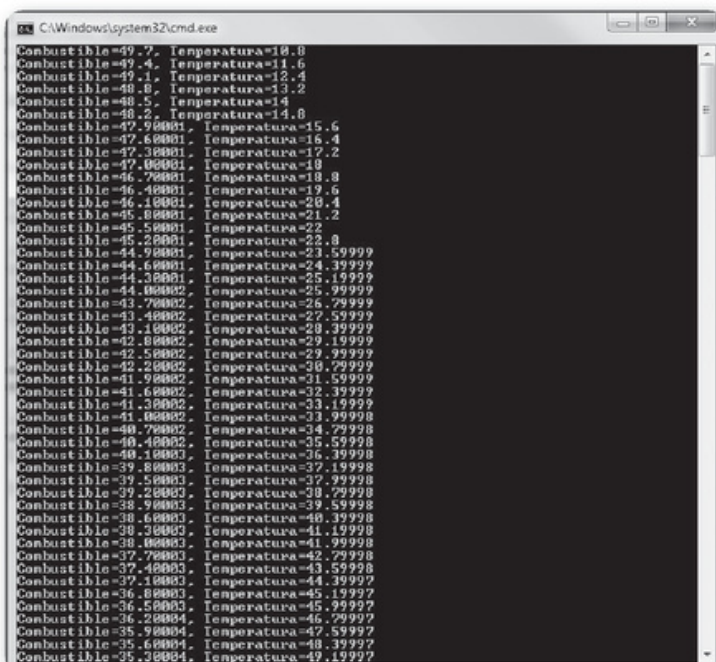
Creamos una nueva clase llamada **CEventoParanoicoSink**, en cuyo interior se implemente el método **EPocoCombustible()**. Este mandará un mensaje de alerta cuando haya poco combustible, pero será diferente del anterior para poder distinguirlo. El código dentro de este handler puede ser cualquier cosa que precisemos para nuestro programa, y no necesariamente debe ser similar el otro handler, como en este ejemplo.

El código de la clase es el que vemos a continuación:

```
// Esta otra clase tambien tiene un handler y procesa su logica de manera un
poco diferente
class CEventoParanoicoSink : IEventosCaldera
{
    // El codigo del handler para EPocoCombustible
    public void EPocoCombustible(float c)
    {
        // Mandamos el mensaje de que queda poco combustible
        // Aquí podemos adicionar más logica que se lleve a cabo cuando suceda el
        evento
        Console.WriteLine("---- PELIGRO ---");
        Console.WriteLine("---- PELIGRO ---");
        Console.WriteLine("---->CASI SIN COMBUSTIBLE, {0} litros", c);
        Console.WriteLine("-----");
    }
}
```

El siguiente cambio que efectuamos es en la función **Main()**:

```
static void Main(string[] args)
{
    // Creamos la instancia de la caldera
    CCaldera miCaldera = new CCaldera(50.0f);
    // Creamos el sink normal
    CEventosCalderaSink sinkNormal = new CEventosCalderaSink();
    // Creamos el sink paranoico
    CEventoParanoicoSink sinkParanoico = new CEventoParanoicoSink();
    // Le indicamos a la caldera que se va a comunicar con los sinks
    miCaldera.AdicionaSink(sinkNormal);
    miCaldera.AdicionaSink(sinkParanoico);
    // Hacemos trabajar a la caldera
    while (miCaldera.Combustible > 0)
    {
        miCaldera.Trabajo();
    }
}
```



```

C:\Windows\system32\cmd.exe
Combustible=49.7, Temperatura=10.8
Combustible=49.4, Temperatura=11.6
Combustible=49.1, Temperatura=12.4
Combustible=48.8, Temperatura=13.2
Combustible=48.5, Temperatura=14
Combustible=48.2, Temperatura=14.8
Combustible=47.90001, Temperatura=15.6
Combustible=47.60001, Temperatura=16.4
Combustible=47.30001, Temperatura=17.2
Combustible=47.00001, Temperatura=18
Combustible=46.70001, Temperatura=18.8
Combustible=46.40001, Temperatura=19.6
Combustible=46.10001, Temperatura=20.4
Combustible=45.80001, Temperatura=21.2
Combustible=45.50001, Temperatura=22
Combustible=45.20001, Temperatura=22.8
Combustible=44.90001, Temperatura=23.59999
Combustible=44.60001, Temperatura=24.39999
Combustible=44.30001, Temperatura=25.19999
Combustible=44.00002, Temperatura=25.99999
Combustible=43.70002, Temperatura=26.79999
Combustible=43.40002, Temperatura=27.59999
Combustible=43.10002, Temperatura=28.39999
Combustible=42.80002, Temperatura=29.19999
Combustible=42.50002, Temperatura=29.99999
Combustible=42.20002, Temperatura=30.79999
Combustible=41.90002, Temperatura=31.59999
Combustible=41.60002, Temperatura=32.39999
Combustible=41.30002, Temperatura=33.19999
Combustible=41.00002, Temperatura=33.99998
Combustible=40.70002, Temperatura=34.79998
Combustible=40.40002, Temperatura=35.59998
Combustible=40.10003, Temperatura=36.39998
Combustible=39.80003, Temperatura=37.19998
Combustible=39.50003, Temperatura=37.99998
Combustible=39.20003, Temperatura=38.79998
Combustible=38.90003, Temperatura=39.59998
Combustible=38.60003, Temperatura=40.39998
Combustible=38.30003, Temperatura=41.19998
Combustible=38.00003, Temperatura=41.99998
Combustible=37.70003, Temperatura=42.79998
Combustible=37.40003, Temperatura=43.59998
Combustible=37.10003, Temperatura=44.39997
Combustible=36.80003, Temperatura=45.19997
Combustible=36.50003, Temperatura=45.99997
Combustible=36.20004, Temperatura=46.79997
Combustible=35.90004, Temperatura=47.59997
Combustible=35.60004, Temperatura=48.39997
Combustible=35.30004, Temperatura=49.19997

```

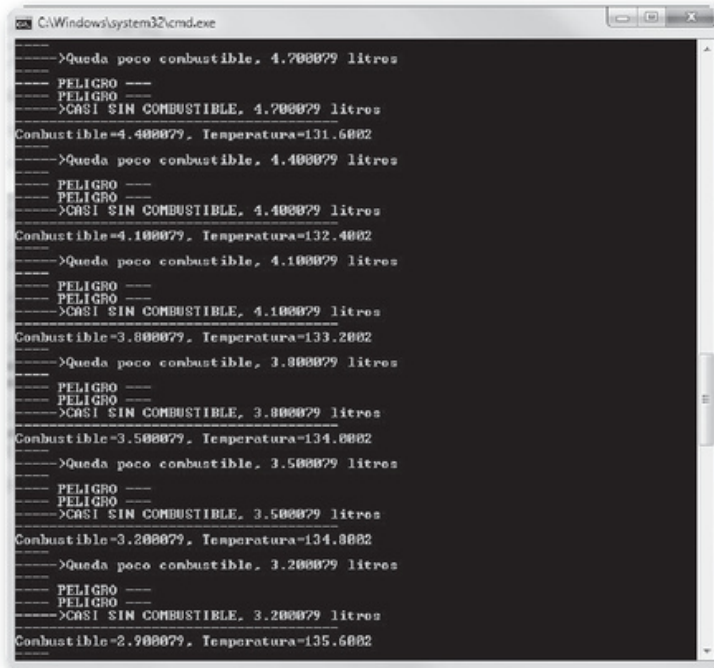
► **Figura 3.** Al inicio de la aplicación, vemos que ninguno de los handlers se ha invocado, como era de esperarse.

Empezamos creando la instancia de **CCaldera**. Luego, creamos las instancias de cada uno de los sinks. Para la nueva clase, el objeto se llama **sinkParanoico**. Ya teniendo las instancias de los sinks, debemos registrarlas en **miCaldera**. Invocamos el método **AdicionaSink()** pasando cada uno de los objetos **sink** como parámetro. Luego, hacemos uso del ciclo **while** para poner a trabajar la caldera. Si todo sale bien, debemos poder observar los dos handlers ejecutándose cuando se cumpla la condición del evento. Ahora podemos compilar y ejecutar la aplicación.



ACTUALIZACIÓN DE INTERFACES

Cada vez que modificamos o actualizamos una interfaz, es necesario que procedamos a realizar la actualización de todas las clases que la implementen, ya que la implementación de la interfaz es total, y de esta forma, no se puede generar la implementación de solamente una parte de ella. De no hacerlo así, el compilador nos mandará mensajes de error.



```
C:\Windows\system32\cmd.exe
>Queda poco combustible, 4.700079 litros
PELIGRO ---
PELIGRO ---
>CASI SIN COMBUSTIBLE, 4.700079 litros
Combustible=4.400079, Temperatura=131.6002
>Queda poco combustible, 4.400079 litros
PELIGRO ---
PELIGRO ---
>CASI SIN COMBUSTIBLE, 4.400079 litros
Combustible=4.100079, Temperatura=132.4002
>Queda poco combustible, 4.100079 litros
PELIGRO ---
PELIGRO ---
>CASI SIN COMBUSTIBLE, 4.100079 litros
Combustible=3.800079, Temperatura=133.2002
>Queda poco combustible, 3.800079 litros
PELIGRO ---
PELIGRO ---
>CASI SIN COMBUSTIBLE, 3.800079 litros
Combustible=3.500079, Temperatura=134.0002
>Queda poco combustible, 3.500079 litros
PELIGRO ---
PELIGRO ---
>CASI SIN COMBUSTIBLE, 3.500079 litros
Combustible=3.200079, Temperatura=134.8002
>Queda poco combustible, 3.200079 litros
PELIGRO ---
PELIGRO ---
>CASI SIN COMBUSTIBLE, 3.200079 litros
Combustible=2.900079, Temperatura=135.6002
```

► **Figura 4.** En cuanto el combustible es menor a 5, ambos handlers son invocados.

Cómo utilizar dos o más eventos

Ya hemos visto que podemos tener más de un handler por evento, pero también es posible que el objeto genere varios eventos. Cada uno de ellos puede producirse por condiciones diferentes y, desde luego, invocar handlers distintos según sea la necesidad de lógica del programa en el cual nos encontramos trabajando.

En nuestro ejemplo, la caldera trabaja también con un dato de temperatura, y podemos producir eventos cuando esta se salga de

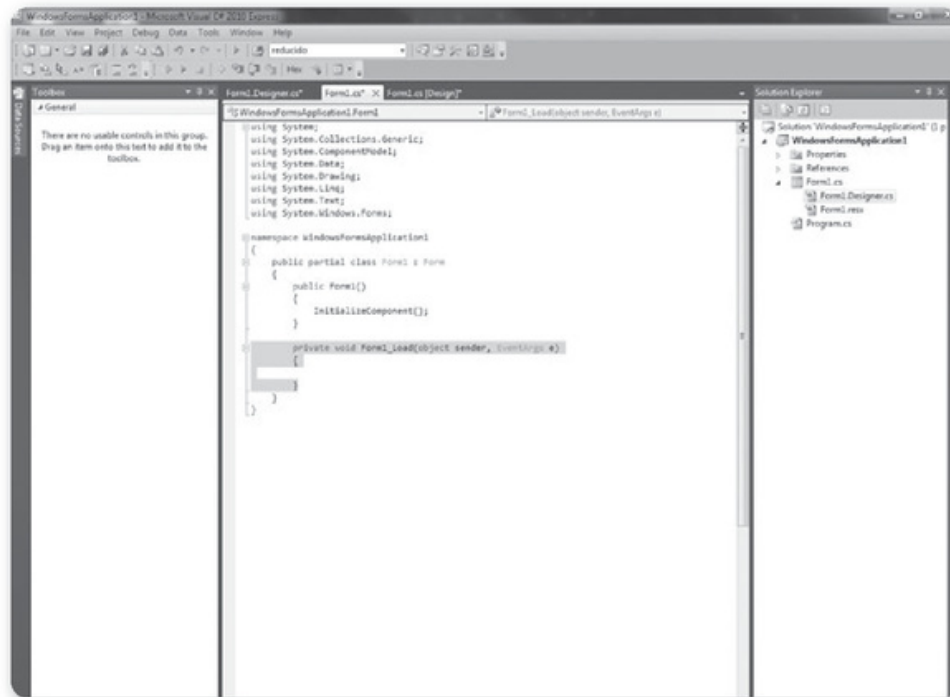


EL USO DE SEALED



Es interesante tener en cuenta que **Sealed** es un modificador poco conocido, que nos permite tener clases selladas. Cuando lo usamos en una clase, es imposible derivar o hacer herencia de esta a otra. Esta clase no puede ser padre de ninguna otra. Podemos usar **sealed** como una forma de proteger una clase a la que no deseamos que se adicionen funcionalidades.

control. Para adicionar más eventos, debemos hacer modificaciones en varias partes del programa. Esto es algo que debemos tomar en cuenta cuando desarrollemos nuestras aplicaciones y hagamos uso de esta técnica. Es mejor planificar los eventos desde la etapa de diseño.



► **Figura 5.** Microsoft Visual Studio también se encarga de crear el esqueleto del handler para el evento que deseamos usar.

Empezamos con la interfaz. Ahora tendremos dos eventos, por lo que tenemos que adicionar un método para el nuevo. En nuestro caso, el nuevo evento está relacionado con la temperatura y lo llamaremos **EAltaTemperatura**. Este método recibirá un parámetro que es la temperatura de la caldera. El código queda de la siguiente manera:

```
// Esta interfaz tiene los eventos propios de la caldera
interface IEventosCaldera
{
    // Este evento se invoca cuando queda poco combustible
    // Envía como parametro la cantidad de combustible en la caldera
```

```
void EPocoCombustible(float c);  
// Este evento se invoca cuando se ha excedido la temperatura de operacion  
// Envia como parametro la temperatura de la caldera  
void EAltaTemperatura(float t);  
}
```

Empezamos actualizando la clase **CEventosCalderaSink**, para la cual adicionamos el método de **EAltaTemperatura()** en el que se mostrará un mensaje indicando que la temperatura es alta y está en grados:

```
// El codigo del handler para EPocoCombustible  
public void EPocoCombustible(float c)  
{  
    // Mandamos el mensaje de que queda poco combustible  
    // Aqui podemos adicionar más logica que se lleve a cabo cuando suceda el  
    evento  
    Console.WriteLine("----");  
    Console.WriteLine("---->Queda poco combustible, {0} litros", c);  
    Console.WriteLine("----");  
}  
// El codigo del handler para EAltaTemperatura  
public void EAltaTemperatura(float t)  
{  
    // Mandamos el mensaje de exceso de temperatura  
    Console.WriteLine("----");  
    Console.WriteLine("---->Temperatura alta, {0} grados", t);  
    Console.WriteLine("----");  
}
```

Luego tenemos que actualizar la clase **CEventoParanoicoSink**. De igual forma, adicionamos el método, pero aprovechamos para colocar una lógica ligeramente más compleja. Dependiendo de la temperatura, variará el mensaje que muestre el handler cuando el evento suceda.

Pero también haremos que la caldera se apague si la temperatura es mayor a **150** grados. El sink no puede apagar la caldera directamente, pero podemos crear una propiedad y aprovecharla para llevar a cabo dicha acción. En esta clase agregamos una variable de tipo **bool** llamada **detener** y creamos una propiedad para ella:

```
private bool detener=false;
public bool Detener
{
    get { return detener; }
}
```

La variable tiene un valor de **false** para indicar que no se debe detener, y se usará el valor de **true** para señalar que es necesario parar la caldera.

En el código que corresponde al método **EAltaTemperatura()**, mandamos a hacer la impresión de los mensajes según el rango de temperatura, pero en el caso de la temperatura mayor a **150**, también colocamos el valor de **true** en el dato **detener**:

```
// El codigo del handler para EAltaTemperatura
public void EAltaTemperatura(float t)
{
    // Mandamos el mensaje segun el caso
    if (t > 150.0f)
    {
        Console.WriteLine("---- PELIGRO ----");
        Console.WriteLine("---- APAGAR LA CALDERA ----");
        Console.WriteLine("-----");
        detener = true;
    }
    else if (t > 130.0f)
    {
        Console.WriteLine("---- PELIGRO ----");
        Console.WriteLine("---- Temperatura muy alta, {0} grados ----", t);
        Console.WriteLine("-----");
    }
    else if (t > 120.0f)
    {
        Console.WriteLine("---- PELIGRO ----");
        Console.WriteLine("---- Temperatura alta, {0} grados ----", t);
        Console.WriteLine("-----");
    }
}
```


Otro código que debemos actualizar es el de la clase **CCaldera**. En ella debemos colocar la lógica necesaria que controla cuándo debe ocurrir el evento relacionado con la temperatura. Para este ejemplo, el evento tiene que generarse cuando la temperatura sea mayor a **115** grados. En el método **Trabajo()** colocamos después de la lógica del evento **combustible** lo necesario para el evento de temperatura:

```
// Este metodo indica el trabajo de la caldera
public void Trabajo()
{
    // Reducimos el combustible
    combustible -= 0.3f;
    // Incrementamos la temperatura
    temperatura += 0.8f;
    // Mostramos la informacion
    Console.WriteLine("Combustible={0}, Temperatura={1}", combustible,
    temperatura);
    // Verificamos si hay la condicion necesaria para hacer una llamada
    // callback y generar el evento
    if (combustible < 5.0f)
    {
        // Recorremos todos los sinks relacionados con la caldera
        // enviandole el mensaje a cada uno de ellos para que actue segun su
        logica
        foreach (IEventosCaldera evento in misSinks)
        {
            // Aqui hacemos la llamada callback
            evento.EPocoCombustible(combustible);
        }
    }
    // Verificamos si hay la condicion necesaria para el evento de la
    temperatura
    if (temperatura > 115.0f)
    {
        // Recorremos todos los sinks relacionados con la caldera
        // enviandole el mensaje a cada uno de ellos para que actue segun su
        logica
    }
}
```

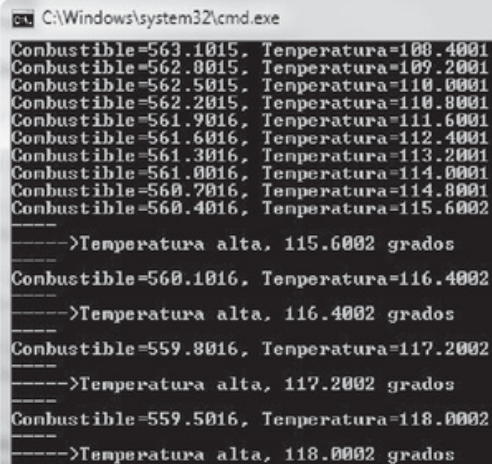
```
        foreach (IEventosCaldera evento in misSinks)
        {
            // Aqui hacemos la llamada callback
            evento.EAltaTemperatura(temperatura);
        }
    }
}
```

Cabe notar que cuando se cumple la condición para el evento relacionado con la temperatura, debemos recorrer todos los sink e invocar el método **EAltaTemperatura()** de cada uno de ellos.

Por último, debemos encargarnos de realizar la actualización en la función **Main()**, que queda de la siguiente manera:

```
static void Main(string[] args)
{
    // Creamos la instancia de la caldera
    CCaldera miCaldera = new CCaldera(600.0f);
    // Creamos el sink normal
    CEventosCalderaSink sinkNormal = new CEventosCalderaSink();
    // Creamos el sink paranoico
    CEventoParanoicoSink sinkParanoico = new CEventoParanoicoSink();
    // Le indicamos a la caldera que se va a comunicar con los sinks
    miCaldera.AdicionaSink(sinkNormal);
    miCaldera.AdicionaSink(sinkParanoico);
    // Hacemos trabajar a la caldera
    while (miCaldera.Combustible > 0 && !sinkParanoico.Detener)
    {
        miCaldera.Trabajo();
    }
}
```

En el código estamos verificando el valor de la propiedad **detener** del objeto **sinkParanoico**. Este valor también determina si el **while** continúa trabajando. La parte importante del código radica en que la comparación se lleva a cabo directamente en la función **Main()**, lo cual indica que **Main()** puede conocer de cierta manera el estado que tiene el sink y también la caldera.



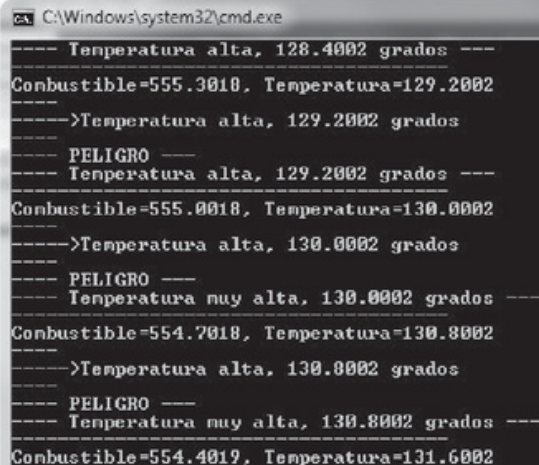
```

C:\Windows\system32\cmd.exe
Combustible=563.1015, Temperatura=108.4001
Combustible=562.8015, Temperatura=109.2001
Combustible=562.5015, Temperatura=110.0001
Combustible=562.2015, Temperatura=110.8001
Combustible=561.9016, Temperatura=111.6001
Combustible=561.6016, Temperatura=112.4001
Combustible=561.3016, Temperatura=113.2001
Combustible=561.0016, Temperatura=114.0001
Combustible=560.7016, Temperatura=114.8001
Combustible=560.4016, Temperatura=115.6002
---->Temperatura alta, 115.6002 grados
Combustible=560.1016, Temperatura=116.4002
---->Temperatura alta, 116.4002 grados
Combustible=559.8016, Temperatura=117.2002
---->Temperatura alta, 117.2002 grados
Combustible=559.5016, Temperatura=118.0002
---->Temperatura alta, 118.0002 grados

```

► **Figura 6.** Aquí observamos la ejecución sin eventos; aún no se detiene el trabajo de la caldera por el exceso de temperatura.

Debemos tener en cuenta que cuando se tiene el valor de **true** en esta propiedad, el ciclo termina, aunque haya combustible. Podemos observar que en el combustible inicial de la caldera se ha colocado el valor de **600**, así podemos proceder a dar la oportunidad de que la temperatura llegue a elevarse lo suficiente.



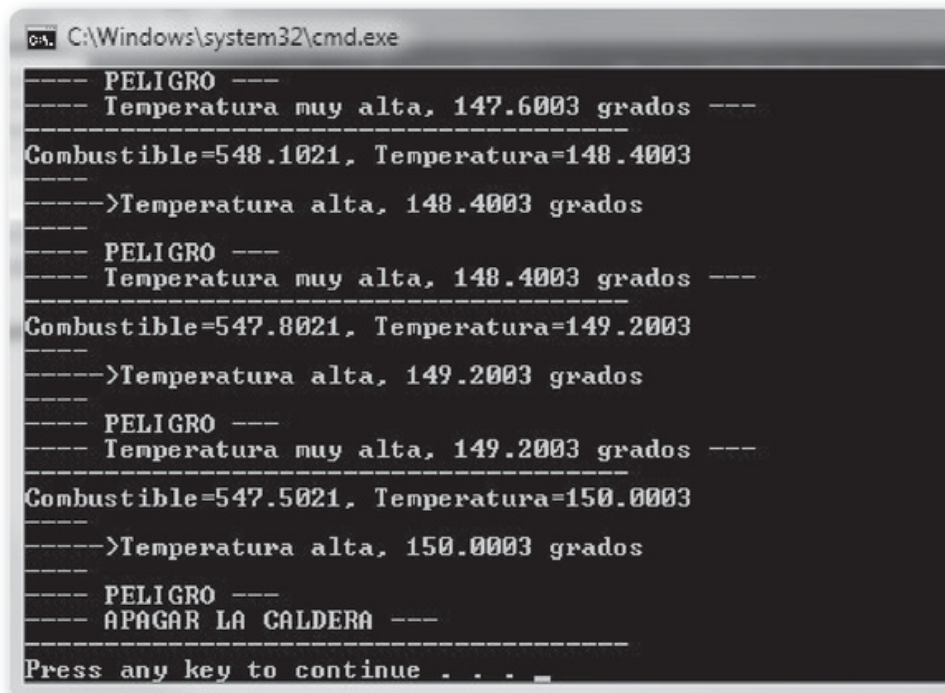
```

C:\Windows\system32\cmd.exe
---- Temperatura alta, 128.4002 grados ----
Combustible=555.3010, Temperatura=129.2002
---->Temperatura alta, 129.2002 grados
---- PELIGRO ----
---- Temperatura alta, 129.2002 grados ----
Combustible=555.0010, Temperatura=130.0002
---->Temperatura alta, 130.0002 grados
---- PELIGRO ----
---- Temperatura muy alta, 130.0002 grados ----
Combustible=554.7010, Temperatura=130.8002
---->Temperatura alta, 130.8002 grados
---- PELIGRO ----
---- Temperatura muy alta, 130.8002 grados ----
Combustible=554.4010, Temperatura=131.6002

```

► **Figura 7.** Esta figura muestra los handlers en ejecución. Tenemos dos mensajes diferentes producidos por el handler relacionado con la temperatura.

Ahora podemos compilar la aplicación y ejecutarla, y comprobar su ejecución en la siguiente figura. Podemos observar cada uno de los comportamientos que estábamos esperando que tuviera nuestro programa y de esta manera comprender mejor el funcionamiento.



```
C:\Windows\system32\cmd.exe
--- PELIGRO ---
--- Temperatura muy alta, 147.6003 grados ---
-----
Combustible=548.1021, Temperatura=148.4003
---
>Temperatura alta, 148.4003 grados
---
PELIGRO ---
--- Temperatura muy alta, 148.4003 grados ---
-----
Combustible=547.8021, Temperatura=149.2003
---
>Temperatura alta, 149.2003 grados
---
PELIGRO ---
--- Temperatura muy alta, 149.2003 grados ---
-----
Combustible=547.5021, Temperatura=150.0003
---
>Temperatura alta, 150.0003 grados
---
PELIGRO ---
APAGAR LA CALDERA ---
-----
Press any key to continue . . . _
```

► **Figura 8.** El handler relacionado con la temperatura se ha encargado de finalizar el trabajo realizado por la caldera.

Los delegados

Ahora que ya conocemos cómo usar las interfaces de callback, podemos pasar a los **delegados**. Los delegados son una forma eficiente, limpia y orientada a objetos para llevar a cabo callbacks. Durante muchos años, antes aún del surgimiento de **.NET**, se han utilizado los callback, principalmente, en la **API** de **Win32**, y eran implementados a la manera tradicional de **C** por medio de apuntadores a funciones. Si bien este mecanismo es eficiente, presenta algunas limitaciones, que son eliminadas al usar delegados.

Una ventaja de los delegados es el uso de un estilo orientado a objetos. Son seguros con relación a los tipos y, en la forma tradicional de apuntadores a funciones, podríamos, por error, acabar apuntando a una función con diferente tipo o distintos parámetros. También pueden ser invocados de manera **síncrona** o **asíncrona**. Esto nos permite llamar al delegado en otro hilo sin tener que usar un objeto **Thread**.

Debemos recordar que el delegado es un objeto que apunta a un método o a varios, que pueden ser invocados posteriormente. Necesita el o los nombres de los métodos por invocar, y también requiere los argumentos de los métodos y el valor de retorno.

Para crear el delegado, hacemos uso de **delegate** seguido del tipo de retorno y el nombre correspondiente, que puede ser cualquiera que deseemos. Luego del nombre, entre paréntesis, indicamos la lista de parámetros. Un ejemplo de delegado es el siguiente:

```
// Aquí definimos el delegado, hay que prestar atención a su tipo y parámetros  
public delegate void DMiMensaje(string m);
```

El delegado creado es **DMiMensaje**, y puede apuntar a cualquier método que tenga un tipo de retorno **void** y que reciba un parámetro de tipo **string**. Si intentamos hacer que el delegado apunte a un método que no tenga estas características, obtendremos un error.

Ejemplo con delegado

Empecemos por ver un primer ejemplo en el uso de los delegados. Vamos a crear un delegado similar al anterior, que apunte a un método que pueda recibir una cadena. El método, entonces, imprimirá la cadena según lo requiera su lógica. El método pertenecerá a otra clase. Por otro lado, para hacer más interesante el programa, durante su ejecución nos encargaremos de cambiar el método al cual apunta el delegado correspondiente.

Creamos un nuevo proyecto y adicionamos una clase llamada **CMostrar**. En el interior de ella colocamos los métodos que serán invocados por medio del delegado adecuado:



SYSTEM.MULTICASTDELEGATE



Debemos tener en cuenta que esta clase provee de varios métodos que nos apoyan en el uso de delegados. Nosotros no invocamos directamente a estos métodos. Algunos de ellos son: **Combine**, **GetInvocationList**, **Remove** y **RemoveAll**. Podemos encontrar más información en el sitio web de MSDN. Aunque no invoquemos estos métodos, es útil conocerlos.

```
// El delegado invocara los metodos de esta clase
public class CMostrar
{
    // Este metodo muestra un mensaje
    public static void MuestraSencillo(string m)
    {
        Console.WriteLine(m);
    }
    // Este metodo muestra el mensaje enmarcado
    public static void MuestraComplicado(string m)
    {
        Console.WriteLine("-----");
        Console.WriteLine(m);
        Console.WriteLine("-----");
    }
}
```

El primer método imprime de manera sencilla la cadena que recibe por parámetro. El segundo método enmarca la cadena al imprimirla. Un punto importante de estos métodos es que ambos tienen el modificador **static**. Los dos métodos poseen el mismo patrón de tipos que el delegado, por lo que puede apuntar hacia ellos sin problema.

Ahora colocamos el código de la función **Main()** y el delegado:

```
namespace Delegados1
{
    // Aqui definimos el delegado, hay que prestar atencion a su tipo y parametros
    public delegate void DMiMensaje(string m);
    class Program
    {
        static void Main(string[] args)
        {
            // Creamos un objeto del delegado y lo referenciamos a un metodo
            DMiMensaje delegadoMensaje = new DMiMensaje(CMostrar.MuestraSencillo);
            // Por medio de delegadoMensaje se invoca el codigo de MuestraSencillo
            delegadoMensaje("Hola a todos");
        }
    }
}
```



```
// Ahora hacemos que el delegado quede referenciado a otro metodo
delegadoMensaje = new DMiMensaje(CMostrar.MuestraComplicado);
// Verificamos que el otro metodo realmente sea invocado por medio del
delegado
delegadoMensaje("Nos vemos");
    }
}
}
```

Primero definimos el delegado. Es importante notar que el delegado no es definido a nivel de la función **Main()**, sino que está siendo declarado a nivel del **namespace**. Ahora, con el delegado declarado dentro de **Main()**, creamos un objeto del delegado, que es de tipo **DMiMensaje** (es decir, el delegado que acabamos de declarar) y se llama **delegadoMensaje**. Como parámetro en el constructor pasamos el método al cual apuntará el delegado. Este es **MuestraSencillo**, un método estático de la clase **CMostrar**. Cuando necesitemos trabajar con el delegado, haremos uso de **delegadoMensaje**.

Ahora deseamos imprimir un mensaje usando el método al cual apunta el delegado, y lo hacemos invocando al delegado y pasando como parámetro el mensaje **"Hola a todos"**. El mensaje se muestra al estilo de **MuestraSencillo()**, pues el delegado apunta a ese método.

Es posible cambiar el método al cual apunta el delegado, y esto se lleva a cabo en la siguiente línea. En este caso, ahora el delegado apunta al método **MuestraComplicado()**. Para continuar, verificamos que esto se realice y mandamos a imprimir un nuevo mensaje, que ahora se presenta al estilo de **MuestraComplicado()**.

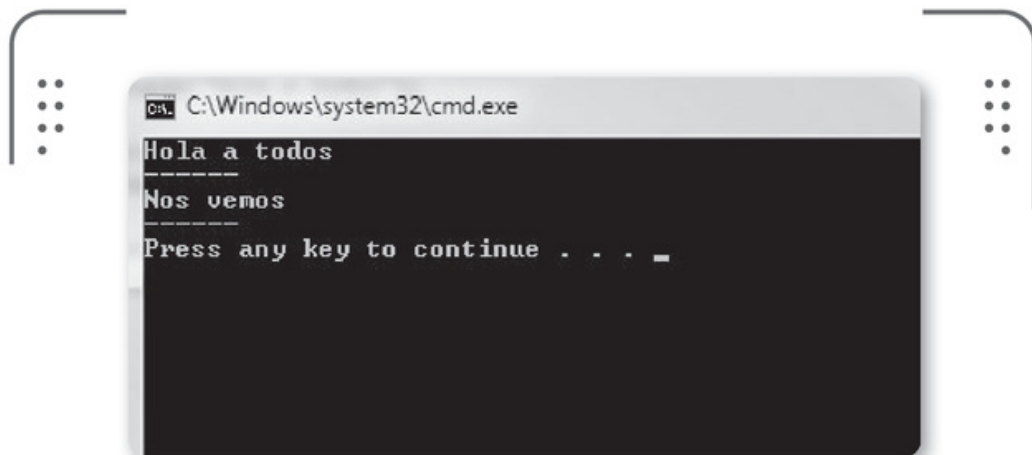
Una vez que accedamos a realizar la compilación y ejecución del programa, podremos ver algo similar a la **Figura 9**.



MÁS SOBRE EL OPERADOR +=



En este punto ya sabemos un poco sobre el uso de += con los delegados. Este es un operador sobrecargado que, en realidad, invoca al método **Combine()** de la clase que hemos creado. Podemos encontrar más información sobre los operadores en MSDN. El uso correcto de la sobrecarga de operadores puede simplificar de manera notable nuestro código.



► **Figura 9.** En la figura que vemos aquí se muestran los mensajes que se han desplegado vía el delegado.

Ahora intentaremos hacer algo similar al control de la caldera con interfaces de callback, pero usando los delegados. Los métodos que serán invocados quedarán en la clase **Program**. Tendremos un método para el evento del combustible y otro para el de la temperatura. La clase **caldera** tendrá en su interior los delegados y la lógica necesaria para controlar cuando se genera el evento.

Empecemos por crear la clase **CCaldera**, que tiene el siguiente código:

```
class CCaldera
{
    // Declaramos los delegados necesarios para la clase

    // Delegado para el poco combustible
    public delegate void DPocoCombustible(float c);
    // Delegado para la alta temperatura
    public delegate void DAltaTemperatura(float t);
    // Creamos las variables para referenciar a los delegados
    // Variable para el poco combustible
    private DPocoCombustible mPocoCombustible;
    // Variable para la alta temperatura
    private DAltaTemperatura mAltaTemperatura;
    // Datos necesarios
    private float combustible;
```

```
private float temperatura;
// Constructor
public CCaldera(float combInicial)
{
    // Asignamos el combustible inicial
    combustible = combInicial;
    // Inicializamos la temperatura
    temperatura = 10.0f;
}
// Estos metodos nos permiten referenciar las variables a los metodos que
se invocaran

public void AdicionaPocoCombustible(DPocoCombustible pmetodo)
{
    // Asignamos el metodo a la variable
    mPocoCombustible = pmetodo;
}
public void AdicionaAltaTemperatura(DAltaTemperatura pmetodo)
{
    // Asignamos el metodo a la variable
    mAltaTemperatura = pmetodo;
}
public float Combustible
{
    get { return combustible; }
}
// Este metodo indica el trabajo de la caldera
public void Trabajo()
{
    // Reducimos el combustible
    combustible -= 0.3f;
    // Incrementamos la temperatura
    temperatura += 0.8f;
    // Mostramos la informacion
    Console.WriteLine("Combustible={0}, Temperatura={1}", combustible,
    temperatura);
    // Verificamos si hay la condicion necesaria para hacer una llamada
```



```
// callback y generar el evento
if (combustible < 5.0f)
{
    // Mandamos el evento de poco combustible
    mPocoCombustible(combustible);
}
// Verificamos si hay la condicion necesaria para el evento de la
temperatura
if (temperatura > 115.0f)
{
    // Mandamos el evento de alta temperatura
    mAltaTemperatura(temperatura);
}
}
```

Lo primero que encontramos es la declaración de los delegados. El delegado para el combustible se llama **DPocoCombustible**, y podrá apuntar a funciones que tienen tipo **void** y reciben un único parámetro de tipo **float**. El delegado para la temperatura lleva el nombre **DAltaTemperatura**, y puede apuntar a métodos **void** y que reciban un parámetro **float**.

Después de declarar los delegados, procedemos a crear una variable de cada uno, ya que son necesarias para trabajar con ellos. Las variables que tenemos son **mPocoCombustible** y **mAltaTemperatura**. Posteriormente,

tendremos el código que permite indicar cuál es la función a la que apuntan.

Inmediatamente, tenemos los datos propios de la clase y el constructor de esta, en el que asignamos la cantidad de combustible y una temperatura inicial de **10**.

El delegado tiene que apuntar a un método, para lo cual creamos un par de métodos de apoyo. El primero es **AdicionaPocoCombustible()**, de tipo **void**, por lo que no regresa ningún valor. El parámetro que recibe es de tipo

DPocoCombustible, lo que nos permite apuntar a cualquier método que tenga las características de tipo definidas en el delegado. El parámetro se llama **pmetodo**. En el interior del método

EL DELEGADO TIENE
QUE APUNTAR A UN
MÉTODO, EL CUAL
LLEVARÁ LA LÓGICA
NECESARIA



simplemente llevamos a cabo la asignación de **pmetodo** a la variable **mPocoCombustible**. Esto es todo lo que necesitamos para que el delegado apunte al método correspondiente. Algo similar ocurre con **AdicionaAltaTemperatura()**, pero en este caso, para el delegado **DAltaTemperatura**.

En la clase tenemos una propiedad llamada **Combustible**, que es usada por la lógica de **Main()** y únicamente nos sirve de apoyo.

En el método **trabajo** tenemos lo usual, pero cambiamos la forma en que se dispara el evento. En este caso, como estamos haciendo uso de los delegados, el método se invoca por medio de la variable de delegado referenciada. Para el evento relacionado con el combustible, hacemos uso de **mPocoCombustible**, y para la temperatura, de **mAltaTemperatura**. Cuando usamos la variable entre paréntesis, pasamos la variable **combustible** como parámetro al método por invocar.

Ahora creamos el código de la función **Main()** y los métodos que serán invocados por los delegados:

EL MÉTODO SE
INVOCA POR MEDIO
DE LA VARIABLE
DE DELEGADO
REFERENCIADA

”

```
class Program
{
    static void Main(string[] args)
    {
        // Creamos la instancia de la caldera
        CCaldera miCaldera = new CCaldera(50.0f);
        // Adicionamos los handlers correspondientes
        miCaldera.AdicionaPocoCombustible(new CCaldera.DPocoCombustible
        (MetodoCombustible));
        miCaldera.AdicionaAltaTemperatura(new CCaldera.DAltaTemperatura
        (MetodoAltaTemperatura));
        // Hacemos trabajar a la caldera
        while (miCaldera.Combustible > 0)
        {
            miCaldera.Trabajo();
        }
    }
}
```

```
public static void MetodoCombustible(float c)
{
    // Mandamos el mensaje de que queda poco combustible
    // Aqui podemos adicionar más logica que se lleve a cabo cuando suceda el
    evento
    Console.WriteLine("----");
    Console.WriteLine("---->Queda poco combustible, {0} litros", c);
    Console.WriteLine("----");
}
public static void MetodoAltaTemperatura(float t)
{
    // Mandamos el mensaje de exceso de temperatura
    Console.WriteLine("----");
    Console.WriteLine("---->Temperatura alta, {0} grados", t);
    Console.WriteLine("----");
}
}
```

Debemos tener en cuenta que en la función **Main()** empezamos por crear una instancia de la clase **CCaldera** y damos un valor inicial de **50** al combustible. Ya teniendo la caldera, adicionamos la referencia al delegado. Primero lo hacemos para el delegado del combustible y, luego, para la temperatura. Hay que notar el nombre de los métodos que quedan referenciados a cada delegado.

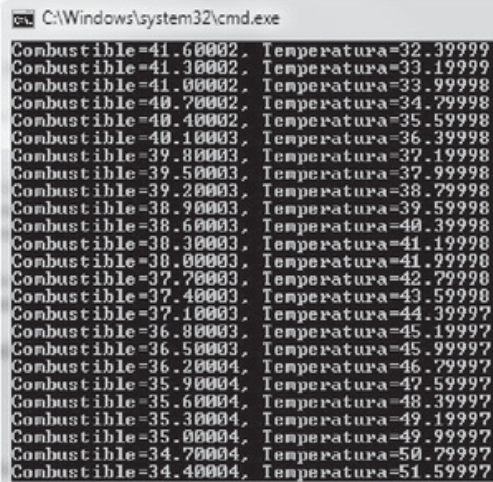
Luego de realizar esta tarea, simplemente tenemos un ciclo **while** donde se lleva a cabo el trabajo correspondiente a la caldera. El ciclo lee la propiedad de **Combustible** y, mientras haya, la caldera se encargará de seguir trabajando normalmente.



MÁS SOBRE EL OPERADOR -=



Ya sabemos un poco sobre el uso del operador **-=** junto a los delegados. Debemos tener en cuenta que se trata de un operador sobrecargado que, en realidad, se ocupa de invocar al método denominado **Remove()** de la clase que hemos creado. De esta forma, podemos entender los operadores como si fueran métodos, pero que poseen nombres que llevan signos en vez de letras; los parámetros que corresponden a estos operadores son los llamados operandos.



```

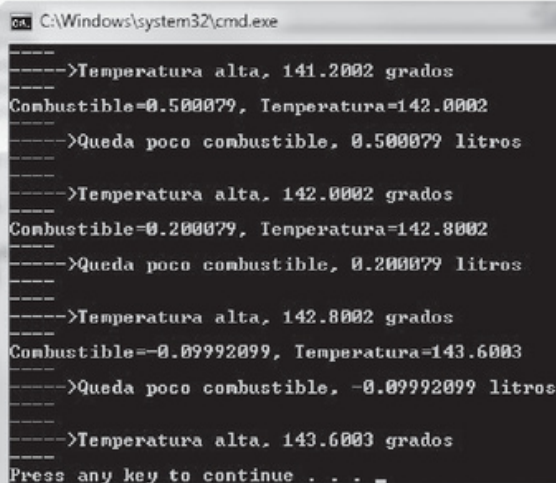
C:\Windows\system32\cmd.exe
Combustible=41.60002, Temperatura=32.39999
Combustible=41.30002, Temperatura=33.19999
Combustible=41.00002, Temperatura=33.99998
Combustible=40.70002, Temperatura=34.79998
Combustible=40.40002, Temperatura=35.59998
Combustible=40.10003, Temperatura=36.39998
Combustible=39.80003, Temperatura=37.19998
Combustible=39.50003, Temperatura=37.99998
Combustible=39.20003, Temperatura=38.79998
Combustible=38.90003, Temperatura=39.59998
Combustible=38.60003, Temperatura=40.39998
Combustible=38.30003, Temperatura=41.19998
Combustible=38.00003, Temperatura=41.99998
Combustible=37.70003, Temperatura=42.79998
Combustible=37.40003, Temperatura=43.59998
Combustible=37.10003, Temperatura=44.39997
Combustible=36.80003, Temperatura=45.19997
Combustible=36.50003, Temperatura=45.99997
Combustible=36.20004, Temperatura=46.79997
Combustible=35.90004, Temperatura=47.59997
Combustible=35.60004, Temperatura=48.39997
Combustible=35.30004, Temperatura=49.19997
Combustible=35.00004, Temperatura=49.99997
Combustible=34.70004, Temperatura=50.79997
Combustible=34.40004, Temperatura=51.59997

```

► **Figura 10.** En esta parte de la ejecución del programa, aún no se generan los eventos que invocarán al método vía los delegados.

Debemos saber que el método que será invocado por el delegado asociado al combustible es el denominado **MétodoCombustible**. Este simplemente se encarga de mandar un mensaje indicando que hay poco combustible y también la cantidad de litros que corresponden. De esta forma, el método tiene el mismo patrón de tipos que el delegado y posee un modificador llamado **static**. Algo similar ocurre con el método **MetodoAltaTemperatura()**.

Para continuar, debemos proceder a realizar la compilación y posterior ejecución de la aplicación, así podremos ver la forma en que actúa el delegado correspondiente.



```

C:\Windows\system32\cmd.exe
----->Temperatura alta, 141.2002 grados
Combustible=0.500079, Temperatura=142.0002
----->Queda poco combustible, 0.500079 litros
----->Temperatura alta, 142.0002 grados
Combustible=0.200079, Temperatura=142.8002
----->Queda poco combustible, 0.200079 litros
----->Temperatura alta, 142.8002 grados
Combustible=-0.09992099, Temperatura=143.6003
----->Queda poco combustible, -0.09992099 litros
----->Temperatura alta, 143.6003 grados
Press any key to continue . . .

```

► **Figura 11.** Aquí podemos observar que los métodos fueron invocados por medio de los delegados.

Delegados con multicasting

Hemos usado los delegados, pero únicamente invocan a un método. Sin embargo, ya tienen provista la funcionalidad para poder invocar a más de uno. Esto es similar a cuando usábamos varios handlers con las interfaces de callback. La adición de varios métodos que pueden ser invocados por el delegado se denomina **multicasting**. El uso de multicasting es muy sencillo y solo tenemos que hacer unos cambios pequeños en el código.

Con él podemos adicionar métodos por ser invocados, pero también, si la lógica del programa lo requiere, podemos eliminar métodos de la lista de invocaciones.

Empezamos por realizar la modificación en la clase llamada **CCaldera**, la cual debe quedar de la siguiente manera:

```
class CCaldera
{
    // Declaramos los delegados necesarios para la clase

    // Delegado para el poco combustible
    public delegate void DPocoCombustible(float c);
    // Delegado para la alta temperatura
    public delegate void DAltaTemperatura(float t);
    // Creamos las variables para referenciar a los delegados
    // Variable para el poco combustible
    private DPocoCombustible mPocoCombustible;
    // Variable para la alta temperatura
    private DAltaTemperatura mAltaTemperatura;
    // Datos necesarios
    private float combustible;
    private float temperatura;
    // Constructor
    public CCaldera(float combInicial)
    {
        // Asignamos el combustible inicial
        combustible = combInicial;
        // Inicializamos la temperatura
        temperatura = 10.0f;
    }
}
```

```
}  
// Estos metodos nos permiten referenciar las variables a los metodos que se  
invocaran  
// Hay que notar que estamos haciendo uso de multicasting  
  
public void AdicionaPocoCombustible(DPocoCombustible metodo)  
{  
    // Asignamos el metodo a la variable  
    mPocoCombustible += metodo;  
}  
public void AdicionaAltaTemperatura(DAltaTemperatura metodo)  
{  
    // Asignamos el metodo a la variable  
    mAltaTemperatura += metodo;  
}  
// Este metodo nos permite eliminar un handler  
public void EliminaPocoCombustible(DPocoCombustible metodo)  
{  
    // Asignamos el metodo a la variable  
    mPocoCombustible -= metodo;  
}  
public float Combustible  
{  
    get { return combustible; }  
    set { combustible = value; }  
}  
// Este metodo indica el trabajo de la caldera  
public void Trabajo()  
{  
    // Reducimos el combustible  
    combustible -= 0.3f;  
    // Incrementamos la temperatura  
    temperatura += 0.8f;  
    // Mostramos la informacion  
    Console.WriteLine("Combustible={0}, Temperatura={1}", combustible,  
        temperatura);  
    // Verificamos si hay la condicion necesaria para hacer una llamada
```



```
// callback y generar el evento
if (combustible < 5.0f)
{
    // Mandamos el evento de poco combustible
    mPocoCombustible(combustible);
}
// Verificamos si hay la condicion necesaria para el evento de la
// temperatura
if (temperatura > 115.0f)
{
    // Mandamos el evento de alta temperatura
    mAltaTemperatura(temperatura);
}
}
```

Veamos los cambios. En el método **AdicionaPocoCombustible()**, notamos que, en el momento en que estamos adicionando el método a la variable, usamos el operador **+=**, con el cual podemos agregar métodos a una lista de los que se invocarán dentro de ese delegado. Recordemos que lo mismo ocurre para el método de **AdicionaAltaTemperatura()**.

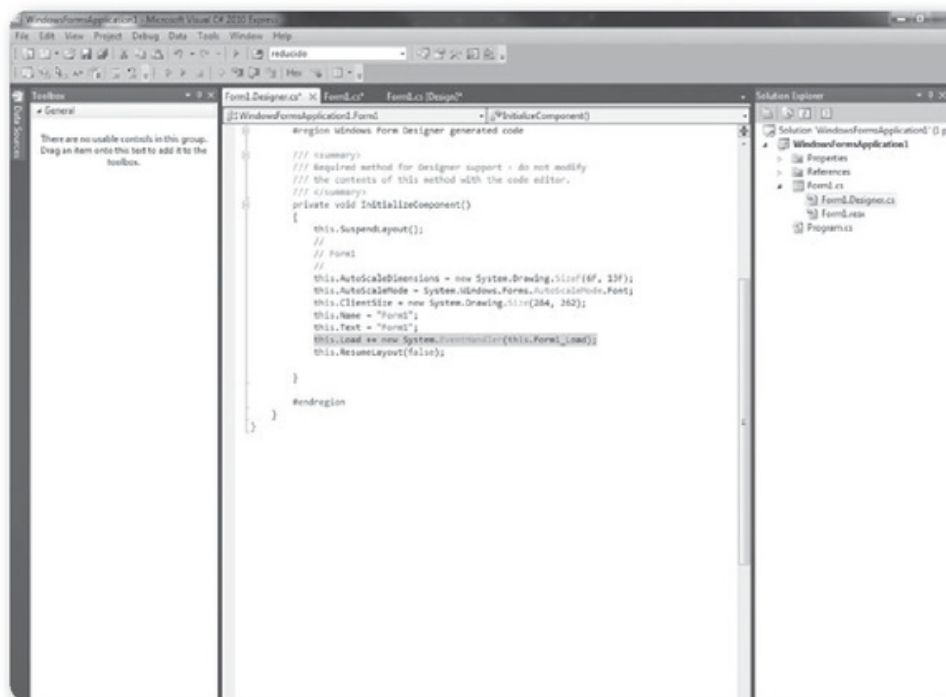
Es necesario tener en cuenta que si deseamos realizar la eliminación de un método que se encuentra en la lista de métodos a invocar por el delegado, podemos hacerlo por medio del operador **-=**. Esto puede verse en la práctica en el método llamado **EliminaPocoCombustible()**, que recibe como parámetro un método del tipo **DPocoCombustible**, que luego es eliminado del delegado.



SOBRECARGA DE +=



Es necesario tener en cuenta que el operador de **+=** es una sobrecarga de operador que, en realidad, invoca a un método encargado de llevar a cabo la adición del método por invocar. Veremos las sobrecargas de operadores en otro capítulo. El método llevará la lógica necesaria para hacer la operación desde la perspectiva de nuestra clase. Los operadores necesitan de operandos, los cuales pueden ser vistos como los parámetros que recibe el método. De esta forma, el valor que se ha calculado por el método es el resultado de la operación correspondiente.



► **Figura 12.** Esta figura muestra el código que Visual Studio genera por nosotros cuando hacemos uso del evento **Load** de la forma.

El último cambio que encontramos en este clase es la propiedad **Combustible**, la cual ahora es de entrada y de salida.

También tenemos que hacer modificaciones y adiciones en la clase **Program**. En primer lugar, procedemos a agregar un nuevo método que deberá ser invocado por el delegado, de la siguiente forma:

```
public static void MetodoCombustibleParanoico(float c)
{
    // Mandamos el mensaje de que queda poco combustible
    // Aqui podemos adicionar más logica que se lleve a cabo cuando suceda el
    // evento
    Console.WriteLine("---- PELIGRO ---");
    Console.WriteLine("---- PELIGRO ---");
    Console.WriteLine("----->CASI SIN COMBUSTIBLE, {0} litros", c);
    Console.WriteLine("-----");
}
```

La función **Main()** debe quedar de la siguiente manera:

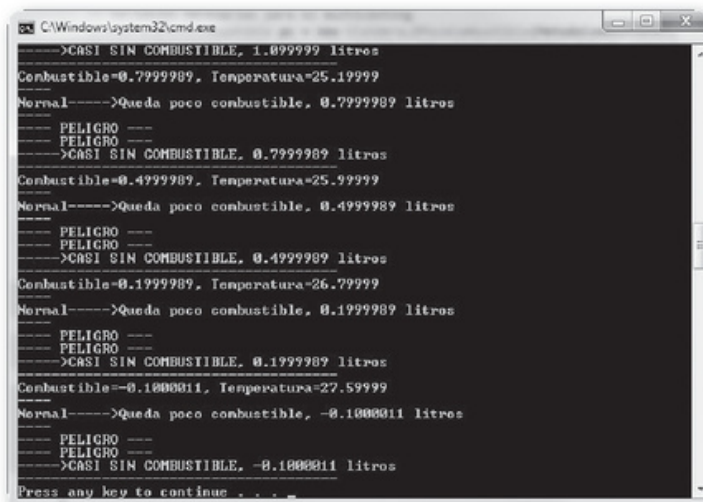
```
static void Main(string[] args)
{
    // Creamos la instancia de la caldera
    CCaldera miCaldera = new CCaldera(6.5f);
    // Variables necesarias para el multicasting
    CCaldera.DPocoCombustible pc = new CCaldera.DPocoCombustible
    (MetodoCombustible);
    CCaldera.DPocoCombustible pcp = new CCaldera.DPocoCombustible
    (MetodoCombustibleParanoico);
    CCaldera.DAltaTemperatura at = new CCaldera.DAltaTemperatura
    (MetodoAltaTemperatura);
    // Adicionamos los handlers correspondientes
    miCaldera.AdicionaPocoCombustible(pc);
    miCaldera.AdicionaPocoCombustible(pcp);
    miCaldera.AdicionaAltaTemperatura(at);
    // Hacemos trabajar a la caldera con todos los handlers
    while (miCaldera.Combustible > 0)
    {
        miCaldera.Trabajo();
    }
    // Eliminamos el handler, solo queda el paranoico
    miCaldera.EliminaPocoCombustible(pc);
    miCaldera.Combustible = 7.0f;
    Console.WriteLine("*****");
    // Hacemos trabajar a la caldera con todos los handlers
    while (miCaldera.Combustible > 0)
    {
        miCaldera.Trabajo();
    }
}
```

En esta función empezamos por crear una instancia de **CCaldera**. Luego, tenemos un cambio que nos facilita el hecho de poder asignar los métodos al delegado. Simplemente, debemos encargarnos de realizar la creación de una serie de variables del tipo del delegado y, por otra parte, por otra parte generamos la instancia pasando como

parámetro el método que nos interesa utilizar. En este caso, tenemos dos variables: una para cada método del delegado asociado al poco combustible y otra para el delegado de la alta temperatura.

Con estas variables, usamos el método denominado **AdicionaPocoCombustible()** para colocar en el delegado asociado al poco combustible los métodos de **MetodoCombustible** y **MetodoCombustibleParanoico**. Lo mismo sucede con el delegado asociado con la alta temperatura y el método **MetodoAltaTemperatura**. El delegado del poco combustible invocará a dos métodos.

Hacemos una prueba de la forma usual; cabe recordar que la caldera se inició con **6.5** litros de combustible para que la prueba sea más rápida. Después de ella, eliminamos uno de los handlers con el método de **EliminaPocoCombustible()**. Le asignamos nuevo combustible a la caldera y hacemos una nueva prueba, con la cual verificamos que, efectivamente, cada delegado ahora trabaja con un handler.



```

C:\Windows\system32\cmd.exe
>CASI SIN COMBUSTIBLE, 1.099999 litros
Combustible=0.7999989, Temperatura=25.19999
Normal---->Queda poco combustible, 0.7999989 litros
PELIGRO ---
PELIGRO ---
>CASI SIN COMBUSTIBLE, 0.7999989 litros
Combustible=0.4999989, Temperatura=25.99999
Normal---->Queda poco combustible, 0.4999989 litros
PELIGRO ---
PELIGRO ---
>CASI SIN COMBUSTIBLE, 0.4999989 litros
Combustible=0.1999989, Temperatura=26.79999
Normal---->Queda poco combustible, 0.1999989 litros
PELIGRO ---
PELIGRO ---
>CASI SIN COMBUSTIBLE, 0.1999989 litros
Combustible=-0.1000011, Temperatura=27.59999
Normal---->Queda poco combustible, -0.1000011 litros
PELIGRO ---
PELIGRO ---
>CASI SIN COMBUSTIBLE, -0.1000011 litros
Press any key to continue . . .

```

► **Figura 13.** En esta figura es posible observar la primera parte de la ejecución del programa con los tres handlers.

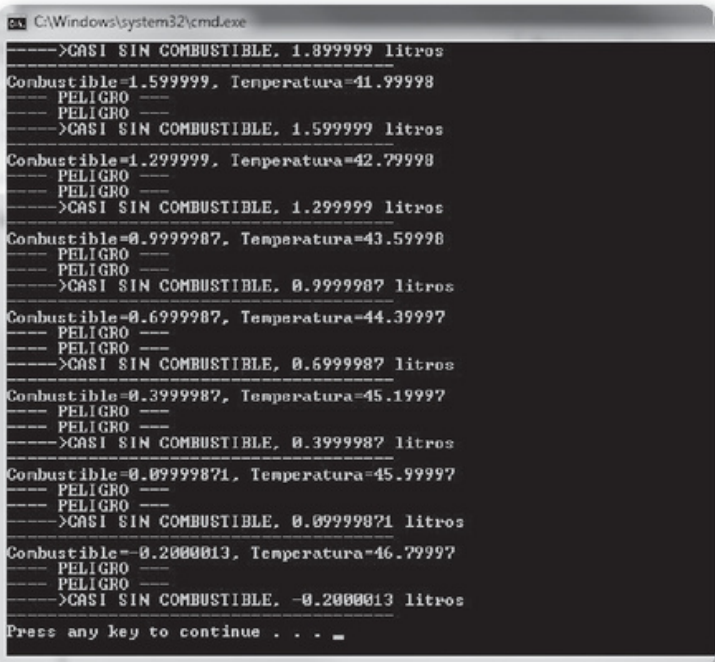


USO DE DELEGADOS COMO PARÁMETROS



Tengamos en cuenta que para tener aún más flexibilidad y llevar a cabo programas más avanzados, podemos hacer uso de los delegados como parámetros. De esta manera, un método puede recibir un delegado y trabajar con él. Con esto, nuestro método puede trabajar con cualquier otra función, evitando que se cierre a expansiones del programa.

Ahora podemos compilar y ejecutar la aplicación.



```
C:\Windows\system32\cmd.exe
>CASI SIN COMBUSTIBLE. 1.899999 litros
Combustible=1.599999, Temperatura=41.99998
PELIGRO
PELIGRO
>CASI SIN COMBUSTIBLE. 1.599999 litros
Combustible=1.299999, Temperatura=42.79998
PELIGRO
PELIGRO
>CASI SIN COMBUSTIBLE. 1.299999 litros
Combustible=0.9999987, Temperatura=43.59998
PELIGRO
PELIGRO
>CASI SIN COMBUSTIBLE. 0.9999987 litros
Combustible=0.6999987, Temperatura=44.39997
PELIGRO
PELIGRO
>CASI SIN COMBUSTIBLE. 0.6999987 litros
Combustible=0.3999987, Temperatura=45.19997
PELIGRO
PELIGRO
>CASI SIN COMBUSTIBLE. 0.3999987 litros
Combustible=0.09999871, Temperatura=45.99997
PELIGRO
PELIGRO
>CASI SIN COMBUSTIBLE. 0.09999871 litros
Combustible=-0.2000013, Temperatura=46.79997
PELIGRO
PELIGRO
>CASI SIN COMBUSTIBLE. -0.2000013 litros
Press any key to continue . . . _
```

► **Figura 14.** Esta es la segunda parte de la ejecución, donde solamente se tienen dos handlers.

Entender los eventos

Los **eventos** nos permiten programar la funcionalidad que hemos aprendido con los delegados de una forma más sencilla y directa. Son muy útiles pues permiten la comunicación entre objetos vía mecanismos de callback. Esta es la manera más usual en que encontraremos el callback en los programas escritos en C#.

Vamos a modificar el programa anterior para hacer uso de los eventos. Cuando deseamos definir un evento, lo hacemos en dos pasos. En primer lugar, definimos un delegado, que se encarga de guardar cuáles son los métodos por invocar; luego tenemos que definir como tal el evento. El delegado y el evento tendrán el mismo nombre y, por medio de esa variable de evento, llevaremos a cabo toda la funcionalidad.

Ya no será necesario crear métodos especializados para adicionar y eliminar métodos de los delegados, pues podremos utilizar directamente los operadores += y -=. Así, todo resulta más sencillo.

Empecemos a realizar los cambios adecuados a la clase **CCaldera**, la cual debe quedar de la siguiente manera:

```
class CCaldera
{
    // Declaramos los delegados necesarios para la clase

    // Delegado para el poco combustible
    public delegate void DPocoCombustible(float c);
    // Delegado para la alta temperatura
    public delegate void DAltaTemperatura(float t);
    // Creamos las variables para referenciar a los delegados
    // Variable para el poco combustible
    public event DPocoCombustible mPocoCombustible;
    // Variable para la alta temperatura
    public event DAltaTemperatura mAltaTemperatura;
    // Datos necesarios
    private float combustible;
    private float temperatura;
    // Constructor
    public CCaldera(float combInicial)
    {
        // Asignamos el combustible inicial
        combustible = combInicial;
        // Inicializamos la temperatura
        temperatura = 10.0f;
    }
    public float Combustible
    {
        get { return combustible; }
        set { combustible = value; }
    }
    // Este metodo indica el trabajo de la caldera
    public void Trabajo()
```



```
{
    // Reducimos el combustible
    combustible -= 0.3f;
    // Incrementamos la temperatura
    temperatura += 0.8f;
    // Mostramos la informacion
    Console.WriteLine("Combustible={0}, Temperatura={1}", combustible,
    temperatura);
    // Verificamos si hay la condicion necesaria para hacer una llamada
    // callback y generar el evento
    if (combustible < 5.0f)
    {
        // Mandamos el evento de poco combustible
        mPocoCombustible(combustible);
    }
    // Verificamos si hay la condicion necesaria para el evento de la
    temperatura
    if (temperatura > 115.0f)
    {
        // Mandamos el evento de alta temperatura
        mAltaTemperatura(temperatura);
    }
}
```

Empezamos con los delegados, uno para cada uno de los eventos que nos interesan. Más abajo tenemos las variables para los eventos; en este caso, también usamos una variable para cada uno de ellos. Todo el mecanismo relacionado con el poco combustible se manejará

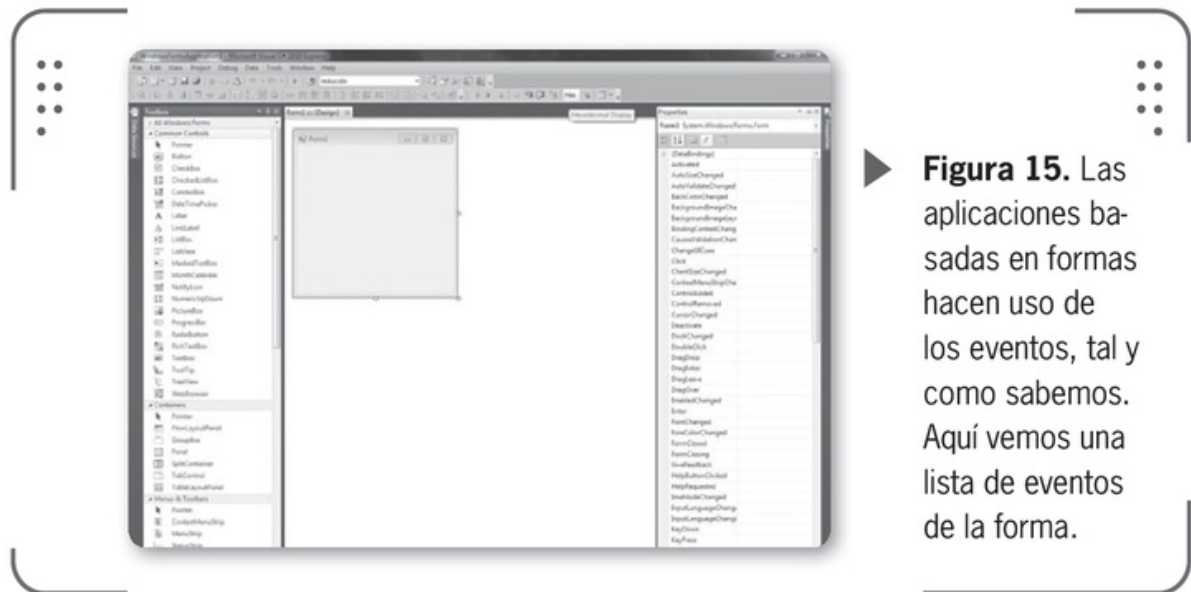


LOS DELEGADOS SON IMPORTANTES



Tengamos en cuenta que es muy importante, si deseamos llegar a niveles avanzados de programación orientada a objetos en cualquier lenguaje, aprender a usar y entender los delegados. Estos nos dan grandes beneficios en la forma como podemos trabajar con nuestra lógica y la forma como diferentes partes de nuestro programa se comunican entre si.

por medio de **mPocoCombustible**, en tanto que el referido a la alta temperatura lo hará por medio de **mAltaTemperatura**. Es necesario recordar que las variables tienen un acceso público para facilitarnos la adición de handlers desde el exterior. El resto del código es prácticamente el mismo que en el ejemplo anterior.



► **Figura 15.** Las aplicaciones basadas en formas hacen uso de los eventos, tal y como sabemos. Aquí vemos una lista de eventos de la forma.

Luego tenemos que hacer los cambios en la clase **Program**. Para este ejemplo, solamente la función **Main()** necesita modificarse; los otros métodos permanecen sin cambio:

```
static void Main(string[] args)
{
    // Creamos la instancia de la caldera
    CCaldera miCaldera = new CCaldera(6.5f);
    // Variables necesarias para el multicasting
    CCaldera.DPocoCombustible pc = new CCaldera.DPocoCombustible
    (MetodoCombustible);
    CCaldera.DPocoCombustible pcp = new CCaldera.DPocoCombustible
    (MetodoCombustibleParanoico);
    CCaldera.DAltaTemperatura at = new CCaldera.DAltaTemperatura
    (MetodoAltaTemperatura);
    // Adicionamos los handlers correspondientes
    miCaldera.mPocoCombustible += pc;
```

```

miCaldera.mPocoCombustible += pcp;
miCaldera.mAltaTemperatura += at;
// Hacemos trabajar a la caldera con todos los handlers
while (miCaldera.Combustible > 0)
{
    miCaldera.Trabajo();
}
// Eliminamos el handler, solo queda el paranoico
miCaldera.mPocoCombustible -= pc;
miCaldera.Combustible = 7.0f;
Console.WriteLine("*****");
// Hacemos trabajar a la caldera con todos los handlers
while (miCaldera.Combustible > 0)
{
    miCaldera.Trabajo();
}
}

```

Tengamos en cuenta que el código es prácticamente el mismo que en los ejemplos anteriores, pero debemos notar que la forma en que estamos adicionando los handlers es por medio del operador `+=`, y lo hacemos directamente a `mPocoCombustible` o `mAltaTemperatura`, los cuales fueron declarados con acceso público. Para eliminar uno de los handlers usamos el operador `-=`.

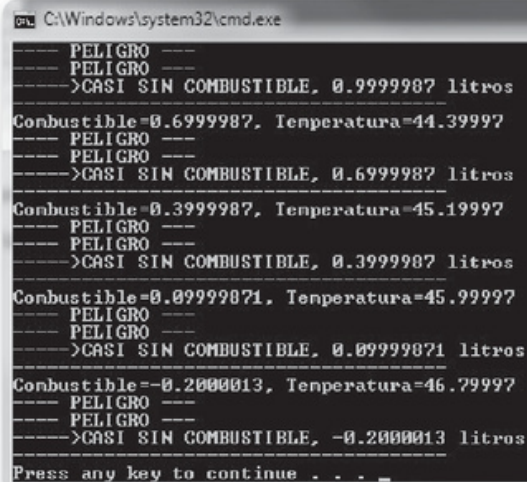
```

C:\Windows\system32\cmd.exe
Combustible=0.4999989, Temperatura=25.99999
Normal----->Queda poco combustible, 0.4999989 litros
-----
PELIGRO ---
PELIGRO ---
----->CASI SIN COMBUSTIBLE, 0.4999989 litros
Combustible=0.1999989, Temperatura=26.79999
Normal----->Queda poco combustible, 0.1999989 litros
-----
PELIGRO ---
PELIGRO ---
----->CASI SIN COMBUSTIBLE, 0.1999989 litros
Combustible=-0.1000011, Temperatura=27.59999
Normal----->Queda poco combustible, -0.1000011 litros
-----
PELIGRO ---
PELIGRO ---
----->CASI SIN COMBUSTIBLE, -0.1000011 litros
Press any key to continue . . .

```

► **Figura 16.** Esta es la primera parte de la ejecución con los tres handlers trabajando por medio de los eventos.

Ejecutemos el programa y veamos cómo se comporta.



```
C:\Windows\system32\cmd.exe
----- PELIGRO -----
----- PELIGRO -----
----->CASI SIN COMBUSTIBLE, 0.9999987 litros
-----
Combustible=0.6999987, Temperatura=44.39997
----- PELIGRO -----
----- PELIGRO -----
----->CASI SIN COMBUSTIBLE, 0.6999987 litros
-----
Combustible=0.3999987, Temperatura=45.19997
----- PELIGRO -----
----- PELIGRO -----
----->CASI SIN COMBUSTIBLE, 0.3999987 litros
-----
Combustible=0.09999871, Temperatura=45.99997
----- PELIGRO -----
----- PELIGRO -----
----->CASI SIN COMBUSTIBLE, 0.09999871 litros
-----
Combustible=-0.2000013, Temperatura=46.79997
----- PELIGRO -----
----- PELIGRO -----
----->CASI SIN COMBUSTIBLE, -0.2000013 litros
-----
Press any key to continue . . .
```

► **Figura 17.** En esta figura vemos la segunda parte de la ejecución, donde ya se ha eliminado un handler.

Ahora, gracias a lo que analizamos en este capítulo, ya conocemos una parte importante de la programación avanzada en C#.



RESUMEN



El propósito principal de este capítulo es conocer los mecanismos de callback, con los cuales un objeto puede comunicarse y mandar mensajes a su invocador o contenedor. Empezamos viendo las interfaces de callback. Con ellas definimos interfaces que presentan los comportamientos que tendrán los handlers cuando el evento se genere. Los delegados nos brindan otra forma de llevar a cabo estos comportamientos, pero son más flexibles y ofrecen multicasting. La manera más sencilla es por medio de los eventos, ya que así aprovechamos muchas funcionalidades incorporadas en el lenguaje.

Actividades

TEST DE AUTOEVALUACIÓN

- 1 ¿Qué es el callback?
- 2 ¿De qué maneras podemos implementar mecanismos de callback?
- 3 ¿Qué es una interfaz de callback?
- 4 ¿Qué es un sink?
- 5 ¿Qué características debe tener un sink?
- 6 ¿Qué es un delegado?
- 7 ¿Cómo se define un delegado?
- 8 ¿De qué forma se invoca un delegado?
- 9 ¿Qué es el multicasting?
- 10 ¿Qué es un evento?
- 11 ¿Para qué se utiliza el operador += en este caso?
- 12 ¿Para qué se utiliza el operador -= en este caso?

EJERCICIOS PRÁCTICOS

- 1 Investigue cómo se relacionan los eventos de los controles de una forma con los eventos que hemos visto.
- 2 Cree un programa con varios objetos que generen eventos usando interfaces de callback.
- 3 Modifique el programa anterior para usar delegados.
- 4 Modifique el programa anterior para usar eventos.
- 5 Modifique el programa anterior para hacer uso de multicasting.



Manejo avanzado de clases

En este capítulo aprenderemos nuevas técnicas de programación que nos permitirán llevar a cabo operaciones avanzadas con las clases. Con la sobrecarga de operadores, podremos tener código más limpio y fácil de programar, en tanto el uso de indexers nos ayudará a acceder a arreglos o estructuras de datos internas a la clase.

▼ Comprender los indexers.....	92	▼ Sobrecarga de operadores de igualdad y comparación.....	112
▼ Sobrecarga de operadores.....	98	▼ Sobrecarga de los operadores < y >	118
▼ Sobrecarga de operadores binarios.....	100	▼ Resumen.....	125
▼ Sobrecarga de operadores unuarios	107	▼ Actividades.....	126





Comprender los indexers

En un capítulo anterior observamos cómo, al implementar la interfaz de **IEnumerable**, podemos acceder a una estructura de datos que esté en el interior de una clase. Este acceso se realiza por medio del iterador **foreach**. Es muy común que tengamos clases que guarden en su interior arreglos o algún otro tipo de **ADT**. De la manera tradicional, tendríamos que crear algún tipo de función de interfaz para leer los datos que deseamos. Si bien esto es correcto, sería mejor que pudiéramos acceder de modo sencillo a esa información, con el fin de simplificar la programación de la lógica que haga uso de esa clase.

Aunque hemos utilizado **IEnumerable**, existe otra forma más sencilla de llevarlo a cabo. Esta nueva forma se conoce como **indexer**. El indexer nos permitirá acceder a ese ADT de manera similar a como utilizamos un arreglo común y corriente. Podemos darnos cuenta de que esto haría más fácil la programación de la lógica.

Un punto importante para notar es que el indexer solo nos da una sintaxis similar a la de un arreglo, pero no obtenemos todas las funcionalidades o métodos de apoyo que este tiene. El indexer es flexible y podemos utilizarlo de forma monodimensional, bidimensional, etc.

La mejor manera de entender cómo funciona un indexer es por medio de un ejemplo. Usaremos la idea de la tienda de televisores que vimos en un capítulo anterior, pero la modificaremos para este caso. En este ejemplo, tenemos una clase que representa al televisor y otra que representa a la tienda. La tienda tiene en su interior un **ArrayList** donde se guardan los televisores. El indexer, desde la función **Main()**, nos permitirá acceder fácilmente a los televisores de ella.

Empecemos con el código de la clase para los televisores, la cual llamaremos **CTelevisor**; este queda de la siguiente manera:



ADT



Es interesante tener en cuenta que las estructuras de datos abstractas son modelos de estructuras que tienen cierto comportamiento que los define, y que también nos permiten guardar información. La manera en que esta se guarda y se accede a ella depende del tipo de estructura que usemos. Un ejemplo de ADT es la lista ligada, y otro es el árbol binario.

```
class CTelevisor
{
    double costo;
    double impuesto;
    string descripcion;
    // Metodos propios
    // Constructor
    public CTelevisor(string desc, double pcosto)
    {
        // Inicializamos los datos
        costo = pcosto;
        descripcion = desc;
    }
    public void ColocarCosto(double nCosto)
    {
        // Asignamos el nuevo costo
        costo = nCosto;
    }
    public void CalculaImpuesto(double imp)
    {
        // Calculamos el impuesto
        impuesto = costo * imp;
    }
    public void MuestraInformacion()
    {
        // Mostramos la informacion necesaria
        Console.WriteLine("--- Producto televisor");
        Console.WriteLine(descripcion);
        Console.WriteLine("Costo {0}, Impuesto {1}", costo, impuesto);
    }
}
```

El código es muy sencillo y tiene tres datos que son: **costo**, **impuesto** y **descripcion**. El constructor nos permite colocar los valores iniciales para la descripción y también el costo del televisor. Hay una función de interfaz para colocar el costo, y un método para calcular el impuesto pasando como parámetro el porcentaje correspondiente.

Por último, hacemos uso de un método que despliega en pantalla la información que corresponde al televisor.

La siguiente clase es la relacionada con la tienda de televisores, en la que colocaremos el indexer. Su código es el siguiente:

```
class CTiendaTelevisor
{
    // Datos
    private ArrayList misTelevisores= new ArrayList();
    public CTiendaTelevisor()
    {
        // Inicializamos el arreglo interno con datos
        misTelevisores.Add(new CTelevisor("Plasma", 1700.00));
        misTelevisores.Add(new CTelevisor("LED", 2300.50));
        misTelevisores.Add(new CTelevisor("Pantalla Plana", 800.33));
    }
    // Creamos el indexer
    public CTelevisor this[int posicion]
    {
        // Para poder leer la informacion en esa posicion
        get { return (CTelevisor)misTelevisores[posicion]; }
        // Para poder escribir la informacion en esa posicion
        set { misTelevisores[posicion] = value; }
    }
}
```

En la clase colocamos un **ArrayList** llamado **misTelevisores**. Dentro de él incluimos los televisores que pertenecen a la tienda. Para utilizar el indexer correctamente, el ADT debe tener información; de lo contrario, corremos el riesgo de obtener errores de ejecución al tratar de usarlo.

Aprovechamos el constructor de la clase para colocar tres televisores en el **ArrayList**. En general, podemos utilizar el constructor para incluir los datos necesarios en el ADT, pero es posible adicionar otros métodos que nos apoyen a introducir nuevos datos o a eliminarlos.

El siguiente elemento que programaremos es el indexer. Para crear un indexer debemos utilizar **this**, que nos permite referenciar al mismo objeto. El acceso que colocamos es público, y el tipo es igual al tipo que guarda el ADT al que nos interesa acceder; en este caso, es **CTelevisor**.

También estamos haciendo uso de un operador de indexado. Vamos a utilizar una variable de tipo entero para el índice y la llamamos **posición**.

El indexer nos recuerda a las propiedades, ya que tenemos que codificar un **get** y un **set**. Hacemos uso de **get** para colocar el código necesario que nos permita obtener el dato que enviaremos hacia el exterior. Cuando deseamos colocar un dato en una posición en particular, usamos **set**. Es necesario que exista previamente un dato en esa posición, o tendremos una excepción al estar fuera de rango.

En **get**, simplemente regresamos el objeto que se encuentre dentro del **ArrayList** en la posición solicitada. Solo debemos notar que estamos haciendo un **type cast** hacia el tipo **CTelevisor**. Esto se debe a que **ArrayList** regresa el objeto como si fuera de tipo **object**.

En el código de **set** simplemente colocamos el objeto recibido por medio de **value** en la posición solicitada del **ArrayList**.

Ahora colocamos el código correspondiente a la función **Main()**:

```
class Program
{
    static void Main(string[] args)
    {
        // Creamos una tienda de televisores
        CTiendaTelevisor miTienda = new CTiendaTelevisor();
        // Recorremos la tienda como si fuera un arreglo
        for (int n = 0; n < 3; n++)
        {
            miTienda[n].CalculaImpuesto(0.16);
            miTienda[n].MuestraInformacion();
        }
    }
}
```



ARRAYLIST Y SYSTEM.COLLECTIONS

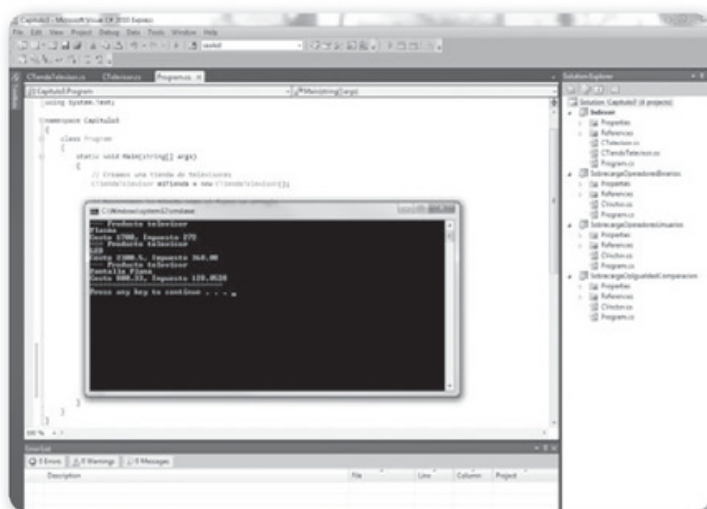


ArrayList es una clase que representa a una estructura de datos de tipo matricial que puede aumentar la cantidad de elementos de forma dinámica en tiempo de ejecución. Para hacer uso de ella tenemos que usar el namespace **System.Collections**. **ArrayList** se encarga de guardar en su interior los objetos como si correspondieran al tipo llamado **Object**.


```
Console.WriteLine("-----");  
// Modificamos los televisores  
// Hay que notar que usamos a miTienda como un arreglo  
// Simplificando el acceso a su array interno de televisores  
miTienda[0] = new CTelevisor("Blanco y negro", 9.0);  
miTienda[1] = new CTelevisor("Descompuesta", 0.50);  
miTienda[2] = new CTelevisor("Sin antena", 8.33);  
// Recorremos la tienda para verificar los cambios  
for (int n = 0; n < 3; n++)  
{  
    miTienda[n].CalculaImpuesto(0.16);  
    miTienda[n].MuestraInformacion();  
}  
}
```

Lo primero que tenemos es la creación de una instancia de la clase **CTiendaTelevisor** a la cual llamamos **miTienda**. Dentro está el **ArrayList** que deseamos recorrer y con el que trabajaremos. Como hemos colocado el indexer, podemos trabajar con el objeto como si fuera un arreglo.

Incluimos un ciclo **for** y, dentro de él, para cada uno de los televisores que tiene **miTienda**, invocamos el método que lleva a cabo el cálculo del impuesto; luego mostramos la información. Es importante notar la forma en que se ha simplificado la sintaxis para lograr esto.

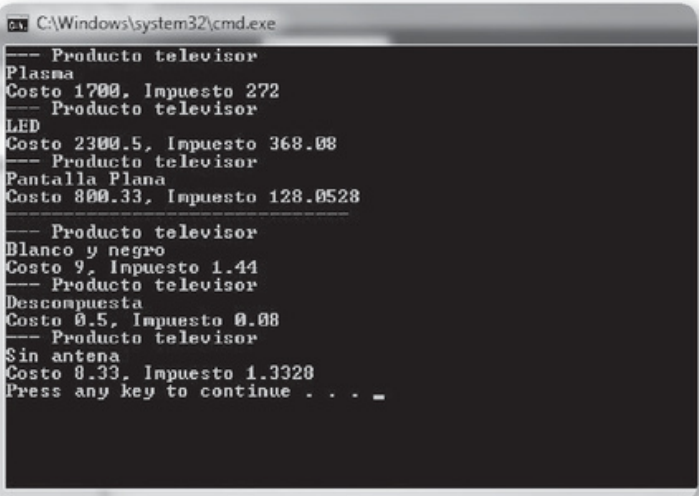


► **Figura 1.**
Podemos observar la primera parte de la ejecución del programa; el indexer trabaja exitosamente.

Debemos tener presente que el `indexer` fue programado de tal modo que podemos leer y escribir información en el ADT que corresponde al objeto. Esto significa que si ya existe ese televisor, entonces podemos modificarlo. No hay que olvidar que, al usar el `set` del `indexer`, solo podemos colocar información en un elemento que ya existe; no va a crear un elemento nuevo.

Modificamos, entonces, los televisores que se encuentran en la tienda; nuevamente, podemos utilizar una sintaxis que es similar a la de un arreglo. Para esto, en cada índice del **ArrayList** colocamos un nuevo objeto con las características que nos interesan.

Por último, verificamos que los cambios se lleven a cabo. Es sencillo hacerlo, ya que únicamente necesitamos un ciclo **for** que calcule el impuesto correspondiente e invoque al método **MuestraInformacion()** del televisor.



```
CA\Windows\system32\cmd.exe
--- Producto televisor
Plasma
Costo 1700, Impuesto 272
--- Producto televisor
LED
Costo 2300.5, Impuesto 368.08
--- Producto televisor
Pantalla Plana
Costo 800.33, Impuesto 128.0528
-----
--- Producto televisor
Blanco y negro
Costo 9, Impuesto 1.44
--- Producto televisor
Descompuesta
Costo 0.5, Impuesto 0.08
--- Producto televisor
Sin antena
Costo 0.33, Impuesto 1.3328
Press any key to continue . . . _
```

► **Figura 2.** Esta es la ejecución completa, donde podemos verificar que los cambios se han llevado a cabo.



OPERADORES UNUARIOS

Sabemos que el lenguaje de programación C# permite realizar la sobrecarga de numerosos operadores unarios. De esta forma, también debemos tener en cuenta que algunos de ellos ya han sido sobrecargados como ejemplo en este capítulo; entre ellos encontramos los siguientes: `+`, `-`, `!`, `~`, `++`, `--`, **true**, **false**. Si tenemos experiencia en el desarrollo con el lenguaje de programación C++, debemos ser cuidadosos, pues la sobrecarga de operadores es diferente en ese lenguaje.

Este ejemplo nos muestra cómo, por medio del indexer, podemos acceder fácilmente desde el exterior de la clase a un ADT. También simplifica la sintaxis por utilizar haciendo que el programa resulte más fácil de entender y depurar.

Sobrecarga de operadores

Cuando programamos con C#, estamos acostumbrados a utilizar tipos como los enteros y los flotantes. Con ellos podemos llevar a cabo una serie de operaciones, como multiplicar dos enteros o efectuar una comparación entre variables dentro de un **if**. Estas operaciones pueden hacerse gracias a los **operadores**. Estos pueden ser aritméticos, como en el caso de: **+**, **-**, *****, **/**, **%**; de comparación o relacionales, como **==**, **!=**, **<**, **>**, etc.; y, por último, lógicos, como **&&**, **||**, **!**.

Para poder trabajar, los operadores necesitan información, que se guarda en los **operandos**. Hay operadores que precisan dos operandos, y entonces se los conoce como **binarios**. La mayoría de los operadores aritméticos son operadores binarios.

```
resultado = valor*impuesto;
```

En este ejemplo, el operador es *****, el primer operando es la variable **valor** y el segundo es la variable **impuesto**. El operador lleva a cabo un proceso (en este caso, la multiplicación), y el resultado o la evaluación de este proceso queda guardado en la variable **resultado**.

Algunos operadores necesitan únicamente un operando, y se los conoce como operadores **unarios**.



SOBRECARGA



La sobrecarga es una técnica de programación que nos permite tener más de una versión de una función o método. Las funciones sobrecargadas poseen el mismo nombre y se distinguen entre sí por los parámetros que utilizan. La sobrecarga aumenta la flexibilidad del código. No solo los métodos se pueden sobrecargar, también algunos operadores lo admiten.


```
edad = edadAnterior++;
```

En este caso, el operador es **++**, y vemos que solo precisa un operando para trabajar; el operando es la variable **edadAnterior**.

Por lo general, podemos usar estos operandos con los tipos más conocidos, pero sería una gran ventaja para nosotros poder definir este tipo de operaciones directamente para nuestros objetos y las clases que tengamos. Resultaría más cómodo programar usando los operadores que haciendo llamadas a funciones.

Por ejemplo, podemos tener un código de la siguiente manera:

```
if( CTelevisores.ComparaTelevisor(miTV,tuTV)==0)
{
    ...
}
```

En este caso, estamos comparando dos televisores por medio de un método estático de la clase **CTelevisores**. Recordemos que el código es fácil de entender y nada complicado, pero sería mejor si pudiéramos hacer algo como lo siguiente:

```
if( miTV==tuTV)
{
    ...
}
```

Como podemos observar, aunque los códigos son equivalentes, debemos tener en cuenta que la segunda versión es más fácil de usar, es intuitiva y produce código más limpio.

Esto quiere decir que podemos emplear los operadores para llevar a cabo operaciones con nuestras propias clases. Para lograr esta tarea, haremos uso de una excelente técnica de programación conocida como **sobrecarga de operadores**.

Para entender el funcionamiento de la sobrecarga, podemos pensar en los operadores como si fueran funciones. Estas funciones, en vez de tener un nombre normal, tienen el nombre del operador, y sus parámetros son los operadores. En pseudocódigo y a manera de ejemplo, podemos tener lo que vemos a continuación:


```
resultado = valor*impuesto;  
resultado = Multiplicacion(valor, impuesto);
```

Como podemos notar, conceptualmente, son equivalentes, solo que lo escribimos de manera diferente. Si pensamos en los operadores como funciones, entonces podemos entender fácilmente la sobrecarga de los operadores.

Sobrecarga de operadores binarios

Empecemos a construir una aplicación en la cual llevaremos a cabo la sobrecarga de algunos operadores binarios. De esta forma, podremos acceder a utilizar los operadores $+$ y $-$ con una clase que nosotros mismos nos encargaremos de construir.

Para este ejemplo, construiremos una clase **CVector** que representará a un vector matemático, y realizaremos las operaciones con él. Crear más sobrecargas requerirá las mismas técnicas que aquí aprenderemos y podremos implementarlas en nuestras propias clases.

Los vectores

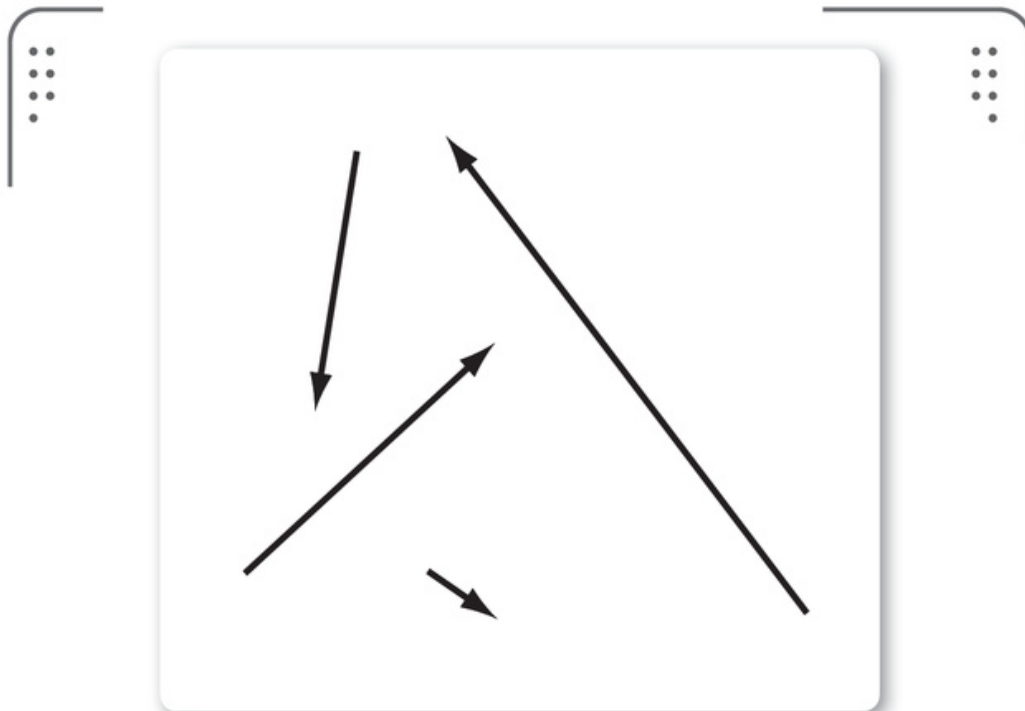
Los **vectores** son entidades matemáticas y geométricas que tienen una dirección y una magnitud. Podemos utilizarlos para representar cualidades físicas como las fuerzas. Cuentan con una gran cantidad de aplicaciones en ingeniería y también en ciencias.



SUMA Y RESTA DE VECTORES



Es importante tener en cuenta que la suma y la resta de vectores son operaciones muy sencillas de realizar. Pensando en esto, recordemos que para sumar, simplemente sumamos todo.



► **Figura 3.** En esta figura podemos observar vectores, los cuales son representados por medio de flechas.

Con los vectores podemos llevar a cabo diversas operaciones: es posible sumarlos y restarlos, y existen diferentes tipos de productos. En nuestra aplicación, llevaremos a cabo estas operaciones aprovechando la sobrecarga de operadores.

Los vectores poseen una propiedad que nos permite saber cuán grandes son; esta propiedad recibe el nombre de **magnitud**. Cuando representamos gráficamente un vector, la magnitud está relacionada con la longitud de la flecha que usamos para mostrar el vector.

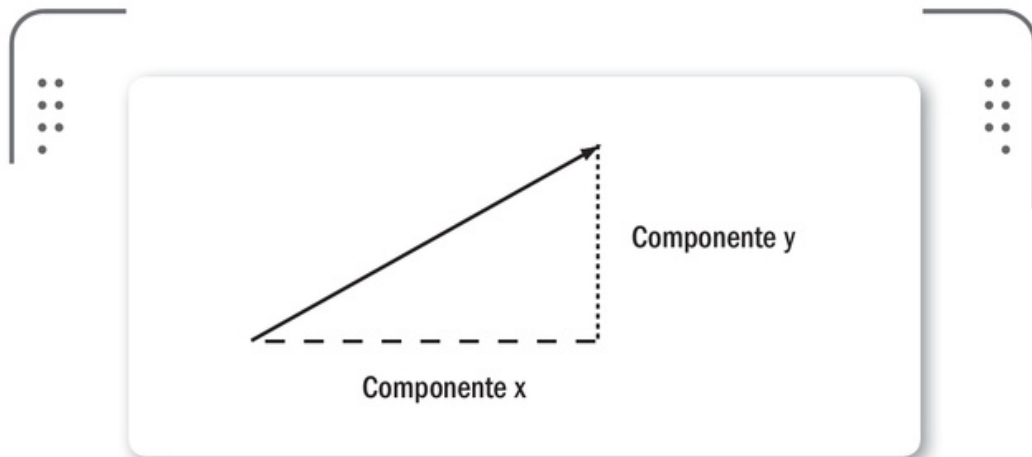


OPERADORES BINARIOS SOBRECARGADOS



En el lenguaje de programación C# podemos acceder a sobrecargar los operadores binarios que mostramos a continuación: `+`, `-`, `*`, `/`, `%`, `&`, `|`, `^`, `<<`, `>>`. En este sentido no hay que olvidar que se necesitan dos operandos para poder trabajar correctamente. Los operandos pueden ser vistos como los parámetros a utilizar por el operador y que contienen los valores.

En nuestro ejemplo, trabajaremos con vectores en dos dimensiones. Estos pueden ser representados por medio de dos componentes: **X** e **Y**. La **Figura 4** nos muestra este caso.



► **Figura 4.** Este vector ha sido descompuesto en sus dos componentes. Esto nos permite trabajar más fácilmente con él.

Como ya conocemos los elementos básicos de los vectores, podemos proceder a crear nuestra aplicación.

Aplicación con sobrecarga de operadores binarios

Empecemos por crear una aplicación nueva a la cual adicionamos una clase llamada **CVector**. La primera parte del código de esta clase es la que mostramos a continuación:

```
class CVector
{
    // Datos
    private double miX;
    private double miY;
    public CVector(double x, double y)
    {
        // Asignamos los valores
```



```
        miX = x;
        miY = y;
    }
    // Propiedades para los datos
    public double X
    {
        get { return miX; }
        set { miX = value; }
    }
    public double Y
    {
        get { return miY; }
        set { miY = value; }
    }
    // Mostramos el vector
    public void Mostrar()
    {
        Console.WriteLine("{0},{1}", miX, miY);
    }
    // Override de ToString
    public override string ToString()
    {
        return string.Format("{0},{1}", miX, miY);
    }
}
```

En primer lugar, tenemos dos variables, una para cada uno de los componentes del vector. En el constructor inicializamos el vector con los valores pasados por los parámetros. A continuación, creamos un par de propiedades, una para cada componente del vector.



OPERADORES DE COMPARACIÓN



Debemos tener en cuenta que los siguientes operadores de comparación se pueden sobrecargar en el lenguaje de programación C#: ==, !=, <, >, <=, >=. No hay que olvidar que estos operadores necesitan ser sobrecargados en parejas, pues no es posible hacerlo solo con uno. El compilador indicará un error si no se encuentra la pareja también sobrecargada.

Para desplegar al vector, tenemos un método llamado **Mostrar()**, que simplemente imprime en la consola los valores de los componentes. Proveemos también de un **override** al método **ToString()**, con el que regresamos una cadena con formato que contiene la información de los componentes.

Con los métodos básicos de la clase codificados, podemos empezar a programar la sobrecarga para el operador **+**:

```
// Sobrecarga del operador +
public static CVector operator +(CVector v1, CVector v2)
{
    // Calculamos el resultado
    double tempX = v1.X + v2.X;
    double tempY = v1.Y + v2.Y;
    // Regresamos el vector resultante
    return new CVector(tempX, tempY);
}
```

Para llevar a cabo la sobrecarga, creamos un método con acceso público y estático. Esto es necesario para que pueda ser invocado desde el exterior de la clase, y no necesite de una instancia para llevar a cabo la operación. Luego, indicamos el tipo de retorno en nuestro

Estos son representados por los parámetros que estamos colocando en el método. El primer parámetro representa al operando a la izquierda del operador, y el segundo es el operando que se encuentra en el lado derecho del operador, esto es:

```
v1 + v2
```

El código dentro del método lleva a cabo esta acción. Tomamos una variable de trabajo para el componente **X** del vector resultante y hacemos lo mismo para el componente **Y**. Luego, simplemente creamos una nueva instancia de tipo **CVector** con estos valores, y este es el objeto que regresamos como resultado de la operación.

Es posible optimizar el código para hacerlo más eficiente. Por ejemplo, podemos llevar a cabo la sobrecarga del operador - de la forma que vemos en el siguiente ejemplo:

```
// Sobrecarga del operador -  
public static CVector operator -(CVector v1, CVector v2)  
{  
    // Calculamos el resultado y lo regresamos  
    return new CVector(v1.X - v2.X, v1.Y - v2.Y);  
}
```

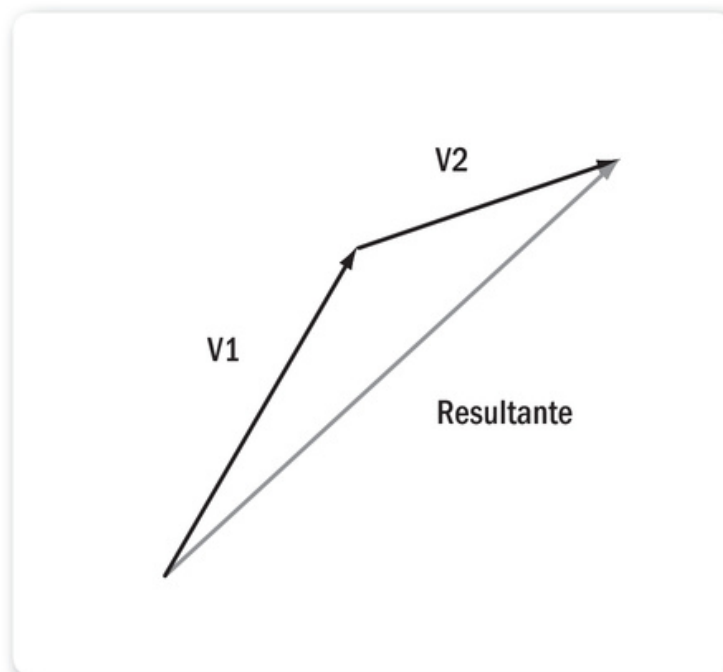
Con esto tenemos todo lo necesario para la clase **CVector** hasta el momento. Ahora podemos probar que funcione al intentar hacer sumas y restas de vectores. Esto se realiza en la función **Main()**:

```
static void Main(string[] args)  
{  
    // Vectores necesarios  
    CVector v1 = new CVector(5, 3);  
    CVector v2 = new CVector(2, 1);  
    CVector v3;  
    // Hacemos la suma  
    v3 = v1 + v2;  
    // Verificamos la suma  
    Console.WriteLine("{0} + {1} = {2}", v1, v2, v3);  
}
```

```
// Hacemos la resta  
v3 = v1 - v2;  
// Verificamos la resta  
Console.WriteLine("{0} - {1} = {2}", v1, v2, v3);  
}
```

Vamos a utilizar tres objetos de tipo **CVector**; dos de ellos nos sirven como operandos, en tanto que el tercero es utilizado para guardar el vector resultante de la operación. Con el primer objeto, llamado **v1**, definimos un vector con componentes **(5,3)**; el segundo vector, llamado **v2**, tiene por componentes **(2,1)**.

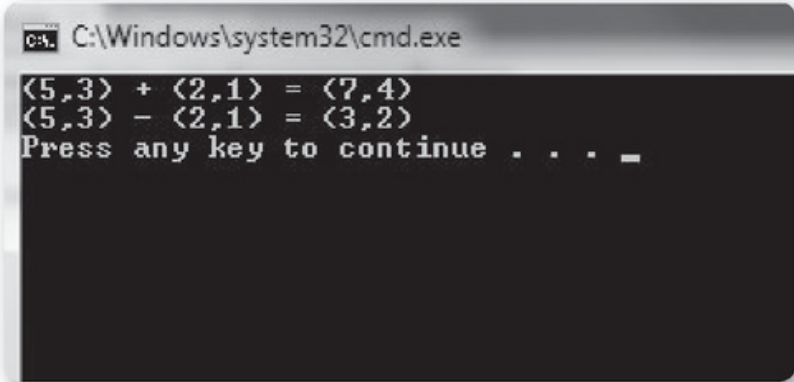
Continuamos probando la sobrecarga del operador al realizar una suma de vectores. Debemos tener en cuenta que esta es una suma que ya se lleva a cabo sobre una clase que nosotros hemos definido anteriormente. La operación es intuitiva y fácil de comprender, el código se ha simplificado. El resultado de la suma se guarda en **v3**, y procedemos a imprimir el resultado en la consola.



► **Figura 5.** Esta figura nos muestra la representación gráfica de la suma de vectores.

Ahora probamos la resta de los vectores. Sabemos que se trata de una operación que también se hace usando directamente el operador sobrecargado `-`; luego, imprimimos los valores para verificar que la operación se lleve a cabo correctamente.

Ahora podemos proceder a realizar la compilación y ejecución de la aplicación, y veremos que, efectivamente, podemos efectuar operaciones con nuestras propias clases.



A screenshot of a Windows command prompt window titled "C:\Windows\system32\cmd.exe". The window displays the following text: `<5,3> + <2,1> = <7,4>`, `<5,3> - <2,1> = <3,2>`, and `Press any key to continue . . . _`. The text is white on a black background.

► **Figura 6.** Podemos comprobar que la sobrecarga de los operandos `+` y `-` ha funcionado correctamente.

➤ Sobrecarga de operadores unuarios

Es el momento de enfocarnos en los operadores unuarios, recordemos que se trata de aquellos que solo necesitan de un operando para poder trabajar. Visto desde la perspectiva de un método, únicamente precisamos un parámetro. Como ejemplo, llevaremos a cabo la sobrecarga de dos operadores unuarios: `++` y `--`.

Cuando efectuamos la sobrecarga de cualquier operador, debemos pensar cuál sería la regla para él dentro del contexto de la clase que estamos ocupando. Sabemos que en los valores numéricos que normalmente usamos, cuando se utiliza `++`, se incrementa en **1** el valor, y cuando se hace uso de `--`, se decrementa en **1**. Pero ¿cómo funcionan

`++` y `--` para un vector? Esta pregunta es la que debemos contestar de acuerdo con el contexto de nuestra aplicación, la clase y la lógica necesaria. Para este ejemplo, incrementaremos en **1** cada componente cuando usamos `++` con el vector, y decrementaremos en **1** cada componente cuando usemos `--`. Pero también podríamos haber tenido una implementación alternativa, e incrementar en **1** la magnitud con `++` y decrementar en **1** la magnitud con `--`. Por eso es importante definir y conocer las reglas de operación de la sobrecarga para nuestra clase antes de empezar a programar.

La sobrecarga es sencilla, y con ella, la clase **CVector** queda de la forma que vemos en el siguiente código:

```
class CVector
{
    // Datos
    private double miX;
    private double miY;
    public CVector(double x, double y)
    {
        // Asignamos los valores
        miX = x;
        miY = y;
    }
    // Propiedades para los datos
    public double X
    {
        get { return miX; }
        set { miX = value; }
```



SOBRECARGA POR PARES



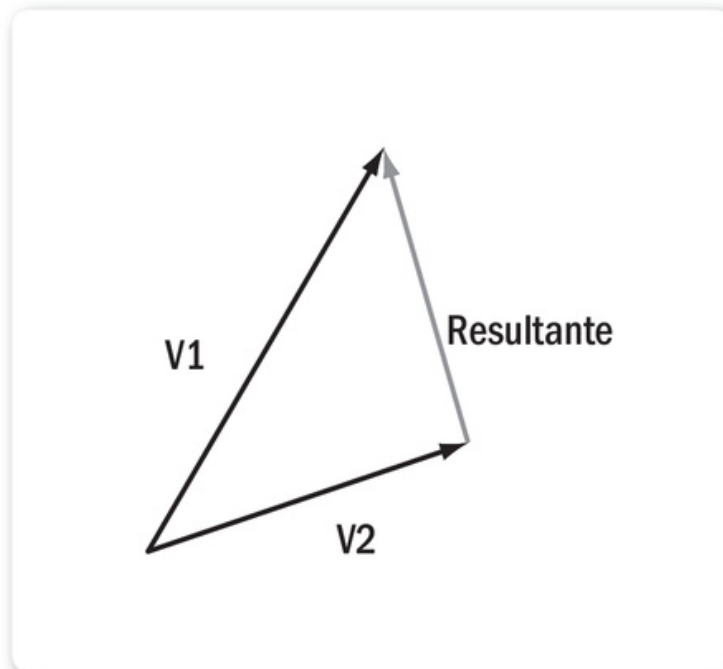
Cuando tenemos que realizar la sobrecarga de un operador, a veces sucede que es necesario también sobrecargar uno complementario. Por ejemplo, al sobrecargar el operador `++`, siempre debemos hacer lo propio con `--`. Por eso, es conveniente verificar si nuestro operador tiene un par, ya que el compilador mostrará un error si no lo implementamos.

```
}  
public double Y  
{  
    get { return miY; }  
    set { miY = value; }  
}  
// Mostramos el vector  
public void Mostrar()  
{  
    Console.WriteLine("{0},{1}", miX, miY);  
}  
// Override de ToString  
public override string ToString()  
{  
    return string.Format("{0},{1}", miX, miY);  
}  
// Sobrecarga del operador +  
public static CVector operator +(CVector v1, CVector v2)  
{  
    // Calculamos el resultado y lo regresamos  
    return new CVector(v1.X + v2.X, v1.Y + v2.Y);  
}  
// Sobrecarga del operador -  
public static CVector operator -(CVector v1, CVector v2)  
{  
    // Calculamos el resultado y lo regresamos  
    return new CVector(v1.X - v2.X, v1.Y - v2.Y);  
}  
// Sobrecarga operador ++  
public static CVector operator ++(CVector v1)  
{  
    // Calculamos el resultado y lo regresamos  
    return new CVector(v1.X + 1, v1.Y + 1);  
}  
// Sobrecarga operador --  
public static CVector operator --(CVector v1)  
{
```

```
// Calculamos el resultado y lo regresamos  
return new CVector(v1.X - 1, v1.Y - 1);  
}  
}
```

El código de la sobrecarga para el operador **++** es simple. Empezamos indicando que el método tiene acceso público y es estático. Este método regresa un objeto de tipo **CVector**; no debemos olvidarnos de colocar la palabra clave **operator** para indicar que estamos llevando a cabo una sobrecarga; inmediatamente después, señalamos que el operador por sobrecargar es **++**. Recibimos un único parámetro, que es el vector por incrementar. En el interior del método incrementamos los valores de los componentes y con esos valores creamos una nueva instancia de la clase **CVector**, la cual es regresada por el método.

De modo similar, tenemos el código para la sobrecarga del operador **--**, para el cual decrementamos los valores de los componentes y regresamos una instancia de tipo **CVector** creada con estos valores.



► **Figura 7.** Esta figura nos muestra la representación gráfica que corresponde a la resta de vectores.

Ahora debemos probar cómo funcionan nuestras sobrecargas a los operadores unuarios. Esto se hace dentro de la función **Main()**:

```
static void Main(string[] args)
{
    // Vectores a utilizar
    CVector v1 = new CVector(5, 3);
    // Incrementamos el vector
    Console.WriteLine("Antes {0}", v1);
    v1++;
    Console.WriteLine("Despues {0}", v1);
    Console.WriteLine("-----");
    // Decrementamos el vector
    Console.WriteLine("Antes {0}", v1);
    v1--;
    Console.WriteLine("Despues {0}", v1);
}
```

Nos encargamos de realizar la creación de un vector llamado **v1**, que tiene como componentes **(5,3)**. Mostramos el vector antes de hacer el incremento, para poder llevar a cabo una comparación. Realizamos el incremento haciendo uso del operador sobrecargado. Luego, otra vez imprimimos el vector y comprobamos que, efectivamente, se realizó el incremento en los componentes.

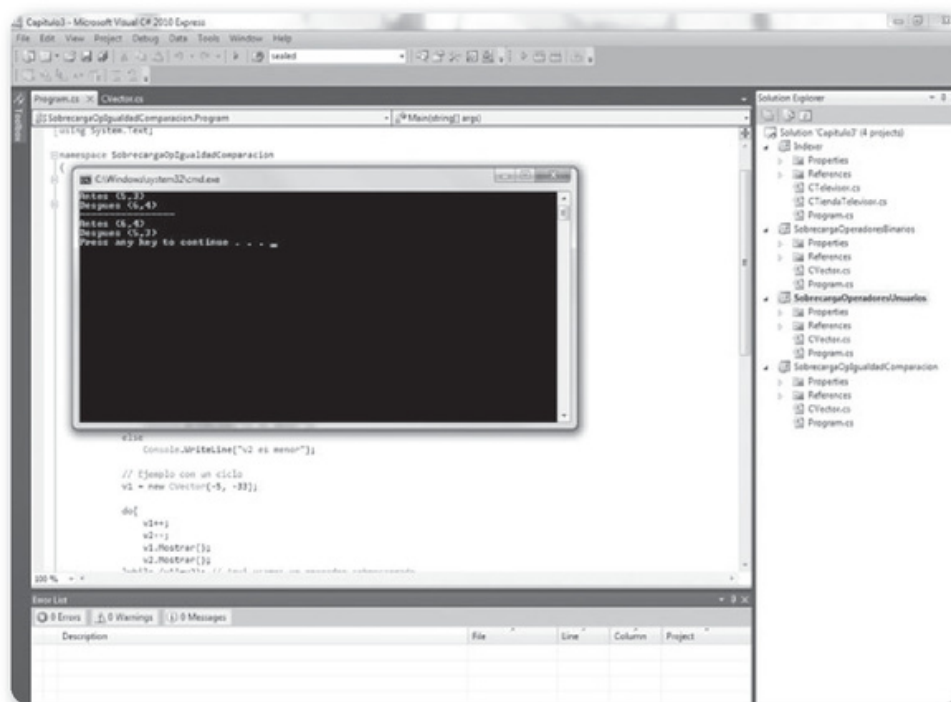
De igual modo, verificamos que el operador **--** funcione correctamente. Mostramos el vector antes y después del decremento, para poder ver los cambios en los componentes. Ahora podemos compilar la aplicación y observar su comportamiento.



OPERADORES SOBRECARGADOS



Es importante señalar que los siguientes operadores también se pueden sobrecargar en el lenguaje de programación C#: **+=**, **-=**, ***=**, **/=**, **%=**, **&=**, **|=**, **^=**, **<<=**, **>>=**. Estos nos brindan ayuda, principalmente, mientras nos damos a la tarea de realizar operaciones aritméticas y también de asignación. Si deseamos investigar más sobre este tema, es posible encontrar información, ejemplos y, por supuesto, documentación sobre ellos en el sitio web de MSDN.



► **Figura 8.** En esta imagen podemos observar el funcionamiento de los operadores que han sido sobrecargados.

➤ Sobrecarga de operadores de igualdad y comparación

Otro tipo de operadores que nos resulta muy útil poder sobrecargar son los de igualdad y de comparación. Estos nos permiten conocer la relación que existe entre dos objetos. Por ejemplo, podemos saber si dos objetos son iguales, uno es mayor o menor que el otro, etc.

Si sobrecargamos este tipo de operadores, podremos usarlos directamente en lugares donde se puedan evaluar expresiones que los utilicen, como en **if**, **while**, **do while**, **for**, etc. Esto ayuda a que el código que utilice nuestras clases sea más sencillo e intuitivo.

Como en cualquier sobrecarga que llevemos a cabo, nosotros somos responsables de definir las reglas de lógica sobre las cuales, en este caso, se llevará a cabo la comparación o la igualdad. Estas reglas dependen de la clase que estemos implementando y de la lógica de la aplicación.

En nuestro ejemplo, tenemos que usar la magnitud del vector para realizar la comparación y la igualdad. Desde el punto de vista matemático, esto no es estrictamente correcto, pero es la regla que se ha seleccionado para simplificar el ejemplo. La clase **CVector** tendrá algunos cambios y varias adiciones, donde nos encargaremos de usar lo que hemos aprendido hasta el momento.

Empecemos con algunas modificaciones al código que ya conocemos:

```
// La clase ahora tambien implementa a IComparable
class CVector: IComparable
{
    // Datos
    private double miX;
    private double miY;
    private double miM;
    public CVector(double x, double y)
    {
        // Asignamos los valores
        miX = x;
        miY = y;
        // Calculamos la magnitud
        Magnitud();
    }
    // Propiedades para los datos
    public double X
    {
        get { return miX; }
        set { miX = value; }
    }
    public double Y
    {
        get { return miY; }
        set { miY = value; }
    }
    public double M
    {
        get { return miM; }
```

```
        set { miM = value; }
    }
    // Mostramos el vector
    public void Mostrar()
    {
        Console.WriteLine("{0},{1} m={2}", miX, miY, miM);
    }
    // Calculamos la magnitud
    private void Magnitud()
    {
        miM = Math.Sqrt(miX * miX + miY * miY);
    }
    // Override de ToString
    public override string ToString()
    {
        return string.Format("{0},{1} m={2}", miX, miY, miM);
    }
    // Sobrecarga del operador +
    public static CVector operator +(CVector v1, CVector v2)
    {
        // Calculamos el resultado y lo regresamos
        return new CVector(v1.X + v2.X, v1.Y + v2.Y);
    }
    // Sobrecarga del operador -
    public static CVector operator -(CVector v1, CVector v2)
    {
        // Calculamos el resultado y lo regresamos
        return new CVector(v1.X - v2.X, v1.Y - v2.Y);
    }
    // Sobrecarga operador ++
    public static CVector operator ++(CVector v1)
    {
        // Calculamos el resultado y lo regresamos
        return new CVector(v1.X + 1, v1.Y + 1);
    }
    // Sobrecarga operador --
    public static CVector operator --(CVector v1)
```



```
{  
    // Calculamos el resultado y lo regresamos  
    return new CVector(v1.X - 1, v1.Y - 1);  
}
```

El primer cambio que encontramos es la implementación de **Comparable**. Hacemos uso de esta interfaz pues llevaremos a cabo comparaciones de nuestros objetos, y lo mejor es emplear lo que ya hemos aprendido. Podríamos haberlo hecho sin esta interfaz, pero de esta forma es más profesional y correcto.

En los datos de la clase **CVector** tenemos uno nuevo llamado **miM**, que representa la magnitud del vector. En el constructor también tenemos un cambio: hemos adicionado una invocación al método **Magnitud()**, el cual veremos más adelante.

También agregamos una nueva propiedad llamada **M**, con la cual podemos acceder al dato **miM** desde el exterior. La propiedad es de entrada y salida. Tanto en el método **Mostrar()** como en **ToString()** hemos adicionado **miM** a la información que se desplegará o que se guardará en la cadena con formato. El método **Magnitud()** calcula la magnitud del vector tomando en cuenta sus componentes y guardando el resultado en **miM**.

Empecemos ahora con el código necesario para hacer la sobrecarga del operador de igualdad **==** y el de desigualdad **!=** :

```
// Sobrecarga operador == y !=  
public override bool Equals(object obj)  
{  
    if (obj is CVector)  
    {  
        CVector temp = (CVector)obj;  
        if (miM == temp.M)  
            return true;  
    }  
    return false;  
}  
public override int GetHashCode()  
{
```

```
        return base.GetHashCode();
    }
    public static bool operator ==(CVector v1, CVector v2)
    {
        return v1.Equals(v2);
    }
    public static bool operator !=(CVector v1, CVector v2)
    {
        return !(v1.Equals(v2));
    }
}
```

Para sobrecargar los operadores `==` y `!=` nos apoyaremos en el método **Equals()**. Llevamos a cabo un **override** de él para poder colocar nuestro código. Este método regresa un **bool**. Si el valor del **bool** es **true**, significa que los objetos comparados son iguales; en caso contrario, regresa un **false**. Recibimos como parámetro el objeto que vamos a comparar contra la instancia que está ejecutando el método. Es necesario tener en cuenta que, por seguridad, lo primero que hacemos es verificar si el objeto que estamos recibiendo es del tipo **CVector**; de no ser así, regresamos un **false**, ya que de todas formas los objetos no serían iguales. Una vez comprobado esto, hacemos uso de **type cast** para dejar el objeto **obj** en términos de **CVector**. Llevamos a cabo la comparación de las magnitudes de ambos objetos y, si las dos son iguales, procedemos a regresar un **true**.

La sobrecarga de los operadores ahora es sencilla. Empezamos sobrecargando el operador `==`. Como siempre, colocamos el acceso de tipo público y el método es estático. El operador `==` regresa un **bool** dependiendo de la forma en que se evalúe la expresión. No hay que



OVERRIDE DE EQUALS



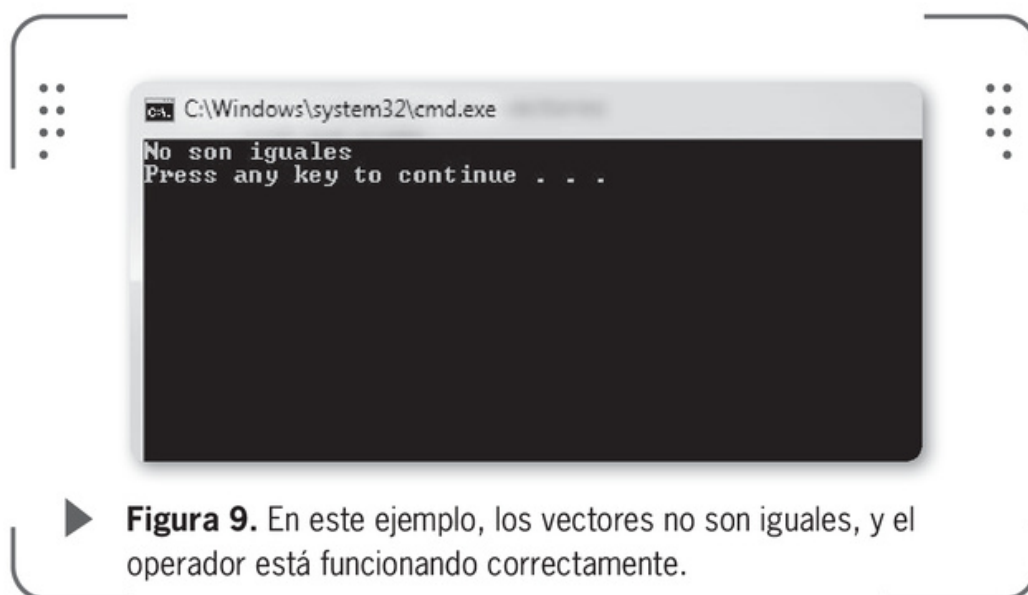
Sabemos que **Equals** nos permite realizar la comparación de dos objetos para saber si son iguales o diferentes. Debemos tener en cuenta que en su interior procedemos a colocar nuestra propia lógica y las reglas de comparación que deseamos utilizar. Si los objetos son iguales, debemos regresar el valor **true**, en tanto que si son diferentes, regresamos **false**. El objeto por comparar es recibido con tipo **object**, por lo que es necesario hacer un **type cast**.

olvidar usar la palabra clave **operator**. Debemos tener en cuenta que nos encargamos de recibir dos parámetros que son los operandos para comparar. En el interior del método, simplemente hacemos uso de **Equals** y aprovechamos su valor de retorno.

El operador **!=** es sobrecargado de forma similar. Cuando usamos **!=**, lo que nos interesa saber es si los operandos son diferentes. Esto significa que el operador regresa un valor **true** cuando los operandos son distintos, y **false** si son iguales. Podemos darnos cuenta de que el código es parecido al anterior; también hacemos uso de **Equals** para llevar a cabo la comparación, pero la diferencia es que en este caso estamos negando el valor antes de regresarlo al exterior.

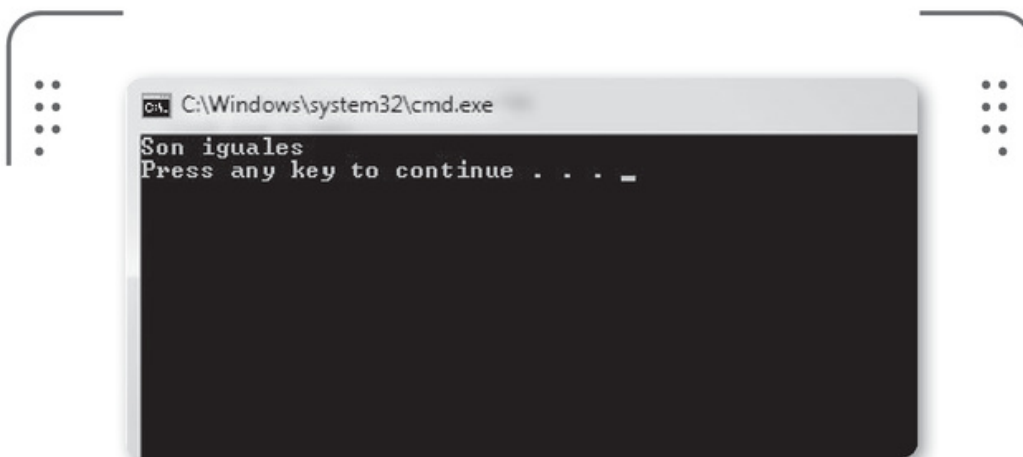
Para probar los operadores, crearemos una comparación sencilla dentro de la función **Main()**, como vemos a continuación:

```
static void Main(string[] args)
{
    CVector v1 = new CVector(5, 3);
    CVector v2 = new CVector(5, 33);
    if (v1 == v2)
        Console.WriteLine("Son iguales");
    else
        Console.WriteLine("No son iguales");
}
```



► **Figura 9.** En este ejemplo, los vectores no son iguales, y el operador está funcionando correctamente.

La comparación de los vectores se efectúa por medio del operador y estamos utilizándolo en un **if**. En este ejemplo, es obvio que los vectores son diferentes, pero también podemos experimentar con componentes iguales para verificar que todo funcione correctamente.



► **Figura 10.** En esta imagen nos encargamos de modificar los vectores para realizar la comprobación de un caso diferente.

➤ Sobrecarga de los operadores < y >

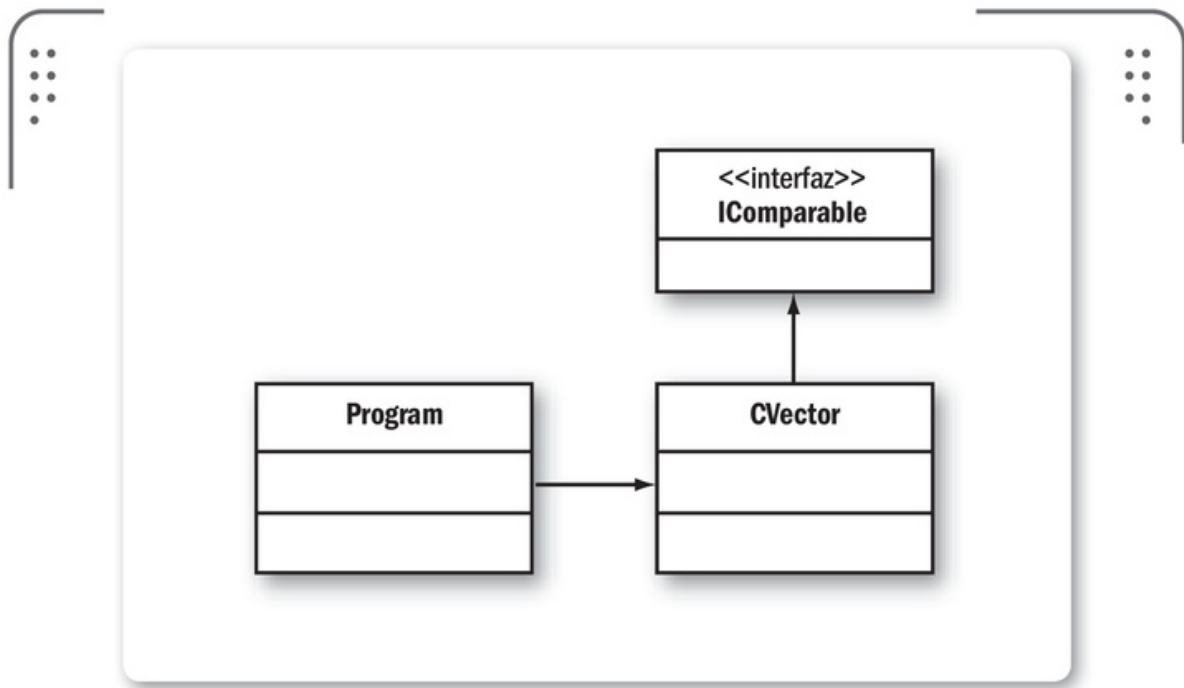
Los operadores que vamos a sobrecargar son < y >. Nos permiten comparar para saber la relación que existe entre los objetos; saber si un objeto es mayor o menor. En nuestro caso, la regla de comparación usará la magnitud del vector para saber cuál es mayor o menor.



EL USO DE IS



En el lenguaje de programación C# existe un operador conocido como **is**; debemos tener en cuenta que este operador no puede ser sobrecargado y sirve para saber, en tiempo de ejecución, si una instancia es compatible o es de un tipo de clase en particular. Nos entrega el valor **true** si esto se cumple y **false** en caso de que no sean compatibles.



► **Figura 11.** En este diagrama podemos darnos cuenta de que **CVector** implementa la interfaz **IComparable** en el ejemplo de la sobrecarga de los operadores de comparación.

Si recordamos lo visto en un capítulo anterior, podemos hacer uso de la implementación de **IComparable** con el método **CompareTo()**, el cual nos ayudará a simplificar el código de la sobrecarga de operadores. Este método regresa el valor **0** si los dos objetos son iguales, **1** si la instancia del objeto ejecutando el método es mayor que el recibido por parámetro, y **-1** si la instancia del objeto ejecutando el método es menor al objeto recibido como parámetro:

```
public int CompareTo(object obj)
{
    if (obj is CVector)
    {
        CVector temp = (CVector)obj;
        if (miM > temp.M)
            return 1;
        if (miM < temp.M)
```

```
        return -1;
    else
        return 0;
    }
    else
        throw new ArgumentException("No se estan comparando objetos de
        tipo CVector");
    }
    public static bool operator <(CVector v1, CVector v2)
    {
        if (v1.CompareTo(v2) < 0)
            return true;
        else
            return false;
    }
    public static bool operator >(CVector v1, CVector v2)
    {
        if (v1.CompareTo(v2) > 0)
            return true;
        else
            return false;
    }
}
```

Como vemos, empezamos con el método denominado **CompareTo()**. En primer lugar, nos encargamos de realizar la verificación para saber si el objeto por comparar corresponde al tipo **CVector**. Si no lo es, lo que debemos hacer es lanzar una excepción del tipo **ArgumentException** indicando que la comparación no se lleva a cabo con un objeto de tipo **CVector**. De la forma habitual, procedemos a realizar el **type cast**

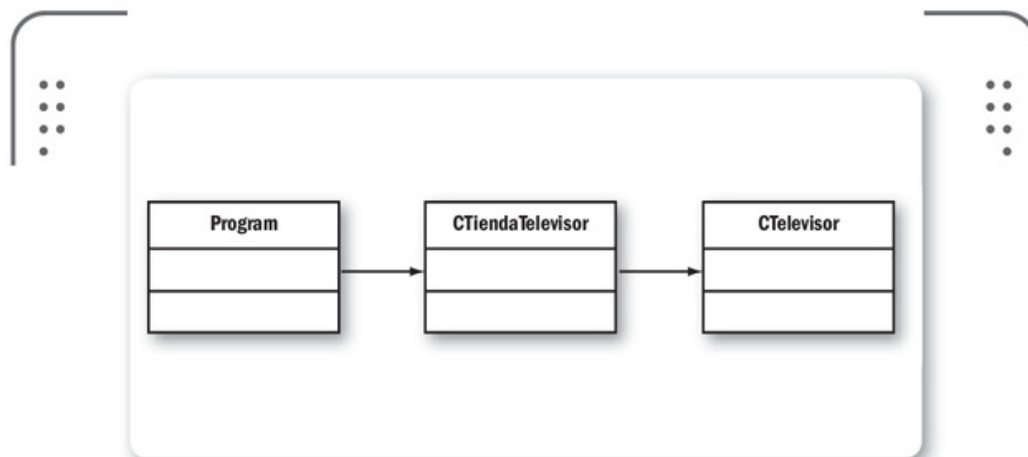


FUERA DE RANGO EN EL INDEXER



Es importante tener en cuenta que el indexer se encarga de manejar un rango de forma similar a como lo hace un arreglo. Este rango, que va de 0 a la cantidad de elementos menos uno, nos indica cuáles son los valores válidos para el índice correspondiente. Si le colocamos al índice un valor fuera de este rango, obtendremos un error en tiempo de ejecución.

al objeto recibido como parámetro, y aplicamos la lógica necesaria para comparar los objetos y, de esta forma, dar el valor de retorno correspondiente. Con esto, el código de las sobrecargas es más sencillo y fácil de implementar.



► **Figura 12.** Este pequeño diagrama nos muestra las relaciones que existen entre las clases del ejemplo de los indexers

Los operadores de comparación se encargan de regresar un valor de tipo **bool**. Si la expresión que se evalúa resulta ser verdadera, tenemos que regresar el valor **true**; en caso contrario, se regresa **false**. En la sobrecarga del operador **<** tenemos dos parámetros: el primero es el operando que va del lado izquierdo del operador, mientras que el segundo es el operando que va del lado derecho. El código es sencillo y, simplemente, nos apoyamos en **CompareTo()**. Como sabemos que el método **CompareTo()** regresa un valor menor que **0** cuando **v1** es menor que **v2**, aprovechamos esto para regresar el valor **true**; en caso contrario, regresamos el valor **false**.

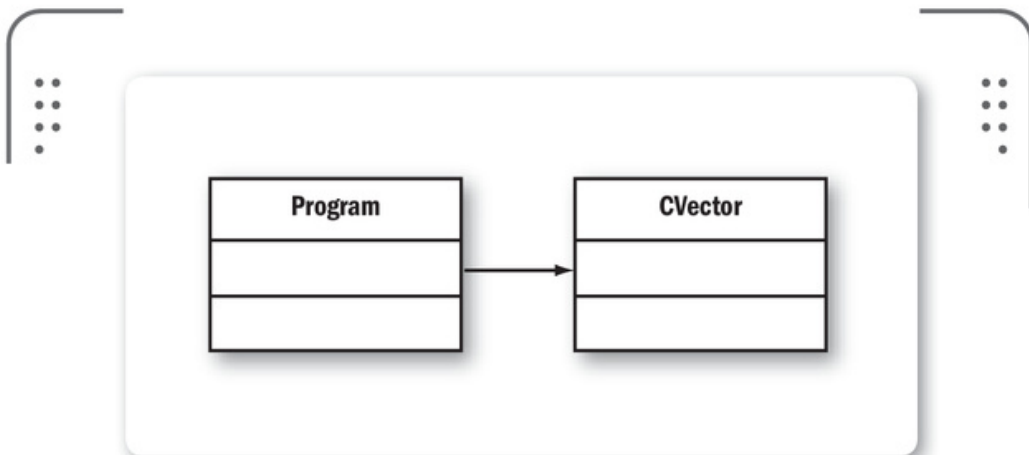


SINTAXIS DEL INDEXER



Debemos tener en cuenta que el indexer nos permite tener una sintaxis similar a un arreglo, pero recordemos que no se trata de un arreglo. No podemos invocar los métodos que usualmente utilizamos con los arreglos, por lo que si deseamos saber la cantidad de elementos en el indexer, tendremos que proceder a implementar este método nosotros mismos.

Debemos tener en cuenta que algo similar ocurre en la sobrecarga del operador `>`. En este caso, sabemos que **CompareTo()** regresa un valor mayor que **0** cuando **v1** es mayor que **v2**. El resto de la lógica es similar a la que vimos en el caso del operador anterior.



► **Figura 13.** En esta figura vemos la relación entre las clases de los ejemplos de la sobrecarga de operadores binarios y unarios.

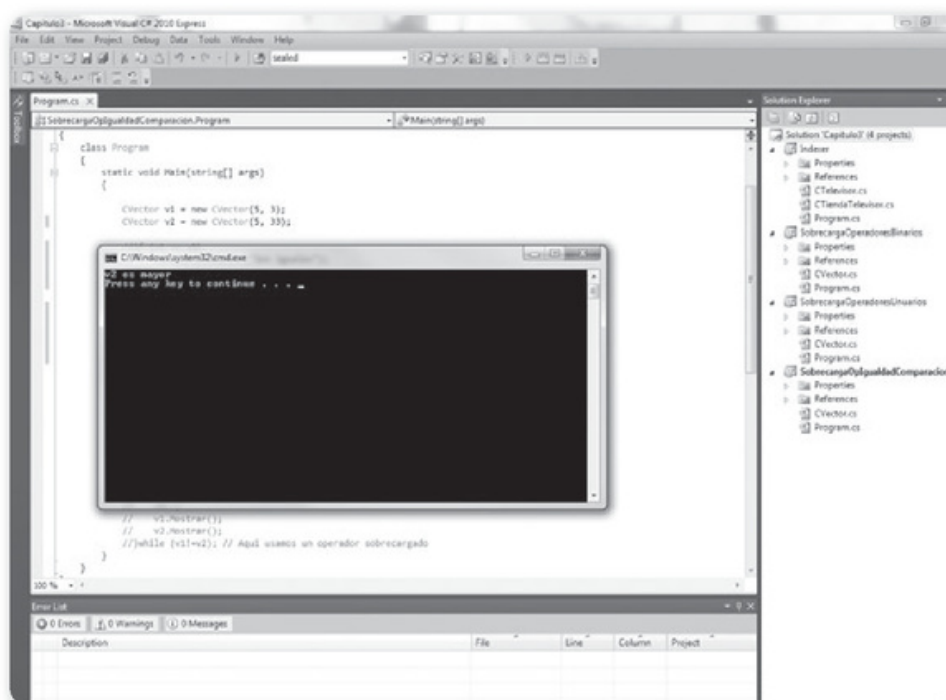
En este momento, podemos proceder a adicionar código a la función **Main()** y posteriormente verificar el comportamiento de estos operadores. La función queda de la siguiente manera:

```
static void Main(string[] args)
{
    CVector v1 = new CVector(5, 3);
    CVector v2 = new CVector(5, 33);
    if (v1 == v2)
        Console.WriteLine("Son iguales");
    else
        Console.WriteLine("No son iguales");
    // Comparamos los vectores
    if (v1 > v2)
        Console.WriteLine("v1 es mayor");
    else
        Console.WriteLine("v2 es mayor");
    if (v1 < v2)
```



```
        Console.WriteLine("v1 es menor");  
    else  
        Console.WriteLine("v2 es menor");  
}
```

La comparación se lleva a cabo usando los operadores; debemos acceder a emplearlos dentro de un **if**. Si ejecutamos la aplicación, podremos ver los mensajes correspondientes.



► **Figura 14.** En esta parte verificamos al operador > y obtenemos el mensaje que esperamos, para comprobar que funciona correctamente.

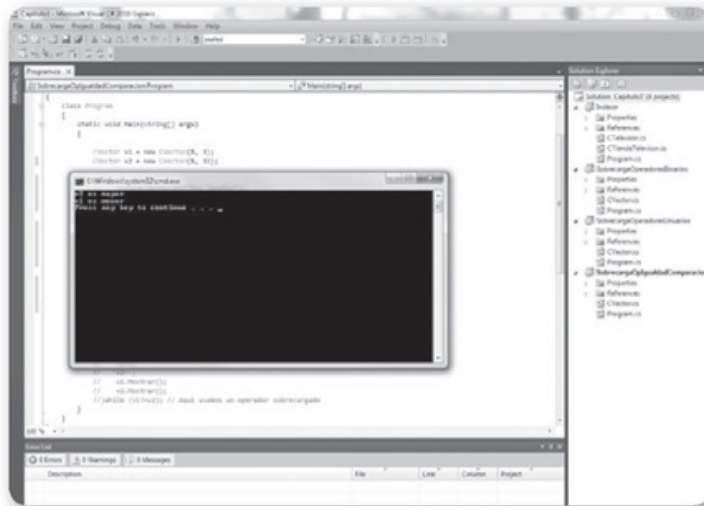


LA IMPORTANCIA DE LAS SOBRECARGAS



Al inicio la sobrecarga nos puede parecer código de más para algo que se puede resolver sencillamente con un método. Pero debemos de recordar que el poder de la sobrecarga se ve en el cliente, es decir el código que hace uso de nuestra clase, ya que ahí es donde se ve la reducción de código, simplificación de la lógica y también facilidad de lectura.

Para verificar que podemos usar la sobrecarga de operadores también en ciclos, hacemos una adición a la función **Main()**:

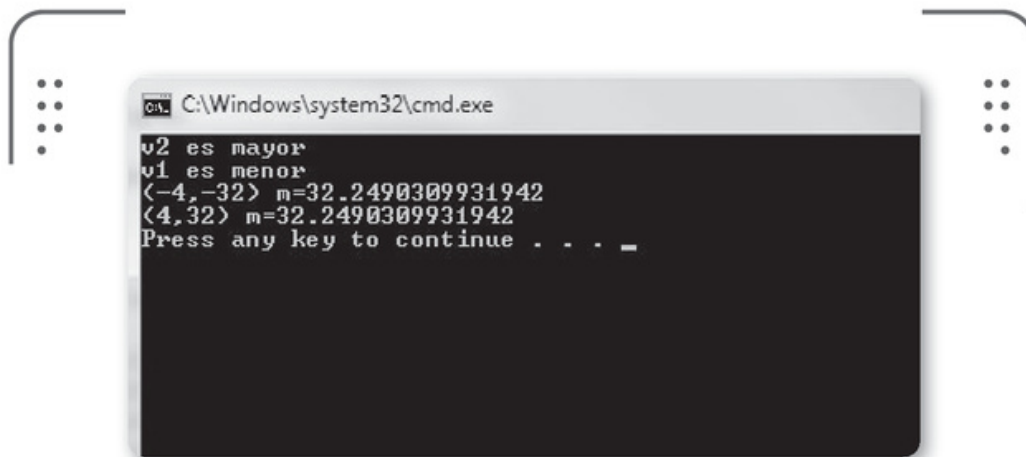


► **Figura 15.** Tengamos en cuenta que con la ejecución completa, también podemos comprobar el operador <.

```
static void Main(string[] args)
{
    CVector v1 = new CVector(5, 3);
    CVector v2 = new CVector(5, 33);
    if (v1 == v2)
        Console.WriteLine("Son iguales");
    else
        Console.WriteLine("No son iguales");
    // Comparamos los vectores
    if (v1 > v2)
        Console.WriteLine("v1 es mayor");
    else
        Console.WriteLine("v2 es mayor");
    if (v1 < v2)
        Console.WriteLine("v1 es menor");
    else
        Console.WriteLine("v2 es menor");
    // Ejemplo con un ciclo
    v1 = new CVector(-5, -33);
    do{
```

```
v1++;  
v2--;  
v1.Mostrar();  
v2.Mostrar();  
}while (v1!=v2); // Aqui usamos un operador sobrecargado  
}
```

El ciclo se va a realizar mientras **v1** sea diferente de **v2**. Podemos utilizar la sobrecarga de operadores en cualquier lugar que necesite evaluar expresiones con esos operadores.



► **Figura 16.** Podemos comprobar que la sobrecarga también puede ser utilizada en ciclos.

Con esto finalizamos el tema de la sobrecarga de operadores. Teniendo estos conocimientos, es posible continuar sobrecargando los operadores que no han sido ejemplificados en este capítulo.



RESUMEN



El uso de indexers nos permite acceder a un ADT que se encuentre en el interior de una clase de forma sencilla y con una sintaxis similar a la que utiliza un arreglo. Esto simplifica la programación en aquellos lugares que utilicen la clase. La sobrecarga se emplea cuando deseamos emplear los operadores con nuestras propias clases. Los operadores pueden ser binarios o unarios. Para llevar a cabo la sobrecarga, recurrimos a la palabra clave **operator**. Los parámetros que recibe el método del operador son los operandos necesarios.

Actividades

TEST DE AUTOEVALUACIÓN

- 1 ¿Cómo funcionan los indexers?
- 2 ¿Por qué es necesario tener cuidado con el índice de un indexer?
- 3 ¿El indexer tiene todos los métodos propios de un arreglo?
- 4 ¿Qué es un ADT?
- 5 ¿Qué es la sobrecarga de operadores?
- 6 ¿Qué es un operador unuario?
- 7 ¿Qué es un operador binario?
- 8 ¿Para qué usamos la palabra clave **operator**?
- 9 ¿Qué ventajas nos da usar sobrecarga de operadores?
- 10 ¿Con qué interfaces nos podemos apoyar para facilitar la programación de la sobrecarga de operadores?

EJERCICIOS PRÁCTICOS

- 1 Haga la sobrecarga del operador ***** para el producto cruz vectorial.
- 2 Implemente un ciclo **for** que aproveche el operador sobrecargado de una clase.
- 3 Realice la sobrecarga de todos los operadores de comparación.
- 4 Invente una clase propia y sobrecargue los operadores más importantes.
- 5 Haga sobrecarga de operadores para los ejemplos del capítulo anterior.

Serialización

Hacer programas es una actividad muy interesante, pero hasta este momento, solamente podemos crear aquellos en los que, en cada ejecución, la información se replica otra vez. Sin embargo, es posible desarrollarlos para que se encarguen de guardar el estado de todos sus objetos en disco. Esto nos permitirá almacenar allí la información o regresar el programa al estado anterior antes de cerrar la aplicación. De esta forma, lograremos dar un paso más en la tarea de convertirnos en expertos desarrolladores. En este capítulo conoceremos los alcances y usos de la serialización.

▼ Guardar el estado de los objetos.....	128
Serialización binaria sencilla	130
Serialización SOAP sencilla.....	138
Serialización XML sencilla.....	143
Serialización de clases con composición en formato binario	146

Serialización con composición usando XML	153
Serialización de varios objetos.....	157

▼ Resumen.....	163
----------------	-----

▼ Actividades.....	164
--------------------	-----



Guardar el estado de los objetos

Recordemos que en gran cantidad de programas, es útil poder guardar el estado de los objetos. Muchas veces, necesitamos mantener de alguna manera los valores calculados o el estado actual de la aplicación, ya que esto nos permite volver a trabajar con ella desde donde nos quedamos antes de cerrarla, en vez de tener que recalcular todo cada vez que se inicie el software.

En un libro anterior exploramos la opción de guardar la información por medio de flujos y archivos. Esa metodología puede servirnos para conservar cualquier cantidad de información, pero es un poco complicada y engorrosa. Lo mejor sería guardar el estado de los objetos de una manera sencilla, de tal modo que todo el objeto sea escrito o leído rápidamente y sin inconvenientes. Esta idea de guardar el estado del objeto se conoce como **persistencia**.

La programación de la persistencia en C# es muy simple, y requiere solo de una decoración de nuestro objeto y de unas simples instrucciones para enviarlo a disco. Cuando guardamos al objeto en disco, podemos seleccionar diferentes formatos para hacerlo, cada uno de los cuales tiene sus propias características, ventajas y desventajas. Durante este capítulo exploraremos los distintos formatos y el modo de utilizarlos. Los tres formatos que aprenderemos son: **binario**, **SOAP** y **XML**.

Cuando se lleva a cabo la **serialización**, el estado del objeto se manda a disco o a un **stream**. La información guardada nos permite hacer luego el proceso de **deserialización**, en el que se crea una instancia y se coloca en su interior un estado igual al del objeto



ESTADO DEL OBJETO



El estado del objeto consta de los valores que tienen todos sus datos en un momento en particular de la ejecución del programa correspondiente. Cambia cada vez que un dato recibe un nuevo valor, ya sea por asignación o por un proceso de cómputo. Si queremos regresar un sistema a un estado anterior, únicamente necesitamos restaurar todos los objetos que lo componen.

serializado. No todos los formatos guardan toda la información del objeto, pero estudiaremos dichos casos en el momento oportuno.

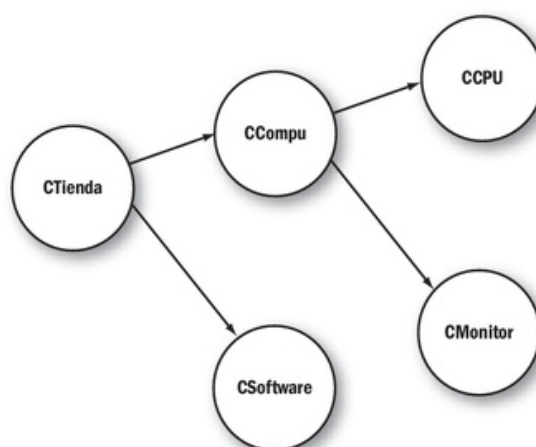
Para llevar a cabo la serialización, necesitamos incluir varios namespaces, dependiendo del formato que vayamos a usar, e, incluso, adicionar una referencia. También tendremos que hacer uso de **[Serializable]**, en las clases que deseamos que sean serializables.

Cuando efectuamos la serialización, se serializa no solo el objeto en particular, sino también todos los relacionados. Este es un punto importante, ya que al planear nuestra serialización, debemos tomar en cuenta las relaciones del objeto, tanto de **es un** como de **tiene un**. Estas relaciones de objetos se conocen como **gráfica de objetos** u **object graph**; aunque pueden mostrar fielmente las relaciones orientadas a objetos, no siempre las mapean de forma directa.

El object graph asigna a cada objeto un número, que funciona como identificador y es único; es decir, ningún número estará repetido en dos o más objetos. Estos números solamente sirven para la representación interna del object graph y no guardan ninguna relación directa con la secuencia que usamos para instanciar los objetos.

Las dependencias de los objetos son mapeadas una vez que se han identificado. Visualmente, las mostramos por medio de flechas, que pueden ser leídas como **depende de** y **se refiere a**.

El uso de un object graph tiene ventajas, pues permite reconstruir fácilmente el estado de la aplicación o los objetos, no solo de sus datos internos, sino también de la forma en que están relacionados entre sí.



► **Figura 1.**
En este diagrama podemos observar una representación de un object graph sencillo.

Serialización binaria sencilla

Empecemos con una serialización sencilla. En este ejemplo, crearemos una clase que puede ser serializada; luego, en la aplicación podremos mandar el objeto a disco o leerlo. Usaremos un formato binario para la serialización. Este es compacto y, de los tres formatos que veremos, es el que produce archivos de menor tamaño. Con él tenemos lo que conocemos como fidelidad de tipo, por lo que es útil si el archivo creado va a utilizarse en el mismo sistema operativo, aunque sea en diferentes computadoras. Cuando hacemos uso del formato binario, todos los datos del objeto son serializados, sin importar si son públicos o privados.

Indicamos que una clase en particular va a ser serializada al decorarla con el atributo **[Serializable]**. En nuestra aplicación necesitamos crear un **stream** a archivo de disco y un objeto del formateador. Con el **formateador** usaremos el método **Serialize()** para enviar el estado del objeto al **stream** y, por lo tanto, al disco. Cuando deseamos leer el objeto de disco, hacemos uso del método **Deserialize()**.

Empecemos por crear una aplicación nueva, a la cual adicionamos una clase llamada **CCPU**, que tiene el siguiente código:

```
[Serializable]
class CCPU
{
    // Datos de la clase
    private string procesador;
    private int memoria;
    private double discoDuro;
    private string fabricante;
    // Constructor
    public CCPU()
    {
        // Colocamos datos de default
        procesador = "i3";
        memoria = 1;
        discoDuro = 80.0;
        fabricante = "PatitoCo";
    }
    // Constructor sobrecargado
```



```
public CCPU(string pprocesador, int mmemoria, double ddiscoDuro)
{
    // Asignamos los valores recibidos
    procesador = pprocesador;
    memoria = mmemoria;
    discoDuro = ddiscoDuro;
    fabricante = "PatitoCo";
}

// Propiedades
public string Procesador
{
    get { return procesador; }
    set { procesador = value; }
}

public int Memoria
{
    get { return memoria; }
    set { memoria = value; }
}

public double DiscoDuro
{
    get { return discoDuro; }
    set { discoDuro = value; }
}

// Hacemos override a ToString()
public override string ToString()
{
    return string.Format("CPU - {0}, {1} GB de memoria, {2} GB de HD",
        procesador, memoria, discoDuro);
}
}
```

Debemos tener en cuenta que antes de que se realice la declaración de la clase correspondiente, será necesario que coloquemos el atributo denominado **[Serializable]**, para indicar que la clase que estamos definiendo a continuación es serializable y podemos llevar a cabo la persistencia de las instancias de ella. Después de esto, procedemos a declarar la clase de la manera tradicional.

La clase tiene cuatro datos con acceso privado que definen algunas de las características de **CPU**. Hay datos de diferentes tipos, para poder comprobar que la serialización se efectúa sin importar el tipo de dato que contenga la clase. A continuación, tenemos el constructor que se encarga de inicializar los datos con algunos valores. Este ha sido sobrecargado para que también podamos inicializar los datos con los valores que deseemos. Solamente el dato de fabricante es inicializado siempre con el mismo valor.

Podemos ver que a tres de los datos les hemos creado propiedades, todas ellas con acceso público. El dato **fabricante** es privado y no tiene ninguna propiedad. Esto es importante porque nos permitirá conocer cómo los distintos formatos tratan a los datos públicos y privados, así como a aquellos que tengan propiedades.

Tengamos en cuenta que la clase tiene un **override** al método **ToString()**, con el fin de facilitarnos la impresión del estado del objeto. En este método damos formato a una cadena que contiene la información de **CPU** indicando su procesador, memoria y disco.

En el documento donde tenemos la función **Main()**, necesitamos adicionar un par de namespaces. Para hacerlo, agregamos lo siguiente en la parte superior del código:

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
// namespaces necesarios  
using System.Runtime.Serialization.Formatters.Binary;  
using System.IO;
```



DATOS NO SERIALIZABLES



Debemos tener en cuenta que es posible seleccionar qué datos no se utilizarán durante la serialización, aun si su acceso es público. Para lograrlo, necesitamos adicionar el atributo **[NonSerialized]** antes del dato. Esto resulta útil si no es preciso guardar cierta información o si, simplemente, por seguridad u otro motivo, no deseamos que esta quede almacenada.

A continuación, procedemos a colocar el código que corresponde a la función **Main()**. Dentro de esta función incluimos el código para llevar a cabo la serialización y la deserialización, pero este puede ir en cualquier parte válida del programa. El código es el siguiente:

```
static void Main(string[] args)
{
    // Datos
    string opcion;
    // Pedimos la opcion al usuario
    Console.WriteLine("1- Salvar objeto, 2- Leer objeto");
    opcion = Console.ReadLine();
    // Salvar el objeto
    if (opcion == "1")
    {
        // Colocamos la informacion / estado del objeto
        CCPU miCPU = new CCPU("i5", 4, 180.5);
        Console.WriteLine("Salvando el objeto");
        // Seleccionamos el formateador
        BinaryFormatter formateador = new BinaryFormatter();
        // Creamos el stream
        Stream miStream = new FileStream("miCPU.cpu", FileMode.Create,
        FileAccess.Write, FileShare.None);
        formateador.Serialize(miStream, miCPU);
        miStream.Close();
    }

    // leer el objeto
    if (opcion == "2")
```



BINARYFORMATTER



El namespace denominado **system.runtime.serialization.formatters.binary** es necesario cuando deseamos serializar o deserializar un objeto o serie de objetos conectados por una gráfica de objetos en formato binario. Debemos saber que en su interior tiene una clase llamada **BinaryFormatter**. Esta es la clase principal que usaremos en la serialización con formato binario.


```
{
    Console.WriteLine("Leyendo el objeto");
    // Seleccionamos el formateador
    BinaryFormatter formateador = new BinaryFormatter();
    Stream miStream = new FileStream("miCPU.cpu", FileMode.Open,
    FileAccess.Read, FileShare.None);
    CCPU miCPU2 = (CCPU)formateador.Deserialize(miStream);
    miStream.Close();
    Console.WriteLine(miCPU2.ToString());
}
```

En la función declaramos una variable de tipo **string** llamada **opcion**. Esta variable es usada para guardar la selección del usuario entre llevar a cabo la serialización o deserializar el objeto. Efectuamos la petición al usuario mostrando un mensaje y pidiendo una entrada por teclado. Esta entrada queda guardada dentro de la variable **opcion**. Las opciones que le damos al usuario son **1** para serializar y **2** para deserializar.

Ahora podemos darnos cuenta de la forma de efectuar la serialización; esta es la opción número uno. Para serializar, lo primero que necesitamos es tener al menos una instancia de la clase que deseamos serializar. Para esto creamos una instancia de **CCPU** llamada **miCPU** y pasamos como parámetros al constructor los datos que deseamos que queden guardados en esa instancia.

Como en este ejemplo quisimos escribir el archivo en formato binario, creamos una instancia de **BinaryFormatter**, que nos provee de la funcionalidad para hacerlo. La instancia se llama **formateador**; recordemos que hacemos uso de su constructor por default.

A continuación, creamos un objeto de tipo **Stream** al cual denominamos **miStream**. Gracias al polimorfismo, usamos el stream para archivos y utilizamos el constructor de **FileStream**. El primer parámetro representa el nombre y la ruta del archivo que se va a crear. En este caso, el archivo se creará en la misma locación que el lugar donde esté el ejecutable. El nombre dado al archivo es **miCPU.cpu**, pero nosotros podemos colocar cualquier nombre y extensión que nos sea conveniente. El modo del archivo es creación, y el acceso que estamos utilizando para él es el de escritura. Es decir que vamos a crear un archivo y lo utilizaremos para escribir la información que corresponde. Al hacer uso

de **FileShare.None**, indicamos que el archivo no puede ser abierto por nadie más mientras nosotros lo estemos utilizando.

Ya con el formateador y el stream listos, procedemos a serializar el objeto. Para hacerlo, invocamos al método **Serialize()**, el cual se encuentra en nuestro formateador. Este método necesita dos parámetros: el primero es el stream, que en este caso ya está referenciado a un archivo en disco; y el segundo es el objeto por serializar.

Al terminar de serializar el objeto deseado, procedemos a cerrar el stream por medio del método **Close()**. No debemos olvidarnos de cerrar cualquier stream que hayamos creado.

Ahora podemos continuar con el proceso de deserialización; es decir, leemos la información de disco y, con ella, el objeto guardado será recreado en memoria. Si tuviéramos una gráfica de objetos guardada, el proceso involucraría la recreación de toda la gráfica.

Cuando llevamos a cabo la deserialización, empezamos por crear la instancia del formateador, que es de tipo **BinaryFormatter**. Es importante usar el mismo tipo de formato para serializar y deserializar, porque de otro modo, la información no podrá leerse correctamente. A continuación, procedemos a crear el stream. Usamos el mismo nombre, ya que deseamos leer el archivo que estamos creando con el programa. El modo del archivo en este caso es **Open** y el acceso corresponde a **Read**, es decir que abrimos el archivo para poder leer sus contenidos. Es importante tener presente que el archivo nuevamente no se comparte, por lo que otras aplicaciones no podrán abrirlo mientras lo use nuestro programa.

La siguiente línea de código es fundamental porque en ella se lleva a cabo la parte de la deserialización. Para leer nuestro objeto, hacemos uso del método **Deserialize()**; este tiene un parámetro que es el stream con la información de archivo por leer. El método **Deserialize()** del formato

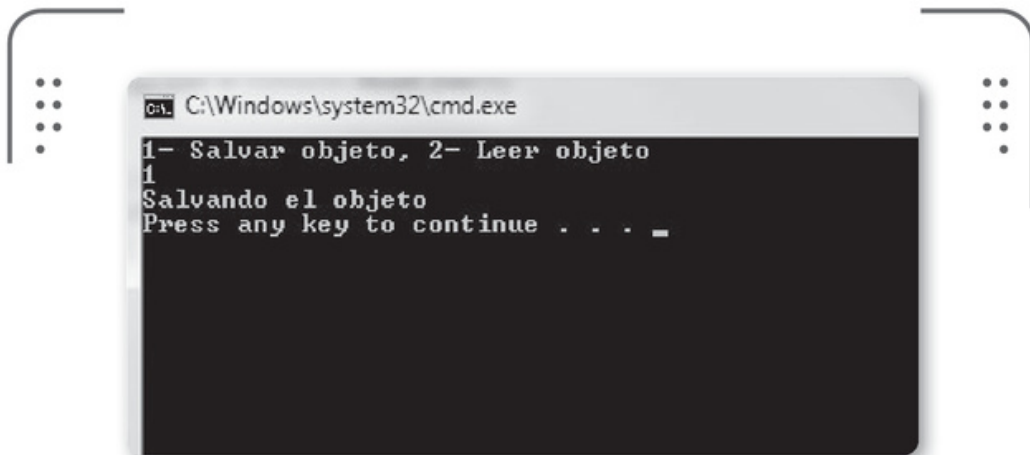


AGREGAR SEGURIDAD A LA SERIALIZACIÓN



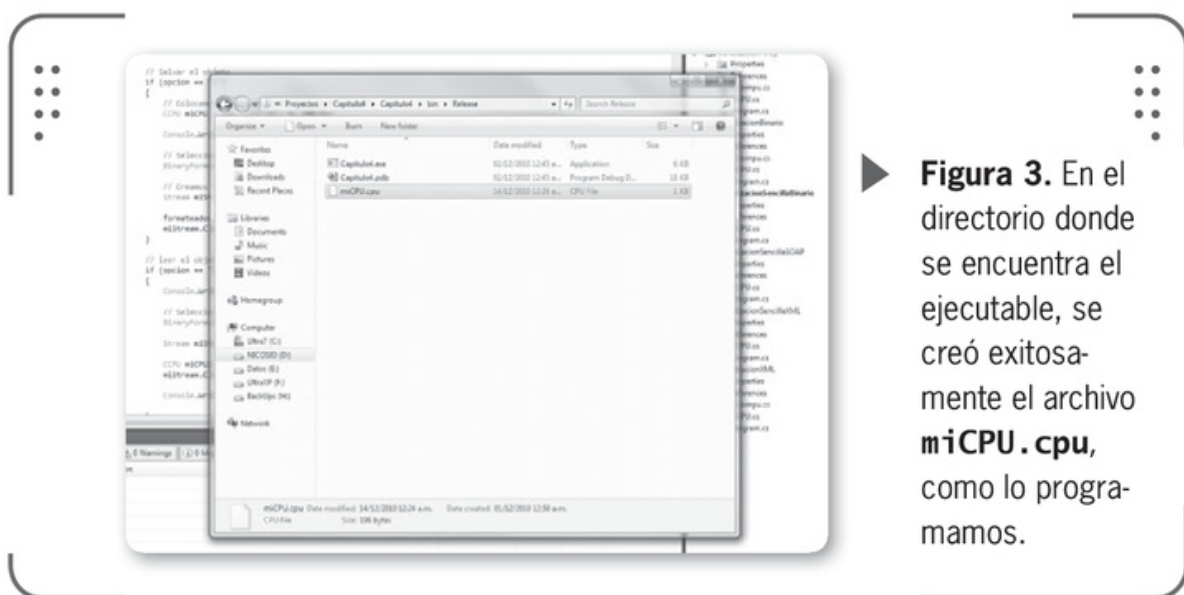
Tengamos en cuenta que cuando llevamos a cabo el proceso de serialización, pueden ocurrir problemas; por ejemplo, debemos saber que puede suceder que el stream que deseamos abrir por algún motivo genere un error, lo cual haría que la aplicación fallara. Lo mejor en este caso es usar el código de serialización o deserialización con un **try catch**.

binario regresa un objeto de tipo **object**, por lo que es necesario usar un **type cast** para que el objeto pueda ser empleado correctamente. El objeto que estamos leyendo quedará guardado en **miCPU2**, que es de tipo **CCPU**. Es preciso verificar que el tipo que estamos leyendo quede guardado en el mismo tipo que cuando fue escrito.



► **Figura 2.** En esta imagen podemos observar la ejecución del programa cuando crea el objeto y lo serializa.

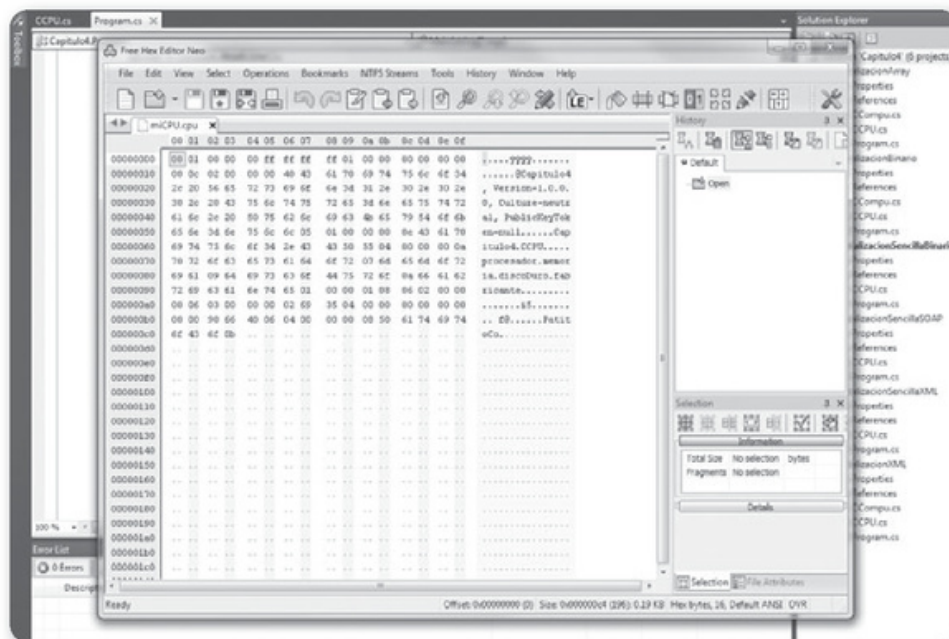
Una vez que hemos leído el objeto, podemos proceder a cerrar el stream. Para verificar que el objeto se leyó correctamente, pasamos a mostrar sus contenidos, que deben ser los mismos que el objeto guardado en un principio.



► **Figura 3.** En el directorio donde se encuentra el ejecutable, se creó exitosamente el archivo **miCPU.cpu**, como lo programamos.

Ahora podemos proceder a probar el software, y verificar la escritura y lectura de nuestro objeto.

La mejor forma que tenemos para poder comprobar que esta funcionando y el archivo de serialización contiene la información es por medio de un editor binario. Este editor nos muestra de una forma que podemos leer e interpretar los datos que están contenidos en el archivo.

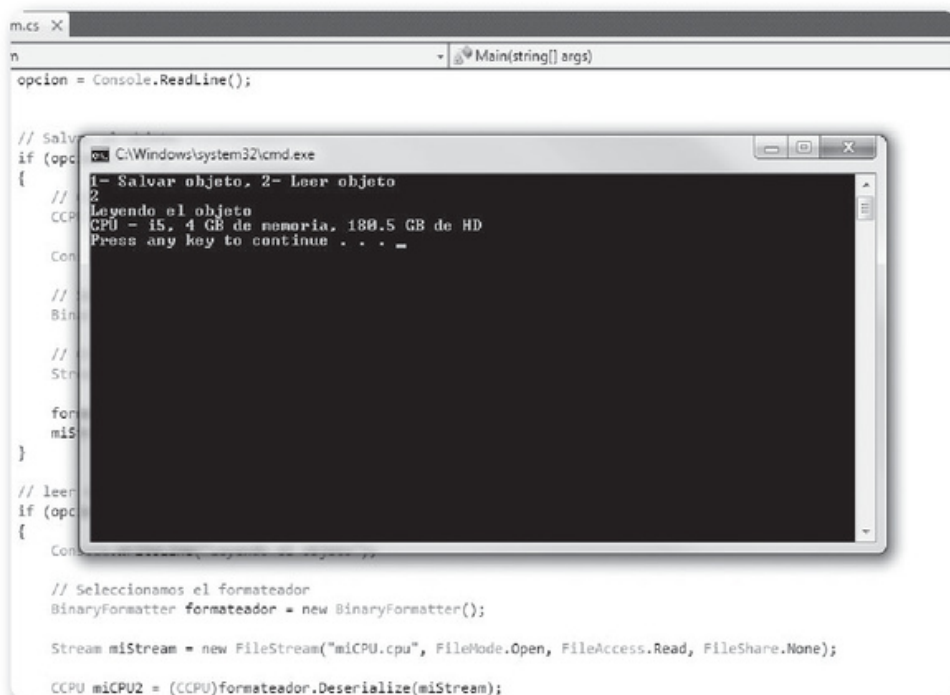


► **Figura 4.** Con la ayuda de un editor de archivos, es posible realizar la observación de los contenidos de **miCPU.cpu**; vemos que en su interior se encuentra la información del objeto.



EDITOR DE ARCHIVOS

Es muy importante tener en cuenta que en el momento en que necesitemos un editor de archivos, podemos proceder a buscarlo en Internet, ya que la Red pone a nuestra disposición una gran cantidad de opciones, tanto comerciales como gratuitas. Hay varios que pueden resultar útiles; como ya sabemos, algunos son gratuitos y otros tienen costo. No todos ofrecen las mismas herramientas, por lo que tendremos que



► **Figura 5.** En esta imagen hacemos la prueba de leer el archivo, y obtenemos el objeto como lo esperábamos.

Con esto hemos comprobado que la aplicación funciona, pero debemos observar qué elementos se han guardado. Cuando hacemos uso del formato binario, se serializan todos los datos del objeto, como los públicos, los privados con propiedades públicas e, incluso, los privados sin propiedades. Esto puede observarse en el archivo, ya que incluso el dato **fabricante** ha sido guardado.

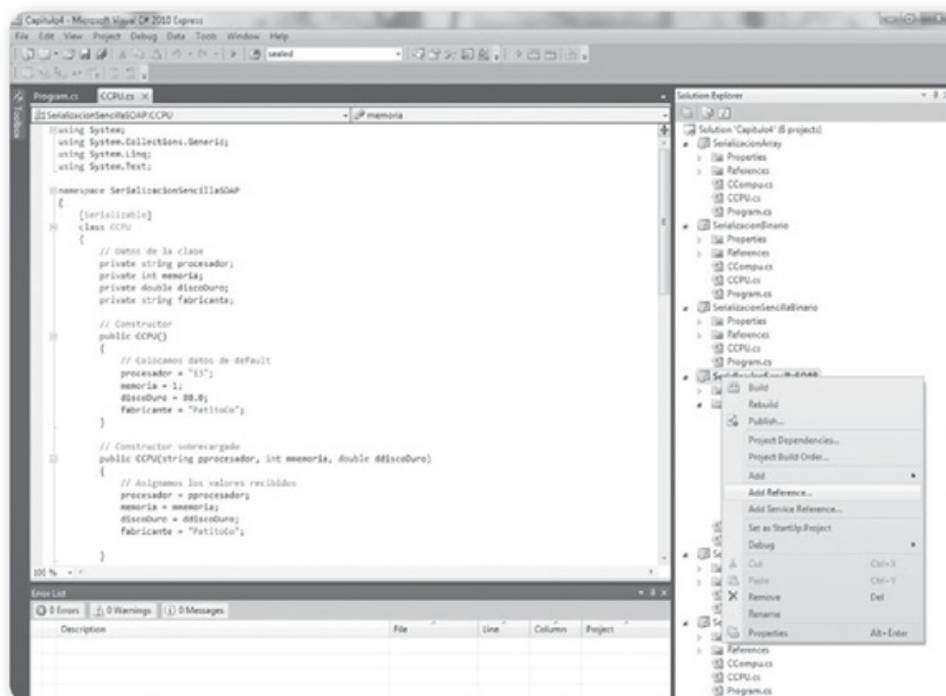
Serialización SOAP sencilla

Veamos otro tipo de formato que puede utilizarse en la serialización; en este caso, es **SOAP**. Se trata de un framework que utiliza **XML** para definir protocolos que nos permitan intercambiar información estructurada en un ambiente distribuido. El uso de SOAP en vez del formato binario no es muy complicado, pero debemos considerar algunas diferencias importantes.

La primera es que no tenemos fidelidad de tipos, pero esto no es grave, porque nos permite intercambiar información más

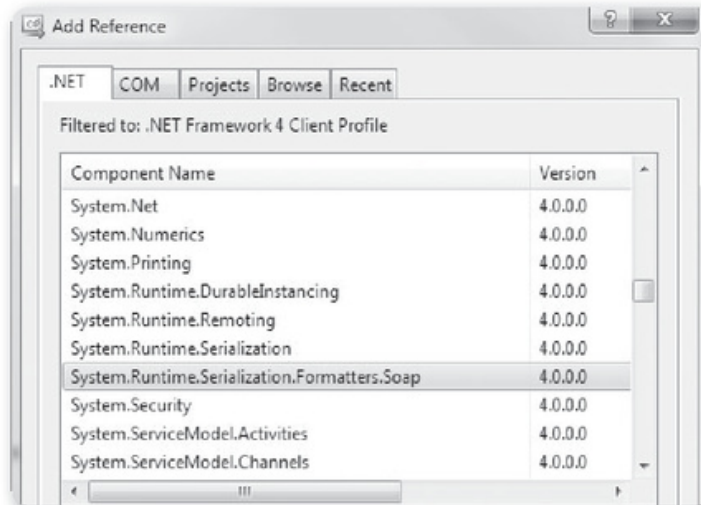
fácilmente con sistemas operativos que son distintos de Windows. De esta manera, es posible leer el dato y colocarlo en su propia representación interna del tipo adecuado.

Cuando deseamos utilizar SOAP, tenemos que adicionar una referencia; la que necesitamos es la denominada **System.Runtime.Serialization.Formatters.Soap** y la forma de adicionarla es muy simple. Para realizar esta tarea lo primero que tenemos que hacer es un clic con el botón derecho del mouse sobre el nombre del proyecto. Aparecerá un menú donde seleccionamos **Adicionar Referencia**.



► **Figura 6.** Es importante saber que debemos adicionar una referencia para poder utilizar el formato de serialización con SOAP.

Luego de realizar las acciones que comentamos, veremos que se presenta un cuadro de diálogo desde el cual podemos elegir la que queremos adicionar. Es importante saber que la que precisamos se encuentra en .NET: es necesario que busquemos la denominada **System.Runtime.Serialization.Formatters.Soap**; luego, simplemente pulsamos el botón llamado **OK**. Con esto, podremos estar seguros de que la referencia habrá sido adicionada.



► **Figura 7.** Seleccionamos la referencia por utilizar, la cual nos permite emplear la funcionalidad de SOAP en la serialización correspondiente.

Ahora procedemos a crear la aplicación; el código de la clase **CCPU** no tiene ningún cambio, pero es diferente para la función **Main()**. No hay que olvidar adicionar los siguientes namespaces para poder hacer uso de los streams y del formato de serialización de SOAP:

```
// Namespace necesarios
using System.IO;
// Necesario para SOAP
using System.Runtime.Serialization.Formatters.Soap;
```

El código de la función **Main()** es el siguiente:

```
class Program
{
    static void Main(string[] args)
    {
        // Datos
        string opcion;
        // Pedimos la opcion al usuario
        Console.WriteLine("1- Salvar objeto, 2- Leer objeto");
        opcion = Console.ReadLine();
        // Salvar el objeto
        if (opcion == "1")
```

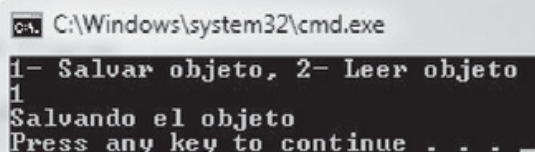
```
{
    // Colocamos la informacion / estado del objeto
    CCPU miCPU = new CCPU("i5", 4, 180.5);
    Console.WriteLine("Salvando el objeto");
    // Seleccionamos el formateador
    SoapFormatter formateador = new SoapFormatter();
    // Creamos el stream
    Stream miStream = new FileStream("miCPU.cpu", FileMode.Create,
    FileAccess.Write, FileShare.None);
    formateador.Serialize(miStream, miCPU);
    miStream.Close();
}

// leer el objeto
if (opcion == "2")
{
    Console.WriteLine("Leyendo el objeto");
    // Seleccionamos el formateador
    SoapFormatter formateador = new SoapFormatter();
    Stream miStream = new FileStream("miCPU.cpu", FileMode.Open,
    FileAccess.Read, FileShare.None);
    CCPU miCPU2 = (CCPU)formateador.Deserialize(miStream);
    miStream.Close();
    Console.WriteLine(miCPU2.ToString());
}
}
```

El código es muy similar al del ejemplo anterior, por lo que solo nos concentraremos en los cambios. El primero de ellos se encuentra en la sección referente a la serialización. En él, vemos que el **formateador** por utilizar es de tipo **SoapFormatter**. En este caso, usamos el constructor por default para él. El resto del código de esta sección es igual: creamos el stream y, luego, con el formateador serializamos hacia el stream el objeto llamado **miCPU**.

En la sección que corresponde a la deserialización tenemos un cambio similar, ya que modificamos el formateador y lo colocamos de tipo **SoapFormatter**. El resto del programa no presenta modificaciones.

Ejecutamos la aplicación y comprobamos su funcionamiento.



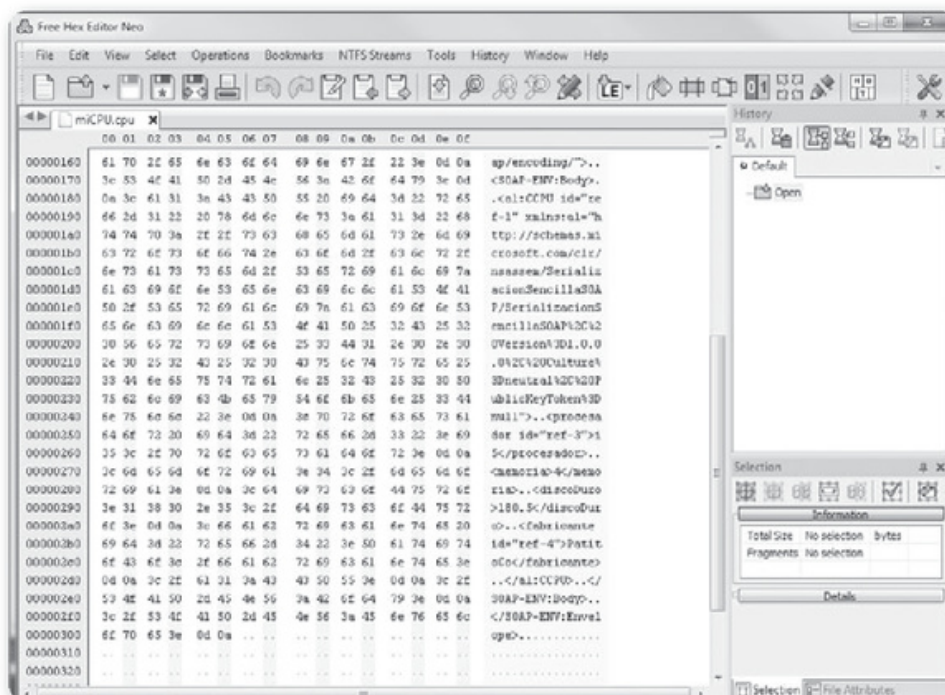
```

C:\Windows\system32\cmd.exe
1- Salvar objeto, 2- Leer objeto
1
Salvando el objeto
Press any key to continue . . . _

```

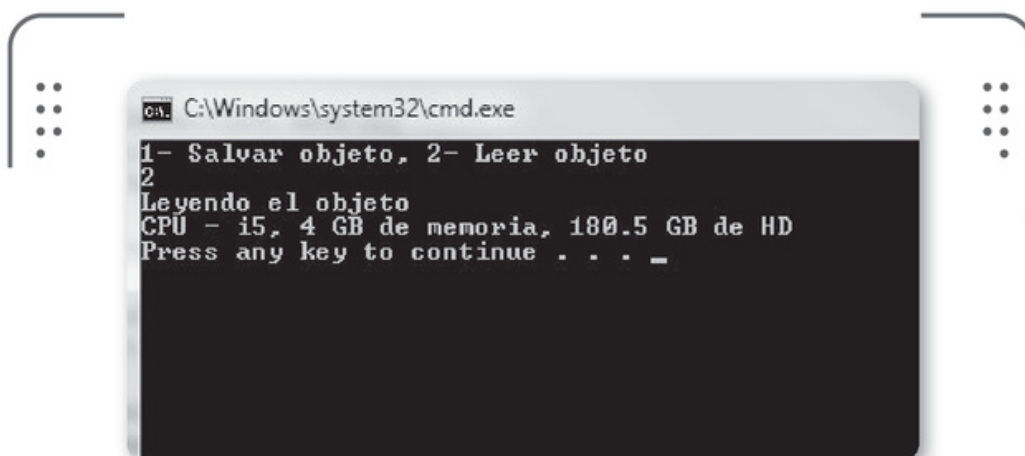
► **Figura 8.** Ejecutamos la aplicación y seleccionamos llevar a cabo la serialización, en este caso, en formato SOAP.

Ahora podemos observar el archivo creado. Es fácil notar que la manera en que se guarda la información es diferente del archivo binario. En este caso, se almacena de acuerdo con las reglas de SOAP:



► **Figura 9.** Estos son los contenidos del archivo creado; en su interior vemos la estructura del archivo según las reglas de SOAP.


```
<SOAP-ENV:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:SOAP-
ENC="http://schemas.xmlsoap.org/soap/encoding/" xmlns:SOAP-ENV="http://
schemas.xmlsoap.org/soap/envelope/" xmlns:clr="http://schemas.microsoft.com/soap/
encoding clr/1.0" SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/
encoding/">
  <SOAP-ENV:Body>
    <a1:CCPU id="ref-1" xmlns:a1="http://schemas.microsoft.com/clr/
nsassem/SerializacionSencillaSOAP/SerializacionSencillaSOAP%2C%20
Version%3D1.0.0.0%2C%20Culture%3Dneutral%2C%20
PublicKeyToken%3Dnull">
      <procesador id="ref-3">i5</procesador>
      <memoria>4</memoria>
      <discoDuro>180.5</discoDuro>
      <fabricante id="ref-4">PatitoCo</fabricante>
    </a1:CCPU>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```



► **Figura 10.** Aquí vemos que el archivo fue leído exitosamente, y sus contenidos se recrearon en un nuevo objeto.

Serialización XML sencilla

En este momento, aprenderemos cómo llevar a cabo la serialización usando el formato de **XML**. Es muy sencillo hacerlo, porque ya conocemos los pasos para hacer la serialización y solo debemos utilizar

el formato adecuado. La clase **CCPU** no se verá modificada; únicamente cambiaremos el código en la función **Main()**.

Para usar este tipo de serialización, también necesitamos recurrir a algunos namespaces, como vemos a continuación:

```
// Namespace necesarios
using System.Xml.Serialization;
using System.IO;
```

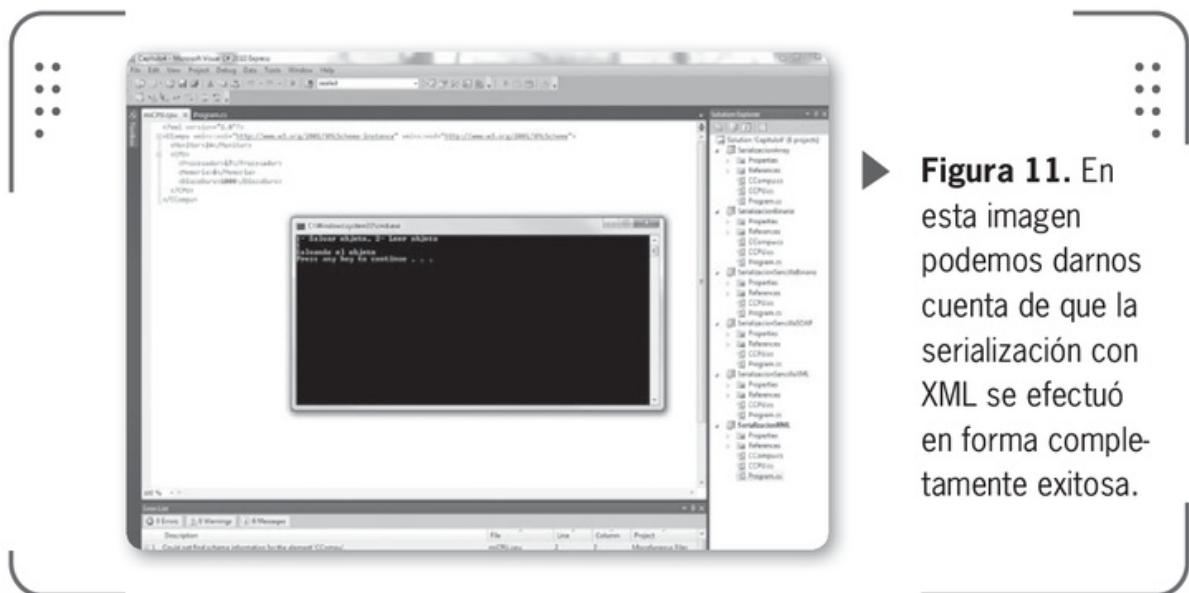
El código de la función **Main()** queda de la siguiente manera:

```
static void Main(string[] args)
{
    // Datos
    string opcion;
    // Pedimos la opcion al usuario
    Console.WriteLine("1- Salvar objeto, 2- Leer objeto");
    opcion = Console.ReadLine();
    // Salvar el objeto
    if (opcion == "1")
    {
        // Colocamos la informacion / estado del objeto
        CCPU miCPU = new CCPU("i5", 4, 180.5);
        Console.WriteLine("Salvando el objeto");
        // Seleccionamos el formateador
        XmlSerializer formateador = new XmlSerializer(typeof(CCPU));
        // Creamos el stream
        Stream miStream = new FileStream("miCPU.cpu", FileMode.Create,
        FileAccess.Write, FileShare.None);
        formateador.Serialize(miStream, miCPU);
        miStream.Close();
    }
    // leer el objeto
    if (opcion == "2")
    {
        Console.WriteLine("Leyendo el objeto");
        // Seleccionamos el formateador
```

```
XmlSerializer formateador = new XmlSerializer(typeof(CCPU));  
Stream miStream = new FileStream("miCPU.cpu", FileMode.Open,  
    FileAccess.Read, FileShare.None);  
CCPU miCPU2 = (CCPU)formateador.Deserialize(miStream);  
miStream.Close();  
Console.WriteLine(miCPU2.ToString());  
}  
}
```

En la sección que corresponde a la serialización, creamos el formateador de tipo **XmlSerializer**. El código luce un poco diferente. El constructor de **XmlSerializer** llevará una serie de parámetros debido a que **XmlaSerializer** necesita información sobre los tipos que definen la gráfica de objeto que se va a serializar. Como en nuestro ejemplo solo estamos serializando una clase, colocamos directamente el tipo de la clase **CCPU**.

En la sección que corresponde a la deserialización, tenemos el mismo caso, de manera que usamos **XmlSerializer**, y en el constructor también indicamos el tipo de la clase por serializar. Ahora podemos ejecutar el programa y llevar a cabo la serialización con formato XML.



► **Figura 11.** En esta imagen podemos darnos cuenta de que la serialización con XML se efectuó en forma completamente exitosa.

Una vez creado el archivo, podemos proceder a ver sus contenidos y, de este modo, compararlo con los otros formatos. La información se guarda de acuerdo con las reglas correspondientes a XML. En este formato, tampoco se tiene fidelidad de tipos, pero debido a la manera

en que trabaja XML y por la facilidad de leer e interpretar esta clase de archivos, debemos tener en cuenta que esto resultará muy útil para poder transferir el estado de los objetos entre diferentes computadoras y sistemas operativos.

```
<?xml version="1.0"?>
<CCPU xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Procesador>i5</Procesador>
  <Memoria>4</Memoria>
  <DiscoDuro>180.5</DiscoDuro>
</CCPU>
```

Si observamos atentamente el archivo XML, identificamos de inmediato la clase **CCPU** y sus datos. En los datos que aparecen en el archivo solo tenemos **procesador**, **memoria** y **discoduro**. El dato **fabricante** no aparece en el archivo, porque el formato XML únicamente serializa los datos que son públicos o los privados que son expuestos por medio de propiedades públicas. El dato **fabricante** es privado y no tiene ninguna propiedad. Este es un punto importante para tener en cuenta cuando usamos XML como formato, ya que si no planeamos la serialización adecuadamente, podemos obtener restauraciones de estado incorrectas en nuestros objetos.

Las serializaciones que hemos visto hasta el momento solo componen una clase, pero también es posible serializar objetos de clases que tengan composición o herencia. Veamos cómo llevar a cabo este procedimiento.

Serialización de clases con composición en formato binario

Es tiempo de ver cómo serializar objetos de clases más complejas. Crearemos una clase que tendrá una composición a otra clase. El hecho de tener este tipo de clases no complica demasiado la composición, pero es preciso tener cuidado con algunos detalles para que todo se lleve a cabo correctamente.

Vamos a tener una clase **CCompu**, compuesta por la clase **CCPU**. Empecemos por crear un nuevo proyecto y adicionar la mencionada:


```
[Serializable]
class CCPU
{
    // Datos de la clase
    private string procesador;
    private int memoria;
    private double discoDuro;
    private string fabricante;
    // Constructor
    public CCPU()
    {
        // Colocamos datos de default
        procesador = "i3";
        memoria = 1;
        discoDuro = 80.0;
        fabricante = "PatitoCo";
    }
    // Constructor sobrecargado
    public CCPU(string pprocesador, int mmemoria, double ddiscoDuro)
    {
        // Asignamos los valores recibidos
        procesador = pprocesador;
        memoria = mmemoria;
        discoDuro = ddiscoDuro;
        fabricante = "PatitoCo";
    }
    // Propiedades
    public string Procesador
    {
        get { return procesador; }
        set { procesador = value; }
    }
    public int Memoria
    {
        get { return memoria; }
        set { memoria = value; }
    }
}
```

```
public double DiscoDuro
{
    get { return discoDuro; }
    set { discoDuro = value; }
}
// Hacemos override a ToString()
public override string ToString()
{
    return string.Format("CPU - {0}, {1} GB de memoria, {2} GB de HD",
        procesador, memoria, discoDuro);
}
```

Esta clase ya es conocida por nosotros, pero cabe notar que hemos usado **[Serializable]**. No hay que olvidar esto, porque es importante para evitar errores en la aplicación que estamos trabajando.

Ahora adicionamos otra clase, que lleva por nombre **CCompu** y cuyo código es el que vemos a continuación:

```
[Serializable]
class CCompu
{
    // Datos necesarios
    private int monitor;
    private CCPU cpu;
    // Constructor
    public CCompu()
    {
        monitor = 20;
        cpu = new CCPU();
    }
    // Sobrecarga del constructor
    public CCompu(int pmonitor, string pprocesador, int mmemoria,
        double ddiscoDuro)
    {
        // Asignamos valores
        monitor = pmonitor;
```

```
        cpu = new CCPU(pprocesador, mmemoria, ddiscoDuro);
    }
    // Propiedades
    public int Monitor
    {
        get { return monitor; }
        set { monitor = value; }
    }
    public override string ToString()
    {
        return string.Format("Monitor {0}, CPU - {1}, {2} GB de memoria, {3} GB de HD", monitor, cpu.Procesador, cpu.Memoria, cpu.DiscoDuro);
    }
}
```

En esta clase, lo primero que hacemos es indicar que es serializable. La clase es muy sencilla y tiene únicamente dos datos: un entero llamado **monitor**, y uno de tipo **CCPU** que denominamos **cpu**. En este segundo dato es en donde sucede la composición. La clase tiene dos constructores y aprovechamos también al constructor de **CCPU**.

Creamos una propiedad para **monitor** y, para ayudarnos a desplegar la información, llevamos a cabo un override al método **ToString()**.

Ahora debemos programar la función **Main()**, como siempre, incluyendo los namespaces necesarios para el stream y la serialización:

```
// namespace necesarios
using System.Runtime.Serialization.Formatters.Binary;
using System.IO;
```



EL USO DE SERIALIZABLE



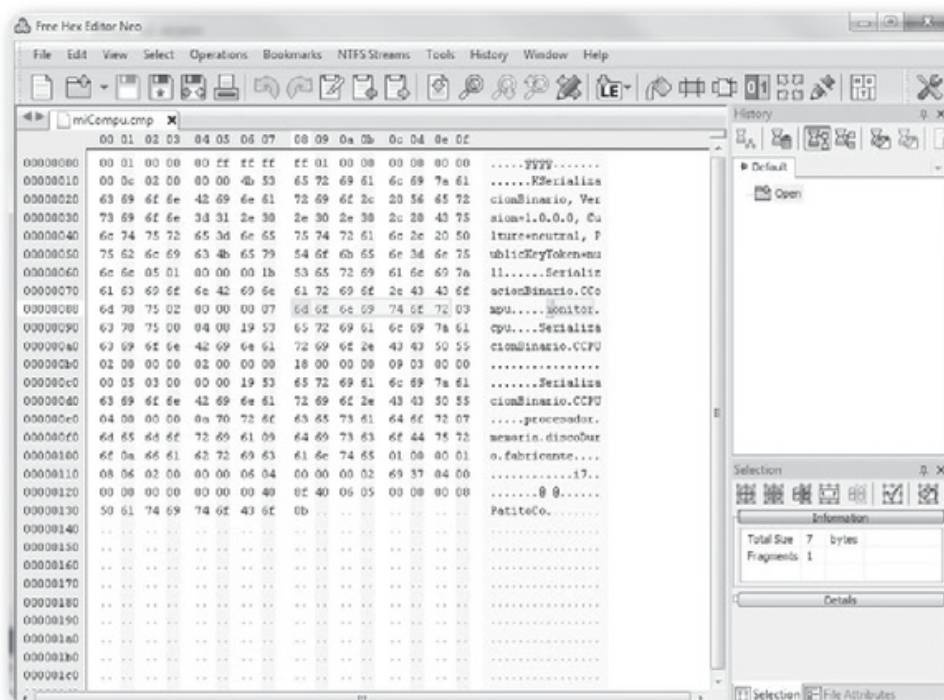
Debemos saber que cuando vamos a llevar a cabo la serialización de un objeto de una clase que tiene composición, no solo la clase debe tener **[Serializable]** sino también todas aquellas clases que la componen. Es importante tener en cuenta que esto es necesario porque estamos serializando gráficas de objetos, y estas guardan las relaciones entre las clases.

Para ejemplificar lo que ya comentamos, el código de la función denominada **Main()** es el que vemos a continuación:

```
static void Main(string[] args)
{
    // Datos
    string opcion;
    // Pedimos la opcion al usuario
    Console.WriteLine("1- Salvar objeto, 2- Leer objeto");
    opcion = Console.ReadLine();
    // Salvar el objeto
    if (opcion == "1")
    {
        // Colocamos la informacion / estado del objeto
        CCompu miCompu = new CCompu(24, "i7", 4, 1000);
        Console.WriteLine("Salvando el objeto");
        // Seleccionamos el formateador
        BinaryFormatter formateador = new BinaryFormatter();
        // Creamos el stream
        Stream miStream = new FileStream("miCompu.cmp", FileMode.
        Create, FileAccess.Write, FileShare.None);
        formateador.Serialize(miStream, miCompu);
        miStream.Close();
    }
    // leer el objeto
    if (opcion == "2")
    {
        Console.WriteLine("Leyendo el objeto");
        // Seleccionamos el formateador
        BinaryFormatter formateador = new BinaryFormatter();
        Stream miStream = new FileStream("miCompu.cmp", FileMode.Open,
        FileAccess.Read, FileShare.None);
        CCompu miCompu2 = (CCompu)formateador.Deserialize(miStream);
        miStream.Close();
        Console.WriteLine(miCompu2.ToString());
    }
}
```


El código es similar al de los ejemplos anteriores, pero podemos observar un cambio: en la sección de serialización estamos instanciando un objeto de tipo **CCompu**, es decir, la clase que tiene la composición. Por medio de su constructor, pasamos la información con la que deseamos que quede inicializado. El siguiente paso consiste en crear el formateador, que, en este caso, es de tipo **BinaryFormatter**. Con esto tendremos el objeto serializado de manera binaria en disco. Se crea el stream de la forma en que acostumbramos para generar y escribir el archivo.

La serialización se lleva a cabo por medio del método denominado **Serialize()**, pasando como parámetros el stream y el objeto por serializar. Luego de esto, finalizamos cerrando el stream.

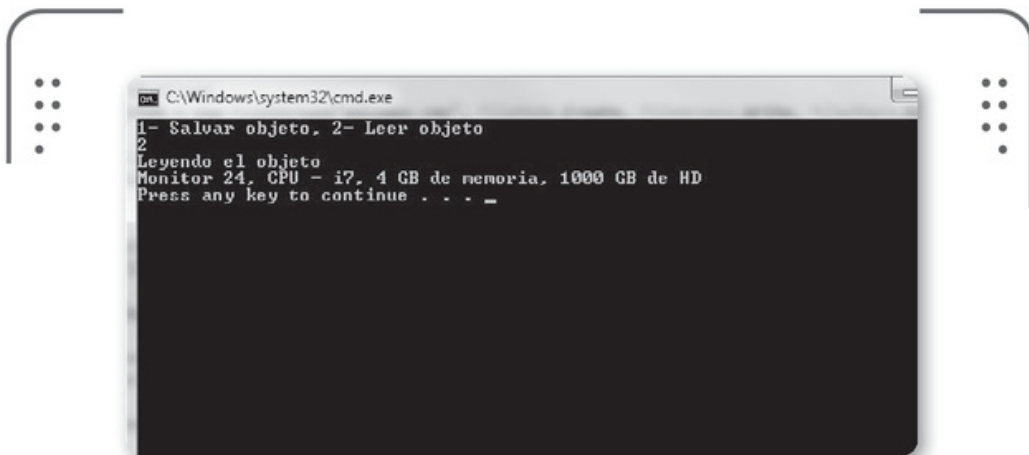


► **Figura 12.** Podemos darnos cuenta de que los contenidos del archivo creado nos muestran que ambos objetos fueron serializados.

La sección relacionada con la deserialización también tiene un cambio, y es el uso de un objeto de tipo **CCompu** para recibir la información que se lee desde el disco. No hay que olvidar el **type cast**. Debemos notar que para que este ejemplo funcione correctamente, las

dos clases son serializables y hemos usado la clase **CCompu** en vez de **CCPU**. La información de ambas tiene que quedar serializada.

Para continuar, será necesario que ejecutemos el programa y posteriormente observamos el archivo creado.



► **Figura 13.** Podemos comprobar que el objeto es recreado correctamente desde el archivo, incluyendo su composición.

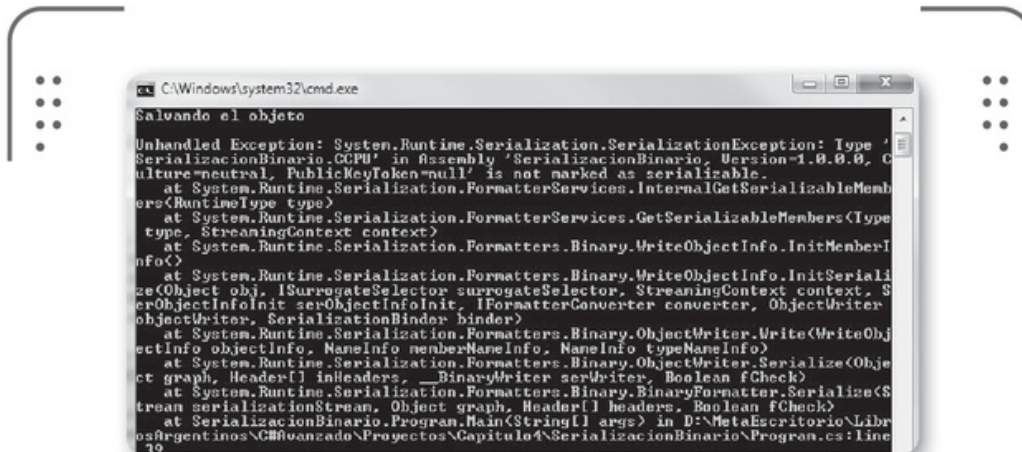
Es importante conocer qué puede suceder si alguna de las clases que utilizamos en la composición no es serializable, ya que esto puede provocar que el programa falle en su ejecución y obtengamos un error en él. Como experimento, podemos proceder a quitar **[Serializable]** de la clase **CCPU**. Con esto, la gráfica de objetos tendrá un problema, porque una de las clases en la composición no es serializable. Si compilamos la aplicación, no tendremos ningún inconveniente, pero al ejecutarla y tratar de llevar a cabo la serialización, el programa mostrará fallas. Nos daremos cuenta de que la excepción se genera, como puede observarse en la **Figura 14**.



SERIALIZAR CON XML



Un punto importante al serializar con XML es no olvidar que únicamente los datos públicos o aquellos que tengan una propiedad pública serán serializados. Esto hace necesario realizar una buena planificación en donde sepamos qué partes del estado del objeto realmente precisamos colocar en el archivo. Tenemos que incluir aquellos datos que sean importantes y necesarios para la restauración del estado.



► **Figura 14.** Como la clase no es serializable, tenemos problemas al ejecutar la aplicación y se genera una excepción.

Serialización con composición usando XML

Para llevar a cabo la serialización de una clase que contiene composición haciendo uso del formato XML, se necesitan más cambios que con el formato binario. En este ejemplo, tendremos que modificar la clase **CCompu** y el código de la función **Main()**.

Empecemos con los cambios de la clase **CCompu**; analicemos el código adecuado, el cual se muestra a continuación:

```
[Serializable]
public class CCompu
{
    // Datos necesarios
    private int monitor;
    private CCPU cpu;
    // Constructor
    public CCompu()
    {
        monitor = 20;
        cpu = new CCPU();
    }
    // Sobrecarga del constructor
```



```
public CCompu(int pmonitor, string pprocesador, int mmemoria, double
ddiscoDuro)
{
    // Asignamos valores
    monitor = pmonitor;
    cpu = new CCPU(pprocesador, mmemoria, ddiscoDuro);
}
// Propiedades
public int Monitor
{
    get { return monitor; }
    set { monitor = value; }
}
public CCPU CPU
{
    get { return cpu; }
    set { cpu = value; }
}
public override string ToString()
{
    return string.Format("Monitor {0}, CPU - {1}, {2} GB de memoria, {3} GB
de HD", monitor, cpu.Procesador, cpu.Memoria, cpu.DiscoDuro);
}
```

Si observamos el dato denominado **cpu**, podemos ver que continúa siendo privado, pero recordemos que hemos adicionado una propiedad pública llamada **CPU**. Al hacerlo, permitimos que toda la información que se encuentra en el dato **cpu** quede serializada en el disco y, posteriormente, podamos recuperarla. Si no hacemos esto, el único dato que se serializaría sería **monitor**.

Ahora podemos concentrarnos en la función llamada **Main()**; de la manera usual, adicionamos los namespaces requeridos para el uso de los streams y la serialización, como vemos a continuación:

```
// Namespace necesarios
using System.Xml.Serialization;
using System.IO;
```


El código de la función **Main()** es el siguiente:

```
static void Main(string[] args)
{
    // Datos
    string opcion;
    // Pedimos la opcion al usuario
    Console.WriteLine("1- Salvar objeto, 2- Leer objeto");
    opcion = Console.ReadLine();
    // Salvar el objeto
    if (opcion == "1")
    {
        // Colocamos la informacion / estado del objeto
        CCompu miCompu = new CCompu(24, "i7", 8, 1000);
        Console.WriteLine("Salvando el objeto");
        // Seleccionamos el formateador
        XmlSerializer formateador = new XmlSerializer(typeof(CCompu), new
        Type[] { typeof(CCPU)});
        // Creamos el stream
        Stream miStream = new FileStream("miCPU.cpu", FileMode.Create,
        FileAccess.Write, FileShare.None);
        formateador.Serialize(miStream, miCompu);
        miStream.Close();
    }
    // leer el objeto
    if (opcion == "2")
    {
        Console.WriteLine("Leyendo el objeto");
        // Seleccionamos el formateador
        XmlSerializer formateador = new XmlSerializer(typeof(CCompu), new
        Type[] { typeof(CCPU)});
        Stream miStream = new FileStream("miCPU.cpu", FileMode.Open,
        FileAccess.Read, FileShare.None);
        CCompu miCompu2 = (CCompu)formateador.Deserialize(miStream);
        miStream.Close();
        Console.WriteLine(miCompu2.ToString());
    }
}
```

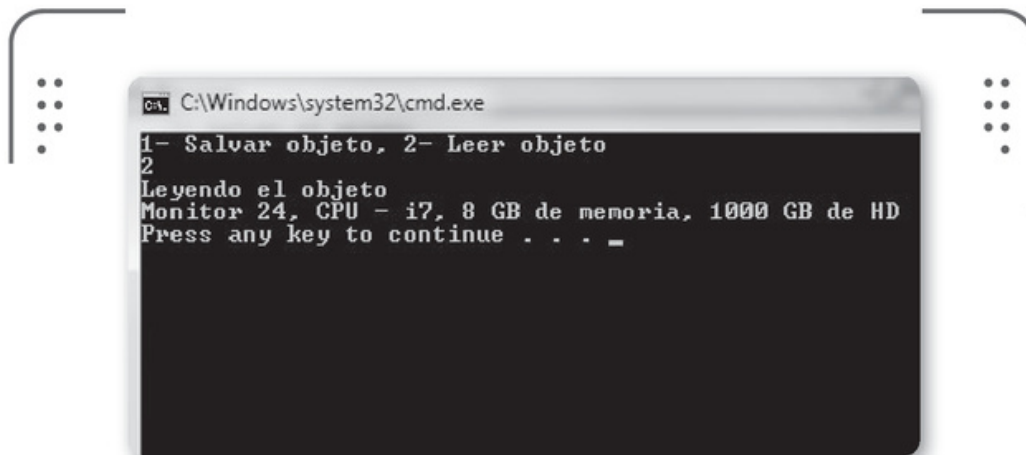
El primer cambio lo encontramos en la sección que corresponde a la serialización, pero en este caso, va más allá del cambio de formateador. Tengamos en cuenta que cuando estamos seleccionando el formato, nos encargaremos de crear una instancia de la clase **XmlSerializer** y la llamamos **formateador**. Podemos darnos cuenta de que el constructor luce diferente, y es necesario entender qué es lo que tenemos ahí. En este caso, el constructor posee dos parámetros. El primero será del tipo que corresponde a la clase principal por serializar, esto es, la clase que contiene a las otras en composición; siempre el primer parámetro debe ser esta clase. El segundo es un arreglo de tipo **Type**. En él debemos listar todos los tipos con los que se lleva a cabo la composición; es decir, las clases que usa la clase principal en su composición. En nuestro ejemplo, la clase principal por serializar es **CCompu** y tiene una composición con **CCPU**. Debido a esto, **CCPU** es colocada dentro del arreglo. Si **CCompu** tuviera composición con más clases, todas ellas quedarían listadas en el arreglo. Recordemos que el resto del código en esta sección es similar a los ejemplos anteriores y ya lo conocemos.

La sección que corresponde a la deserialización presenta un cambio similar. Cuando creamos el objeto de tipo **XmlSerializer**, usamos el constructor de la forma anterior, indicando el tipo principal y listando los tipos con los que se tiene la composición. El resto del código también es similar a los ejemplos anteriores.

Para continuar, podemos proceder a ejecutar la aplicación, ver cómo funciona e, incluso, verificar los contenidos del archivo que se ha creado, en este caso, en formato XML:

```
<?xml version="1.0"?>
<CCompu xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Monitor>24</Monitor>
  <CPU>
    <Procesador>i7</Procesador>
    <Memoria>8</Memoria>
    <DiscoDuro>1000</DiscoDuro>
  </CPU>
</CCompu>
```

El archivo creado es muy fácil de leer e interpretar. Podemos observar la clase **CCompu** y el dato **Monitor** dentro de ella. También, como dato de esta clase se encuentra **CPU**, pero este tiene, a su vez, sus propios datos, que fueron guardados con éxito.



► **Figura 15.** El objeto ha sido serializado y deserializado correctamente. Vemos que incluye información de la clase en composición.

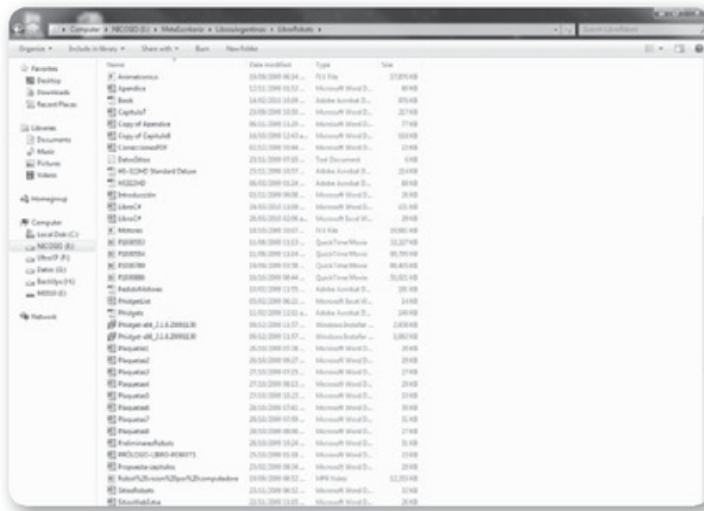
Serialización de varios objetos

Con lo visto hasta aquí, ya conocemos las técnicas básicas de serialización, pero hasta este momento, solo hemos serializado un objeto a la vez, y como sabemos, en la mayoría de nuestros programas, será necesario que realicemos este proceso con varios. La manera de hacerlo también es sencilla y veremos un ejemplo a continuación. En este caso, tendremos tres objetos que serán serializados a disco y, luego, cargados.



TYPE

Es necesario que tengamos en cuenta que **type** es una clase que forma parte del network de .NET, y que se utiliza para representar el tipo en una declaración. De esta forma debemos saber que puede encargarse de representar el tipo para clases, interfaces, arreglos, enumeraciones y también para tipos genéricos. Esta clase tiene varias propiedades interesantes y también métodos que nos permiten obtener información sobre el tipo particular. De esta forma se trata de una clase muy importante y versátil, que debemos tener en cuenta para los proyectos en los cuales estamos trabajando.



► **Figura 16.** Muchos archivos que usamos y encontramos en la computadora tienen un formato binario y son serializaciones de objetos.

Tengamos en cuenta que la técnica más sencilla para lograrlo es realizar la creación de un objeto que se encargue de contener a todos los que deseamos serializar y, después, serializar este objeto que actúa como contenedor. Recordemos que como objeto contenedor es posible acceder a utilizar un **ArrayList** o también cualquier otro **ADT** que nos resulte conveniente para nuestro proyecto.



► **Figura 17.** En Internet podemos encontrar mucha información útil sobre SOAP. Uno de los sitios recomendados es W3C.

Es necesario saber que las clases denominadas **CCPU** y **CCompu** no sufren ningún cambio, ya que solo nos encargaremos de realizar modificaciones en el código de la función **Main()**:


```
static void Main(string[] args)
{
    // Datos
    string opcion;
    // Pedimos la opcion al usuario
    Console.WriteLine("1- Salvar objeto, 2- Leer objeto");
    opcion = Console.ReadLine();
    // Salvar el objeto
    if (opcion == "1")
    {
        // Colocamos la informacion / estado del objeto
        //CCompu miCompu = new CCompu(24,"i7", 8, 1000);
        ArrayList compus = new ArrayList();
        compus.Add(new CCompu(20,"i3",1,300));
        compus.Add(new CCompu(24,"i5",2,500));
        compus.Add(new CCompu(32,"i7",4,1200));
        Console.WriteLine("Salvando el objeto");
        // Seleccionamos el formateador
        XmlSerializer formateador = new XmlSerializer(typeof(ArrayList),
            new Type[] { typeof(CCompu), typeof(CCPU) });
        // Creamos el stream
        Stream miStream = new FileStream("miCPU.cpu", FileMode.Create,
            FileAccess.Write, FileShare.None);
        formateador.Serialize(miStream, compus);
        miStream.Close();
    }
    // leer el objeto
    if (opcion == "2")
    {
        Console.WriteLine("Leyendo el objeto");
        // Seleccionamos el formateador
        XmlSerializer formateador = new XmlSerializer(typeof(ArrayList),
            new Type[] { typeof(CCompu), typeof(CCPU) });
        Stream miStream = new FileStream("miCPU.cpu", FileMode.Open,
            FileAccess.Read, FileShare.None);
        ArrayList compus2 = (ArrayList)formateador.Deserialize(miStream);
        miStream.Close();
    }
}
```

```

for (int n = 0; n < compus2.Count; n++)
{
    CCompu temp = (CCompu)compus2[n];
    Console.WriteLine(compus2[n].ToString());
}
}
}

```

En la opción para llevar a cabo la serialización, hemos comentado la línea donde se instancia el objeto **miCompu** y, en su lugar, hemos creado un **ArrayList** llamado **compus**. A este le adicionamos tres objetos de tipo **CCompu**. A continuación, realizamos la serialización. Notemos que en el constructor de **XmlSerializaer** colocamos como tipo principal **ArrayList**, y en la lista de tipos secundarios, **CCompu** y **CCPU**. Esto es necesario pues el objeto que vamos a serializar es **compus** y este contiene a los objetos de tipo **CCompu**, los cuales poseen composición con **CCPU**. Después de esto, la serialización se efectúa de la manera usual.



► **Figura 18.** Esta imagen muestra el sitio del editor binario que podemos usar (www.hhdsoftware.com/free-hex-editor).

En la sección dedicada a la deserialización, el **formateador** tiene exactamente los mismos cambios. Cuando hacemos uso del método **Deserialize()**, hacemos que los contenidos se guarden en un objeto de tipo **ArrayList** con su respectivo **type cast** y, luego, por medio de un ciclo **for**, mostramos los objetos que corresponden en la consola.



► **Figura 19.** Es importante tener en cuenta que también existen muchos sitios web donde podemos encontrar información sobre XML.

Ahora podemos acceder a probar el programa y también realizar la verificación del archivo XML que se ha creado:



PROBLEMAS CON LA SERIALIZACIÓN



Es importante tener en cuenta que algunas veces pueden ocurrir problemas al llevar a cabo el proceso de serialización o deserialización, por lo que es conveniente colocar el código usando **try catch**. Recordemos que ciertas excepciones que necesitamos conocer son las que mencionamos a continuación: **ArgumentNullException**, **SerializationException** y **SecurityException**.

```
<?xml version="1.0"?>

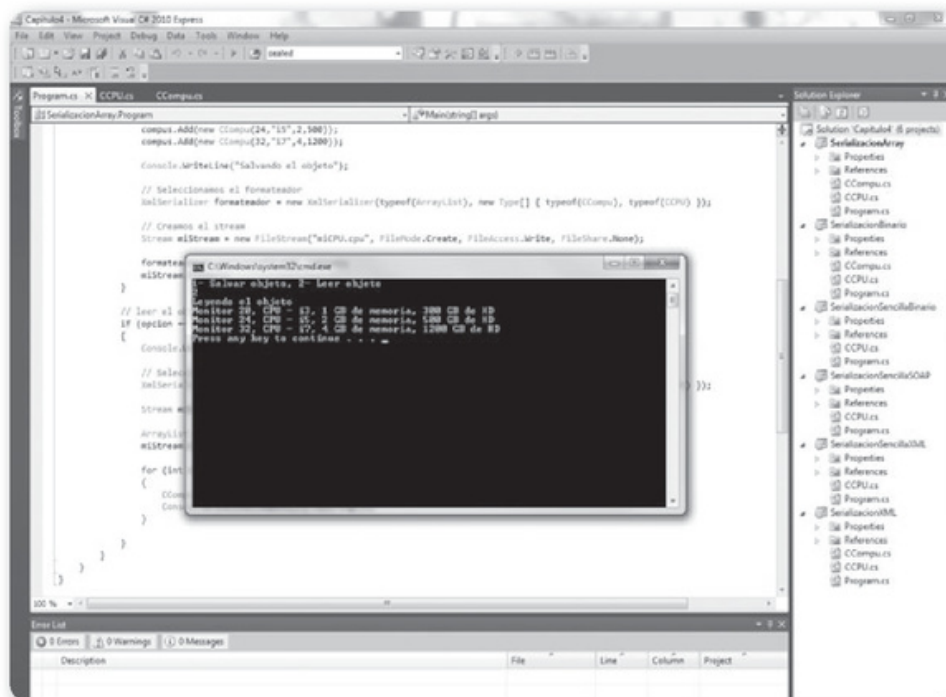
<ArrayOfAnyType xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <anyType xsi:type="CCompu">
    <Monitor>20</Monitor>
    <CPU>
      <Procesador>i3</Procesador>
      <Memoria>1</Memoria>
      <DiscoDuro>300</DiscoDuro>
    </CPU>
  </anyType>
  <anyType xsi:type="CCompu">
    <Monitor>24</Monitor>
    <CPU>
      <Procesador>i5</Procesador>
      <Memoria>2</Memoria>
      <DiscoDuro>500</DiscoDuro>
    </CPU>
  </anyType>
  <anyType xsi:type="CCompu">
    <Monitor>32</Monitor>
    <CPU>
      <Procesador>i7</Procesador>
      <Memoria>4</Memoria>
      <DiscoDuro>1200</DiscoDuro>
    </CPU>
  </anyType>
</ArrayOfAnyType>
```



INFORMACIÓN SOBRE SERIALIZACIÓN



Es interesante saber que un sitio web donde podemos obtener más información sobre el proceso de serialización es el de MSDN, donde se ofrece documentación y artículos sobre este tema. El sitio web principal de la serialización es el que encontramos en la siguiente dirección: <http://msdn.microsoft.com/en-us/library/7ay27kt9%28v=VS.100%29.aspx>.



► **Figura 20.** Hemos visto que, efectivamente, los tres objetos fueron guardados y posteriormente recuperados.

Con esto hemos conocido y analizado cada uno de los elementos básicos de la serialización, y ahora podemos guardar el estado de nuestra aplicación a disco para recuperarlo posteriormente.



RESUMEN



En este capítulo hemos aprendido sobre la serialización, que nos permite guardar el estado de los objetos a disco o a un stream. Tenemos tres formatos para poder hacerlo: binario, SOAP y XML. Es necesario crear un objeto del formato, un stream y hacer uso del método `Serialize()` o `Deserialize()`. Dependiendo del tipo de

Actividades

TEST DE AUTOEVALUACIÓN

- 1 ¿Qué es la serialización?
- 2 ¿Cuáles son los formatos de serialización?
- 3 ¿Cómo funciona el formato binario?
- 4 ¿Qué es SOAP?
- 5 ¿Qué es necesario para usar el formato SOAP?
- 6 ¿Cómo funciona el formato XML?
- 7 ¿Cómo funcionan los datos públicos en la serialización?
- 8 ¿Cómo funcionan los datos privados en la serialización?
- 9 ¿Por qué es necesario usar un type cast en la deserialización?
- 10 ¿Qué parámetros son necesarios en el formato XML?
- 11 ¿Para qué se usa **Type**?
- 12 ¿Cuál es la forma más sencilla de serializar varios objetos?

EJERCICIOS PRÁCTICOS

- 1 Lleve a cabo la serialización de varios objetos en formato binario.
- 2 Realice la serialización de varios objetos en formato SOAP.
- 3 Haga la serialización de objetos que tengan herencia en formato binario.
- 4 Efectúe la serialización de objetos que tengan herencia en formato XML.
- 5 Haga la serialización de varios objetos que sean de clases diferentes.

Introducción a WPF y XAML

WPF presenta una nueva forma en la que podemos crear interfaces de usuario para nuestras aplicaciones gráficas. Pero no es solamente una nueva manera de hacer lo mismo: es un framework que permite tomar ventaja del hardware y llevar a cabo interfaces modernas y avanzadas que serían difíciles de construir con las técnicas tradicionales. En este capítulo analizamos los detalles fundamentales para entender los alcances de WPF y XAML en nuestros desarrollos.

▼ Entender WPF 166

Arquitectura de WPF..... 167

Entender los contenedores..... 169

▼ Los paneles 169

▼ Primera aplicación WPF 194

▼ Resumen..... 197

▼ Actividades..... 198



Entender WPF

Con la versión 3.0 de .NET se introdujo WPF. Las herramientas tradicionales funcionan muy bien si únicamente deseamos construir aplicaciones que usen una forma con varios controles, pero que no proveen funcionalidades extra al usuario. Si queremos construir una interfaz de usuario avanzada, con muchos elementos interactivos, es necesario recurrir a diferentes tecnologías, como **GDI+** y **DirectX**. El

problema reside en que estas no se encuentran directamente relacionadas entre sí, lo que complica el desarrollo de la interfaz.

XAML NOS PERMITE
REALIZAR LA
DEFINICIÓN DE LA
INTERFAZ DE USUARIO

Microsoft estaba consciente de las limitaciones del modelo anterior para la creación de interfaces y de la problemática para poder crear el tipo de interfaz que requieren las nuevas aplicaciones. Por esta razón, decidió crear WPF, sigla de **Windows Presentation Foundation**. Se trata del framework que Microsoft está recomendando para las nuevas aplicaciones tanto de escritorio, **Silverlight** o para

dispositivos móviles. Con WPF, todo lo que necesitamos para construir interfaces de usuario avanzadas queda unificado dentro de un solo framework. Con él mismo podemos trabajar con controles, gráficas 2D y 3D, video, etc. Una de las ventajas que ofrece WPF es el uso de estilos y templates en los controles, con lo cual podemos cambiar el aspecto de la aplicación de modo muy rápido y sencillo.

En WPF se busca llevar a cabo una separación entre la forma en que luce la aplicación y la lógica empleada para controlar dicha aplicación. Para lograrlo, se hace uso de **XAML**. XAML nos permite definir la interfaz de usuario; los programadores que trabajan con aplicaciones web lo encontrarán muy fácil de utilizar. La interfaz definida con XAML es, entonces, controlada con el framework de WPF mediante cualquiera de los lenguajes .NET; en nuestro caso, C#.

XAML está basado en XML y es un lenguaje declarativo. Con él podemos definir y especificar características para clases con una sintaxis basada en XML. Este nos permite hacer cosas como declarar variables o definir propiedades. No es posible definir la lógica del programa con XAML, pero al poder establecer las características de las clases, lo usamos para definir la interfaz de usuario y los

elementos gráficos de nuestra aplicación. La lógica se colocará en C#, y esto es lo que se conoce como **code behind**.

WPF puede utilizarse para crear diferentes tipos de aplicaciones. La forma más usual es la creación de aplicaciones para escritorio. Todas las aplicaciones para desktop que hemos creado con las herramientas habituales pueden hacerse con WPF, adicionando las grandes capacidades interactivas de este.

Otro tipo de aplicación es la conocida como **XBAP**, que corre en un navegador de Internet. Cuando damos el **URI** de la aplicación, esta se descarga en la máquina y se instala. Estas aplicaciones son mucho más robustas y con mejores interfaces que las páginas tradicionales de Internet. El navegador funciona, simplemente, como un contenedor, y lo que se ejecuta es una aplicación WPF dentro de él. El sistema de navegación del navegador es adoptado por la aplicación.

También es posible crear aplicaciones que se conocen como basadas en navegación. Son similares a las de desktop, pero corren dentro de un navegador web y toman sus funcionalidades de navegación. En ellas creamos varias páginas, dentro de cada una de las cuales tenemos una sección de la interfaz de la aplicación. Aunque estas aplicaciones lucen similares a las de la Web, se ejecutan en forma local.

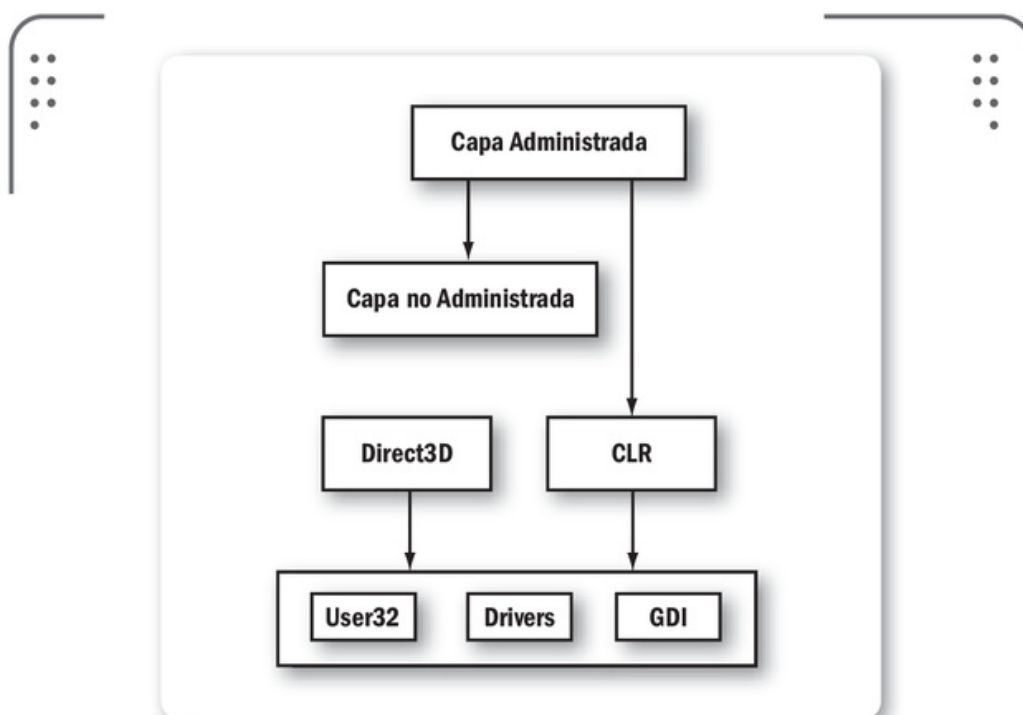
Arquitectura de WPF

Debemos saber que la arquitectura que corresponde a WPF es sencilla, ya que está constituida por diferentes assemblies. Estos assemblies pueden ser clasificados en tres grupos: la capa administrada, la capa no administrada y las APIs centrales.

La función de la capa administrada es dar sustento al framework de WPF y llevar a cabo la comunicación con la capa no administrada. El assembly más importante de esta capa es **PresentationFramework.dll**, cuya función es crear los elementos de alto nivel que usaremos para definir la interfaz de usuario. Estos elementos son los paneles, los controles, las ventanas, etc. Otro assembly importante es **PresentationCore.dll**, que contiene los tipos base más importantes a partir de los cuales se crean los controles y las figuras en **PresentationFramework.dll**; uno de estos tipos es **UIElement**. Un tercer assembly que debemos conocer es **WindowsBase.dll**, que también contiene elementos importantes, cuya característica es poder ser utilizados por afuera del ambiente de WPF.

La capa no administrada también es conocida como **milcore**, acrónimo para *Media Integration Library code*. Esta capa se encuentra, principalmente, en el assembly **milcore.dll**. WPF utiliza Direct3D para llevar a cabo todas las funciones de dibujo. Esto lo hace mucho más rápido y eficiente en el dibujo de las interfaces que la metodología tradicional. La capa no administrada se dedica a traducir los objetos de alto nivel a objetos que puede utilizar directamente. Es el lugar donde se realiza todo el proceso de renderizado de la interfaz. **WindowsCodecs.dll** contiene diversos códecs que son empleados en la parte gráfica, especialmente, aquellos relacionados con gráficas vectoriales.

En las APIs centrales tenemos una colección que provee a WPF de las funcionalidades necesarias para efectuar su trabajo. WPF se apoya en estas APIs, y eso nos facilita el trabajo. Dentro de ellas tenemos a Direct3D, usado para la parte baja y de renderizado en WPF; también es aprovechado gracias a la aceleración por hardware que ofrece. Para la parte de administración de memoria y de los diversos procesos que componen la aplicación, se usa User32.



► **Figura 1.** En esta figura podemos observar un diagrama básico de la forma en que está compuesta la arquitectura que corresponde a WPF.

Entender los contenedores

Cuando creamos una ventana en WPF, esta se deriva de una clase conocida como **ContentControl**. Y describe a un control que puede guardar en su interior cualquier tipo de contenido. La limitante de **ContentControl** consiste en que únicamente puede contener un solo elemento. Otro tipo de contenedor es **HeaderedContentControl**, muy similar a **ContentControl**, pero con un encabezado para su contenido. Con él podemos crear controles como **Expander**, **GroupBox**, etc.

En el contenedor **ItemsControl** es posible colocar cualquier cantidad de contenidos, y puede usarse para crear elementos como un **ListBox** o un **ListView**. Algo similar ocurre con **HeaderedItemsControl**, el cual también es capaz de guardar cualquier cantidad de contenidos, aunque adiciona el encabezado que corresponde.



Los paneles

Los paneles son utilizados como herramienta principal para el diseño de la interfaz gráfica de nuestra aplicación. Por medio de ellos podemos colocar los diferentes controles en las posiciones en que sean necesarios para la interfaces. Hay distintos tipos de paneles, y también es importante conocer cómo funcionan para decidir cuál es el mejor para nuestra interfaz en particular.

El mejor modo de entender cómo funcionan los paneles es programándolos y experimentando con ellos. Para esto, crearemos un programa WPF. Iniciamos Visual Studio 2010 y generamos un proyecto nuevo de C#. Aparece una ventana que muestra los diferentes tipos de proyectos disponibles; allí seleccionamos **WPF Application** y le damos

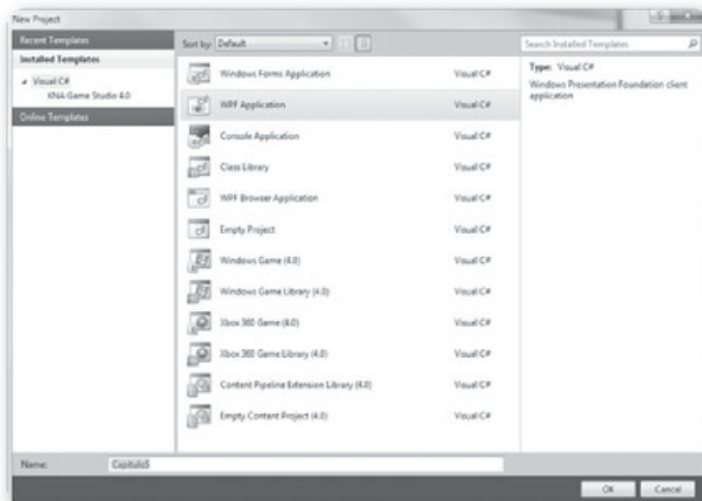


GRÁFICAS VECTORIALES



Debemos tener en cuenta que las gráficas vectoriales tienen la ventaja de no perder calidad sin importar la cantidad de acercamiento que se les dé. La imagen no se guarda en píxeles, sino que se guarda la información geométrica de los objetos que la componen. Al tener la información geométrica, es posible dibujar en cualquier escala, siempre con la mejor calidad.

como nombre de proyecto **Capítulo5**. Al oprimir el botón de **OK**, Visual Studio crea para nosotros la solución y aparece el editor, donde colocaremos el código XAML de nuestra aplicación y podremos ver, al mismo tiempo, cómo luce aquello que estamos programando.



► **Figura 2.** Debemos seleccionar la **Aplicación WPF** para crear el proyecto que usaremos en este capítulo.

GridPanel

Es importante tener en cuenta que empezaremos con el panel más sencillo, conocido como **Grid**, que funciona como una matriz o malla en la que tenemos celdas. Una vez definida la cantidad de renglones y columnas, procedemos a indicar el contenido de cada celda y, de esta forma, ensamblamos la interfaz de usuario.

Empecemos utilizando **Grid** de su manera más sencilla; para esto colocamos el siguiente código XAML en la aplicación:

```
<Window x:Class="Grid.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="MainWindow" Height="350" Width="525">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition/>
            <RowDefinition/>
            <RowDefinition/>
        </Grid.RowDefinitions>
```



```

</Grid.RowDefinitions>

<Grid.ColumnDefinitions>
    <ColumnDefinition/>
    <ColumnDefinition/>
</Grid.ColumnDefinitions>

<!--Colocamos un elemento en cada celda-->
<Border Background="AliceBlue" Grid.Row="0" Grid.Column="0">
    <TextBlock Text="Hola"/>
</Border>
<Border Background="AntiqueWhite" Grid.Row="0" Grid.Column="1">
    <TextBlock Text="Mundo"/>
</Border>
<Border Background="Aqua" Grid.Row="1" Grid.Column="0">
    <TextBlock Text="Hello"/>
</Border>
<Border Background="Aquamarine" Grid.Row="1" Grid.Column="1">
    <TextBlock Text="World"/>
</Border>
<Border Background="Azure" Grid.Row="2" Grid.Column="0">
    <TextBlock Text="--- Hola ---"/>
</Border>
<Border Background="Beige" Grid.Row="2" Grid.Column="1">
    <TextBlock Text="--- Mundo ---"/>
</Border>
</Grid>
</Window>

```



UNIDADES PARA EL TAMAÑO DE LA VENTANA



Tengamos presente que, en forma predeterminada, la unidad con la que se colocan las dimensiones de la ventana son los píxeles, tomados con **96** píxeles por pulgada. También es posible colocar las unidades en pulgadas, centímetros o también puntos. Para hacer uso de estas opciones, simplemente, después del valor, ponemos **in**, **cm** o **pt**, respectivamente.

JUNTO CON WPF
ES NECESARIO
E IMPORTANTE
APRENDER XAML



En primer lugar, encontramos la definición de la clase **Window**, para la cual tenemos algunas propiedades que estamos utilizando. La propiedad **Title** se emplea para colocar el título de la ventana; es el texto que aparece en la parte superior de la ventana de la aplicación. La propiedad **Height** nos sirve para indicar cuál es la altura de la ventana y, de manera similar, **Width** se usa para indicar el ancho de esta.

Ya dentro de la ventana, definimos nuestro **Grid**. Para hacerlo, es necesario indicar cuántos renglones y columnas vamos a utilizar y con qué características. Para definir los renglones, recurrimos a la propiedad **RowDefinitions** de **Grid**. Deseamos tener tres renglones y, por eso, definimos tres objetos de tipo **RowDefinition**.

El **Grid** también tendrá dos columnas, para darnos un total de seis celdas. Las columnas son definidas por medio de la propiedad

ColumnDefinition de **Grid** y, en su interior,

colocamos dos objetos de tipo **ColumnDefinition**. Con esto tenemos las características generales del **Grid**; ahora debemos proceder a colocar elementos en cada una de las celdas que corresponden.

Si deseamos colocar comentarios en el código de XAML, podemos hacerlo por medio del tag `<!-->`. A continuación del segundo guión, ponemos el comentario que deseamos aplicar.

Dentro de cada celda, colocamos un texto, y la celda tendrá un color de fondo único. De esta manera, será sencillo saber cuál es aquella con la que estamos trabajando, y la experimentación resultará más fácil.

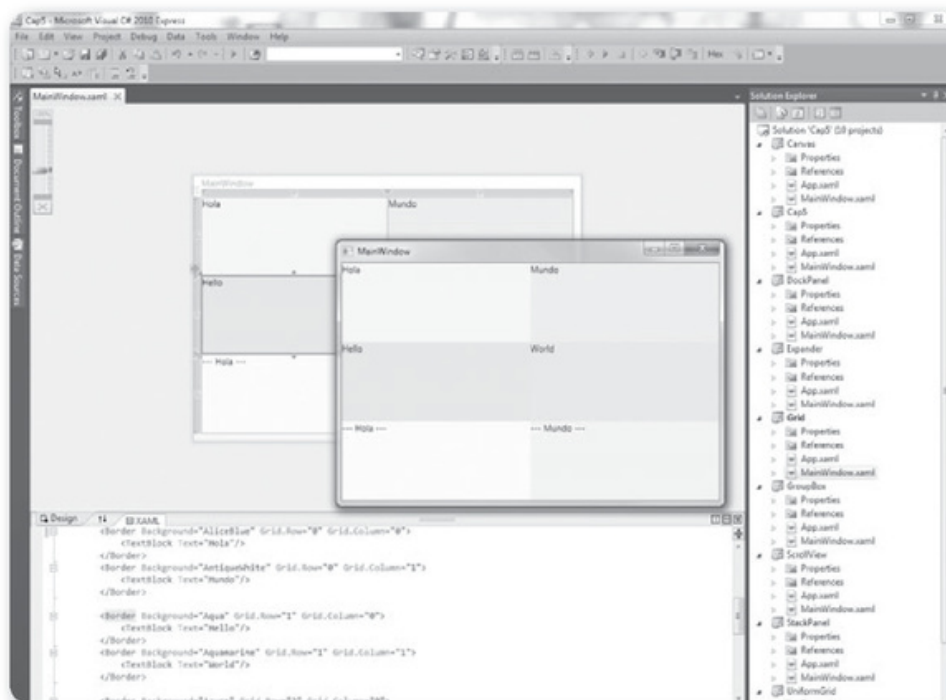
La clase **Border** se usa para dibujar ya sea el fondo de un elemento o un marco alrededor de él. Nosotros la aplicamos para colocar el color de fondo que nos interesa. La propiedad **Background** se usa para indicar el color del fondo. Cuando la escribimos, Visual Studio nos muestra automáticamente una lista de colores que podemos utilizar para la brocha que se empleará en el dibujo de color del fondo.

Nosotros estamos definiendo un borde que va a contener un elemento, pero necesitamos una forma de indicar en cuál de todas las celdas del **Grid** quedará ese borde. Para lograrlo, indicamos las coordenadas de la celda en cuestión, medidas a partir de la esquina superior izquierda del **Grid**. Debemos indicar la celda que vamos a utilizar mencionando en qué renglón y columna se encuentra. El renglón

se indica por medio de la propiedad **Row** del **Grid**, y la columna, por medio de **Column** del mismo **Grid**. Las celdas empiezan a contarse a partir del cero; por ejemplo, la celda superior izquierda se encuentra en **Row 0** con **Column 0**, y la inferior derecha está en **Row 2** con **Column 1**.

En el interior de cada **Border** colocaremos un elemento que es el texto que deseamos mostrar. Para colocar un texto en nuestra interfaz hacemos uso de la clase **TextBlock**. Debemos saber que esta clase es ligera y muy útil cuando queremos mostrar una cantidad pequeña de texto. Con la propiedad **Text**, ponemos el texto correspondiente.

Ya que conocemos estas clases, en nuestro código colocamos un borde con su mensaje en cada una de las celdas del **Grid**. Ahora podemos compilar y ejecutar la aplicación que estamos creando.



► **Figura 3.** Como podemos darnos cuenta, en esta figura observamos la ventana que contiene el **Grid** programado.

El **Grid**, como lo hemos definido, no tiene unas dimensiones particulares para las celdas, por lo que si procedemos a cambiar el tamaño que corresponde a la ventana, veremos que esta sigue quedando dividida en seis celdas con dos columnas.

Podemos llevar a cabo algunas modificaciones al código y ver la forma en que luce el **Grid** que estamos programando. Algunos ejemplos de estos cambios son los siguientes:

```
<Window x:Class="Grid.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="MainWindow" Height="150" Width="525">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition/>
            <RowDefinition/>
            <RowDefinition/>
        </Grid.RowDefinitions>

        <Grid.ColumnDefinitions>
            <ColumnDefinition/>
            <ColumnDefinition/>
        </Grid.ColumnDefinitions>

        <!--Colocamos un elemento en cada celda-->
        <Border Background="AliceBlue" Grid.Row="0" Grid.Column="0"
            BorderBrush="DarkViolet" BorderThickness="3">
            <TextBlock Text="Hola"/>
        </Border>
        <Border Background="AntiqueWhite" Grid.Row="0" Grid.Column="1">
            <TextBlock Text="Mundo"/>
        </Border>
        <Border Background="Aqua" Grid.Row="1" Grid.Column="0">
            <TextBlock Text="Hello"/>
        </Border>
        <Border Background="Aquamarine" Grid.Row="1" Grid.Column="1">
            <TextBlock Text="World"/>
        </Border>
        <Border Background="Azure" Grid.Row="2" Grid.Column="0">
            <TextBlock Text="--- Hola ---"/>
        </Border>
```



```

    <Border Background="Beige" Grid.Row="2" Grid.Column="1">
        <TextBlock Text="--- Mundo ---"/>
    </Border>
</Grid>
</Window>

```

StackPanel

El siguiente panel que vamos a utilizar se conoce como **StackPanel** y funciona, efectivamente, como un stack. Allí podemos adicionar cualquier cantidad de elementos, cada uno de los cuales se agrega al stack después del último. Los elementos se colocan uno después del otro, lo que previene que se encimen entre sí. Es posible colocar en su interior cualquier elemento válido de WPF.

Cuando usamos **StackPanel** podemos seleccionar entre dos orientaciones: **Horizontal** y **Vertical**. Si seleccionamos y usamos **Horizontal**, los elementos se irán agregando de izquierda a derecha, mientras que con **Vertical**, lo harán de arriba hacia abajo. La orientación se controla por medio de la propiedad **Orientation**.

Veamos el siguiente código que ejemplifica el uso de **StackPanel**:

```

<Window x:Class="StackPanel.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="MainWindow" Height="350" Width="410">
    <StackPanel Orientation="Horizontal">
        <Border Background="AliceBlue">
            <TextBlock Text="Hola"/>
        </Border>
        <Border Background="AntiqueWhite" Padding="30">
            <TextBlock Text="Mundo"/>
        </Border>
        <Border Background="Aqua">
            <TextBlock Text="Hello"/>
        </Border>
        <Border Background="Aquamarine" Padding="10,20,50,0">
            <TextBlock Text="World"/>
        </Border>
    </StackPanel>
</Window>

```

```
</Border>
<Border Background="Azure">
  <TextBlock Text="--- Hola ---"/>
</Border>
<Border Background="Beige">
  <TextBlock Text="--- Mundo ---"/>
</Border>
</StackPanel>
</Window>
```

Podemos observar que, en este caso, no necesitamos dar ningún tipo de coordenada para los elementos correspondientes, ya que estos se adicionan en el orden en el que los programamos.

En este ejemplo, estamos haciendo uso de una nueva propiedad conocida como **Padding**, que indica la distancia que debe haber desde un contenido en particular hasta los bordes. En el caso del segundo **Border** se está colocando el código que vemos a continuación:

```
<Border Background="AntiqueWhite" Padding="30">
```

En este ejemplo hemos indicado un **Padding** de **30**, lo que debe dar una distancia mínima de **30** píxeles a partir del mensaje "**Mundo**" hasta los límites del borde. Cuando se coloca un solo valor en el **Padding**, este se usa para la parte superior, inferior, izquierda y derecha.

En el cuarto **Border** tenemos código ligeramente diferente:

```
<Border Background="Aquamarine" Padding="10,20,50,0">
```



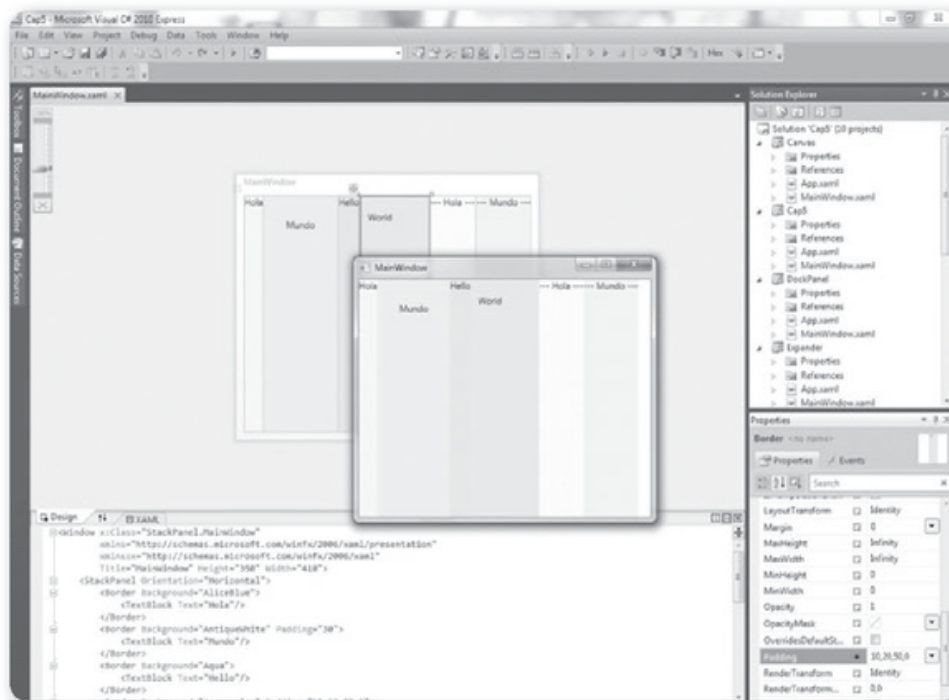
PRONUNCIACIÓN DE XAML



Debemos tener en cuenta que la sigla XAML, tan conocida en la actualidad, puede pronunciarse como una sola palabra o también letra por letra. Eso sí, consideremos que en el primer caso, debe decirse "**zammel**". Cualquiera de las dos formas se considera correcta, aunque en algunos países de Latinoamérica se da preferencia a nombrarla letra por letra. En este sentido es muy importante conocer ambas formas de pronunciación por si nos encontramos con personas de habla inglesa. De esta forma no nos sorprenderá oírlo nombrar de esa forma.

Para este caso, hemos colocado un valor distinto de **Padding** en cada uno de los límites: **10** a la izquierda, **20** en la parte superior, **50** en el extremo derecho y **0** para la parte inferior.

Ahora podemos compilar la aplicación y ver los resultados del código.



► **Figura 4.** En este ejemplo se pone de manifiesto el comportamiento del **StackPanel** con orientación horizontal.

WrapPanel

Otro panel que resulta fácil de utilizar es **WrapPanel**. Es similar al anterior, ya que los elementos se colocan uno después del otro. En el **StackPanel** los elementos se van adicionando de forma indefinida, pero en el **WrapPanel**, lo hacen cuando ya no hay espacio en la ventana, de esta forma el elemento continúa en el siguiente renglón o columna.

WrapPanel también tiene orientación **Horizontal** y **Vertical**, las cuales se controlan por medio de la propiedad **Orientation**. Para tener un poco más de control sobre la forma en que se dibujan los elementos, lo mejor es hacer uso de **Padding**; de esta manera, tenemos, al menos, un tamaño mínimo garantizado en el área que corresponde a **Border**.

El código de una aplicación que usa **WrapPanel** es el siguiente:

```
<Window x:Class="WrapPanel.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="MainWindow" Height="350" Width="525">
  <WrapPanel Orientation="Horizontal">
    <Border Background="AliceBlue">
      <TextBlock Text="Hola"/>
    </Border>
    <Border Background="AntiqueWhite" Padding="30">
      <TextBlock Text="Mundo"/>
    </Border>
    <Border Background="Aqua">
      <TextBlock Text="Hello"/>
    </Border>
    <Border Background="Aquamarine" Padding="10,20,50,0">
      <TextBlock Text="World"/>
    </Border>
    <Border Background="Azure">
      <TextBlock Text="--- Hola ---"/>
    </Border>
    <Border Background="Beige">
      <TextBlock Text="--- Mundo ---"/>
    </Border>
  </WrapPanel>
</Window>
```

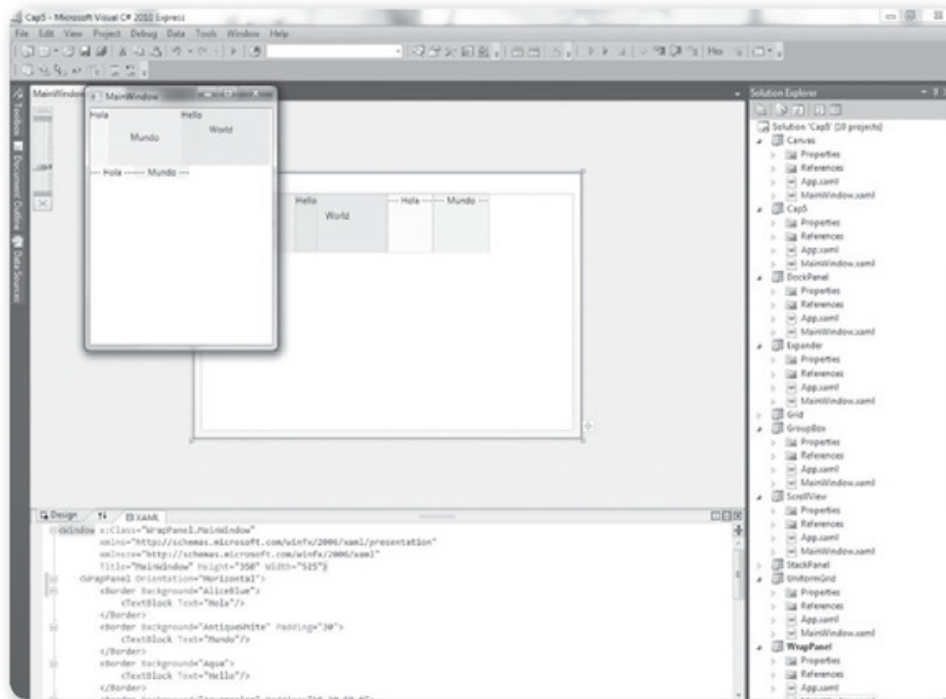
Ahora podemos ejecutar la aplicación. No olvidemos modificar el tamaño de la ventana para ver la forma en que se dibuja **WrapPanel**.



LA ORIENTACIÓN POR DEFAULT



Es importante tener en cuenta que cuando estamos trabajando con **WrapPanel** y no colocamos explícitamente la propiedad de **Orientation**, por default se toma el valor de **Horizontal**. Esto también sucede con otros paneles que tienen propiedades de orientación. Este es un punto para tomar en cuenta si nuestra aplicación tendrá una orientación vertical.



► **Figura 5.** **WrapPanel** dibuja los elementos y, al llegar al final de la ventana, continúa dibujándolos en el siguiente renglón.

DockPanel

El siguiente panel es conocido como **DockPanel**, y es más complicado que los anteriores. Cuando vamos a utilizarlo, es necesario planificar correctamente la interfaz, ya que puede ser un poco complicado prever la forma en que aparecen los diferentes elementos en el panel.

Al colocar un elemento en el **DockPanel**, debemos considerar el espacio que está libre. En él es posible colocar el elemento en la parte superior, inferior, derecha o izquierda. Una vez hecho esto, quedará un nuevo espacio libre, en el que se ubicará el siguiente elemento, otra vez, dependiendo de donde queramos colocarlo. Este proceso se repite hasta que ya no tengamos más elementos para ubicar.

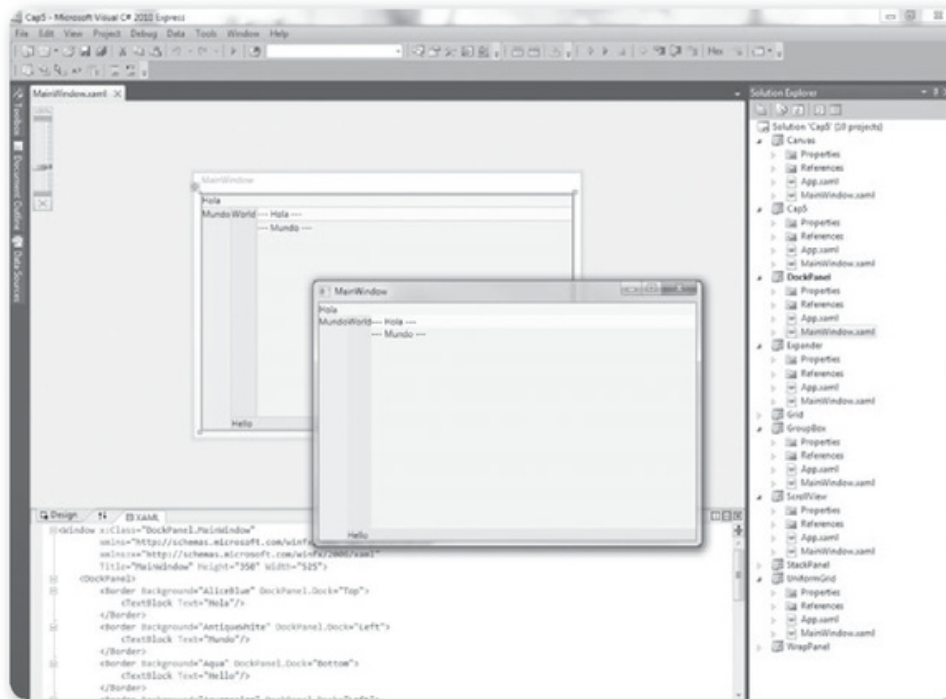
La forma de indicar dónde se colocará el elemento es por medio de la propiedad **Dock** de **DockPanel**. Si deseamos que se coloque en la parte superior del espacio libre, usamos el valor de **Top**, mientras que para la parte inferior se utiliza **Bottom**. En caso de que el objeto deba colocarse a la izquierda, debemos cuidar de poner el valor de **Left**, y para hacerlo en la derecha, usamos **Right**.

Veamos el código que ejemplifica estas opciones:

```
<Window x:Class="DockPanel.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="MainWindow" Height="350" Width="525">
  <DockPanel>
    <Border Background="AliceBlue" DockPanel.Dock="Top">
      <TextBlock Text="Hola"/>
    </Border>
    <Border Background="AntiqueWhite" DockPanel.Dock="Left">
      <TextBlock Text="Mundo"/>
    </Border>
    <Border Background="Aqua" DockPanel.Dock="Bottom">
      <TextBlock Text="Hello"/>
    </Border>
    <Border Background="Aquamarine" DockPanel.Dock="Left">
      <TextBlock Text="World"/>
    </Border>
    <Border Background="Azure" DockPanel.Dock="Top">
      <TextBlock Text="--- Hola ---"/>
    </Border>
    <Border Background="Beige" DockPanel.Dock="Left">
      <TextBlock Text="--- Mundo ---"/>
    </Border>
  </DockPanel>
</Window>
```

Podemos observar que, en este caso, colocamos un borde azul en la parte superior. Entonces, el espacio que nos queda libre es todo lo que se encuentra entre dicho borde y la parte inferior de la ventana. En ese lugar ponemos otro elemento a la derecha. Queda entonces definido un nuevo espacio libre, y se instala un **Border** de color **Aqua** en la parte inferior de este lugar. Luego, se instala otro **Border** a la derecha, y en el espacio libre, otro más en la parte superior. Para finalizar, se pone un último **Border** de color beige situado a la izquierda; como es el último elemento colocado, utiliza todo el espacio libre disponible.

Ahora podemos compilar la aplicación y ver el dibujo del **DockPanel**.



► **Figura 6.** El **DockPane1** se ha dibujado; es conveniente entender la forma en que se ubica cada elemento.

Canvas

El siguiente panel que aprenderemos a usar se conoce como **Canvas** y es sumamente sencillo de manejar. En él nos imaginamos que tenemos un lienzo sobre el cual vamos a dibujar los diferentes controles o elementos de nuestra interfaz. Es el panel más parecido a la forma tradicional de desarrollar las interfaces. Allí colocamos los diferentes elementos dando sus coordenadas dentro del **Canvas**.

Las coordenadas pueden darse referenciadas a la esquina superior izquierda del **Canvas**. Si buscamos un esquema cartesiano de coordenadas similar a **(X,Y)**, podemos utilizar la propiedad **Left** para indicar a qué distancia del borde izquierdo deseamos el elemento, y la propiedad **Top** para definir la distancia a la que el elemento queda del borde superior del **Canvas**. De esta manera, resulta sencillo ubicar los objetos en una posición en particular. Cabe notar que si los elementos se traslapan o se enciman, el orden en que fueron definidos indica cuál se dibuja antes de cuál. Tengamos en cuenta que es posible cambiar esto programáticamente.

Podemos utilizar las propiedades **Height** y **Width** para dar un tamaño definido al control que vamos a utilizar o, también, hacer uso del **Padding** cuando sea necesario y, de esta forma, dar un margen alrededor de los contenidos del control.

El código que nos muestra cómo usar **Canvas** es el siguiente:

```
<Window x:Class="Canvas.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
  <Canvas>
    <Border Background="AliceBlue" Canvas.Left="50" Canvas.Top="70">
      <TextBlock Text="Hola"/>
    </Border>
    <Border Background="AntiqueWhite" Padding="30" Canvas.Left="92"
            Canvas.Top="21">
      <TextBlock Text="Mundo"/>
    </Border>
    <Border Background="Aqua" Padding="15" Canvas.Left="160" Canvas.
            Top="85">
      <TextBlock Text="Hello"/>
    </Border>
    <Border Background="Aquamarine" Padding="10,20,50,0" Canvas.
            Left="176" Canvas.Top="61">
      <TextBlock Text="World"/>
    </Border>
    <Border Background="Azure" Canvas.Left="28" Canvas.Top="36">
      <TextBlock Text="--- Mundo ---" />
    </Border>
  </Canvas>
</Window>
```



DEFINIR LA POSICIÓN DE LOS ELEMENTOS

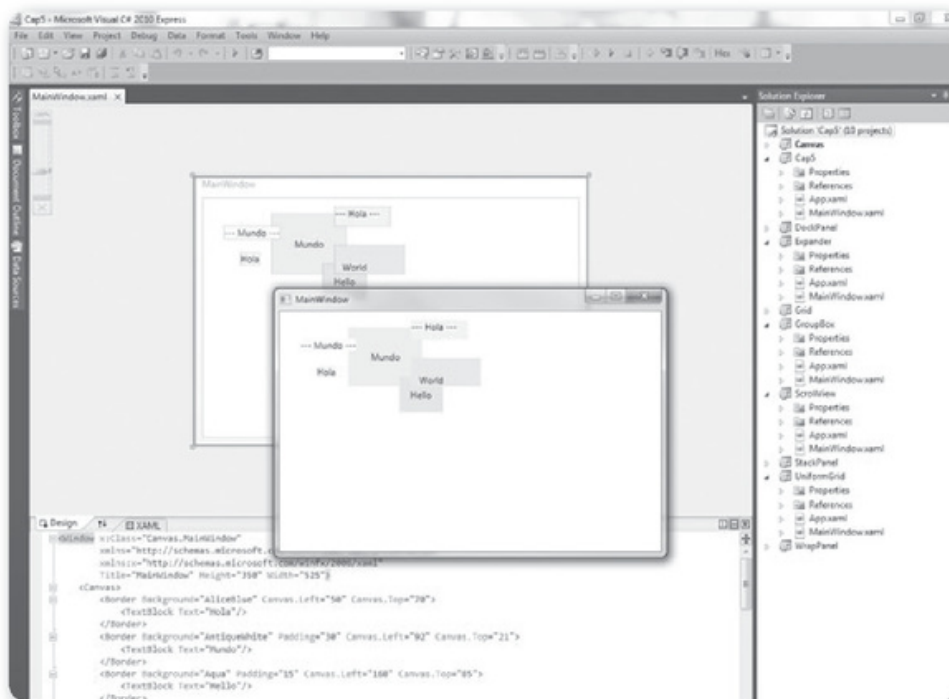


Si bien es bastante cómodo poder colocar las coordenadas del objeto usando las propiedades **Left** y **Top**, ya que nos recuerdan al sistema cartesiano, también es posible indicar la posición en términos de distancia desde **Right** y **Bottom**. La manera de hacerlo depende de nosotros, y podemos usar la más cómoda o la que mejor funcione con nuestra lógica.


```

</Border>
<Border Background="Beige" Canvas.Left="176" Canvas.Top="12"
Height="25" Width="75">
    <TextBlock Text="--- Hola ---" />
</Border>
</Canvas>
</Window>

```

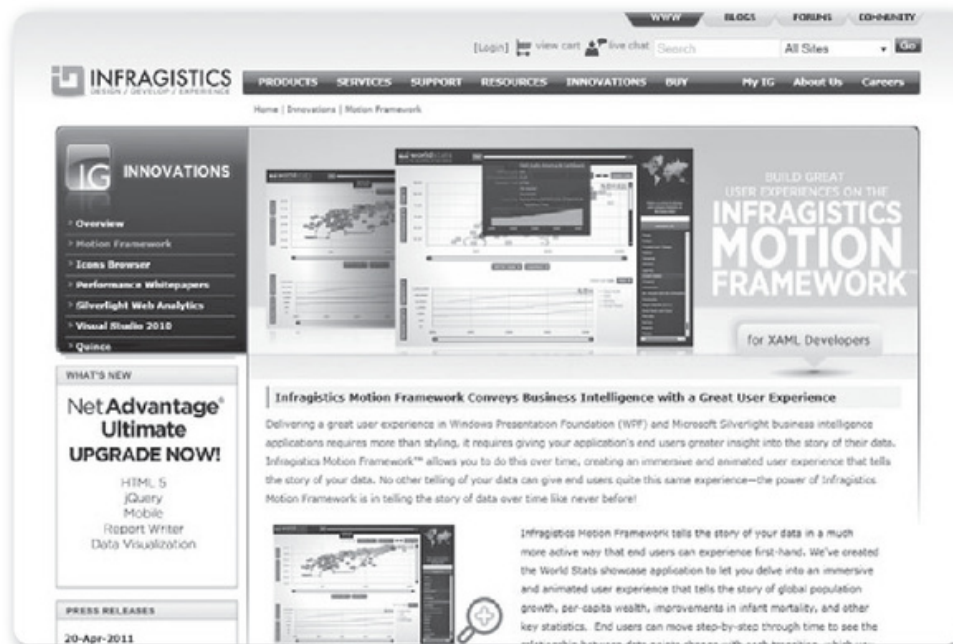


► **Figura 7.** Los diferentes elementos fueron dibujados en las coordenadas especificadas en el código.

UniformGrid

UniformGrid puede ser una forma sencilla y eficiente de sustituir a **Grid**. Es similar a **Grid**, ya que tendrá una serie de celdas. En **UniformGrid** definimos la cantidad de renglones y columnas que queremos tener; automáticamente, la ventana se dividirá de modo uniforme para contener esas celdas. La indicación de cuántas columnas y renglones deseamos se hace por medio de las propiedades **Columns** y **Rows** de **UniformGrid**.

Cuando usamos **Grid**, debemos señalar las coordenadas de la celda que vamos a utilizar para el control. En el caso de **UniformGrid**, las celdas se van rellenando según vamos adicionando los controles. El relleno se realiza de izquierda a derecha y de arriba hacia abajo.



► **Figura 8.** Existen empresas que han desarrollado controles basados en WPF y que podemos utilizar en nuestras aplicaciones. En esta figura observamos el sitio web de una de ellas (www.infragistics.com).

Si lo que necesitamos es que la ventana esté dividida de manera uniforme para todos los controles, **UniformGrid** es la mejor opción, porque nos ayuda a reducir la cantidad de código necesario. La ventana siempre estará dividida de manera uniforme sin importar sus dimensiones, por lo que si cambiamos su tamaño, aún tendremos las mismas divisiones pero con dimensiones diferentes.

Veamos un ejemplo con una aplicación que usa **UniformGrid**:

```
<Window x:Class="UniformGrid.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

```

    Title="MainWindow" Height="350" Width="525">
<UniformGrid Columns="2" Rows="3">
    <Border Background="AliceBlue">
        <TextBlock Text="Hola"/>
    </Border>
    <Border Background="AntiqueWhite" Padding="30">
        <TextBlock Text="Mundo"/>
    </Border>
    <Border Background="Aqua">
        <TextBlock Text="Hello"/>
    </Border>
    <Border Background="Aquamarine" Padding="10,20,50,0">
        <TextBlock Text="World"/>
    </Border>
    <Border Background="Azure">
        <TextBlock Text="--- Hola ---"/>
    </Border>
    <Border Background="Beige">
        <TextBlock Text="--- Mundo ---"/>
    </Border>
</UniformGrid>
</Window>

```

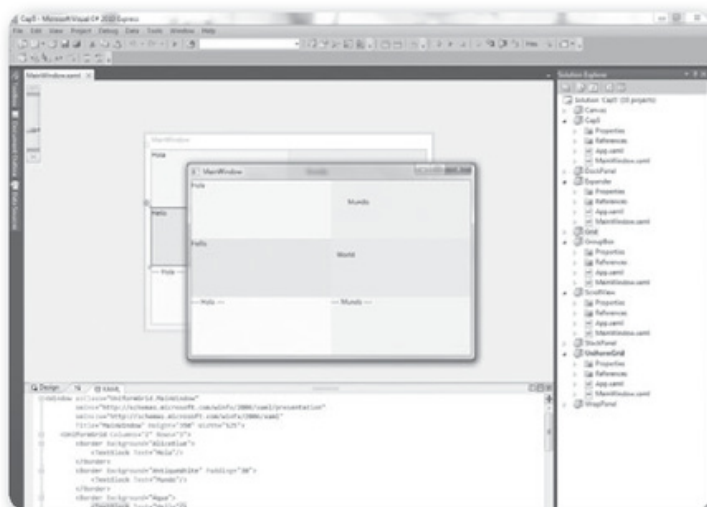
Ahora podemos realizar la ejecución de la aplicación en la cual estamos trabajando. No debemos olvidar que es completamente necesario modificar el tamaño de la ventana cuando se está ejecutando para ver cómo se comporta.



CAMBIAR EL ORDEN DE RELLENO



UniformGrid se encarga de ir rellenando las celdas de izquierda a derecha, pero tengamos en cuenta que si por algún motivo necesitamos que las celdas y los controles se completen en el sentido contrario, es decir, de derecha a izquierda, debemos usar la propiedad denominada **FlowDirection** con el valor **RightToLeft**. Esto es útil si nuestro software lo requiere, por ejemplo, en caso de usar idiomas que se escriben en esa dirección particular. Es importante experimentar cuando menos una vez con esta propiedad para poder observar su comportamiento y sentirnos cómodos usándola.



► **Figura 9.**
UniformGrid dividió la ventana en dos columnas con tres renglones, para dar el mismo tamaño a cada una de las celdas.

ScrollViewer

Ahora que hemos visto los paneles más importantes, es momento de analizar algunos de los controles que nos ayudarán a realizar nuestras interfaces. En nuestros programas es común que la cantidad de elementos para mostrar sea mayor de lo que puede presentar la ventana. En este caso, necesitamos tener un par de controles de scroll que nos permitan movernos dentro de los contenidos.

Incorporar los scrolls en una aplicación de WPF es muy sencillo. Lo que hacemos es crear un objeto de la clase **ScrollViewer**. Esta clase representa un área que tiene elementos visibles, sobre la cual es posible llevar a cabo el scroll. Para demostrar esto, haremos que la aplicación con el panel **Canvas** tenga scrolls. El código se ha modificado de la siguiente manera:

```
<Window x:Class="ScrollView.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="MainWindow" Height="300" Width="300">
    <ScrollViewer HorizontalScrollBarVisibility="Auto">
        <Canvas Height="500" Width="500">
            <Border Background="AliceBlue" Canvas.Left="50" Canvas.Top="70">
                <TextBlock Text="Hola"/>
            </Border>
        </Canvas>
    </ScrollViewer>
```



```

<Border Background="AntiqueWhite" Padding="30" Canvas.Left="92"
Canvas.Top="21">
    <TextBlock Text="Mundo"/>
</Border>
<Border Background="Aqua" Padding="15" Canvas.Left="160" Canvas.
Top="85">
    <TextBlock Text="Hello"/>
</Border>
<Border Background="Aquamarine" Padding="10,20,50,0" Canvas.
Left="176" Canvas.Top="61">
    <TextBlock Text="World"/>
</Border>
<Border Background="Azure" Canvas.Left="28" Canvas.Top="36">
    <TextBlock Text="--- Mundo ---" />
</Border>
<Border Background="Beige" Canvas.Left="176" Canvas.Top="12">
    <TextBlock Text="--- Hola ---" />
</Border>
</Canvas>
</ScrollView>
</Window>

```

Es necesario que empecemos por cambiar el tamaño de la ventana, para que el área del **Canvas** sea mayor que el de la ventana y podamos observar el scroll que corresponde. El tamaño que se ha dado es de **300** tanto para **Height** como para **Width**.

A continuación, tenemos el **ScrollView**; en este caso, estamos indicando que deseamos que la barra de scroll horizontal se presente automáticamente cuando sea necesario. Para esto hacemos uso de la

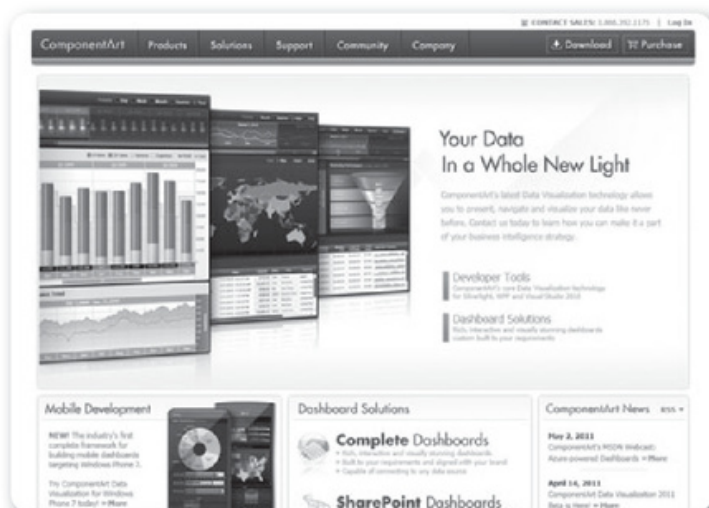


VISIBILIDAD DEL SCROLL



Es muy importante saber que es posible dar diferentes valores a la visibilidad del scroll; por ejemplo, **Disabled** impide que se muestre el scroll, **Auto** se encarga de mostrarlo solo cuando es necesario, y también **Visible** lo muestra en todo momento, aun si no es preciso hacerlo. Cuando estamos diseñando la interfaz de usuario, debemos tomar esto en cuenta.

propiedad **HorizontalScrollBarVisibility** con el valor **Auto**. Entre el tag de apertura y el de cierre de **ScrollViewer** colocamos aquellos elementos que queremos que queden dentro del área de scroll; por eso, en esta sección ubicamos el panel de **Canvas** y sus contenidos.



► **Figura 10.** Los controles basados en WPF nos permiten crear interfaces de usuario avanzadas, para aplicaciones de escritorio o móviles.

Un punto que debemos notar es que a **Canvas** le estamos dando un tamaño en especial, de **500** por **500**, con el fin de hacerlo más grande que la ventana y lograr que se muestren los scrolls. Para indicar el tamaño del **Canvas**, simplemente hacemos uso de las propiedades denominadas **Height** y **Width**, que ya conocemos de otros controles.

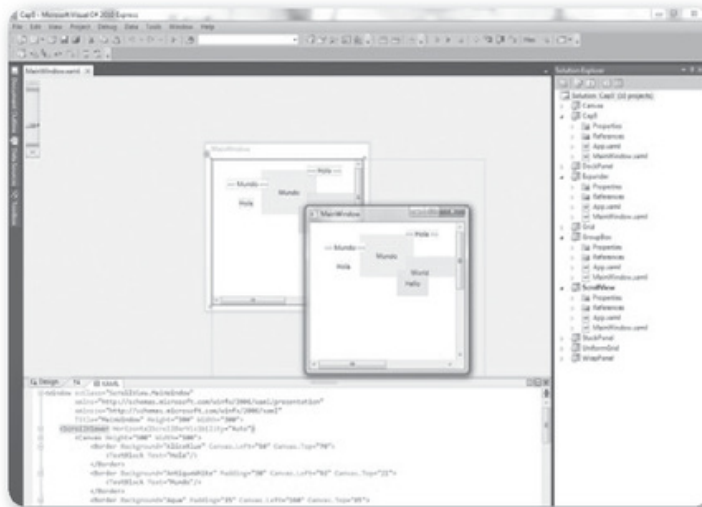
Podemos ejecutar la aplicación y experimentar con las barras de scroll. También es buena idea probar a cambiar el tamaño de la ventana y ver qué ocurre con el scroll correspondiente.



WPF SIN XAML



Debemos tener en cuenta que es posible que accedamos a realizar aplicaciones de WPF sin tener la necesidad de escribir código de XAML. En este caso, se pueden definir los objetos y dar valores a las propiedades directamente en C#. Se trata de una interesante opción si no estamos acostumbrados a XAML, pero debemos recordar que perdemos parte de la flexibilidad que nos permite el código en él. Es bueno conocer esta alternativa de programación, aunque la recomendación es aprender XAML ya que es lo más utilizado y nos otorgará la flexibilidad necesaria para nuestros proyectos.



► **Figura 11.** En esta figura podemos observar las barras de scroll, ya que el **Canvas** es más grande que la ventana.

GroupBox

Debemos tener presente que **GroupBox** es un control que nos permite realizar la tarea de agrupar en su interior diferentes controles, según lo que necesitemos para la aplicación en la cual nos encontramos trabajando. Esta agrupación se despliega visualmente por medio de un marco que envuelve a los controles adecuados y tiene también un título que nos indica qué tipo de contenidos tenemos. El uso de **GroupBox** es muy útil porque permite que los elementos comunes de las interfaces de usuario queden agrupados de una manera en que resulte fácil utilizarlos.



► **Figura 12.** Algunas empresas producen colecciones de controles que podemos utilizar e, incluso, ofrecen pruebas gratis de los mismos.

GroupBox admite que solo coloquemos un elemento en su interior, ya que descende de **HeaderedContentControl**. Si por alguna razón necesitamos poner más de uno, será necesario que coloquemos un panel de los que ya hemos estudiado y, dentro de él, nos encarguemos de incluir todos los elementos que deseamos.

Veamos el código que lleva a cabo esta acción:

```
<Window x:Class="GroupBox.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="MainWindow" Height="211" Width="250">
  <Grid>
    <GroupBox Header="Mi grupo de controles" Margin="30">
      <StackPanel>
        <Button Content="Un boton"/>
        <TextBlock Text="Mi texto"/>
        <TextBox Text="dato"/>
        <Button Content="Otro boton"/>
      </StackPanel>
    </GroupBox>
  </Grid>
</Window>
```

Estamos creando un objeto **GroupBox** y, por medio de la propiedad **Header**, colocamos el texto que servirá como título; en este caso, hemos puesto **Mi grupo de controles**, pero podemos elegir cualquier texto que deseemos. Hacemos uso de otra propiedad, llamada **Margin**, con un valor de **30**, para crear un espacio en blanco alrededor del control desde el borde de su contenedor.



DAR FORMATO A GROUPBOX



Es posible utilizar una gran cantidad de propiedades para hacer lucir el **GroupBox** de la manera en que nosotros necesitamos. Podemos indicar sus dimensiones, color de fondo y del borde, y todas las características de la fuente que será utilizada para la propiedad **Header**. Para conocer la lista completa de propiedades, podemos visitar MSDN.

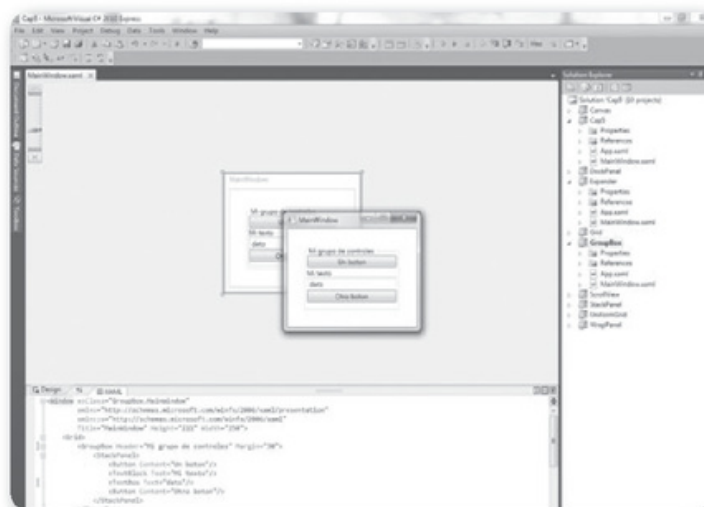


► **Figura 13.** Existen controles de WPF que nos permite trabajar con información geográfica, como en este ejemplo de RadMap.

Ya en el interior, colocamos un **StackPanel**, con el fin de ubicar más de un control allí. En el interior de **StackPanel** ponemos cuatro elementos (algunos de ellos los veremos con más detalle en un ejemplo próximo). En la parte superior ubicamos un botón, luego un texto seguido de un **TextBox** y, en la parte inferior, otro botón. Ahora, ninguno de esos botones tiene funcionalidad, pero luego veremos cómo usarlos.

Por medio de **StackPanel** o de cualquier otro panel podemos diseñar el interior del **StackPanel** y hacerlo lucir según nuestras necesidades.

Ahora podemos acceder a realizar la compilación de la aplicación y posteriormente ver los resultados en la ventana cuando se ejecute.

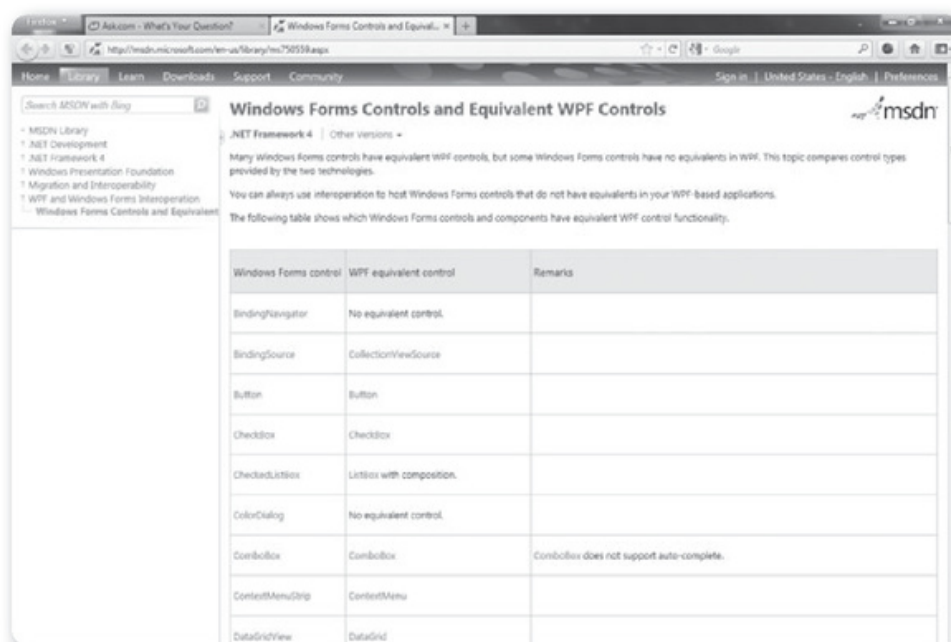


► **Figura 14.** El **GroupBox** se ve claramente en la ventana, encerrando a los controles que hemos definido en su interior.

Expander

El control **Expander** es muy similar a **GroupBox** ya que nos permite agrupar en su interior otros controles relacionados entre sí. La diferencia es que **Expander** puede contraerse o expandirse para mostrar sus contenidos cuando sea necesario. La forma más usual para hacer esto es por medio de un clic del mouse en el título del control.

Al igual que **GroupBox**, solo permite tener un elemento en su interior, por lo que si deseamos colocar más de uno, tendremos que apoyarnos en algún panel y colocar en su interior todo lo que precisemos.



Windows Forms control	WPF equivalent control	Remarks
BindingNavigator	No equivalent control.	
BindingSource	CollectionViewSource	
Button	Button	
CheckBox	CheckBox	
CheckBoxList	ListBox with composition.	
ColorDialog	No equivalent control.	
ComboBox	ComboBox	ComboBox does not support auto-complete.
ContextMenu	ContextMenu	
DataGridView	DataGrid	

► **Figura 15.** En el sitio web de MSDN podemos encontrar una tabla de equivalencias entre los controles tradicionales de las formas y los controles de WPF. Muy útil cuando queremos saber cómo hacer algo.

El código que se presenta a continuación muestra la forma en que podemos utilizar las características de **Expander**:

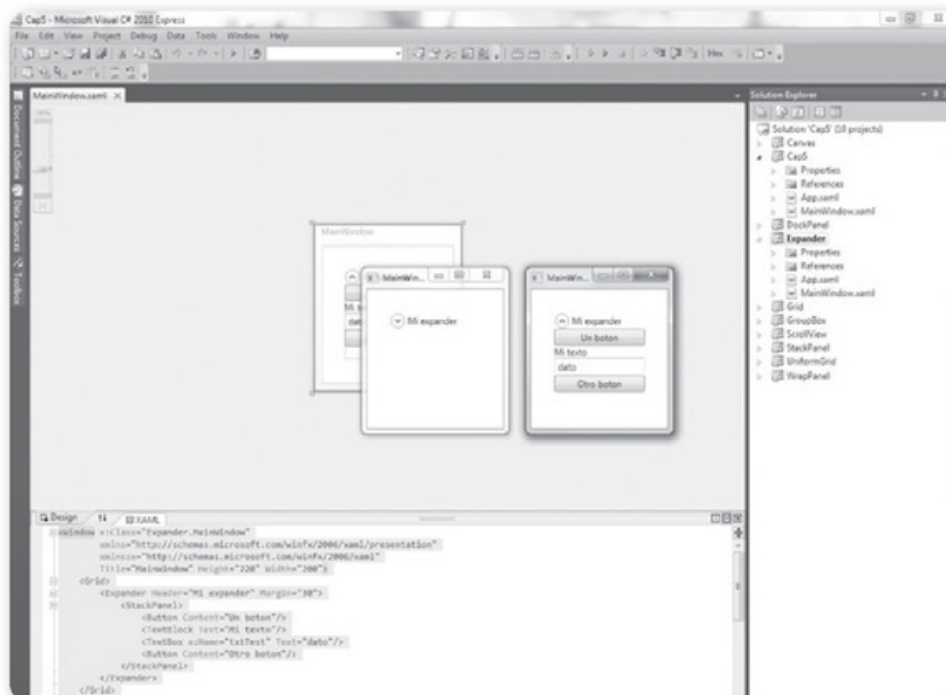
```
<Window x:Class="Expander.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

```

Title="MainWindow" Height="220" Width="200">
<Grid>
  <Expander Header="Mi expander" Margin="30">
    <StackPanel>
      <Button Content="Un boton"/>
      <TextBlock Text="Mi texto"/>
      <TextBox x:Name="txtTest" Text="dato"/>
      <Button Content="Otro boton"/>
    </StackPanel>
  </Expander>
</Grid>
</Window>

```

El código no es muy diferente del visto en el ejemplo anterior, pero si compilamos y ejecutamos la aplicación, veremos que el comportamiento es distinto. No hay que olvidar hacer clic con el botón del mouse sobre el título de **Expander**.



► **Figura 16.** Podemos observar la forma como se comporta el control **Expander**: ya sea mostrando sus contenidos u ocultándose.

Primera aplicación WPF

Es momento de crear nuestra primera aplicación de WPF. Para esto haremos uso de lo aprendido. La aplicación es sencilla: simplemente, crearemos un sumador. Por medio de una interfaz definida en XAML, pediremos al usuario dos valores y, cuando se oprima un botón, el resultado será desplegado en la pantalla. El código es fácil de entender, pero se encarga de sentar las bases para que podamos comprender cómo podemos utilizar todo lo que ya conocemos con una interfaz XAML:

```
<Window x:Class="Cap5.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="MainWindow" Height="88" Width="525">
    <Grid>
        <!--Definimos el grid-->
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto"/>
            <ColumnDefinition Width="100"/>
            <ColumnDefinition Width="*" />
        </Grid.ColumnDefinitions>

        <!-- Colocamos los contenidos de las celdas-->
        <TextBlock Text="Valor a:" Grid.Row="0" Grid.Column="0"/>
```



WRAPPANEL Y LA VENTANA



La forma en que se dibujan los elementos correspondientes a **WrapPanel** en la ventana depende del tamaño que esta tenga. Esto significa que si se cambian las dimensiones de la ventana, los diferentes elementos pueden variar su posición. Hay que tomar esto en cuenta para los diseños en los cuales trabajemos. Un cambio de posición puede destruir nuestro diseño de interfaz.


```
<TextBox x:Name="txtA" Grid.Row="0" Grid.Column="1" />
<Button Content="Calcular" Grid.Row="0" Grid.Column="2"
Click="btnCalcular_click" x:Name="btnCalcular" />
<TextBlock Text="Valor b:" Grid.Row="1" Grid.Column="0"/>
<TextBox x:Name="txtB" Grid.Row="1" Grid.Column="1" />
<TextBlock x:Name="txtResultado" Text="Resultado =" Grid.Row="1"
Grid.Column="2"/>
</Grid>
</Window>
```

Empezamos utilizando un **Grid** para construir la interfaz, que tiene tres columnas y dos renglones. Por medio de **TextBlock**, colocamos el mensaje para **Valor a:**. Luego agregamos un **TextBox**, que nos permite introducir valores desde la interfaz de usuario. Algo que debemos notar es el uso de la propiedad **x:Name**, que nos permite colocar el nombre con el que identificaremos al control. Con este nombre, el código escrito en C# puede tener acceso al control y a sus propiedades. El nombre es **txtA**.

Hemos creado un botón llamado **btnCalcular**, que muestra el texto **Calcular** por medio de la propiedad **Content**. Cuando se oprime el botón, deseamos que se invoque un método que llevará a cabo el procesamiento; es decir, queremos colocar un handler para el evento clic del botón. Hacemos uso de **Click** porque representa el evento que nos interesa, y le colocamos como valor **btnCalcular_click**, que corresponde al nombre del handler para dicho evento.

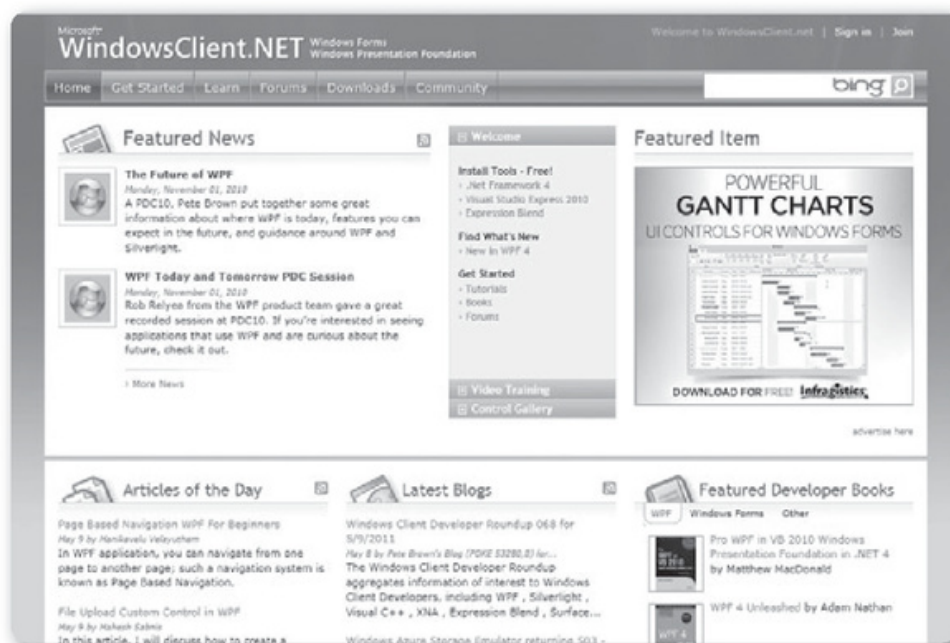
Luego ponemos otro **TextBlock** y posteriormente un **TextBox** que serán usados para el segundo valor que debe introducir el usuario. Un último **TextBlock** es definido, al que le ponemos un nombre porque necesitamos identificarlo, dado que nuestro código en C# lo usará para escribir el resultado de la operación.



DIBUJAR UN MARCO



Es importante tener en cuenta que con la clase llamada **Border** podemos hacer uso de la propiedad denominada **BorderBrush**, la cual nos permite seleccionar el color que tendrá el marco correspondiente de dicho elemento. Junto con esta propiedad, debemos usar también la llamada **BorderThickness**, que indica cuán grueso es el marco que deseamos dibujar.



► **Figura 17.** Microsoft ha creado un sitio web conocido como WindowsClient.NET, que se especializa en la creación de interfaces de usuario para aplicaciones basadas en formas o en WPF.

Ahora es tiempo de colocar el código del handler. La forma más sencilla de hacerlo es, simplemente, dando doble clic sobre el botón para que Visual Studio cree el esqueleto del método por nosotros. Aparece el código C# e incluimos lo siguiente:

```
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
    }

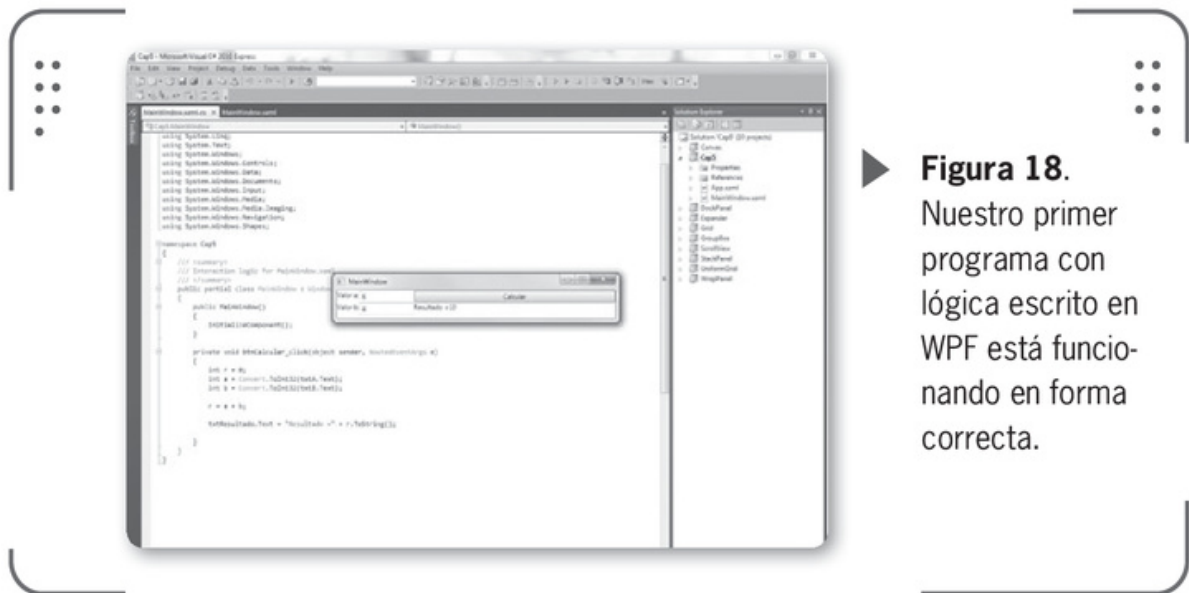
    private void btnCalcular_click(object sender, RoutedEventArgs e)
    {
        int r = 0;
        int a = Convert.ToInt32(txtA.Text);
        int b = Convert.ToInt32(txtB.Text);
```

```

        r = a + b;
        txtResultado.Text = "Resultado =" + r.ToString();
    }
}

```

El código es muy simple, y solo debemos observar que hemos usado los nombres de los controles colocados en el código XAML. Ya podemos proceder a ejecutar y probar la aplicación.



► **Figura 18.** Nuestro primer programa con lógica escrito en WPF está funcionando en forma correcta.

Como ya conocemos las bases que corresponden a WPF, podemos continuar trabajando con elementos más avanzados.



RESUMEN

WPF es una nueva tecnología que nos permite tener, bajo un mismo techo, lo que antes se hacía con diversas APIs no directamente conectadas entre sí. Esto facilita la creación de interfaces de usuario complejas e interactivas, al mismo tiempo que su renderizado es más rápido por ser acelerado por hardware. XAML se emplea para definir la interfaz de usuario. Los elementos definidos en XAML pueden conectarse al código de C#. Los paneles permiten colocar diferentes objetos y tener control sobre la posición de estos.

Actividades

TEST DE AUTOEVALUACIÓN

- 1 Investigue sobre la historia de WPF.
- 2 Modifique el programa de Grid para que cada elemento tenga un tamaño de **Border** específico.
- 3 Modifique los programas para dar el tamaño de la ventana en centímetros.
- 4 Haga un programa que presente nueve cuadros, cada uno de ellos con un grosor y un color de borde diferente.
- 5 Experimente haciendo **StackPanels** horizontales y verticales.
- 6 Realice un programa que use **StackPanels** anidados, uno de ellos horizontal y el otro vertical.
- 7 Experimente con la propiedad **Orientation** de **WrapPanel**.
- 8 Haga un desarrollo que use el **DockPanel** para crear la interfaz del programa de suma visto en este capítulo.
- 9 Haga uso de **Canvas** y **Border** para dibujar una carita en la ventana.
- 10 Cambie el programa anterior para que el código de C# modifique las posiciones de **Border**.

EJERCICIOS PRÁCTICOS

- 1 Haga un programa que lleve a cabo todas las operaciones aritméticas básicas, cuya interfaz esté en XAML.
- 2 Desarrolle un programa que cambie el color de la ventana dependiendo de la selección del usuario.
- 3 Haga un programa que permita jugar “gato” (tic-tac-toe).
- 4 Modifique el programa del ejercicio 1 colocando la selección de las operaciones en un **Expander**.
- 5 Experimente anidando varios paneles.

XAML y las interfaces de usuario

El uso principal de XAML en las aplicaciones de WPF es la definición de las interfaces de usuario. Durante este capítulo nos concentraremos en analizar y aprender en diferentes técnicas que nos permitirán mejorar la vista y el diseño de la interfaz. De esta forma obtendremos un conocimiento más acabado sobre el desarrollo de las interfaces de usuario para nuestras aplicaciones.

▼ Efectos y brochas para el fondo de Border	200
Degradado lineal.....	204
Degradado radial	213
Control de marcos.....	219

▼ Creación de una aplicación que controle a la interfaz	228
▼ Resumen.....	233
▼ Actividades.....	234



➤ Efectos y brochas para el fondo de Border

Empezaremos por aprender diferentes efectos que podemos utilizar con los **Border**. El conocimiento que adquirimos sobre **Border** bien puede usarse en otros controles dentro de la forma, pero por su propia naturaleza, es más sencillo explicarlo con él.

Podemos emplear distintos efectos para la parte del fondo del control y, también, hay algunos que podemos utilizar para su borde. Además de los efectos, existe una gran cantidad de propiedades que nos permitirán controlar prácticamente todos los parámetros de despliegue del control.

Empecemos por experimentar con la forma en que podemos controlar el borde, para lo cual en este momento nos enfocaremos en el fondo. Para hacerlo, vamos a crear una nueva aplicación WPF en Visual Studio, en la que colocaremos diversos **Border**. A cada uno de ellos le modificaremos sus propiedades para entender qué controla cada una.



► **Figura 1.** En el sitio web Real World Software Development podemos encontrar muchas herramientas para trabajar con XAML.

El código completo de este ejemplo se muestra a continuación, pero nos concentraremos de manera específica en cada uno de los **Border**:

```
<Window x:Class="Border.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="MainWindow" Height="342" Width="525">
<ScrollViewer>
    <StackPanel>
        <!--Borde normal basico-->
        <Border Width="150" Height="100">
            <TextBlock Text="Borde Normal"/>
        </Border>
        <!--Borde normal con marco-->
        <Border Width="150" Height="100" BorderBrush="Black" Border
            Thickness="2">
            <TextBlock Text="Borde Normal con marco"/>
        </Border>
        <!--Borde con relleno-->
        <Border Width="150" Height="100">
            <!--Colocamos una brocha solida al background-->
            <Border.Background>
                <SolidColorBrush Color="DeepSkyBlue"/>
            </Border.Background>
            <TextBlock Text="Borde relleno"/>
        </Border>
        <!--Borde con relleno lineal vertical-->
        <Border Width="150" Height="100">
```



CONFIGURAR LOS CONTROLES



Debemos tener en cuenta que Microsoft Visual Studio nos provee de herramientas gráficas que permiten editar y controlar las diferentes propiedades de despliegado de los controles. Si no conocemos el código necesario para modificar alguna propiedad o si deseamos trabajar más rápido, podemos utilizarlas. La lista completa de propiedades puede consultarse en MSDN.

```
<!--Colocamos una brocha de degradado lineal-->
<Border.Background>
  <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
    <!--Colocamos las paradas del degradado-->
    <GradientStop Color="Red" Offset="0"/>
    <GradientStop Color="Lime" Offset="0.5"/>
    <GradientStop Color="Blue" Offset="1.0"/>
  </LinearGradientBrush>
</Border.Background>
<TextBlock Text="Borde relleno vertical"/>
</Border>
<!--Borde con relleno lineal horizontal-->
<Border Width="150" Height="100">
  <!--Colocamos una brocha de degradado lineal-->
  <Border.Background>
    <LinearGradientBrush StartPoint="0,0" EndPoint="1,0">
      <!--Colocamos las paradas del degradado-->
      <GradientStop Color="Red" Offset="0"/>
      <GradientStop Color="Lime" Offset="0.5"/>
      <GradientStop Color="Blue" Offset="1.0"/>
    </LinearGradientBrush>
  </Border.Background>
  <TextBlock Text="Borde relleno horizontal"/>
</Border>
<!--Borde con relleno lineal horizontal variacion-->
<Border Width="150" Height="100">
  <!--Colocamos una brocha de degradado lineal-->
  <Border.Background>
    <LinearGradientBrush StartPoint="0.5,0" EndPoint="1,0">
      <!--Colocamos las paradas del degradado-->
      <GradientStop Color="Red" Offset="0"/>
      <GradientStop Color="Lime" Offset="0.5"/>
      <GradientStop Color="Blue" Offset="1.0"/>
    </LinearGradientBrush>
  </Border.Background>
  <TextBlock Text="Borde relleno horizontal"/>
</Border>
```



```

<!--Borde con relleno lineal diagonal-->
<Border Width="150" Height="100">
  <!--Colocamos una brocha de degradado lineal-->
  <Border.Background>
    <LinearGradientBrush StartPoint="0,0" EndPoint="1,1">
      <!--Colocamos las paradas del degradado-->
      <GradientStop Color="Red" Offset="0"/>
      <GradientStop Color="Lime" Offset="0.4"/>
      <GradientStop Color="Red" Offset="0.6"/>
      <GradientStop Color="Blue" Offset="1.0"/>
    </LinearGradientBrush>
  </Border.Background>
  <TextBlock Text="Borde relleno diagonal"/>
</Border>
<!--Borde relleno radial-->
<Border Width="150" Height="100">
  <Border.Background>
    <RadialGradientBrush GradientOrigin="0.5,0.5" RadiusX="0.5"
    RadiusY="0.5">
      <GradientStop Color="Lime" Offset="0"/>
      <GradientStop Color="Blue" Offset="1"/>
    </RadialGradientBrush>
  </Border.Background>
  <TextBlock Text="Borde relleno radial"/>
</Border>
<!--Borde relleno radial variacion-->
<Border Width="150" Height="100">
  <Border.Background>
    <RadialGradientBrush GradientOrigin="0.3,0.1" RadiusX="0.5"
    RadiusY="0.5">
      <GradientStop Color="Lime" Offset="0"/>
      <GradientStop Color="Blue" Offset="1"/>
    </RadialGradientBrush>
  </Border.Background>
  <TextBlock Text="Borde relleno radial origen"/>
</Border>
<!--Borde relleno radial variacion-->

```

```

<Border Width="150" Height="100">
  <Border.Background>
    <RadialGradientBrush GradientOrigin="0.5,0.5" RadiusX="0.1"
      RadiusY="0.9">
      <GradientStop Color="Lime" Offset="0"/>
      <GradientStop Color="Blue" Offset="1"/>
    </RadialGradientBrush>
  </Border.Background>
  <TextBlock Text="Borde relleno radial radios"/>
</Border>
<!--Borde relleno radial-->
<Border Width="150" Height="100">
  <Border.Background>
    <RadialGradientBrush GradientOrigin="0.5,0.5" RadiusX="0.5"
      RadiusY="0.5">
      <GradientStop Color="Lime" Offset="0.5"/>
      <GradientStop Color="Blue" Offset="0.7"/>
    </RadialGradientBrush>
  </Border.Background>
  <TextBlock Text="Borde relleno radial offset"/>
</Border>
</StackPanel>
</ScrollView>
</Window>

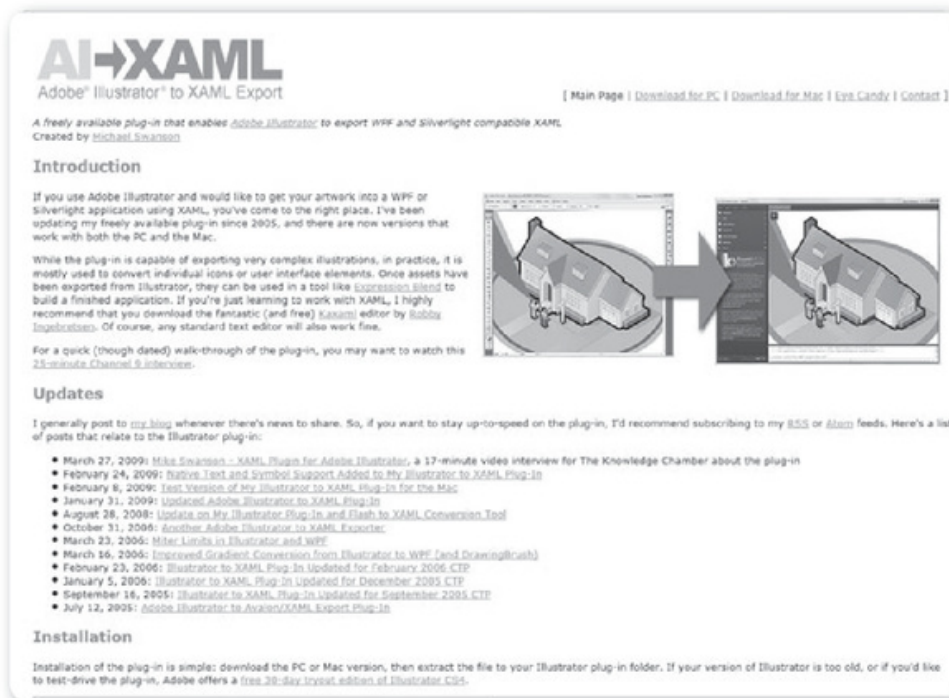
```

La aplicación es sencilla. Estamos creando una ventana para nuestra aplicación que tendrá una altura de **342** y un ancho de **525**. Como vamos a presentar muchos elementos en su interior, lo primero que hacemos es colocar un **ScrollView**, ya que tendremos que movernos a través de la ventana. Para guardar todos los elementos que vamos a mostrar, hacemos uso de un **StackPanel**, aunque también podremos usar otro tipo de panel.

Degradado lineal

En nuestro ejemplo, comenzamos con tres **Border** que son similares a

El primer **Border** es muy sencillo, porque solo damos sus dimensiones; se dibuja como un rectángulo blanco. Para el segundo creamos un marco de color negro por medio de la propiedad **BorderBrush** y le damos un grosor de **2** haciendo uso de **BorderThickness**.



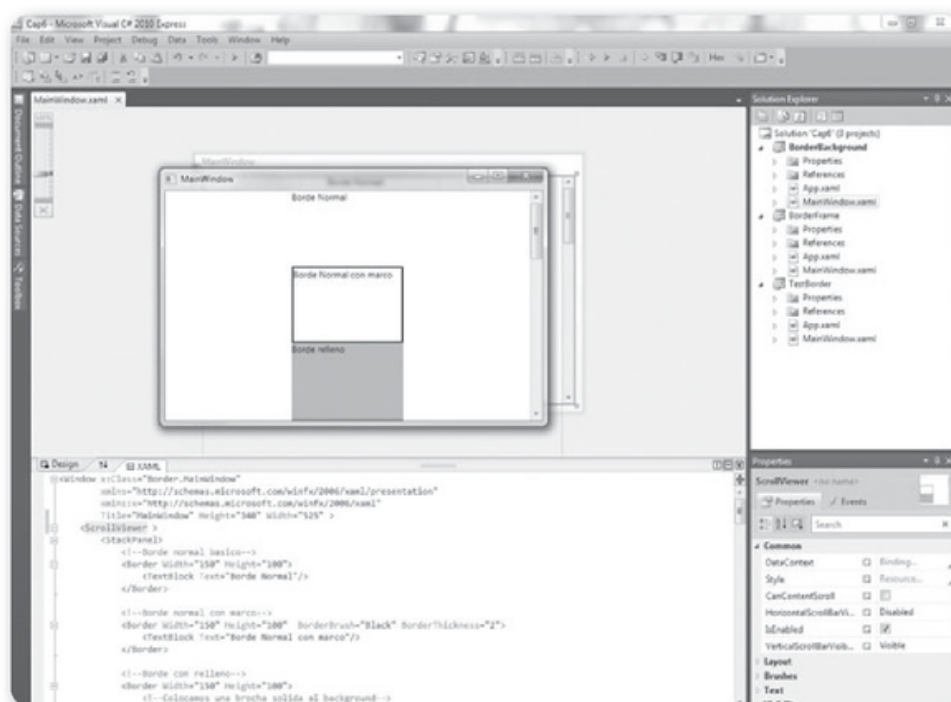
► **Figura 2.** Si lo deseamos, es posible llevar a cabo el arte de nuestra aplicación con Adobe Illustrator y después exportarlo a XAML con este plugin.

Como podemos darnos cuenta, el tercer **Border** es más interesante; a continuación analizamos el código que corresponde:

```
<!--Borde con relleno-->
<Border Width="150" Height="100">
  <!--Colocamos una brocha solida al background-->
  <Border.Background>
    <SolidColorBrush Color="DeepSkyBlue"/>
  </Border.Background>
  <TextBlock Text="Borde relleno"/>
</Border>
```

En este ejemplo, hacemos uso del tag **<Border.Background>**, usado para colocar el tipo de brocha, con sus efectos, que deseamos usar para rellenar el área dentro de él. La brocha en cuestión es **SolidColorBrush**, la cual pinta de un solo color el interior del **Border**. El que hemos seleccionado en este caso es **DeepSkyBlue**.

Si ejecutamos ahora la aplicación, podremos observar los tres primeros **Border** que no sirven de referencia.



► **Figura 3.** Como vemos en esta imagen, los **Border** con los elementos básicos han sido colocados en la forma que corresponde.



FORMATOS PARA COLOCAR EL COLOR

Como sabemos, hasta este momento hemos estado utilizando el nombre del color para indicar nuestra selección de color que corresponde al **Border**. Sin embargo, es interesante tener en cuenta que existen siete maneras de indicar el color que deseamos utilizar para nuestro proyecto. Muchas de estas formas de colocar el color usan valores hexadecimales para los componentes, incluido alpha en dos casos. Así nos damos cuenta de que la selección de color es un proceso no estático.

El siguiente **Border** es más interesante, porque vamos a utilizar más de un color en él; el código adecuado es el siguiente:

```
<!--Borde con relleno lineal vertical-->
<Border Width="150" Height="100">
  <!--Colocamos una brocha de degradado lineal-->
  <Border.Background>
    <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
      <!--Colocamos las paradas del degradado-->
      <GradientStop Color="Red" Offset="0"/>
      <GradientStop Color="Lime" Offset="0.5"/>
      <GradientStop Color="Blue" Offset="1.0"/>
    </LinearGradientBrush>
  </Border.Background>
  <TextBlock Text="Borde relleno vertical"/>
</Border>
```

El **Border** tendrá un degradado vertical. Todo va a suceder dentro de **Background**. Necesitamos colocar un tipo de brocha especial que se ocupe de producir este tipo de efecto: la brocha en cuestión es **LinearGradientBrush**. Tenemos dos propiedades que estamos usando con ella: **StartPoint** indica las coordenadas donde inicia el degradado, y **EndPoint**, las coordenadas donde finaliza. Estas coordenadas no son píxeles, sino que, más bien, vemos la brocha como una unidad. Esto quiere decir que la esquina superior izquierda siempre será la coordenada **0,0**, y la inferior derecha, **1,1**, sin importar el tamaño en píxeles del **Border**. En este caso, estamos manteniendo el valor de **X** constante y solo modificamos **Y**, lo que nos da un degradado vertical del borde superior hasta el borde inferior.



WPF Y 3D



Es necesario tener en cuenta que WPF nos permite crear interfaces de usuarios que tengan elementos de tres dimensiones, gracias a la conexión que tiene con Direct3D. De esta forma, nuestras aplicaciones pueden hacer uso de 3D de una manera más sencilla y rápida. Es necesario conocer un poco sobre Direct3D para realmente sacar provecho de esta opción.

Para poder hacer el degradado, necesitamos indicar los colores que utilizaremos y el rango en el que ese color va a actuar. Por medio de **GradientStop** podemos conseguirlo. En nuestro ejemplo, tenemos tres **GradientStop**, pues deseamos utilizar tres colores en el degradado. Para indicar el color se hace uso de la propiedad **Color**, y podemos dar el nombre de él directamente o los valores que definen al color.

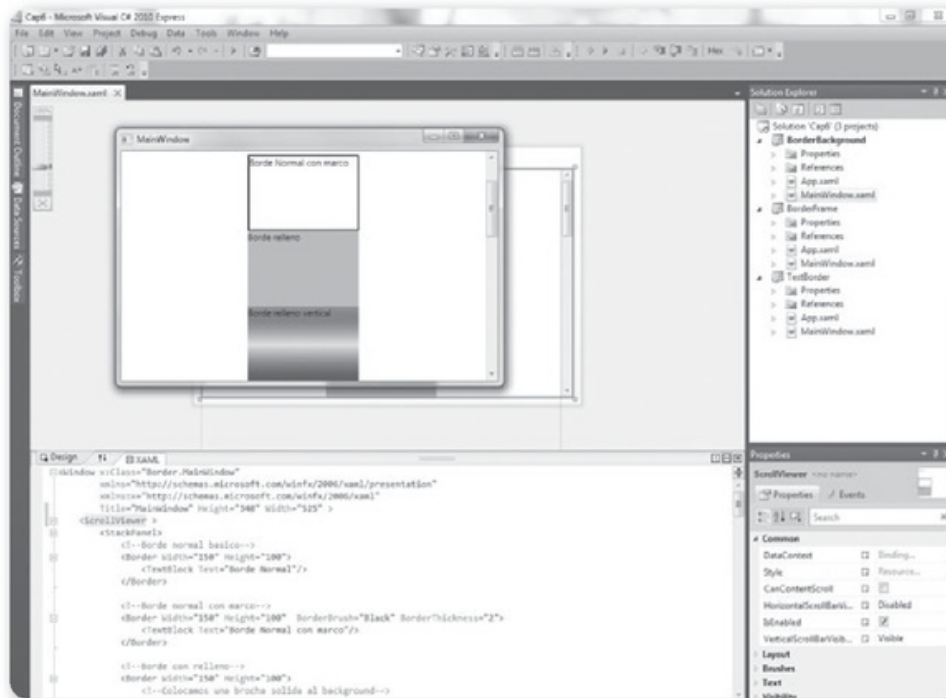


► **Figura 4.** Tengamos en cuenta que también es posible hacer la exportación de archivos creados con Flash a XAML.

El control sobre el rango que va a cubrir el color es un poco más complicado de usar. Podemos imaginar que la propiedad **Offset** nos indica el punto en el cual ese color en particular será mostrado al 100%, es decir, sin mezcla de los tonos aledaños. La posición del **Offset** es independiente del tamaño del **Border** y podemos manejarla como un porcentaje. Si colocamos el valor de **0.5** en el **Offset** de algún color, significa que ese tendrá su 100% a la mitad del recorrido del gradiente en el control. Si damos el valor de **0.3**, tendrá su 100% aproximadamente a un tercio del recorrido del gradiente en el control. El valor **0** indica que el 100% está al inicio del recorrido del gradiente, y el valor **1.0**, que

se encuentra al final. A partir de la posición indicada con **Offset**, el color empezará a desvanecerse o mezclarse con sus vecinos, desde luego, respetando el 100% del color vecino o los límites del **Border**.

Comparemos la forma en que este **Border** funciona en relación a la brocha sólida, para lo cual ejecutamos la aplicación.



► **Figura 5.** El borde con degradado muestra perfectamente los tres colores y la forma en que cambia el color de manera paulatina.

Empecemos a hacer experimentos con este tipo de brocha. El siguiente **Border** tiene cambios, ya que tendrá el gradiente en sentido horizontal:



CONTROLES PROPIOS

Tengamos en cuenta que al igual que con cualquier aplicación del sistema operativo Windows, podemos crear nuestros propios controles. Estos pueden llevar a cabo nuestra propia lógica o, simplemente, pueden ser modificaciones o adiciones a los ya existentes. Si creamos controles propios, lo mejor es hacerlo en una librería, para poder reutilizarlos en otro momento.

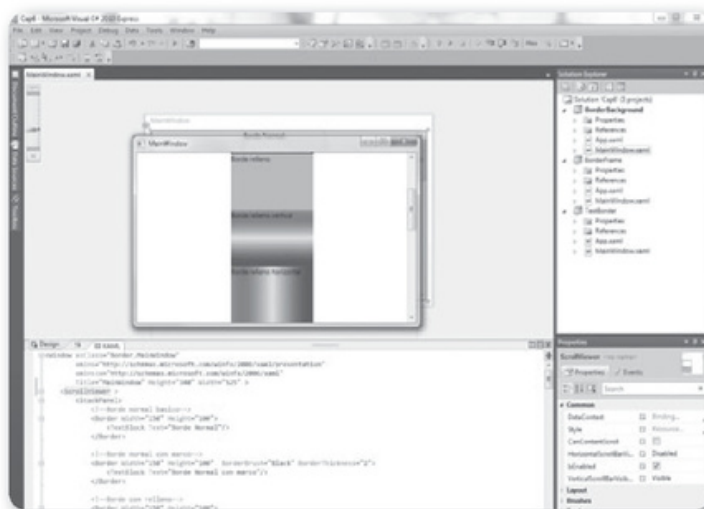

```

<!--Borde con relleno lineal horizontal-->
<Border Width="150" Height="100">
  <!--Colocamos una brocha de degradado lineal-->
  <Border.Background>
    <LinearGradientBrush StartPoint="0,0" EndPoint="1,0">
      <!--Colocamos las paradas del degradado-->
      <GradientStop Color="Red" Offset="0"/>
      <GradientStop Color="Lime" Offset="0.5"/>
      <GradientStop Color="Blue" Offset="1.0"/>
    </LinearGradientBrush>
  </Border.Background>
  <TextBlock Text="Borde relleno horizontal"/>
</Border>

```

Este **Border** no es muy distinto del anterior, pero podemos observar que el valor de **EndPoint** ha sido modificado. Ahora mantenemos constante el valor de **Y**, en tanto que el de **X** va desde **0** hasta **1**. Debemos tener en cuenta que esto significa que el recorrido del gradiente es de izquierda a derecha. De esta forma sabemos que el valor **0** en **Offset** es el límite izquierdo del **Border**, y el valor **1** corresponde al límite derecho. El color verde lima se encuentra a la mitad del **Border** en el sentido horizontal.

Veamos el comportamiento de este **Border** con degradado horizontal al ejecutar nuestra aplicación.



► **Figura 6.** El degradado ahora recorre el **Border** en el sentido horizontal; los tres colores aún están presentes.

Tengamos en cuenta que aún es posible que accedamos a experimentar más con **LinearGradientBrush**. Las propiedades **StartPoint** y **EndPoint** no solo nos sirven para colocar de manera vertical y horizontal el degradado: podemos realizar un mejor control de la forma en que este se lleva a cabo.

A continuación, veamos el siguiente **Border**:

```
<!--Borde con relleno lineal horizontal variacion-->
<Border Width="150" Height="100">
  <!--Colocamos una brocha de degradado lineal-->
  <Border.Background>
    <LinearGradientBrush StartPoint="0.5,0" EndPoint="1,0">
      <!--Colocamos las paradas del degradado-->
      <GradientStop Color="Red" Offset="0"/>
      <GradientStop Color="Lime" Offset="0.5"/>
      <GradientStop Color="Blue" Offset="1.0"/>
    </LinearGradientBrush>
  </Border.Background>
  <TextBlock Text="Borde relleno horizontal"/>
</Border>
```

Podemos notar que el degradado correrá de forma horizontal, pero en este caso, hemos colocado el punto de inicio del eje **X** en **0.5**. Con esto nos encargamos de indicar que el degradado empieza a la mitad del control y, a partir de ese punto, se lleva a cabo según los valores y colores colocados en **GradientStop**. Del inicio del **Border** hasta la mitad, se rellenará con un color sólido; en este caso, el primer color de los **GradientStop**, es decir, rojo.



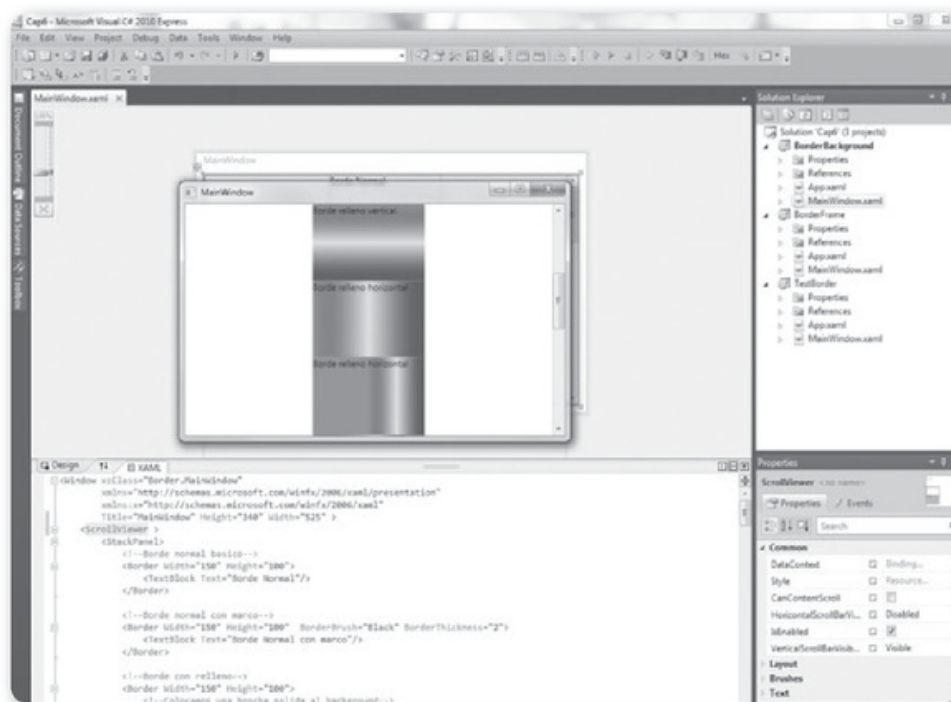
MODIFICAR EL DIBUJO DEL GRADIENTE



Cuando hacemos uso de **LinearGradientBrush** tenemos dos formas en las cuales realizar el interpolado del color en el dibujo del gradiente. Es importante recordar que esto se hace por medio de la propiedad denominada **ColorInterpolationMode**. El valor que se presenta como predeterminado es

SRColorInterpolationMode. Es importante saber que podemos intentar...

Luego de realizar las tareas mencionadas, accedemos a observar cómo luce el **Border** en la imagen que se presenta a continuación:



► **Figura 7.** Al cambiar la coordenada de inicio, podemos controlar el lugar exacto donde queremos que inicie el degradado en el **Border**.

El uso de **StartPoint** y **EndPoint** también nos permite colocar degradados con diferentes ángulos para obtener resultados distintos. Un ejemplo fácil e interesante de realizar es poner un degradado en forma diagonal, como se indica a continuación:



COORDENADAS EN PÍXELES



Es interesante tener en cuenta que aunque en los ejemplos de este libro no estamos usando las coordenadas correspondientes a **LinearGradientBrush** en píxeles, es posible hacerlo sin complicaciones. Debemos recordar, al usar píxeles, nuestras unidades se presentan como absolutas. La propiedad que nos permite cambiar esto es **MappingMode**. Si usamos píxeles, es importante considerar los efectos que se obtendrán en monitores con mayor resolución.

```
<!--Borde con relleno lineal diagonal-->
<Border Width="150" Height="100">
  <!--Colocamos una brocha de degradado lineal-->
  <Border.Background>
    <LinearGradientBrush StartPoint="0,0" EndPoint="1,1">
      <!--Colocamos las paradas del degradado-->
      <GradientStop Color="Red" Offset="0"/>
      <GradientStop Color="Lime" Offset="0.4"/>
      <GradientStop Color="Red" Offset="0.6"/>
      <GradientStop Color="Blue" Offset="1.0"/>
    </LinearGradientBrush>
  </Border.Background>
  <TextBlock Text="Borde relleno diagonal"/>
</Border>
```

Cuando hemos colocado degradados horizontales y verticales, una de las coordenadas permanece constante en el inicio y el fin. Si deseamos tener degradados en diferentes ángulos, la manera más sencilla de hacerlo es variando tanto las coordenadas de **X** como las de **Y**. En nuestro ejemplo, vemos que el punto de inicio es **0,0** y el final, **1,1**; no hemos mantenido constante ninguno de los dos ejes. Como ambos se modificaron en la misma proporción, el degradado aparece a **45** grados. Para mejorar el efecto, en este caso usamos cuatro colores.

Degradado radial

Ahora podemos experimentar con otro tipo de brocha de degradado: una que dibuja el degradado de forma radial. Esto quiere decir que tendremos un punto central donde inicia el degradado y, a medida que

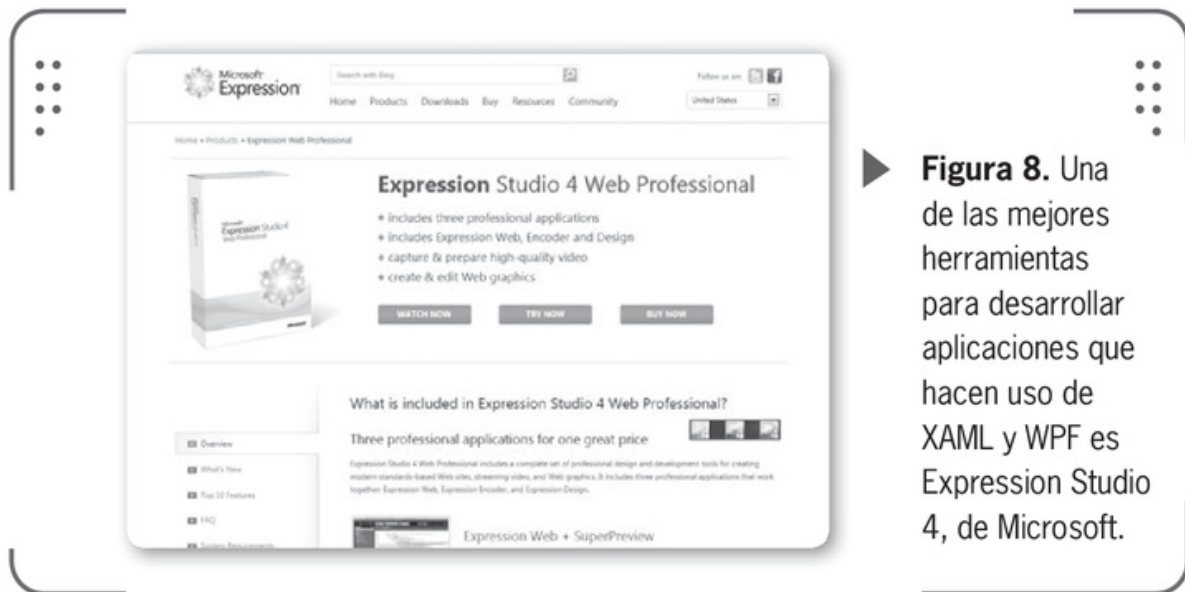


RADIALGRADIENTBRUSH



La brocha de degradado radial comparte algunas propiedades con la de degradado lineal. Es importante verificar la documentación para poder usarlas correctamente y mejorar el control que tenemos sobre el efecto. Algunas de las propiedades más importantes de esta brocha son: **Center**, **ColorInterpolationMode**, **MappingMode**, **Opacity** y **Transform**.

nos alejamos de este punto, el color con el que se dibuja cambia. Esto se puede imaginar como un degradado que se dibuja de forma circular como el sol, anillos o halos. Recordemos que este degradado es conocido como **RadialGradientBrush** y su uso es sencillo. Veremos diferentes variaciones para entender su funcionamiento.



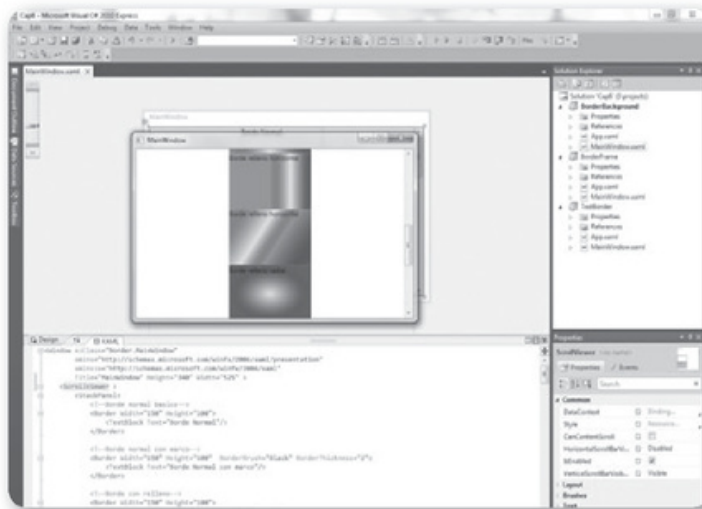
► **Figura 8.** Una de las mejores herramientas para desarrollar aplicaciones que hacen uso de XAML y WPF es Expression Studio 4, de Microsoft.

El primer **Border** con este tipo de degradado tiene el siguiente código:

```
<!--Borde relleno radial-->
<Border Width="150" Height="100">
  <Border.Background>
    <RadialGradientBrush GradientOrigin="0.5,0.5" RadiusX="0.5"
      RadiusY="0.5">
      <GradientStop Color="Lime" Offset="0"/>
      <GradientStop Color="Blue" Offset="1"/>
    </RadialGradientBrush>
  </Border.Background>
  <TextBlock Text="Borde relleno radial"/>
</Border>
```

Podemos colocar el lugar donde nos interesa ubicar el origen del degradado, a partir del cual este empezará a dibujarse. Para hacerlo, usamos la propiedad **GradientOrigin**. En este caso, estamos colocando el origen justo en el centro del **Border**.

Debemos saber que hay un par de propiedades más, que son **RadiusX** y **RadiusY**. Estas colocan el radio del color más externo del degradado. El radio se maneja independientemente en cada eje, para poder tener más flexibilidad. Para colocar los colores del degradado, usamos una vez más **GradientStop**; las propiedades son las mismas que en los ejemplos anteriores. Al ejecutar el código, encontramos un bonito degradado radial que va del verde al azul.

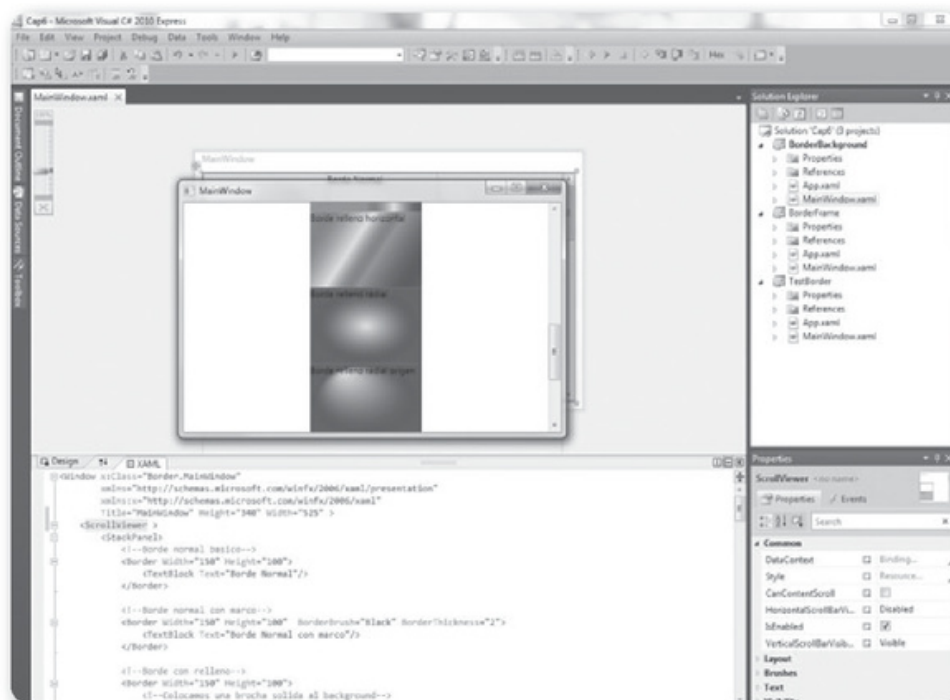


► **Figura 9.** Este es un ejemplo del degradado radial más sencillo que podemos tener, en el que se usan solamente dos colores.

Para seguir avanzando comenzaremos a realizar la tarea de experimentar con este tipo de brocha, adicionando el siguiente **Border**; podemos ver un claro ejemplo a continuación:

```
<!--Borde relleno radial variacion-->
<Border Width="150" Height="100">
  <Border.Background>
    <RadialGradientBrush GradientOrigin="0.3,0.1" RadiusX="0.5"
      RadiusY="0.5">
      <GradientStop Color="Lime" Offset="0"/>
      <GradientStop Color="Blue" Offset="1"/>
    </RadialGradientBrush>
  </Border.Background>
  <TextBlock Text="Borde relleno radial origen"/>
</Border>
```

En este ejemplo hemos decidido modificar el lugar donde está el origen y mantener las demás propiedades igual. El origen ahora se sitúa en **0.3,0.1**. Esto lo coloca cerca del borde superior y un poco hacia la izquierda. A partir de ahí, se empieza a dibujar el degradado. El efecto es diferente del que produce el **Border** anterior.



► **Figura 10.** El punto a partir del cual se empieza a dibujar el degradado se ha modificado, por lo que el efecto es un poco diferente del anterior.

Podemos continuar haciendo experimentos con **Border**; ahora llevamos a cabo el siguiente cambio:



CONSTRUCTOR DE RADIALGRADIENTBRUSH

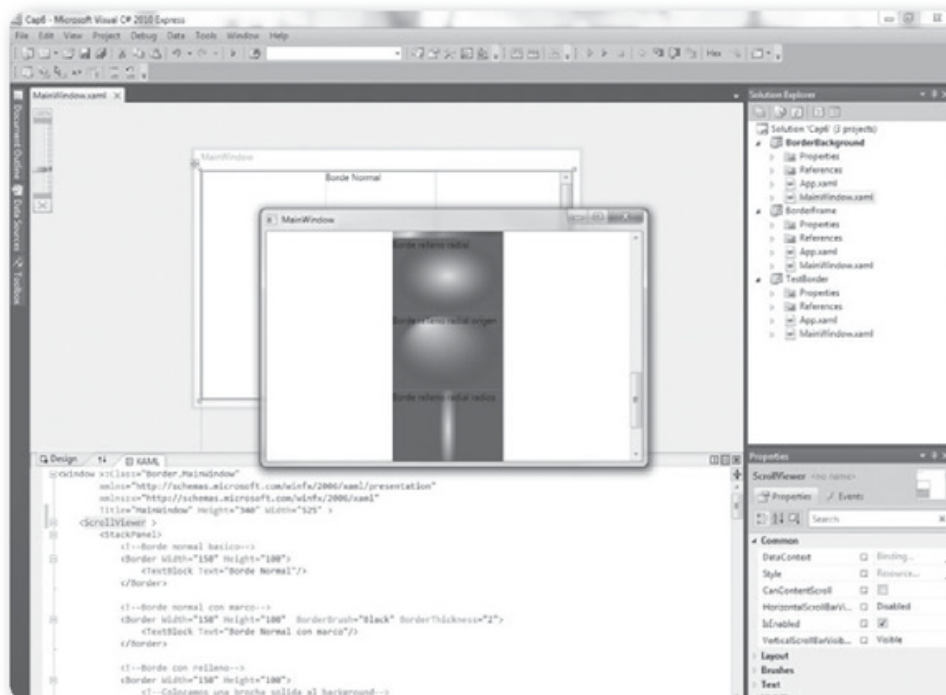
El constructor conocido como **RadialGradientBrush** tiene tres sobrecargas. Una de ellas nos permite pasar como parámetro una colección correspondiente a **GradientStop** que incluya todos los colores por utilizar. Otra versión necesita dos parámetros, que son el color de inicio y el color final del degradado, en caso de que únicamente utilizemos dos tonos.

```

<!--Borde relleno radial variacion-->
<Border Width="150" Height="100">
  <Border.Background>
    <RadialGradientBrush GradientOrigin="0.5,0.5" RadiusX="0.1"
      RadiusY="0.9">
      <GradientStop Color="Lime" Offset="0"/>
      <GradientStop Color="Blue" Offset="1"/>
    </RadialGradientBrush>
  </Border.Background>
  <TextBlock Text="Borde relleno radial radios"/>
</Border>

```

El código es muy similar al del primer ejemplo de degradado radial, pero hemos cambiado el valor de las propiedades **RadiusX** y **RadiusY**. Con los valores que colocamos, solamente tendremos un radio de **0.1** en el eje **X** para llevar a cabo el degradado, y de **0.9** en **Y**. Esto nos da como resultado una figura en forma de elipse con el eje mayor.



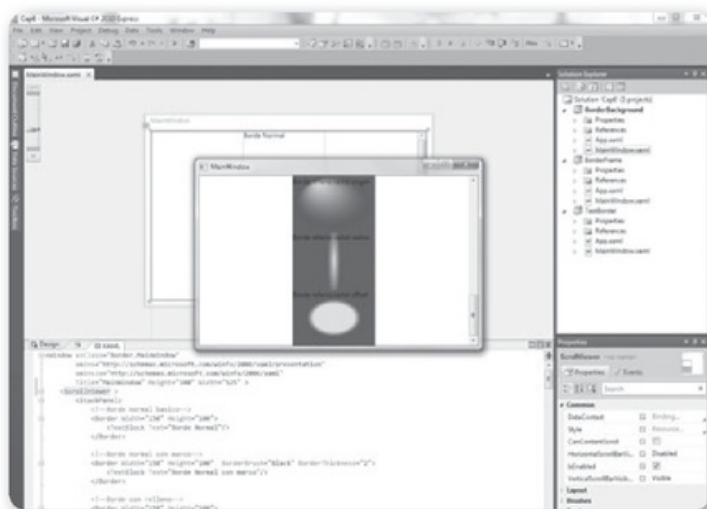
► **Figura 11.** Por medio de las propiedades **RadiusX** y **RadiusY**, podemos modificar la figura creada por el degradado.

La propiedad de **Offset** también nos ayuda a realizar el control de la forma en que se lleva a cabo el degradado radial. Podemos experimentar al hacer los siguientes cambios:

```
<!--Borde relleno radial-->
<Border Width="150" Height="100">
    <Border.Background>
        <RadialGradientBrush GradientOrigin="0.5,0.5" RadiusX="0.5"
RadiusY="0.5">
            <GradientStop Color="Lime" Offset="0.5"/>
            <GradientStop Color="Blue" Offset="0.7"/>
        </RadialGradientBrush>
    </Border.Background>
    <TextBlock Text="Borde relleno radial offset"/>
</Border>
```

En el tipo de brocha anterior, la cual se realiza mediante el **Offset** a partir del límite del **Border**; debemos encargarnos de utilizar el origen como el punto de referencia correspondiente.

En el código anterior pudimos darnos cuenta de que colocamos, para el color verde lima, un **Offset** que corresponde a **0.5**. Como es el primer color desde el origen hasta la ubicación que corresponde a **0.5** de distancia de ese origen, se dibuja un color sólido. El efecto de estos cambios se aprecia muy bien en la **Figura 12**.



► **Figura 12.** Gracias al uso de **Offset**, ahora podemos colocar una zona con color sólido dentro de nuestro degradado radial.

Control de marcos

Hasta el momento, a lo largo de este capítulo, hemos estado trabajando con la forma en que el **Border** se rellena; sin embargo, también es posible trabajar con el marco y obtener algunos efectos que hagan lucir más atractiva nuestra aplicación.

Para continuar, empecemos a trabajar con efectos de sombra alrededor del **Border**. Para esto creamos un proyecto WPF nuevo y comenzamos a adicionar algo de código como el siguiente:

```
<Window x:Class="BorderFrame.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="1092" Width="525">
  <ScrollViewer>
    <StackPanel>
      <!--Borde con sombra-->
      <Border Width="150" Height="100" Background="IndianRed"
            Margin="25,25,25,25">
        <!--Colocamos el efecto para el borde-->
        <Border.Effect>
          <DropShadowEffect BlurRadius="10" Color="Blue" Direction="220"
            Opacity="0.5" RenderingBias="Quality" ShadowDepth="10"/>
        </Border.Effect>
        <TextBlock Text="Sombra"/>
      </Border>
    </StackPanel>
  </ScrollViewer>
</Window>
```



USO DE EFECTOS



Debemos tener en cuenta que la propiedad **Effect** se hereda de **UIElement**, por lo que podemos utilizarla con cualquier elemento que descienda de esta clase. Esta clase es una de las bases de WPF. Es recomendable leer la documentación sobre ella, pues tiene una gran cantidad de métodos y propiedades empleados en los diferentes elementos de la interfaz de usuario.

EL USO CORRECTO DE BORDER PUEDE INCREMENTAR LA CALIDAD DE NUESTRA INTERFAZ



El código es sencillo: estamos creando un **Border** con dimensiones de **150** por **100**, y color rojo en el fondo. Colocamos un margen de **25** alrededor del **Border** para hacer notar la sombra, sin que los elementos posteriores procedan a dibujarse sobre ella.

Dentro del **Border** usaremos una propiedad nueva llamada **Effect**, que permite colocar o aplicar efectos. En este caso, vamos a utilizarla para crear una sombra. Empezamos con el efecto conocido como **DropShadowEffect**.

Este efecto crea sombras y tiene varias propiedades que empezaremos a explorar y experimentar. La sombra se desvanece o luce borrosa cerca de su borde. La propiedad **BlurRadius** controla, precisamente, el tamaño de esta zona.

En este primer ejemplo colocamos un valor de **10**; posteriormente experimentaremos cambiando los valores. Para indicar el color de la sombra, usamos la propiedad **Color**. Podemos colocar el color de la forma habitual, escribiendo directamente el nombre. La propiedad **Color** tiene un valor por default en **Black**, de modo que si no colocamos esta propiedad, la sombra será automáticamente negra.

La sombra puede tener diferentes direcciones, y este parámetro es controlado por la propiedad **Direction**, que necesita un valor numérico: los grados de la dirección de la sombra. Esta propiedad también tiene un valor predeterminado que corresponde a **315** grados.



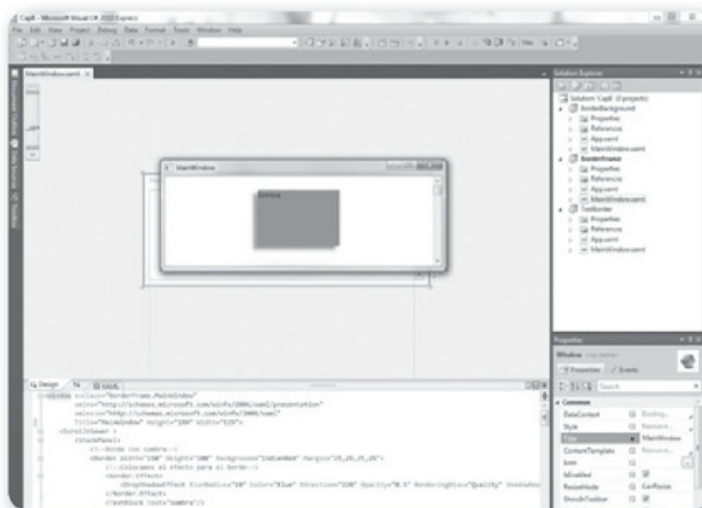
► **Figura 13.** Existe software comercial que permite simplificar el desarrollo de aplicaciones que usan XAML, como Aurora XAML Designer.

La propiedad **Opacity** controla la intensidad de la sombra. Si damos un valor pequeño, como **0.1**, la sombra será muy tenue, pero con valores más grandes, resultará más intensa. El valor predefinido es la sombra más intensa, es decir, de **1**.

Encontramos también otra propiedad, conocida como **RenderingBias**. Con ella, simplemente indicamos qué nos interesa más al momento de dibujar el efecto: la velocidad o la calidad. Si deseamos que la calidad sea más importante, colocamos el valor denominado **Quality**. Para indicar que nos interesa que el dibujo sea rápido y eficiente, debemos indicar el valor llamado **Performance**.

La última propiedad que tenemos es **ShadowDepth**, que sirve para indicar la distancia que hay entre el **Border** o el elemento usado, y la sombra. Cuanto más grande sea el valor, mayor será la distancia entre los dos. El valor predefinido es **5**, lo que da una sombra bastante cercana. Al usar correctamente esta propiedad en diferentes controles, podemos generar la sensación de que algunos controles se encuentran más arriba que otros. Con un poco de código que anime la sombra, daremos la sensación de que el control sube o también baja.

Con esto hemos creado un efecto de sombra básico. Lo único que nos falta es compilar la aplicación y ver cómo luce.



► **Figura 14.** Nuestro primer **Border** con sombra se muestra; hemos colocado valores en las propiedades más importantes.

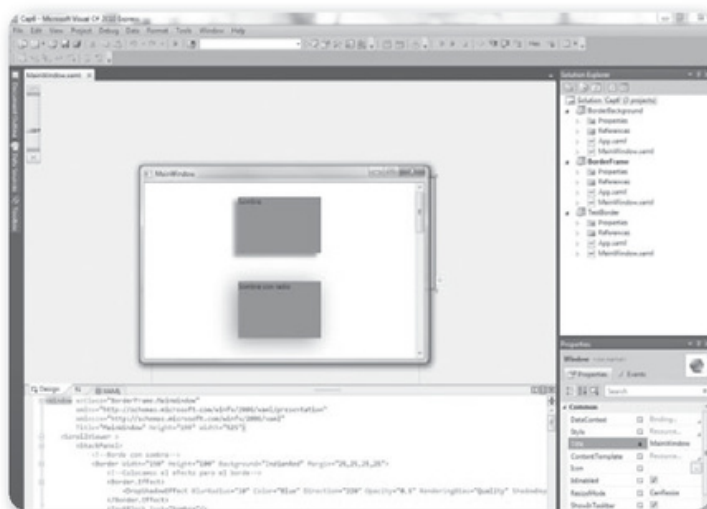
Debemos tener en cuenta que para trabajar con nuestro primer experimento, nos encargaremos de adicionar el siguiente código, el cual se ocupa de crear un nuevo **Border**:


```

<!--Borde con sombra cambia radio-->
<Border Width="150" Height="100" Background="IndianRed"
Margin="25,25,25,25">
    <!--Colocamos el efecto para el borde-->
    <Border.Effect>
        <DropShadowEffect BlurRadius="50" Color="Blue" Direction="220"
        Opacity="0.5" RenderingBias="Quality" ShadowDepth="10"/>
    </Border.Effect>
    <TextBlock Text="Sombra con radio"/>
</Border>

```

Estamos modificando el radio para el desvanecimiento, con un valor de **50**. Con esto, lograremos obtener como resultado una sombra más suave y dispersa, la cual se distribuye a mayor distancia del objeto. La **Figura 15** nos muestra este efecto.



► **Figura 15.** Al aumentar el valor del radio de desvanecimiento, se nota la diferencia con el primer tipo de sombra.



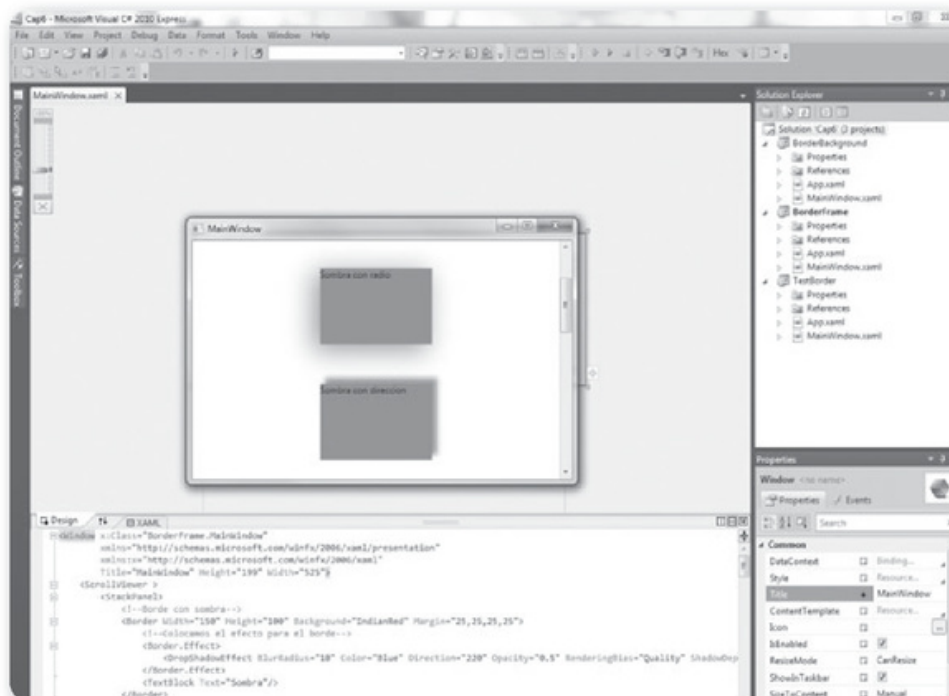
ANIMACIÓN DE LOS CONTROLES

Es necesario saber que todas las propiedades que hemos visto pueden manejarse por medio de código en el lenguaje C#. De esta forma es posible cambiar programáticamente los valores colocados y, así, crear animaciones o variaciones en la interfaz que muestren diferentes estados del software. Esto ayuda a llamar la atención del usuario hacia información importante.

Llegó el momento de experimentar con la dirección de la sombra. Cabe recordar que podemos modificar tantas propiedades como deseemos:

```
<!--Borde con sombra y direccion-->
<Border Width="150" Height="100" Background="IndianRed"
Margin="25,25,25,25">
  <!--Colocamos el efecto para el borde-->
  <Border.Effect>
    <DropShadowEffect BlurRadius="10" Color="Blue" Direction="45"
      Opacity="0.5" RenderingBias="Quality" ShadowDepth="10"/>
  </Border.Effect>
  <TextBlock Text="Sombra con direccion"/>
</Border>
```

Con este código, hemos colocado la sombra a **45** grados. Los grados se miden de la misma manera que en la geometría y crecen en el sentido contrario al de las manecillas del reloj.



► **Figura 16.** Aquí podemos ver que la sombra se encuentra a **45°**; esto es, en dirección hacia la derecha y arriba.

Para continuar, debemos proceder a escribir el siguiente código y agregarlo al programa en el cual estamos trabajando:

```
<!--Borde con sombra y opacidad-->
<Border Width="150" Height="100" Background="IndianRed"
Margin="25,25,25,25">
  <!--Colocamos el efecto para el borde-->
  <Border.Effect>
    <DropShadowEffect BlurRadius="10" Color="Blue" Direction="220"
      Opacity="0.9" RenderingBias="Quality" ShadowDepth="10"/>
  </Border.Effect>
  <TextBlock Text="Sombra"/>
</Border>
```

Esto nos permite experimentar con la propiedad de opacidad. El valor es **0.9**, que nos da una opacidad fuerte. La sombra debe notarse más. Este tipo de valores es útil cuando queremos tener sombras bien definidas; valores pequeños nos dan sombras tenues.

Finalicemos nuestros experimentos con las propiedades adicionando el siguiente código al programa:

```
<!--Borde con sombra y shadow depth-->
<Border Width="150" Height="100" Background="IndianRed"
Margin="25,25,25,25">
  <!--Colocamos el efecto para el borde-->
  <Border.Effect>
    <DropShadowEffect BlurRadius="10" Color="Blue" Direction="220"
      Opacity="0.5" RenderingBias="Quality" ShadowDepth="50"/>
  </Border.Effect>
  <TextBlock Text="Sombra"/>
</Border>
```

En este caso, hemos modificado el valor de **ShadowDepth** por uno de **50**, que aleja la sombra del **Border**. La modificación en la distancia de la sombra puede dar la sensación de altura en el **Border**.

El efecto de altura puede servirnos para llamar la atención del usuario a partes importantes de la interfaz. Es otro mecanismo que podemos aprovechar para comunicarnos con el usuario final.

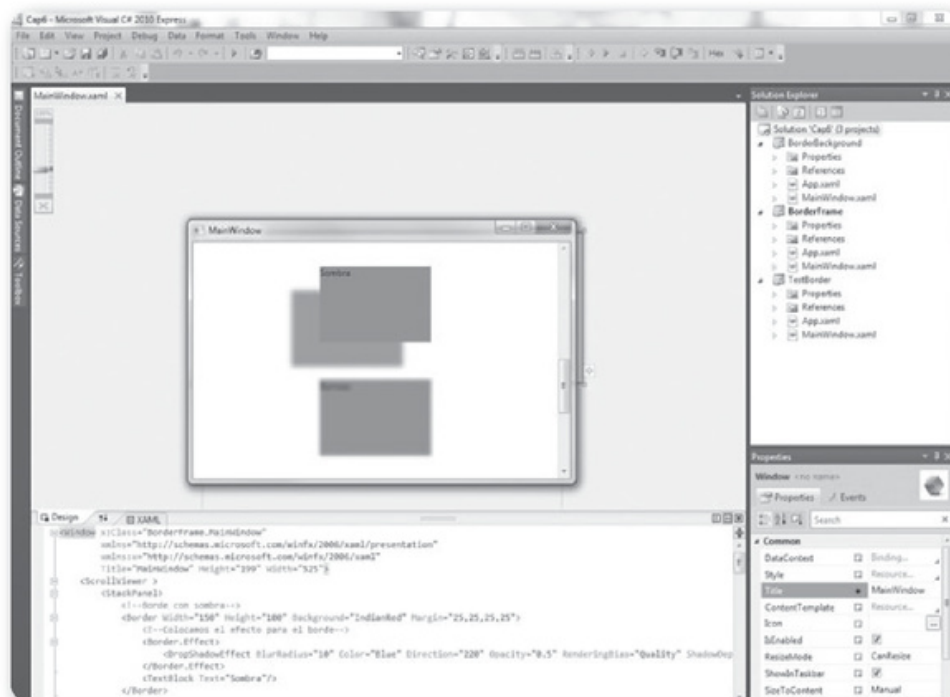


► **Figura 17.** En esta figura se muestra la opacidad de la sombra y su profundidad.

El siguiente efecto que usaremos se conoce como **BlurEffect**, y hace que el **Border** o el elemento que lo emplee se vea borroso o difuminado. Empecemos por agregar el siguiente código en nuestra aplicación:

```
<!--Borde borroso-->
<Border Width="150" Height="100" Background="IndianRed"
Margin="25,25,25,25">
  <!--Colocamos el efecto para el borde-->
  <Border.Effect>
    <BlurEffect Radius="7" KernelType="Gaussian"
      RenderingBias="Quality"/>
  </Border.Effect>
  <TextBlock Text="Borroso"/>
</Border>
```

En este caso, usamos pocas propiedades, aunque existen más. La primera es el radio que se emplea para crear el efecto borroso. El valor colocado es de **7**, pero el predefinido es de **5**. Valores más altos contribuyen a que el elemento se vea más borroso. La propiedad **KernelType** se usa para seleccionar el tipo de curva empleada en el cálculo del efecto. Existen dos posibles valores para esta propiedad: **Gaussian** y **Box**. La propiedad **RenderingBias** actúa exactamente igual que en el efecto anterior. A continuación veamos cómo luce este tipo de efecto, simplemente, compilando y ejecutando la aplicación.



► **Figura 18.** El efecto **BlurEffect**, aplicado en este caso, hace que el **Border** y sus contenidos se vean borrosos.

Ahora podemos proceder a realizar la adición de otro borde y también experimentar un poco con sus propiedades:

```
<!--Borde borroso y radio-->
<Border Width="150" Height="100" Background="IndianRed"
Margin="25,25,25,25">
  <!--Colocamos el efecto para el borde-->
  <Border.Effect>
    <BlurEffect Radius="20" KernelType="Gaussian"
      RenderingBias="Quality"/>
  </Border.Effect>
  <TextBlock Text="Borroso y radio"/>
</Border>
```

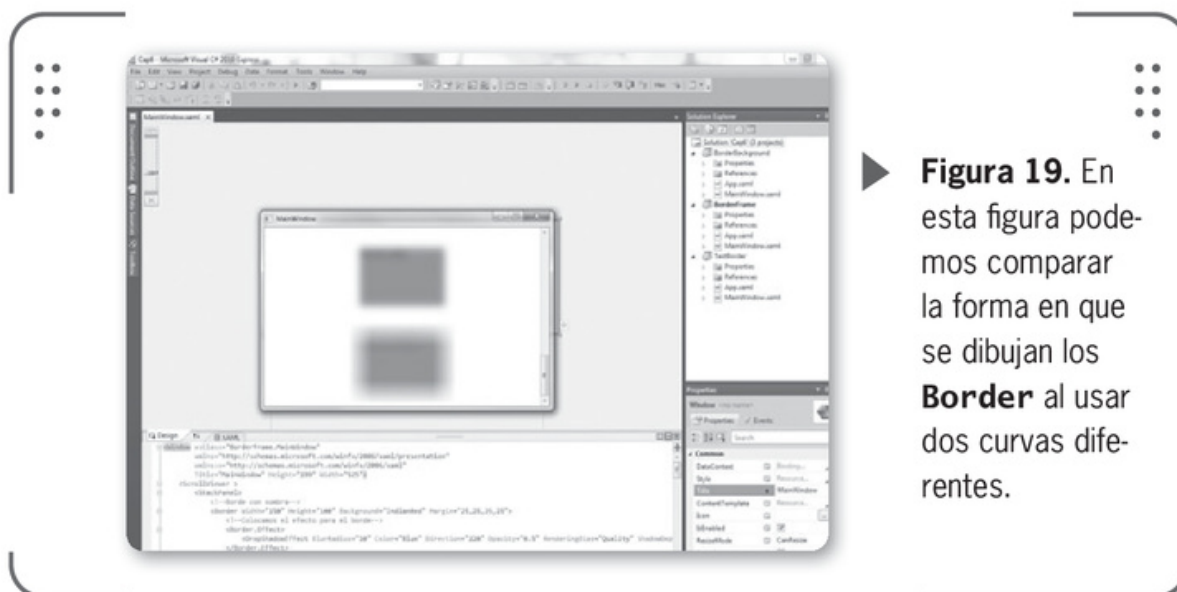
En este ejemplo hemos modificado el radio, lo cual producirá un efecto borroso más acentuado en el **Border**. La curva usada para el cálculo del efecto modifica la forma en que este se despliega.


```

<!--Borde borroso y suavizado-->
<Border Width="150" Height="100" Background="IndianRed"
Margin="25,25,25,25">
  <!--Colocamos el efecto para el borde-->
  <Border.Effect>
    <BlurEffect Radius="20" KernelType="Box"
    RenderingBias="Quality"/>
  </Border.Effect>
  <TextBlock Text="Borroso y suavizado"/>
</Border>

```

En este código mantenemos un radio de **20**, pero cambiamos el valor de **KernelType** a **Box**. Los resultados son muy diferentes.



EL USO DE LOS RADIOS

Por medio del uso de las propiedades denominadas **RadiusX** y también **RadiusY**, es posible acceder a realizar el control de la figura que se dibuja en el degradado radial. Tengamos en cuenta que si los dos valores son iguales en un **Border** cuadrado, como resultado obtendremos un círculo, mientras que si son diferentes, tendremos algún tipo de elipse. Tengamos presente que la proporción del ancho y alto del **Border** afecta también la forma como se dibuja el degradado radial. De esta manera, podremos aprovechar estas características para aplicarlas en nuestros proyectos con distintos resultados.



Creación de una aplicación que controle a la interfaz

Con lo que hemos aprendido hasta ahora, llegó el momento de realizar una aplicación sencilla que modifique los diferentes elementos de la interfaz de usuario. La aplicación tendrá un botón con un degradado de colores en su interior. Tendremos un **TextBox** dentro de un **Border** con sombra. Podemos colocar cualquier texto, que se imprimirá en un **TextBlock**. Cada vez que el usuario oprima el botón, la sombra se modificará, y el **TextBlock** cambiará de posición y orientación.

El código que corresponde a esta interfaz de usuario es el siguiente:

```
<Window x:Class="TestBorder.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="MainWindow" Height="350" Width="525">
    <StackPanel>
        <!--Colocamos un boton-->
        <Button x:Name="Boton" Content="Mi Boton" Click="MiBotonClick"
            MouseEnter="MiBotonEnter" MouseLeave="MiBotonLeave">
            <Button.Background>
                <LinearGradientBrush EndPoint="1,0.5" StartPoint="0,0.5">
                    <GradientStop Color="#FF922799" Offset="0" />
                    <GradientStop Color="#FF7B8AE4" Offset="0.869" />
                    <GradientStop Color="#FFFF9D9D" Offset="0.443" />
                </LinearGradientBrush>
            </Button.Background>
        </Button>
    </StackPanel>
</Window>
```



CONFIGURAR LA VENTANA



La ventana que se encarga de contener a la forma con la que trabajamos posee muchas propiedades con las que podemos experimentar en nuestros desarrollos. Estas se agrupan en siete categorías: **common**, **layout**, **brushes**, **text**, **visibility**, **transform** y **other**. Es conveniente conocer cada una de estas propiedades para tener más control sobre la forma correspondiente.

```

</Button>
<!--Borde con sombra y direccion-->
<Border Width="150" Height="100" Background="IndianRed"
Margin="25,25,25,25">
  <!--Colocamos el efecto para el borde-->
  <Border.Effect>
    <DropShadowEffect x:Name="efecto" BlurRadius="10"
    Color="Blue" Direction="220" Opacity="0.5" RenderingBias="Quality"
    ShadowDepth="10"/>
  </Border.Effect>
  <TextBox x:Name="txtTexto" Text="Hola" FontSize="32" />
</Border>

<!--Colocamos un textblock-->
<TextBlock x:Name="txbMensaje" FontSize="42" FontWeight="Bold"
FontFamily="Comic Sans MS">

  <!--Indicamos que vamos a transformarlo-->
  <TextBlock.RenderTransform>
    <!--Hay mas de una transformacion por lo que necesitamos un grupo-->
    <TransformGroup>
      <RotateTransform x:Name="Rotacion" Angle="20"/>
      <TranslateTransform x:Name="Traslacion" X="20"/>
    </TransformGroup>
  </TextBlock.RenderTransform>
</TextBlock>
</StackPanel>
</Window>

```



BIBLIOTECA DE CONTROLES



Es una buena idea que conforme programamos nuestros controles creamos también una biblioteca para los mismos, de esta forma, podremos aprovecharlos nuevamente en varios proyectos y con el tiempo tendremos un catalogo interesante de controles a utilizar. También podremos intercambiar o vender esta biblioteca con otros programadores.

NOMBRAR
CORRECTAMENTE
LOS CONTROLES NOS
SIMPLIFICA USAR EL
CÓDIGO ADECUADO

Podremos darnos cuenta de que el primer elemento con el que nos encontramos es un botón, con el que usaremos tres eventos: **Click**, **MouseEnter** y **MouseLeave**. Cada uno de ellos tiene referenciado su handler. El nombre asignado es, simplemente, **Boton**. En su interior

colocamos un degradado lineal de tres colores.

Posteriormente, ponemos un **Border** al cual le damos un efecto de sombra. El único punto que debemos tomar en cuenta es que al efecto le hemos asignado un nombre por medio de la propiedad **x:Name**. Este nombre nos servirá para manipular las propiedades del efecto desde el código en C#. En el mismo **Border** también colocamos un **TextBox**. Tengamos en cuenta que para indicar el tamaño del texto, usamos la propiedad **FontSize** con un valor de **32**.

Por último, agregamos un **TextBlock**, en cuyo interior hacemos uso de **RenderTransform**, que permite modificar la posición y el ángulo en el cual deseamos que se coloque el texto. Como vamos a usar más de una transformación, es necesario crear un **TransformGroup** y, en su interior, poner todas las transformaciones que hagan falta.

Para indicar la rotación usamos **RotateTransform**. El ángulo de rotación está dado por medio de la propiedad **Angle**. Colocamos un nombre, pues a través del código de C#, modificaremos el valor de rotación. La posición del texto cambiará gracias a **TranslateTransform**, que nos permite trasladar el texto. En este momento, solamente nos interesa trabajar con la coordenada **X**. También le hemos colocado un nombre para poder modificar la posición desde el código en C#.



COLECCIÓN DE CONTROLES



Es importante tener en cuenta que en Internet es posible encontrar muchos controles que podemos utilizar en nuestras aplicaciones y modificarlos a nuestras necesidades, algunos de ellos se ofrecen en forma completamente gratuita mientras que otros se distribuyen en forma comercial, aunque siempre podremos acceder a las versiones de demostración o evaluación de estos últimos. Estos controles adicionados a los que programamos nosotros mismos nos permite a futuro poder crear las interfaz de las aplicaciones de manera más rápida y eficiente.

A continuación, debemos ingresar el código de C# que controla los elementos de la interfaz. Si lo deseamos, hacemos doble clic en el botón y colocamos el código para el evento **Click**, seguido del correspondiente para los demás eventos:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;
namespace TestBorder
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }
        private void MiBotonClick(object sender, RoutedEventArgs e)
        {
            // Incrementamos el radio del blur
            efecto.BlurRadius += 5;
            // Modificamos los valores de transformacion
            Rotacion.Angle += 5;
            Traslacion.X += 10;
            // Colocamos el texto
```

```

        txbMensaje.Text = txtTexto.Text;
    }
    private void MiBotonEnter(object sender, MouseEventArgs e)
    {
        // Colocamos la nueva direccion
        efecto.Direction = 45;
    }
    private void MiBotonLeave(object sender, MouseEventArgs e)
    {
        // Colocamos la nueva direccion
        efecto.Direction = 220;
    }
}
}

```

En el evento **Click** vamos a modificar varias cosas. Empezamos por incrementar el radio en el efecto, simplemente, adicionándole **5** al valor actual. Cabe notar que estamos usando el nombre dado al efecto y, de la forma tradicional, se asigna el valor a la propiedad. Después de esto, procedemos a modificar las transformaciones. La

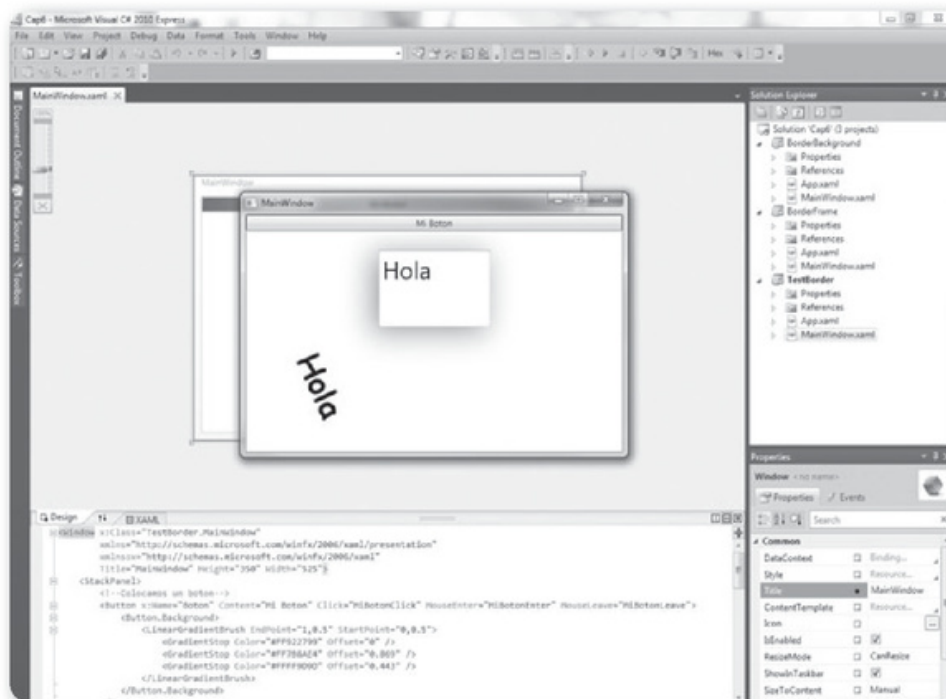
**LAS PROPIEDADES
PUEDEN MODIFICARSE
DINÁMICAMENTE,
ESTO ENTREGA MÁS
INTERACTIVIDAD**



primera es **Rotación**, e incrementamos su ángulo en **5**. Para **Traslación** aumentamos el valor de la propiedad **X** en **10**. Luego, simplemente tomamos el texto escrito en el **TextBox** y lo mostramos en el **TextBlock**. Con esto, vemos que podemos modificar cualquier propiedad de cualquier elemento de la interfaz, siempre y cuando tengamos un nombre con el cual referenciarlo.

Debemos tener en cuenta que cuando el puntero del mouse entra al botón, se invoca el handler denominado **MiBotonEnter**, dentro

del cual, simplemente, nos encargamos de realizar el cambio en la dirección de la sombra correspondiente, colocando una de **45** grados. Para cuando el puntero abandona el botón, se procederá a invocar al handler **MiBotonLeave**. En este handler colocamos otra vez la sombra en su dirección original, con un valor que corresponde a **220**. Ahora podemos ejecutar la aplicación y probar cómo los diferentes elementos son controlados por el código.



► **Figura 20.** Los diferentes elementos de la interfaz de usuario son controlados por el código en C#.

Con esto, ya tenemos los conocimientos básicos sobre el control de interfaces en XAML para la creación de aplicaciones WPF. Aún quedan muchos temas que se pueden aprender sobre el uso de WPF, para aplicaciones tanto 2D como también 3D.



RESUMEN

XAML nos permite tener control sobre diferentes propiedades que se encargan de regular el despliegue de los elementos en la pantalla. Al combinar XAML con C#, podemos controlar programáticamente y de forma avanzada la interfaz del usuario. El fondo del **Border** y otros elementos pueden dibujarse de diferentes maneras, con brochas sólidas, de gradiente y radiales. Con los efectos para el marco, podemos tener sombras o hacer borroso el elemento.

Actividades

TEST DE AUTOEVALUACIÓN

- 1 Haga un programa que cambie con el movimiento del mouse el tamaño de un **Border**.
- 2 Desarrolle un programa que cambie los colores de un degradado lineal al oprimir un botón.
- 3 Haga un programa que cambie el sentido de un degradado lineal al oprimir un botón.
- 4 Coloque los colores del degradado lineal con valores hexadecimales.
- 5 Realice un programa que modifique los valores de **Offset** de los colores de degradado automáticamente, para crear una animación.
- 6 Haga un programa que modifique la dirección del degradado lineal dependiendo de la posición del puntero del mouse.
- 7 Experimente con la propiedad **ColorInterpolationMode**.
- 8 Experimente con la propiedad **MappingMode** y coloque las coordenadas en píxeles.
- 9 Haga un programa que tenga un **Border** con degradado radial. Es posible modificar el origen del degradado dependiendo del lugar donde se haga clic.
- 10 Realice un programa que efectúe una animación cambiando los radios de un degradado radial.
- 11 Investigue cómo hacer una colección de **GradientStop**.
- 12 Experimente con diferentes valores de **Offset** en el degradado radial.

EJERCICIOS PRÁCTICOS

- 1 Cree una calculadora completa con toda la interfaz en WPF.
- 2 Haga un programa para seleccionar colores con WPF.
- 3 Experimente con las herramientas de Visual Studio para los controles en WPF.
- 4 Haga un pequeño editor de texto en el que pueda cambiar los colores de la fuente.
- 5 Coloque las brochas y los efectos que hemos visto a diferentes controles.

*07

=

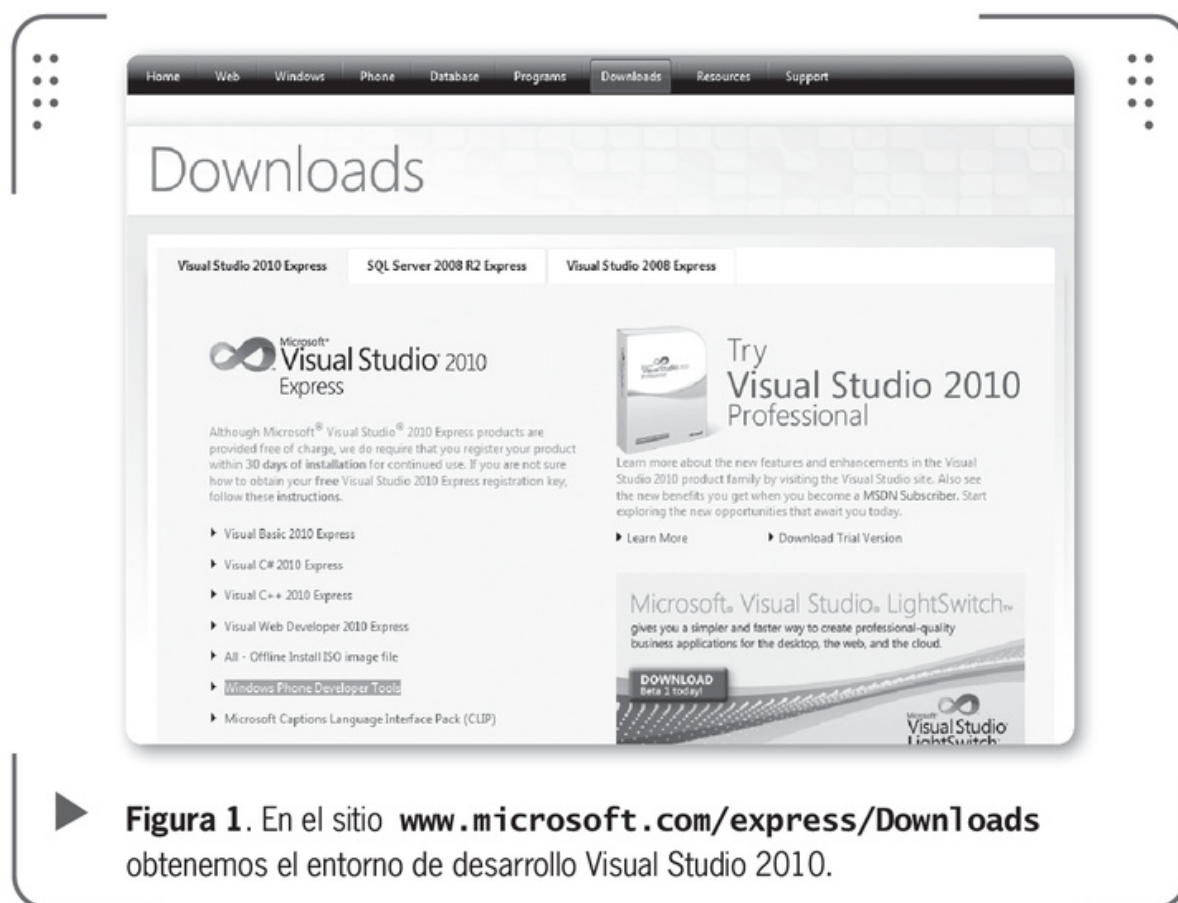


Programar para Windows Phone 7

Hace poco tiempo, Microsoft tomó la decisión de incursionar en el mercado de los teléfonos celulares con un esquema nuevo llamado Windows Phone 7. En el pasado ya se habían usado sistemas operativos de Microsoft en este tipo de dispositivos, pero con Windows Phone 7, se espera llegar más

Entender Windows Phone 7

La programación de aplicaciones para Windows Phone 7 no es complicada: podemos usar C# para hacerlo y apoyarnos en nuestros conocimientos previos. Para empezar a trabajar con Windows Phone, necesitamos tener Visual Studio 2010 Express para Windows Phone. Esta herramienta es una versión de Visual Studio, pero con los elementos y las funciones de edición necesarios para poder crear aplicaciones destinadas a esta plataforma. También obtenemos un emulador de Windows Phone en el cual podemos probar las aplicaciones que desarrollemos. El conocimiento sobre WPF nos ayudará en el diseño de las interfaces, mientras que con C# crearemos la lógica de la aplicación.



► **Figura 1.** En el sitio www.microsoft.com/express/Downloads obtenemos el entorno de desarrollo Visual Studio 2010.

Cuando desarrollamos para Windows Phone 7, tenemos dos plataformas en las cuales podemos trabajar: una es **Silverlight** y la otra, **XNA**. Silverlight tiene sus bases en **WPF** y se ha utilizado, principalmente, en la Web para crear interfaces de usuario avanzadas y

muy dinámicas. Podemos considerar, de alguna forma, que Silverlight es competidora de Flash. Cuando trabajamos con ella, usamos nuestro conocimiento de XAML para definir las interfaces. Por su parte, XNA fue diseñada para aplicaciones gráficas de alto desempeño, como videojuegos. Con ella se pueden crear juegos en 2D y 3D. Inicialmente, se utilizaba para aplicaciones de PC o de Xbox 360, pero ahora también podemos usarla en Windows Phone.

Creación de un proyecto para Windows Phone

Una vez instalada la herramienta, la creación de una aplicación para Windows Phone es muy sencilla. Simplemente, debemos hacer clic en el icono de la herramienta que se encuentra adentro de la carpeta de Microsoft Visual Studio 2010 Express, en el botón de inicio de nuestra computadora. Podemos seleccionar el menú **Archivo** y, luego, **Nuevo Proyecto**. Aparecerá un nuevo cuadro de diálogo, que muestra los posibles proyectos que podemos crear para Windows Phone; en caso de tener instalado XNA, también se muestran sus proyectos. En general, seleccionamos la opción **Aplicación de Windows Phone**.

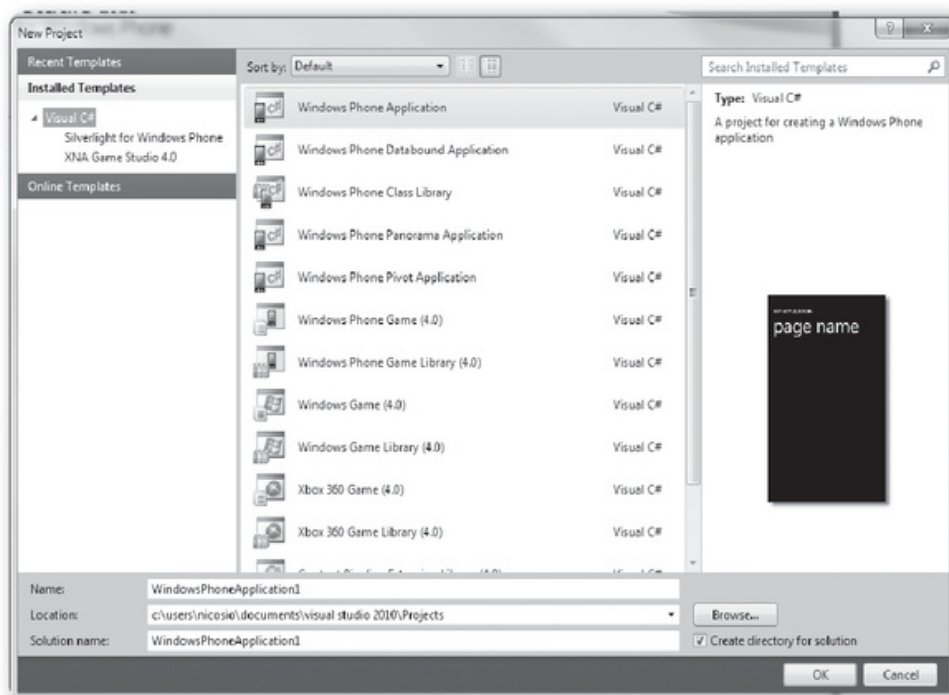
En este cuadro de diálogo, aparecen de manera predeterminada todos los templates de proyectos, pero si lo deseamos, podemos observar solo aquellos que estén basados en Silverlight o XNA. Es necesario tener en cuenta que en la parte izquierda del cuadro están las opciones de listado e, incluso, si lo deseamos, podemos optar por templates que se encuentren en Internet.



VISUAL STUDIO 2010 EXPRESS



Es importante saber que la versión que necesitamos para nuestras aplicaciones de Windows Phone se distribuye en forma completamente gratuita. Tenemos que instalarla para desarrollar los programas, y contar con las herramientas y los namespaces necesarios. Es posible obtenerla desde el siguiente sitio: www.microsoft.com/express/Downloads.



► **Figura 2.** Podemos ver que este cuadro de diálogo nos permite seleccionar el tipo de proyecto que deseamos crear.

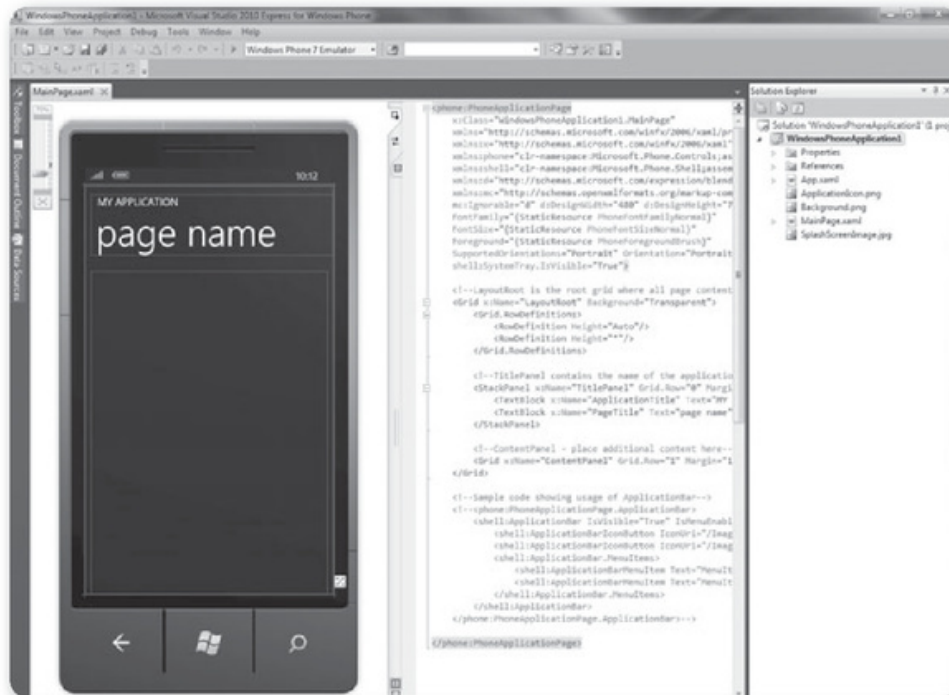
Una vez colocado el nombre de la solución, pulsamos el botón de **OK** y Visual Studio creará por nosotros el esqueleto de la aplicación. A partir de este punto, podemos empezar a codificar.

Visual Studio siempre crea este tipo de esqueleto de aplicación para nosotros, pero también es posible que nosotros mismos diseñemos un proyecto y colocarlo como template para utilizarlo posteriormente.



SILVERLIGHT

Silverlight no solamente es utilizado en Windows Phone: Microsoft también está apostando fuertemente a esta plataforma porque en la actualidad es el framework utilizado para las interfaces de las aplicaciones para la nube y Azure. Es una buena idea aprender Silverlight si nos interesa mantenernos actualizados en las tecnologías desarrolladas por Microsoft. Sin embargo se rumora al tiempo de escribir este libro que HTML5 tendrá una fuerte influencia en las interfaces de usuario para la próxima versión de Windows. Es otra tecnología que hay que aprender.



► **Figura 3.** Visual Studio ha creado el esqueleto de la aplicación por nosotros, y nos muestra los editores y las herramientas necesarios.

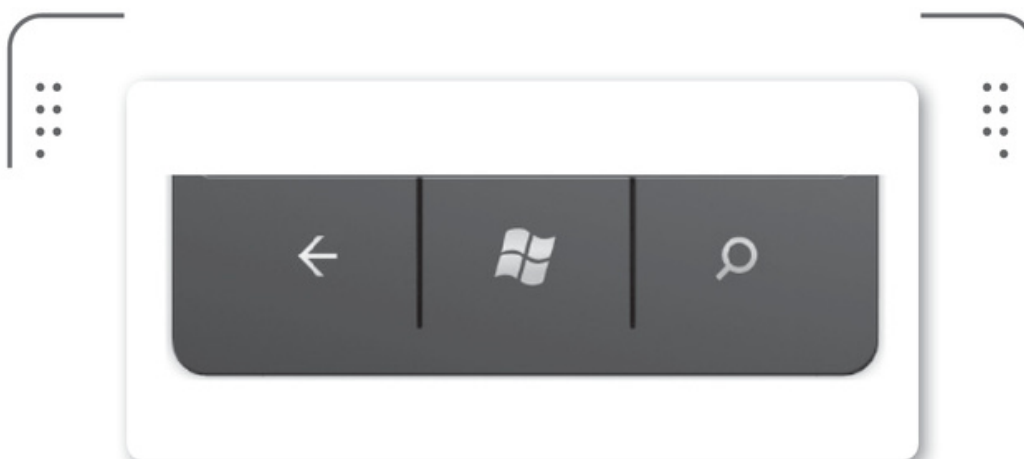
Entender el hardware

Microsoft ha definido las características del hardware que deben cumplir aquellas empresas interesadas en construir teléfonos que usen Windows Phone 7. En este sentido, la firma ha sido estricta, y todos los aparatos deben tener, al menos, algunas de estas características y no modificar otras.

Una de las características que no pueden cambiar es el tamaño de la pantalla, aunque se plantea que, en el futuro, es posible que existan dos tamaños estándar con los cuales trabajar. La pantalla debe ser **multi-touch**; esto quiere decir, capaz de sensar correctamente cuando la tocamos en más de un lugar. Debe tener una resolución de 480 x 800 píxeles, que se conoce, de forma coloquial, como **large** o grande. La otra posible pantalla tendrá una resolución de 320 x 480, y se la conoce como **small** o pequeña. Es poco

se desarrolla la mayoría de las aplicaciones para Windows Phone. Por el contrario, cuando el lado mayor se coloca en sentido horizontal, se conoce como **Landscape** o paisaje. Es posible diseñar una aplicación para que se ajuste a las diferentes orientaciones de la pantalla.

Si observamos los teléfonos que usan Windows Phone 7 podemos notar que presentan tres botones: estos se conocen como **Back** o anterior, **Start** o inicio y **Search** o búsqueda.



► **Figura 4.** En esta figura podemos observar los tres botones que debe tener todo teléfono que use Windows Phone 7.

El botón de **Back** se usa, principalmente, para navegar en la aplicación, y tiene como icono una pequeña flecha que apunta hacia la izquierda. Nuestras aplicaciones pueden tener múltiples páginas, tal como sucede en un sitio web, y si necesitamos regresar a una anterior, pulsamos este botón, que recuerda al botón de **Anterior** que existe en los navegadores web. Una característica de este botón es que finaliza la aplicación si se lo oprime desde la página principal del programa.



EL CHASSIS



De forma coloquial se llama **chassis** a las características de hardware impuestas por Microsoft a los fabricantes de teléfonos que usen el sistema Windows Phone 7. Esto tiene ciertas ventajas, ya que es más fácil desarrollar para una plataforma bien establecida. Los fabricantes deben crear sus teléfonos con estas características de chassis forzosamente.



► **Figura 5.** En esta imagen se muestra el teléfono con la orientación de pantalla conocida como **Portrait**.

Los otros dos botones tienen comportamientos más sencillos. **Start** lleva directamente a la pantalla de inicio. En el caso de **Search**, tiene como icono una lupa y se utiliza para realizar búsquedas.



► **Figura 6.** Cuando se rota el teléfono de esta forma, la orientación de la pantalla es conocida como **Landscape**.

WINDOWS PHONE TIENE HARDWARE QUE DEBEMOS CONOCER PARA NUESTRAS APLICACIONES



Además de la pantalla, los teléfonos que usan Windows Phone 7 deben tener cierto hardware instalado que ofrezca funcionalidad, al que se conoce, generalmente, como **sensores**, pero también se incluye otro tipo de dispositivos. Dentro de este hardware encontramos un **acelerómetro**, que permite sensar cambios en la velocidad. Funciona en 3D, por lo que

podemos obtener fácilmente la dirección a la que se mueve el teléfono y cuán rápido lo hace. El aparato también tiene **Wi-Fi** para establecer la conexión a Internet, aunque no es la única forma de hacerlo: es posible realizarlo vía **3G** usando el servicio de algún proveedor de telefonía celular.

El teléfono también contiene el hardware necesario para obtener la locación geográfica del aparato. En general, esto se efectúa por medio del uso de **GPS** y se complementa con información obtenida de la Web. La **cámara** que

debe tener el teléfono tiene que ser de, al menos, 5 megapíxeles, y es muy posible que en el mercado obtengamos equipos con resoluciones mayores. Debe incluir **flash**. Algo común en todos los teléfonos actuales es el uso de un **vibrador**. Una **radio FM** también forma parte de las especificaciones de hardware.

Colocar la orientación de la aplicación

Hemos visto que es posible colocar el teléfono en diferentes orientaciones, y es preciso que nuestra interfaz de usuario se adapte a ellas. Afortunadamente, con Silverlight es muy sencillo lograrlo.



ORIENTACIÓN EN VISUAL STUDIO



Tengamos en cuenta que es posible cambiar la orientación del teléfono en el editor de la interfaz de Microsoft Visual Studio. Para lograrlo, simplemente debemos proceder a hacer clic derecho sobre el teléfono y seleccionar el menú de **Switch** o **Cambiar a**. De esta manera, podemos diseñar la interfaz de nuestra aplicación para alguna orientación en particular.

Por predefinición, nuestra aplicación tendrá la orientación **Portrait**, y no se ajustará automáticamente; sin embargo, solo debemos hacer un cambio para conseguirlo. Si observamos el código que crea Visual Studio por nosotros, encontramos lo siguiente:

```
<phone:PhoneApplicationPage
  x:Class="Convertidor.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:phone="clr-namespace:Microsoft.Phone.Controls;assembly=Microsoft.
  Phone"
  xmlns:shell="clr-namespace:Microsoft.Phone.Shell;assembly=Microsoft.
  Phone"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  mc:Ignorable="d" d:DesignWidth="480" d:DesignHeight="768"
  FontFamily="{StaticResource PhoneFontFamilyNormal}"
  FontSize="{StaticResource PhoneFontSizeNormal}"
  Foreground="{StaticResource PhoneForegroundBrush}"
  SupportedOrientations="Portrait" Orientation="Portrait"
  shell:SystemTray.IsVisible="True">
```

Podemos observar que la propiedad **SupportedOrientations** tiene el valor de **Portrait**. Entonces, debemos cambiarla para que sea de **PortraitOrLandscape**. Esta sencilla modificación nos permite que la interfaz se ajuste a la orientación.

En ocasiones, debido a la forma en que se ha creado la interfaz, necesitamos modificarla de manera programática. Para hacerlo, lo mejor es aprovechar un handler cuando ocurra un evento de cambio



MULTI-TOUCH Y EL EMULADOR



Tengamos en cuenta que el emulador de Windows Phone que viene con Microsoft Visual Studio no puede representar el multi-touch plenamente; el mouse puede representar solo un dedo. Por esta razón, si necesitamos probar aplicaciones multi-touch en el emulador, será preciso tener un teléfono real o también podemos utilizar un monitor que posea esta función integrada.

de orientación. **PhoneApplicationPage** y **PhoneApplicationFrame** tienen eventos relacionados con la orientación. **OnOrientationChanged** es un método al cual le podemos hacer el override si necesitamos ejecutar este tipo de código. Para saber la orientación, debemos usar el objeto **args** y leer su propiedad **Orientation**:

```
protected override void OnOrientationChanged(OrientationChangedEventArgs
args)
{
    // El código aquí
}
```

El sistema multi-touch

Muchas de las entradas y la interacción con la interfaz de usuario dentro de Windows Phone 7 se llevan a cabo por medio de la pantalla. Esta es sensible al tacto, y nuestro programa tiene que poder responder a los toques que da el usuario. En un principio, parece que esto es sencillo y que será algo similar a trabajar con el puntero del mouse, pero se trata de una tarea un poco más complicada.

Tengamos en cuenta que la pantalla es un sistema multi-touch, lo que significa que puede ser capaz de detectar múltiples toques en forma simultánea. Microsoft especifica que la cantidad mínima de toques simultáneos que debe reconocer el teléfono es cuatro.

La manera de adquirir la información de multi-touch en Silverlight y XNA es diferente. En Silverlight, nos apoyamos en el uso de handlers para los eventos correspondientes; pero en XNA, debemos adquirir la información en el método **Update()**, apoyándonos en una clase estática que contiene los datos que necesitamos.



PROGRAMACIÓN DE WINDOWS PHONE 7

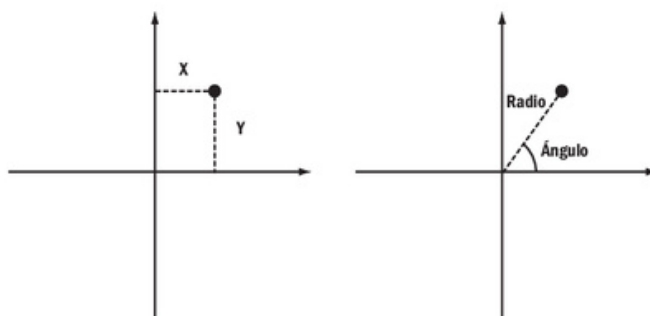


Es importante saber que el siguiente sitio web presenta una gran cantidad de recursos y documentación para ayudarnos a crear aplicaciones para Windows Phone 7. Es recomendable visitarlo, pues puede tener información y consejos que necesitemos en el futuro; lo encontramos en la dirección [http://msdn.microsoft.com/en-us/library/ff402535\(VS.92\).aspx](http://msdn.microsoft.com/en-us/library/ff402535(VS.92).aspx).

En Silverlight, existen dos modos de trabajar con multi-touch: de alto nivel y de bajo nivel. En el segundo caso, usamos un evento estático conocido como **Touch.FrameReported** y hacemos uso de la clase **TouchPoint**. En el caso del alto nivel, recurrimos a tres eventos, que son: **ManipulationStarted**, **ManipulationDelta** y **ManipulationCompleted**. A estos generalmente se los conoce como eventos de manipulación, y nos permiten trabajar con múltiples toques de dedos en la pantalla para obtener información de movimiento y escala.

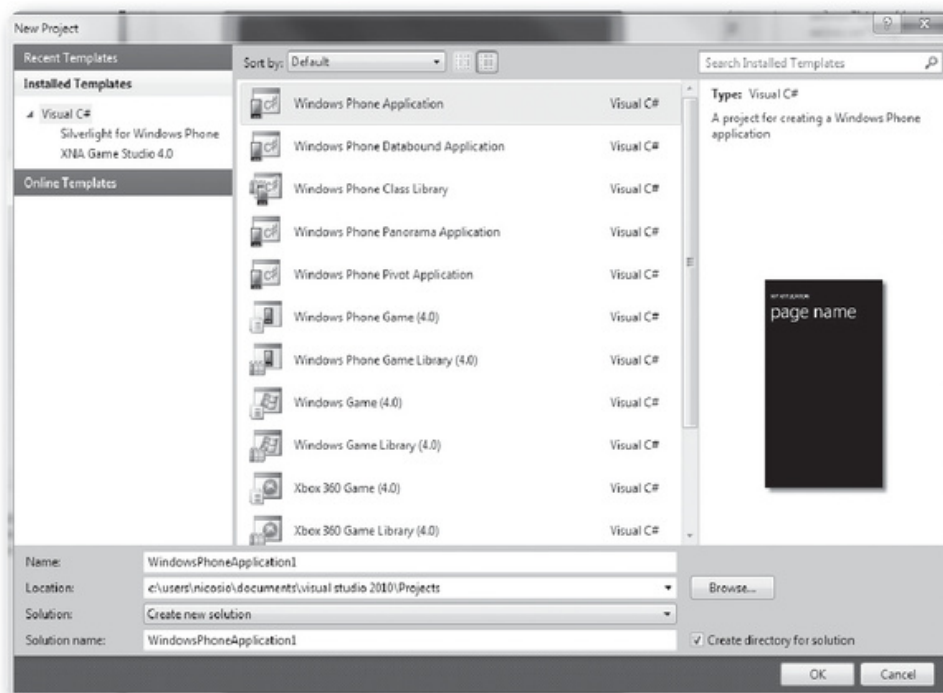
Crear una aplicación de prueba

Llegó el momento de empezar a desarrollar una pequeña aplicación de prueba, con la cual pondremos en práctica nuestro conocimiento sobre Windows Phone 7, XAML y C#. La aplicación es simple pero contiene todos los elementos requeridos. Consiste en un pequeño programa que lleve a cabo la conversión de coordenadas polares a coordenadas cartesianas, y viceversa. El usuario podrá escribir los valores necesarios y, al oprimir un botón, se efectuará la conversión. No solo mostraremos el resultado en la pantalla: también presentaremos una gráfica que indique el punto en las coordenadas correspondientes. Los datos que necesitamos para indicar la posición de un punto en coordenadas cartesianas son las coordenadas **X** e **Y** de dicho punto. En el caso de las coordenadas polares, precisamos el radio y el ángulo que definen a dicho punto.



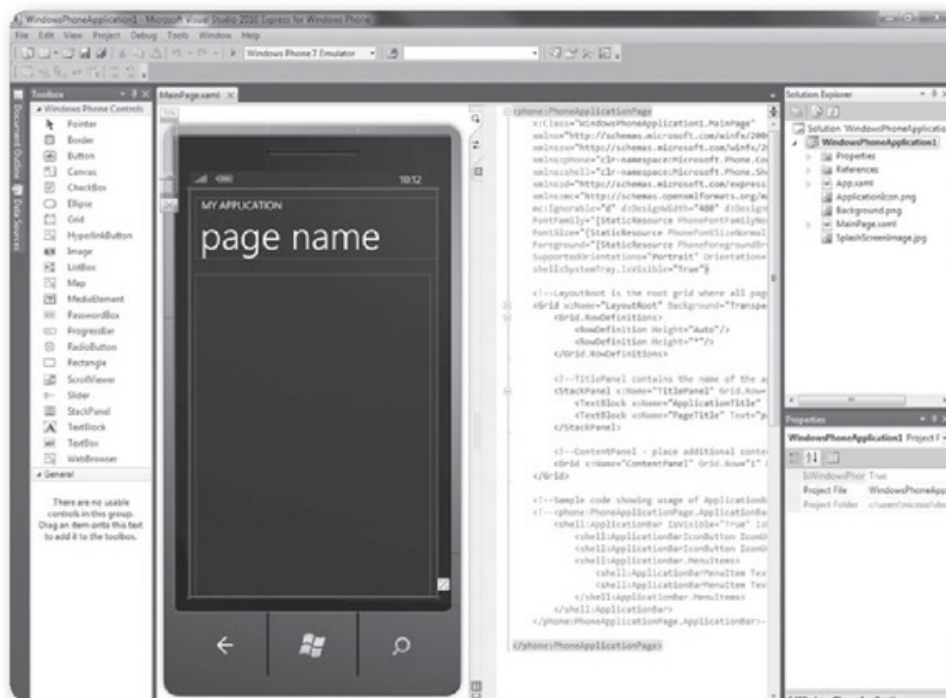
► **Figura 7.** Esta figura nos muestra la representación del mismo punto en coordenadas cartesianas y polares.

Empecemos por crear una aplicación nueva con Visual Studio 2010 para Windows Phone. En nuestra pantalla aparece una ventana con muchas opciones, ya que hay varios tipos de proyectos que podemos desarrollar para Windows Phone 7. Seleccionamos la opción denominada **Aplicación para Windows Phone**.



► **Figura 8.** Aquí es necesario que procedamos a seleccionar la opción llamada **Aplicación para Windows Phone**.

Debemos tener en cuenta que Microsoft Visual Studio se encarga de crear el esqueleto de la aplicación para nosotros y de la misma forma aparece el editor, dividido en diferentes secciones. En el extremo izquierdo está la caja de herramientas, donde encontramos los controles más usuales que podemos adicionar a la interfaz gráfica de la aplicación. Luego, tenemos una representación de Windows Phone en la cual podemos observar la forma en que va a lucir la interfaz gráfica. Enseguida vemos el editor de código, en donde colocamos el código XAML que define a la interfaz gráfica. En el extremo derecho, si así lo tenemos activado, aparecen el explorador de soluciones y la ventana de propiedades.



► **Figura 9.** En esta imagen podemos ver Visual Studio para Windows Phone, con el proyecto que vamos a desarrollar.

Crear la interfaz de usuario

Visual Studio ha creado algo de código para nosotros y tenemos el esqueleto básico sobre el cual podemos construir nuestra interfaz de usuario. La forma de construir la interfaz puede ser por medio de código XAML, o utilizando los controles de la caja de herramientas, arrastrándolos y colocando sus propiedades.

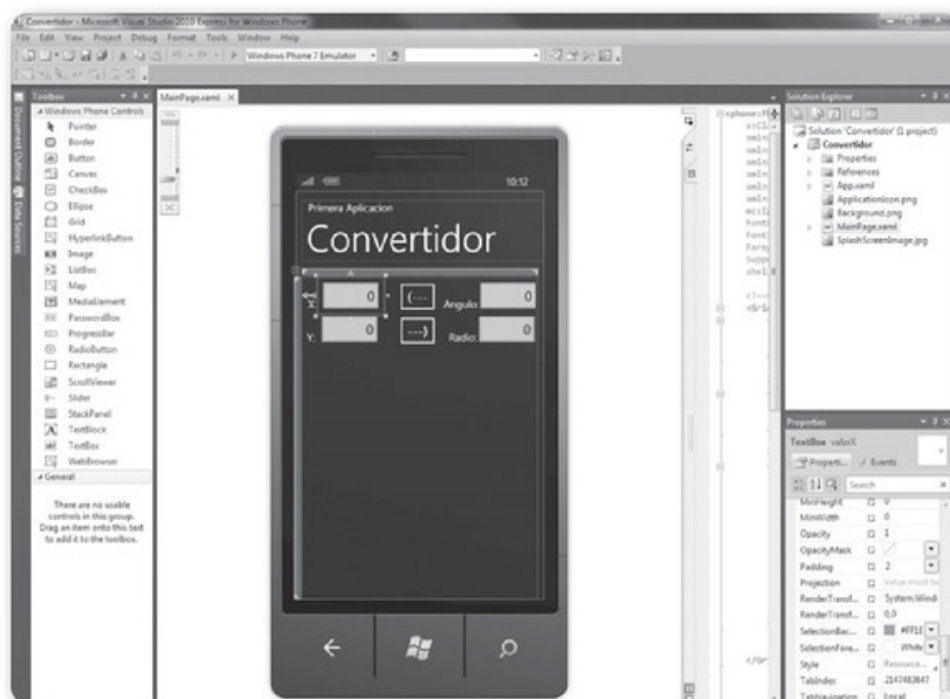
Empecemos a crear la interfaz. Esta necesita cuatro **TextBox**, dos botones y cuatro etiquetas. Para guiarnos, observamos la **Figura 10**.



REVISTA MSDN



En esta revista electrónica encontramos artículos sobre diversos temas referidos al desarrollo de software en la plataforma .NET. También es posible consultar información para el desarrollo de aplicaciones de Windows Phone 7, como el artículo que encontramos en la dirección <http://msdn.microsoft.com/en-us/magazine/gg598932.aspx>.



► **Figura 10.** En necesario que nuestra aplicación tenga una interfaz de usuario como la que se muestra en esta figura.

Colocamos los elementos de los datos para las coordenadas cartesianas. Ponemos un par de **TextBlock**; el primero tiene en su propiedad de **Text** el valor de “X:”, y el segundo, el valor de “Y:”. A cada uno de ellos les ponemos los **TextBox** correspondientes. La propiedad **Width** debe ser de **130**, y **Height**, de **72**. Para evitar cualquier problema, ponemos el valor “0” en la propiedad de **Text**. El primer **TextBox** es utilizado para guardar la coordenada **X**, por lo que, en la propiedad **Name**, colocamos **valorX**. Por medio de este nombre, el código en C# tendrá acceso al control en particular. El segundo **TextBox** se utiliza para la coordenada **Y**, por lo que en su propiedad **Name** colocamos **valorY**.

A continuación, agregamos dos botones. El primero se usa para convertir coordenadas polares en cartesianas, y el segundo, para convertir cartesianas en polares. El ancho de los botones es de **89** y la altura, de **72**. El primer botón se encuentra en la parte superior y le colocamos **btnPaC** en su propiedad **Name**. Para la propiedad de **Text** escribimos “(---”. El segundo botón tiene el valor de **botonCaP** en la propiedad **Name**, y en la propiedad de **Text** colocamos “---)”.

Ahora podemos incluir la sección que corresponde a la información de las coordenadas polares. Empezamos por poner dos **TextBlock**. En el primero, colocamos en la propiedad **Text** el valor de “**Angulo:**”, y en el segundo escribimos “**Radio:**” también en la propiedad **Text**. Luego, adicionamos los **TextBox** que son utilizados para introducir los valores necesarios. Sus dimensiones son de **130** por **72**. Es conveniente dar un valor de inicio a la propiedad **Text**, que en este caso hemos seleccionado como “**0**”. Esto nos ahorra problemas porque si el usuario oprime el botón de conversión sin que exista un valor, tendremos un error en el programa. El primer **TextBox** tiene en su propiedad **Name** el valor **valorAngulo**, y en el segundo colocamos **valorRadio** en esta misma propiedad.

Conforme hemos adicionado estos controles al usar la caja de herramientas y arrastrarlos, se ha agregado nuevo código automáticamente. Necesitamos observarlo ya que aún tenemos que hacer algunas pequeñas adiciones para tener la interfaz lista. El código final de la interfaz debe lucir de la siguiente manera:

CONOCER XAML NOS
PERMITE CONTROLAR
PRECISAMENTE LOS
ELEMENTOS DE LA
INTERFAZ

```
<phone:PhoneApplicationPage
  x:Class="Convertidor.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:phone="clr-namespace:Microsoft.Phone.Controls;assembly=Microsoft.
  Phone"
  xmlns:shell="clr-namespace:Microsoft.Phone.Shell;assembly=Microsoft.
  Phone"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  mc:Ignorable="d" d:DesignWidth="480" d:DesignHeight="768"
  FontFamily="{StaticResource PhoneFontFamilyNormal}"
  FontSize="{StaticResource PhoneFontSizeNormal}"
  Foreground="{StaticResource PhoneForegroundBrush}"
  SupportedOrientations="Portrait" Orientation="Portrait"
  shell:SystemTray.IsVisible="True">
  <!--LayoutRoot is the root grid where all page content is placed-->
```



```

<Grid x:Name="LayoutRoot" Background="Transparent">
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="*/"/>
    </Grid.RowDefinitions>
    <!--TitlePanel contains the name of the application and page title-->
    <StackPanel x:Name="TitlePanel" Grid.Row="0" Margin="12,17,0,28">
        <TextBlock x:Name="ApplicationTitle" Text="Primera Aplicacion"
            Style="{StaticResource PhoneTextNormalStyle}"/>
        <TextBlock x:Name="PageTitle" Text="Convertidor" Margin="9,-7,0,0"
            Style="{StaticResource PhoneTextTitle1Style}"/>
    </StackPanel>
    <!--ContentPanel - place additional content here-->
    <Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
        <TextBlock Height="30" HorizontalAlignment="Left" Margin="12,37,0,0"
            Name="textBlock1" Text="X:" VerticalAlignment="Top" />
        <TextBox Height="72" HorizontalAlignment="Left"
            Margin="29,0,0,0" Name="valorX" Text="0" VerticalAlignment="Top"
            Width="130" TextAlignment="Right" />
        <TextBlock Height="30" HorizontalAlignment="Left"
            Margin="10,100,0,0" Name="textBlock2" Text="Y:"
            VerticalAlignment="Top" />
        <TextBox Height="72" HorizontalAlignment="Left"
            Margin="27,63,0,0" Name="valorY" Text="0" VerticalAlignment="Top"
            Width="130" TextAlignment="Right" />

        <TextBlock Height="30" HorizontalAlignment="Left"
            Margin="274,37,0,0" Name="textBlock3" Text="Angulo:"
            VerticalAlignment="Top" />
    </Grid>
</Grid>

```



CÓDIGO PARA WINDOWS PHONE 7



Una de las mejores formas que tenemos para aprender a desarrollar aplicaciones para Windows Phone es por medio de ejemplos de código. En el siguiente sitio web se presentan muchos casos de aplicaciones que podemos aprovechar para aprender y, posteriormente, utilizar en nuestros proyectos; encontramos este sitio web visitando la dirección web [http://msdn.microsoft.com/en-us/library/ff431744\(v=VS.92\).aspx](http://msdn.microsoft.com/en-us/library/ff431744(v=VS.92).aspx).


```

<TextBox Height="72" HorizontalAlignment="Left" Margin="333,0,0,0"
Name="valorAngulo" Text="0" VerticalAlignment="Top" Width="130"
TextAlignment="Right" />
<TextBlock Height="30" HorizontalAlignment="Left"
Margin="284,100,0,0" Name="textBlock4" Text="Radio:"
VerticalAlignment="Top" />
<TextBox Height="72" HorizontalAlignment="Left" Margin="331,63,0,0"
Name="valorRadio" Text="0" VerticalAlignment="Top" Width="130"
TextAlignment="Right" />

<Button Content="(----" Height="72" HorizontalAlignment="Left"
Margin="179,2,0,0" Name="btnPaC" VerticalAlignment="Top"
Width="89" Click="btnPaC_Click" />
<Button Content="---)" Height="72" HorizontalAlignment="Left"
Margin="179,66,0,0" Name="btnCaP" VerticalAlignment="Top"
Width="89" Click="btnCaP_Click" />
</Grid>

</Grid>

</phone:PhoneApplicationPage>

```

Al observar el código, vemos que se ha creado un **Grid** para nosotros. Lo usaremos para crear nuestra interfaz de usuario y, dentro de él, podemos adicionar otros contenedores, cada uno para un uso particular de la interfaz. Dentro del **Grid** tenemos un **StackPanel** llamado **TitlePanel**. En general, se usa este **StackPanel** para colocar el título de la aplicación y el título de la página en particular en donde nos encontramos. Esto no es estrictamente necesario, pero en general se le da esta función. El título de la aplicación se ubica en el **TextBlock** llamado **ApplicationTitle**; para lograrlo, simplemente ponemos el título que nos interesa en su propiedad **Text**. Para el ejemplo que estamos programando colocaremos **"Primera Aplicación"**. Si observamos la interfaz de usuario, encontramos

Grid llamado **ContentPanel**, en donde va realmente la parte importante de nuestra interfaz de usuario, como los botones, **TextBox** y otros elementos. Podemos observar que los controles que fueron arrastrados se encuentran codificados en esta sección.

Debemos localizar el código de los botones porque necesitamos adicionar los handlers para el evento **Click**. Cada botón tiene su propio evento, donde, posteriormente, colocaremos el código necesario para hacer la conversión y el desplegado correspondiente. El código de los botones es el siguiente:

```
<Button Content="(----" Height="72" HorizontalAlignment="Left"
Margin="179,2,0,0" Name="btnPaC" VerticalAlignment="Top"
Width="89" Click="btnPaC_Click" />
<Button Content="---)" Height="72" HorizontalAlignment="Left"
Margin="179,66,0,0" Name="btnCaP" VerticalAlignment="Top"
Width="89" Click="btnCaP_Click" />
```

Para el primer botón, debemos encargarnos de adicionar el handler del evento **Click** con el nombre **btnPaC_Click**. En el caso del segundo botón, el handler del evento se llama **btnCaP_Click**.

El código de C# que debemos proceder a colocar en el handler de **btnPaC_Click** es el que vemos a continuación:

```
private void btnPaC_Click(object sender, RoutedEventArgs e)
{
    // Variables necesarias
    double angulo;
    double radio;
    double x;
    double y;
    // Obtenemos los valores
    angulo = Convert.ToDouble(valorAngulo.Text);
    radio = Convert.ToDouble(valorRadio.Text);
    // Convertimos el angulo a radianes
    angulo = angulo * (Math.PI/180.0);
    // Calculamos los valores
    x = radio * Math.Cos(angulo);
```

```
y = radio * Math.Sin(angulo);
// Escribimos los resultados
valorX.Text = x.ToString();
valorY.Text = y.ToString();
// Dibujamos
Canvas canv = new Canvas();
this.Content = canv;
canv.Margin = new Thickness(0, 0, 0, 0);
// Dibujamos eje X
Rectangle ejex = new Rectangle();
ejex.Fill = new SolidColorBrush(Colors.Blue);
ejex.Stroke = new SolidColorBrush(Colors.Blue);
ejex.Width = 450;
ejex.Height = 5;
ejex.SetValue(Canvas.LeftProperty, (double)0);
ejex.SetValue(Canvas.TopProperty, (double)350);
canv.Children.Add(ejex);
// Dibujamos eje Y
Rectangle ejey = new Rectangle();
ejey.Fill = new SolidColorBrush(Colors.Blue);
ejey.Stroke = new SolidColorBrush(Colors.Blue);
ejey.Width = 5;
ejey.Height = 800;
ejey.SetValue(Canvas.LeftProperty, (double)225);
ejey.SetValue(Canvas.TopProperty, (double)0);
canv.Children.Add(ejey);
// Dibujamos el punto
Rectangle punto = new Rectangle();
punto.Fill = new SolidColorBrush(Colors.Red);
punto.Stroke = new SolidColorBrush(Colors.Red);
punto.Width = 5;
punto.Height = 5;
punto.SetValue(Canvas.LeftProperty, (double)225+x);
punto.SetValue(Canvas.TopProperty, (double)350-y);
canv.Children.Add(punto);
// Escribimos los resultados
TextBlock resultadoX = new TextBlock();
```



```
resultadoX.Text = "X=" + x.ToString();
TextBlock resultadoY = new TextBlock();
resultadoY.Text = "Y=" + y.ToString();
resultadoY.SetValue(Canvas.TopProperty, (double)50);
canv.Children.Add(resultadoX);
canv.Children.Add(resultadoY);
}
```

La conversión se lleva a cabo de manera muy sencilla, pero con este código podemos aprender sobre nuevas clases que necesitamos utilizar. En el inicio del handler, vemos la declaración de las variables que vamos a emplear. Debemos tener en cuenta que estas son de tipo **double**, y sirven para guardar la información dada por el usuario y los valores calculados por nuestra aplicación.

Empezamos por obtener los valores de **ángulo** y **radio**, que son leídos de los **TextBox** **valorAngulo** y **valorRadio**. La información que escribe el usuario en estos controles se encuentra disponible en la propiedad **Text**. Aquí debemos tener cuidado, ya que **Text** guarda cadenas, pero nosotros necesitamos valores numéricos. Para resolver este problema, hacemos uso del método **ToDouble()**, perteneciente a la clase **Convert**.

El siguiente paso consiste en convertir el ángulo a radianes, ya que las funciones matemáticas que vamos a utilizar requieren los valores dados en radianes. Para hacer la conversión, usamos lo guardado en **Math.PI**.

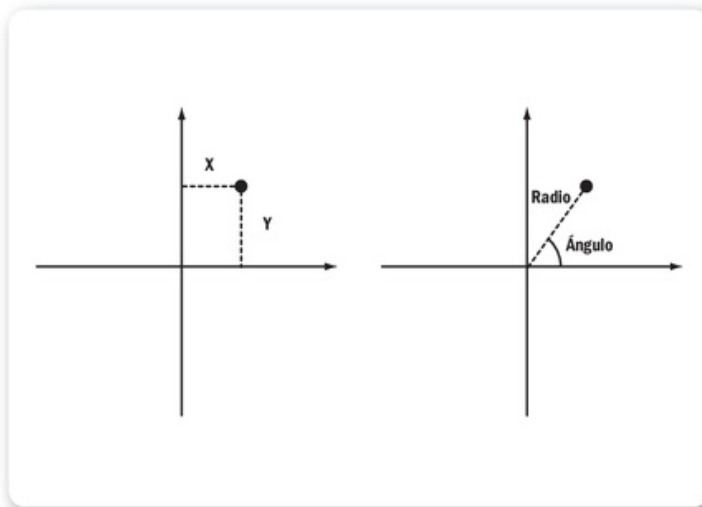
Con esto ya tenemos todos los datos necesarios para efectuar la conversión. Si trazamos una línea del origen al punto que queremos convertir, obtendremos un triángulo rectángulo cuya hipotenusa es igual al radio. Por medio de las funciones trigonométricas, ahora es fácil encontrar el valor de **X** e **Y**, que son los catetos del triángulo. Esto puede verse en la **Figura 11**.



LA CLASE MATH



Es necesario tener en cuenta que el framework de .NET se encarga de proveernos de una clase que nos apoya con diferentes operaciones y funciones matemáticas. Se trata de **Math** y pertenece al namespace **System**. Si necesitamos una función matemática, lo primero es buscarla en **Math**, antes de perder tiempo implementándola nosotros mismos.



► **Figura 11.** Esta figura muestra la relación trigonométrica que nos permite convertir de coordenadas polares a cartesianas.

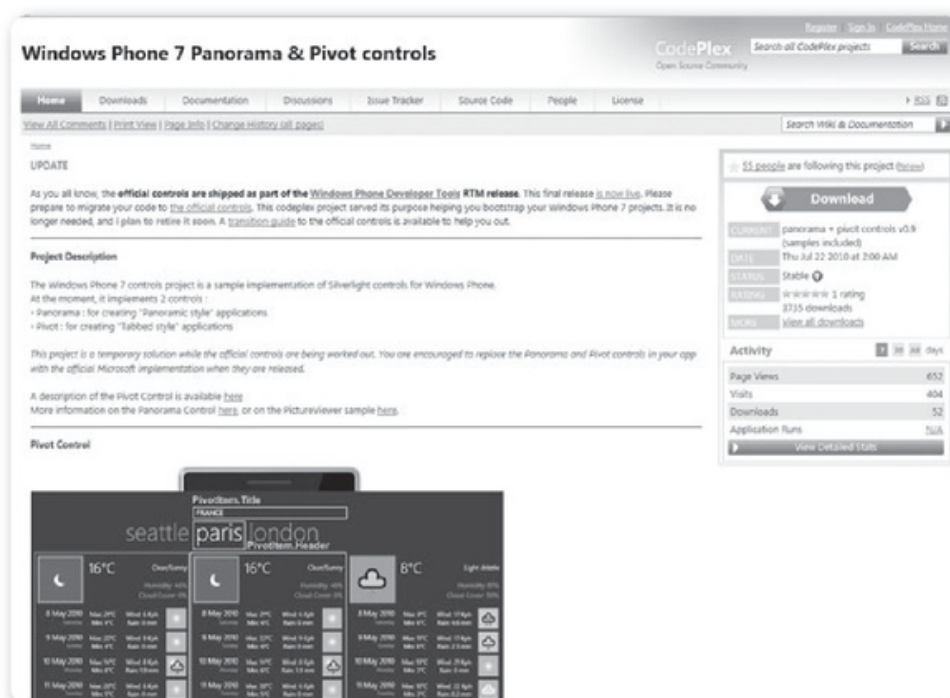
Con los valores de las coordenadas cartesianas, procedemos a escribirlos en los **TextBox** correspondientes. Después de hacerlo, podemos dibujar el punto convertido. El código que escribimos en XAML tiene un equivalente en C#; de hecho, los **tags** de XAML en su mayoría corresponden a clases que podemos usar de la manera tradicional.



► **Figura 12.** Se pueden encontrar en el mercado bibliotecas de controles avanzados para las interfaces de Windows Phone.

Para hacer nuestro dibujo, vamos a utilizar un contenedor de tipo **Canvas**, por lo cual creamos una instancia de la clase **Canvas** y la llamamos **canv**. Colocamos como contenido de la aplicación el **Canvas** que acabamos de crear, para que sea el elemento que se desplegará.

Para dibujar los ejes, usamos rectángulos. Empezamos con el eje **X** y creamos un objeto de tipo **Rectangle** llamado **ejex**. Luego procedemos a colocar las propiedades más importantes del rectángulo. El nuestro estará relleno de color azul usando una brocha sólida, para lo cual recurrimos a la propiedad **Fill**. La propiedad **Stroke** indica de qué forma se va a dibujar el marco o borde del rectángulo; para él, también hemos seleccionado una brocha sólida de color azul.



► **Figura 13.** Existen diversos sitios web en los que se encuentran controles que podemos utilizar para nuestros proyectos de Windows Phone.



EL MÉTODO SETVALUE()

Cuando estamos trabajando desde el código de C#, debemos tener en cuenta que es muy común que precisemos colocar un valor a una propiedad de dependencia del objeto que estamos utilizando en nuestra aplicación. Recordemos que, para lograrlo, contamos con el método denominado **SetValue()**. En el primer parámetro es necesario que indicar la propiedad a la que deseamos colocar el valor, en tanto que el segundo parámetro corresponde al valor por colocar.

El eje **Y** es un rectángulo más largo que alto. Para colocar sus dimensiones hacemos uso de las propiedades **Width** y **Height**. El ancho tiene un valor de **450** y el alto es de **5**. El rectángulo necesita ser posicionado en algún lugar dentro del canvas para que se muestre en la pantalla. Queremos que se vea aproximadamente en el centro, de modo que indicamos el valor desde el borde izquierdo y el borde superior del canvas. Como estamos colocando el código desde C#, entonces hacemos uso de la función **SetValue()**.

La distancia a partir del borde izquierdo es de **0**, pero la distancia que colocamos a partir del borde superior es de **350**. Con esto tenemos el rectángulo cerca del centro de la pantalla. Ahora ya tenemos el objeto tal como nos interesa, pero falta ponerlo en el interior del **Canvas** para que sea dibujado. **Canvas** tiene en su interior una colección llamada **Children**, en la que se guardan todos los elementos que pertenecen al **Canvas** y que queremos que sean desplegados. Lo único que debemos hacer es adicionar el elemento a esta colección. Para esto, usamos el método **Add()**, pasando como parámetro el elemento que deseamos adicionar.



**BUSCAR INFORMACIÓN
EN EL SITIO WEB DE
MICROSOFT ES DE
GRAN UTILIDAD**



Para el dibujo del eje **Y** hacemos algo similar. Emplearemos un rectángulo de color azul, pero necesitamos modificar algunos valores para que tenga la forma que precisamos. Para este eje, la propiedad de **Width** debe tener el valor de **5**, y la de **Height**, de **800**. Con esto creamos un rectángulo delgado de gran altura. También tenemos que posicionar

el rectángulo de forma correcta; en este caso, indicamos la distancia desde el borde izquierdo en **225**, y la correspondiente desde el borde superior en **0**. Esto se lleva a cabo por medio del método **SetValue()** y, luego, simplemente adicionamos el rectángulo **ejeY** al **canvas**.

El siguiente elemento por dibujar es el punto que hemos convertido a coordenadas cartesianas. Empezamos por crear un objeto de tipo **Rectangle** al cual llamamos **punto**. Este será dibujado de color rojo, por lo que hacemos uso

de una brocha de color sólido para su propiedad **Fill** y la propiedad **Stroke**. En ambas usamos **Colors.Red**. Nuestro punto tiene que ser pequeño y no tomar tanto espacio como los ejes. También es necesario que el rectángulo tenga el mismo valor para el ancho y el alto. Entonces, debemos tener en cuenta que en las propiedades **Width** y **Height** ponemos un valor de **5**.

Colocar la posición del punto adentro del **Canvas** es un poco más complejo. Es cierto que ya tenemos las coordenadas del punto, pero estas no son coordenadas de pantalla; por lo tanto, debemos transformarlas tomando en cuenta que nuestro origen se ubica en las coordenadas **(225,350)** y que el eje **Y** se incrementa hacia abajo. Afortunadamente, esto es muy sencillo de hacer, y solo tenemos que emplear una traslación para solucionarlo:



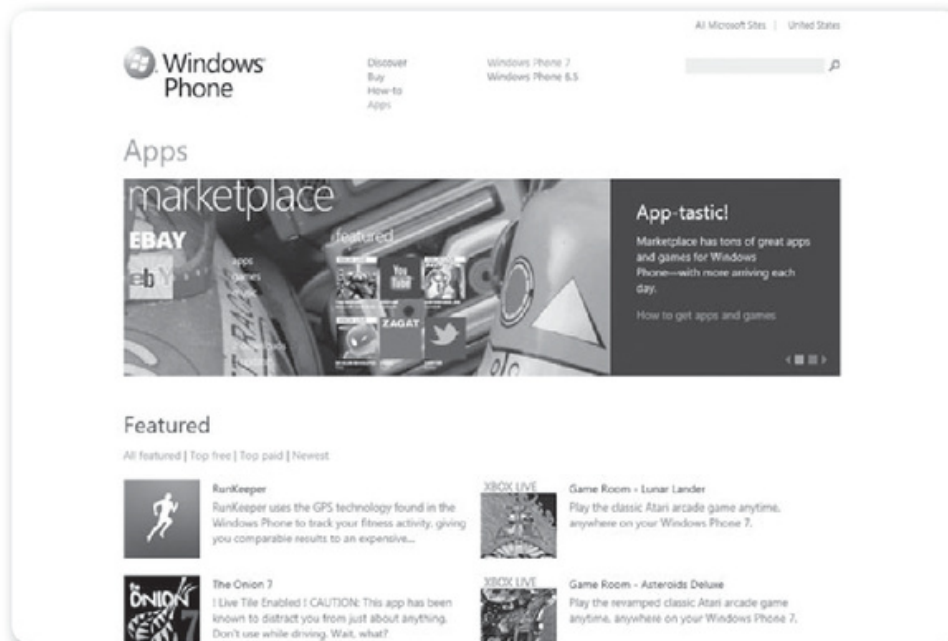
DESARROLLO PARA WINDOWS PHONE 7



En el sitio web que se ubica en la dirección [http://msdn.microsoft.com/en-us/library/ff402531\(v=VS.92\).aspx](http://msdn.microsoft.com/en-us/library/ff402531(v=VS.92).aspx) podemos encontrar información muy útil para conocer a fondo la manera en que trabaja Windows Phone 7 y aprender más sobre esta plataforma. Puede tomar tiempo leer todos los artículos, pero es valioso hacerlo ya que poseen información importante.


```
punto.SetValue(Canvas.LeftProperty, (double)225+x);  
punto.SetValue(Canvas.TopProperty, (double)350-y);
```

Para colocar correctamente el valor en el eje **X**, adicionamos **225** al valor calculado por la fórmula. Esto se debe a que el origen se encuentra a **225** unidades del borde izquierdo. En el caso del eje **Y**, el origen se ubica a **350** unidades del borde superior del **Canvas**; la razón por la que estamos restando el valor de **Y** se debe a que en el **Canvas** este valor se incrementa hacia abajo, pero en las coordenadas cartesianas, lo hace hacia arriba; como van en sentido contrario, restamos el valor. Después de hacerlo, simplemente adicionamos el punto al **Canvas**.



► **Figura 15.** Microsoft ofrece mecanismos para que las aplicaciones para Windows Phone 7 sean distribuidas (<http://marketplace.windowsphone.com/Default.aspx>).

Por último, debemos encargarnos de escribir

Para que los **TextBlock** no se superpongan en el **Canvas** ponemos **resultadoY** un poco más abajo. Es sencillo hacerlo: colocamos el valor de **50** en la propiedad **TopValue** por medio del método **SetValue()**. Luego, adicionamos los **TextBlock** al **Canvas** que corresponde.

Con esto, ya podemos probar la aplicación. Simplemente, colocamos algunos valores en el radio y el ángulo. Al oprimir el botón de conversión, se mostrarán la gráfica y los resultados.



► **Figura 16.** En esta figura podemos observar la gráfica que muestra el resultado de la conversión.

Ahora ya podemos pasar al código correspondiente al handler del botón para que se realice la conversión de coordenadas cartesianas a coordenadas polares, de la siguiente forma:

```
private void btnCaP_Click(object sender, RoutedEventArgs e)
{
    // Variables necesarias
    double angulo;
```

```
double radio;
double x;
double y;
// Obtenemos los valores
x = Convert.ToDouble(valorX.Text);
y = Convert.ToDouble(valorY.Text);
// Calculamos los valores
radio = Math.Sqrt(x * x + y * y);
angulo = Math.Atan(y / x);
angulo = angulo * (180.0 / Math.PI);
// Escribimos los resultados
valorAngulo.Text = angulo.ToString();
valorRadio.Text = radio.ToString();
// Dibujamos
Canvas canv = new Canvas();
this.Content = canv;
canv.Margin = new Thickness(0, 0, 0, 0);
// Dibujamos el circulo para el radio
Ellipse el = new Ellipse();
el.Fill = new SolidColorBrush(Colors.Green);
el.Stroke = new SolidColorBrush(Colors.Green);
el.Width = radio * 2;
el.Height = radio * 2;
el.SetValue(Canvas.LeftProperty, (double)225 - radio);
el.SetValue(Canvas.TopProperty, (double)350 - radio);
canv.Children.Add(el);
// Dibujamos el eje X
Rectangle ejex = new Rectangle();
```



BASES DE WINDOWS PHONE 7



En el sitio web que encontramos en la dirección <http://msdn.microsoft.com/en-us/ff380145.aspx> podemos hallar información básica para comprender el funcionamiento y la programación de Windows Phone 7. Los temas empiezan para principiantes, y finalizan en un nivel intermedio. Después de leer este capítulo, es un buen lugar para continuar aprendiendo.

```
ejex.Fill = new SolidColorBrush(Colors.Blue);
ejex.Stroke = new SolidColorBrush(Colors.Blue);
ejex.Width = 450;
ejex.Height = 5;
ejex.SetValue(Canvas.LeftProperty, (double)0);
ejex.SetValue(Canvas.TopProperty, (double)350);
canv.Children.Add(ejex);
// Dibujamos el eje Y
Rectangle ejey = new Rectangle();
ejey.Fill = new SolidColorBrush(Colors.Blue);
ejey.Stroke = new SolidColorBrush(Colors.Blue);
ejey.Width = 5;
ejey.Height = 800;
ejey.SetValue(Canvas.LeftProperty, (double)225);
ejey.SetValue(Canvas.TopProperty, (double)0);
canv.Children.Add(ejey);
// Dibujamos el punto
Rectangle punto = new Rectangle();
punto.Fill = new SolidColorBrush(Colors.Red);
punto.Stroke = new SolidColorBrush(Colors.Red);
punto.Width = 5;
punto.Height = 5;
punto.SetValue(Canvas.LeftProperty, (double)225 + x);
punto.SetValue(Canvas.TopProperty, (double)350 - y);
canv.Children.Add(punto);
// Escribimos los valores
TextBlock resultadoR = new TextBlock();
resultadoR.Text = "Radio=" + radio.ToString();
```



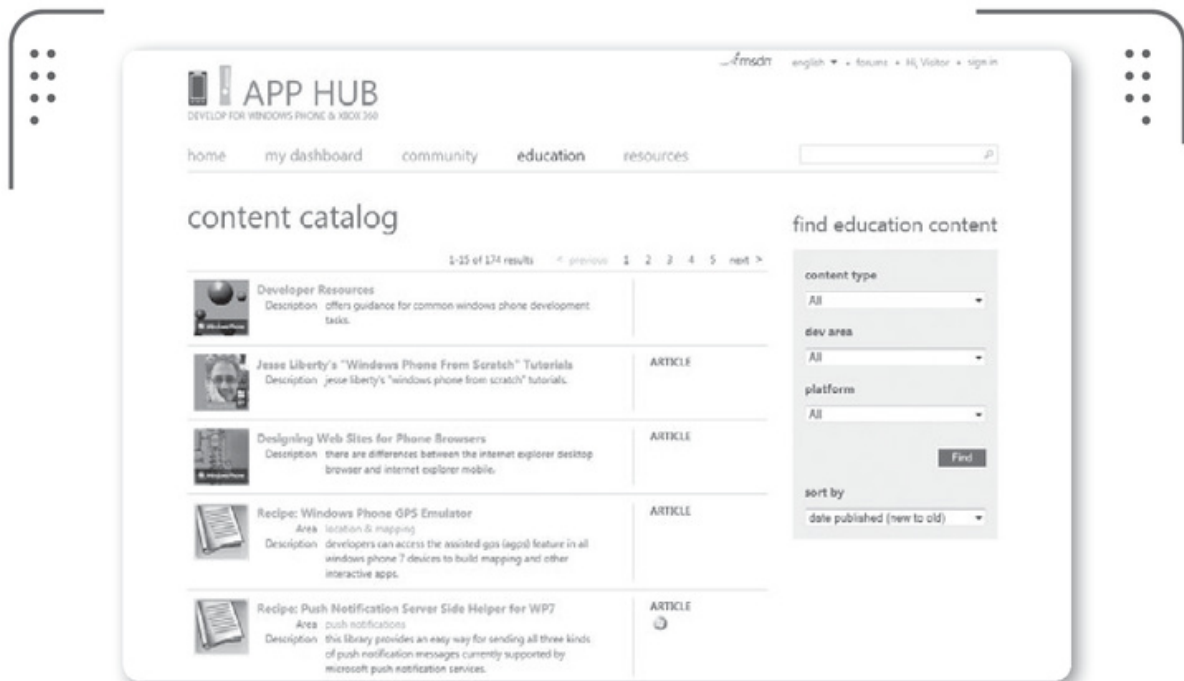
CURSO DE WINDOWS PHONE 7



Microsoft ha creado un curso de entrenamiento para el desarrollo de aplicaciones en Windows Phone 7. Está diseñado para principiantes, y es un muy buen lugar para empezar a aprender esta tecnología. Con este curso obtendremos las bases necesarias para empezar a desarrollar correctamente: <http://msdn.microsoft.com/en-US/wp7trainingcourse.aspx>.


```
TextBlock resultadoA = new TextBlock();
resultadoA.Text = "Angulo=" + angulo.ToString();
resultadoA.SetValue(Canvas.TopProperty, (double)50);
canv.Children.Add(resultadoR);
canv.Children.Add(resultadoA);
}
```

El código es sencillo. Es necesario que empecemos por obtener los valores de los **TextBox** y convertir esas coordenadas cartesianas a coordenadas polares. El ángulo queda calculado en radianes, por lo que es preciso hacer la conversión a grados.



► **Figura 17.** En el App Hub de MSDN podemos encontrar, en la parte de educación, muchos cursos y recursos para aprender más sobre la programación de aplicaciones para Windows Phone.

Debemos tener en cuenta que de manera similar al handler anterior, creamos un canvas. En este caso, dibujamos los ejes, y también, un círculo con un radio igual al calculado. Ya sabemos cómo dibujar rectángulos, pero en este caso precisamos un círculo. Entonces, recurrimos a la clase conocida como **Ellipse**.

Para rellenar la elipse, usamos una brocha de color sólido. El marco de la figura también se pinta con una brocha sólida del mismo color. A las propiedades de **Width** y **Height** les colocamos el doble del radio como valor. El centro del círculo debe estar sobre el origen, por lo que también llevamos a cabo una traslación para colocarlo correctamente.

Los ejes y el punto se dibujan de forma similar al handler anterior. Para finalizar, simplemente se muestran los valores correspondientes. Ahora podemos compilar la aplicación y probar esta conversión.



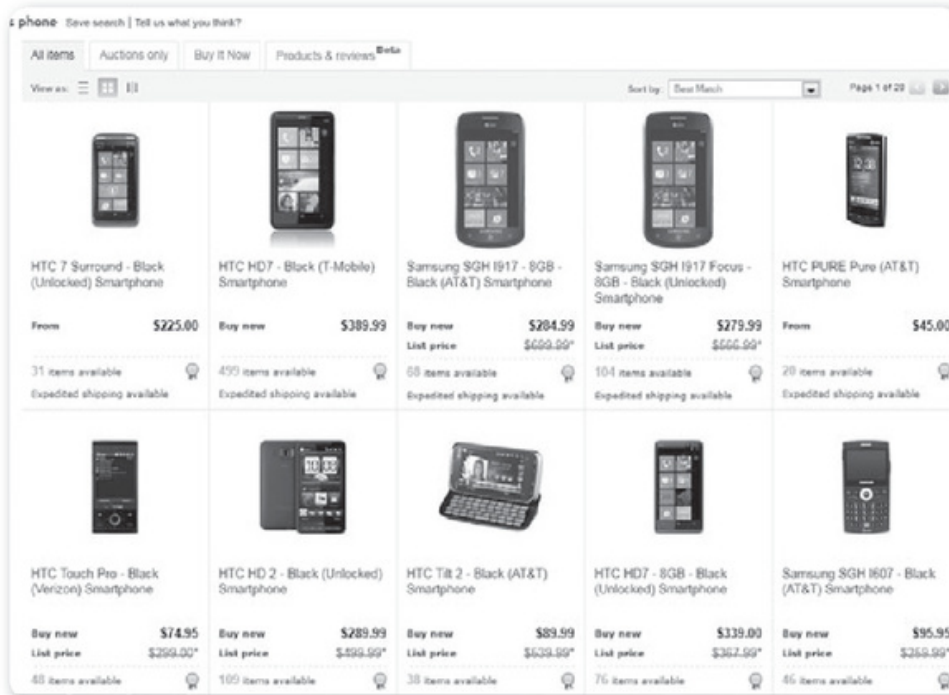
► **Figura 18.** En esta imagen podemos darnos cuenta de que se muestran las coordenadas polares del punto que corresponde.



LA CLASE ELLIPSE

Es necesario tener en cuenta que la clase **Ellipse** sirve para dibujar tanto círculos como elipses. Para lograr esto, será necesario que procedamos a colocar los valores correctos en las propiedades **Width** y **Height**. Si ambas propiedades tienen el mismo valor, entonces obtendremos un círculo, mientras que si tienen valores diferentes, lograremos una elipse.

Con todo lo visto hasta el momento, tenemos los conocimientos básicos necesarios para programar una aplicación sencilla de Windows Phone 7. El siguiente paso es conocer más sobre Silverlight y XNA, y manejar a fondo el framework de Windows Phone 7.



► **Figura 19.** En el mercado encontramos varios teléfonos que ya se encargan de utilizar esta tecnología.



RESUMEN

Microsoft desarrolló Windows Phone 7 para competir en el mercado de los smartphones. Dentro de su diseño, incluyó una serie de especificaciones mínimas que los fabricantes tienen que cumplir. De este modo, se estandariza el hardware y se simplifica el desarrollo de las aplicaciones. Para el hardware existen especificaciones de pantalla, acelerómetro, radio, Wi-Fi, etc. Las aplicaciones pueden desarrollarse en C# haciendo uso de Silverlight o de XNA. La pantalla del teléfono tiene orientación, y nuestras aplicaciones pueden cambiar dinámicamente la interfaz de usuario. La pantalla también tiene capacidades multi-touch, por lo que puede detectar varios puntos de toque.

Actividades

TEST DE AUTOEVALUACIÓN

- 1 ¿Cuál es el objetivo de Windows Phone 7?
- 2 ¿Por qué Microsoft da especificaciones estrictas para el hardware de Windows Phone 7?
- 3 ¿Qué tipos de aplicaciones se pueden crear para Windows Phone 7?
- 4 ¿Cuáles son las características de la pantalla?
- 5 ¿Qué es un sistema multi-touch?
- 6 ¿De qué forma se puede utilizar el multi-touch?
- 7 ¿Cuáles son las posiciones de la pantalla?
- 8 ¿Qué se precisa para cambiar la orientación de la interfaz de usuario?
- 9 ¿Qué herramienta es necesaria para desarrollar aplicaciones de Windows Phone 7?
- 10 ¿Cómo funciona el emulador?
- 11 ¿Qué tipo de aplicaciones podemos desarrollar con Silverlight?
- 12 ¿Qué tipo de aplicaciones podemos desarrollar con XNA?

EJERCICIOS PRÁCTICOS

- 1 Desarrolle una aplicación que responda al cambio de orientación de la pantalla.
- 2 Desarrolle una aplicación básica de XNA.
- 3 Realice una aplicación que utilice diversos controles en la interfaz de usuario.
- 4 Experimente con el touch de la pantalla usando el emulador.
- 5 Investigue sobre la programación de Silverlight.

Programar en WCF

Windows Communication Foundation, también conocido como WCF, forma parte del framework de .NET y es de gran ayuda para construir aplicaciones distribuidas de manera sencilla y eficiente. Ahora, podemos crear software distribuido que se ejecute en nuestra propia máquina o en la Red de una manera simple y sin complicaciones. En este interesante capítulo nos encargaremos de conocer y analizar la forma correcta en que debemos programar en WCF para sacarle el máximo provecho.

▼ **Entender Windows
Communication Foundation..268**
El modelo de comunicación 272

▼ **Creación de
aplicaciones WCF277**

Código para
un servicio WFC..... 283

▼ **Resumen.....297**

▼ **Actividades.....298**



Entender Windows Communication Foundation

La programación de aplicaciones distribuidas es un área interesante dentro de las ciencias computacionales y la programación de computadoras. En un principio, este tipo de programación era compleja y requería de muchos conocimientos para llevarla a cabo. Por suerte, con el tiempo surgieron diversas APIs que nos permitieron simplificar un poco la creación de aplicaciones.

Podemos considerar que un sistema distribuido es aquel que está constituido por dos o más computadoras ejecutando procesos que se comunican entre sí y tienen un objetivo común. Cuando se tiene un problema que requiere de mucho poder de cómputo para resolverse, es posible usar un sistema distribuido para que sea más rápido. En ese caso, varias computadoras trabajan simultáneamente, cada una de ellas resolviendo una parte del problema.

En la actualidad, podemos usar el sistema distribuido incluso si no se trata de problemas complejos: pueden ser programas sencillos que lleven a cabo funciones diferentes del sistema en cada una de las máquinas usadas. También, ya es posible considerar que un sistema distribuido puede ejecutarse en una sola computadora, corriendo dos programas que se comunican entre sí.

En un principio, era posible programar el sistema distribuido totalmente a mano, pero esta era una labor tediosa y podíamos tener errores en el código. Con el tiempo, empezaron a surgir algunas API que intentaron simplificar este proceso. Una de ellas es **DCOM**, que utilizaba objetos de tipo **COM** para llevar a cabo el sistema distribuido. Sin embargo, los sistemas de este tipo no



TRANSPARENCIA DE LOCACIÓN



Es interesante tener en cuenta que el término **Location Transparency**, que es una de las principales características de DCOM, indica que la locación física de los objetos remotos no importa, y que las conexiones se verán de forma similar a las locales. Esto se encarga de facilitar la programación porque el objeto se maneja de la misma manera que los locales.

solo se ejecutan bajo el mismo sistema operativo, y DCOM era empleado, principalmente, en sistemas operativos Windows, la cual se presentaba como una de sus principales desventajas.

Con el tiempo surgió otra API, denominada **COM+**, que es una evolución sobre la anterior, y aumenta considerablemente sus características y propiedades. Ella permite administrar las transacciones, la seguridad, el tiempo de vida del objeto, y otros parámetros. Su uso ayudó mucho a reducir la cantidad de código necesario. Con el framework de .NET se puede emplear COM+, por lo que podemos realizar aplicaciones compatibles con ella.

Cuando se trabaja con un sistema distribuido, siempre se corre el riesgo de que exista algún problema con la red o las computadoras conectadas. Si esto sucede, puede llegar a perderse la información que se envía. La manera habitual de solucionar este inconveniente es mantener la información que se va a enviar hasta que se reestablezca la comunicación, lo que se conoce como **queuing data**. Microsoft desarrolló **MSMQ**, que significa **Microsoft Message Queuing**, para tener un sistema que use el queuing data y garantice el envío correcto de la información.

Otra API que se desarrolló con la evolución de .NET es **.NET Remoting**. Básicamente, es un sustituto para DCOM dentro del framework de .NET. Una adición interesante de esta API es el uso de los archivos de tipo ***.config**, que dan mucha flexibilidad al sistema distribuido ya que podemos cambiar su configuración sin recompilar. Simplemente, es necesario modificar el archivo ***.config** y arrancar otra vez la aplicación. Con .NET Remoting se puede tener un sistema

WCF NOS PERMITE
HACER APLICACIONES
DISTRIBUIDAS
DE MANERA MUY
SENCILLA



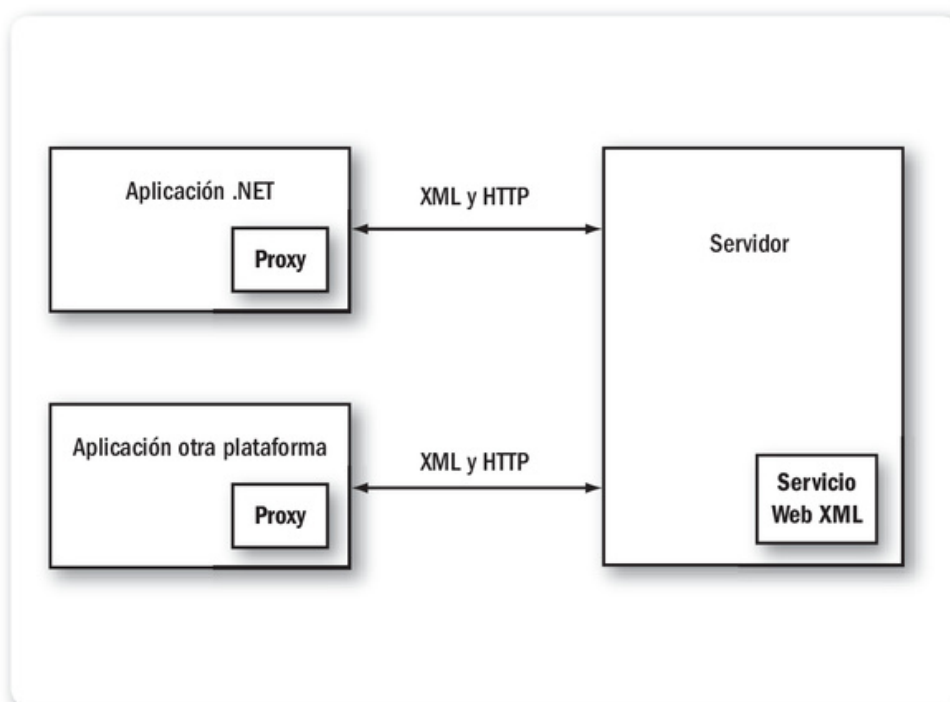
MICROSOFT



Microsoft nos provee, en MSDN, de un sitio con mucha información interesante sobre WCF. Este recurso es importante y podemos aprender mucho sobre esta tecnología visitando las diferentes secciones. Encontraremos documentación, ejemplos, artículos y cursos en la dirección <http://msdn.microsoft.com/es-mx/library/ms735119%28v=VS.90%29.aspx>.

distribuido que trabaje tanto en Windows como en Linux, si hacemos uso de las características que nos ofrece Mono.

En ocasiones, es necesario tener un sistema distribuido que trabaje con diferentes sistemas operativos y distintos lenguajes de programación, como C++, Java y C#. Frente a este caso, lo más común es hacer uso de XML y servicios web. De esta manera, se simplifican el envío y la recepción de la información, pero hay que pagar un precio con relación al desempeño. Al usar HTTP y XML, la información puede fluir sin problema en el sistema haciendo uso de las tecnologías actuales. No importa si el cliente en una máquina corre en Java con Linux, y en otra tenemos Windows y una aplicación de C#: el flujo de la información es sencillo.



► **Figura 1.** Este diagrama muestra la forma en que puede funcionar un sistema distribuido que hace uso de XML y web services.

Cuando diseñamos un sistema distribuido, debemos considerar varios factores para la selección del API que sea más conveniente. El sistema distribuido puede correr adentro de la misma empresa o hacerlo en el exterior. En el primer caso, es más sencillo tener cierto

control sobre los sistemas operativos en los que se va a ejecutar. Si corremos con suerte, todas las computadoras tendrán el mismo SO, lo que nos permite hacer una selección del API que nos dé el mayor desempeño. En cambio, cuando el sistema corre en el exterior, tenemos poco o nada de control sobre las plataformas en las que se va a ejecutar. Entonces, debemos seleccionar una API que sea flexible en su comunicación, aunque el desempeño no sea muy grande.

La seguridad también es un punto importante, y no todos los sistemas distribuidos necesitan el mismo grado de protección. Algunos que corren en la misma oficina o en una sola máquina requerirán niveles mínimos. También hay casos de sistemas que transmiten información muy importante entre computadoras que se encuentran lejanas, en cuyo caso la seguridad tendrá que ser significativamente mayor.

WCF nos permite tomar lo mejor de las tecnologías existentes, y reunir las en un solo paquete poderoso y sencillo de utilizar. Esto reduce considerablemente los tiempos de desarrollo de las aplicaciones distribuidas y mantiene un alto grado de compatibilidad con diversas API para llevar a cabo la comunicación. WCF aparece en la versión 3.0 del framework de .NET. Unifica .NET Remoting, **Web Services**, **WSE**, **Enterprise Services** y **Messaging**. El modelo que usa es orientado a servicios, y maneja una gran cantidad de estándares para web services, como **SML**, **SOAP**, **XSD**, **XPath** y **WSDL**. También soporta **WS-Security**, **WS-Addressing**, **WS-Policy**, **WS-Trust**, **WS-Reliable Messaging**, **WS-Secure Conversation**, **WS-Atomic Transaction**, **WS-Policy** y **WS-Coordination**.

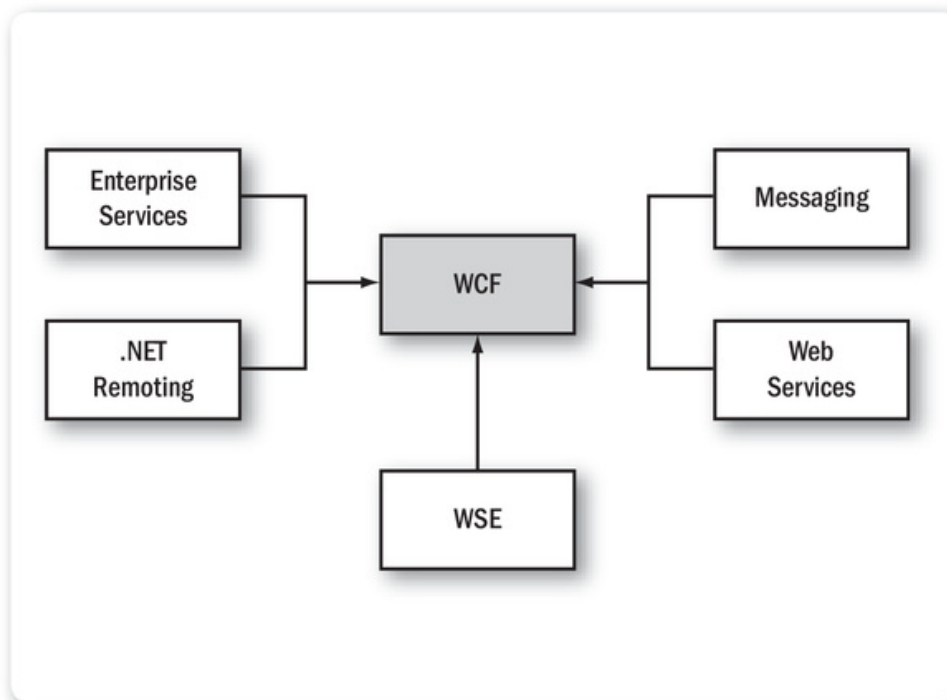
WCF PERMITE
DIFERENTES NIVELES
DE SEGURIDAD
Y DESEMPEÑO
PARA NUESTRAS
APLICACIONES



QUÉ ES UN ENDPOINT



Debemos tener en cuenta que el concepto de **endpoint** es muy sencillo: simplemente, se trata de una locación bien definida por medio de la cual los mensajes pueden ser enviados o recibidos. Es importante recordar que los servicios pueden tener uno o varios endpoints, que es el punto donde se lleva a cabo la salida o entrada de la información enviada.



► **Figura 2.** En este diagrama vemos las tecnologías que unifica WCF.

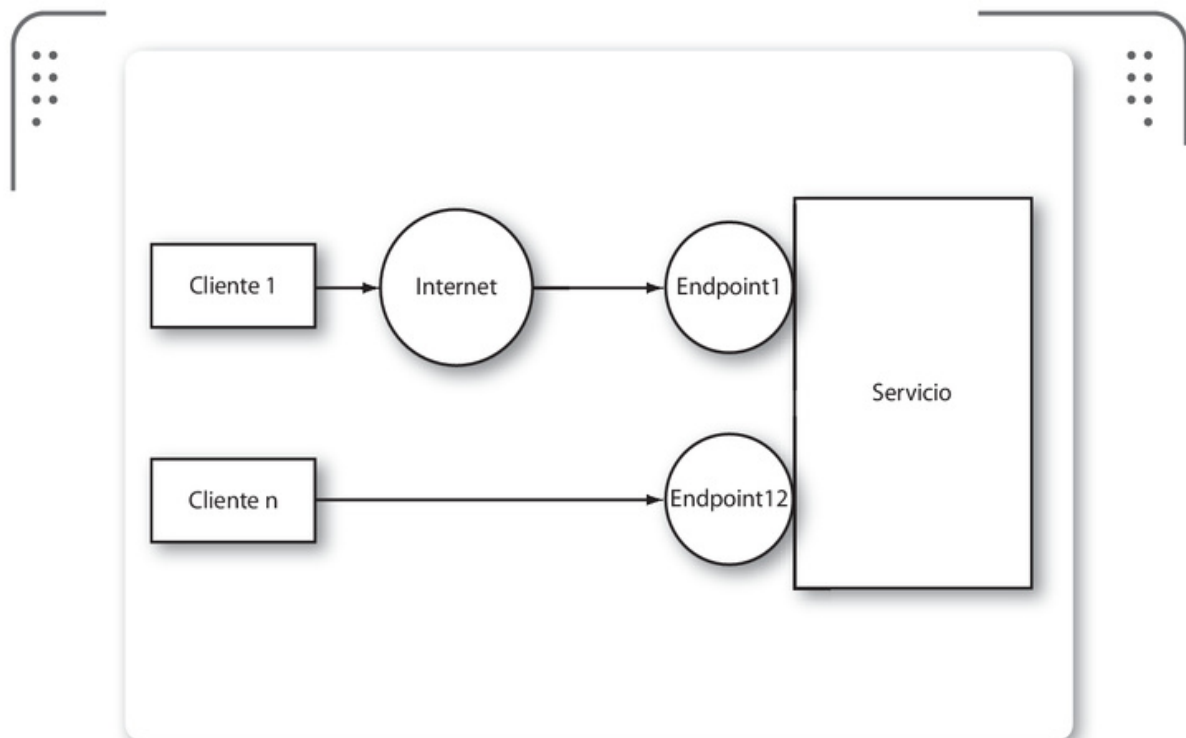
El modelo de comunicación

El modelo de comunicación que utiliza WCF es muy sencillo y se basa en **SOA**. En su forma más simple, tendremos una aplicación que funciona como servicio, y uno o varios clientes que pueden conectarse a él. El servicio expone al exterior una serie de **endpoints**, y el cliente los utiliza para tener acceso a los servicios. Para mandar el mensaje, es muy importante tener interfaces bien definidas.

Recordemos que SOA posee cuatro reglas de diseño que son utilizadas por WFC. En primer lugar, cualquier servicio debe tener sus interfaces correctamente definidas, ya que la única forma que es posible usar para comunicarse con el servicio es por medio de la interfaz. Como en cualquier interfaz, tenemos que conocer bien cada miembro, el valor de retorno así como también los parámetros con sus tipos respectivos. Los servicios que creamos en nuestras aplicaciones WFC tienen que ser autónomos.

Otro punto importante es la comunicación que llevan a cabo los servicios: estos deben comunicarse exclusivamente por contratos.

Esto significa que cuando alguien hace uso del servicio, no tiene que conocer las implementaciones internas de este; lo único que importa es la interfaz. En caso de que estemos usando tipos definidos por el programador, debemos tener también un contrato de datos donde se defina la estructura de estos datos. WCF toma en serio la compatibilidad de servicios basados en **pólizas** o **policy**. Esto es muy bueno ya que es posible acceder a separar las descripciones del servicio, de la forma en que se encarga de trabajar.



► **Figura 3.** Este diagrama muestra, de forma básica, el modelo de comunicación que corresponde a WCF.

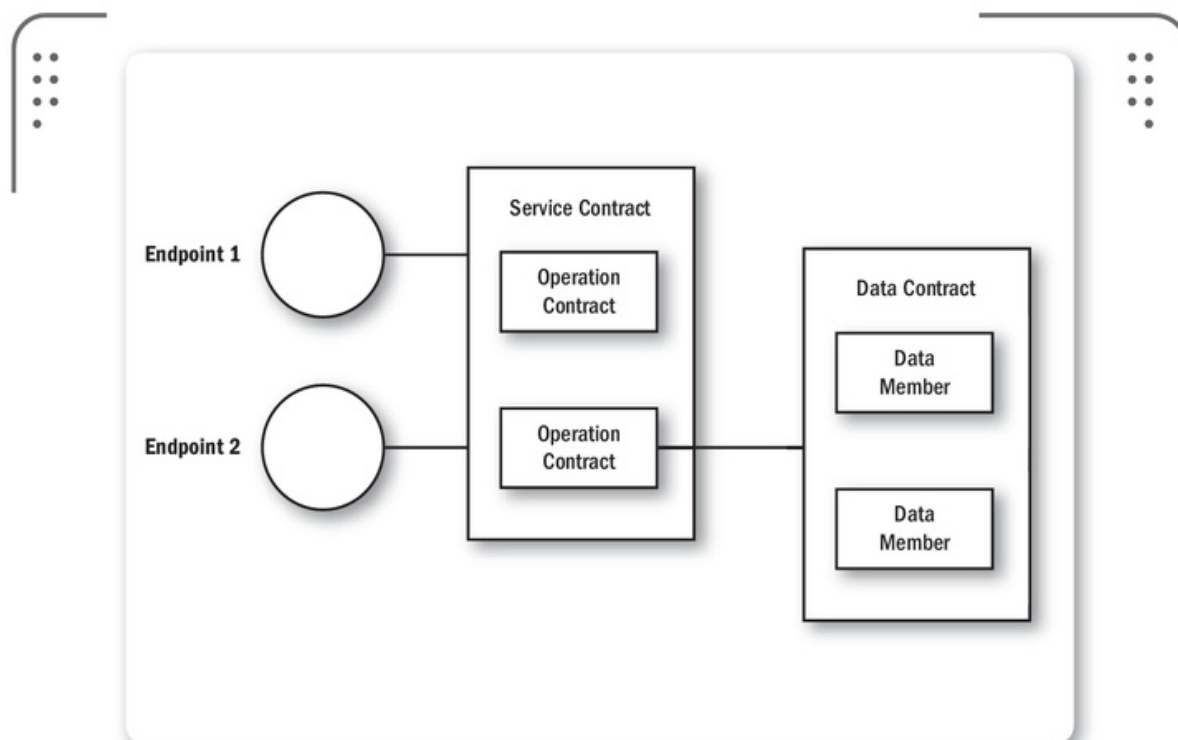


GLOSARIO DE TÉRMINOS



Algunas veces podemos tener dudas sobre el significado de determinado término en WCF. Afortunadamente, Microsoft ha creado un glosario de términos que se encuentra en línea. Conocer los términos facilita el aprendizaje y el desarrollo. Lo encontramos en la dirección <http://msdn.microsoft.com/es-mx/library/bb675151%28v=VS.90%29.aspx>.

El servicio tiene una serie de componentes, y debemos comprenderlos para utilizarlo en forma correcta. Estos componentes son: contrato de servicio, contrato de operaciones, contrato de datos y datos miembro.



► **Figura 4.** En este diagrama podemos observar la forma en que se relacionan entre sí los diferentes componentes de un contrato.

WCF permite el envío de mensajes utilizando los tipos definidos en el CLR, y también existe la opción de hacerlo por tipos definidos por el programador. También podemos usar diferentes **bindings** para mandar la información. Podemos hacer uso de **HTTP**, **MSMQ**, **TCP**,



LAS TIPOS Y [SERVICECONTRACT]



Es muy importante tener en cuenta que cuando en nuestro contrato de servicios vamos a usar una interfaz que emplea totalmente tipos CLR, podemos recurrir a **[ServiceContract]** directamente. En caso de usar tipos definidos por el programador, tenemos que especificar un namespace para ellos, porque es necesario garantizar que sean únicos.

etc. Un factor importante es el de la seguridad. WCF ha aprendido esto de las APIs anteriores, y presenta los protocolos de seguridad que ofrecen Windows y el framework de .NET, así como técnicas de seguridad de los estándares de servicios web. Ya tocando el tema de los servicios web, no debemos olvidarnos de que también brinda soporte para las especificaciones **WS***.

Contrato de servicios

Cuando tenemos dos programas que van a estar comunicados, es importante conocer la forma en que se lleva a cabo la conversación y los mensajes que se envían. El contrato de servicios hace precisamente esto, porque permite especificar la dirección y el tipo de mensajes.

Aunque esto parece muy complicado de realizar, en realidad, es muy sencillo, pues lo único que necesitamos hacer es definir una interfaz o, en algunos casos, una clase. Con esto, creamos una puerta para que las aplicaciones que usan nuestros servicios hagan uso de las funciones expuestas. Un punto que no tenemos que olvidar es que cada servicio debe tener, al menos, un contrato de servicios.

Para proceder a realizar la definición del contrato de servicios, recurrimos al atributo **[ServiceContract]**, que decora nuestra interfaz convirtiéndola en un contrato de servicios. En su forma más básica, usaremos la sintaxis que mostramos a continuación:

```
[ServiceContract]
public interface IMiServicio
{
    // Aquí van los métodos a exponer
}
```

El atributo **[ServiceContract]** tiene una serie de propiedades que podemos emplear. Dependiendo del binding por utilizar, en algunos casos optamos por algunas de ellas. Veamos brevemente cuáles son. La propiedad **Name** es muy sencilla y sirve para nombrar el tipo de servicio. También está **Namespace**, usada para identificar al namespace del tipo del servicio. En caso de que necesitemos un sistema de comunicación de dos vías, podemos hacerlo por medio de callbacks, y para eso contamos con la propiedad **CallbackContract**. Con relación

a la seguridad, niveles de encriptación y firmas digitales, entre otras opciones interesantes, hacemos uso de la propiedad **ProtectionLevel**. Por su parte, **ConfigurationName** es usada cuando necesitamos encontrar el servicio dentro de un archivo de configuración. Por último, **SessionMode** se emplea para indicar si las sesiones van a ser requeridas, permitidas o no en el contrato de servicios.

El contrato de operaciones

Ya que tenemos definido el contrato de servicio, es necesario indicar cuál de los métodos de dicha interfaz se expondrá al exterior. La mayoría de las veces, todos los métodos van a exponerse, pero en algunos casos, es posible que solo deseemos exponer algunos. Por eso, debemos tener el contrato de operaciones, que indica cuál de los métodos se expone al exterior. Para lograr esto, también vamos a hacer uso de un atributo, que en este caso es **[OperationContract]**, con el cual decoramos los métodos correspondientes. Un ejemplo de esto es el siguiente código:

```
[ServiceContract]
public interface IMiServicio
{
    [OperationContract]
    bool MiMetodo1 (int valor);
    [OperationContract]
    double MiMetodo2 (double valor1, double valor2, double valor3);
}
```



EJEMPLOS DE WCF



Si buscamos acceder y ver interesantes ejemplos de aplicaciones desarrolladas con WCF ya construidas y completamente funcionales, porque puede ayudarnos a comprender la forma en que trabaja este tipo de programas, es posible buscarlas en la Red. Algunos ejemplos con diferentes grados de dificultad pueden encontrarse en el sitio web que se ubica en el siguiente enlace: <http://msdn.microsoft.com/es-mx/library/ms751532%28v=VS.90%29.aspx>.

En este caso se han definido dos métodos; desde luego, podemos tener la cantidad que sean necesarios. Los nombres, tipos y cantidad de parámetros también pueden variar.

Al igual que **[ServiceContract]**, **[OperationContract]** posee una serie de propiedades que podemos utilizar. Una de las más importantes es **AsyncPattern**, la cual indica si la implementación del método se lleva a cabo de forma asíncrona. La propiedad **IsOneWay** es utilizada cuando el método solo va a recibir entradas y no debe dar ninguna salida. Si deseamos indicar que alguna operación en particular puede ser la primera en una sesión, optamos por la propiedad **IsInitiating**. De forma similar, podemos utilizar **IsTerminating** para indicar que se debe tratar de finalizar la sesión después de que el método ha llevado a cabo su trabajo, así obtendremos una mayor seguridad.



Creación de aplicaciones WCF

Ya conocemos algunos de los conceptos básicos de WCF, pero ahora tenemos que enfocarnos más en la creación de la aplicación. Cuando creamos una aplicación de WCF, tendremos como mínimo tres assemblies que debemos utilizar. En primer lugar, necesitamos el assembly del servicio. Este, en forma de **dll**, contendrá las interfaces y/o clases que, como tal, se expondrán al exterior. Es la interfaz del servicio que queremos proveer.

Ya que tenemos el assembly de servicio, procedemos a tener el **host** o anfitrión correspondiente. Esta parte de software toma el assembly del

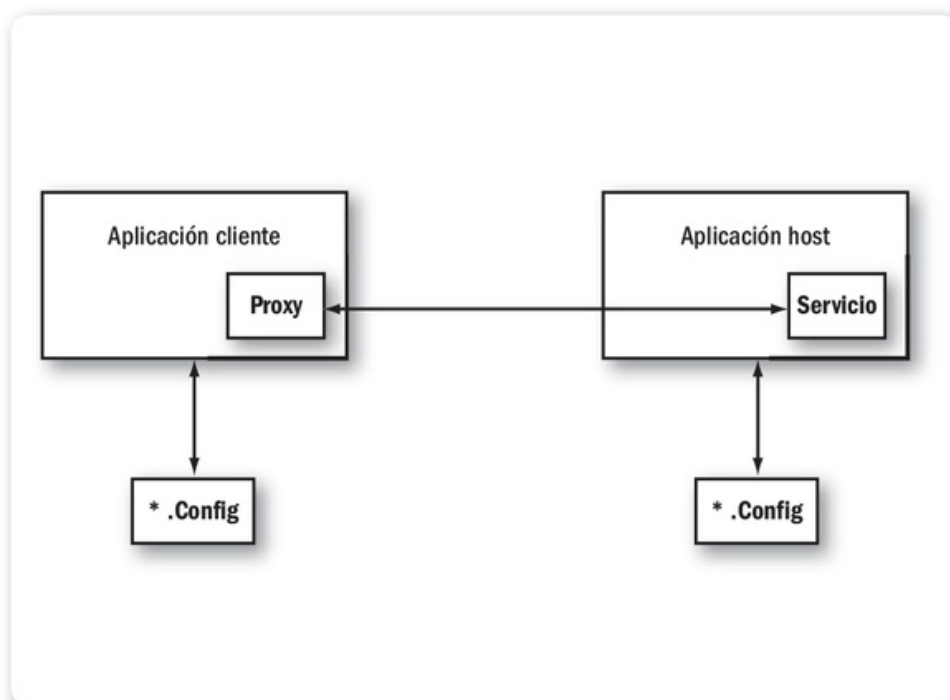


CREAR NUESTRO PROPIO BINDING



Debemos tener en cuenta que WCF ofrece muchos **bindings** que podemos utilizar en el desarrollo de nuestras aplicaciones. Pero en algunas ocasiones, necesitamos nuestro propio estilo de binding. Para esto, tenemos que extender el tipo **CustomBinding** que provee WCF. La creación de un binding es un tema avanzado dentro de la programación con WCF.

servicio y lo mantiene activo en su interior para que el exterior pueda llevar a cabo la conexión. Necesitamos a alguien que desee conectarse, y este software es conocido como el cliente. El cliente se conecta y puede hacer uso de las funcionalidades que ofrece el servicio. Cualquier aplicación que creamos puede actuar como host, y los clientes también pueden ser aplicaciones tradicionales de .NET; solo hay que agregar algunos elementos para lograr las conexiones.



► **Figura 5.** Esta figura muestra la forma como se conectan entre sí los diferentes elementos de nuestro sistema distribuido en WCF.

Tradicionalmente, en WCF tenemos un mnemónico en inglés conocido como **ABC**, que sirve para recordar los componentes que necesita tener la aplicación de WCF. Veamos su significado. En primer lugar, tenemos la letra A, que simboliza **Address**; es decir, dirección. Luego está la B, que denota **Binding**, y puede ser tomado como vínculo. Por último, la letra C es usada para recordar **Contract**; es decir, el contrato.

Usamos la dirección pues es necesario conocer dónde se encuentra el servicio. Si no conocemos su ubicación, no será posible conectarse a él. La dirección puede colocarse en el archivo de configuración *.config,

pero también es posible hacer uso de **System.Uri**. Con relación al binding, tenemos muchas opciones que veremos brevemente un poco más adelante. La parte del contrato ya nos es conocida, y son los métodos expuestos por el servicio. Nos sirve para indicar la forma en que los clientes tendrán acceso a las funcionalidades de los servicios. Al crear WCF, Microsoft ha planeado la parte del binding muy bien y presenta muchas opciones. La mayoría de las veces usaremos alguna de estas opciones, pero incluso si alguna no fuera conveniente para nuestra aplicación, podemos crear nuestro propio binding.

El binding que seleccionemos dependerá de las necesidades del sistema, ya que cada uno de ellos tiene ciertas ventajas que podemos aprovechar dependiendo de nuestros requerimientos. Los bindings poseen determinadas particularidades que debemos tomar en cuenta. Lo primero que encontramos es la capa de transporte, que será utilizada para mover la información que deseamos enviar. Se puede utilizar, por ejemplo, TCP, HTTP o MSMQ. La información puede ser de un solo sentido, de dos vías o **duplex**, o de petición-respuesta, por lo que los canales usados para el transporte también son una característica importante. La información puede ser enviada de diferentes maneras, lo que pone en evidencia la relevancia del mecanismo de codificación. Si el sistema distribuido se ejecuta exclusivamente en Windows con software en .NET, lo mejor es usar una codificación en binario; pero si vamos a utilizar diversos sistemas operativos con clientes escritos en otras plataformas, una codificación en XML es la mejor alternativa. Otra de las características que debemos considerar son los protocolos para servicios web, como **WS-Transactions** o **WS-Security**.

DEBEMOS ELEGIR
EL TIPO DE BINDING
DE ACUERDO A LAS
NECESIDADES DEL
SISTEMA



VERSIONES ANTERIORES



Tengamos en cuenta que si precisamos obtener compatibilidad con software que se ha desarrollado con versiones anteriores de .NET y que se encarga de emplear servicios web ASP.NET, la opción de binding requerida es la llamada **BasicHttpBinding**. Al hacerlo de esta forma, nos ahorraremos problemas, y tiempo de desarrollo y depuración.

PARA CONOCER LOS
BINDINGS LO MEJOR
ES EXPERIMENTAR
CON ELLOS

En WCF hay cuatro tipos de binding que se basan en HTTP; ellos son: **BasicHttpBinding**, **WSHttpBinding**, **WSDualHttpBinding** y **WSFederationHttpBinding**. Todos hacen uso de XML para llevar a cabo la codificación de la información que se envía. Esto es muy útil cuando

los clientes van a usar diferentes sistemas operativos que son desarrollados en diferentes lenguajes de programación.

La clase **BasicHttpBinding** se usa para construir un servicio de WCF de tipo **WS-I BasicProfile 1.1**. Los mensajes quedan codificados como XML, que se puede leer como archivo de texto. El siguiente tipo de binding basado en HTTP es **WSHttpBinding**, muy similar al anterior, pero con más funcionalidades. Podemos hacer uso de características de servicios web como transacciones, direccionamiento

WS y mensajes confiables. Este binding también puede emplear codificación en binario utilizando **MTOM** o *Message Transmission Optimization Mechanism*. Cuando necesitamos tener comunicación duplex, es decir, en ambos sentidos, lo que más nos conviene es utilizar **WSDualHttpBinding**. Es similar al binding anterior, pero ofrece la gran ventaja de que tanto el cliente como el servicio pueden mandar mensajes de dos vías. El tipo de seguridad que usa es SOAP. Por último, tenemos el binding **WSFederationHttpBinding**, que también es seguro pues hace uso del protocolo **WS-Federation**. Si la seguridad de la información es lo más importante, debemos utilizar este tipo de binding. Con **WSFederationHttpBinding** podemos usar **WS-Trust**, **WS-Security** y **WS-SecureConversation**.

También tenemos bindings basados en TCP; en este caso, hay solamente tres tipos diferentes: **NetNamedPipeBinding**,



CREACIÓN DE CLIENTES PARA WCF



Una de las partes más importantes de WCF es la creación de los clientes que van a utilizar el servicio. Algunas veces, esta etapa del proceso puede resultar ligeramente complicada, por lo que es recomendable visitar el siguiente sitio y leer la explicación sobre este tema: <http://msdn.microsoft.com/es-mx/library/ms730825%28v=VS.90%29.aspx>.

NetPeerTcpBinding y **NetTcpBinding**. La mayor ventaja de un binding basado en TCP se da cuando tenemos el sistema distribuido totalmente basado en Windows y haciendo uso del framework de .NET es su versión 4.0. En este caso, existen numerosas mejoras de desempeño, porque toda la información es codificada en formato binario, lo que reduce su tamaño en comparación con XML. La única desventaja es que estamos limitados a que el sistema distribuido se encuentre totalmente basado en la implementación de .NET.

Hay que recordar que es posible tener un sistema distribuido que se ejecute en una sola computadora. Cuando este es el caso, lo mejor es utilizar el binding **NetNamedPipeBinding**, que está optimizado para este tipo de trabajo. Este binding es seguro sin importar que el sistema se ejecute en una sola computadora. Permite tener comunicaciones seguras, y hacer uso de transacciones y sesiones confiables; la desventaja es que no es posible comunicarse con otras computadoras. También podemos tener máquinas conectadas en **P2P**; si este es el caso, el binding por utilizar es el llamado **NetPeerTcpBinding**.

Cuando necesitamos la transferencia entre computadoras que no usen un esquema P2P, la opción de binding TCP es **NetTcpBinding**, el cual es seguro y optimizado para la transferencia de datos. También es posible que accedamos a realizar comunicaciones seguras, transacciones y sesiones confiables.

En caso de que deseemos integrar el sistema distribuido con un servidor de Microsoft MSMQ, podemos utilizar dos opciones de binding: **MsmqIntegrationBinding** o **NetMsmqBinding**. La primera permite

TENGAMOS EN
CUENTA QUE
DIFERENTES BINDINGS
DAN DIFERENTES
DESEMPEÑOS



COMPRENDER P2P



Es importante tener en cuenta que la arquitectura de P2P, también conocida como **Peer to Peer**, distribuye el trabajo entre todos los miembros de la red de forma igual y con los mismos privilegios. Esta interesante y funcional arquitectura no necesita la existencia de un coordinador central o servidor especializado. Torrent hace uso de este tipo de arquitectura.

enviar y recibir mensajes a aplicaciones que usan COM. Recordemos que si precisamos que la comunicación sea al estilo .NET, entonces será necesario que recurramos a la segunda.

Ya conocemos la manera en que puede llevarse a cabo el vínculo entre los clientes y el servicio; sin embargo, para que los clientes puedan conectarse, necesitan conocer dónde se encuentra el servicio. Esto se hace por medio de su dirección, que debe colocarse según los requerimientos de WCF, pero no es muy diferente de la manera como se lleva a cabo en otros sistemas. Una dirección WCF tiene el formato que vemos en el código que se presenta a continuación:

```
Protocolo://<NombreMaquina>[:Puerto]/Ruta
```

En **Protocolo** tenemos que colocar el de transporte que hemos decidido utilizar. **NombreMaquina** indica el dominio de la máquina a la cual deseamos solicitar el servicio. En **Puerto** indicamos el número de puerto que utilizamos para efectuar la transferencia de los datos, por ejemplo, para HTTP es **80**. La colocación del puerto es opcional, pero debemos incluirlo si no estamos trabajando con alguno predefinido para el protocolo seleccionado. Por último, tengamos en cuenta que la **Ruta** indica la correspondiente dentro de la máquina que nos lleva al servicio deseado.

Por ejemplo, si vamos a utilizar HTTP, podemos colocar la dirección de la siguiente forma:

```
http://localhost/servicios/MiServicioHTTP
```

En este caso, no indicamos el puerto porque estamos usando el predefinido. Un ejemplo para vínculos que usen TCP es el siguiente:



TUTORIAL WCF



Es interesante tener en cuenta que un muy buen tutorial sobre WCF, que nos permite conocer, incluso, temas avanzados sobre este tipo de aplicación, se encuentra en el sitio web creado por Saravanakumar. Podemos consultarlo en la dirección **www.wcftutorial.net/Home.aspx**. Se trata de un excelente lugar para seguir aprendiendo sobre WCF.


```
net.tcp://localhost:8080/servicios/MiServicioTCP
```

Tengamos en cuenta que también tenemos algo similar cuando hacemos uso de un vínculo con MSMQ:

```
net.msmq://localhost/private$/MiServicioMSMQ
```

Esta ruta luce un poco diferente de las demás. El primer cambio que observamos es la ausencia de puerto. Esto es natural porque los puertos no tienen sentido en MSMQ. También debemos tomar en cuenta que el queue puede ser público o privado, sin olvidar que los privados solamente pueden ser accedidos desde la misma máquina.

Si hacemos uso de **NetNamedPipeBinding** para llevar a cabo nuestro binding, la dirección puede lucir similar a la siguiente:

```
net.pipe://localhost/servicios/MiServicioNamed
```

Código para un servicio WFC

Ahora podemos ver algo de código que nos sirva de guía para crear una aplicación distribuida utilizando WFC. Con el fin de facilitar el aprendizaje, desarrollaremos una aplicación muy sencilla; en este caso, una calculadora. Las operaciones aritméticas básicas residen en el servicio, y el cliente llevará a cabo las peticiones correspondientes.

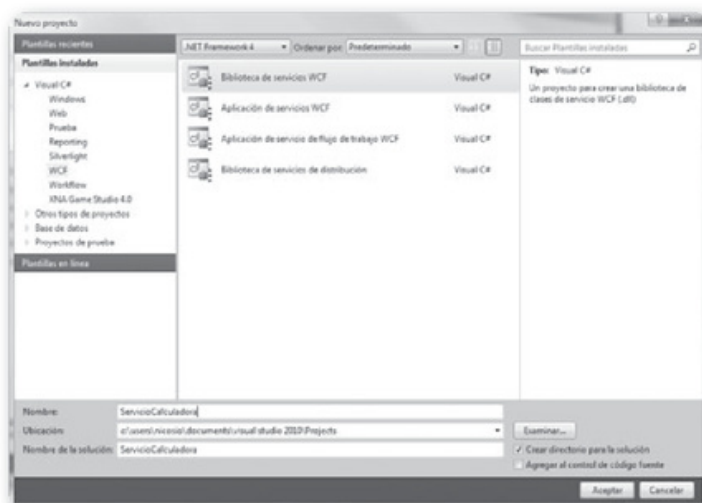
Empezamos por crear un nuevo proyecto en Microsoft Visual Studio. En la sección de las plantillas seleccionamos WCF, y en el tipo de proyecto, marcamos **Biblioteca de servicios WCF**. Como nombre colocamos **ServicioCalculadora** y oprimimos el botón **OK**, tal como podemos observar en la **Figura 6**.



VERSIÓN DE VISUAL STUDIO NECESARIA

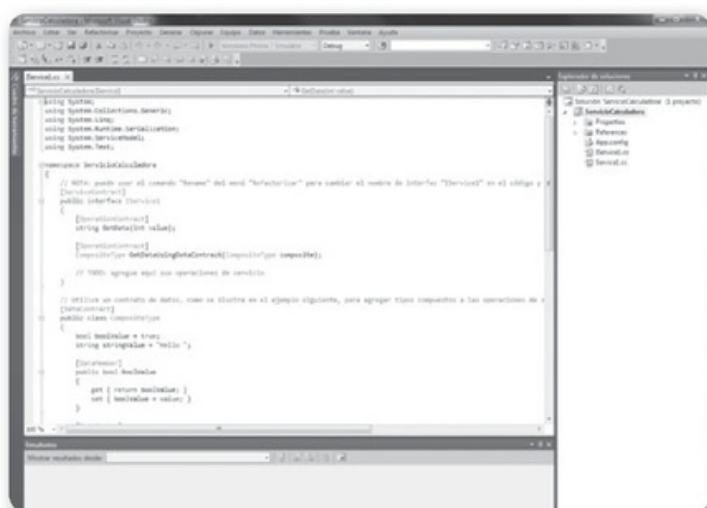


No es posible crear proyectos de WFC en la versión Express de Microsoft Visual Studio, por lo que necesitamos contar con la versión profesional. En ocasiones, es posible obtener esta versión en forma gratuita durante un período de prueba de 30 días. Podemos consultar el sitio de Microsoft Visual Studio en el sitio web oficial de Microsoft, en www.microsoft.com.



► **Figura 6.**
Aquí podemos observar el tipo de proyecto que necesitamos crear para nuestro servicio de WCF.

El proyecto que se ha creado contiene una interfaz y una clase que la implementa. Visual Studio crea para nosotros una plantilla con un servicio simple, pero, en realidad, no necesitamos este código, por lo que lo borramos y reemplazamos por el nuestro propio.



► **Figura 7.** Visual Studio ha creado código en la interfaz para nosotros, pero no lo utilizaremos, sino que colocaremos el nuestro propio.

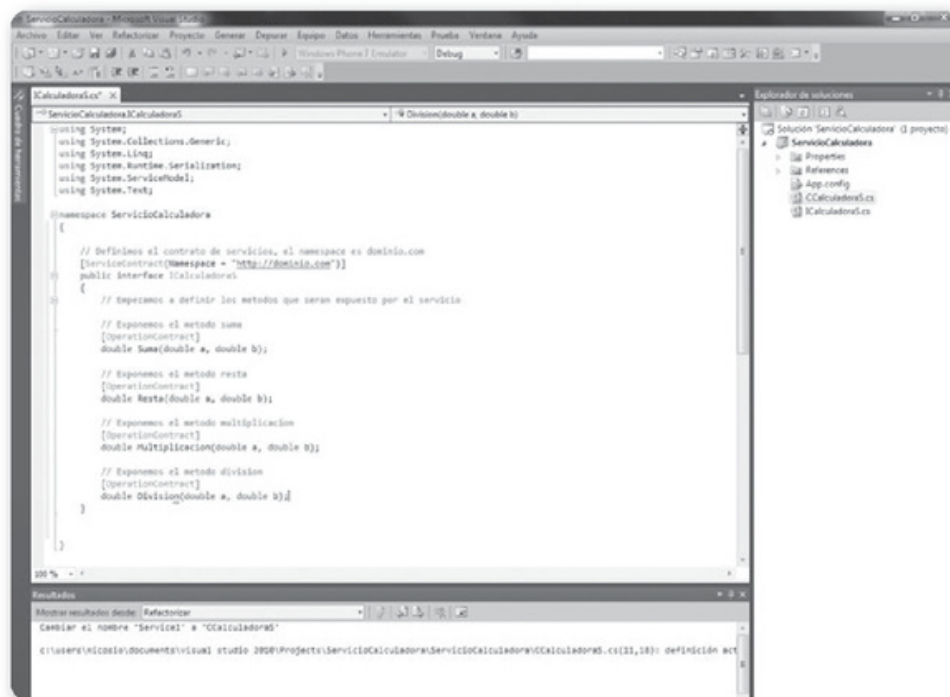
Para continuar, nos daremos a la tarea de modificar el código correspondiente a la interfaz de nuestra aplicación, de esta forma accederemos a realizar la indicación de cuáles son los métodos que expondremos en el servicio adecuado. El código necesario para esta tarea es el que vemos a continuación:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Runtime.Serialization;
using System.ServiceModel;
using System.Text;
namespace ServicioCalculadora
{
    // Definimos el contrato de servicios, el namespace es dominio.com
    [ServiceContract(Namespace = "http://dominio.com")]
    public interface ICalculadoraS
    {
        // Empezamos a definir los metodos que seran expuestos por el servicio
        // Exponemos el metodo suma
        [OperationContract]
        double Suma(double a, double b);
        // Exponemos el metodo resta
        [OperationContract]
        double Resta(double a, double b);
        // Exponemos el metodo multiplicacion
        [OperationContract]
        double Multiplicacion(double a, double b);
        // Exponemos el metodo division
        [OperationContract]
        double Division(double a, double b);
    }
}
```

La interfaz es muy sencilla. Vamos a decorarla con el atributo **[ServiceContract]** para indicar que será nuestro contrato de servicios. Hacemos uso de la propiedad **Namespace** para controlar el nombre del namespace XML que define el tipo de servicio. El nombre que usamos para nuestra interfaz es **ICalculadoraS**. Dentro de ella definimos los métodos que se encargan de caracterizar el comportamiento y que serán expuestos por el servicio. Debemos tener en cuenta que cada uno de ellos se define de la manera usual, pero son decorados por el atributo **[OperationContract]**, para indicar que es una operación dentro del contrato, es decir, que se va a exponer al exterior para que el cliente

pueda invocarla. El nombre del documento que contiene la interfaz debe cambiarse a la llamada **ICalculadoraS.cs**.

Ahora podemos ver la clase que contiene la implementación de la interfaz. En este caso, también encontramos que existe código creado por Visual Studio. Este código no nos sirve, por lo que lo eliminaremos y pondremos el nuestro.



► **Figura 8.** En este caso podemos ver que nuestro código fue colocado en vez del propuesto por Visual Studio.

El código para colocar en la implementación es el siguiente:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Runtime.Serialization;
using System.ServiceModel;
using System.Text;
namespace ServicioCalculadora
```



```
{
    // La clase CCalculadoraS debe implementar a la interfaz ICalculadoraS
    public class CCalculadoraS : ICalculadoraS
    {
        // Implementacion del metodo suma
        public double Suma(double a, double b)
        {
            // Variable para el resultado
            double r;
            // Calculamos el resultado
            r = a + b;
            // Mandamos al cliente el valor calculado
            return r;
        }
        // Implementacion del metodo resta
        public double Resta(double a, double b)
        {
            // Variable para el resultado
            double r;
            // Calculamos el resultado
            r = a - b;
            // Mandamos al cliente el valor calculado
            return r;
        }
        // Implementacion del metodo multiplicacion
        public double Multiplicacion(double a, double b)
        {
            // Variable para el resultado
            double r;
            // Calculamos el resultado
            r = a * b;
            // Mandamos al cliente el valor calculado
            return r;
        }
        // Implementacion del metodo suma
        public double Division(double a, double b)
        {
```

```
// Variable para el resultado
double r;
// Calculamos el resultado
r = a / b;
// Mandamos al cliente el valor calculado
return r;
}
}
}
```

El código es muy sencillo. En primer lugar, indicamos que la clase implementa la interfaz **ICalculadoraS** y que el nombre de la clase es **CCalculadoraS**. El resto del código es una implementación usando código tradicional en C#. La implementación de los métodos es muy simple y fácil de entender, por lo que no vamos a comentarla.

En el proyecto también se ha creado un archivo **App.config**, con el código que vemos a continuación, en su interior:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.web>
    <compilation debug="true" />
  </system.web>
  <!-- Al implementar el proyecto de la biblioteca de servicios, el contenido del
  archivo de configuración se debe agregar al archivo
  app.config del host. La configuración del sistema no admite archivos de
  configuración en las bibliotecas. -->
  <system.serviceModel>
    <services>
      <service name="ServicioCalculadora.CCalculadoraS">
        <endpoint address="" binding="wsHttpBinding"
        contract="ServicioCalculadora.IService1">
          <identity>
            <dns value="localhost" />
          </identity>
        </endpoint>
        <endpoint address="mex" binding="mexHttpBinding"
        contract="IMetadataExchange" />
      </service>
    </services>
  </system.serviceModel>
</configuration>
```

```
<host>
  <baseAddresses>
    <add baseAddress="http://localhost:8732/Design_Time_Addresses/
      ServicioCalculadora/CCalculadoraS/" />
  </baseAddresses>
</host>
</service>
</services>
<behaviors>
  <serviceBehaviors>
    <behavior>
      <!-- Para evitar revelar información de los metadatos,
        establezca el valor siguiente en false y quite el extremo superior de los
        metadatos antes de la implementación -->
      <serviceMetadata httpGetEnabled="True"/>
      <!-- Para recibir detalles de las excepciones en los fallos, con el fin de
        poder realizar la depuración,
        establezca el valor siguiente en true. Para no revelar información sobre las
        excepciones
        establézcalo en false antes de la implementación -->
      <serviceDebug includeExceptionDetailInFaults="False" />
    </behavior>
  </serviceBehaviors>
</behaviors>
</system.serviceModel>
</configuration>
```

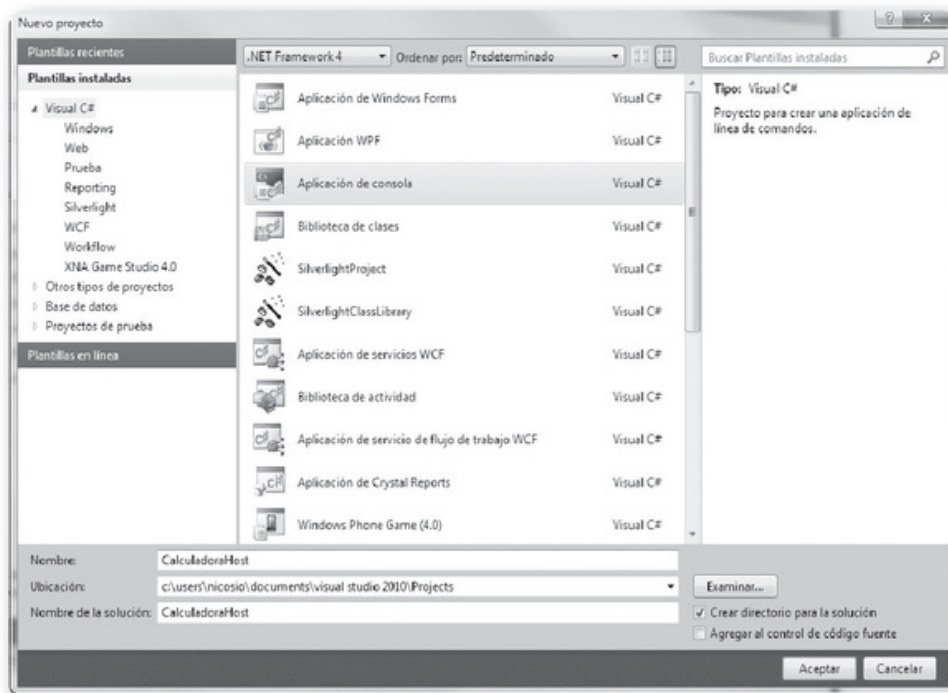


LOS NOMBRES DE COM+



Es interesante saber que Microsoft empezó a darse cuenta de que hacía mucha falta tener una API con más características que DCOM para proceder a desarrollar eficientemente las aplicaciones distribuidas. Para lograr esta tarea, se encargó de desarrollar **MTS**, que significa **Microsoft Transaction Server**. Posteriormente, MTS cambió de nombre a COM+. En este sentido debemos tener en cuenta que existe la posibilidad de que el cambio de nombre tuviera que ver con razones de posicionamiento y que diera la idea de evolución sobre la versión anterior.

A continuación, debemos crear el host para el servicio. Para hacerlo, generamos una aplicación de consola. Para el nombre de proyecto escribimos **CalculadoraHost** y, luego, oprimimos el botón **OK**. Visual Studio crea el proyecto para nosotros.



► **Figura 9.** Para el host tenemos que utilizar un proyecto de consola. El host no requiere de código complicado.

Ahora adicionamos dos referencias necesarias para el correcto funcionamiento de la aplicación. La primera es **System.ServiceModel**. Para agregarla en el explorador de soluciones, buscamos en el

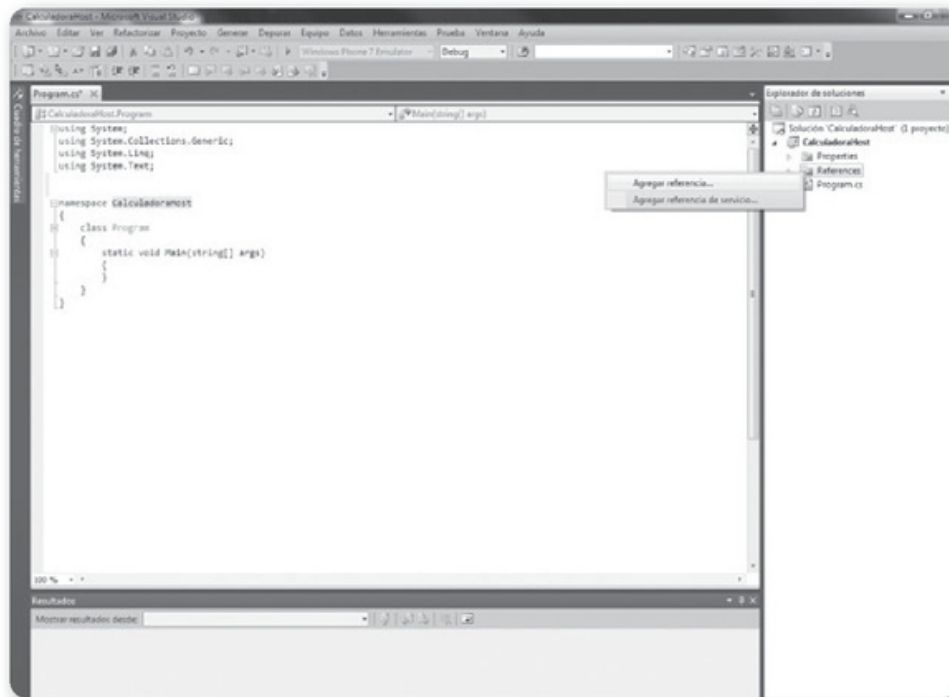


CREACIÓN DE UNA APLICACIÓN WCF



Una guía sencilla y rápida, paso a paso, para la creación de una aplicación WCF puede consultarse en el sitio web de The Code Project. Es bueno consultarla si tenemos problemas en los procedimientos que debemos llevar a cabo en un desarrollo de este tipo. La dirección web correspondiente es la que vemos a continuación: www.codeproject.com/KB/WCF/WCFPart1.aspx.

interior de nuestra solución la carpeta marcada con el nombre **References** y, con un clic del botón derecho, seleccionamos **Agregar referencia** del menú contextual que aparece.



► **Figura 10.** En esta imagen vemos el menú que debemos utilizar para llevar a cabo la adición de una referencia.

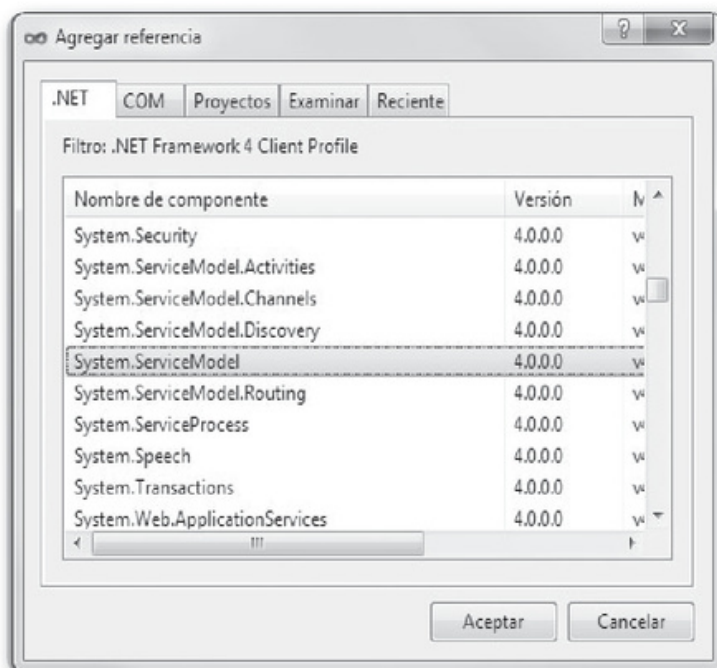
Tengamos en cuenta que se abre un cuadro de diálogo dentro del cual debemos proceder a seleccionar la pestaña de **.NET** y, posteriormente, buscamos en su interior el componente **System.ServiceModel**; lo seleccionamos y oprimimos el botón **Aceptar**.



PRÁCTICA PARA EL DESARROLLO



Es necesario tener en cuenta la importancia de practicar mucho el desarrollo de las aplicaciones distribuidas. Recordemos que no necesariamente deben de ser programas complicados, ya que tan solo puede ser algo sencillo como una calculadora, en este sentido lo importante es experimentar y probar cada una de las tecnologías que tenemos a nuestra disposición.



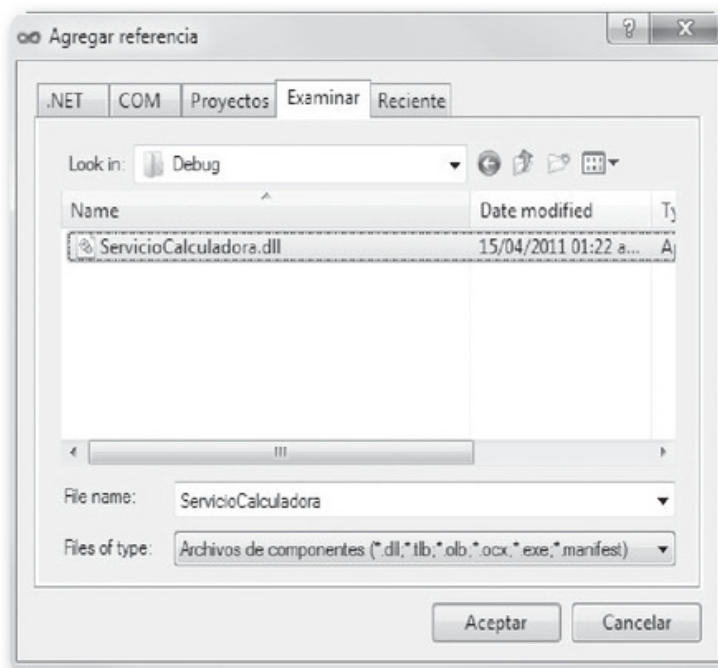
► **Figura 11.** En esta ventana es necesario seleccionar el componente llamado **System.ServiceModel** para adicionar la referencia necesaria.

El siguiente paso es agregar la referencia al **dll** creado en el servicio. De manera similar, seguimos los pasos para adicionar una referencia, pero en esta ocasión, cuando aparece el cuadro de diálogo, seleccionamos la pestaña **Examinar** y buscamos la carpeta del proyecto del servicio. En su interior localizamos la compilación del **dll** correspondiente; debe llamarse **ServicioCalculadora.dll**.



NO HAY QUE OLVIDAR LAS REFERENCIAS

Debemos tener en cuenta que uno de los errores que suele suceder cuando empezamos a trabajar con WCF es el olvidar colocar las referencias que necesitamos. Por esta razón debemos tener cuidado cuando estemos desarrollando nuestras aplicaciones, en este sentido, si al crear nuestro programa vemos que el entorno de desarrollo Microsoft Visual Studio no realiza el reconocimiento de namespaces o clases que necesitamos usar, lo más seguro es que falte adicionar la referencia. En este será necesario proceder a corregir el error antes mencionado.



► **Figura 12.** Para usar el servicio debemos adicionar **ServicioCalculadora.dll**, el que nos encargamos de construir en el programa anterior.

Ya con las referencias, podemos adicionar el código para esta aplicación. Lo que debemos colocar es lo siguiente:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
// Necesitamos utilizar estos namespace para usar el servicio de la calculadora
using System.ServiceModel;
using ServicioCalculadora;
namespace CalculadoraHost
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Inicializamos el host de la calculadora");
        }
    }
}
```

```
// Creamos el service host
using (ServiceHost miSH = new ServiceHost(typeof(CCalculadoraS)))
{
    // Abrimos el servicio
    miSH.Open();
    // Mostramos la informacion del host
    Console.WriteLine("Informacion del host");
    foreach (System.ServiceModel.Description.ServiceEndpoint endpoint
        in miSH.Description.Endpoints)
    {
        Console.WriteLine("Direccion: {0}", endpoint.Address);
        Console.WriteLine("Vinculo: {0}", endpoint.Binding.Name);
        Console.WriteLine("Contrato: {0}", endpoint.Contract.Name);
        Console.WriteLine();
    }
    Console.WriteLine();
    // Quedamos en espera de que el usuario termine el servicio
    Console.WriteLine("Oprima cualquier tecla para terminar
        el servicio");
    Console.ReadLine();
}
}
```

El código es muy sencillo. Procedemos a crear un **ServiceHost** para el servicio de la calculadora. Por medio del método **Open()**, lo abrimos. Si lo deseamos, podemos colocar el código que muestra la información del **ServiceHost**; para hacerlo, recorreremos todos los **EndPoint** que



WCF EN TODO TIPO DE APLICACIONES

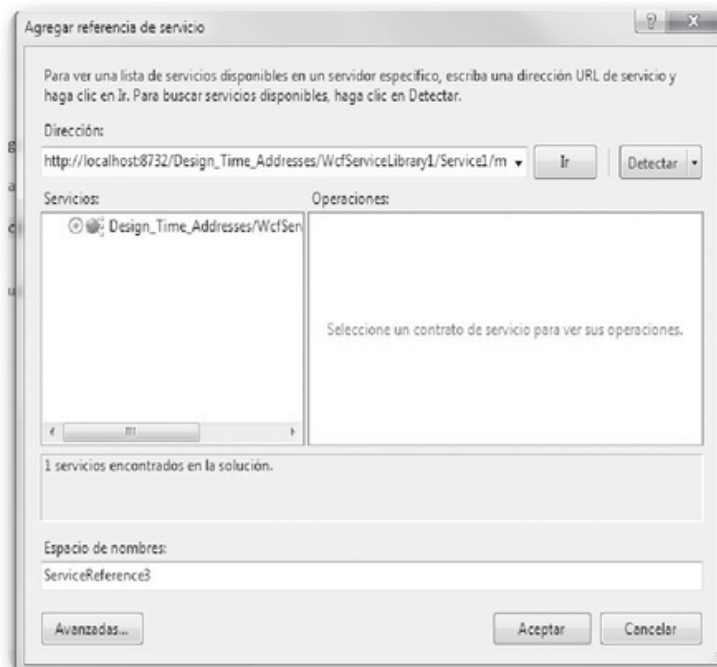


Recordemos que en este capítulo nos encargamos de utilizar WCF para una aplicación de consola, sin embargo, es necesario que tengamos en cuenta que es posible hacer uso de WCF en todo tipo de aplicaciones. Es posible crear aplicaciones distribuidas con interfaces gráficas de usuario, ya sean creadas de la forma tradicional, o por medio de WPF y XAML.

tiene y desplegamos la información de cada uno de ellos. El servicio permanecerá abierto hasta que la aplicación finalice. Para evitar una finalización temprana, colocamos un **Console.ReadLine()**, porque, de esta forma, el usuario tendrá que oprimir una tecla para terminar el programa.

En este momento ya podemos crear el cliente. Primero compilamos y ejecutamos el host, que se presentará como un servicio. Es posible que Windows muestre una ventana de alerta indicando que el servicio está corriendo, recordemos que esto es natural.

Podemos crear entonces un nuevo proyecto de tipo aplicación de consola sin cerrar la ejecución del servicio. En el proyecto es necesario crear un **proxy**. Afortunadamente, Visual Studio cuenta con las herramientas para hacerlo. Localizamos en el explorador de soluciones la carpeta **Service Refences** y, con el botón derecho del mouse, la agregamos. Se abre un cuadro de diálogo que permite añadir el servicio adecuado. En él presionamos **Detectar**, para que aparezca nuestro servicio en la lista del lado izquierdo. En la lista del lado derecho veremos los métodos que expone. Luego, oprimimos el botón **Aceptar**.



► **Figura 13.** El cuadro de diálogo que se presenta nos permite adicionar el servicio correspondiente; no olvidemos oprimir el botón llamado **Detectar**.

El código que adicionamos para el cliente es el siguiente:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
// Necesario para el proxy
using ServicioCalculadora.ServiceReference;
namespace CalculadoraClient
{
    class Program
    {
        static void Main(string[] args)
        {
            // Datos
            double r;
            Console.WriteLine("Usamos la calculadora");
            using (CCalculadoraClient cal = new CCalculadoraClient
                ("BasicHttpBinding_ICalculadoraS"))
            {
                // Invocamos los metodos de la calculadora
                r = cal.Suma(5, 6);
                Console.WriteLine("La suma es {0}", r);
                r = cal.Resta(5, 6);
                Console.WriteLine("La resta es {0}", r);
                r = cal.Multiplicacion(5, 6);
                Console.WriteLine("La multiplicacion es {0}", r);
            }
        }
    }
}
```



COMPARAR WCF CON OTRAS TECNOLOGÍAS



Tengamos en cuenta que WCF se encarga de reemplazar a varias de las tecnologías que en el pasado se usaban para poder crear sistemas distribuidos para sistemas operativos Windows. En este sentido una práctica interesante para comprender más los sistemas distribuidos puede ser intentar crear el ejercicio que fue presentado en este capítulo con otra de las tecnologías que tenemos a nuestra disposición. De esta forma podremos acceder a comprender de mejor forma los sistemas distribuidos y por otro lado todas las ventajas que nos ofrece el uso de WCF.

```
        r = cal.Division(5, 6);  
        Console.WriteLine("La division es {0}", r);  
    }  
}  
}
```

Tengamos en cuenta que el código que observamos es muy sencillo: simplemente, instanciamos el cliente y llevamos a cabo las llamadas a los métodos expuestos. Con esto tenemos las bases para crear aplicaciones distribuidas. Aún queda mucho material de WCF por cubrir; esto es solo el inicio.

Hasta aquí hemos visto las bases de un sistema distribuido y la forma en que podemos colocar código especializado que se ejecute en otro proceso o computadora y de esta forma hacer uso del mismo en un cliente. El ejemplo ha sido sencillo, pero se muestran todos los conceptos básicos. WCF aún tiene mucho material, por lo que se recomienda visitar los recursos recomendados en este capítulo y llevar a cabo los ejercicios descritos.



RESUMEN



WCF forma parte del framework de .NET y tiene como objetivo facilitar la creación de aplicaciones distribuidas. En el pasado, era complicado desarrollar este tipo de programas, pero WCF unifica varias API y simplifica considerablemente el desarrollo. El modelo de comunicación se basa en SOA, y el servicio se expone por medio de endpoints. Los componentes más importantes son el contrato de servicio, el contrato de operaciones, el contrato de datos y los datos miembro. WCF ofrece diferentes bindings para el envío de la información, lo que nos da gran flexibilidad con relación al tipo de sistema distribuido que vamos a generar.

Actividades

TEST DE AUTOEVALUACIÓN

- 1 ¿De qué forma WCF facilita la creación de sistemas distribuidos?
- 2 ¿Qué tecnologías engloba WCF?
- 3 ¿Qué es un endpoint?
- 4 ¿Qué es un binding?
- 5 ¿Cuáles son los bindings utilizados en WCF?
- 6 ¿En qué consiste el contrato de servicios?
- 7 ¿En qué consiste el contrato de operaciones?
- 8 ¿Qué simboliza ABC en WCF?
- 9 ¿Es posible crear un binding propio?
- 10 ¿Qué binding hay que utilizar cuando el sistema se ejecuta en solo una computadora?
- 11 ¿Cómo se crean las direcciones para cada tipo de binding?
- 12 ¿Cómo creamos una biblioteca de servicios WCF?

EJERCICIOS PRÁCTICOS

- 1 Investigue la creación del proxy para la aplicación WCF.
- 2 Haga la aplicación de la calculadora para cada tipo de binding.
- 3 Realice un pequeño software de chat usando WCF.
- 4 Desarrolle un software que transfiera un archivo usando WCF.
- 5 Haga un sistema que tenga varios endpoints.



Los assemblies

Los assemblies son componentes muy importantes de las aplicaciones de .NET. Todas las aplicaciones que hacemos se construyen al colocar assemblies, que, al trabajar juntos, nos permiten obtener el resultado. Muchos de estos assemblies son provistos por el mismo framework de .NET, y nosotros, sin saberlo, creamos otros cada vez que programamos una aplicación, en consola o con interfaz gráfica.



▼ Entender los assemblies300
Entender los namespaces..... 302

▼ Creación y uso
de un assembly.....303



Entender los assemblies

Comprender los assemblies no es complicado: podemos verlos, simplemente, como archivos binarios que son utilizados por el **CLR**, y que tienen información que los describe. Los assemblies ofrecen ventajas sobre otros tipos de archivos binarios, como los que provienen de **COM**. Pueden llevar varios tipos de extensiones de archivo, como ***.EXE** y ***.DLL**.

El framework de .NET contiene numerosos assemblies en forma de ***.DLL**, y con nuestras aplicaciones hacemos uso de ellos. Por ejemplo, **mscorlib.dll** es referenciado de forma automática en las aplicaciones y, si hacemos uso de una que tenga interfaz gráfica de usuario, estaremos utilizando **System.Windows.Forms.dll**. Prácticamente cada vez que usamos un **using** estamos incorporando un assembly en nuestra aplicación.

Una de las grandes ventajas que nos otorga el uso de assemblies es la reutilización de código. Y no solo se trata de reutilizar el código desarrollado por Microsoft en el framework de .NET: también tenemos la ventaja de poder llevar a cabo la reutilización de nuestro propio código. Imaginemos que hemos escrito determinada clase que realiza algo útil para nosotros. Luego, deseamos desarrollar otra aplicación que hace uso de ella. Si no sabemos emplear los assemblies, lo más seguro es, simplemente, copiar y pegar el código. Esto puede parecer una solución sencilla, pero genera ciertos problemas. El primero con el que nos enfrentamos es el mantenimiento de nuestro código. Si la clase que usamos sufre cambios o actualizaciones, tendremos que llevar a cabo los cambios en dos lugares o más si es que la usamos en diversos proyectos.

La mejor opción es hacer un uso correcto de los assemblies y crear uno de ellos para nuestra clase. Esto nos recuerda la forma en que, en otros lenguajes, se programan bibliotecas. Al quedar la clase en su propio assembly, podemos utilizarla en cualquier proyecto que



EL CLR



Entender el **CLR** es un tema básico en la programación de C#. Por eso, es conveniente recordar aquí que se trata de un **runtime** que permite la compilación de nuestra aplicación al **CIL**. Este runtime provee de toda la infraestructura necesaria para ejecutar los programas .NET. Cada vez que ejecutamos un programa de C# estamos haciendo uso del CLR.

necesitemos sin tener que copiar y pegar; únicamente deberemos referenciarla. También el código tendrá que actualizarse en un solo lugar, lo que facilitará su mantenimiento. Esta es una de las maneras más comunes en las que se aplican los assemblies.

No debemos olvidar que los assemblies también tienen versiones, por lo que, en nuestra aplicación, podemos indicar la que necesitamos. Esto es bueno pues podemos tener diferentes versiones del mismo assembly en la máquina y no habrá conflictos entre ellas.

Otra ventaja que nos brindan los assemblies es que actúan como contenedores para los tipos definidos por el programador. Si tenemos dos clases que se llaman igual en el mismo proyecto, entonces tendremos problemas al compilarlo. En cambio, si usamos los assemblies de la forma en que corresponde, y cada una de estas clases pertenece a un assembly, entonces será posible usar ambas sin inconveniente, ya que el assembly provee el contexto en el cual cada clase existe al usar el nombre calificado.

Dentro de los assemblies encontramos gran cantidad de información. Un assembly puede necesitar uno o varios assemblies externos para poder funcionar, y este tipo de información se coloca en su interior. Esta información es localizada en lo que conocemos como el manifiesto del assembly. Y también encontramos información sobre todas las clases y las interfaces junto con todo lo necesario para poder trabajar con ellas.

Cuando trabajamos con assemblies, podemos liberarlos de dos maneras diferentes: hacer assemblies privados o compartidos. Cuando creamos uno de tipo privado, este permanecerá en el mismo directorio que nuestra aplicación. Si creamos un dll, este aparecerá al lado de nuestro **EXE**. También es posible, con un poco de configuración, colocar los dll en un subdirectorio. Este tipo es útil cuando el assembly solo va a ser usado por esa aplicación en particular.



SISTEMA DE VERSIÓN PARA LOS ASSEMBLIES



Tengamos en cuenta que en forma predeterminada, cuando creamos un assembly, Visual Studio le coloca la versión 1.0.0.0, pero es posible modificar este valor. Cada uno de los números significa algo, y la manera en que los ve .NET es la siguiente: **<mayor>.<menor>.<construcción>.<revisión>**. Esto ayuda a mantener el orden y a identificar las distintas versiones.

Hay ocasiones en las cuales el assembly puede ser usado por muchas aplicaciones diferentes que residen en distintos directorios dentro del disco duro; en este caso, nos conviene más hacer uso de los assemblies compartidos, que debemos colocar en un directorio especial conocido como **GAC** o *Global Assembly Cache*, por su siglas en inglés.

Entender los namespaces

Cuando creamos assemblies, también hacemos uso de **namespaces**, que nos permiten agrupar de manera lógica los tipos que definimos en su interior. Es posible definir nuestros propios namespaces (de hecho, lo hacemos con frecuencia), aunque algunos ya existen en el framework de .NET, como **System**.

Cuando vamos a hacer uso de un assembly, necesitamos importar el namespace en el cual fue construido, con el fin de que el compilador reconozca los tipos definidos en él. Para definir un assembly, simplemente hacemos uso del namespace seguido por el nombre de este y, luego, con un bloque de código encerramos todo lo que forma parte de ese namespace. Por ejemplo:

```
namespace MiNamespace
{
    // Aquí todos los elementos que componen al namespace
}
```



```
// Importamos el namespace
using MiNamespace;
namespace OtroNamespace
{
    public class MiClase
    {
        // Aqui ya podemos hacer uso de los tipos definidos en
        MiNamespace
    }
}
```

Todas las aplicaciones que creamos con Visual Studio tienen un namespace, ya que la herramienta lo crea automáticamente para nosotros: se lo conoce como **default namespace**. El nombre de este namespace es igual al del proyecto que estamos creando.

Creación y uso de un assembly

Ya conocemos algunas bases teóricas de los assemblies; ahora podemos crear un proyecto donde aprovechar esos conocimientos. Empezaremos por crear un assembly de un solo archivo y, luego, generaremos una aplicación que lo consuma. Básicamente, será como crear una librería en dll y, tener un programa que la utilice. El código será sencillo, pues nos interesa más el concepto. El assembly contendrá una clase, que tiene como métodos operaciones aritméticas.

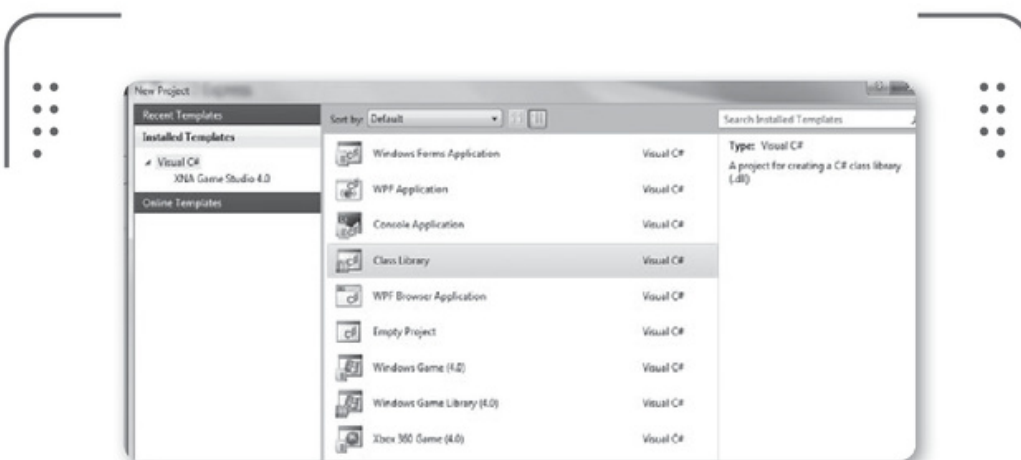


NOMBRES ÚNICOS PARA LOS NAMESPACE



Empezamos por crear un nuevo proyecto en Visual Studio, para lo cual seleccionamos el menú **Archivo**, seguido de **Nuevo Proyecto**.

Visual Studio nos muestra una ventana en la cual podemos observar los diferentes tipos de proyectos que es posible crear. En nuestro caso, el assembly va a actuar como una biblioteca, por lo que debemos seleccionar **Biblioteca de Clases**. En la parte inferior de esta ventana, encontramos un cuadro de texto donde colocamos el nombre del proyecto; en este ejemplo, usaremos **MisMates**. Luego, oprimimos **OK**.



► **Figura 1.** Es necesario seleccionar **Biblioteca de Clases** como tipo de proyecto.

Visual Studio crea para nosotros el esqueleto de la aplicación. Si observamos el código generado, tendremos un namespace llamado **MisMates**, que es el default namespace. En este momento llevamos a cabo unas pequeñas modificaciones. La primera consiste en cambiar el nombre del documento a **Aritmetica.cs**. Luego, debemos acceder a modificar también el nombre de la clase a **Aritmetica**.



EL NOMBRE DEL DEFAULT NAMESPACE



En la mayoría de los casos, este nombre puede sernos útil, pero a veces, es necesario usar otro para el namespace por alguna razón específica. Cuando esto sucede, podemos cambiar el nombre del default namespace a través de la ventana de propiedades del proyecto. En ella encontramos un apartado donde podemos indicar el nombre por utilizar.



► **Figura 2.** De esta forma debe quedar nuestro proyecto, con la clase **Aritmetica** lista para recibir código.

Aunque no estamos creando una aplicación ejecutable, sí estamos generando un assembly. Luego, este trabajará junto con otros para tener una aplicación totalmente funcional. Los *.dll también son assemblies, al igual que las aplicaciones que hemos creado en el pasado.

El código que colocamos para nuestro assembly es el siguiente:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
// Este es el namespace que estamos utilizando
namespace MisMates
{
    // La clase queda definida en el namespace
    // puede ser usada por cualquiera que adicione el assembly
    public class Aritmetica
    {
        // Metodo para la suma
        public double Suma(double a, double b)
        {
            double r;
            r = a + b;
            return r;
        }
    }
}
```

```
// Metodo para la resta
public double Resta(double a, double b)
{
    double r;
    r = a - b;
    return r;
}

// Metodo para la multiplicacion
public double Multiplicacion(double a, double b)
{
    double r;
    r = a * b;
    return r;
}

// Metodo para la division
// Cuidado no se esta colocando ningun codigo de seguridad apropiado
// es solo un ejemplo
public double Division(double a, double b)
{
    double r=0;
    if(b!=0)
        r = a + b;
    return r;
}

} // Fin de la clase
} //Cierre del namespace
```

El código es muy sencillo, pero aun así, vale la pena describirlo. En primer lugar, encontramos la definición del namespace, que se realiza usando **namespace** y colocando el nombre **MisMates**. Esto es importante pues será el namespace que usaremos para el assembly cuando otra aplicación haga uso de él.

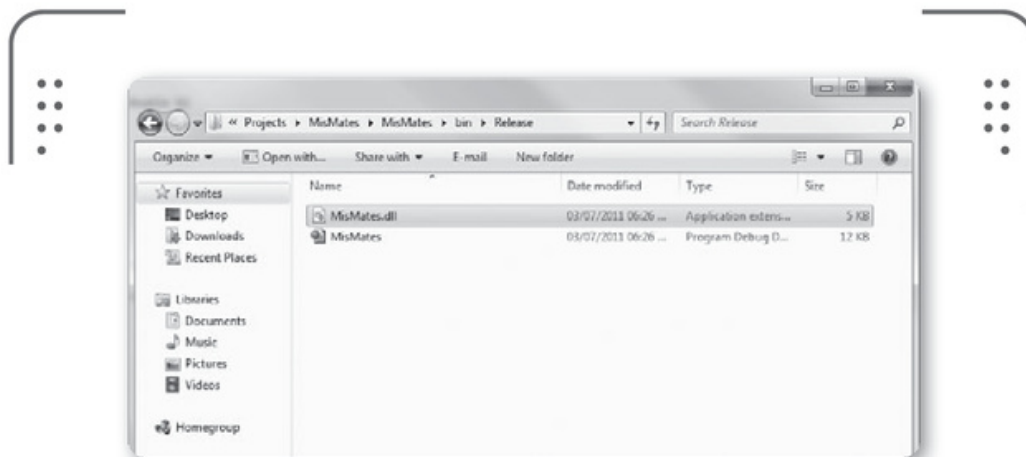
Para continuar, debemos crear un bloque de código en cuyo interior colocamos la clase que llevará a cabo el trabajo. Cualquier aplicación que haga uso del assembly podrá emplear esta clase y los métodos que ella contiene. Nuestra clase se llama **Aritmética**.

Después procedemos a colocar los métodos. Tenemos cuatro que realizarán las operaciones de suma, resta, multiplicación y división. El

código de cada uno de ellos es muy sencillo, por lo que no es necesario describirlo. Al final, no debemos olvidarnos de cerrar correctamente el bloque de código de la clase y del namespace.

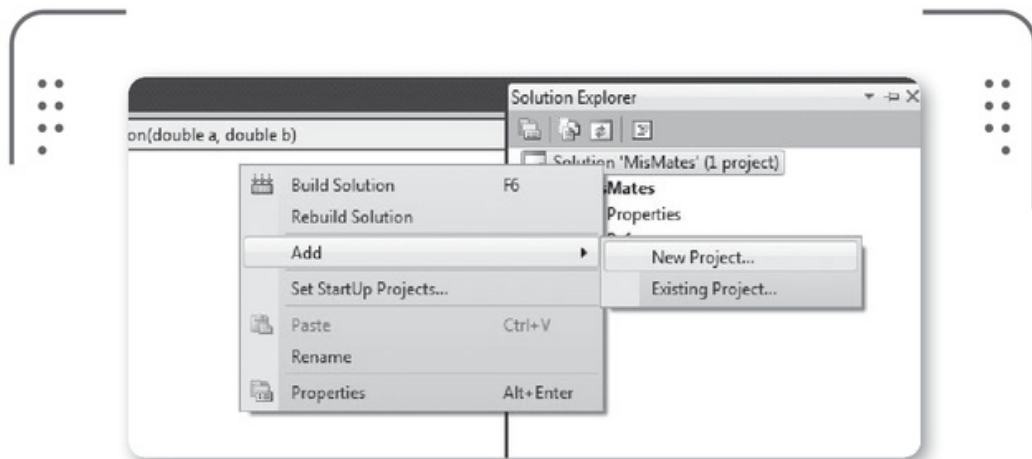
Ahora podemos pasar a construir la aplicación, pero no hay que hacerlo por medio del botón **ejecutar**. Lo mejor en este caso, ya que estamos construyendo una biblioteca de clases, es hacerlo desde el menú **Depuración** y, luego, **Construir Aplicación**. Todo tiene que salir bien, y la biblioteca debe compilar sin problema.

Cuando compilamos una aplicación, obtenemos un archivo ejecutable, pero en el caso de una biblioteca de clases, lo que obtenemos es un dll, que se encontrará en un directorio dentro del directorio propio del proyecto. En el directorio del proyecto hallaremos un directorio llamado **bin** y, dentro de él, dependiendo del tipo de construcción que llevemos a cabo, dos posibles directorios: uno llamado **Release** y el otro llamado **Debug**. Si entramos en ellos, veremos el dll de nuestro assembly.



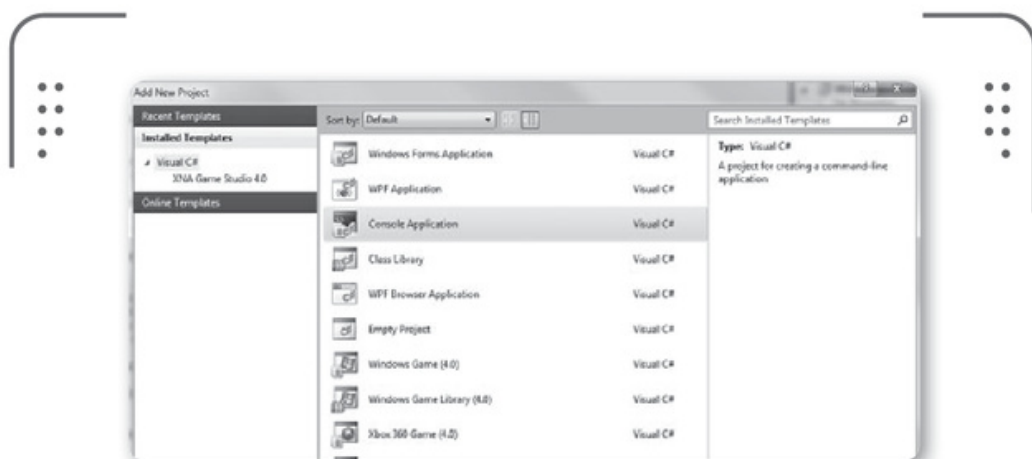
► **Figura 3.** En el directorio de **Release** encontramos el dll que corresponde al assembly de nuestra biblioteca de clases.

Ahora, lo que necesitamos hacer es crear una aplicación, y por lo tanto, otro assembly, que haga uso del correspondiente a la biblioteca de clases que acabamos de hacer. Utilizamos la solución que ya tenemos, y solo adicionamos un proyecto nuevo. Seleccionamos la solución en la ventana del explorador de soluciones y, haciendo clic con el botón derecho del mouse, veremos que aparece un menú dentro del cual elegimos **Adicionar/Nuevo proyecto**. De este modo, se agrega un proyecto a la solución en vez de crear una solución nueva con su proyecto.



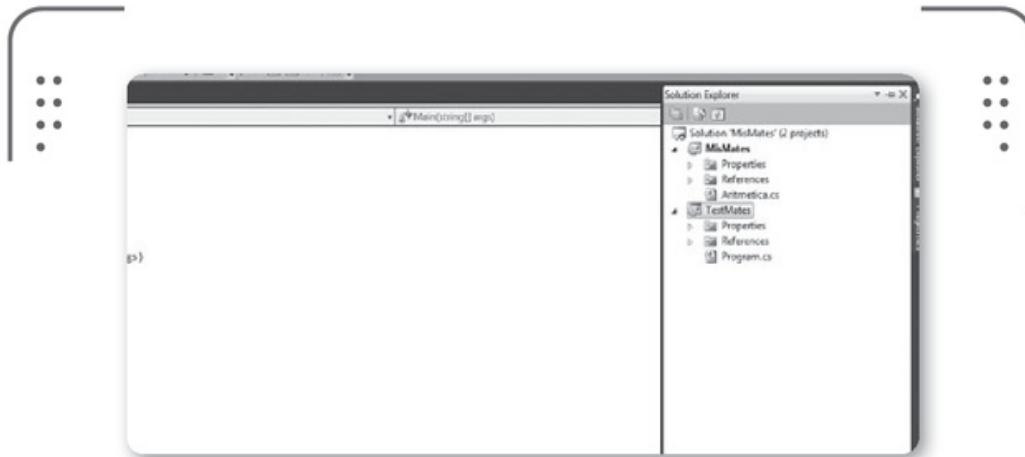
► **Figura 4.** Podemos adicionar más de un proyecto a cada solución por medio del Explorador de soluciones.

Aparecerá una ventana que nos permite seleccionar el tipo de proyecto que deseamos agregar a la solución. En nuestro caso, deseamos crear una aplicación que pueda hacer uso del assembly de la biblioteca de clases. Para esto, podríamos usar varias opciones, como una aplicación de Windows Forms, una de consola e, incluso, una WPF. Para mantener las cosas sencillas, seleccionamos la aplicación de consola y, como nombre para este proyecto, escribimos **TestMates**. Luego de realizar lo que hemos expuesto, simplemente oprimimos el botón **OK** o **Aceptar** en la ventana.



► **Figura 5.** Debemos seleccionar un proyecto de tipo aplicación de consola, aunque otras opciones también pueden funcionar.

Debemos tener en cuenta que Visual Studio crea para nosotros el esqueleto de la aplicación. En la ventana del Explorador de soluciones podemos observar que ahora tenemos dos proyectos. Además del assembly de **MisMates**, también aparece **TestMates**. Ya tenemos dos assemblies, uno de los cuales utilizará al otro. Necesitamos hacer algunos cambios en el namespace **TestMates**.



► **Figura 6.** Ya tenemos dos proyectos en nuestra solución; ahora es necesario colocar el código de la aplicación.

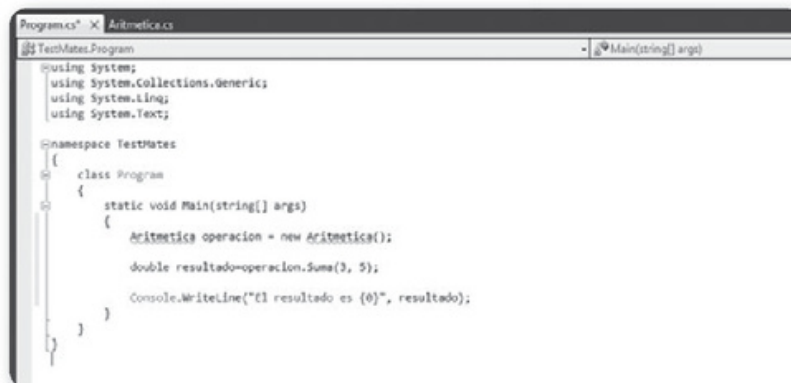
Dentro del proyecto que corresponde a la aplicación de la consola, incluimos el siguiente código:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace TestMates
{
    class Program
    {
        static void Main(string[] args)
        {
            // Creamos una instancia de la clase
            Aritmetica operacion = new Aritmetica();
            // LLevamos a cabo la operacion
        }
    }
}
```

```
double resultado=operacion.Suma(3, 5);  
// Mostramos el resultado  
Console.WriteLine("El resultado es {0}", resultado);  
}  
}  
}
```

El código es muy sencillo: simplemente, estamos creando una instancia de la clase **Aritmetica**, la cual llamamos **operacion**. Luego invocamos al método **Suma** pasando como parámetros los valores **3** y **5**. El resultado de este método se guarda en la variable **resultado**, la cual es de tipo **double**. Por último, imprimimos el valor de resultado.

En este punto ya tenemos el código que necesitamos, pero encontramos algo interesante en la ventana que corresponde a la lista de errores, tal como lo vemos en la **Figura 7**.



► **Figura 7.** Ya tenemos el código; sin embargo podemos darnos cuenta de que aparecerán algunos errores.

Estos errores no se deben a que tengamos un problema de sintaxis o lógica en el código, sino a que estamos tratando de usar una clase que no reconoce nuestra aplicación, porque todavía no hemos referenciado en ella al assembly que contiene la clase por utilizar. Para proceder a corregir este inconveniente, será absolutamente preciso que realicemos un cambio en el código que corresponde y, luego, procedamos a referenciar el assembly necesario. El código ahora debe quedar de la siguiente manera:


```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
// Usamos el namespace necesario
using MisMates;
namespace TestMates
{
    class Program
    {
        static void Main(string[] args)
        {
            // Creamos una instancia de la clase
            Aritmetica operacion = new Aritmetica();
            // LLevamos a cabo la operacion
            double resultado=operacion.Suma(3, 5);
            // Mostramos el resultado
            Console.WriteLine("El resultado es {0}", resultado);
        }
    }
}
```

Como observamos, hemos adicionado el namespace **MisMates**, que es el correspondiente al assembly de la biblioteca de clases. Pero esto no es suficiente; aún precisamos agregar la referencia al assembly. Nos dirigimos al Explorador de soluciones y, en el proyecto **TestMates**,



¿TE RESULTA ÚTIL?

Lo que estás leyendo es el fruto del **trabajo de cientos de personas** que ponen todo de sí para lograr un **mejor producto**. Utilizar versiones "**pirata**" desalienta la inversión y da lugar a publicaciones de **menor calidad**.

NO ATENTES CONTRA LA LECTURA. NO ATENTES CONTRA TI. COMPRA SÓLO PRODUCTOS ORIGINALES.

Nuestras publicaciones se comercializan en kioscos o puestos de voceadores; librerías; locales cerrados; supermercados e internet (usershop.redusers.com). Si tienes alguna duda, comentario o quieres saber más, puedes contactarnos por medio de usershop@redusers.com

debemos buscar una carpeta llamada **References**. Para cotinuar, hacemos clic con el botón derecho del mouse sobre ella y aparecerá un menú contextual, del cual seleccionamos la opción **Adicionar Referencia**.

Se abre una nueva ventana en la pantalla, que nos permite adicionar la referencia. Hay diversas clasificaciones de referencias que podemos agregar, y están organizadas. Nosotros hemos creado nuestra propia biblioteca de clases, por lo que tenemos que buscarla seleccionando **Browse**. En caso de que fuera un assembly que perteneciera al framework, seleccionaríamos .NET. En la misma ventana aparece un navegador, por lo que simplemente nos dirigimos a la carpeta donde está el dll de nuestra biblioteca de clases. En su interior hallaremos **MisMates.dll**; lo seleccionamos y presionamos el botón **OK** o **Aceptar**.

Podemos verificar que el assembly ha sido adicionado en forma sencilla notando que, cuando intentamos escribir el **using** del namespace del assembly, este ya aparecerá en el autocompletar.

En la ventana del Explorador de soluciones elegimos el nombre del proyecto y hacemos clic con el botón derecho del mouse. Elegimos **Seleccionar como Proyecto de Inicio** y el proyecto arrancará.

Ahora podemos intentar otra vez ejecutar la aplicación, y veremos que todo funciona como corresponde. El proyecto que se ejecuta se denomina **TestMates**, el cual es una aplicación de consola, y vemos su ejecución en el monitor de computadora. **TestMates** funciona con el dll de la biblioteca de clases para **MisMates**. Con esto, ya hemos creado una aplicación que hace uso de un assembly que contiene una biblioteca de clases. Aunque esto resulto sencillo en este apéndice, es de gran ayuda y muy útil para nuestros propios proyectos.



RESUMEN



Los assemblies son componentes a partir de los cuales se crean las aplicaciones de .NET. El assembly es un archivo binario que usa el CLR. El propio framework de .NET está compuesto por muchos assemblies que usamos en las aplicaciones. Una de las ventajas que ofrecen es la reutilización de código; también sirven de contenedores para los tipos que definimos como programadores y tienen un sistema de versiones que es bastante útil. Dentro de los assemblies se guarda información que permite describirlos y para que el CLR sepa cómo trabajar con ellos. El namespace nos ayuda a agrupar de manera lógica los diferentes tipos que usamos para construir una aplicación.



Servicios al lector

En esta sección nos encargaremos de presentar un útil índice temático para poder encontrar en forma sencilla los términos que necesitamos. Además, podremos ver una interesante selección de sitios y programas que se encuentran relacionados con el contenido de esta obra.



▼ Índice temático 314



Índice temático

A

ABC	278
Acelerómetro	242
Add	18
ADT	92
Angle	230
Animación	222
API	168
Arquitectura WPF.....	167, 168
Array.....	31, 42
ArrayList.....	54
Assemblies	300

B

Back.....	240
Background	172
Biblioteca de controles.....	229
Binarios.....	98
Binding.....	277
Block.....	225
Bool	15
Border.....	172, 200
BinaryFormatter	133, 134

C

C# Express.....	16
Callback	50
Canvas.....	181
Chassis	240
CIL.....	300
Clase Ellipse	264
Clase Math.....	254
Clases abstractas.....	19
Cientes para WCF	280
Clonado.....	32
Clone().....	35
CLR	300
Code behind	167

C

Colección de controles.....	230
COM.....	268
COM+	290
Combine	69
Contenedores	169
ContentControl.....	169
Contrato de operaciones.....	276
Contrato de servicios.....	275
Control de marcos	219
Controles.....	201
Controles propios	209
Coordenadas en píxeles.....	212

D

Datos no serializables.....	132
DCOM	268
Debug.....	307
Degradado lineal	204
Degradado radial.....	213
Delegados	55, 68, 86
Delegate.....	69
Deserialización.....	128
DirectX	166
DLL	305
DockPanel.....	179
Double.....	307

E

Editor de archivos	137
Efectos.....	219
Ejemplos WCF	276
EndPoint	211
Enterprise services.....	271
Equals	116
Eventos	84
Expander	192
Explorador de soluciones	17, 18

F

False	97
For	96
Foreach	25, 26
Framework.....	25

G

GAC	302
Gaussian.....	225
GDI+.....	166
Get	95
Gráficas vectoriales	169
Grid.....	170, 174, 175
GridPanel.....	170
GroupBox.....	189, 190

H

Handlers.....	51
Handlers múltiples	52
Herencia	17
Herencia múltiple.....	19

I

Icloneable	32
Icomparable	39
IEnumerable	25, 29
If	118
Indexers	92
Interface	14
Is.....	118
Iterador.....	30

M

Magnitud.....	101
Main().....	34
Marco	195
Messaging	271
Método SetValue()	256
Métodos abstractos	14
Milcore	168
Modificador de acceso.....	31
Mscorlib.dll	300
MSDN.....	27

M

MSMQ	269
Multicasting	78
Multitouch.....	239, 243, 244

N

Namespace.....	302
Niveles de seguridad	271
Nombres únicos.....	303

O

Object.....	14, 35
Objetos	128
Offset	218
Opacity	221
Operador ++.....	99
Operador -=.....	76
Operador +=.....	71
Operadores.....	98
Operadores de comparación	103
Operadores no sobrecargados.....	104
Operadores sobrecargados.....	99
Operadores unuarios.....	97, 108
Operandos	98
Orden de relleno.....	185
Orientación predeterminada.....	178
Orientation.....	175
Override	104

P

Padding.....	176
Paneles	169
Performance	221
Persistencia.....	128
Polimorfismo.....	41
Portrait	239
Posición de los elementos.....	182
Program.....	72

R

Radios	227
RadiusX.....	215
RadiusY.....	215

R

References	311
Release	307
Remove	69
RemoveAll.....	69
Revista MSDN	247
Row.....	173
Runtime	300

S

Scroll	187
ScrollViewer	186
Sealed.....	61
Search.....	241
Sensores.....	242
Serializable	149
Serialización	128
Serialización binaria sencilla.....	130
Serialización SOAP	138
Serialización XML	143, 152
Servicio.....	278
Servicio WCF	283
SessionMode	276
Set	95
Silverlight.....	166, 238
Sink.....	51
Sintaxis del indexer.....	121
SOA	272
SOAP	128
Sobrecarga.....	80
Sobrecarga de operadores.....	99
Sobrecarga por pares.....	108
StackPanel	175
StartPoint	211
Static	70
Stream	128
Suma y resta de vectores	100
System.collections.....	95
System.object.....	15
System.uri	279

T

Tamaño de ventana	171
Text.....	172
TextBox.....	228
This.....	94
Title	172
Top.....	181
True.....	97
Tutorial WCF.....	282
Type	157
Type cast	54

U

UIElement	167
Unidades	171
UniformGrid.....	183
User32.....	168
Using.....	300

V

Vectores	100
Versiones anteriores de .NET	279
Visual Studio 2010.....	14
Visual Studio 2010 Express	237

W

Web-services.....	271
Windows	172
Windows Communcation Foundation.....	268
Windows Phone 7	236
WindowsBase.dll	167
WindowsCodecs.dll.....	168
WPF	166
WPF y 3D	207
WrapPanel.....	177
WS-Security.....	279
WS-Transactions	279

X

XAML	166, 176
XBAP.....	167
XML	166
XNA.....	236

CLAVES PARA COMPRAR UN LIBRO DE COMPUTACIÓN

1 SOBRE EL AUTOR Y LA EDITORIAL

Revise que haya un cuadro "sobre el autor", en el que se informe sobre su experiencia en el tema. En cuanto a la editorial, es conveniente que sea especializada en computación.

2 PRESTE ATENCIÓN AL DISEÑO

Compruebe que el libro tenga guías visuales, explicaciones paso a paso, recuadros con información adicional y gran cantidad de pantallas. Su lectura será más ágil y atractiva que la de un libro de puro texto.

3 COMPARE PRECIOS

Suele haber grandes diferencias de precio entre libros del mismo tema; si no tiene el valor en tapa, pregunte y compare.

4 ¿TIENE VALORES AGREGADOS?

Desde un sitio exclusivo en la Red, un Servicio de Atención al Lector, la posibilidad de leer el sumario en la Web para evaluar con tranquilidad la compra, y hasta la presencia de adecuados índices temáticos, todo suma al valor de un buen libro.

5 VERIFIQUE EL IDIOMA

No solo el del texto; también revise que las pantallas incluidas en el libro estén en el mismo idioma del programa que usted utiliza.



usershop.redusers.com

VISITE NUESTRO SITIO WEB

» Vea información más detallada sobre cada libro de este catálogo.

» Obtenga un capítulo gratuito para evaluar la posible compra de un ejemplar.

» Conozca qué opinaron otros lectores.

» Compre los libros sin moverse de su casa y con importantes descuentos.

» Publique su comentario sobre el libro que leyó.

» Manténgase informado acerca de las últimas novedades y los próximos lanzamientos.

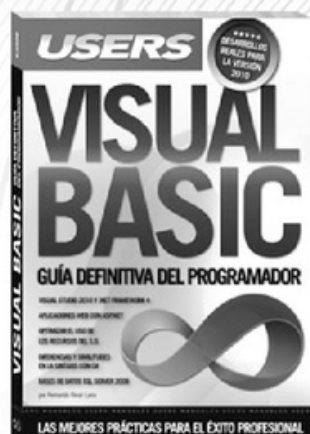
**TAMBIÉN PUEDE CONSEGUIR NUESTROS LIBROS EN KIOSCOS O
PUESTOS DE PERIÓDICOS, LIBRERÍAS, CADENAS COMERCIALES,
SUPERMERCADOS Y CASAS DE COMPUTACIÓN.**



LLEGAMOS A TODO EL MUNDO VÍA »OCA * Y **

* SOLO VÁLIDO EN LA REPÚBLICA ARGENTINA // ** VÁLIDO EN TODO EL MUNDO EXCEPTO ARGENTINA

usershop.redusers.com // **usershop@redusers.com**



Visual Basic

Este libro está escrito para aquellos usuarios que quieran aprender a programar en VB.NET. Desde el IDE de programación hasta el desarrollo de aplicaciones del mundo real en la versión 2010 de Visual Studio, todo está contemplado para conocer en profundidad VB.NET al finalizar la lectura.

- COLECCIÓN: MANUALES USERS
- 352 páginas / 978-987-1773-57-2



Microcontroladores

Este manual es ideal para aquellos que quieran iniciarse en la programación de microcontroladores. A través de esta obra, podrán conocer los fundamentos de los sistemas digitales, aprender sobre los microcontroladores PIC 16F y 18F, hasta llegar a conectar los dispositivos de forma inalámbrica, entre muchos otros proyectos.

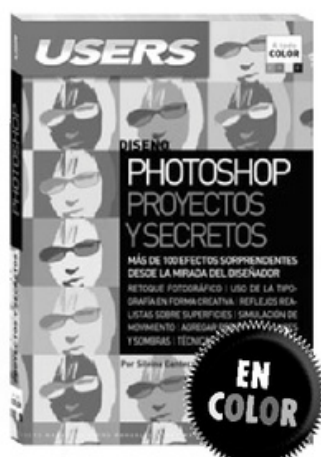
- COLECCIÓN: MANUALES USERS
- 320 páginas / 978-987-1773-56-5



Programador.NET

Este libro está dirigido a todos aquellos que quieran iniciarse en el desarrollo bajo lenguajes Microsoft. A través de los capítulos del manual, aprenderemos sobre POO y la programación con tecnologías .NET, su aplicación, cómo interactúan entre sí y de qué manera se desenvuelven con otras tecnologías existentes.

- COLECCIÓN: MANUALES USERS
- 352 páginas / 978-987-1773-26-8



Photoshop: proyectos y secretos

En esta obra aprenderemos a utilizar Photoshop, desde la original mirada de la autora. Con el foco puesto en la comunicación visual, a lo largo del libro adquiriremos conocimientos teóricos, al mismo tiempo que avanzaremos sobre la práctica, con todos los efectos y herramientas que ofrece el programa.

- COLECCIÓN: MANUALES USERS
- 320 páginas / 978-987-1773-25-1



WordPress

Este manual está dirigido a todos aquellos que quieran presentar sus contenidos o los de sus clientes a través de WordPress. En sus páginas el autor nos enseñará desde cómo llevar adelante la administración del blog hasta las posibilidades de interacción con las redes sociales.

- COLECCIÓN: MANUALES USERS
- 352 páginas / 978-987-1773-18-3



Administrador de servidores

Este libro es la puerta de acceso para ingresar en el apasionante mundo de los servidores. Aprenderemos desde los primeros pasos sobre la instalación, configuración, seguridad y virtualización; todo para cumplir el objetivo final de tener el control de los servidores en la palma de nuestras manos.

- COLECCIÓN: MANUALES USERS
- 352 páginas / ISBN 978-987-1773-19-0

¡Léalo antes Gratis!

En nuestro sitio, obtenga GRATIS un capítulo del libro de su elección antes de comprarlo.



Windows 7: Trucos y secretos

Este libro está dirigido a todos aquellos que quieran sacar el máximo provecho de Windows 7, las redes sociales y los dispositivos ultrapotables del momento. A lo largo de sus páginas, el lector podrá adentrarse en estas tecnologías mediante trucos inéditos y consejos asombrosos.

- COLECCIÓN: MANUALES USERS
- 352 páginas / ISBN 978-987-1773-17-6



Desarrollo PHP + MySQL

Este libro presenta la fusión de dos de las herramientas más populares para el desarrollo de aplicaciones web de la actualidad: PHP y MySQL. En sus páginas, el autor nos enseñará las funciones del lenguaje, de modo de tener un acercamiento progresivo, y aplicarlo aprendido en nuestros propios desarrollos.

- COLECCIÓN: MANUALES USERS
- 432 páginas / ISBN 978-987-1773-16-9



Excel 2010

Este manual resulta ideal para quienes se inician en el uso de Excel, así como también para los usuarios que quieran conocer las nuevas herramientas que ofrece la versión 2010. La autora nos enseñará desde cómo ingresar y proteger datos hasta la forma de imprimir ahorrando papel y tiempo.

- COLECCIÓN: MANUALES USERS
- 352 páginas / ISBN 978-987-1773-15-2



Técnico Hardware

Esta obra es fundamental para ganar autonomía al momento de reparar la PC. Aprenderemos a diagnosticar y solucionar las fallas, así como a prevenirlas a través del mantenimiento adecuado, todo explicado en un lenguaje práctico y sencillo.

- COLECCIÓN: MANUALES USERS
- 320 páginas / ISBN 978-987-1773-14-5



PHP Avanzado

Este libro brinda todas las herramientas necesarias para acercar al trabajo diario del desarrollador los avances más importantes incorporados en PHP 6. En sus páginas, repasaremos todas las técnicas actuales para potenciar el desarrollo de sitios web.

- COLECCIÓN: MANUALES USERS
- 400 páginas / ISBN 978-987-1773-07-7



AutoCAD

Este manual nos presenta un recorrido exhaustivo por el programa más difundido en dibujo asistido por computadora a nivel mundial, en su versión 2010. En sus páginas, aprenderemos desde cómo trabajar con dibujos predeterminados hasta la realización de objetos 3D.

- COLECCIÓN: MANUALES USERS
- 384 páginas / ISBN 978-987-1773-06-0

USERS**PRESENTA...**

¡EL PRIMER EBOOK USERS!

Sí, ya podés leer Hackers al descubierto en tu PC, notebook, Amazon Kindle, iPad, en el celular...

**CONSEGUILO
DESDE CUALQUIER
PARTE DEL MUNDO**

**A UN PRECIO
INCREÍBLE**

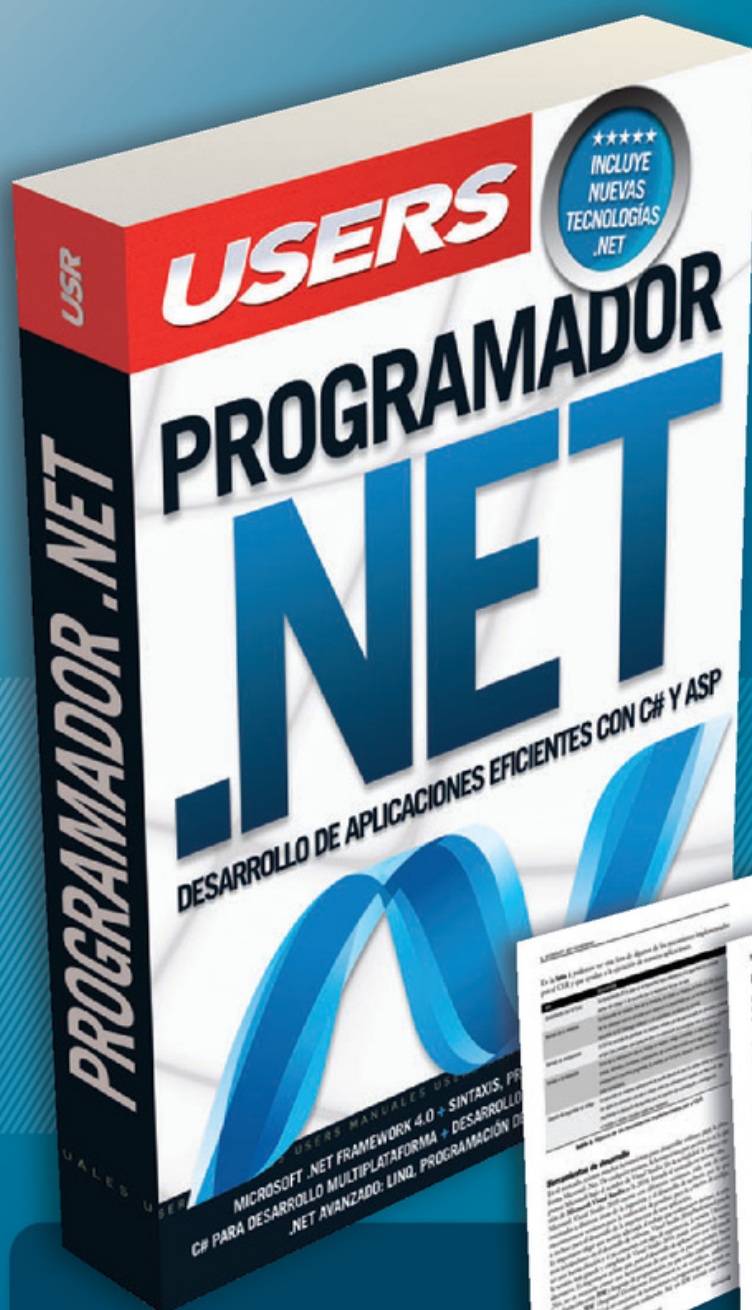
**¿QUÉ ESTÁS
ESPERANDO?**



**¡LEELO
DONDE
QUIERAS!**

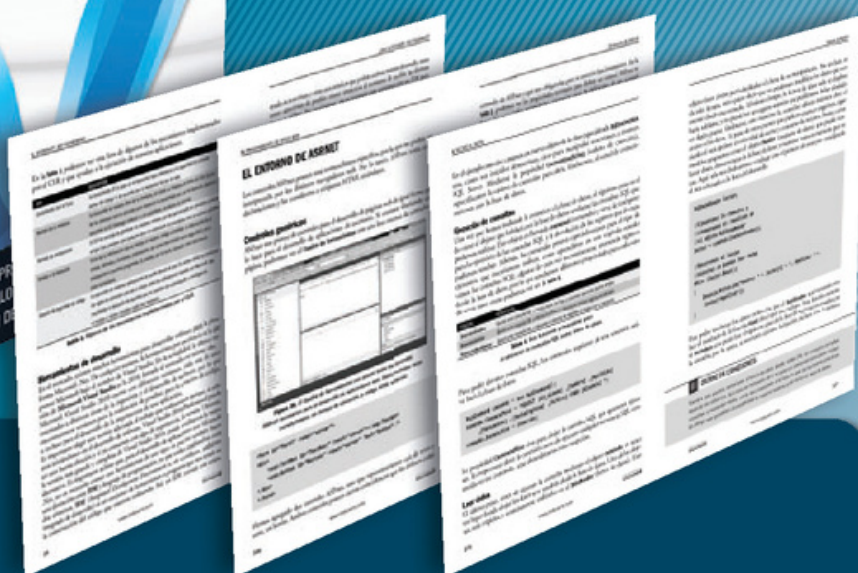
INGRESA YA A [USERSHOP.REDUSERS.COM](http://usershop.redusers.com) Y ENTERATE MÁS

DESARROLLO DE APLICACIONES EFICIENTES CON C# Y ASP



Este libro está dirigido a todos aquellos que quieran iniciarse en el desarrollo bajo lenguajes Microsoft. A través de los capítulos del manual, aprenderemos sobre POO y la programación con tecnologías .NET, su aplicación, cómo interactúan entre sí y de qué manera se desenvuelven con otras tecnologías existentes.

» DESARROLLO
» 352 PÁGINAS
» ISBN 978-987-1773-26-8



LLEGAMOS A TODO EL MUNDO VÍA **DOCA** * Y **DHL** **

* SÓLO VÁLIDO EN LA REPÚBLICA ARGENTINA // ** VÁLIDO EN TODO EL MUNDO EXCEPTO ARGENTINA

usershop.redusers.com // usershop@redusers.com

CONTENIDO

1 | INTERFACES

Entender las interfaces / Definir la interfaz /
Comparación entre interfaces y clases abstractas /
Creación de una aplicación simple con interfaz /
El uso de IEnumerable / El uso de ICloneable /
El uso de IComparable

2 | CALLBACK, DELEGADOS Y EVENTOS

Creación de la interfaz de callback / Entender el sink /
Cómo utilizar dos o más eventos / Los delegados /
Delegados con multicasting / Entender los eventos

3 | MANEJO AVANZADO DE CLASES

Indexers / Vectores / Aplicación con sobrecarga
de operadores binarios / Sobrecarga de operadores
unarios / Sobrecarga de operadores de igualdad
y comparación / Sobrecarga de los operadores < y >

4 | SERIALIZACIÓN

Guardar el estado de los objetos / Serialización binaria /
Serialización SOAP / Serialización XML / Serialización
de clases con composición en formato binario /
Serialización usando XML / Serialización de varios objetos

5 | INTRODUCCIÓN A WPF Y XAML

Entender WPF / Arquitectura de WPF / Entender
los contenedores / Los paneles

6 | XAML Y LAS INTERFACES DE USUARIO

Efectos y brochas para el fondo de Border / Degradado /
Control de marcos / Control de la interfaz

7 | PROGRAMAR PARA WINDOWS PHONE 7

Windows Phone 7 / Proyecto para Windows Phone /
Hardware / Colocar la orientación de la aplicación /
El sistema multi-touch

8 | PROGRAMAR EN WCF

Entender Windows Communication Foundation /
El modelo de comunicación / Creación de aplicaciones
WCF / Código para un servicio WCF

NIVEL DE USUARIO

PRINCIPIANTE

INTERMEDIO

AVANZADO

EXPERTO

C# AVANZADO

SOLUCIONES REALES, CON LAS ESTRATEGIAS DE DESARROLLO MÁS ACTUALES

Este libro abre nuevas perspectivas para la programación de aplicaciones con C#, al llevar al lector a recorrer temas más avanzados del lenguaje y del framework .NET.

Con la lectura de esta obra, conoceremos técnicas avanzadas de programación que nos permitirán hacer un desarrollo más eficiente y tener soluciones más fáciles de programar. Aprenderemos cómo guardar el estado de los objetos en disco, crear interfaces de usuario avanzadas por medio del uso de XAML y WPF, utilizar las bases de la programación de Windows Phone 7, y generar aplicaciones distribuidas aprovechando el poder de WCF. Luego, a medida que avancemos, el libro se adentrará en los temas más complejos del framework, como WPF, XAML y WCF.

Todos los procedimientos son expuestos de manera práctica con el código fuente de ejemplo (disponible en Internet), diagramas conceptuales y la teoría necesaria para comprender en profundidad cada tema presentado.

Al finalizar el recorrido, seremos capaces de realizar desarrollos complejos y profesionales, mediante código sencillo, pero, al mismo tiempo, poderoso.

Nicolás Arriola Landa Cosío, catedrático, investigador y consultor, es además autor de múltiples libros sobre C#, publicados en esta editorial, y es quien nos llevará en este nuevo recorrido por el máximo nivel del lenguaje de Microsoft.

RedUSERS.com

En este sitio encontrará una gran variedad de recursos y software relacionado, que le servirán como complemento al contenido del libro. Además, tendrá la posibilidad de estar en contacto con los editores, y de participar del foro de lectores, en donde podrá intercambiar opiniones y experiencias.

Si desea más información sobre el libro puede comunicarse con nuestro Servicio de Atención al Lector: usershop@redusers.com

ADVANCED C#

This book offers new perspectives for those already programming with C# or for those who seek to improve their skills. It also includes advanced concepts and exercises that will greatly enhance the developer's experience.

