

USERS

APRENDA A
PROGRAMAR SIN
CONOCIMIENTOS
PREVIOS



GUÍA TOTAL DEL PROGRAMADOR

VARIABLES, EXPRESIONES Y OPERADORES

DESARROLLO DE APLICACIONES
PARA EL FRAMEWORK .NET 4.0

ARREGLOS, COLECCIONES Y CADENAS

PROGRAMACIÓN ORIENTADA A OBJETOS

CLASES, MÉTODOS Y PROPIEDADES

por Nicolás Arrioja Landa Cosio



MANUALES USERS MANUALES USERS MANUALES USERS MANUALES USERS MANUALES

EXPERIMENTE TODO EL PODER DE VISUAL STUDIO 2010

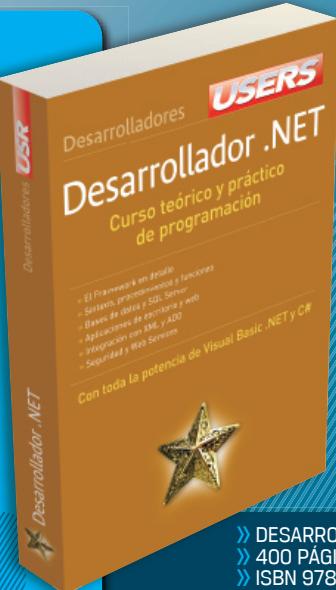
www.FreeLibros.me

CONÉCTESE CON LOS MEJORES LIBROS DE COMPUTACIÓN

LLEGAMOS A TODO EL MUNDO
VÍA **DOCA*** Y **DHL****
usershop.redusers.com
usershop@redusers.com

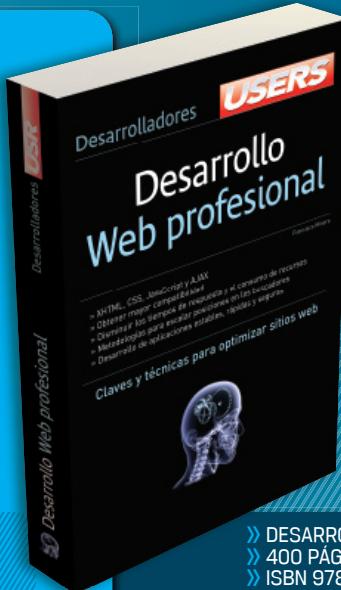


SÓLO VÁLIDO EN LA REPÚBLICA ARGENTINA // **VÁLIDO EN TODO EL MUNDO EXCEPTO ARGENTINA



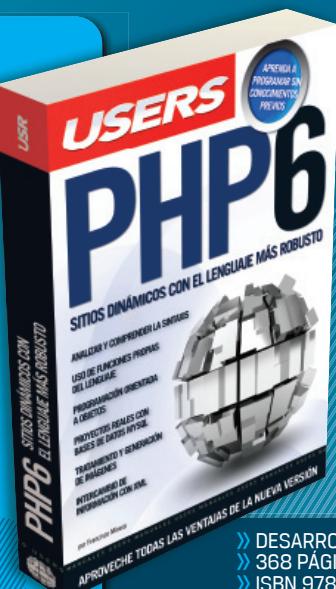
LA PREPARACIÓN
IDEAL PARA
DESARROLLADOR
DE MICROSOFT

- » DESARROLLO / MICROSOFT
- » 400 PÁGINAS
- » ISBN 978-987-1347-74-2



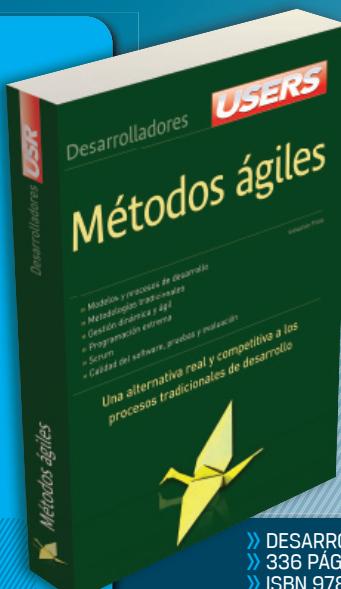
APRENDA
XHTML,
CSS,
JAVASCRIPT
Y AJAX

- » DESARROLLO / INTERNET
- » 400 PÁGINAS
- » ISBN 978-987-1347-70-4



APRENDA A CREAR
SITIOS DINÁMICOS
CON EL LENGUAJE
MÁS ROBUSTO

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



UNA ALTERNATIVA
COMPETITIVA A
LOS MÉTODOS
TRADICIONALES

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

C#

GUÍA TOTAL DEL PROGRAMADOR

por Nicolás Arrioja Landa Cosio

RedUSERS



TÍTULO: C#
AUTOR: Nicolás Arrioja Landa Cosio
COLECCIÓN: Manuales USERS
FORMATO: 17 x 24 cm
PÁGINAS: 400

Copyright © MMX. Es una publicación de Fox Andina en coedición con Gradi S.A. Hecho el depósito que marca la ley 11723. Todos los derechos reservados. No se permite la reproducción parcial o total, el almacenamiento, el alquiler, la transmisión o la transformación de este libro, en cualquier forma o por cualquier medio, sea electrónico o mecánico, mediante fotocopias, digitalización u otros métodos, sin el permiso previo y escrito del editor. 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, MMX.

ISBN 978-987-26013-5-5

Landa Cosio, Nicolás Arrioja
C#. - 1a ed. - Buenos Aires : Fox Andina; Gradi; Banfield - Lomas de Zamora, 2010.
400 p. ; 24x17 cm. - (Manual Users; 195)

ISBN 978-987-26013-5-5

1. Informática. I. Título

CDD 005.3



LÉALO ANTES GRATIS

EN NUESTRO SITIO PUEDE OBTENER, DE FORMA GRATUITA, UN CAPÍTULO DE CADA UNO DE LOS LIBROS

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  *  **

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

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

Nicolás Arrioja Landa Cosío



Catedrático de IUNIVERTECH y de la Universidad Madero, elegido el mejor profesor de los ciclos 2006-2007 y 2007-2008, se dedica a la investigación, consulta y capacitación en áreas relacionadas con la realidad virtual, la visualización científica y los videojuegos. En 1997, desarrolló el primer videojuego de realidad virtual inmersiva en Latinoamérica, conocido como **VRaptor**. También desarrolló el primer lenguaje de programación para realidad virtual en Latinoamérica **CND-VR**, que es usado en el medio académico para enseñar de forma sencilla los conceptos de programación de gráficas 3D. Ha sido catedrático en algunas de las universidades más importantes de México durante más de diez años. En esas instituciones enseñó desde las bases de programación en lenguaje C hasta Inteligencia Artificial. Ha otorgado más de 25 conferencias relacionadas con el desarrollo de la realidad virtual y los videojuegos en diversas universidades de México. Tiene una patente referida a interfaces cinemáticas para video juegos. Diseñó el plan y el programa de estudios para la Maestría en Realidad Virtual y Video Juegos en IUNIVERTECH. Es autor del los libros DirectX, Inteligencia Artificial y Robótica Avanzada de esta misma editorial. Actualmente, realiza una investigación sobre Inteligencia Artificial no-Algorítmica y tiene disposición a realizar investigación y dar conferencia en estas áreas en cualquier parte del mundo.

Dedicatoria

A María Eugenia Pérez Duarte por todo su apoyo y gran amistad.

Agradecimientos

A todas aquellas personas que hace 25 años me introdujeron a la programación de computadoras y me enseñaron las bases de esta disciplina. De **Sistem Club** al Ing. Alfonso Fernández de la Fuente y a Jacky, Annie y David Fox, Herbert Schildt y Kris Jamsa. Un agradecimiento especial a mis alumnos y al equipo de la editorial.

PRÓLOGO

La facilidad con que hoy en día se tiene acceso a una computadora ha llevado a las personas a emplearlas en gran cantidad de actividades, usando para ello software que realiza tareas específicas. Cuando se requiere que la computadora realice otro tipo de acciones y los programas con los que se cuenta no lo hacen, es necesario decirle en su propio idioma cómo deberá actuar, y para esto se recurre a los lenguajes de programación. Los lenguajes de programación intentan parecerse a nuestra propia forma de expresarnos, para ello cuentan con características propias de sintaxis y semántica, así como símbolos que permiten decirle a la computadora qué procesos y qué datos debe manejar, cómo almacenarlos, qué acciones realizar con ellos y cómo presentarlos al usuario. Por esta razón es importante contar con una metodología que permita aprender hábitos correctos y efectivos para resolver problemas, los cuales, al ser subdivididos para su mejor comprensión, facilitarán su resolución.

Una vez que se ha comprendido el problema, se construye una secuencia de los pasos a realizar, mejor conocida como algoritmo, el cual permitirá probar si la lógica es correcta y si los resultados que se obtendrán serán los adecuados. El potencial de los actuales lenguajes de programación es enorme, ya que permiten automatizar muy diversos tipos de actividades, disminuyendo de ese modo el tiempo que se invierte al hacerlo de forma manual y, si se aprende a usarlos correctamente, se podrá hacer más sencilla la vida de los usuarios que emplean dichos programas.

La experiencia adquirida en el desarrollo de sistemas por parte del Dr. Nicolás Arrioja lo ha convertido en una autoridad en cuanto a programación y buenas prácticas, compartiéndolas y enseñando a alumnos de diferentes instituciones de nivel superior, con resultados excelentes. Es por eso que esta nueva versión del libro podrá guiar paso a paso al lector en el uso del poderoso lenguaje de programación C# del Visual Studio 2010, de modo que al finalizar su aprendizaje, esté listo para hacer sus propios programas.

Aprender a programar es una tarea que requiere práctica y paciencia, pero también es una actividad fascinante, y si además se cuenta con un excelente libro y la constancia necesaria, los resultados serán excelentes. ¡Felicidades por este nuevo reto que está por comenzar!

María Eugenia Pérez Duarte
*Maestra en Administración de Tecnologías de Información
Coordinadora de Ingenierías en Sistemas
Computacionales y Desarrollo de Software
Universidad Madero de Méjico*

EL LIBRO DE UN VISTAZO

El desarrollo de aplicaciones con C# se ha visto modificado en los últimos tiempos, y este libro apunta a afianzar los conocimientos indispensables que nos permitirán generar aplicaciones profesionales de nivel. Cada capítulo está dedicado a una técnica específica e ilustrado mediante ejemplos prácticos listos para implementar.

Capítulo 1

C# Y .NET

Este capítulo nos introduce a la historia del desarrollo de los programas para Windows y cómo el Framework de .NET ayuda a resolver muchos problemas. Aprenderemos cómo crear y ejecutar nuestro primer programa.

Capítulo 2

LOS ELEMENTOS BÁSICOS DE UN PROGRAMA

Aprender cómo resolver problemas en la computadora es aún más importante que conocer la sintaxis de un lenguaje. Aquí aprenderemos los elementos básicos de un programa, pero también la metodología para diseñar buenos programas desde el inicio.

Capítulo 3

EL PROGRAMA TOMA DECISIONES

Los programas de computadora necesitan lógicas complejas, por lo que es importante conocer las estructuras necesarias que nos permitirán decirle a la computadora cómo tomar una decisión.

Capítulo 4

CREACIÓN DE CICLOS

Los ciclos nos permiten repetir algo un número de veces. Los diferentes tipos de ciclos, sus partes y aplicaciones serán enseñados en este capítulo.

Capítulo 5

FUNCIONES Y MÉTODOS

Las funciones y los métodos nos brindan flexibilidad y la capacidad de poder volver a utilizar código fácilmente, y también nos ayudan a ordenar nuestros desarrollos, y facilitan su mantenimiento.

Capítulo 6

LOS ARREGLOS

Los arreglos, muy útiles en todos los casos, nos permiten administrar la información de una manera fácil y práctica, como si trabajásemos con una planilla de cálculo. En este capítulo estudiaremos arreglos de una y dos dimensiones, y por supuesto no dejaremos de lado los arreglos de tipo jagged.

Capítulo 7

LAS COLECCIONES

Las colecciones son estructuras de datos que nos permiten trabajar con grupos de información. Veremos aquellas más importantes, como el ArrayList, Stack, Queue y Hashtable.

Capítulo 8

LAS CADENAS

Las cadenas permiten guardar y manipular texto o frases. C# nos provee de éstas para su manejo. Aprenderemos cómo hacer uso de las cadenas y trabajar con ellas.

Capítulo 9**ESTRUCTURAS Y ENUMERACIONES**

En este capítulo aprenderemos cómo crear nuestros propios tipos por medio de las estructuras y enumeraciones. Las estructuras también nos permiten agrupar la información de una manera óptima, lo cual nos ayuda a manejarla más fácilmente.

Capítulo 10**CÓMO CREAR NUESTRAS PROPIAS CLASES**

En este capítulo empezaremos a conocer los conceptos de la programación orientada a objetos, lo más utilizado hoy en día. Podremos crear nuestras propias clases y nuestros propios objetos, y también trabajaremos con métodos definidos en las clases.

Capítulo 11**FLUJOS Y ARCHIVOS**

Es muy frecuente que necesitemos guardar información desde un programa al disco. Por medio de los flujos y los archivos lo podemos hacer. Guardaremos y leeremos cadenas, y veremos que la manipulación de archivos también es posible.

Capítulo 12**DEPURACIÓN**

La depuración nos permite corregir los problemas que tiene un programa. Conoceremos los tipos de errores y cómo usar el depurador para eliminarlos. También aprenderemos cómo administrar las excepciones de la mejor manera .

**INFORMACIÓN COMPLEMENTARIA**

A lo largo de este manual encontrará una serie de recuadros que le brindarán información complementaria: curiosidades, trucos, ideas y consejos sobre los temas tratados.

Cada recuadro está identificado con uno de los siguientes iconos:



CURIOSIDADES
E IDEAS



ATENCIÓN



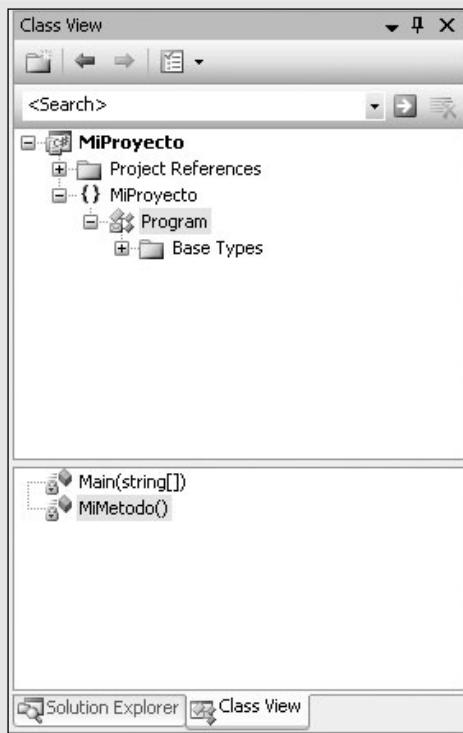
DATOS ÚTILES
Y NOVEDADES



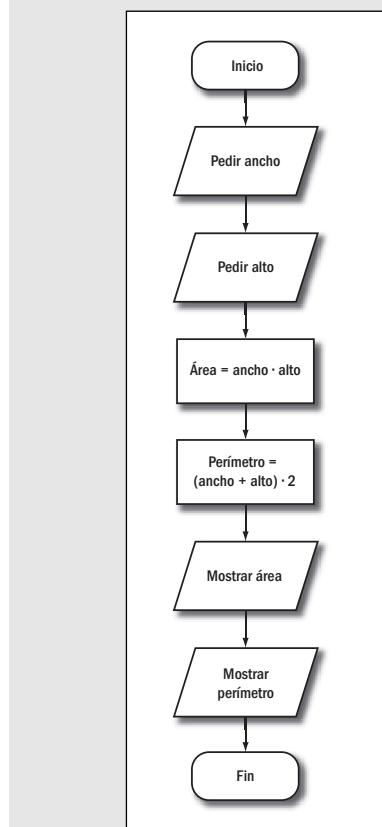
SITIOS WEB

CONTENIDO

Sobre el autor	4
Prólogo	5
El libro de un vistazo	6
Introducción	12
Capítulo 1	
C# Y .NET	
Breve historia de la programación para Windows	14
Descubrir .NET	15
Cómo crear una aplicación .NET	17
Conseguir un compilador de C#	18
El ambiente de desarrollo	18
Crear nuestra primera aplicación	19
Resumen	29
Actividades	30

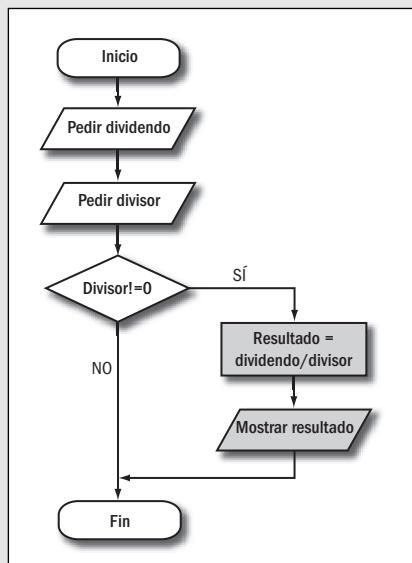
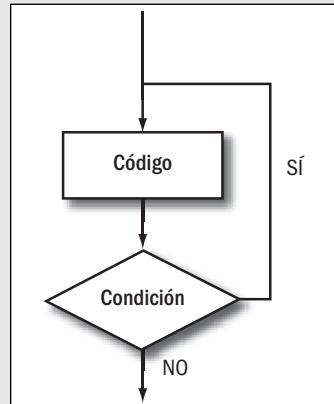


Capítulo 2	
LOS ELEMENTOS BÁSICOS DE UN PROGRAMA	
Los lenguajes de programación	32
Los programas de computadora	32
Las variables	42
Operaciones aritméticas	49
Cómo pedirle datos al usuario	56
Cómo resolver problemas en la computadora	60
Resolución de problemas en la computadora	64
Resumen	69
Actividades	70



Capítulo 3

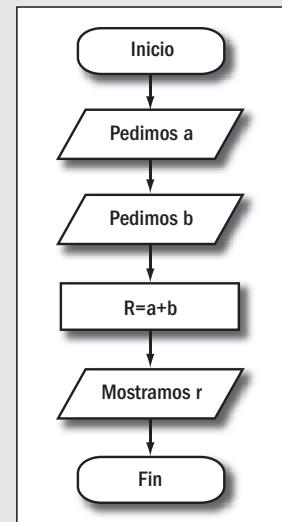
EL PROGRAMA TOMA DECISIONES	
La toma de decisiones	72
Expresiones relacionales	72
El uso de if	76
El uso de else	84
Cómo usar if anidados	89
Escalera de if-else	92
Expresiones lógicas	96
El uso de switch	105
Resumen	109
Actividades	110

**Capítulo 4**

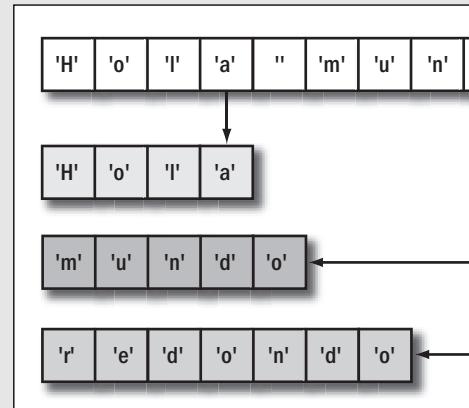
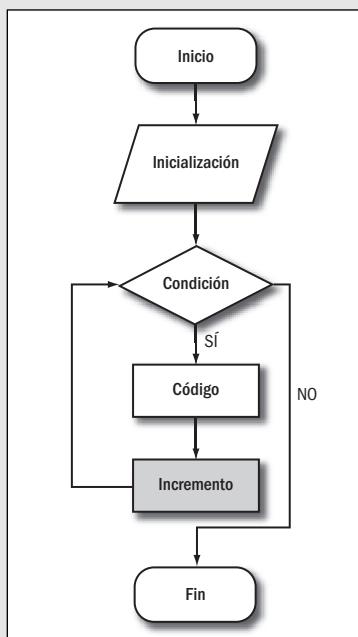
CREACIÓN DE CICLOS	
El ciclo for	112
El valor de inicio	117
El límite de conteo del ciclo	119
Control del incremento	121
Ejemplos con el ciclo for	128
El ciclo do while	134
El ciclo while	141
Resumen	145
Actividades	146

Capítulo 5

FUNCIONES Y MÉTODOS	
Las funciones	148
Funciones que ejecutan código	150
Funciones que regresan un valor	156
Funciones que reciben valores	159
Funciones que reciben parámetros y regresan un valor	162
Optimizar con funciones	171
Paso por copia y paso por referencia	178
Resumen	185
Actividades	186

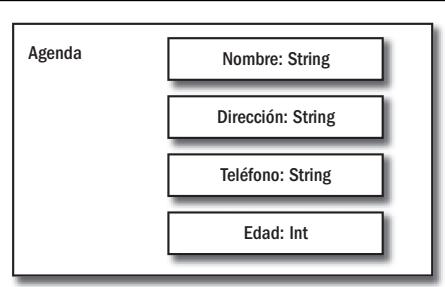


Capítulo 6		Capítulo 8	
LOS ARREGLOS		LAS CADENAS	
Los arreglos	188	El uso de las cadenas	256
Declaración de los arreglos de una dimensión	189	Cómo declarar la cadena	256
Asignación y uso de valores	191	El método ToString()	256
Arreglos de dos dimensiones	196	Cómo convertir y formatear fechas a cadenas	257
Arreglos de tipo jagged	205	Para darles formato a valores numéricos	259
Los arreglos como parámetros a funciones	212	Cómo concatenar cadenas	260
Resumen	215	Uso de StringBuilder	261
Actividades	216	Comparación de cadenas	263
Capítulo 7		Para encontrar una subcadena	264
LAS COLECCIONES		Para obtener una subcadena	265
Las colecciones más importantes	218	Para determinar si una cadena finaliza en una subcadena	266
El ArrayList	218	Cómo copiar una parte de la cadena	267
El Stack	232	Cómo insertar una cadena	268
El Queue	241	Para encontrar la posición de una subcadena	269
El Hashtable	249	Justificación del texto	270
Resumen	253	Para eliminar caracteres de la cadena	271
Actividades	254	Cómo reemplazar una subcadena	271



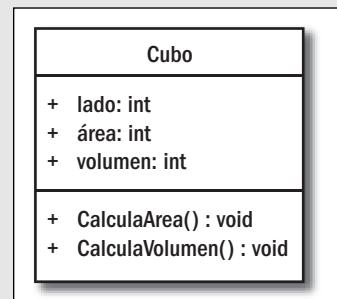
Capítulo 9

ESTRUCTURAS Y ENUMERACIONES	
Las estructuras	280
Cómo definir una estructura	280
Cómo crear una variable del nuevo tipo	282
Cómo acceder a los campos de la estructura	283
Cómo mostrar los campos de la estructura	283
Creación de un constructor para la estructura	286
Estructuras enlazadas	299
Las enumeraciones	309
Resumen	315
Actividades	316

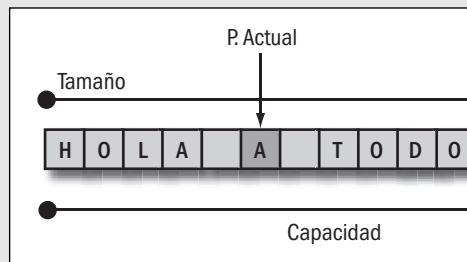
**Capítulo 10**

CÓMO CREAR NUESTRAS PROPIAS CLASES	
La programación orientada a objetos	318
Las clases	318
Cómo declarar la clase y los datos	320
Cómo crear la instancia de nuestra clase	324
Cómo asignarles valores a datos públicos	324
Cómo invocar métodos de un objeto	325
Cómo imprimir un dato público	325
Protección de datos y creación de propiedades	329
Cómo acceder a las propiedades	333
Métodos públicos y privados	333

Convertir un objeto a cadena	334
El constructor en las clases	336
Sobrecarga del constructor	339
Resumen	347
Actividades	348

**Capítulo 11**

FLUJOS Y ARCHIVOS	
Los flujos	350
Los stream en la memoria	351
El uso de archivos	363
Resumen	371
Actividades	372

**Capítulo 12**

DEPURACIÓN	
Cómo empezar a depurar un programa	374
Corregir los errores de compilación	374
Cómo corregir los errores en tiempo de ejecución	379
Cómo manejar los errores	383
Resumen	391
Actividades	392

INTRODUCCIÓN

La programación de computadoras es una actividad fascinante, el poder hacer una aplicación a la medida y de acuerdo a nuestras necesidades es una actividad interesante y llena de recompensas. Mucha gente llega a pensar que solamente con aprender la sintaxis de un lenguaje de programación es suficiente para poder hacer programas, pero se equivocan. Aun más importante que la sintaxis del lenguaje es la técnica de resolución del problema y el análisis adecuado. El tener buenos hábitos de análisis y programación desde el inicio es fundamental para convertirnos en buenos programadores.

Una de los lenguajes que está adquiriendo gran popularidad es C#, un lenguaje sencillo, amigable y poderoso. Con C# se pueden crear todo tipo de aplicaciones, desde programas de consola para Windows y para páginas web en unión con ASP, hasta video juegos para Xbox 360 con XNA.

El presente libro introduce a cualquier persona, aun sin conocimientos de programación, al lenguaje C#. Esta obra está actualizada para Visual Studio 2010 y el Framework de .NET 4.0. En un inicio veremos la forma de realizar el análisis de un problema e iremos construyendo el conocimiento hasta llegar a las bases de la programación orientada a objetos y, por supuesto, la depuración.

El convertirse en un buen programador requiere de mucha práctica y experimentación, pero también de una guía adecuada que vaya construyendo paso a paso las habilidades necesarias. Esperamos que este libro sea de utilidad a todos aquellos que, aun sin experiencia en programación, tengan el deseo de aprender como hacer sus propios programas.

Nicolás Arrioja Landa Cosio

C#.NET

Este libro está dirigido a todas aquellas personas que desean aprender el lenguaje de programación C# y tienen conocimientos básicos de programación o no tienen ninguna experiencia previa programando computadoras. El enfoque del libro es guiar paso a paso en el aprendizaje del lenguaje y aunque C# es un lenguaje orientado a objetos, en los primeros capítulos tomaremos una filosofía estructurada con el fin de hacer todavía más fácil el aprendizaje para todos nosotros.

Breve historia de la programación para Windows	14
Comprendiendo .NET	15
Cómo se crea una aplicación .NET	16
Cómo conseguir un compilador de C#	17
El ambiente de desarrollo	18
Cómo crear nuestra primera aplicación	19
Resumen	29
Actividades	30

BREVE HISTORIA DE LA PROGRAMACIÓN PARA WINDOWS

Hace algunos años la única forma como se podía programar para Windows era hacer uso de un compilador de C o C++ y de un **API** de Windows. El API es una gran colección de funciones que se relacionan, las que nos permiten comunicarnos con el sistema operativo. Por medio del API de **Win32** se programaban las ventanas, botones y demás elementos.

El problema de este tipo de programación es que el API de Win32 es realmente complejo y enorme. Con miles de funciones en su interior, por lo que pocos programadores podían conocerlo en su totalidad. Pero la complejidad no solamente estaba en la cantidad de funciones, también en la sintaxis y la forma como se programa.

Para facilitar la programación de aplicaciones para Windows surgen diferentes opciones; la finalidad de estos intentos era poder hacer las aplicaciones sin tener que pasar por la complejidad de Win32. Uno de estos intentos fue conocido como **OWL**; sin embargo, obtuvo más éxito **MFC**, creado por Microsoft.

MFC es un conjunto de clases que envuelve a Win32 y facilita su programación. Con MFC los procesos más comunes se agrupan en funciones de tal forma que con una simple llamada a una función de MFC se puede hacer una determinada tarea, para la que antes necesitábamos por lo menos 10 llamadas en Win32 y muchos parámetros. Sin embargo Win32 está debajo de MFC; la programación MFC simplifica mucho las cosas, pero muchos programadores que venían del paradigma de programación estructurada no se sentían a gusto con él.

Otra de las opciones que surgieron es **Visual Basic**, este lenguaje logró gran popularidad, especialmente en Latinoamérica. Visual Basic también trabaja por arriba de Win32, pero basa su sintaxis en el antiguo lenguaje **Basic**. Es muy sencillo de aprender y una de las características que le dio gran popularidad fue la facilidad con la que se podían crear interfaces de usuario y conectividad a bases de datos. Pero hasta antes de la versión .NET, este lenguaje tenía ciertos limitantes ya que no se podía llevar a cabo programación orientada a objetos con él.

Otro lenguaje que surge, pero con su propio **Framework**, es **JAVA**; su principal ventaja es ser multiplataforma. Una de sus características es el uso de un **runtime**, la aplicación en lugar de correr directamente en el microprocesador, se ejecuta en un programa llamado runtime y este se encarga de ejecutar el código en el microprocesador correspondiente. Si se tiene el runtime para Windows, sin problema se ejecuta el programa de JAVA.

Cuando nosotros deseábamos tener un programa que se pudiera ejecutar, era necesario **compilarlo**. Cada uno de los lenguajes tenía su propio **compilador**, por ello no era sencillo poder compartir código de C++ con código de Visual Basic ya que el traducir entre lenguajes era difícil. Para poder compartir código entre los lenguajes surge un mo-

délo conocido como **COM**, éste nos permite crear componentes binarios, esto quiere decir que es posible programar un componente en Visual Basic y un programador de C++ puede tomarlo y hacer uso de él. Esto se debe a que el componente ya es código compilado y no código fuente en el lenguaje de origen; la programación de COM también tenía sus complejidades y surge **ATL** para ayudar en su desarrollo.

Con todo esto, llega el momento en el cual es necesario ordenar, facilitar y organizar el desarrollo de las aplicaciones para Windows, con esta filosofía surge **.NET**.

Descubrir .NET

El **Framework** de .NET es una solución a toda la problemática en torno al desarrollo de aplicaciones, brinda grandes beneficios no solamente al desarrollador, sino también al proceso de desarrollo. En primer lugar .NET permite trabajar con código ya existente, podemos hacer uso de los componentes COM, e incluso, si lo necesitáramos usar el API de Windows. Cuando el programa .NET está listo es mucho más fácil de instalar en la computadora de los clientes, que las aplicaciones tradicionales ya que se tiene una integración fuerte entre los lenguajes.

Un programador de C# puede entender fácilmente el código de un programador de Visual Basic .NET y ambos pueden programar en el lenguaje con el que se sienten más cómodos. Esto se debe a que todos los lenguajes que hacen uso de .NET comparten las librerías de .NET, por lo que no importa en qué lenguaje programemos, las reconocemos en cualquiera. A continuación conoceremos los diferentes componentes de .NET: **CLR**, **assembly** y **CIL**.

CLR

El primer componente de .NET que conoceremos es el **Common Language Runtime**, también denominado **CLR**. Este es un programa de ejecución común a todos los lenguajes. Este programa se encarga de leer el código generado por el compilador y empieza su ejecución. Sin importar si el programa fue creado con C#, con Visual Basic .NET o algún otro lenguaje de .NET el CLR lo lee y ejecuta.

Assembly

Cuando tenemos un programa escrito en un lenguaje de .NET y lo compilamos se genera el **assembly**. El assembly contiene el programa compilado en lo que conocemos como CIL y también información sobre todos los tipos que se utilizan en el programa.

CIL

Los programas de .NET no se compilan directamente en código ensamblador del compilador, en su lugar son compilados a un lenguaje intermedio conocido como CIL. Este lenguaje es leído y ejecutado por el runtime. El uso del CIL y el runtime es lo que le da a .NET su gran flexibilidad y su capacidad de ser multiplataforma.

El Framework de .NET tiene lo que se conoce como las **especificaciones comunes de lenguaje** o **CLS** por sus siglas en inglés, estas especificaciones son las guías que cualquier lenguaje que deseé usar .NET debe de cumplir para poder trabajar con el runtime. Una ventaja de esto es que si nuestro código cumple con las CLS podemos tener interoperabilidad con otros lenguajes, por ejemplo, es posible crear una librería en C# y un programador de Visual Basic .NET puede utilizarla sin ningún problema.

Uno de los puntos más importantes de estas guías es el **CTS** o **sistema de tipos comunes**. En los lenguajes de programación, cuando deseamos guardar información, ésta se coloca en una variable, las variables van a tener un tipo dependiendo del la información a guardar, por ejemplo, el tipo puede ser para guardar un número entero, otro para guardar un número con decimales y otro para guardar una frase o palabra. El problema con esto es que cada lenguaje guarda la información de manera diferente, algunos lenguajes guardan los enteros con **16 bits** de memoria y otros con **32 bits**; incluso algunos lenguajes como C y C++ no tienen un tipo para guardar las **cadenas** o frases.

Para solucionar esto el Framework de .NET define por medio del CTS cómo van a funcionar los tipos en su entorno. Cualquier lenguaje que trabaje con .NET debe de usar los tipos tal y como se señalan en el CTS. Ahora que ya conocemos los conceptos básicos, podemos ver cómo es que todo esto se une.

Cómo se crea una aplicación .NET

Podemos crear una aplicación .NET utilizando un lenguaje de programación, para este efecto será C#; con el lenguaje de programación creamos el código fuente del programa (instrucciones que le dicen al programa qué hacer).

Cuando hemos finalizado con nuestro código fuente, entonces utilizamos el compilador. El compilador toma el código fuente y crea un assembly para nosotros, este assembly tendrá el equivalente de nuestro código, pero escrito en CIL; esto nos lleva a otra de las ventajas de .NET: nuestro código puede ser optimizado por el compilador para la plataforma hacia la cual vamos a usar el programa, es decir que el mismo programa puede ser optimizado para un dispositivo móvil, una PC normal o un servidor, sin que nosotros tengamos que hacer cambios en él.

III .NET ES MULTIPLATAFORMA

El Framework de .NET se puede ejecutar en muchas plataformas, no solo en Windows. Esto significa que podemos programar en una plataforma en particular y si otra plataforma tiene el runtime, nuestro programa se ejecutará sin ningún problema. Un programa .NET desarrollado en Windows puede ejecutarse en Linux, siempre y cuando se tenga el runtime correspondiente.

Cuando nosotros deseamos invocar al programa, entonces el runtime entra en acción, lee el assembly y crea para nosotros todo el entorno necesario. El runtime empieza a leer las instrucciones CIL del assembly y conforme las va leyendo las compila para el microprocesador de la computadora en la que corre el programa; esto se conoce como **JIT** o **compilación justo a tiempo**. De esta manera conforme se avanza en la ejecución del programa se va compilando; todo esto ocurre de manera transparente para el usuario.

El Framework de .NET provee, para los programas que se están ejecutando, los servicios de **administración de memoria** y **recolector de basura**. En lenguajes no administrados como C y C++ el programador es responsable de la administración de memoria, en programas grandes esto puede ser una labor complicada, que puede llevar a errores durante la ejecución del programa. Afortunadamente lenguajes administrados como C# tienen un modelo en el cual nosotros como programadores ya no necesitamos ser responsables por el uso de la memoria. El recolector de basura se encarga de eliminar todos los objetos que ya no son necesarios, cuando un objeto deja de ser útil el recolector lo toma y lo elimina. De esta forma se liberan memoria y recursos.

El recolector de basura trabaja de forma automática para nosotros y ayuda a eliminar toda la administración de recursos y memoria que era necesaria en Win32. En algunos casos especiales como los archivos, las conexiones a bases de datos o de red se tratan de recursos no administrados, para estos casos debemos de indicar explícitamente cuando es necesario que sean destruidos.

Cómo conseguir un compilador de C#

Existen varias opciones de compiladores para C#, en este libro utilizaremos la versión de C# que viene con **Visual Studio 2010**, pero también es posible utilizar versiones anteriores. Este compilador, al momento de publicación de este libro, utiliza la versión 4.0 del Framework. La mayoría de los ejemplos deben poder compilarse y ejecutarse sin problema, aún utilizando versiones anteriores del Framework, esto es válido incluso para aquellos programadores que decidan trabajar con el llamado Mono, como alternativa libre y gratuita.

III EL COMPILADOR JIT

Al **compilador JIT** también se le conoce como **Jitter**. Forma parte del runtime y es muy eficiente, si el programa necesita volver a ejecutar un código que ya se ha compilado, el Jitter en lugar de volver a compilar, ejecuta lo ya compilado, mejorando de esta forma el desempeño y los tiempos de respuesta de cara al usuario.

Una mejor opción, en caso de que no podamos conseguir la versión profesional de Visual Studio 2010, es la versión **Express**. Esta versión es gratuita y se puede descargar directamente de Internet, lo único que se necesita es llevar a cabo un pequeño registro por medio de cualquier cuenta de **Hotmail** o **passport**.

Para descargar el compilador de **C# Express** de forma completamente gratuita, podemos acceder al sitio web que se encuentra en la dirección www.microsoft.com/express, dentro del portal de Microsoft. Para realizar esta descarga sólo será necesario completar un formulario de registro.

Una vez que hemos descargado el compilador, podemos proceder a llevar a cabo la instalación; esta tarea es muy similar a la instalación de cualquier otro programa de Windows, pero no debemos olvidar registrarlo antes de 30 días.

El ambiente de desarrollo

Una vez finalizada la instalación podemos iniciar el programa seleccionándolo desde el menú **Inicio** de Windows, veremos una ventana como la que aparece a continuación, que muestra la interfaz de uso de la aplicación.

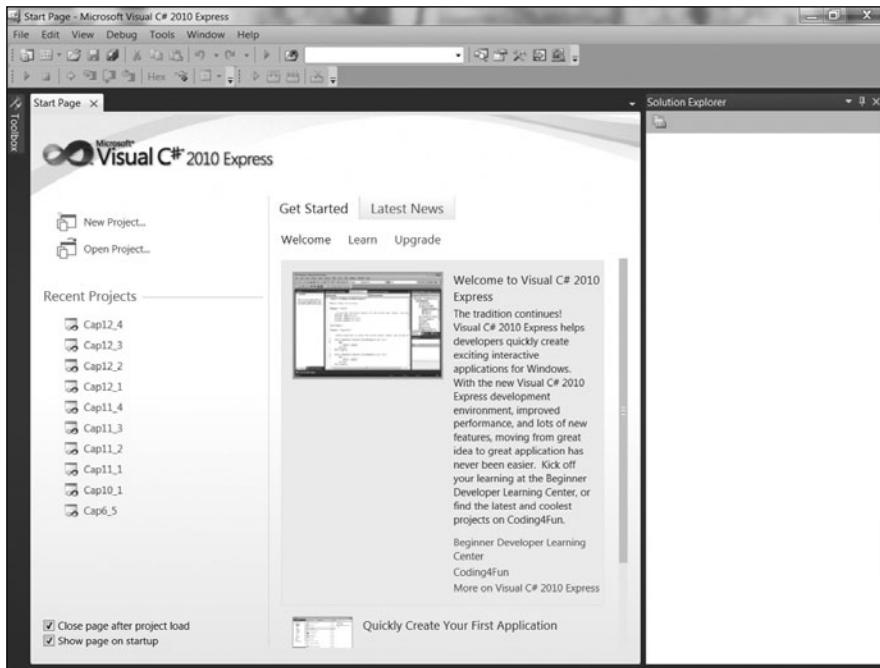


Figura 1. Aquí vemos la ventana principal de **C# Express 2010**. Esta interfaz aparece cada vez que iniciamos el programa.

Básicamente la interfaz está dividida en dos partes, en el centro tenemos la página de inicio, aquí encontramos enlaces a nuestros proyectos más recientes, pero también

podemos recibir comunicados y noticias sobre el desarrollo de software usando C#. En esta misma parte de la interfaz pero como páginas diferentes, tendremos las zonas de edición. Estas zonas de edición nos permitirán editar el código del programa, editar la interfaz de usuario, iconos y demás recursos.

Editar el código del programa es una actividad muy sencilla que no debe preocuparnos, si sabemos utilizar cualquier tipo de editor de textos como **Microsoft Word** o el **Bloc de notas de Windows**, entonces podemos editar el código del programa. Los demás editores también son bastante amigables para el usuario, pero no los necesitaremos para realizar los ejemplos mostrados en este libro.

Del lado derecho tenemos una ventana que se conoce como **Explorador de soluciones**, en esta misma área aparecerán otras ventanas que nos dan información sobre el proyecto que estamos desarrollando. Veremos los detalles mostrados por algunas de estas ventanas un poco más adelante en este libro.

En la parte inferior encontramos otra ventana, en esta zona generalmente aparecen los apartados que utilizará el compilador para comunicarse con nosotros. Por ejemplo, en esta zona veremos la ventana que nos indica los errores que tiene nuestro programa, en la parte superior aparecen los menús y las barras de herramientas.

Cómo crear nuestra primera aplicación

Para familiarizarnos más con C# Express, lo mejor es crear un primer **proyecto** y trabajar sobre él. Generaremos un pequeño programa que nos mande un mensaje en la consola, luego adicionaremos otros elementos para que podamos explorar las diferentes ventanas de la interfaz de usuario.

Para crear un proyecto tenemos que seleccionar el menú **Archivo** y luego de esto debemos hacer clic en la opción llamada **Nuevo Proyecto**, de esta forma veremos un cuadro de diálogo que muestra algunas opciones relacionadas.

En la parte central del cuadro de diálogo aparecen listados los diferentes tipos de proyectos que podemos crear, para este ejemplo debemos seleccionar el que indica aplicación de consola. En la parte inferior escribiremos el nombre de nuestro proyecto, que en este caso será **MiProyecto**. Cada nuevo proyecto que crearemos tendrá su propio nombre; después, simplemente oprimimos el botón **OK**.



ENCONTRAR CÓDIGO FUENTE RÁPIDAMENTE

Si por algún motivo llegamos a cerrar el editor del código fuente o no aparece en nuestra interfaz de usuario, la forma más sencilla de encontrarlo es por medio del Explorador de soluciones. Simplemente debemos dirigirnos al documento que representa nuestro código fuente y hacer doble clic en él, la ventana con el editor de código fuente aparecerá.

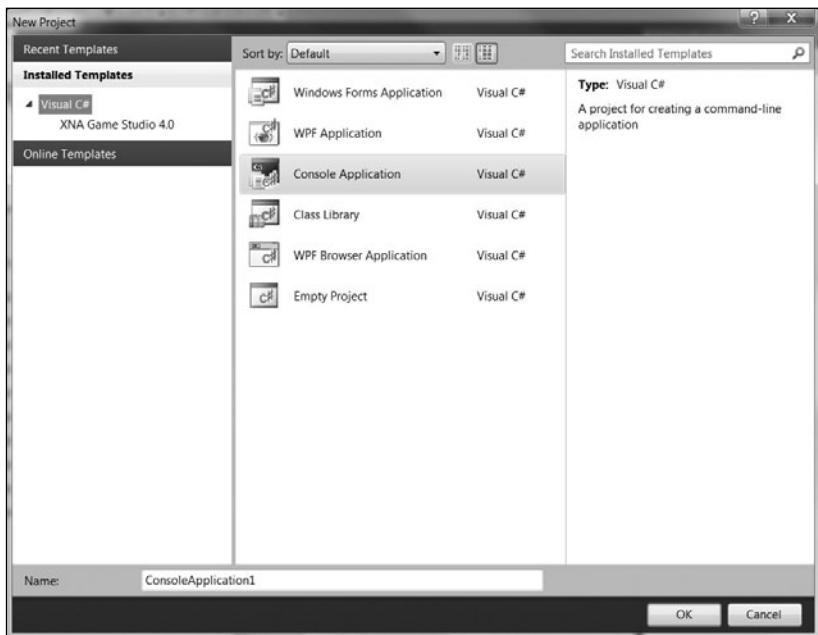


Figura 2. Éste es el cuadro de diálogo que usamos para crear un nuevo proyecto, gracias a sus opciones podemos generar diferentes tipos de proyectos.

En unos pocos segundos C# Express crea el proyecto para nosotros. Visual Studio y la versión Express hacen uso de **soluciones** y proyectos, una solución puede tener varios proyectos, por ejemplo, Office tiene diferentes productos como Word, Excel y PowerPoint. Office es una solución y cada producto es un proyecto. El proyecto puede ser un programa independiente o una librería, y puede tener uno o varios documentos, estos documentos pueden ser el código fuente y recursos adicionales. Podemos observar que nuestra interfaz de usuario ha cambiado un poco, las ventanas ya nos muestran información y también podemos ver que en la zona de edición se muestra el esqueleto para nuestro programa.

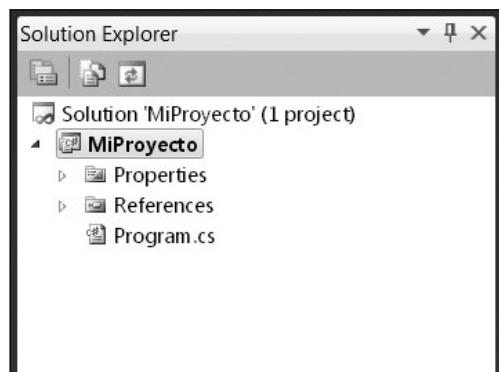


Figura 3. Esta figura nos muestra el panel derecho denominado **Solution Explorer**.

El **Explorador de soluciones** nos muestra la información de la solución de forma lógica, si observamos es como un pequeño árbol. En la raíz encontramos a la solución, cada proyecto que tengamos en esa solución será una rama, cada proyecto, a su vez, tendrá también sus divisiones. En nuestro caso vemos tres elementos, dos de esos elementos son carpetas, en una se guardan las propiedades del proyecto y en la otra las referencias (durante este libro no haremos uso de estas carpetas). El tercer elemento es un documento llamado **Program.cs**, éste representa al documento donde guardamos el código fuente de nuestra aplicación. Vemos que la extensión de los programas de C# es .CS.

En el área de edición podemos observar que tenemos un esqueleto para que, a partir de ahí, podamos crear nuestro propio programa. Para entender lo que tenemos ahí es necesario conocer un concepto: **namespace**. El namespace es una agrupación lógica, por ejemplo, todo el código que podemos tener relacionado con matemáticas puede quedar agrupado dentro del namespace de **Math**. Otro uso que tiene el namespace es el de resolver conflictos con los nombres, por ejemplo, supongamos que tenemos un proyecto muy grande y varios programadores trabajando en él. Es posible que ambos programadores crearan un método que tuviera el mismo nombre, esto nos genera un conflicto ya que el programa no podría saber cual versión utilizar. La forma de resolver esto es que cada programador tenga su propio namespace y hacer referencia al namespace correspondiente según la versión que deseáramos utilizar.

El Framework de .NET nos provee de varios namespaces donde tenemos miles de clases y métodos ya creados para nuestro uso. Cuando deseamos utilizar los recursos que se encuentran en un namespace programado por nosotros o por otros programadores, debemos hacer uso de un comando de C# conocido como **using**.

Como podemos ver en la parte superior del código, tenemos varios **using** haciendo referencia a los namespaces que necesita nuestra aplicación; si necesitáramos adicionar más namespaces, lo haríamos en esta sección.

Más abajo se está definiendo el namespace propio de nuestro proyecto, esto se hace de la siguiente manera:

```
namespace MiProyecto
{
}
```

El namespace que estamos creando se llama **MiProyecto**, como podemos ver el namespace usa {} como delimitadores, esto se conoce como un **bloque de código**, todo lo que se coloque entre {} pertenecerá al namespace; ahí es donde será necesario escribir el código correspondiente a nuestra aplicación.

Dentro del bloque de código encontramos la declaración de una **clase**, C# es un lenguaje orientado a objetos y por eso necesita que declaremos una clase para nuestra aplicación. La clase tiene su propio bloque de código y en nuestra aplicación se llamará **Program**. El concepto de clase se verá en el **Capítulo 10** de este libro.

Todos los programas necesitan de un punto de inicio, un lugar que indique dónde empieza la ejecución del programa, en C#, al igual que en otros lenguajes, el punto de inicio es la función **Main()**; esta función también tiene su propio bloque de código. Dentro de esta función generalmente colocaremos el código principal de nuestra aplicación, aunque es posible tener más funciones o métodos y clases. Las partes y características de las funciones se ven en el **Capítulo 5** de este libro.

Ahora es el momento de crear nuestra primera aplicación. Vamos a modificar el código de la función tal y como se muestra a continuación. Cuando estemos adicionando la sentencia dentro de **Main()**, debemos fijarnos que sucede inmediatamente después de colocar el punto.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace MiProyecto
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hola Mundo!");
        }
    }
}
```

Si logramos observar, cuando se escribió la palabra **Console** y luego el punto, apareció un recuadro listando diferentes elementos. Este cuadro se llama **autocomplete** y nos permite trabajar rápidamente y reducir la cantidad de errores de sintaxis. El cuadro de autocomplete nos muestra sugerencia de la palabra, comando, variable, etc. que se podría usar a continuación. Si lo deseamos, lo seleccionamos de esa lista, y él lo escribe por nosotros, conforme avancemos y nos familiaricemos con la programación de C# veremos que es de gran ayuda.

En nuestro ejemplo hemos colocado una sentencia que escribirá el mensaje **Hola Mundo** en la consola. Es un programa muy sencillo, pero nos permitirá aprender cómo llevar a cabo la compilación de la aplicación y ver el programa ejecutándose.

Para compilar la aplicación

Una vez que terminemos de escribir nuestro programa, podemos llevar a cabo la compilación, como aprendimos anteriormente esto va a generar el assembly que luego usará el runtime cuando se ejecute.

Para poder compilar la aplicación debemos seleccionar el menú de **Depuración** o **Debug** y luego **Construir Solución** o **Build Solution**. El compilador empezará a trabajar y en la barra de estado veremos que nuestra solución se ha compilado exitosamente.

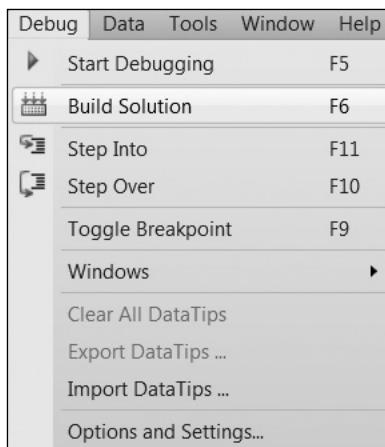


Figura 4. Para compilar nuestra aplicación, primero debemos construirla, seleccionando la opción adecuada desde el menú.

Para ejecutar la aplicación

Una vez que la compilación ha sido exitosa, podemos ejecutar nuestro programa y ver cómo trabaja. Para esto tenemos dos opciones: **ejecutar con depuración** y **ejecutar sin depurador**. En la versión Express únicamente aparece la ejecución con depuración, pero podemos usar la ejecución sin depurador con las teclas **CTRL+F5** o adicionándola usando el menú de herramientas.

El **depurador** es un programa que nos ayuda a corregir errores en tiempo de ejecución y también errores de lógica. En el **Capítulo 12** de este libro aprenderemos cómo utilizarlo. De preferencia debemos utilizar la ejecución sin depurador y hacer



INTERFAZ GRÁFICA

Éste es un libro de inicio a la programación con C#, por eso todos los programas se ejecutarán en la consola. Una vez comprendidos los conceptos principales que se enseñan, es posible aprender cómo se programan las formas e interfaz gráfica en C#. Este tipo de programación no es difícil, pero si requiere tener los conocimientos básicos de programación orientada a objetos.

uso de la ejecución con depuración únicamente cuando realmente la necesitemos. Ahora ejecutaremos nuestra aplicación, para esto oprimimos las teclas **CTRL+F5**.

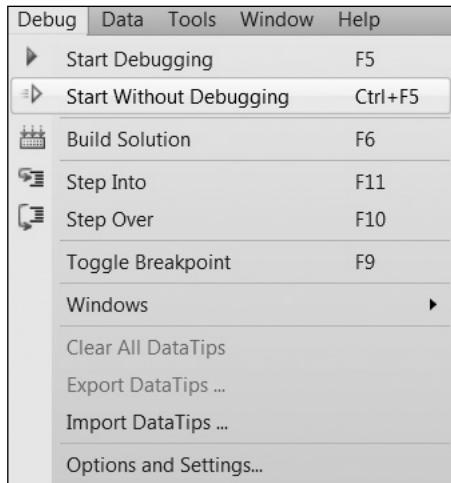


Figura 5. Con el menú *Debug/Start Without Debugging* nuestro programa se ejecutará.

Cuando el programa se ejecuta, aparece una ventana, a ella la llamamos consola y se encarga de mostrar la ejecución del programa. De esta forma podremos leer el mensaje que habíamos colocado en nuestro ejemplo.

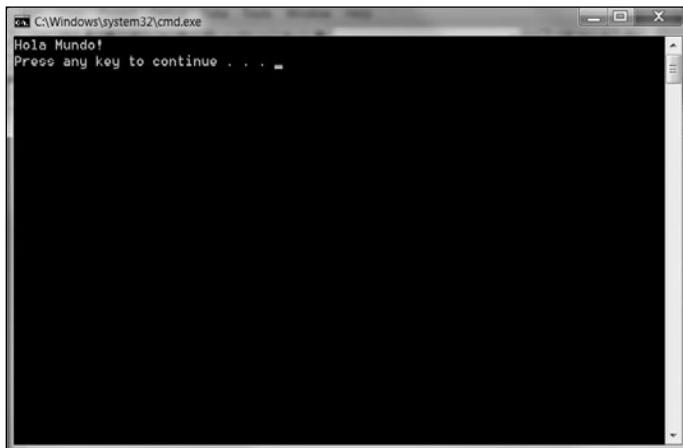


Figura 6. Ahora podemos ver nuestra aplicación ejecutándose en la consola.

Cómo detectar errores en un programa

En algunas ocasiones puede ocurrir que escribimos erróneamente el programa, cuando esto sucede, la aplicación no podrá compilarse ni ejecutarse; si esto llegara a ocurrir debemos cambiar lo que se encuentra mal.

Escribamos a propósito un error para que veamos cómo se comporta el compilador.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace MiProyecto
{
    class Program
    {
        static void Main(string[] args)
        {
            Consola.WriteLine("Hola Mundo!");
        }
    }
}
```

En el programa hemos cambiado **Console** por **Consola**, esto provocará un error; ahora podemos tratar de compilar nuevamente y ver lo que sucede.

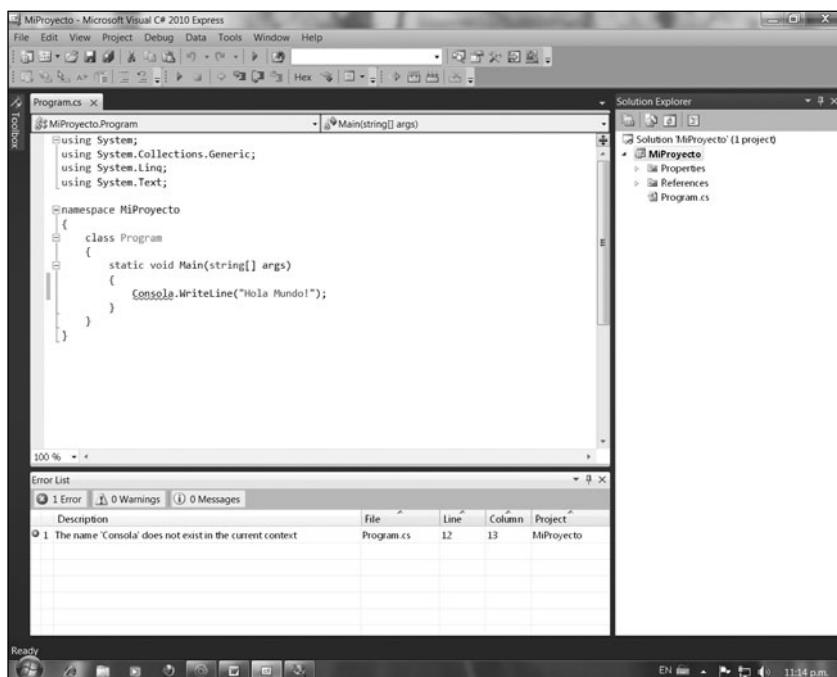


Figura 7. En esta imagen vemos que el programa tiene un error y por lo tanto no se puede compilar.

Como ya sabemos, la ventana que aparece en la parte inferior de nuestra interfaz de usuario es utilizada por el compilador para comunicarse, vemos que aparece una ventana que nos entrega un listado de errores; en el **Capítulo 12**, dedicado a la depuración, aprenderemos a utilizarla, ahora simplemente debemos saber que siempre es necesario resolver el primer problema de la lista y que podemos ir directamente al error haciendo doble clic con el mouse sobre él.

La vista de clases

Ahora revisaremos la **vista de clases**. En esta vista podemos obtener información sobre la solución que estamos creando, pero a diferencia del Explorador de soluciones, la información se encuentra ordenada en forma lógica.

Con esta vista podemos encontrar rápidamente los namespace de nuestra solución y dentro de ellos las clases que contienen; si lo deseamos, podemos ver los métodos que se encuentran en cada clase. Esta vista no solamente permite observar la información lógica, también nos da la posibilidad de navegar rápidamente en nuestro código.

En la versión Express no viene la opción previamente configurada, por lo que es necesario adicionar el comando al menú **Vista**.

Para mostrar la vista de clases debemos de ir al menú **Vista** y luego seleccionar **Vista de Clases**, una ventana aparecerá en nuestra interfaz de usuario.

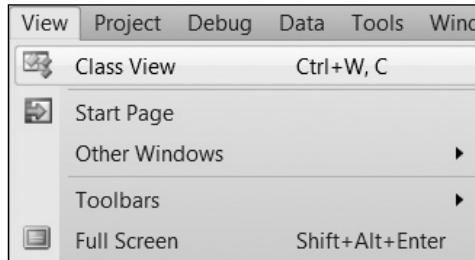


Figura 8. Para mostrar la vista de clases debemos de usar el menú **View**.

La ventana de la vista de clases está dividida en dos secciones, la sección superior nos muestra la relación lógica y jerárquica de los elementos mientras que en la parte inferior vemos los métodos que componen a alguna clase en particular.



.NET EN LINUX

Si deseamos trabajar bajo **Linux**, es posible utilizar **Mono**. Solamente debemos recordar que no siempre la versión .NET que maneja Mono es la más reciente de Microsoft. El sitio de este proyecto lo encontramos en la dirección web www.mono-project.com/Main_Page, desde ella podemos descargarlo en forma completamente gratuita.

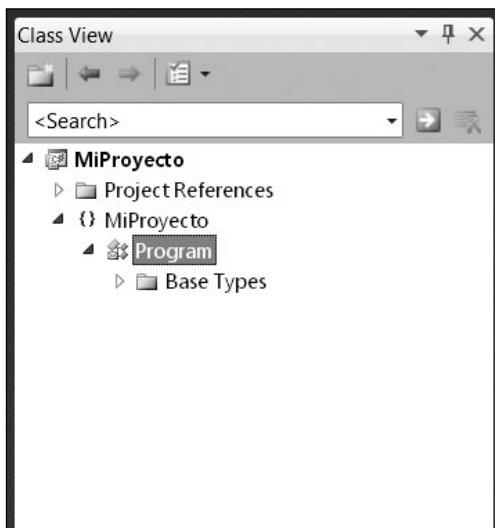


Figura 9. Aquí podemos observar a la vista de clases en el lado derecho. El árbol está abierto para poder ver los elementos que lo componen.

En la raíz del árbol encontramos al proyecto, éste va a contener las referencias necesarias y el namesapace de **MiProyecto**. Si tuviéramos más namespaces estos aparecerían ahí, al abrir el namespace de **Miproyecto** encontramos las clases que están declaradas dentro de él. En este caso solamente tenemos la clase **Program**, si la seleccionamos veremos que en la parte inferior se muestran los elementos que estén declarados en su interior. En nuestro ejemplo tenemos el método **Main()**.

Si en este momento damos doble clic en cualquiera de los métodos, nos dirigiremos automáticamente al código donde se define, como nuestro programa es muy pequeño, posiblemente no vemos la ventaja de esto, pero en programas con miles de líneas de código, el poder navegar rápidamente es una gran ventaja.

Configurar los menús del compilador

Para poder adicionar las opciones que nos hacen falta en los menús debemos seguir una serie de pasos muy sencillos. Nos dirigimos primero al menú **Tools** o **Herramientas** y seleccionamos la opción **Customize**.



Uno de los sitios más importantes que tenemos que visitar cuando desarrollamos para .NET es el llamado **MSDN**. En este sitio encontraremos toda la documentación y variados ejemplos para todas las clases y métodos que componen a .NET. El sitio se encuentra en la dirección web www.msdn.com.

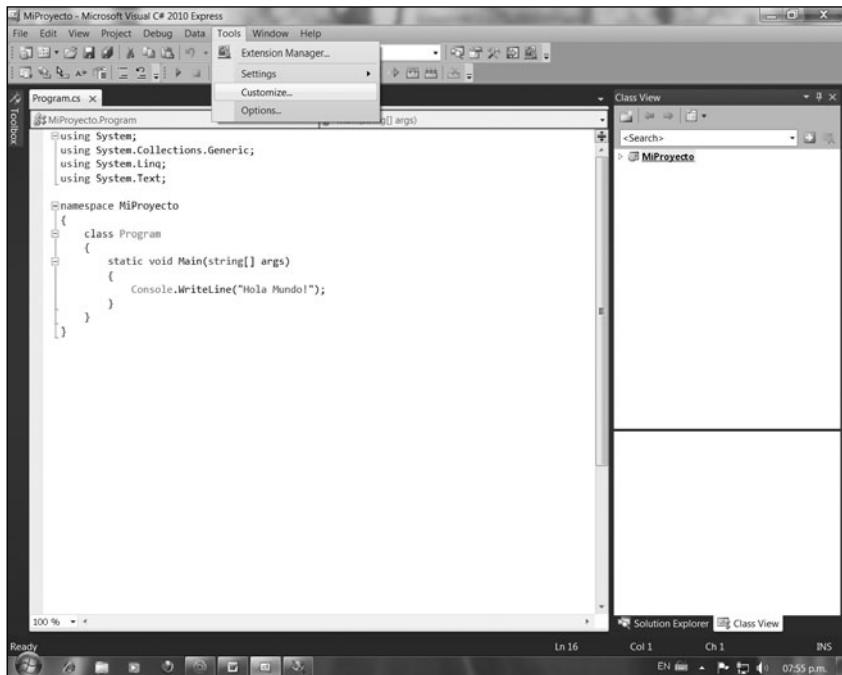


Figura 10. Es necesario seleccionar la opción de *Customize...* en el menú **Tools**.

Nos aparece un cuadro de diálogo, en el cual debemos seleccionar **Commands**, en esta sección podemos elegir el menú al cual queremos adicionar un comando, por ejemplo el menú **View**.

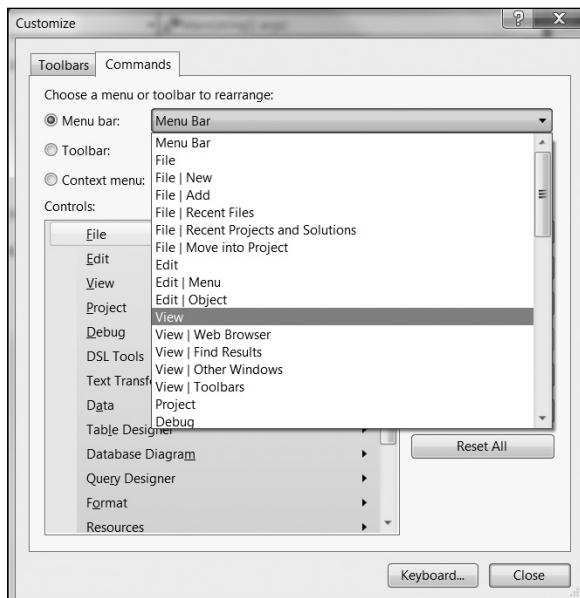


Figura 11. En esta ventana debemos seleccionar el menú que deseamos modificar.

Aparecen listados los comandos actuales del menú, damos clic en la zona donde deseamos que se inserte el nuevo comando y oprimimos el botón **Add Command**. Con esto aparece un nuevo cuadro de dialogo que muestra todos los posibles comandos que podemos adicionar clasificados en categorías.

En las categorías seleccionamos **View** y en los comandos **Class View**, este comando es el que nos permite tener la vista de clases. El comando de **Iniciar Sin Depurar** se encuentra en la categoría **Debug**.

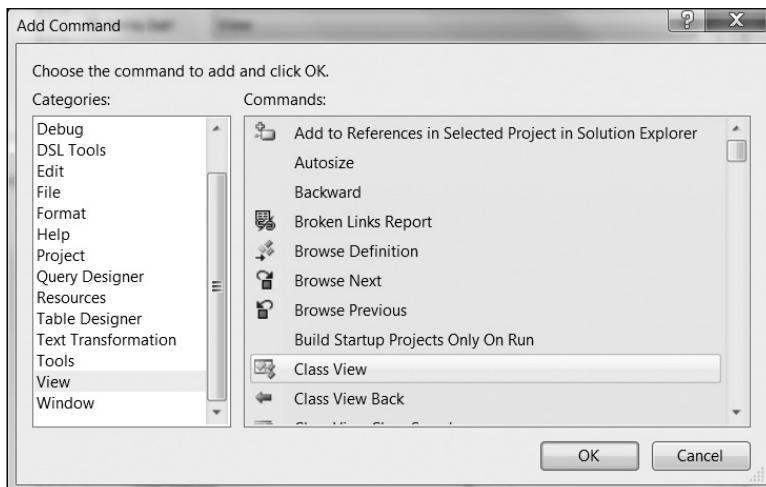


Figura 12. En esta ventana seleccionamos el comando que deseamos insertar.

Hasta aquí hemos analizado los conceptos básicos más importantes necesarios para comprender el funcionamiento de .NET, a partir del próximo capítulo comenzaremos con nuestro aprendizaje de programación y del lenguaje C#.

RESUMEN

Hace algunos años la programación para Windows resultaba complicada y requería de un alto grado de especialización. El Framework de .NET soluciona muchos de los problemas relacionados con la programación en Windows y la interoperabilidad, las aplicaciones de .NET se compilan a un assembly que contiene el programa escrito en CIL; éste es un lenguaje intermedio que lee el runtime cuando se ejecuta la aplicación. El CLR compila el CIL para el microprocesador según va siendo necesario, el uso de un runtime le da a .NET la flexibilidad de ser multiplataforma. Todos los lenguajes .NET deben cumplir con los lineamientos que encontramos en el CLS.



ACTIVIDADES

TEST DE AUTOEVALUACIÓN

- 1** ¿Cuál es la historia del desarrollo de las aplicaciones para Windows?
- 2** ¿Qué problemas ayuda a resolver .NET?
- 3** ¿Qué es un assembly?
- 4** ¿Cuál es la definición de CIL y CLR?
- 5** Describa qué hace el CLR con el assembly.
- 6** ¿Cuál es el significado de Jitter?
- 7** Mencione algunos compiladores de C# que podemos utilizar.
- 8** ¿Qué es el CTS?
- 9** ¿Porqué .NET puede ser multiplataforma?
- 10** ¿Qué otro lenguaje que use .NET existe?
- 11** ¿Cuál es la última versión de .NET?
- 12** ¿Qué otro lenguaje que use .NET existe?

EJERCICIOS PRÁCTICOS

- 1** Cree un proyecto nuevo que imprima su nombre en la ventana de la consola.
- 2** Coloque un error a propósito y vea cómo se comporta el compilador.
- 3** Explore el proyecto usando la vista de clases.
- 4** Experimente con el explorador de soluciones y vea la información que le provee.
- 5** Agregue más mensajes a la aplicación que ha creado.

Los elementos básicos de un programa

Llegó el momento de aprender cómo desarrollar programas de computadoras.

Algunas personas piensan que programar software es complejo y difícil, sin embargo, esto no es cierto, o lo fue hace muchísimo tiempo atrás.

Para ser programadores, sólo necesitamos conocer un lenguaje de programación y las técnicas necesarias para desarrollar programas. A lo largo de estos capítulos aprenderemos ambas.

Los lenguajes de programación	32
Los programas de computadora	32
Las variables	42
Operaciones aritméticas	49
Cómo pedirle datos al usuario	56
Cómo resolver problemas en la computadora	60
Resolución de problemas en la computadora	64
Resumen	69
Actividades	70

LOS LENGUAJES DE PROGRAMACIÓN

Cuando queremos comunicarnos con una persona, hacemos uso de un idioma en común para entendernos. Nosotros, con este libro, nos comunicamos por medio del lenguaje español, pero si llegara a visitarnos una persona de otro país donde no se hable español, entonces tendríamos un problema. La solución a este problema es muy sencilla: necesitamos aprender el idioma de esta persona para poder comunicarnos con ella. Lo mismo sucede con la computadora si queremos programarla, es necesario comunicarnos con ella, y la forma correcta de hacerlo es hablando su idioma. Entonces podemos decir que el idioma de la computadora se conoce como **lenguaje de programación** y el lenguaje de programación que aprenderemos es **C#**. Este lenguaje es sencillo y poderoso a su vez.

Una vez que sepamos el lenguaje de programación, entonces la computadora entenderá qué es lo que queremos que haga.

Los lenguajes de programación existen desde hace muchos años y son de diferentes tipos. Algunos han evolucionado y han surgido nuevos lenguajes como C#. Otros lenguajes son especializados para determinadas labores y muchos otros se han extinguido o han quedado obsoletos.

Los programas de computadora

Todos o casi todos sabemos cómo cocinar una torta, y de hecho, éste es un buen ejemplo para saber cómo hacer un programa de computadora. Cuando queremos cocinar una torta, lo primero que necesitamos son los ingredientes. Si falta alguno, entonces no tendremos una rica torta. Ya una vez que hemos preparado o recolectado nuestros ingredientes, seguimos la receta al pie de la letra. Ésta nos dice paso a paso qué es lo que tenemos que hacer.

Pero, ¿qué sucedería si no seguimos los pasos de la receta?, ¿qué sucede si ponemos los ingredientes primero en el horno y luego los revolvemos?, ¿el resultado es una torta? La respuesta a estas preguntas es evidente: el resultado no es una torta. De esto deducimos que para lograr algo, necesitamos una serie de pasos, pero también es necesario seguirlos en el orden adecuado. A esta serie de pasos la llamamos receta.

III CÓMO EVITAR ERRORES

Un error que se puede tener al iniciarnos con la programación de computadoras con C# es olvidar colocar ; al finalizar la sentencia. Si nuestro programa tiene algún error y todo parece estar bien, seguramente olvidamos algún punto y coma. Es bueno reconocer estos tipos de errores ya que nos permite reducir el tiempo de corrección necesario.

Cualquier persona puede pensar en una actividad y mencionar la lista de pasos y el orden en el que se deben cumplir. Podemos hacer sin problema la lista de pasos para cambiar el neumático del automóvil o para alimentar a los peces del acuario. Con esto vemos que no encontraremos problema alguno para listar los pasos de cualquier actividad que hagamos.

Ahora, seguramente nos preguntamos: ¿qué tiene esto que ver con la programación de computadoras? y la respuesta es simple. Cualquier persona que pueda hacer una lista de pasos puede programar una computadora. El programa de la computadora no es otra cosa que una lista de los pasos que la computadora tiene que seguir para hacer alguna actividad. A esta lista de pasos la llamamos **algoritmo**. Un algoritmo son los pasos necesarios para llevar a cabo una acción. Una característica importante del algoritmo es que tiene un punto de **inicio** y un punto de **fin**, lo que indica que los pasos se llevan a cabo de forma **secuencial**, uno tras otro. El algoritmo sería equivalente a la receta de la torta y los ingredientes necesarios generalmente serán los datos o la información que usaremos.

Hagamos un pequeño programa para ver cómo funcionaría esto. Para esto creamos un proyecto, tal y como vimos en el **Capítulo 1**. Nuestro programa mostrará los pasos para hacer una torta o un pastel. Colocamos el siguiente código en nuestro programa y veremos cómo se corresponde con lo que hemos aprendido.

```
using System;
using System.Collections.Generic;
using System.Text;

namespace AplicacionBase
{
    class Program
    {
        // Esta es la función principal del programa
        // Aquí inicia la aplicación
        static void Main(string[] args)
```

III LAS FUNCIONES EN C#

Las funciones son secciones del programa en las que podemos tener código especializado. La función **Main()** es invocada por el sistema operativo cuando queremos que nuestro programa se inicie. Es la única función que deben tener todos los programas que desarrollemos.

```

    {
        Console.WriteLine("1- Precalentar el horno");
        Console.WriteLine("2- Batir margarina con azucar");
        Console.WriteLine("3- Agregar huevos");
        Console.WriteLine("4- Cernir harina");
        Console.WriteLine("5- Agregar a la mezcla y leche");
        Console.WriteLine("6- Hornear por 40 minutos");
        Console.WriteLine("7- Decorar y comer");
    }
}

```

C# es un **lenguaje orientado a objetos** y necesita que definamos una **clase** para poder crear un programa. En este momento no veremos las clases, pero regresaremos a ellas en un capítulo posterior. Lo primero que debemos encontrar es una **función** que se llama **Main()**, que es muy importante, como vimos en el **Capítulo 1**.

Ya sabemos que los algoritmos necesitan un inicio y un fin. En nuestro programa el punto donde siempre inicia es la función **Main()**. Esta función indica el punto de arranque de nuestro programa y aquí colocaremos los pasos que debe seguir nuestro programa en el orden adecuado.

Para indicar todos los pasos que pertenecen a **Main()** necesitamos colocarlos dentro de su **bloque de código**. Un bloque de código se define entre llaves **{ }**. Todo lo que se encuentre en ese bloque de código pertenece a **Main()**, lo que se encuentre afuera de él, no pertenece.

Es importante que notemos que la **clase** también tiene un bloque de código y **Main()** se encuentra adentro de él. De igual forma, el **namespace** también tiene su propio bloque de código y la clase se encuentra adentro, aunque por ahora sólo nos concentraremos en el bloque de código de **Main()**.

Un error común cuando se crean bloques de código es olvidar cerrarlos. Como recomendación, una vez que abrimos un bloque de código con **{**, inmediatamente debemos de colocar el cierre con **}** y luego programar su contenido. De esta forma, no

III ASIGNACIÓN DE VALORES FLOTANTES

Cuando colocamos un valor con decimales en C# automáticamente se lo interpreta como de tipo **double**. Sin embargo, si necesitamos colocar ese valor en una variable de tipo **float**, debemos usar el sufijo **f**. Éste le dice a C# que use ese valor como flotante. Ejemplo:

resultado = a * 3.57f;

olvidaremos cerrarlo. Dentro de **Main()** encontramos los pasos de nuestro algoritmo. En este momento nos pueden parecer extraños, pero aprenderemos qué significan. Tenemos que recordar que debemos aprender el idioma de la computadora y éstas son una de sus primeras frases.

Lo que encontramos adentro de la función **Main()** es una serie de **sentencias**. Las sentencias nos permiten colocar instrucciones que nuestro programa pueda ejecutar, que son utilizadas para mostrar un mensaje en la pantalla. El mensaje que nosotros queremos mostrar en la pantalla se muestra al invocar el método **WriteLine()**. Los métodos son similares a las funciones, existen adentro de las clases y nos permiten hacer uso de las funcionalidades internas. Este método pertenece a la clase **Console**, y en esa clase encontraremos todo lo necesario para que podamos trabajar con la consola, ya sea colocando mensajes o leyendo la información proporcionada por el usuario. Como **WriteLine()** pertenece a **Console** cuando queremos utilizarlo, debemos escribir:

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

WriteLine() necesita cierta información para poder trabajar. Esta información es el mensaje que deseamos mostrar. A los métodos se les pasa la información que necesitan para trabajar por medio de sus **parámetros**. Los parámetros necesarios se colocan entre **paréntesis()**. **WriteLine()** necesitará un parámetro de tipo **cadena**.

Una **cadena** es una colección de caracteres, es decir, letras, números y signos. La cadena se delimita con **comillas dobles** “”. En el ejemplo, el mensaje que queremos que se muestre es: **Hola**. Por eso pasamos el parámetro de tipo cadena “**Hola**”.

Las sentencias se finalizan con **punto y coma** ;. Esto es importante y no debemos olvidarlo ya que este carácter en particular es reconocido por C# para indicar que la sentencia escrita hasta allí, ha finalizado.

En cada una de las sentencias invocadas hasta aquí por nuestro programa, hemos enviado un mensaje tras otro con cada paso necesario para cocinar nuestro pastel o nuestra torta. Por último, sólo resta que ejecutemos el programa escrito y veamos el resultado que aparece en la pantalla.

III PROBLEMAS CON LOS MÉTODOS

Cuando se invoca un método es necesario no solamente colocar su nombre, sino que también se deben poner los paréntesis. Éstos se deben poner aun cuando el método no necesite parámetros. No hay que olvidar usar el paréntesis de cierre). Si olvidamos colocar los paréntesis, el compilador nos marcará errores y no podremos ejecutar el programa.

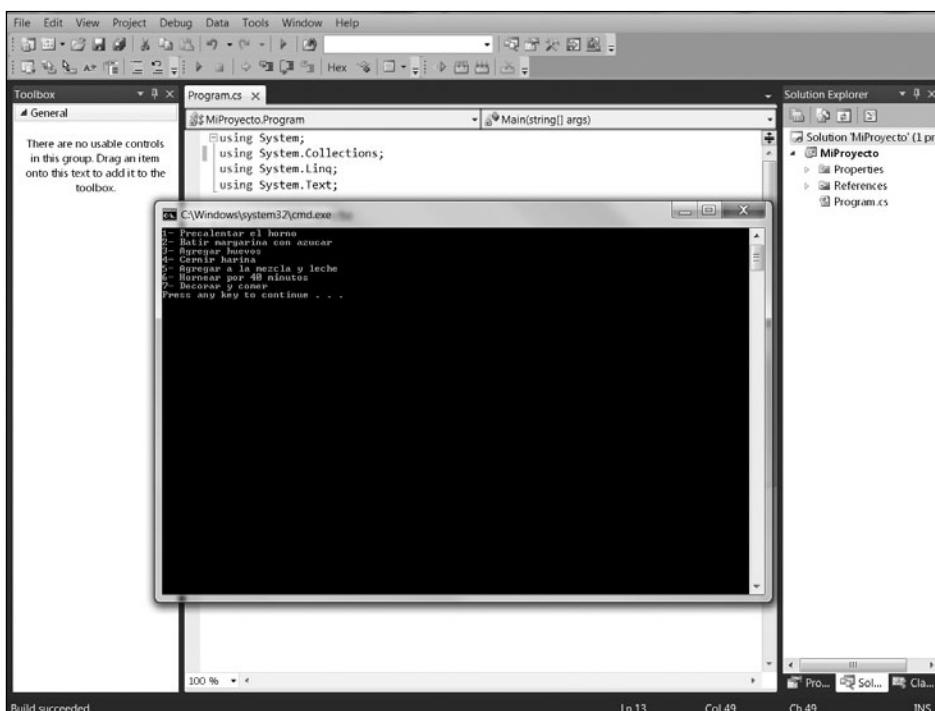


Figura 1. Podemos observar los mensajes que enviamos a la pantalla por medio del método `WriteLine()`.

Podemos observar que la ejecución ha mostrado los mensajes en el orden en el que los hemos puesto. Ahora podemos hacer un pequeño experimento. Cambiaremos el orden de las sentencias y observaremos qué es lo que sucede. El programa quedará de la siguiente forma:

```

using System;
using System.Collections.Generic;
using System.Text;

namespace AplicacionBase
{
    class Program
    {
        // Esta es la funcion principal del programa
        // Aqui inicia la aplicacion
        static void Main(string[] args)
        {
            Console.WriteLine("1- Precalentar el horno");

```

```

        Console.WriteLine("2- Batir margarina con azucar");
        Console.WriteLine("6- Hornear por 40 minutos");
        Console.WriteLine("5- Agregar a la mezcla y leche");
        Console.WriteLine("7- Decorar y comer");
        Console.WriteLine("3- Agregar huevos");
        Console.WriteLine("4- Cernir harina");
    }
}
}

```

Ejecutémoslo y veamos qué ocurre.

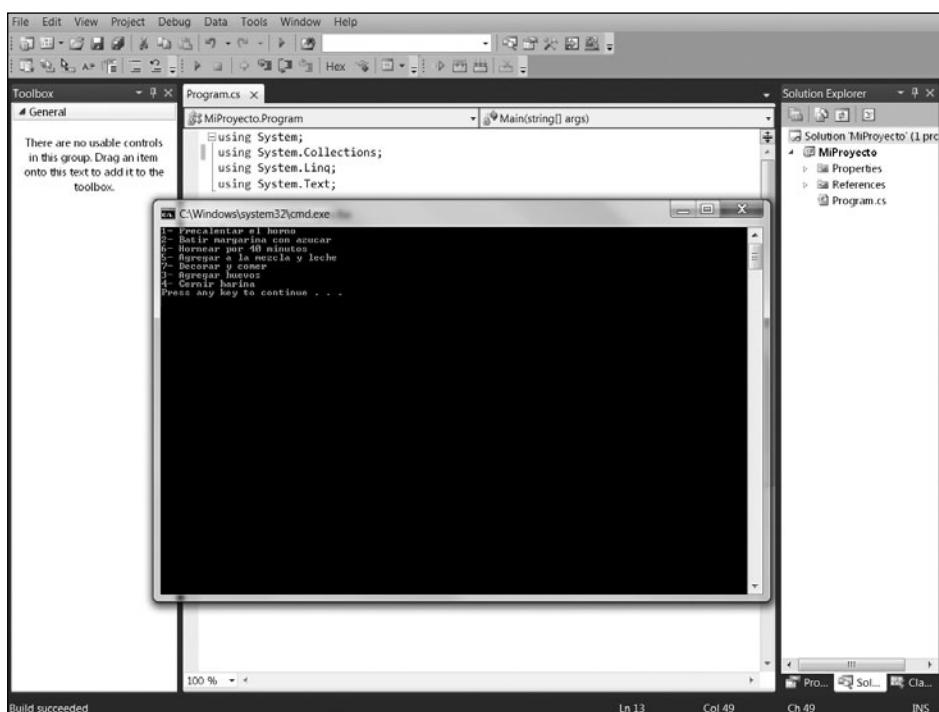


Figura 2. Aquí vemos que los mensajes se colocan en el orden en que se encuentran en la función *main()* y no en el orden en que esperábamos.

Podemos observar que cuando se ejecuta, las sentencias se ejecutan en el mismo orden como están en el código del programa, no se ejecutan como nosotros sabemos que se deben de ejecutar. Este punto es importante, ya que nos muestra que colocar las sentencias del programa en el orden de ejecución correcto es responsabilidad del programador. La computadora no puede leernos la mente y saber qué es lo que realmente deseamos, ésta sólo ejecuta fielmente las sentencias

en el orden en que las colocamos. Si en la vida real primero horneamos y luego batimos, no podemos esperar obtener un pastel o una torta. Por lo tanto, si en la computadora no ponemos las sentencias del programa en el orden adecuado, no podemos esperar tener un programa funcional.

Errores en los programas

Una parte importante de la programación es reconocer los errores que podemos tener con el fin de corregirlos y permitir que el programa trabaje correctamente. Podemos tener dos tipos de errores: los **errores de sintaxis** y los **errores de lógica**. Los errores de sintaxis son fáciles de corregir y se deben a que escribimos en el orden incorrecto las sentencias del programa. Estos errores se muestran cuando el programa se compila, y de esa forma se pueden buscar fácilmente para corregirlos. Los errores de lógica se deben a que hemos creado nuestro algoritmo en forma errónea y aunque todas las sentencias estén correctas, el programa no lleva a cabo lo que deseamos. Estos errores son más difíciles de corregir y la mejor forma de evitarlos es por medio de un análisis previo adecuado del problema a resolver. Más adelante aprenderemos cómo llevar a cabo este análisis.

Empecemos a reconocer los errores de sintaxis y los mensajes que da el compilador cuando los tenemos. El reconocer estos mensajes y cómo fueron ocasionados nos permitirá resolverlos rápidamente en el futuro. Para esto empezaremos a hacer unos experimentos. En el código de nuestro programa, en cualquiera de las sentencias, eliminamos el punto y coma que se encuentra al final, y luego lo compilamos para ver qué sucede. Debido a que hemos eliminado el punto y coma, tenemos un error de sintaxis y el compilador nos dice que el programa no se ha compilado exitosamente.

Dentro de nuestro compilador se mostrará la ventana de lista de errores. En esta ventana se colocan los errores encontrados por el compilador. Siempre es conveniente solucionar el problema que se encuentra en la parte superior de la lista y no el último, debido a que muchas veces, solucionando el primero, quedan solucionados otros que arrastraban su problema. El mensaje que vemos nos indica que se esperaba punto y coma, tal y como se ve en la **figura 3**.

III ERROR COMÚN DE LAS CADENAS

Algunas veces sucede que olvidamos colocar la cadena entre comillas o que solamente se ha colocado la comilla del inicio. Es importante no olvidar colocar las cadenas siempre entre las dos comillas. Si no lo hacemos de esta forma, el compilador no puede encontrar correctamente la cadena o dónde finaliza. La sentencia nos marcará un error de compilación.

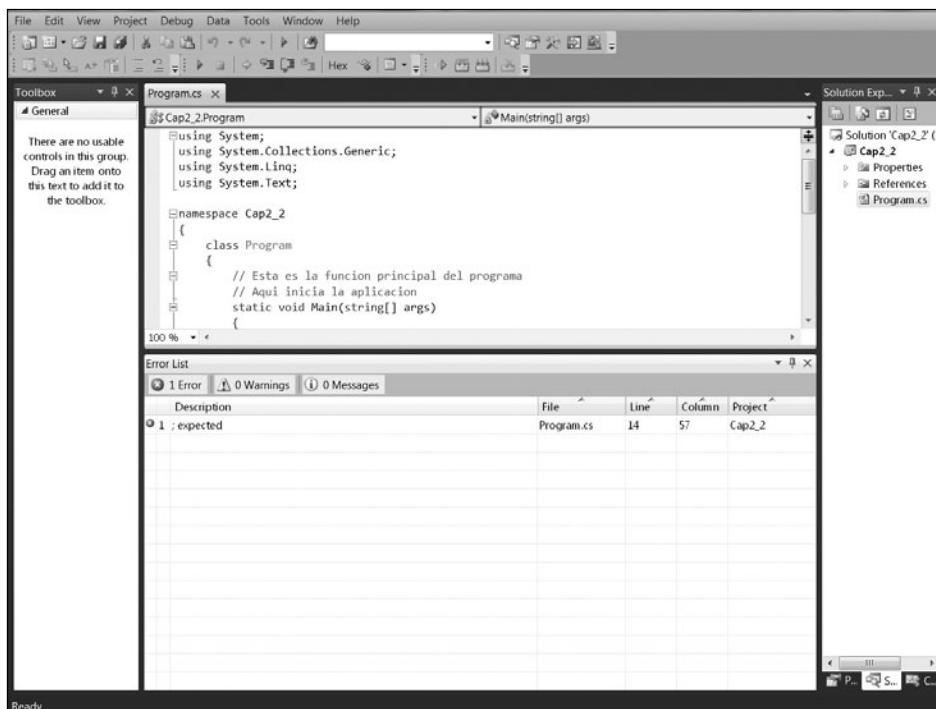


Figura 3. Podemos ver la ventana con la lista de errores y el error de sintaxis encontrado por el compilador.

La lista de errores nos da una descripción del error y también nos permite ver el número de línea donde se encuentra el error. Una forma sencilla de llegar a éste es simplemente hacer doble clic sobre la descripción del error, así el entorno C# nos llevará directamente a dónde se encuentra éste. Coloquemos nuevamente el punto y coma en el lugar donde estaba en un principio y compilemos. En esta ocasión, el programa puede compilar y ejecutar sin problemas, en base a la corrección que hemos realizado con antelación. El error se encuentra solucionado. La solución que nos brinda C# para corregir errores es práctica y sencilla de implementar. Ahora experimentemos otro error. A la primera sentencia le borramos el paréntesis de cierre y luego compilamos para ver qué ocurre:

III ERRORES CON LOS NOMBRES DE VARIABLES

Un error común que los principiantes hacen cuando nombran sus variables es que olvidan cómo nombraron a la variable y la escriben a veces con minúscula y a veces con mayúscula. Esto hace que se produzcan errores de lógica o de sintaxis. Para evitarlo, podemos utilizar una tabla que contenga la información de las variables de nuestro programa.

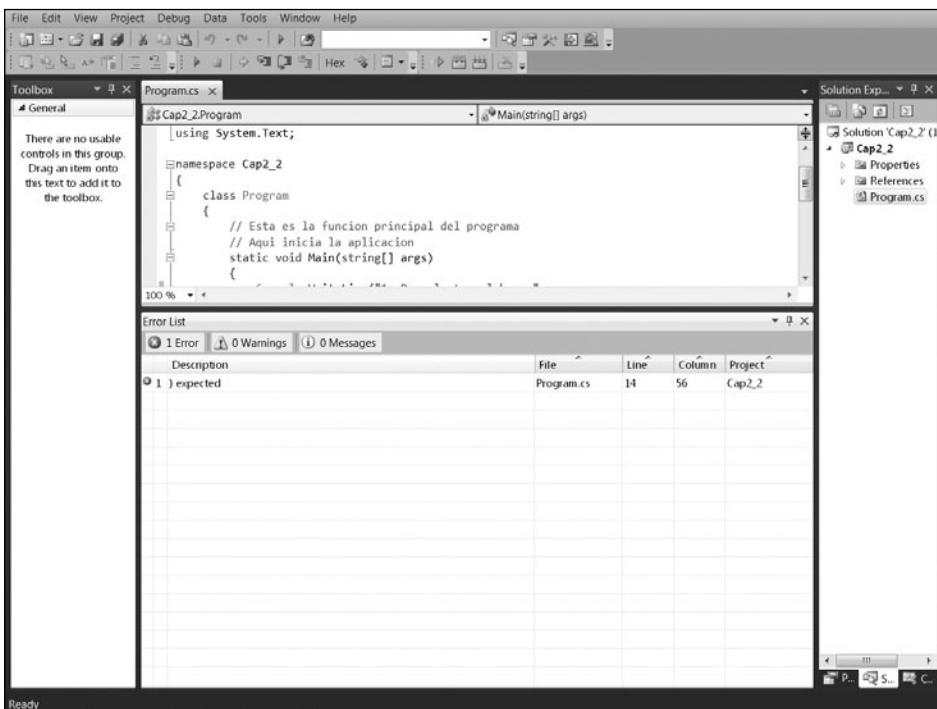


Figura 4. Aquí vemos el mensaje de error generado por no cerrar el paréntesis.

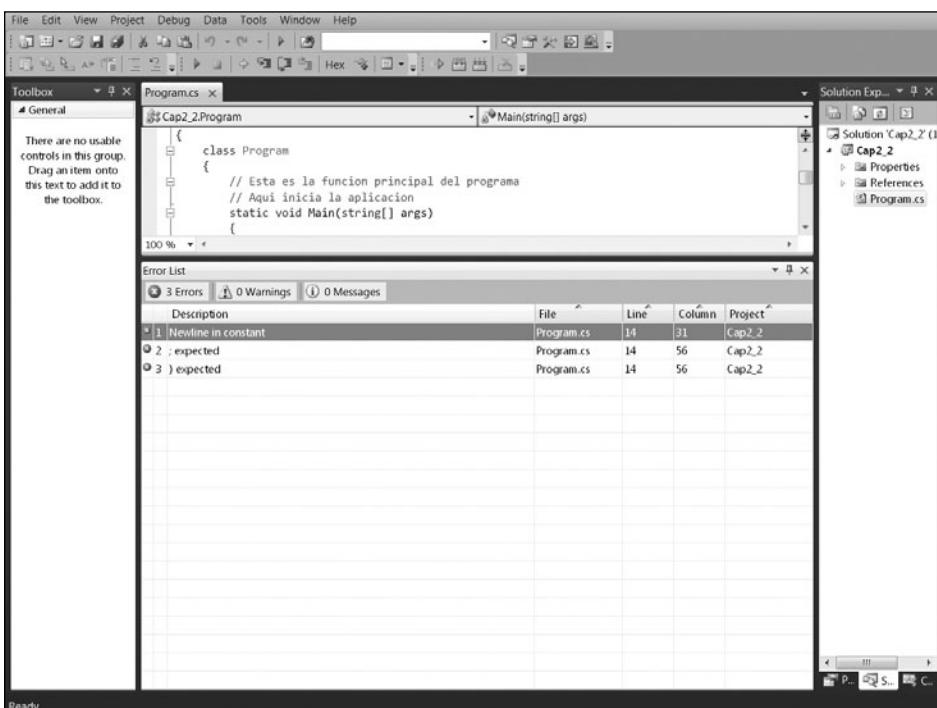


Figura 5. Ahora nuestra lista de errores muestra tres errores en el programa.

Corrijamos el error y compilemos de nuevo. Todo debe de estar bien. Pero lo que haremos ahora es borrar las comillas de cierre de la cadena en la primera sentencia. Este error nos presenta un escenario más interesante y nos conviene entender qué es lo que sucede. Compilemos el programa y veamos nuestra lista de errores. Ésta será mucho más extensa que las anteriores:

Aquí ha sucedido algo interesante, a pesar de que solamente hemos creado un error: en la lista de errores aparecen tres errores. Esto se debe a que un error puede ocasionar errores extras en el código siguiente. Al no cerrar la cadena, el compilador no sabe dónde termina la sentencia y cree que el paréntesis de cierre y el punto y coma forman parte de la cadena. Por eso también nos muestra que faltan el paréntesis y el punto y coma.

Debido a que los errores pueden ser en cadena, como en este ejemplo, siempre necesitamos corregir el error que se encuentra en primer lugar. Si nosotros intentáramos corregir primero el error del punto y coma, no resultaría evidente ver cuál es el error en esa línea, ya que veríamos que el punto y coma sí está escrito.

A veces sucede que nuestro programa muestra muchos errores, pero cuando corregimos el primer error, la gran mayoría desaparece. Esto se debe a factores similares a los que acabamos de ver.

La diferencia entre los métodos `Write()` y `WriteLine()`

Hemos visto que el método `WriteLine()` nos permite mostrar un mensaje en la consola, pero existe un método similar que se llama `Write()`. La diferencia es muy sencilla. Después de escribir el mensaje `WriteLine()` inserta un salto de línea, por lo que lo próximo que se escriba aparecerá en el renglón siguiente. Por su parte, el método `Write()` no lleva a cabo ningún salto de línea y lo siguiente que se escriba será en la misma línea. Veamos todo esto en el siguiente ejemplo:

```
using System;
using System.Collections.Generic;
using System.Text;

namespace AplicacionBase
{
    class Program
    {
        // Esta es la función principal del programa
        // Aquí inicia la aplicación
        static void Main(string[] args)
        {
            Console.WriteLine("Mensaje 1 ");
            Console.Write("Mensaje 2 ");
        }
    }
}
```

```

        Console.WriteLine("Mensaje 2 ");
        Console.WriteLine("Mensaje 3 ");
        Console.WriteLine("Mensaje 4 ");
        Console.WriteLine("Mensaje 5 ");

    }

}

}

```

Al ejecutarlo, vemos cómo en efecto después de usar **Write()** lo siguiente que se escribe se coloca en el mismo renglón, pero al usar **WriteLine()** lo siguiente aparece en el renglón siguiente. Esto es un ejemplo claro de lo explicado anteriormente:

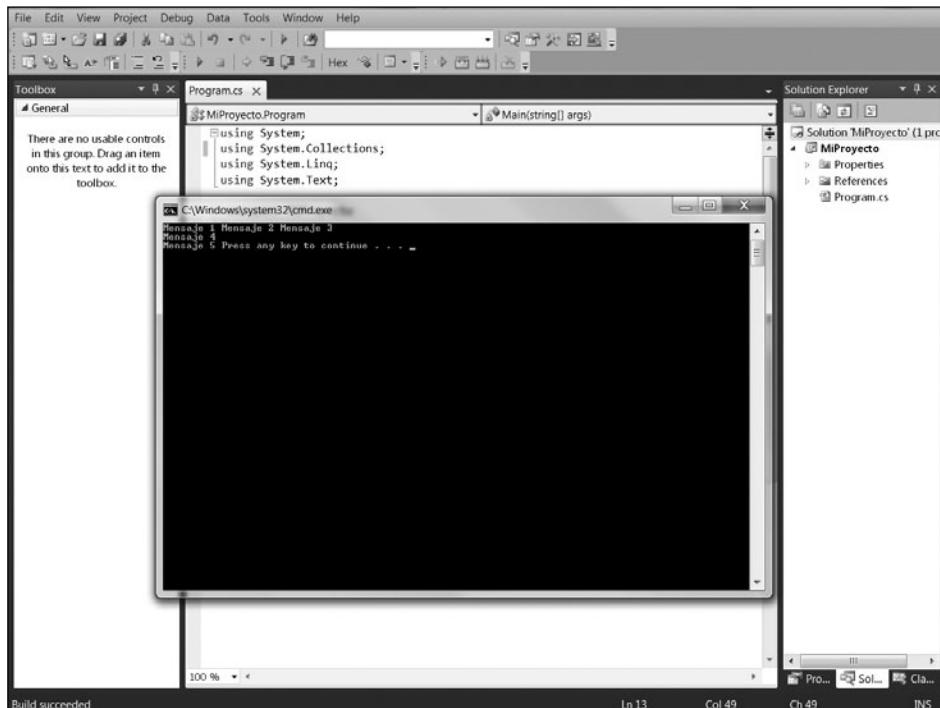


Figura 6. Podemos observar la diferencia entre los métodos **Write()** y **WriteLine()**.

Las variables

Las computadoras no solamente escriben mensajes en la pantalla, sino que deben hacer cosas útiles y para poder hacerlas necesitan información. La información debe almacenarse y manipularse de alguna forma.

Cuando queremos guardar algo en la vida cotidiana podemos utilizar cajas, por lo que cualquier cosa que deseamos almacenar puede ser colocada en una caja de

cartón, por ejemplo. Sin embargo, el tamaño de la caja dependerá de lo que deseamos almacenar. Unos zapatos pueden caber en una caja pequeña, pero en esa caja no podemos colocar un televisor. Si empezamos a almacenar muchas cosas en cajas, pronto tendremos demasiadas y será difícil poder encontrar lo que hemos guardado. Entonces, una forma de solucionar esto es por medio de etiquetas. A cada caja le ponemos una etiqueta con el contenido y de esta forma podemos encontrar rápidamente lo que buscamos.

Las **variables** en la computadora funcionan de manera similar a nuestras cajas de almacenaje. Las podemos imaginar como pequeñas cajas que existen en la memoria de la computadora y su tamaño dependerá de la información que deben guardar. Esto se conoce como tipo de la variable. Para poder acceder a esa caja o variable le ponemos un nombre que sería la etiqueta con la que la identificamos.

Para hacer uso de una variable, lo primero que tenemos que hacer es **declararla**. La declaración de éstas es algo muy sencillo. Como primer paso tenemos que colocar el **tipo** y luego el **nombre**. Las variables en C# deben nombrarse de acuerdo con unas recomendaciones sencillas:

- Los nombres de las variables deben empezar con letra.
- Es posible colocar números en los nombres de las variables, pero no empezar el nombre con ellos.
- Los nombres de las variables no pueden llevar signos a excepción del guión bajo _ .
- Las variables no pueden llevar acentos en sus nombres.

Cuando nombramos las variables, hay que tener en cuenta que C# es **sensible** a las **mayúsculas** y **minúsculas**, por lo que una variable llamada **costo** no es la misma que otra variable llamada **COSTO** u otra llamada **Costo**.

Es recomendable nombrar a las variables con nombres que hagan referencia a la información que guardarán. Si nos acostumbramos a hacer esto desde que empezamos a programar, evitaremos muchos problemas en el futuro y será mucho más sencillo corregir nuestros programas cuando tengan errores.

Veamos un ejemplo de cómo podemos declarar una variable que guarde valores **nnuméricos enteros**. El tipo que guarda estos valores se conoce como **int**.

III SELECCIÓN DEL TIPO

Es importante conocer los tipos y la información que pueden guardar, ya que esto nos permitirá guardar la información necesaria y utilizar la menor cantidad de memoria. Podemos aprender los rangos de los tipos o imprimir una tabla y tenerla a mano. Con la práctica podremos utilizar los tipos sin tener que verificar los rangos constantemente.

```
int costo;
```

Vemos que al final de la sentencia hemos colocado el punto y coma. Si necesitamos declarar más variables lo podemos hacer de la siguiente forma:

```
int costo;
int valor;
int impuesto;
```

Pero también es posible declarar las variables en una sola línea. Para esto simplemente sepáramos los nombres de las variables con comas. No hay que olvidar colocar el punto y coma al final de la sentencia.

```
int costo, valor, impuesto;
```

C# nos provee de muchos tipos para poder usar con nuestras variables, que podemos conocer en la siguiente tabla:

TIPO	INFORMACIÓN QUE GUARDA
bool	Es una variable booleana, es decir que solamente puede guardar los valores verdadero o falso (true o false) en términos de C#.
byte	Puede guardar un byte de información. Esto equivale a un valor entero positivo entre 0 y 255.
sbyte	Guarda un byte con signo de información. Podemos guardar un valor entero con signo desde -128 hasta 127.
char	Puede guardar un carácter de tipo Unicode.
decimal	Este tipo puede guardar un valor numérico con decimales. Su rango es desde ±1.0 ? 10?28 hasta ±7.9 ? 1028.
double	También nos permite guardar valores numéricos que tengan decimales. El rango aproximado es desde ±5.0 ? 10?324 hasta ±1.7 ? 10 308.

III FORMA CORRECTA DE HACER UNA ASIGNACIÓN

Un punto importante a tener en cuenta es que la asignación siempre se lleva a cabo de derecha a izquierda. Esto quiere decir que el valor que se encuentra a la derecha del signo igual se le asigna a la variable que se encuentra a la izquierda del signo igual. Si no lo hacemos de esta forma, el valor no podrá ser asignado o será asignado a la variable incorrecta.

TIPO	INFORMACIÓN QUE GUARDA
float	Otro tipo muy utilizado para guardar valores numéricos con decimales. Para este tipo el rango es desde $\pm 1.5 \times 10^{-45}$ hasta $\pm 3.4 \times 10^{38}$.
int	Cuando queremos guardar valores numéricos enteros con signo, en el rango de -2,147,483,648 hasta 2,147,483,647.
uint	Para valores numéricos enteros positivos, su rango de valores es desde 0 hasta 4,294,967,295.
long	Guarda valores numéricos enteros realmente grandes con un rango desde -9,223,372,036,854,775,808 hasta 9,223,372,036,854,775,807.
ulong	Guarda valores numéricos enteros positivos. Su rango de valores varía desde 0 hasta 18,446,744,073,709,551,615.
short	Guarda valores numéricos enteros, pero su rango es menor y varía desde -32,768 hasta 32,767.
ushort	Puede guardar valores numéricos enteros positivos con un rango desde 0 hasta 65,535.
string	Este tipo nos permite guardar cadenas.

Tabla 1. Ésta es la tabla de los tipos más utilizados en C#.

Es útil para seleccionar el tipo adecuado para las variables.

Una vez que hemos declarado la variable, debemos **inicializarla**. Inicializar una variable es asignarle un valor por primera vez. Esto nos permite darle un valor inicial que puede ser útil en la ejecución de nuestro programa (no sólo podemos asignarle a una variable un valor fijo, sino también puede ser desde un texto ingresado por el usuario o desde el registro de una base de datos).

Para asignarle un valor a una variable, ya sea durante la inicialización o durante el transcurso del programa, usamos un **operador de asignación**, el signo **igual**. Veamos a continuación un ejemplo de cómo asignar valores:

```
costo = 50;
valor = 75;
impuesto = 10;
```

En este caso, la variable **costo** almacena el valor de 50, la variable **valor** almacena el 75, y la variable **impuesto** guarda el valor 10 .



LOS COMENTARIOS Y EL COMPILADOR

Los comentarios solamente nos sirven a nosotros como seres humanos para facilitarnos a comprender el código del programa. El compilador ignora todos los comentarios y no los toma en cuenta en el momento de la compilación. Los comentarios no hacen el programa ni más lento ni más rápido. Sin embargo, sí son muy útiles cuando nosotros vemos el código.

Una asignación no válida sería la siguiente:

```
50 = costo;
```

En este caso nada queda asignado a la variable **costo**. Siempre la variable que recibe el valor se coloca a la izquierda del signo igual.

Si lo necesitamos, es posible inicializar la variable al momento de declararla. Esto es muy cómodo y si lo hacemos así evitamos olvidar la inicialización de la variable. Para hacerlo colocamos el tipo de la variable seguido del nombre a usar en la variable e inmediatamente con el operador de asignación colocamos el valor.

```
int costo = 50;
```

Comentarios en el programa

Los programas de computadora pueden ser muy grandes, por lo que es necesario colocar **comentarios** de código en el programa. Los comentarios sirven para documentar las partes del programa, explicar lo que hace cierta parte del código o simplemente colocar un recordatorio.

Podemos hacer comentarios de una sola línea, que se colocan por medio de la doble barra // seguido del texto del comentario. El texto que se encuentre a partir de la doble barra // y hasta el fin del renglón es ignorado por el compilador. Algunos ejemplos de comentarios pueden ser:

```
// Declaro mis variables
int ancho, alto;

int tarea; // Esta variable guarda el área calculada.

// La siguiente sentencia nunca se ejecuta
// Console.WriteLine("Hola");
```

Si llegáramos a necesitar colocar mucha información en nuestros comentarios, puede ser útil explayarla en varios renglones, es decir, hacer un **bloque de comentarios**. Para esto, abrimos nuestro bloque de comentarios con una barra seguida de un asterisco, /*, y finalizamos éste con un asterisco en primer lugar, seguido de una barra */. Todo lo que se encuentre entre estos delimitadores será tomado como un comentario y el compilador lo pasará por alto.

Veamos un ejemplo de un comentario de múltiples líneas:

```
/* La siguiente sección calcula el área y usa las variables
alto - para guardar la altura del rectángulo
ancho - para guardar la base del rectángulo
*/
```

Así vemos cómo los comentarios colocados en varios renglones, se toman como un solo comentario. Esto es algo muy útil para recordatorios futuros.

Mostrar los valores de las variables en la consola

Ya que podemos guardar valores, también nos gustaría poder mostrarlos cuando sea necesario. Afortunadamente, por medio del método **WriteLine()** o el método **Write()** podemos hacerlo. En este caso no usaremos la cadena como antes, lo que usaremos es una **cadena de formato** y una **lista de variables**. La cadena de formato nos permite colocar el mensaje a mostrar e indicar qué variable usaremos para mostrar su contenido y también en qué parte de la cadena queremos que se muestre. En la lista de variables simplemente colocamos las variables que queremos mostrar.

La cadena de formato se trata como una cadena normal, pero agregamos {} donde deseamos que se coloque el valor de una variable. Adentro de {} debemos colocar el **índice** de la variable en nuestra lista. Los índices inician en **cero**. Si queremos que se muestre el valor de la primera variable en la lista colocamos **{0}**, si es la segunda entonces colocamos **{1}** y así sucesivamente. La lista de variables simplemente se coloca después de la cadena, separando con comas las diferentes variables.

Por ejemplo, podemos mostrar el valor de una variable de la siguiente forma:

```
Console.WriteLine("Se tiene {0} en la variable", costo);
```

Como solamente tenemos una variable en nuestra lista de variables entonces colocamos **{0}**. Pero también podemos mostrar el valor de dos variables o más, como lo muestra el siguiente ejemplo:

```
Console.WriteLine("La primera es {0} y la segunda es {1}", costo, valor);
```

Ahora veamos un ejemplo con lo que hemos aprendido.

```
using System;
using System.Collections.Generic;
```

```
using System.Text;

namespace AplicacionBase
{
    class Program
    {
        // Esta es la función principal del programa
        // Aquí inicia la aplicación
        static void Main(string[] args)
        {

            // Declaramos variables
            int costo;
            int valor, precio;

            // Inicializamos las variables
            costo = 50;
            valor = 75;
            precio = 125;

            // Declaramos e inicializamos
            int impuesto = 10;

            // Mostramos un valor
            Console.WriteLine("El valor adentro de costo es {0}", costo);

            // Varias variables
            Console.WriteLine("Valor es {0} y precio es {1}", valor,
                precio);

            // Dos veces la misma variable
            Console.WriteLine("Valor es {0} y precio es {1} con valor de
                {0}", valor, precio);

            /* No olvidemos
            mostrar el valor
            de la variable impuesto
            */
            Console.WriteLine("Y el valor que nos faltaba mostrar
                {0}", impuesto);
```

```

        }
    }
}

```

Una vez que hemos compilado la aplicación, obtendremos el resultado que mostramos en la figura a continuación.

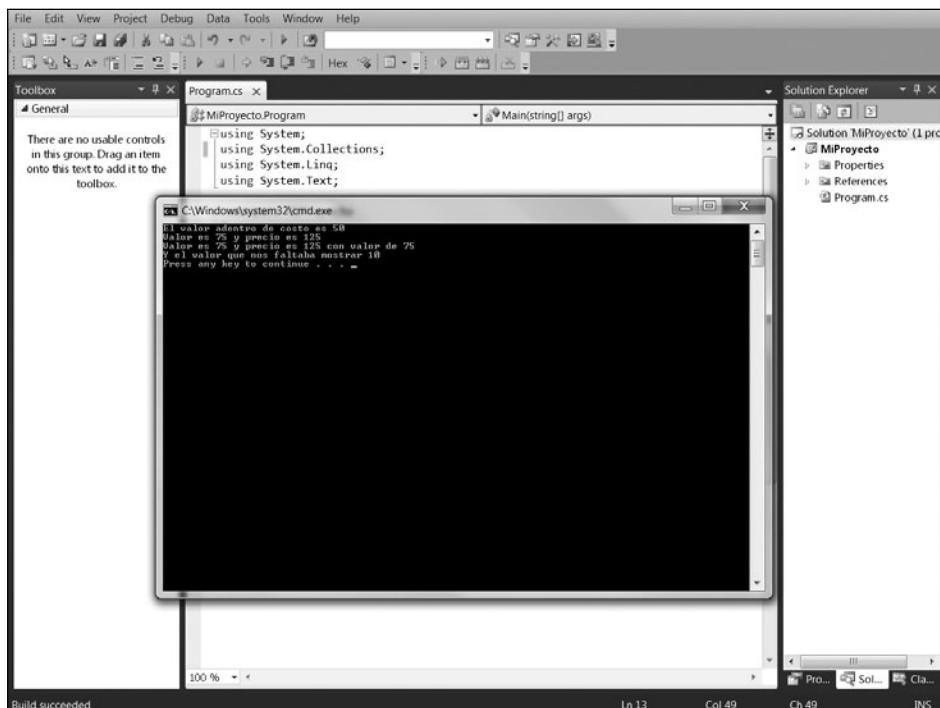


Figura 7. En la consola observamos que los valores guardados en las variables fueron desplegados de acuerdo con lo colocado en la cadena de formato.

Operaciones aritméticas

Poder guardar la información es muy importante, pero es mucho más importante que podamos recuperar y manipular esa información para hacer algo útil con ella. Para esto, debemos comenzar a realizar las operaciones aritméticas básicas: suma, resta, multiplicación y división, algo muy común en cualquier sistema informático.

Cuando queremos o necesitamos realizar una operación aritmética debemos hacer uso de un **operador**. Cada operación tiene su propio operador, que es el encargado de procesar los números, realizando el determinado cálculo, para luego devolvernos el resultado deseado.

OPERADOR	DESCRIPCIÓN
=	Asignación. Este operador ya es conocido por nosotros.
+	Suma. Nos permite sumar los valores de las variables o los números
-	Resta. Para restar los valores de las variables o los números.
*	Multiplicación. Multiplica los valores de las variables o los números.
/	División. Divide los valores de las variables o los números.
%	Módulo. Nos da el residuo de la división.

Tabla 2. Ésta es la tabla de operadores aritméticos en C#.

Veamos algunos ejemplos de cómo utilizar estos operadores. Imaginemos que ya hemos declarado e inicializado las variables. Para guardar en la variable **resultado** la suma de dos números, hacemos lo siguiente:

```
resultado = 5 + 3;
```

Como hemos visto, la expresión a la derecha se le asigna a la variable, por lo que en este caso la expresión **5 + 3** se evalúa y su resultado que es **8** es almacenado en la variable **resultado**. Al finalizar esta sentencia, la variable **resultado** tiene un valor de **8**, que podrá ser utilizado dentro de la misma función o en el procedimiento donde fue declarado.

Las operaciones también se pueden realizar con las variables y los números. Supongamos que la variable **a** ha sido declarada y se le ha asignado el valor de **7**.

```
resultado = a - 3;
```

De nuevo se evalúa primero la expresión **a-3**, como el valor adentro de **a** es **7**, entonces la evaluación da el valor de **4**, que se le asigna a la variable **resultado**.

Si queremos podemos trabajar únicamente con variables. Ahora supondremos que la variable **b** ha sido declarada y que le hemos asignado el valor **8**.

```
resultado = a * b;
```

Entonces, se evalúa la expresión **a*b** que da el valor **56**, que queda asignado a la variable **resultado** al finalizar la expresión.

Si lo que deseamos es la división, podemos hacerla de la siguiente forma:

```
resultado = a / b;
```

En este caso, se divide el valor de **a** por el valor de **b**. El resultado es **0,875**. Como el número tiene valores decimales, debemos usar variables de algún tipo que nos permita guardarlos como **float** o **double**.

Y por último nos queda el módulo.

```
resultado = a % b;
```

En este caso recibimos en resultado el valor del residuo de dividir **a** por **b**.

Veamos un pequeño programa donde se muestra lo aprendido en esta sección.

```
using System;
using System.Collections.Generic;
using System.Text;

namespace AplicacionBase
{
    class Program
    {
        // Esta es la función principal del programa
        // Aquí inicia la aplicación
        static void Main(string[] args)
        {
            // Declaramos las variables, ahora de tipo flotante
            float a, b, resultado;

            // Inicializamos las variables
            a = 7;
            b = 8;
```

III

CÓMO EVITAR ERRORES EN LAS EXPRESIONES MATEMÁTICAS

Los errores más comunes en las expresiones matemáticas son olvidar el punto y coma, hacer la asignación errónea, escribir mal el nombre de la variable y agrupar incorrectamente las operaciones. Utilizar los paréntesis para agrupar ayuda mucho a reducir errores.

```
resultado = 0;

// Sumas
Console.WriteLine("Sumas");

resultado = 3 + 5;
Console.WriteLine("Resultado = {0}", resultado);

resultado = a + 3;
Console.WriteLine("Resultado = {0}", resultado);

resultado = a + b;
Console.WriteLine("Resultado = {0}", resultado);

// Restas
Console.WriteLine("Restas");

resultado = a - b;
Console.WriteLine("Resultado = {0}", resultado);

resultado = b - 5;
Console.WriteLine("Resultado = {0}", resultado);

resultado = b - a; // A la variable b se le resta a
Console.WriteLine("Resultado = {0}", resultado);

// Multiplicaciones
Console.WriteLine("Multiplicaciones");

resultado = a * 5;
Console.WriteLine("Resultado = {0}", resultado);

resultado = a * 3.5f;
Console.WriteLine("Resultado = {0}", resultado);

resultado = a * b;
Console.WriteLine("Resultado = {0}", resultado);

// Divisiones
Console.WriteLine("Divisiones");
```

```

        resultado = a / 3;
        Console.WriteLine("Resultado = {0}", resultado);

        resultado = a / b;
        Console.WriteLine("Resultado = {0}", resultado);

        resultado = b / 2.5f;
        Console.WriteLine("Resultado = {0}", resultado);

        // Modulo
        Console.WriteLine("Modulo");

        resultado = a % b;
        Console.WriteLine("Resultado = {0}", resultado);
    }
}
}

```

Podemos ver la ejecución en la siguiente figura:

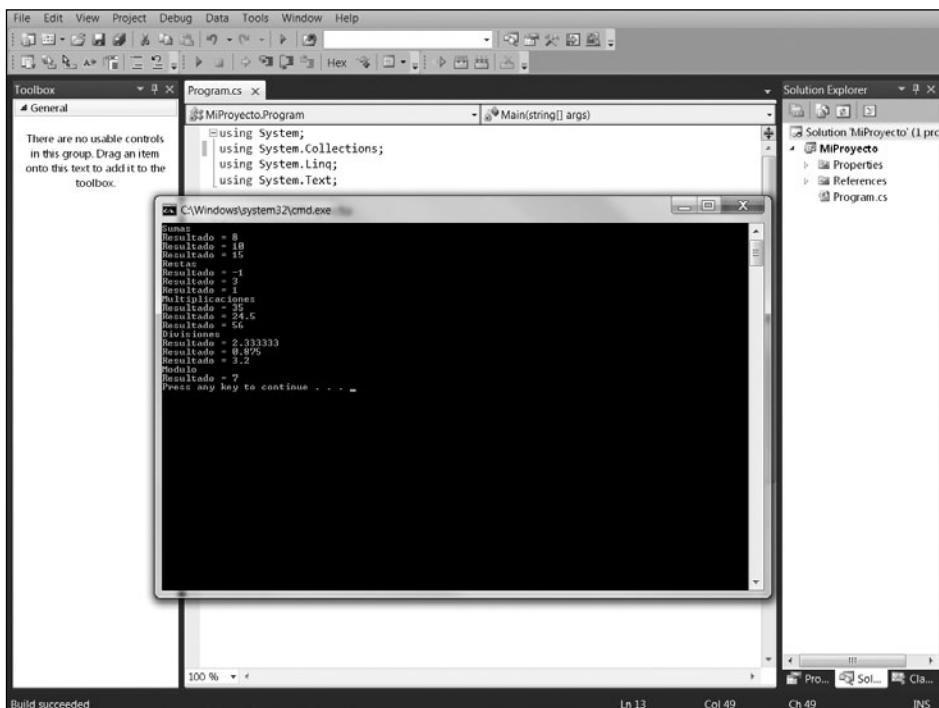


Figura 8. Vemos los resultados de las operaciones aritméticas sobre nuestras variables.

Precedencia de operadores

Hasta el momento hemos utilizado expresiones aritméticas sencillas, pero en la vida real se pueden necesitar expresiones más complicadas. Entonces, si estas expresiones no se colocan adecuadamente, esto nos puede ocasionar problemas.

Por ejemplo, veamos la siguiente expresión.

```
resultado = 3 * 5 + 2;
```

Esta expresión presenta un problema para nosotros en este momento. No sabemos si primero hace $3*5$ y le suma 2, que da como resultado 17, o si suma $5+2$ y luego lo multiplica por 3, lo que da como resultado 21. ¿Cuál es la respuesta correcta?

La forma de saberlo es por medio de la **precedencia de operadores**. Ésta nos indica el orden en el que se llevan a cabo las operaciones. Este orden depende del tipo de operador que se utiliza. Algunos operadores tienen más precedencia que otros, es decir, se ejecutan primero.

La siguiente tabla muestra la precedencia de los operadores que hemos visto hasta el momento en C#. Se encuentran listados de mayor a menor precedencia.

OPERADOR	DESCRIPCIÓN
*	Multiplicación
/	División
%	Módulo
+	Adición
-	Sustracción

Tabla 3. Esta tabla nos muestra la precedencia de operadores en C#. La multiplicación tiene más precedencia y se ejecuta primero.

Esto quiere decir que cuando tenemos una expresión aritmética como la del ejemplo, primero se llevaría a cabo la multiplicación y luego la suma, entonces con esto podemos deducir cómo C# evaluaría la expresión.

Pero algunas veces sería mejor si nosotros pudiéramos organizar nuestra expresión con el fin de hacerla más fácil de leer para nosotros o para que podamos indicar de forma precisa cómo hacer la operación. Para que podamos organizar una expresión hacemos uso de los paréntesis. Cada sección que tengamos en los paréntesis se evalúa como una expresión y el resultado se pasa al resto de la expresión. Supongamos que tenemos la siguiente expresión:

```
resultado = (3*5) + 2;
```

Vemos que por medio de los paréntesis se ha agrupado una parte de la expresión, por lo que se evalúa y equivale a:

```
resultado = 15 + 2;
```

Lo que al final evalúa en:

```
resultado = 17;
```

Veamos otro ejemplo de cómo podemos usar los paréntesis para indicar cómo deseamos que se evalúe la expresión.

```
resultado = 3 * (5+2);
```

Lo primero que sucede es que se evalúa la expresión contenida en el paréntesis.

```
resultado = 3 * 7;
```

Que nos da el valor:

```
resultado = 21;
```

Veamos un ejemplo un poco más complicado.

```
resultado = (3+7) * (36 + 4 * (2+5));
```

Veamos qué sucede:

```
resultado = 10 * (36 + 4 * 7);
```

Luego se continúa evaluando:

```
resultado = 10 * (36 + 28);
```

Lo que da como resultado:

```
resultado = 10 * 64;
```

Y al final obtenemos:

```
resultado = 640;
```

Afortunadamente, C# hace todas estas evaluaciones y todos estos cálculos por nosotros, pero ahora sabemos cómo sucede y sabemos que podemos usar los paréntesis para organizar nuestras expresiones aritméticas.

Cómo pedirle datos al usuario

Hemos utilizado variables para guardar valores y con los operadores aritméticos hemos podido realizar cálculos matemáticos. Sin embargo, todos los valores que hemos usado se encuentran colocados directamente en el código del programa. Esto no es muy cómodo, ya que si deseamos hacer un cálculo con diferentes valores, es necesario modificar el código y compilar nuevamente. Sería más útil si pudiéramos hacer que el usuario colocara los valores que necesita cuando el programa se ejecuta. Para esto, C# nos provee de un método que pertenece a la clase **Console**. El método se llama **ReadLine()**. Éste no necesita ningún parámetro, por lo que sus paréntesis permanecerán vacíos, y nos regresa una cadena que contiene lo que escribió el usuario con el teclado.

Ésta es una forma sencilla para pedirle información al usuario. Supongamos que deseamos pedirle al usuario su nombre, que utilizaremos a posteriori para saludarlo. Para eso colocamos el siguiente código:

```
string entrada = " ";
```



AGRUPACIÓN DE LOS PARÉNTESIS

Un error común con expresiones grandes es olvidar cerrar un paréntesis en forma adecuada. En Visual Studio es fácil ver cuáles paréntesis son pareja ya que al seleccionar uno de ellos su pareja se ilumina. Para cada paréntesis de apertura forzosamente debe de existir un paréntesis de cierre. Cuando esto no se logra se denomina desbalance de paréntesis.

```

Console.WriteLine("Escribe tu nombre");
entrada=Console.ReadLine();

Console.WriteLine("Hola {0}, como estas?",entrada);

```

En este fragmento de programa es importante notar cómo el valor obtenido por **ReadLine()** se le asigna a la variable de tipo cadena **entrada**. Si no colocamos una variable que recibe el valor de lo escrito por el usuario, éste se perderá. Cuando usemos **ReadLine()** debemos hacerlo como se muestra. Una vez que recibimos la entrada del usuario podemos trabajar con ella sin problemas.

Conversión de variables

El método de **ReadLine()** nos presenta una limitación en este momento ya que sólo regresa el valor como cadena, pero no podemos utilizar una cadena en operaciones aritméticas o asignarla directamente a una variable numérica.

Lo que debemos hacer es convertir ese dato numérico guardado como cadena a un valor numérico útil para nuestros propósitos. C# nos provee de una clase que permite hacer conversiones entre los diferentes tipos de variables. Esta clase se conoce como **convert** y tiene una gran cantidad de métodos para la conversión. Aquí sólo veremos cómo convertir a enteros y flotantes, ya que son los tipos con los que trabajaremos en este libro principalmente. Sin embargo, los demás métodos se usan de forma similar y podemos encontrar la información necesaria sobre éstos en **MSDN** (*MicroSoft Developer Network*).

Si queremos convertir del tipo **string** al tipo **int** usaremos el método **ToInt32()**. Éste necesita un parámetro, que es la cadena que deseamos convertir. El método regresa un valor de tipo **int**, por lo que es necesario tener una variable que lo reciba.

Un ejemplo de esto es:

```
a = Convert.ToInt32(entrada);
```



MSDN

MSDN es un sitio web donde encontraremos información técnica sobre todas las clases y funciones, y todos los métodos que conforman los lenguajes que componen Visual Studio. También es posible encontrar ejemplos y descripciones de los errores de compilación. Tiene un buen sistema de búsquedas que resulta muy útil. La dirección es www.msdn.com.

De forma similar podemos convertir la cadena a una variable de tipo **float**. En este caso usaremos el método **ToSingle()**. Éste también necesita un parámetro, que es la cadena que contiene el valor a convertir. Como regresa un valor flotante, entonces debemos tener una variable de tipo **float** que lo reciba.

```
valor = Convert.ToSingle(entrada);
```

Ya que tenemos los valores convertidos a números, entonces podemos hacer uso de ellos. Veamos un ejemplo donde le pedimos información al usuario.

```
using System;
using System.Collections.Generic;
using System.Text;

namespace AplicacionBase
{
    class Program
    {
        // Esta es la función principal del programa
        // Aquí inicia la aplicación
        static void Main(string[] args)
        {
            // Declaramos variables
            string entrada = "";
            int a = 0, b = 0, resultado = 0;

            // Leemos una cadena
            Console.WriteLine("Escribe tu nombre");
            entrada = Console.ReadLine();

            Console.WriteLine("Hola {0}, como estas?", entrada);

            // Leemos dos valores y los sumamos.

            Console.Write("Dame un entero:");
            entrada = Console.ReadLine();

            // Convertimos la cadena a entero
            a = Convert.ToInt32(entrada);
```

```

Console.WriteLine("Dame otro numero entero:");
entrada = Console.ReadLine();

// Convertimos la cadena a entero
b = Convert.ToInt32(entrada);

// Sumamos los valores
resultado = a + b;

// Mostramos el resultado
Console.WriteLine("El resultado es {0}", resultado);

}

}
}

```

Ahora compilemos y ejecutemos la aplicación. Para introducir los datos, simplemente los escribimos con el teclado y al finalizar oprimimos la tecla **ENTER** o **RETURN**.

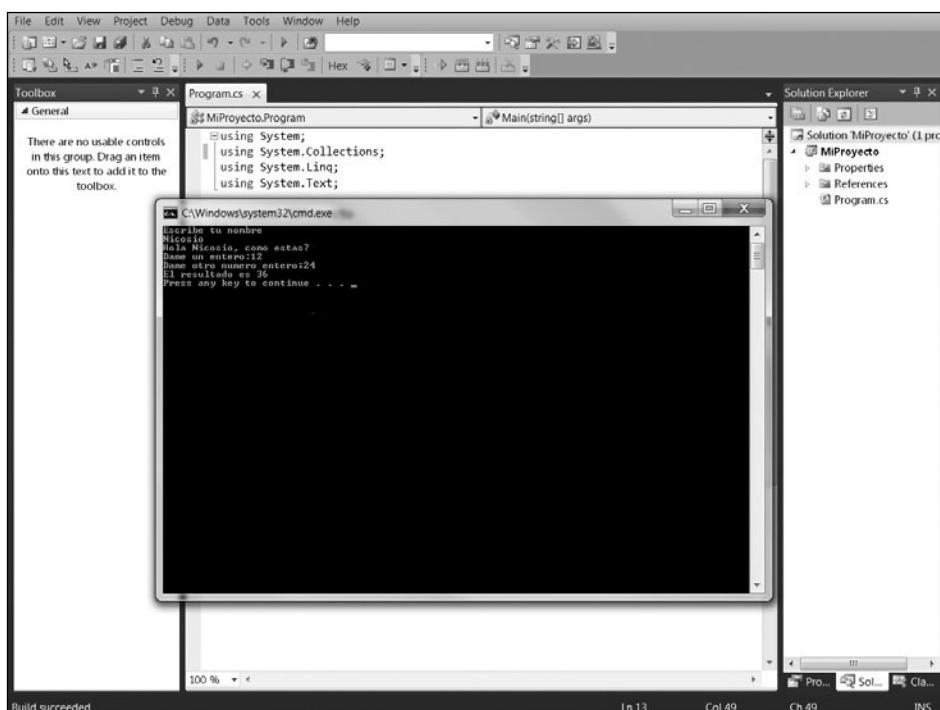


Figura 9. Éste es el resultado de la ejecución, donde vemos que la suma se lleva a cabo con los valores introducidos por el usuario.

Cómo resolver problemas en la computadora

Ya que conocemos algunos elementos del lenguaje, ahora es importante aprender cómo resolver los problemas en la computadora. Existen diferentes metodologías para hacerlo, pero como éste es un libro dirigido a gente sin experiencia en programación aprenderemos un método sencillo. Como ya hemos mencionado, C# es un lenguaje orientado a objetos. Sin embargo, primero aprendaremos una metodología estructurada, lo que nos permitirá tener experiencia en la resolución de problemas y luego podremos aprender la metodología orientada a objetos de manera más sencilla.

La metodología que aprendemos se basa en el concepto de subdivisión de problemas. Es decir que tomamos el problema general y luego lo dividimos en problemas más pequeños y si es necesario, dividimos estos subproblemas en problemas más pequeños, hasta que podamos resolver el subproblema directamente.

Para resolver cualquier problema, lo primero que tenemos que hacer es entenderlo. Hay que entender de forma precisa qué es lo que se pide. Los problemas tendrán información. En muchos casos, el problema nos da directamente la información. A este tipo de información se lo conoce como **información explícita**. Pero también a veces sucede que el problema no nos da directamente la información, por eso es importante entenderlo bien. En este caso nosotros debemos inferir la información de otros datos que nos proporciona o buscarla en algún otro lugar. A este tipo de información se lo conoce como **información implícita**. Afortunadamente, la información implícita en muchos casos es fácil de encontrar o forma parte de los conocimientos generales de cualquier persona.

Ya que tenemos el problema subdividido y los datos reconocidos, procedemos a realizar el algoritmo. Recordemos que el algoritmo es la serie de pasos necesarios para resolver el problema. Además, éste puede definirse por medio de un diagrama de flujo o escribirse en algo que denominaremos **pseudocódigo**. El programador puede crear el algoritmo con su método de preferencia.

Cuando se ha terminado el algoritmo, procedemos a probarlo. En esta etapa podemos probar en el papel si todo funciona bien. Si hubiera algún resultado erróneo, entonces se puede corregir el algoritmo y el proceso se repite hasta que tengamos un algoritmo que funcione adecuadamente.

III CONVERSIÓN DE NÚMEROS A CADENAS

Tambien es posible convertir de números a cadenas. Para esto se usa el método **ToString()**, que está disponible en todos los tipos numéricos de C#. De esta forma podemos utilizar la información de las variables numéricas en funciones o métodos que trabajen con cadenas. El método no cambia el tipo, simplemente regresa una cadena que contiene el texto de ese valor.

El último paso consiste en pasar el algoritmo a programa de computadora. Si todo se ha realizado correctamente, el programa deberá ejecutarse al primer intento. Mucha gente se sorprende que para hacer un programa de computadora, el último paso sea colocarlo en la computadora, pero ésta es la forma correcta. Algunos programadores no profesionales intentan hacer el programa directamente en la computadora, pero esto acarrea muchos errores de lógica y datos, y el tiempo en que resuelve el mismo problema es mayor y es difícil que se ejecute correctamente al primer intento. Aunque la metodología que acabamos de presentar parece larga, en realidad puede ahorrar hasta un 50% del tiempo de resolución en comparación con alguien que programe directamente en la computadora. Ahora que empezamos, es bueno adquirir hábitos correctos de programación desde el inicio. En el futuro, en especial cuando los problemas que se resuelvan sean complicados, se apreciarán los beneficios.

Elementos de un diagrama de flujo

Es momento de aprender los componentes básicos de un **diagrama de flujo**. Con estos diagramas podemos representar el algoritmo y son fáciles de hacer y entender. Otra ventaja que tienen es que son independientes del lenguaje, lo que nos da la flexibilidad de resolver el problema y usar nuestra solución en cualquier lenguaje.

El diagrama de flujo es una **representación gráfica del algoritmo** y tiene diferentes componentes. Cada componente es una figura y ésta representa un tipo de actividad. En el interior de la figura colocamos una descripción del paso que está sucediendo ahí. Por medio de flechas indicamos cuál es el paso siguiente y nos da la idea del flujo general del algoritmo. En algunas ocasiones, a las flechas les agregamos información extra, pero esto lo veremos más adelante. Aunque en este momento no utilizaremos todos los componentes del diagrama de flujo, sí conoceremos los más importantes, ya que en los capítulos posteriores los utilizaremos.

El primer componente tiene forma de ovalo y se lo conoce como **terminador**. Éste nos sirve para indicar el inicio o el final del algoritmo. Al verlo podemos saber inmediatamente dónde inicia el algoritmo y poder continuar a partir de ahí. Hay que recordar que los algoritmos deben tener un final para que sean válidos. Si el terminador es de inicio, colocamos en su interior el mensaje: **Inicio**. Si el terminador indica dónde se acaba el algoritmo, entonces se coloca el mensaje: **Final**.

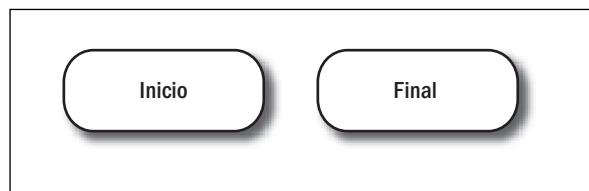


Figura 10. En esta figura podemos observar el terminador tanto en su papel como punto de inicio como punto final del algoritmo.

El siguiente elemento se conoce como **proceso**. Éste tiene la forma de un rectángulo y nos sirve para indicar que precisamente un proceso debe llevarse a cabo en ese lugar. La descripción del proceso se indica en su interior.

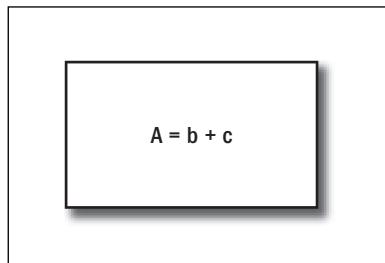


Figura 11. El proceso se indica por medio de un rectángulo. En su interior indicamos lo que se tiene que procesar.

Similar al proceso, tenemos el **proceso predefinido**. Su forma también es un rectángulo, pero éste tiene dos franjas a los lados. Se usa para indicar que haremos uso de un proceso que ya ha sido definido en otra parte. Un ejemplo de esto sería un método, como los que ya hemos usado. Principalmente lo usaremos para indicar llamadas a funciones o métodos definidos por nosotros mismos. En un capítulo posterior aprenderemos cómo hacer nuestros propios métodos.

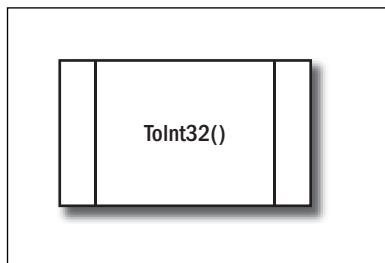


Figura 12. Éste es el símbolo usado para el proceso predefinido.

Nuestro próximo elemento es conocido como **condición**. Utilizaremos este elemento cuando aprendamos cómo hacer que el programa tome sus propias

III DATOS DE TRABAJO

Algunas veces necesitaremos variables que nos apoyen para resolver algún cálculo o proceso. A éstas las llamaremos datos de trabajo. Los datos de trabajo se descubren cuando hacemos el análisis del problema. Si llegamos a omitir alguno durante el análisis siempre es posible agregarlo durante el desarrollo, pero esto debe ser la excepción a la regla.

decisiones. Básicamente, sirve para hacer una selección entre las posibles rutas de ejecución del algoritmo, que dependerá del valor de la expresión evaluada. Su forma es la de un rombo y a partir de las esquinas de éste surgirán las flechas hacia las posibles rutas de ejecución del algoritmo.

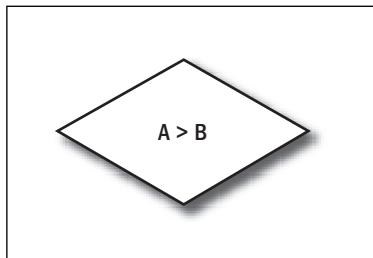


Figura 13. La decisión se representa con un rombo.

Luego nos encontramos con un componente conocido como **datos**. Su forma es la de un paralelogramo y podemos usarlo para indicar la salida o la entrada de los datos. Un ejemplo de esto sería cuando llevamos a cabo la petición de un valor al usuario o cuando mostramos un valor en pantalla. Adentro de él indicamos qué dato se pide o se presenta.

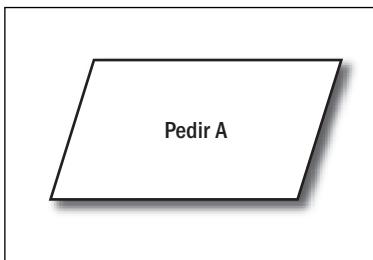


Figura 14. En este caso usamos el dato para indicar la petición del valor de una variable.

Hemos visto los elementos principales del diagrama de flujo. Ahora podemos hacer un ejercicio que muestre la solución de un problema en la computadora.



C# EXPRESS...

Es una versión gratuita de C# con las herramientas básicas para programar en este lenguaje. Esta versión nos provee todo lo necesario para los temas que aprenderemos en este libro. Aunque la versión es gratuita, es necesario registrarla para poder utilizarla por más de treinta días. La podemos descargar de <http://msdn2.microsoft.com/en-us/express/aa700756.aspx>.

Resolución de problemas en la computadora

Empecemos por un problema sencillo y fácil de entender. Para este problema seguiremos todos los pasos que hemos mencionado hasta llegar al programa de cómputo final. En primer lugar, tenemos la descripción del problema.

Hacer un programa de cómputo que calcule el área y el perímetro de un rectángulo dados sus lados. Una vez que tenemos el problema y lo hemos entendido, entonces procedemos a subdividirlo en problemas más pequeños que sean fáciles de resolver. Esta subdivisión puede ser hecha por medio de un diagrama y para esto hay que recordar que si alguno de los problemas aún continua siendo difícil, lo podemos seguir subdividiendo. A modo de ejemplo exageraremos un poco la subdivisión en este caso. El problema quedaría subdividido como se muestra en la siguiente figura.

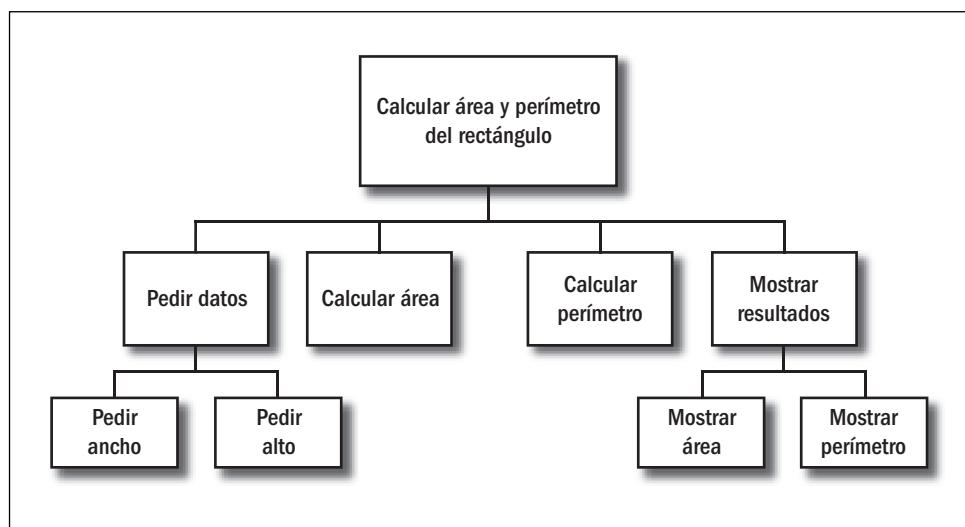


Figura 15. Aquí tenemos nuestro problema subdividido en problemas menores y fáciles de resolver.

Ya que conocemos el problema y lo hemos subdividido en varios subproblemas, podemos identificar de manera más clara los datos que nos hacen falta para poder trabajar en él. La descripción del problema nos proporciona dos datos explícitos: el



MSDN

Si queremos encontrar información sobre cualquier clase o método de C#, el lugar apropiado para hacerlo es el sitio web MSDN que nos provee Microsoft. Cuando realicemos la búsqueda debemos filtrar para buscar únicamente contenidos del lenguaje C# y así evitar confusiones con información de otros lenguajes. La dirección es www.msdn.com.

área y el **perímetro**. Pero también reconocemos inmediatamente dos datos implícitos que son necesarios: el **alto** y el **ancho** del rectángulo.

Nosotros sabemos que necesitamos una cadena para recibir el valor escrito por el usuario, pero como tal, esta cadena no forma parte del problema. Sin embargo, nos ayuda a resolverlo, por lo que la cadena será un dato de trabajo. Los datos que hemos identificados serán colocados en una tabla de la siguiente forma:

NOMBRE	TIPO	VALOR INICIAL
área	Float	0.0
perímetro	Float	0.0
ancho	Float	1.0
alto	Flota	1.0
valor	String	“”

Tabla 4. Tabla de variables para el programa que estamos resolviendo.

Hemos decidido usar el tipo **float**, ya que podemos tener valores decimales en el cálculo del perímetro. Como un rectángulo con ancho o alto de 0 no tiene sentido, hemos seleccionado como valor inicial para estas variables a 1.

Ahora viene el paso más importante: la creación del algoritmo. Sin embargo, si observamos la forma como subdividimos el problema, podemos ver que nos da una indicación de los pasos que son necesarios colocar en el algoritmo.

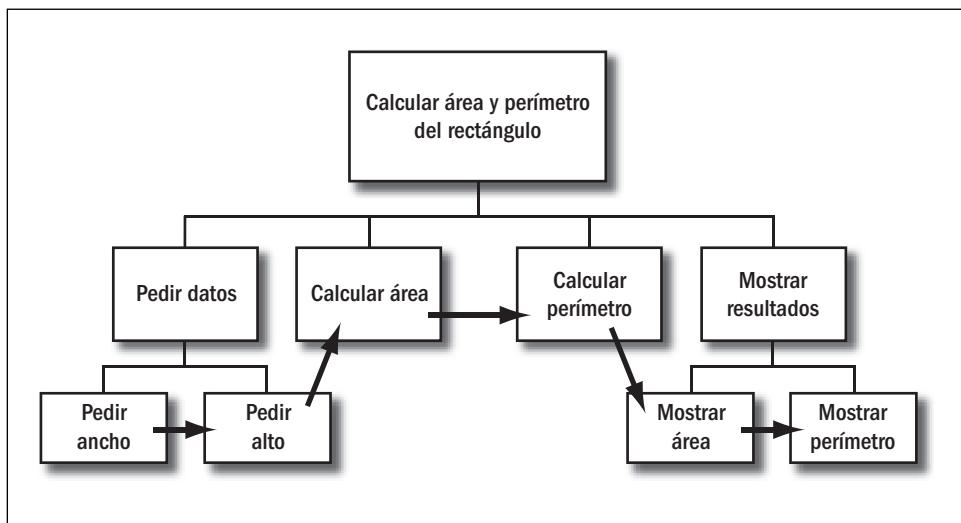
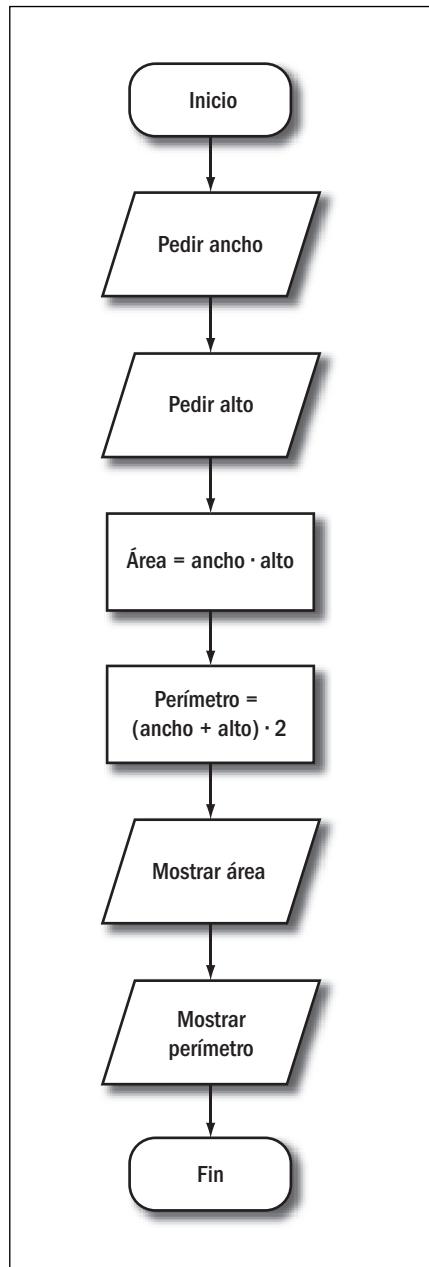


Figura 16. La subdivisión nos muestra una lista de los pasos posibles necesarios en la subdivisión del algoritmo.

Con esta información creamos el algoritmo con el diagrama de flujo. En éste aparece la secuencia de pasos en el orden correcto. Veamos cómo queda el diagrama.

**Figura 17.** El diagrama de flujo muestra el algoritmo.

Las diferentes figuras nos indican el tipo de actividad que se realiza.

Podemos observar cómo hemos utilizado el trapezoide cuando es necesario pedirle o mostrarle un dato al usuario. Los cálculos con la fórmula correspondiente están en los rectángulos, ya que son procesos, y el inicio y fin del algoritmo están representados por los terminadores.

El código de nuestra aplicación

Ya tenemos todo lo necesario para poder crear el código del programa. Para esto haremos uso de la aplicación base que ya tenemos. Como creamos una tabla de datos, ya conocemos todas las variables necesarias para la aplicación. Entonces tomamos esta tabla y colocamos los datos como variables.

```
// Declaramos las variables que necesitamos
float area = 0.0f;
float perimetro = 0.0f;
float ancho = 1.0f;
float alto = 1.0f;
string valor = "";
```

Con las variables definidas, entonces procedemos a codificar cada uno de los pasos del algoritmo. Recorremos paso por paso y en el mismo orden en que se encuentran, y vemos que lo primero es pedir el ancho del rectángulo.

```
Console.WriteLine("Dame el ancho del rectángulo: ");
valor = Console.ReadLine();

ancho = Convert.ToSingle(valor); // Convertimos a flotante
```

Lo siguiente que hace el algoritmo es pedir el alto del rectángulo, entonces el código necesario es el siguiente.

```
Console.WriteLine("Dame el alto del rectángulo: ");
valor = Console.ReadLine();

alto = Convert.ToSingle(valor); // Convertimos a flotante
```



EXPERIMENTANDO

Aprender a programar requiere de mucha práctica y mucha experimentación. Es bueno probar con los programas y ver qué es lo que sucede. No debemos temer en cometer errores cuando estamos aprendiendo. También es bueno verificar la documentación para las funciones, las clases y los métodos ya que es posible encontrar utilidades en éstas para el futuro.

Una vez que hemos obtenido todos los datos de entrada, podemos decir que ya tenemos un proceso. Este proceso será el encargado de calcular el área que ocupa el rectángulo. Veamos el ejemplo a continuación:

```
area = ancho * alto;
```

En el próximo proceso que tenemos se calcula el perímetro.

```
perímetro = (ancho + alto) * 2;
```

Si vemos con detenimiento el algoritmo y también su paso siguiente, encontraremos que volvemos a trabajar con datos nuevamente, pero en este último caso sólo mostramos en pantalla el valor de área calculado.

Veamos para comprender mejor de qué hablamos.

```
Console.WriteLine("El área es : {0} unidades cuadradas", area);
```

Como último paso del algoritmo que se está ejecutando, sólo queda que mostremos en nuestra aplicación de consola, el valor calculado del perímetro.

A continuación escribamos el último código necesario en nuestra aplicación:

```
Console.WriteLine("El perímetro es : {0} unidades", perímetro);
```

Hemos visto hasta aquí una forma completa y compleja de cómo manejar desde una aplicación de consola, la manipulación de datos, variables y operaciones matemáticas, con un claro ejemplo estudiado paso a paso. Con esto ya hemos finalizado el código de la aplicación. Ahora lo podemos compilar y probar, y todo debe funcionar correctamente, si no hemos olvidado de colocar nada en él.

III SITIO CON RECURSOS SOBRE C#

Un sitio web que nos presenta diferentes recursos de C# y cuyo contenido está en idioma inglés es: www.csharpHelp.com. Aquí podremos encontrar información cuando tengamos algún problema con el lenguaje o deseemos aprender cómo se realiza algo en particular. También hay información para programar con formas de Windows.

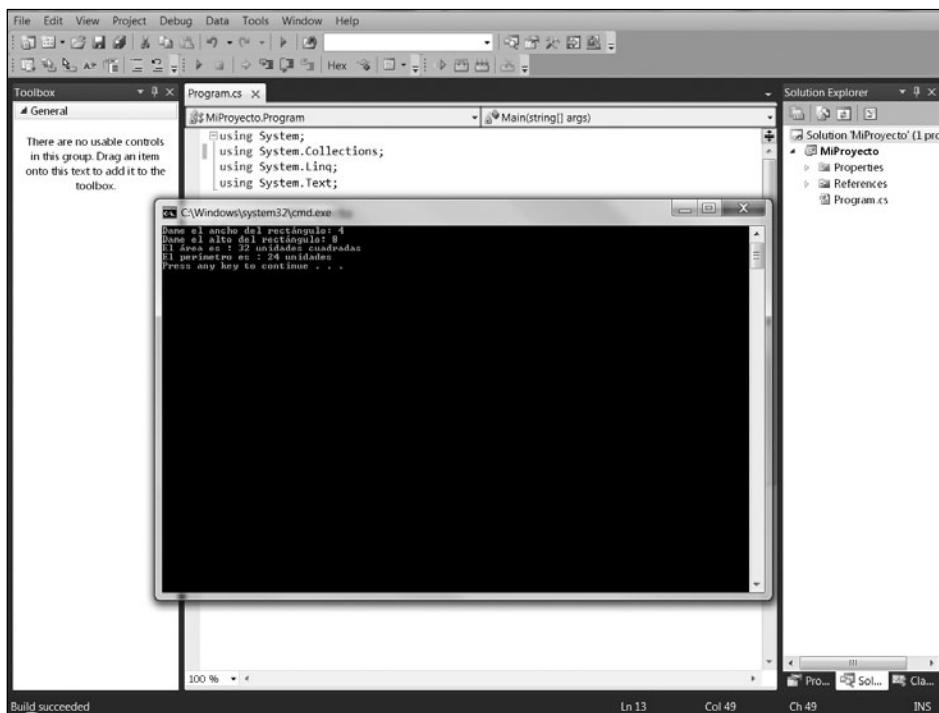


Figura 18. Aquí podemos ver la ejecución de nuestro programa, la forma como pide los datos y los resultados calculados.

... RESUMEN

Para programar software para computadoras necesitamos conocer un lenguaje de programación. Uno de estos lenguajes es C#. Los programas deben ser ordenados e indicar paso a paso lo que se debe hacer, y para esto hacemos uso de los algoritmos. La información se guarda en variables y el tipo indica la información que puede tener en su interior por lo que es posible mostrar datos en la consola y también pedírselos al usuario. Los operadores aritméticos nos permiten llevar a cabo operaciones matemáticas, siempre que usemos una metodología correcta de resolución de problemas.



ACTIVIDADES

TEST DE AUTOEVALUACIÓN

1 ¿Qué es un algoritmo?

2 ¿Qué características tienen los algoritmos?

3 ¿Qué es una sentencia?

4 ¿Cómo se finalizan las sentencias?

5 ¿Cuál es la diferencia entre Write() y WriteLine()?

6 ¿Qué es una cadena?

7 ¿Qué es una variable?

8 ¿Cómo mostramos el valor de una variable?

9 ¿Cómo le solicitamos un dato al usuario?

10 ¿Cómo se convierte una cadena a un valor numérico?

11 ¿Qué son los operadores aritméticos?

12 ¿De qué forma podemos agrupar operaciones aritméticas?

EJERCICIOS PRÁCTICOS

1 Hacer un programa que calcule el perímetro de cualquier polígono regular.

2 Generar en este proyecto un error a propósito y ver cómo se comporta el compilador.

3 Hacer un programa que transforme de grados a radianes.

4 Hacer un programa que transforme de grados centígrados a grados Fahrenheit.

5 Hacer un programa que transforme entre dólares y euros y que también pida el tipo de cambio del día.

El programa toma decisiones

En los programas creados hasta ahora, hemos visto que los algoritmos tenían un solo sentido de resolución. En este capítulo aprenderemos otras formas diferentes de hacer que nuestros programas tomen decisiones en base al análisis de diversos parámetros, y también aprenderemos a crear algoritmos con diferentes rutas de ejecución para poder solucionar problemas en base a un previo análisis.

La toma de decisiones	72
Expresiones relacionales	72
El uso de if	76
El uso de else	84
Cómo usar if anidados	89
Escalera de if-else	92
Expresiones lógicas	96
El uso de switch	105
Resumen	109
Actividades	110

LA TOMA DE DECISIONES

No todos los problemas se resuelven linealmente, a veces es necesario tener que tomar una decisión o ejecutar determinadas acciones cuando una condición se cumple, y otras cuando no lo hace. Supongamos que nuestro problema consiste en mantener la temperatura de un balde con agua tibia. Para hacerlo nosotros podemos agregar agua caliente o agua fría. En este problema necesitaríamos tomar una decisión sobre qué tipo de agua agregar. De igual forma, hay muchos problemas en los que para poder resolverlos necesitamos conocer una **condición** o tomar una **decisión**. En C# es sencillo poder lograr esto, ya que el lenguaje nos provee diferentes herramientas para poder lograrlo. Tendremos que utilizar expresiones y éstas se evaluarán. En este caso usaremos **expresiones relacionales y expresiones lógicas**, que se evaluarán, y dependiendo del resultado de esa evaluación, se llevarán a cabo ciertos pasos del algoritmo u otros.

Empecemos por conocer las expresiones que necesitamos.

Expresiones relacionales

Las **expresiones relacionales** se usan para expresar la **relación** que existe entre dos valores. Los valores pueden estar contenidos adentro de **variables** o ser colocados **explícitamente**. Estas expresiones, al igual que las expresiones aritméticas, tienen sus propios **operadores**. La expresión será evaluada, pero el resultado de la evaluación tendrá únicamente dos valores posibles: **true** o **false**.

Ambos valores son de tipo **bool** y **true** es usado para indicar que la expresión evaluada es **verdadera**. El valor de **false** por su parte se utiliza para indicar que la expresión evaluada es **falsa**.

Empecemos por conocer primero a los operadores relacionales y luego veremos ejemplos de expresiones relacionales con su resolución.

Operadores relacionales

En la **tabla 1** podemos apreciar los operadores relacionales en C#. La forma como están escritos sus signos es la forma como debemos colocarlos en nuestro programa.

III ERROR MUY COMÚN CON LA IGUALDAD

Un error muy común que sucede cuando se empieza a programar en C# es el de confundir el operador de igualdad con el de asignación. Si se obtiene un error en una expresión, hay que verificar esto. No olvidemos que la asignación lleva un signo igual y la igualdad doble signo igual.

SIGNO	OPERADOR
<code>==</code>	Igualdad
<code>!=</code>	No igual
<code>></code>	Mayor que
<code><</code>	Menor que
<code>>=</code>	Mayor que igual
<code><=</code>	Menor que igual

Tabla 1. Esta tabla nos muestra los diferentes operadores relacionales de C#.

El operador de **igualdad** se representa con dos signos igual juntos y sin espacio. Colocar un solo signo igual, indica que el operador será de **asignación**. Esto puede derivar en errores de lógica, porque una asignación siempre evalúa a verdadero, pero una igualdad no. Este operador es el más sencillo. Supongamos que ya tenemos las variables y las inicializamos con los siguientes valores:

```
int a = 5;
int b = 7;
int c = 5;
int d = 4;
```

Ahora crearemos nuestra primera expresión y la evaluaremos:

```
a == c
```

Como el valor contenido en **a** es **5** y el valor contenido en **c** también es **5**, vemos que se cumple **5 igual a 5**, por lo que la expresión se evalúa como **true**.

Veamos qué sucede en la siguiente expresión:

```
a == d
```

En este caso el valor contenido en **d** es **4**, por lo que la expresión **5 igual a 4** no se cumple y el valor de la expresión es **false**. En el lado derecho y en el lado izquierdo de la expresión podemos colocar cualquier valor, variable o expresión. Si colocásemos una expresión, primero se evaluaría ésta y luego se procedería a evaluar la expresión relacional principal.

```
a == (3 + 2)
```

En este caso **3 + 2** se evalúa como **5** y luego **5 igual a 5**, lo que nos da como valor final de la expresión **true**. Otro ejemplo es el siguiente:

```
(b - 2) == d
```

Al evaluar **b - 2** obtenemos **5**, entonces la expresión es **5 igual a 4**, lo que evidentemente nos da como resultado final **false**.

Ésta es la forma como se evalúan las expresiones relacionales y ahora veremos más ejemplos con otros operadores. También tenemos al operador de **desigualdad**, que se crea por medio de un signo de admiración y luego el signo de igual.

```
a != 7
```

En este caso tenemos **5** no es igual a **7**, lo cual es cierto, y obtenemos el valor **true** como resultado de la evaluación.

```
a != c
```

Para esta expresión tenemos **5** no es igual a **5**, y se evalúa la expresión como falsa, lo que nos da **false** como valor final. El operador **>** sirve para evaluar una relación del tipo **mayor que**. Si el valor del lado izquierdo es mayor que el valor del lado derecho, regresará **true**, en caso contrario regresará **false**.

```
b > a
```

Esto es **7** mayor que **5** y da como resultado **true**.

```
a > b
```

En este caso **5** mayor que **7** es falso y el resultado de la expresión es **false**.

```
b > b
```

Este expresión nos da **6** mayor que **6**, lo cual, si analizamos todos los pasos hasta aquí, nos indica que es falso también y obtenemos **false** como resultado.

En el operador `<` evaluamos una relación del tipo **menor que**. Ésta da **true** como resultado si el valor del lado izquierdo es menor que el valor que encontramos en el lado derecho. Si esto no se cumple, entonces el valor regresado será **false**.

```
d < a
```

Tenemos **4** menor que **5**, lo que resulta cierto y el resultado es **true**.

```
d < (a - 3)
```

La expresión a evaluar es **4** menor que **2**, y se tiene **false** como resultado de la expresión.

```
d < d
```

Esta expresión también da como resultado **false**, ya que **4** menor que **4** no es verdadera. Para poder evaluar si un valor es **mayor que o igual** a otro valor usamos el operador `>=`. Su forma de trabajo es similar a la de los otros operadores.

```
b >= a
```

Para esta expresión tenemos **7** mayor que igual a **5**, es cierta y da **true** de resultado.

```
b >= (a + c)
```

En este caso obtendríamos **7** mayor que igual a **10**, lo cual sabemos que es falso y, por lo cual, como resultado obtendríamos **false**.

III COLOCAR LOS OPERADORES ADECUADAMENTE

Los operadores que se escriben con dos signos deben escribirse correctamente. No es lo mismo `>=` que `>>`. Esto puede llevarnos a errores de sintaxis, pero al escribirlos en la forma correcta se verán corregidos. También debemos recordar que se escriben sin espacios entre los signos, ya que esto nos lleva a otro error de sintaxis.

b >= b

Aquí comparamos la diferencia de resultado con relación a usar `>=` en lugar de `>`. Para esta expresión evaluamos **7** mayor que igual a **7**. Esto es verdadero y el resultado final es **true**. De igual manera tenemos al operador `<=`, pero éste evalúa a la inversa.

a <= b

La expresión a evaluar es **5** menor que igual a **7**, y obtenemos **true** como resultado.

(a + c) <= b

En este caso tenemos **10** menor que igual a **7** y resulta falso.

(a + 2)<= b

En esta última expresión **7** menor que igual a **7** es verdadero y nuevamente obtenemos **true** como resultado.

El uso de if

Ya aprendimos a usar las expresiones relacionales y el tipo de valor devuelto. Es el momento de hacer algo práctico con ellas. Este tipo de expresiones se usa en diferentes casos, pero generalmente en las estructuras selectivas o repetitivas.

Las estructuras selectivas serán estudiadas en este capítulo, las repetitivas en un capítulo posterior. Las **estructuras selectivas** son aquellas que nos permiten hacer una selección entre dos o varias rutas de ejecución posibles. La selección se llevará a cabo según el valor de una expresión. Esta expresión puede ser una expresión relacional. Nuestra primera estructura selectiva se conoce como **if**, que es un **si condicional**. Si tal cosa sucede, entonces haz tal cosa. El uso del **if** es sencillo:

```
if(expresión)
    Sentencia a ejecutar
```

El uso de **if** requiere que coloquemos una expresión a evaluar entre paréntesis. Si el resultado de esta expresión es **true**, entonces la sentencia a ejecutar se lleva a cabo.

Si el resultado de la evaluación es **false**, entonces la sentencia a ejecutar nunca se lleva a cabo, o dicho de otra forma, es ignorada.

En el diagrama de flujo **if** se representa por medio de un **rombo**. En el interior del rombo colocamos la expresión a evaluar, de las esquinas sacamos una ruta de ejecución en el caso de que sí se cumpla la condición o en el caso de que no se cumpla.

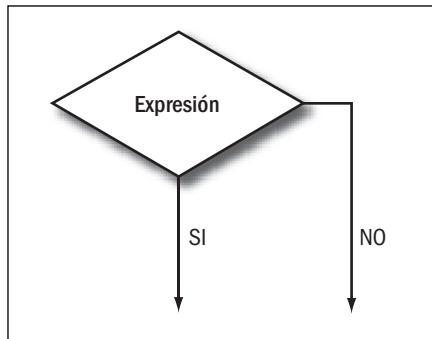


Figura 1. Este rombo simboliza a **if** con la expresión en su interior.

Veamos un primer ejemplo donde nos puede servir **if** y las expresiones. Crearemos un programa que le pida al usuario un número, y la computadora debe decir si el número es positivo o negativo. Para ver el algoritmo de este programa, creamos su diagrama de flujo e incluimos los **if** necesarios.

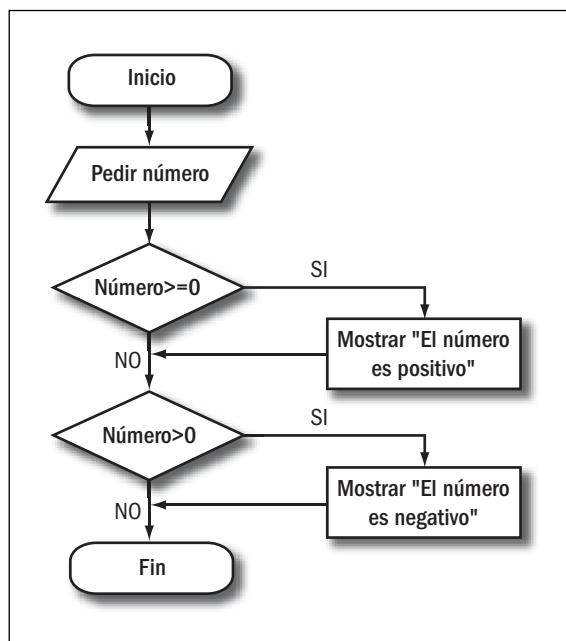


Figura 2. Éste es el diagrama de flujo del algoritmo para resolver el problema. Podemos observar que tenemos dos rutas de ejecución en cada **if**.

```

using System;
using System.Collections.Generic;
using System.Text;

namespace AplicacionBase
{
    class Program
    {
        // Esta es la función principal del programa
        // Aquí inicia la aplicación
        static void Main(string[] args)
        {
            int numero = 0; // Donde guardamos el número
            string valor = ""; // Para guardar la cadena dada por el usuario

            // Pedimos el número
            Console.Write("Dame un numero entero:");
            Valor = Console.ReadLine();
            numero = Convert.ToInt32(valor); // Convertimos la cadena a entero

            // Hacemos uso de if con la expresión para el caso de los positivos
            if (numero >= 0)
                Console.WriteLine("El numero {0} es positivo", numero); // se ejecuta si se
                cumple numero>=0

            // Hacemos uso de if con la expresión para el caso de los negativos
            if (numero < 0)
                Console.WriteLine("El numero {0} es negativo", numero); // se ejecuta si se
                cumple numero<0
        }
    }
}

```

III EL USO CORRECTO DE LA SENTENCIA CON IF

Cuando hacemos uso de **if**, nunca debemos colocar punto y coma después del paréntesis ya que hacerlo no es un error de sintaxis sino de lógica. El compilador interpreta esto como si deseamos que se ejecute una sentencia vacía cuando **if** se cumple. Si vemos que la sentencia **if** siempre se cumple sin importar cómo se evalúa la expresión, es posible que tengamos este error.

```

    }
}

```

La primera parte del programa es de muy fácil comprensión. Luego de ésta, nos encontramos con la primera sentencia **if**. Para este **if** se tiene la expresión **numero >= 0**. Si esa expresión se evalúa como **true**, entonces la siguiente sentencia se ejecuta, de lo contrario será ignorada y continuará el programa.

Supongamos que el valor guardado en **numero** es **3**, entonces la expresión relacional regresa un valor de **true**. Al obtener **true**, **if** lleva a la ejecución y el mensaje que dice que se ejecuta un número positivo.

Luego se continúa con el siguiente **if**. Para éste, la expresión es **numero < 0**, como en este ejemplo **numero** vale **3**, entonces la expresión se evalúa como **false**, por lo que, mientras el valor de la expresión sea **false**, **if** no ejecutará en ningún momento el mensaje que dice que el número es negativo. Esto es exactamente lo que deseamos que realice desde un principio este programa.

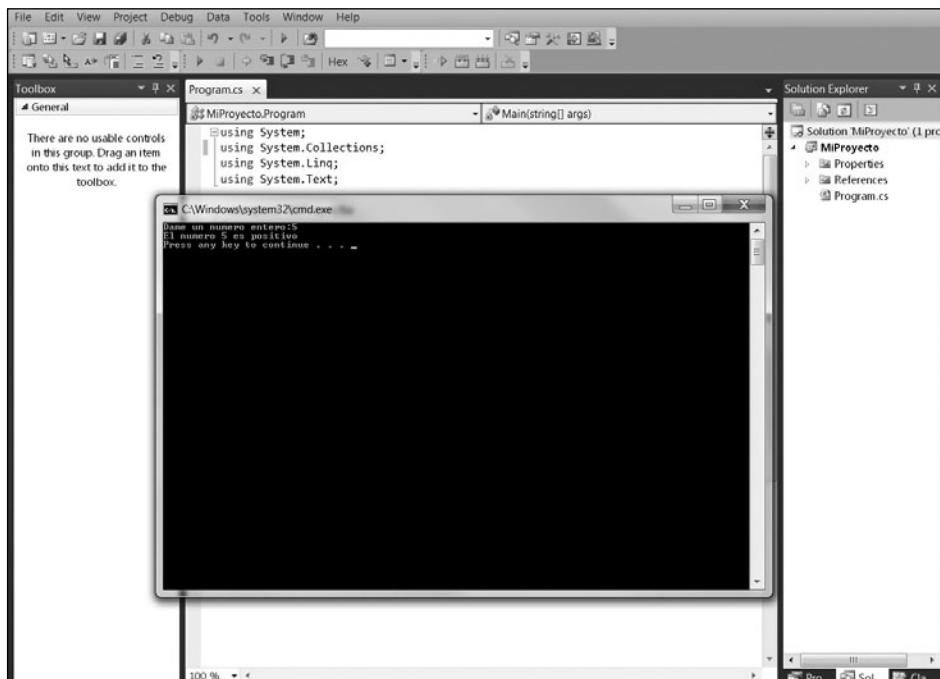


Figura 3. Éste es el resultado de la ejecución cuando pasamos un número positivo al programa.

Éste es un buen momento para comenzar a experimentar en nuestros desarrollos, cambiando los valores, tanto negativos como positivos, incluido el cero y ver cómo responderá nuestro programa.

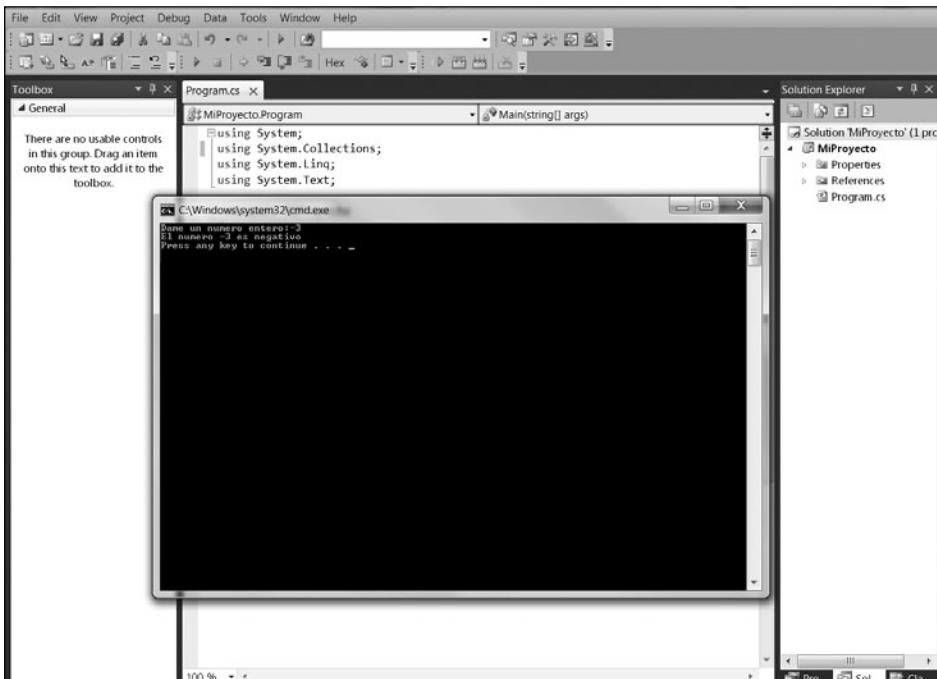


Figura 4. En este caso damos un número negativo y podemos observar el comportamiento del programa.

Bloque de código con if

Como vimos, **if** efectivamente puede sernos de mucha utilidad para que el programa lleve a cabo la decisión correcta, sin embargo, solamente ha podido ejecutar una sentencia. Con frecuencia nos encontraremos con el caso cuando necesitamos que se ejecute más de una sentencia si la condición se cumple. Una buena opción para resolver este inconveniente encontrado es colocarle un **bloque de código** a **if**. En este caso, el bloque de código se ejecutará siempre que la expresión dentro de **if** sea evaluada como **true**, y el bloque de código completo será ignorado siempre que la expresión sea evaluada como **false**.

El bloque de código es un conjunto de sentencias agrupadas y debe ser abierto con **{** y cerrado con **}**, quedará compuesto de la siguiente forma:

```
if(expresión)
{
    Sentencia 1;
    Sentencia 2;
    ...
    Sentencia n;
}
```

Veamos un ejemplo donde hacemos uso de esto. Sabemos que la división entre cero no está definida, entonces debemos hacer un programa que le pregunte al usuario el dividendo y el divisor, pero que cuando el divisor sea cero no lleve a cabo la división. Como siempre, primero hacemos nuestro análisis y creamos el algoritmo en forma de diagrama de flujo.

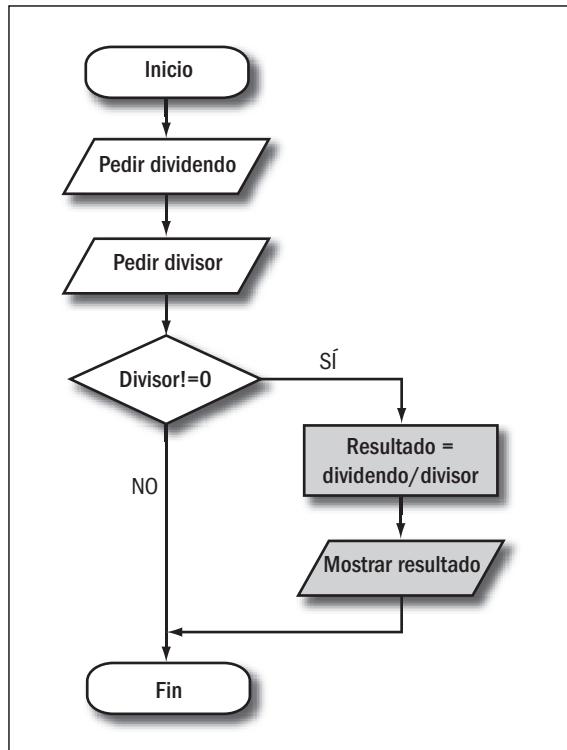


Figura 5. Éste es el diagrama de flujo del algoritmo. Podemos pensar que lo que hemos agrupado es lo que se encuentra adentro del bloque de código de *if*.

El programa queda de la siguiente forma:

```

using System;
using System.Collections.Generic;
using System.Text;

namespace AplicacionBase
{
    class Program
    {
        // Esta es la función principal del programa
  
```

```

// Aquí inicia la aplicación
static void Main(string[] args)
{
    // Variables necesarias
    float dividendo = 0.0f;
    float divisor = 1.0f;
    float resultado = 0.0f;
    string valor = "";

    // Pedimos el dividendo
    Console.WriteLine("Dame el dividendo:");
    valor = Console.ReadLine();
    dividendo = Convert.ToInt32(valor);

    // Pedimos el divisor
    Console.WriteLine("Dame el divisor:");
    valor = Console.ReadLine();
    divisor = Convert.ToInt32(valor);

    // Si el divisor es válido, entonces hacemos la división
    if (divisor != 0.0f)
    {
        // Hacemos la operación
        resultado = dividendo / divisor;

        // Mostramos el resultado
        Console.WriteLine("El resultado de la división es {0}",
                          resultado);
    }
}
}

```

Podemos observar que hemos colocado un bloque de código para **if**. Si la expresión de **divisor != 0.0f** es verdadera, entonces se ejecuta el bloque de código donde tenemos la operación y el despliegado del resultado. Si la expresión se evalúa como falsa, todo el bloque de código es ignorado y no ejecutan las sentencias que están en su interior. Ahora podemos compilar el programa y dar valores para hacer pruebas y observar el comportamiento. Vemos que efectivamente cuando el divisor tiene un valor de cero, la operación y el mensaje no se ejecutan.

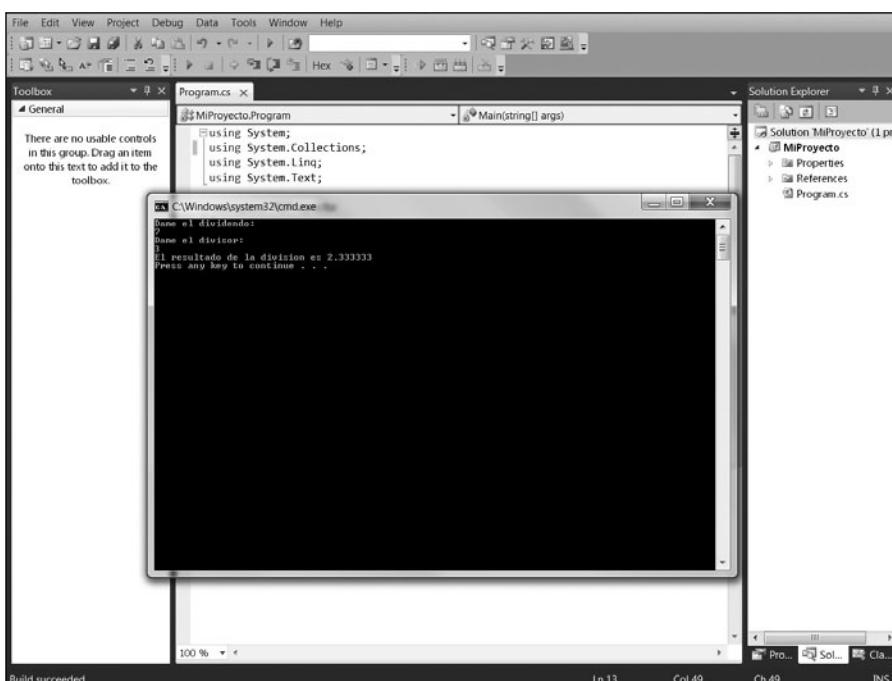


Figura 6. En este caso visto aquí, es ejecutada la división y a continuación se muestra el resultado.

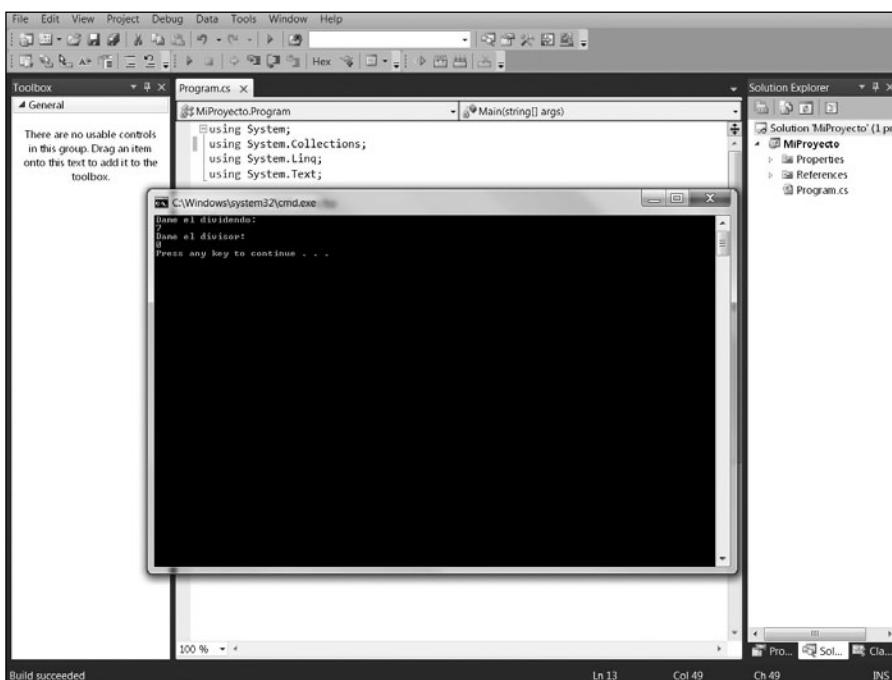


Figura 7. Como hemos colocado el valor de cero en el divisor, la división no se lleva a cabo.

El uso de else

Con los programas anteriores con los que estuvimos trabajando, donde utilizamos la sentencia **if**, hemos visto que podemos indicar que determinada sentencia se ejecute cuando deseamos que una condición se cumpla. Sin embargo, puede ocurrir que a veces necesitamos que una sentencia se ejecute cuando se cumple la condición y que otra sentencia se ejecute cuando esa condición no se cumpla. Si nos encontramos en este caso, una forma de resolverlo sería colocar un **if** con una condición y luego otro **if** con una condición que sea complemento de la primera. Esto ya lo hemos hecho anteriormente en el programa de los números positivos y negativos. Sin embargo, existe otra forma más sencilla que la utilizada en este programa para hacerlo. Esta forma puede sernos de mucha utilidad para simplificar la lógica y que no nos veamos obligados de tener que colocar tantos **if** dentro de una sentencia. Para lograr esto haremos uso de **else**.

Siempre utilizaremos **else** a continuación de una sentencia **if**, ya que **else** no se puede usar sólo. Tenemos que colocar nuestro código de la siguiente forma:

```
if(condición)
    Sentencia1;
else
    Sentencia2;
```

Si la condición es evaluada como verdadera, entonces se ejecuta la sentencia 1, en cambio, cuando la condición se evalúa como falsa, se ejecuta la sentencia 2. A continuación analizaremos cómo hacer uso de este caso para simplificar nuestro código. Para esto elegimos para representar este ejemplo el programa de los números positivos y negativos.

Lo primero que tenemos para analizar es el gráfico del **diagrama de flujo**, y, como utilizamos una estructura o sentencia de tipo **if-else**, una de las salidas del rombo corresponde al código que se ejecuta cuando la expresión es evaluada como verdadera (**if**) y la otra salida es la que utilizamos para la sentencia que se ejecuta cuando la expresión es evaluada como falsa (**else**).

III VERIFICAR EL PROGRAMA

Siempre es conveniente verificar la ejecución del programa y colocar valores que puedan provocar problemas con el fin de detectar cualquier error de lógica. Podemos probar valores positivos, negativos, el cero, fraccionarios, etcétera. Si encontramos problemas con los valores, podemos hacer uso del **if** para validar los valores antes de usarlos.

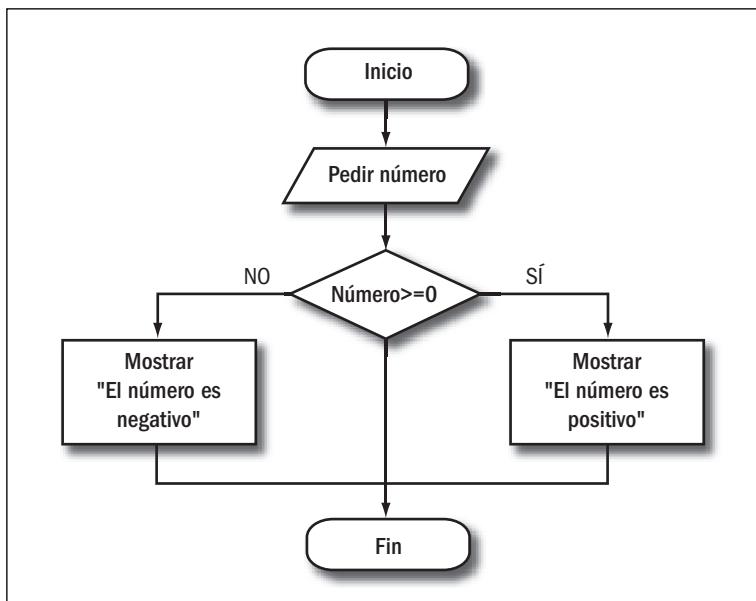


Figura 8. Aquí vemos claramente la ruta de *if* y la ruta de *else*.

El código de nuestro programa quedaría compuesto de la siguiente forma:

```

using System;
using System.Collections.Generic;
using System.Text;

namespace AplicacionBase
{
    class Program
    {
        // Esta es la función principal del programa
        // Aquí inicia la aplicación
        static void Main(string[] args)
    }
}
  
```

III LAS SENTENCIAS EN IF-ELSE

Cuando tenemos una estructura del tipo **if-else**, solamente la sentencia **if** o la sentencia **else** se ejecutarán. En ningún caso se ejecutarán las dos sentencias. Solamente una de ellas puede ejecutarse. Es importante tener esto en cuenta para evitar errores de lógica y no olvidar que **else** siempre requiere de un **if**.

```

{
    int numero = 0; // Donde guardamos el número
    string valor = ""; // Para guardar la cadena dada por el
                       usuario
    // Pedimos el numero
    Console.WriteLine("Dame un número entero:");
    valor = Console.ReadLine();
    numero = Convert.ToInt32(valor); // Convertimos la cadena a
                                    entero
    // Hacemos uso de if con la expresión para el caso de los
    positivos
    if (numero >= 0)
        Console.WriteLine("El número {0} es positivo", numero); // se
        ejecuta si se cumple numero>=0
    else
        Console.WriteLine("El numero {0} es negativo", numero); // se
        ejecuta si NO se cumple numero>=0

}
}
}

```

Podemos ver que el programa es más sencillo, se ha simplificado la lógica y esto lo hace más fácil de leer y mantener. Esto no significa que siempre debamos usar **if-else**. Hay que hacer uso de él cuando tenemos algo que deseamos ejecutar cumpliendo previamente la condición especificada, y algo que deseamos ejecutar cuando ésta no se cumpla. Abusar de **if-else** o usarlo sin sentido complicará la lógica del programa, pero usarlo bien nos ayuda.

El uso de **else** con bloque de código

Al igual que con **if**, nosotros podemos colocar un bloque de código en **else**. Esto nos permitiría que más de una sentencia se ejecute cuando no se cumple la condición del **if**. No debemos olvidar colocar el bloque de código empezando con una llave, **{**, y finalizándolo con la llave de cierre **}**. Adentro del bloque de código podemos colocar cualquier sentencia válida de C#.

Por ejemplo, podemos modificar el programa de la división para hacer uso de la sentencia **else**. En este caso, lo que haremos será que, cuando el divisor sea igual a cero enviaremos a la consola un mensaje de error y cuando no lo sea, llevaremos a cabo la división. Primero veamos cómo es el diagrama de flujo con estos cambios. Es fácil notar la ruta de **else**.

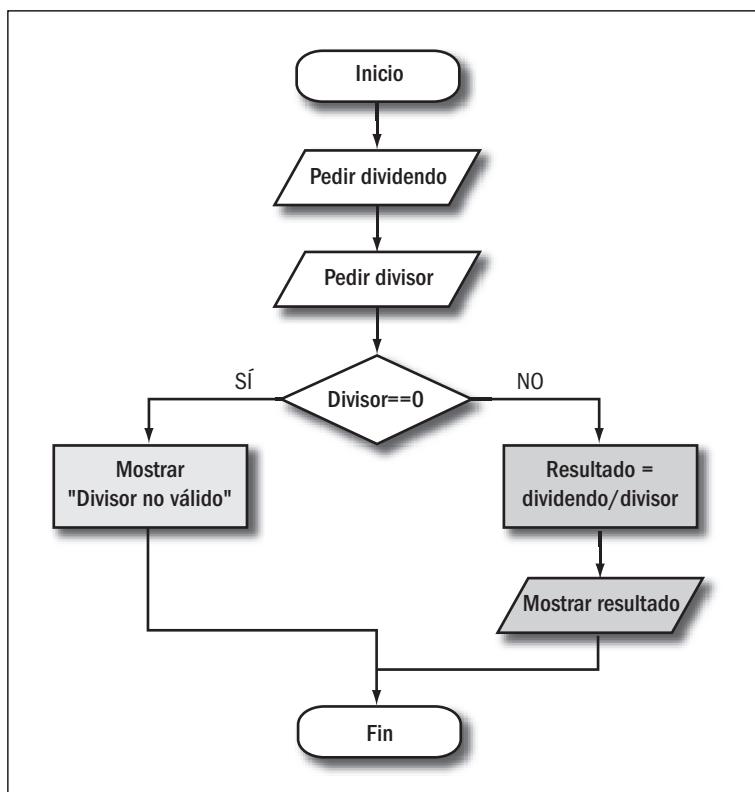


Figura 9. Aquí podemos ver el diagrama de flujo con las dos rutas de ejecución posibles, una para *if* y la otra para *else*.

El código del programa se modifica de la siguiente forma:

```

using System;
using System.Collections.Generic;
using System.Text;

namespace AplicacionBase
{
    class Program
    {
        // Esta es la función principal del programa
        // Aquí inicia la aplicación
        static void Main(string[] args)
        {
            // Variables necesarias
            float dividendo = 0.0f;
  
```

```

float divisor = 1.0f;
float resultado = 0.0f;
string valor = "";

// Pedimos el dividendo
Console.WriteLine("Dame el dividendo:");
valor = Console.ReadLine();
dividendo = Convert.ToInt32(valor);

// Pedimos el divisor
Console.WriteLine("Dame el divisor:");
valor = Console.ReadLine();
divisor = Convert.ToInt32(valor);

if (divisor == 0)
    Console.WriteLine("El divisor no es válido");
else
{
    // Hacemos la operación
    resultado = dividendo / divisor;

    // Mostramos el resultado
    Console.WriteLine("El resultado de la división es {0}",
                      resultado);
}
}
}
}

```

De esta forma podemos manejar dos opciones, una para el divisor con el valor de cero y otra para cuando el divisor es válido.

III BLOQUES DE CÓDIGO CON IF-ELSE

Cuando hacemos uso de **if-else**, cualquiera de los dos o ambos pueden tener una sentencia o un bloque de código. No es necesario que ambos sean iguales. La selección entre la sentencia o el bloque de código nos la da el algoritmo y las necesidades de lógica que tengamos. Algunos programadores colocan el bloque de código por costumbre, aunque éste sólo tenga una sentencia.

Cómo usar if anidados

La sentencia o el bloque de código que ejecuta **if** puede ser cualquier sentencia válida o bloque de código válido en C#, esto incluye a otro **if**. Esto significa que nosotros podemos colocar **if** adentro del código a ejecutar de un **if** anterior. Cuando hacemos esto hay que ser cuidadosos para evitar errores de lógica. Esto se conoce como **if anidados**. Veamos un ejemplo de esto. Tenemos que hacer un programa que pedirá el tipo de operación aritmética que deseamos ejecutar y luego los dos números con los que llevar a cabo la operación. El resultado se desplegará en la pantalla.

El programa tendrá el siguiente código:

```
using System;
using System.Collections.Generic;
using System.Text;

namespace AplicacionBase
{
    class Program
    {
        // Esta es la función principal del programa
        // Aquí inicia la aplicación
        static void Main(string[] args)
        {
            // Variables necesarias
            float a = 0.0f;
            float b = 0.0f;
            float resultado = 0.0f;
            string valor = "";
            int opcion = 0;

            // Mostramos el menú
            Console.WriteLine("1- Suma");
            Console.WriteLine("2- Resta");
            Console.WriteLine("3- División");
            Console.WriteLine("4- Multiplicación");
            Console.Write("Que operación deseas hacer: ");
            valor=Console.ReadLine();
            opcion = Convert.ToInt32(valor);

            // Pedimos el primer número
            Console.Write("Dame el primer numero:");
        }
    }
}
```

```

        valor = Console.ReadLine();
        a = Convert.ToSingle(valor);
        // Pedimos el segundo número
        Console.Write("Dame el segundo numero:");
        valor = Console.ReadLine();
        b = Convert.ToSingle(valor);
        // Verificamos para suma
        if (opcion == 1)
            resultado = a + b;
        // Verificamos para resta
        if (opcion == 2)
            resultado = a - b;
        // Verificamos para division
        if (opcion == 3)
            if (b != 0) // este if esta anidado
                resultado = a / b;
            else // Este else pertenece al segundo if
                Console.WriteLine("Divisor no valido");
        // Verificamos para la multiplicacion
        if (opcion == 4)
            resultado = a * b;
        // Mostramos el resultado
        Console.WriteLine("El resultado es: {0}", resultado);
    }
}
}

```

El programa es sencillo, solamente necesitamos **if** para cada operación, pero para el caso de la división necesitaremos colocar otro **if** para verificar que el divisor no sea cero. Es decir que para la división tendremos **if** anidados. El diagrama de flujo de la aplicación se muestra en la siguiente figura.

III UN IF ADENTRO DE OTRO IF

Cuando colocamos un **if** adentro del código que puede ejecutar un **if** anterior, estamos diciendo que tenemos **if** anidados. Este tipo de estructura es muy útil y puede ayudarnos a optimizar los programas, pero debemos tener cuidado de usarla correctamente para evitar errores de lógica. Cuando se codifica hay que hacerlo en forma ordenada.

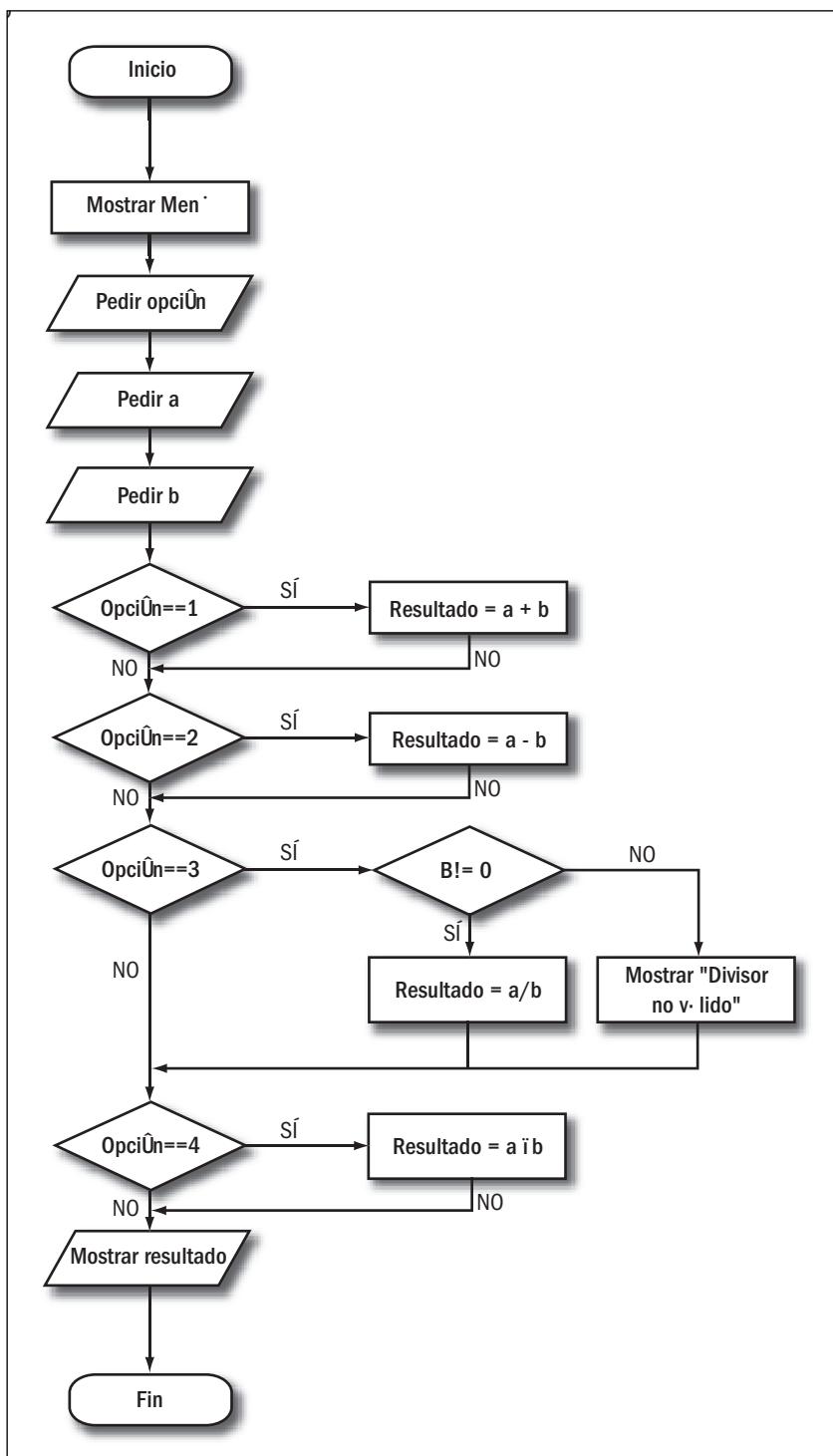


Figura 10. Éste es el diagrama de flujo.
Es fácil observar en la división el anidamiento de *if*.

En el programa anterior observamos cómo se ha usado **if** anidado para la división, sólo tenemos que ser cuidadosos con **else** para poder saber a qué **if** pertenece. Si en nuestros programas hacemos uso de la **indentación**, es decir colocar espacios antes de la sentencia, podemos facilitar reconocer qué parte del código pertenece a qué sección. Algunos editores de código hacen esto por nosotros de forma automática y visualmente es más fácil reconocer el código.

Escalera de if-else

Otra estructura que se puede utilizar es la **escalera de if-else**. Una de sus funciones principales es optimizar la ejecución del código. Al igual que con los **if** anidados, no hay que utilizarlos sin planeación, ya que al no programarlos correctamente, nos puede llevar a errores de lógica que pueden resultar difíciles de solucionar.

Ahora aprendamos cómo los podemos usar para que nuestro código se vuelva más eficiente. Si observamos el programa de las operaciones matemáticas, tenemos un **if** para cada operación. Esto significa que cuando nuestro programa se ejecuta, sin importar si se ha seleccionado la suma o la multiplicación, siempre se tienen que ejecutar al menos cuatro **if**. Los **if** pueden ser operaciones costosas ya que se tienen que evaluar expresiones.

La forma de optimizar esto es con una cadena de **if-else**. El concepto es sencillo y consiste en colocar un **if** en la sentencia del **else**.

Esto quedaría como muestra el siguiente código:

```
if(expresión 1)
    Sentencia 1
else if(expresión 2)
    Sentencia 2
else if(expresión 3)
    Sentencia 3
```

III ¿A QUIÉN LE PERTENECE ELSE?

A veces sucede que cuando tenemos anidamiento, uno o varios de los **if** pueden tener **else** y luego no sabemos a quién pertenece. **Else** siempre pertenecerá al **if** más cercano que ya no tenga un **else** asignado. La mejor forma de evitar problemas con esto es hacer uso de bloques de código, aunque solamente tengamos una sentencia en el bloque.

Como podemos ver se genera una figura que parece una escalera y por eso su nombre. Modifiquemos el programa de las operaciones matemáticas. En primer lugar su diagrama de flujo queda de la forma que se muestra en la figura.

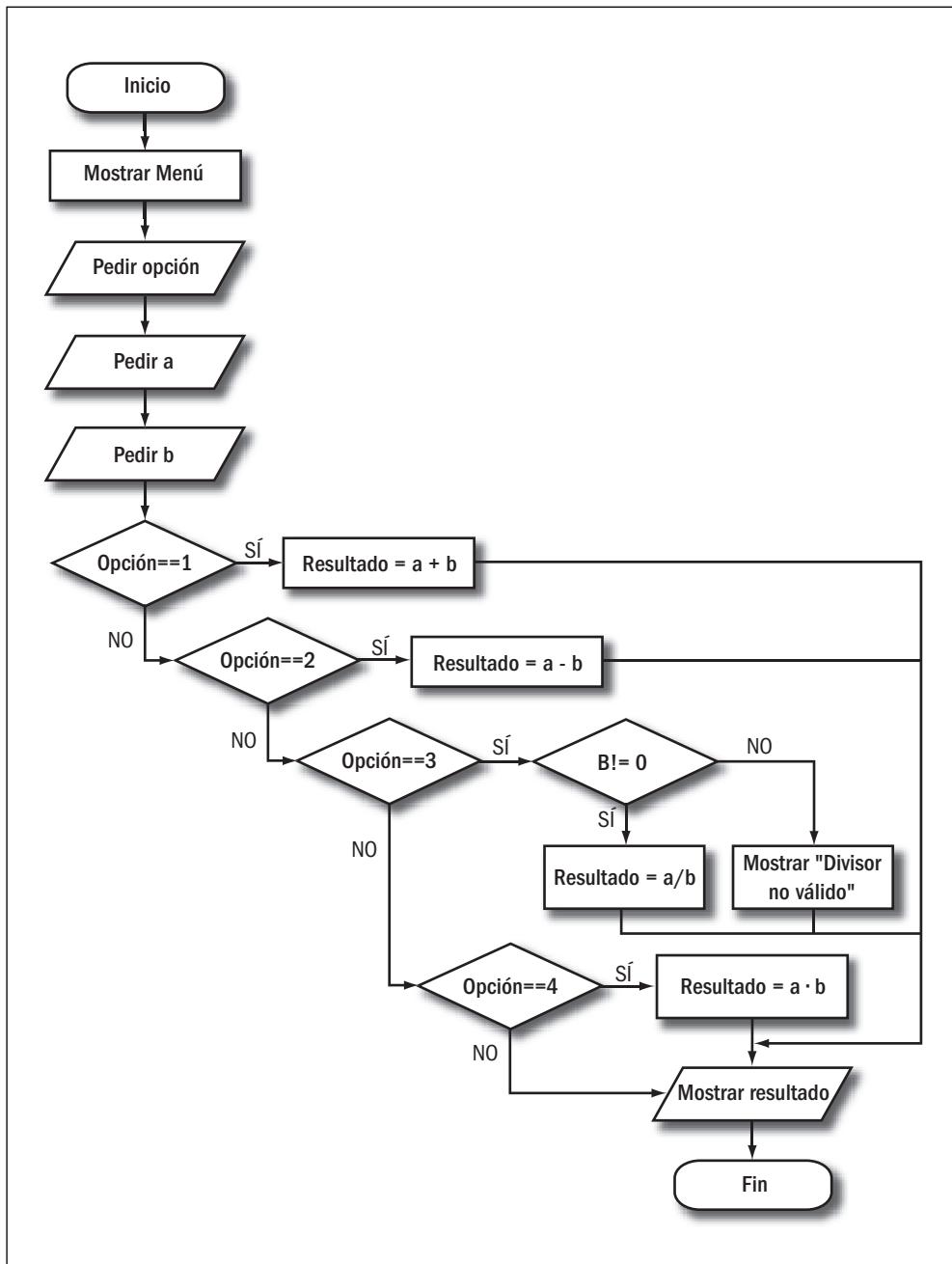


Figura 11. En este diagrama de flujo podemos observar nuestra escalera de *if-else*.

Modifiquemos el código del programa, colocándolo de la siguiente manera:

```
using System;
using System.Collections.Generic;
using System.Text;

namespace AplicacionBase
{
    class Program
    {
        // Esta es la función principal del programa
        // Aquí inicia la aplicación
        static void Main(string[] args)
        {
            // Variables necesarias
            float a = 0.0f;
            float b = 0.0f;
            float resultado = 0.0f;
            string valor = "";
            int opcion = 0;

            // Mostramos el menu
            Console.WriteLine("1- Suma");
            Console.WriteLine("2- Resta");
            Console.WriteLine("3- División");
            Console.WriteLine("4- Multiplicación");
            Console.Write("Que operación deseas hacer: ");
            valor = Console.ReadLine();
            opcion = Convert.ToInt32(valor);

            // Pedimos el primer numero
            Console.Write("Dame el primer numero:");
            valor = Console.ReadLine();
            a = Convert.ToSingle(valor);

            // Pedimos el segundo número
            Console.Write("Dame el segundo número:");
            valor = Console.ReadLine();
            b = Convert.ToSingle(valor);
        }
    }
}
```

```
// Verificamos para suma
if (opcion == 1)
    resultado = a + b;

// Verificamos para resta
else if (opcion == 2)
    resultado = a - b;

// Verificamos para division
else if (opcion == 3)
    if (b != 0) // este if esta
        anidado
        resultado = a / b;
    else // Este else pertenece al segundo if
        Console.WriteLine("Divisor no
válido");

// Verificamos para la multiplicacion
else if (opcion == 4)
    resultado = a * b;

// Mostramos el resultado
Console.WriteLine("El resultado es: {0}", resultado);

}
```

Si ejecutamos el programa, veremos que hace exactamente lo mismo que la versión anterior. ¿Pero dónde está la optimización? Si observamos el código, podemos encontrarla. Supongamos que el usuario ha seleccionado la opción 1, es decir la suma.

{ }

FACILITAR LA LECTURA DE NUESTRO CÓDIGO

Si la escalera **if-else** parece difícil de leer, podemos utilizar bloques de código para agrupar las secciones. Esto hará que entender y leer la lógica de la aplicación sea más fácil. La aplicación se puede resolver con **if** normales, no es forzoso utilizarlos. Se pueden utilizar cuando empezamos a tener más experiencia.

Con esto se cumple la condición del primer **if** y se ejecuta la operación de suma. Pero nosotros sabemos que se ejecuta **if** o **else**, nunca ambos. Si observamos, vemos que todas las demás operaciones están en la ruta de **else**, por lo que son ignoradas.

Antes teníamos que hacer cuatro **if** siempre, ahora solamente se hizo uno y los demás se evitaron, ahí es donde está la optimización. En otras palabras, en el programa anterior siempre eran cuatro **if** y en el nuevo, en el mejor de los casos tenemos un solo **if**, pero en el peor de los casos tenemos cuatro. Así es cómo se llevó a cabo la optimización.

Expresiones lógicas

Al igual que las expresiones aritméticas y relacionales, las expresiones lógicas tienen sus propios operadores. Éstos son: **y**, **o** y **no**. En inglés los conocemos como: **and**, **or** y **not**.

OPERADOR	SIGNIFICADO
&&	y
 	o
!	no

Tabla 2. Aquí tenemos los operadores lógicos utilizados en C#.

El uso de la conjunción

Empecemos a conocerlos. El primer operador es **y**, conocido como **conjunción**. Para usar este operador es necesario tener dos expresiones. Una expresión a la **izquierda** y la otra a la **derecha**. Las expresiones se evalúan devolviendo valores **true** o **false**. Con la conjunción, la expresión se evalúa como verdadera únicamente cuando **ambas expresiones son verdaderas**. La siguiente es la **tabla de verdad** para este operador:

P	Q	P&Q
true	true	true
false	true	false
true	false	false
false	false	true

Tabla 3. La tabla de verdad de la conjunción.

III RECORDAR LA CONJUNCIÓN

Recordar la conjunción es fácil, únicamente es verdadera cuando ambas expresiones son verdaderas. Podemos recordar la regla o simplemente tener una impresión de la tabla a mano. Debemos recordar que se necesitan dos operandos, uno a la derecha y el otro a la izquierda. Con un solo operando no es posible hacerlo.

Supongamos que tenemos que decidir si debemos llenar el tanque de gasolina del vehículo. Esto lo hacemos si el tanque tiene menos del 50% y si la distancia a recorrer es de más de 200 km. Como vemos, tenemos que usar la conjunción ya que tenemos dos condiciones y ambas se deben cumplir para que suceda la carga de la gasolina. Si colocamos esto como expresión quedaría:

```
if(tanque < 50 && distancia > 200)
    Cargar gasolina
```

Supongamos que **tanque** es **25** y **distancia** es **350**, al evaluar **25 < 50 y 350 > 200** tenemos lo siguiente.

```
if(true && true)
    Cargar gasolina
```

En nuestra tabla sabemos que cuando ambas expresiones son verdaderas, entonces se cumple la conjunción y la expresión lógica nos evalúa a **true**.

```
if(true)
    Cargar gasolina
```

Por lo que cargaremos gasolina.

Ahora supongamos que **tanque** tiene el valor de **90** y **distancia** es nuevamente **350**. Las primeras evaluaciones nos darían:

```
if(false && true)
    Cargar gasolina
```

III RECORDAR LA DISYUNCIÓN

Es fácil recordar la tabla de verdad de la disyunción. Ésta es verdadera cuando cualquiera de las expresiones es verdadera. Ya que si una es verdadera o la otra es verdadera, entonces la expresión total es verdadera. También es bueno que tengamos esta tabla a mano hasta que aprendamos la disyunción de memoria.

Al ver nuestra tabla, vemos que **false** y **true** se evalúan como **false**, por lo que no realizaremos la carga de gasolina.

El uso de la disyunción

Para la **disyunción** hacemos uso del operador **o**. Éste también necesita dos expresiones, una en el lado derecho y otra en el lado izquierdo. Esta disyunción tiene la siguiente tabla de verdad:

P	Q	$P \mid\mid Q$
true	true	true
false	true	true
true	false	true
false	false	false

Tabla 4. La tabla de verdad de la disyunción.

Veamos un ejemplo con la disyunción. Tenemos que tomar la sombrilla si llueve o si hay mucho sol. La expresión podría quedar de la siguiente manera:

```
if(lluvia == true || muchosol == true)
    Tomar sombrilla
```

Supongamos que hay lluvia pero no hay sol, tendríamos la expresión:

```
if(true || false)
    Tomar sombrilla
```

En este caso, si vemos la tabla de verdad, podemos observar que el resultado de evaluar la expresión nos da **true**, por lo que sí debemos tomar la sombrilla.

Ahora veamos qué sucede si no hay lluvia y no hay sol. La expresión quedaría como:

III RECORDAR LA NEGACIÓN

La negación simplemente intercambia **true** por **false** y **false** por **true**. Resulta fácil recordarla. Se invierte el valor del operando del lado derecho. No debemos olvidar que este operador solamente necesita un operando y siempre se encuentra del lado derecho. No hacerlo así puede originar problemas con nuestro programa.

```
if(false || false)
    Tomar sombrilla
```

En este caso, la expresión se evalúa como **false** y no se toma la sombrilla.

El uso de la negación

Nos falta conocer un operador más, el de la **negación**. Ese operador es muy sencillo y solamente necesita hacer uso de un operando del lado derecho. Con estas condiciones dadas, esta expresión será negada por el operador.

p	Negación
true	false
false	true

Tabla 5. La tabla de verdad de la negación.

Esto quiere decir que si el operando del lado derecho es **true**, el operador regresa **false**. Y si el operando del lado derecho es **false**, el operador regresa **true**.

Veamos un ejemplo para poder comprender mejor esto.

Tenemos una habitación con una bombilla eléctrica y supongamos que tenemos que prender la luz del cuarto cuando **no es** de día. Entonces nuestra expresión queda de la siguiente forma:

```
if(!dia == true)
    Prender la luz
```

Veamos qué sucede cuando la expresión **dia** tiene como valor verdadero. La expresión **dia == true** se evalúa como **true** y luego éste se niega, lo que al final nos da como resultado un valor **false**. Como **if** recibe el valor **false**, se ejecutará la orden o el método **prender la luz**.

Pero si **dia** es **false**, entonces la expresión **dia == true** se evalúa como **false**, que es negado e **if** recibe **true**. Como **if** recibe **true**, se ejecuta **prender la luz**.

Ahora podemos analizar un ejemplo de cómo hacer uso de una expresión lógica. Esto podemos aplicarlo en nuestro programa de las operaciones matemáticas. Podemos observar que la división se debe hacer cuando el usuario ha seleccionado entre las opciones presentadas, la número 3, y el divisor tiene un valor diferente de cero. Éste es un caso en el que podemos hacer uso de la conjunción. Si procedemos a modificar la aplicación para realizar esto, nuestro diagrama de flujo debería modificarse, quedando similar a la forma que vemos a continuación:

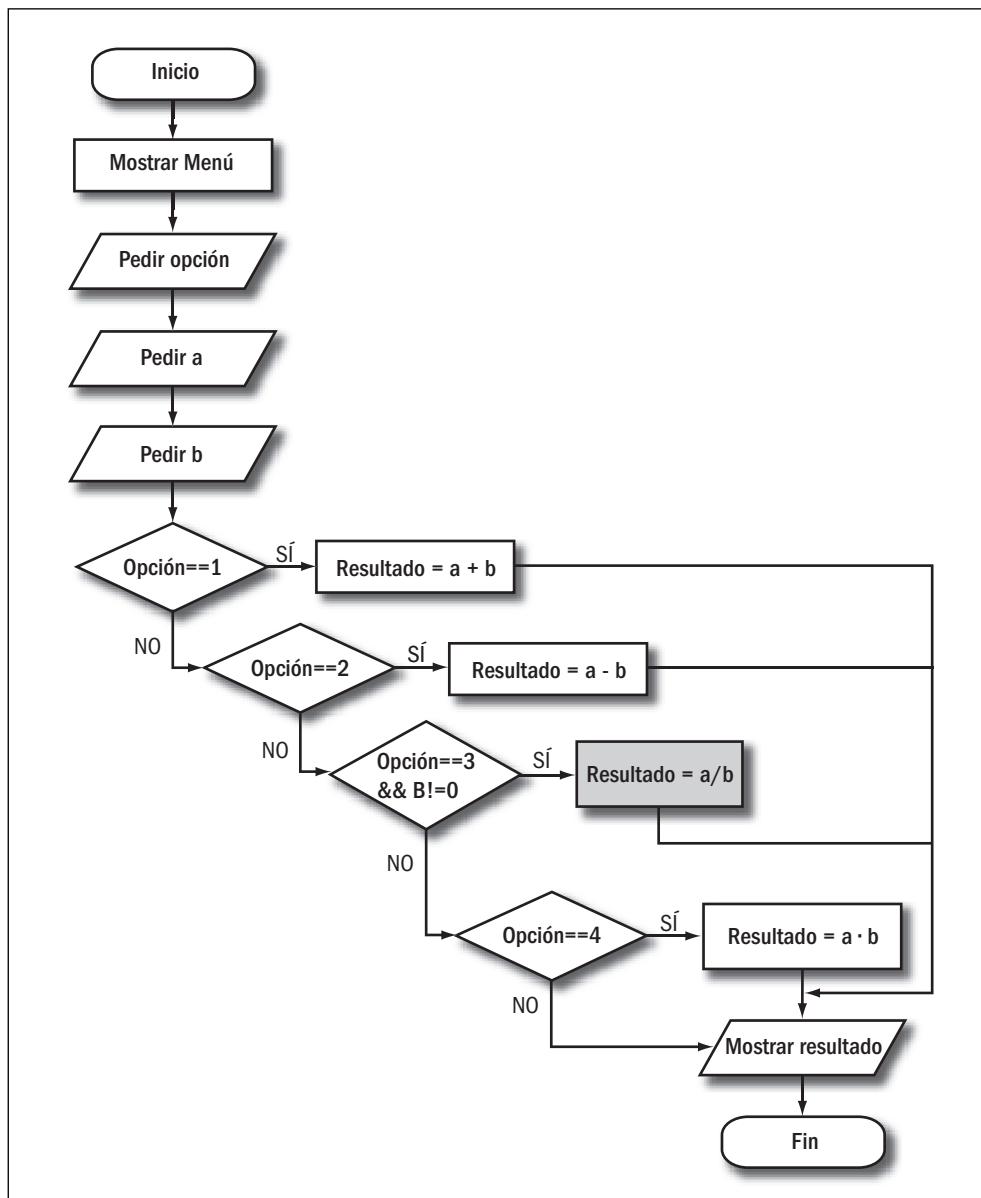


Figura 12. Podemos observar cómo la expresión lógica se encuentra dentro del rombo en la condición de la división.

El código del programa quedaría de la siguiente forma:

```

using System;
using System.Collections.Generic;
using System.Text;

```

```
namespace AplicacionBase
{
    class Program
    {
        // Esta es la función principal del programa
        // Aquí inicia la aplicación
        static void Main(string[] args)
        {
            // Variables necesarias
            float a = 0.0f;
            float b = 0.0f;
            float resultado = 0.0f;
            string valor = "";
            int opcion = 0;

            // Mostramos el menú
            Console.WriteLine("1- Suma");
            Console.WriteLine("2- Resta");
            Console.WriteLine("3- División");
            Console.WriteLine("4- Multiplicación");
            Console.Write("Que operación deseas hacer: ");
            valor = Console.ReadLine();
            opcion = Convert.ToInt32(valor);

            // Pedimos el primer número
            Console.Write("Dame el primer numero:");
            valor = Console.ReadLine();
            a = Convert.ToSingle(valor);

            // Pedimos el segundo número
            Console.Write("Dame el segundo numero:");
            valor = Console.ReadLine();
            b = Convert.ToSingle(valor);

            // Verificamos para suma
            if (opcion == 1)
                resultado = a + b;

            // Verificamos para resta
            else if (opcion == 2)
```

```

        resultado = a - b;

        // Verificamos para división
        else if (opcion == 3 && b!=0) // Aquí usamos una expresión
            lógica
            resultado = a / b;

        // Verificamos para la multiplicación
        else if (opcion == 4)
            resultado = a * b;

        // Mostramos el resultado
        Console.WriteLine("El resultado es: {0}", resultado);

    }
}
}

```

Veamos otro ejemplo. Supongamos que tenemos que hacer un programa que nos indique si una persona puede conducir un automóvil, y las condiciones para que lo conduzca son que tenga más de 15 ó 18 años y que tenga permiso de sus padres. Si observamos el problema vemos que necesitamos dos variables, una para la edad y otra para el permiso de los padres. La expresión lógica podría quedar de la siguiente forma:

```

if(edad > 18 || (edad > 15 && permiso == true))
    Puede conducir

```

Vemos que en este caso usamos una expresión más compleja y al igual que con las expresiones aritméticas, hemos utilizado los paréntesis para ayudarnos y hacer que

III EXPRESIONES COMPLEJAS

Cuando tenemos expresiones complejas, para evitar errores es útil hacer una tabla de verdad de la expresión y probar todos sus casos. De esta forma podemos ver si funciona tal y como lo esperábamos. La tabla de verdad debe ser similar a las que hemos utilizado y nos podemos apoyar en ella para encontrar el valor final de la expresión.

la lógica sea más fácil de leer. Nuestra expresión es una disyunción, en el lado izquierdo del **or** tenemos **edad > 18**, en el lado derecho tenemos una expresión (**edad > 15 && permiso == true**). La segunda expresión es una conjunción y deberá evaluarse primero. El resultado que se obtenga será usado por la disyunción. Ahora podemos hacer el programa de ejemplo para esta expresión. Primero, observemos el diagrama de flujo a continuación:

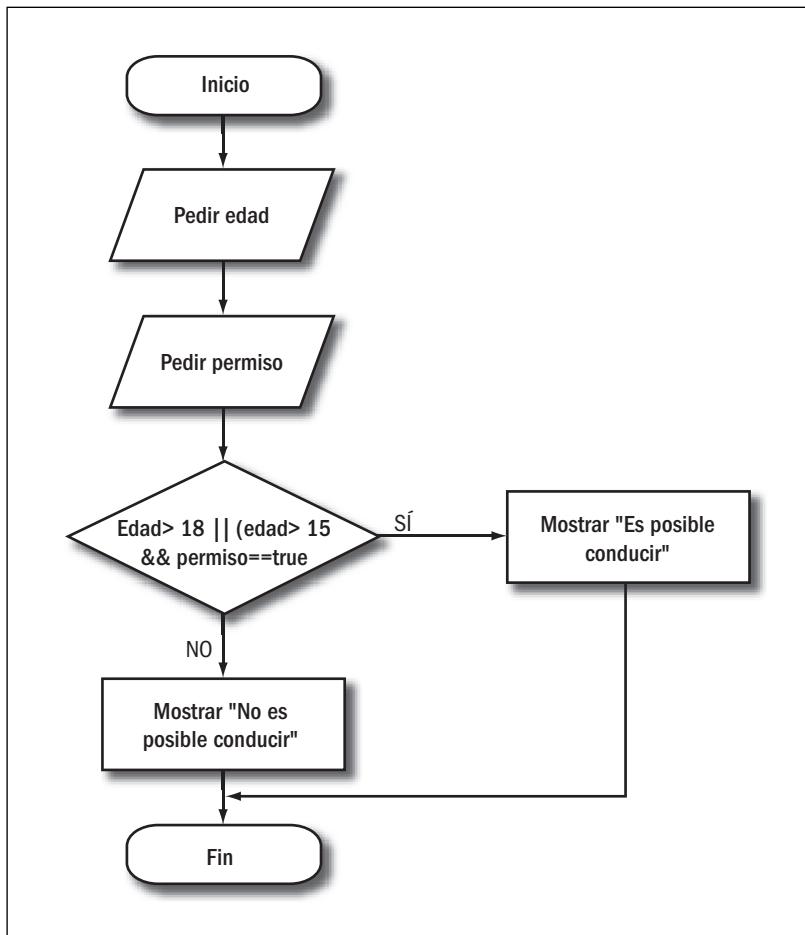


Figura 13. Al igual que en el caso anterior, debemos colocar la expresión dentro del rombo sin importar su complejidad.

El programa podría quedar de la siguiente manera:

```

using System;
using System.Collections.Generic;
using System.Text;
  
```

```

namespace AplicacionBase
{
    class Program
    {
        // Esta es la función principal del programa
        // Aquí inicia la aplicación
        static void Main(string[] args)
        {
            // Variables necesarias
            int edad = 0;
            bool permiso = false;
            string valor = "";

            // Obtenemos los datos
            Console.Write("Dame la edad: ");
            valor = Console.ReadLine();

            edad = Convert.ToInt32(valor);

            Console.Write("Tiene permiso de los padres
(true/false): ");
            valor = Console.ReadLine();

            permiso = Convert.ToBoolean(valor);

            // Verificamos que se cumpla la regla
            if(edad > 18 || (edad > 15 & permiso == true))
                Console.WriteLine("Es posible conducir");
            else
                Console.WriteLine("No puedes conducir");
        }
    }
}

```

III TIPOS DE DATOS PARA EL SWITCH

Con **switch** nosotros podemos comparar una variable contra diferentes valores. La variable puede ser de tipo entero o cadena. El valor del caso debe ser apropiado para que funcione correctamente. De nuestro análisis conocemos los posibles valores para cada caso. El tipo de variable de comparación dependerá del algoritmo en particular.

```

        }
    }
}

```

Ahora aprenderemos a usar otra estructura selectiva.

El uso de **switch**

En uno de los ejemplos anteriores, en los que realizamos las operaciones matemáticas, hemos observado que para cada operación debemos utilizar un **if**. Esto es correcto, y en varias ocasiones veremos que es necesario tomar diferentes opciones dependiendo del valor de una variable.

Como esto puede ocurrir frecuentemente, tenemos una estructura selectiva que nos ayuda y nos permite colocar el código para estos casos con facilidad. Esta estructura se conoce como **switch**. Para ésta necesitamos una variable y varios casos. Por ejemplo, en el programa de las operaciones, cada una de ellas es un caso. El valor de la variable se compara con un valor para cada caso. Si el valor coincide, entonces se empieza a ejecutar el código a partir de esa línea.

Cuando usamos **switch** es necesario colocar entre paréntesis la variable que utilizaremos para llevar a cabo las comparaciones. Luego tenemos que crear un bloque de código y colocar adentro de él los casos y el código a ejecutar para cada caso. Para indicar un caso, usamos **case** seguido del valor de comparación y dos puntos.

Existe un caso llamado **default**, que podemos utilizar si lo deseamos. Este caso siempre debe ser el último caso definido. Cuando la variable de comparación no ha encontrado su valor en ninguno de los casos, entonces se ejecuta el código del caso **default**. En los casos podemos colocar cualquier código C# que sea válido.

Para indicar dónde termina el código de un caso debemos hacer uso de **break**, que nos permitirá salir de la ejecución de una estructura selectiva o repetitiva. Esto lo veremos con más detalle en otro capítulo. Si no hacemos uso de **break** y el caso está vacío, entonces el programa continuará ejecutando el código del próximo caso y así sucesivamente hasta el final del bloque del código que pertenece al **switch**.

III EVITAR ERRORES EN EL SWITCH

Para evitar errores en el **switch** debemos recordar que todos los casos deben terminar con **break** o **return**. Si no lo hacemos así, el compilador nos puede dar problemas. El uso de **return** lo aprenderemos en el capítulo de funciones. Es bueno tener bien definidos los casos para evitar problemas de lógica.

Switch puede ser representado en el diagrama de flujo por medio del **rombo**. En su interior colocamos la variable de comparación y de las esquinas o los lados podemos sacar las diferentes rutas de ejecución que puede tener. Como ejemplo modificaremos el programa de las operaciones aritméticas para que haga uso del **switch**. La variable de comparación será la variable **opción**, ya que el valor de ésta será comparado para cada operación.

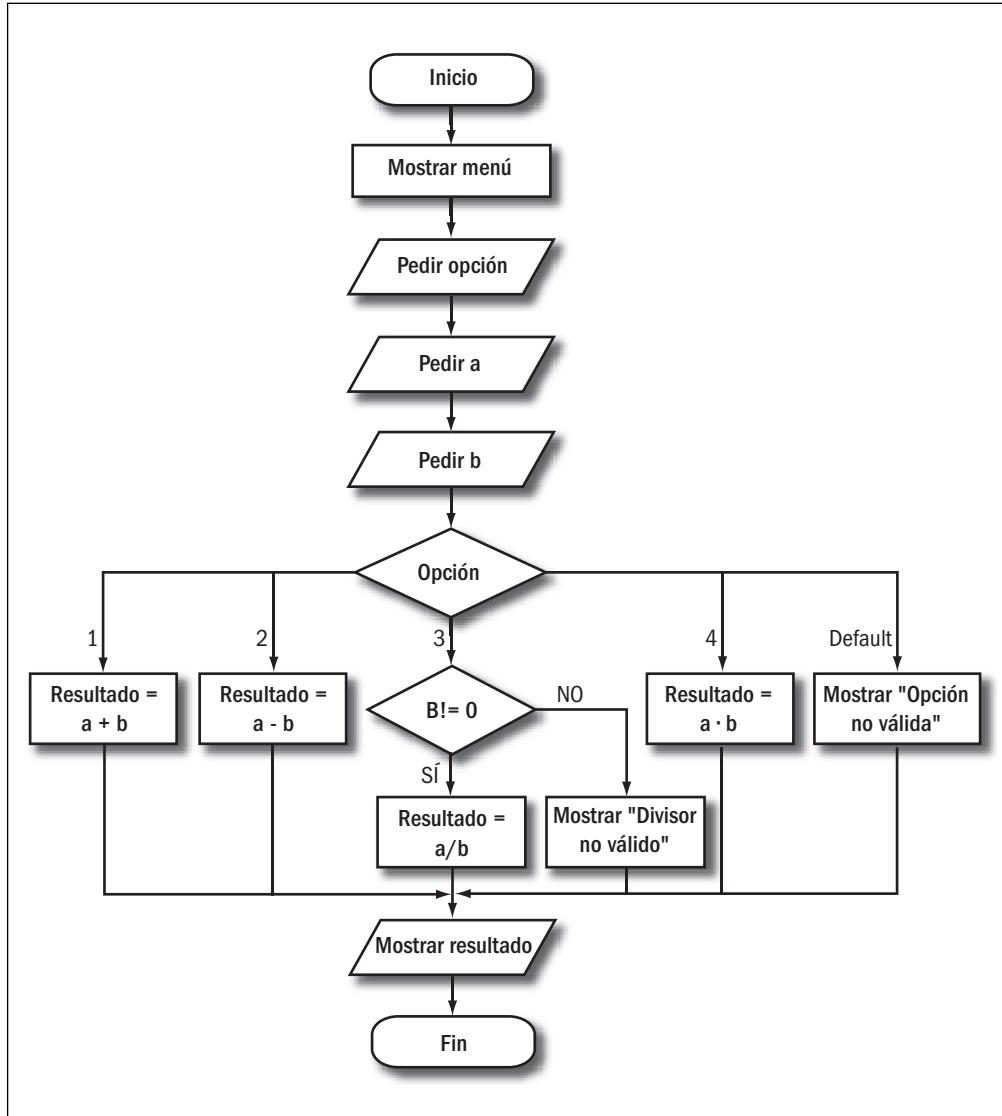


Figura 14. En este caso las diferentes rutas de ejecución del *switch* salen de los lados y las esquinas del rombo.

El código del programa quedará de la siguiente manera:

```

using System;
using System.Collections.Generic;
using System.Text;

namespace AplicacionBase
{
    class Program
    {
        // Esta es la función principal del programa
        // Aquí inicia la aplicación
        static void Main(string[] args)
        {
            // Variables necesarias
            float a = 0.0f;
            float b = 0.0f;
            float resultado = 0.0f;
            string valor = "";
            int opcion = 0;

            // Mostramos el menú
            Console.WriteLine("1- Suma");
            Console.WriteLine("2- Resta");
            Console.WriteLine("3- División");
            Console.WriteLine("4- Multiplicación");
            Console.Write("Que operación deseas hacer: ");
            valor = Console.ReadLine();
            opcion = Convert.ToInt32(valor);

            // Pedimos el primer número
            Console.Write("Dame el primer numero:");
            valor = Console.ReadLine();
        }
    }
}

```

III DIAGRAMAS DE FLUJO

Es posible encontrar diferentes programas de computadora que nos permitan crear diagramas de flujo. Estos programas nos pueden ayudar a hacer nuestros programas más rápido. Un programa que hace esto es Visio de Microsoft, pero también se pueden encontrar programas gratuitos en varios sitios web en Internet como www.download.com.

```
a = Convert.ToSingle(valor);

// Pedimos el segundo número
Console.WriteLine("Dame el segundo numero:");
valor = Console.ReadLine();
b = Convert.ToSingle(valor);

switch (opcion)
{
    // Verificamos para suma
    case 1:
        resultado = a + b;
        break;

    // Verificamos para resta
    case 2:
        resultado = a - b;
        break;

    // Verificamos para división
    case 3:
        if (b != 0) // este if esta anidado
            resultado = a / b;
        else // Este else pertenece al segundo if
            Console.WriteLine("Divisor no valido");
        break;

    // Verificamos para la multiplicación
    case 4:
        resultado = a * b;
        break;

    // Si no se cumple ninguno de los casos
    // anteriores
    default:
        Console.WriteLine("Opción no valida");
        break;
}
```

```
// Mostramos el resultado  
Console.WriteLine("El resultado es: {0}", resultado);  
  
}  
}  
}
```

Con esto hemos visto las estructuras selectivas básicas en C# y cómo podemos usarlas.

RESUMEN

Las estructuras selectivas nos sirven para llevar a cabo la toma de decisiones. Tenemos varias de estas estructuras: if, if-else y switch. Podemos hacer uso de las expresiones relacionales y lógicas en if. Esto nos permite indicar de forma precisa la condición que se debe cumplir para que se ejecute determinado código. El uso de else nos permite tener código que se puede ejecutar cuando la condición no se cumple. Para comparar una variable contra diferentes valores y ejecutar un código en particular cuando su contenido es igual a un determinado valor hacemos uso del switch. Es importante no olvidar el uso del break al finalizar el código de cada caso.



ACTIVIDADES

TEST DE AUTOEVALUACIÓN

- 1** ¿Qué son las estructuras selectivas?
- 2** ¿Cuándo se ejecuta la sentencia de if?
- 3** ¿Qué colocamos entre paréntesis en if?
- 4** ¿Qué es una expresión relacional?
- 5** ¿Cuáles son los operadores de una expresión relacional?
- 6** ¿A qué valores posibles puede evaluar una expresión relacional o lógica?
- 7** ¿Qué es una expresión lógica?
- 8** ¿Cuáles son los operadores de las expresiones lógicas?
- 9** ¿Qué operador solamente necesita un operando?
- 10** ¿Cómo funciona el switch?
- 11** ¿Qué es la variable de comparación y cómo se coloca?
- 12** ¿Cómo definimos los casos y cómo usamos break?

EJERCICIOS PRÁCTICOS

- 1** Hacer un programa que le pida al usuario un número y la computadora responda si es par o impar.
- 2** Hacer un programa que transforme de grados a radianes o de radianes a grados dependiendo de lo que necesite el usuario.
- 3** Hacer un programa que calcule el impuesto de un producto, pero coloque un impuesto del 0% si el producto es medicina.
- 4** Hacer un programa que le pida al usuario un número del 1 al 7 y escriba el nombre del día que corresponde ese número en la semana.
- 5** Hacer una programa que pueda calcular el perímetro y el área de cualquier polígono regular, pero que le pregunte al usuario qué desea calcular.

Creación de ciclos

En la vida real a veces repetimos la misma actividad muchas veces. También en los programas de computadora veremos que es necesario repetir algo un número de veces. Para esto haremos uso de los ciclos, que harán que nuestro código sea más sencillo al mismo tiempo que podemos repetir una actividad cuantas veces sea necesario.

El ciclo for	112
El valor de inicio	117
El límite de conteo del ciclo	119
Control del incremento	121
Ejemplos con el ciclo for	128
El ciclo do while	134
El ciclo while	141
Resumen	145
Actividades	146

EL CICLO FOR

El primer ciclo que aprenderemos en este capítulo se llama ciclo **for**, pero para poder entenderlo mejor necesitamos ver un problema donde sea necesario utilizarlo. Imaginemos que tenemos que crear un programa para una escuela, y este programa debe sacar el **promedio** de las calificaciones para tres alumnos. Si recordamos, el promedio se calcula al sumar todas las cantidades y el resultado de la suma se divide por la cantidad de cantidades que hemos sumado. El programa es sencillo, debido a que en el salón de clases solamente hay tres alumnos.

El diagrama para resolver este programa es el que se muestra en la siguiente figura:

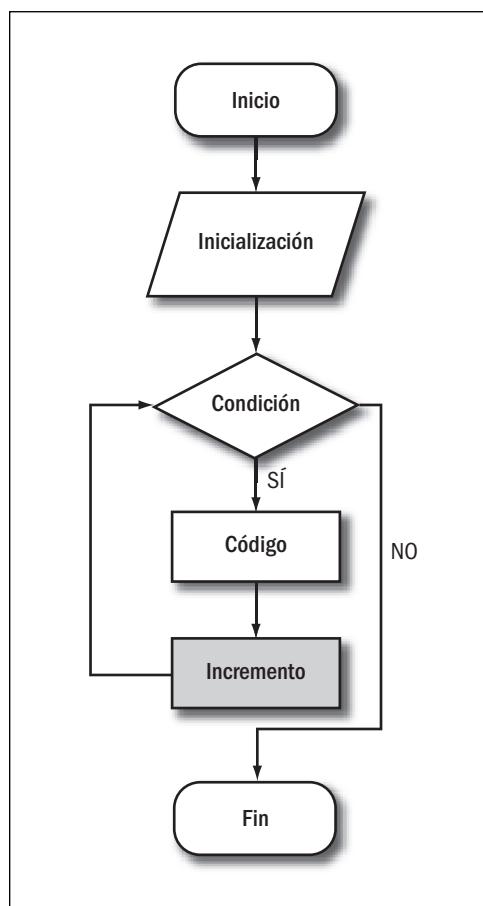


Figura 1. Podemos observar que el programa es muy sencillo.

El código fuente queda de la siguiente manera:

```
using System;
```

```
using System.Collections.Generic;
using System.Text;

namespace AplicacionBase
{
    class Program
    {
        // Esta es la función principal del programa
        // Aquí inicia la aplicación
        static void Main(string[] args)
        {
            // Variables necesarias
            float cal1 = 0.0f, cal2 = 0.0f, cal3 = 0.0f;
            float promedio = 0.0f;
            string valor = "";

            // Pedimos los datos
            Console.WriteLine("Dame la primera calificación:");
            valor = Console.ReadLine();
            cal1 = Convert.ToSingle(valor);

            Console.WriteLine("Dame la segunda calificación:");
            valor = Console.ReadLine();
            cal2 = Convert.ToSingle(valor);

            Console.WriteLine("Dame la tercera calificación:");
            valor = Console.ReadLine();
            cal3 = Convert.ToSingle(valor);

            // Calculamos el promedio
            promedio = (cal1 + cal2 + cal3) / 3;
        }
    }
}
```

III DECLARACIÓN DE LA VARIABLE DE CONTROL

La variable de control que usamos en el ciclo **for** puede ser declarada antes del ciclo, pero en algunos casos es posible declararla e inicializarla en la sección de inicialización del ciclo. Si la declaramos ahí, hay que tomar en cuenta que el **ámbito** de la variable es solamente el ciclo y no se conocerá por afuera de él.

```
// Mostramos el promedio
Console.WriteLine("El promedio es {0}", promedio);

}
}
```

Vemos que el programa es muy sencillo. Pedimos tres valores y calculamos el promedio. Al final simplemente desplegamos el resultado. No hay ningún problema con el programa, de hecho es correcto. Sin embargo, nuestro programa tiene un defecto: es poco flexible.

Ahora imaginemos que la escuela desea el promedio para un grupo con cinco alumnos. Al parecer esto no es difícil, simplemente podríamos agregar dos peticiones de variables más y modificar la fórmula. Esto no parece problemático. ¿Pero qué sucede si el grupo es de quince alumnos? Esta solución de agregar más peticiones, aunque es viable, no es cómoda. ¿Si nos piden el promedio para toda la escuela con 2500 alumnos? Entonces es evidente que no podemos seguir con este tipo de solución.

Sin embargo, de este problema podemos observar algo. Todas las peticiones de calificaciones se hacen de la misma forma y pedir las calificaciones es algo que tenemos que repetir un número de veces. Cuando tenemos código que se debe repetir un número de veces podemos utilizar el ciclo **for**. Éste nos permite repetir la ejecución de un código un número **determinado** de veces.

Antes de resolver nuestro problema con el ciclo **for**, lo primero que tenemos que hacer es aprender a utilizar el ciclo. La sentencia del ciclo se inicia con la palabra clave **for** seguida de paréntesis. Adentro de los paréntesis colocaremos expresiones que sirven para controlar el ciclo. El ciclo **for** tiene cuatro partes principales:

```
for( inicializacion; condicion; incremento)
    código
```

El ciclo **for** necesitará una o varias **variables de control**, aunque en la mayoría de los casos usaremos únicamente una variable. Veamos las diferentes partes del ciclo **for**. En primer lugar encontramos la **inicialización**. En esta sección le damos a la variable de control su valor inicial. El valor inicial se lleva a cabo por medio de una **asignación** normal, tal y como las que hemos trabajado. La segunda sección lleva una **condición** en forma de una expresión relacional. Ésta nos sirve para controlar cuándo termina el ciclo y generalmente aquí indicamos la cantidad de vueltas que da el ciclo. Luego tenemos el **código**. En esta sección colocamos la parte del código que deseamos que se repita. Esta sección puede ser una sentencia o un bloque de código. Por

último, se ejecuta la sección del **incremento**. Aquí indicamos cómo se modificará el valor de la variable de control para cada vuelta del ciclo.

Cuando se ejecuta el ciclo **for**, lo primero que se lleva a cabo es la inicialización y luego la condición. Si la condición es verdadera, entonces se pasa al código y después al incremento. Seguido del incremento vamos nuevamente a la condición y se repite el proceso. Si la condición no se cumple, entonces decimos que el ciclo finaliza, se salta al código y continúa la ejecución del programa con lo que sea que encontremos después del ciclo.

Esto lo podemos apreciar mejor en el siguiente diagrama de flujo.

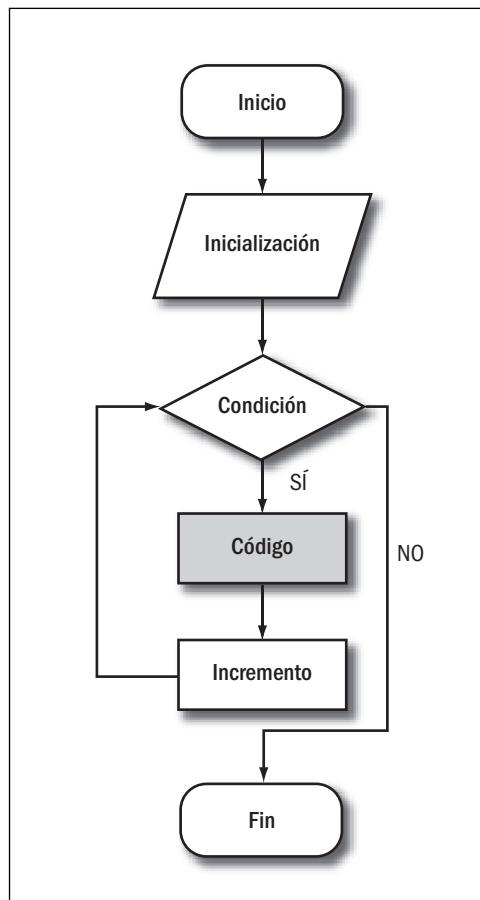


Figura 2. Este diagrama de flujo nos muestra los pasos de la ejecución del ciclo **for**.

El poder comprender el ciclo **for** requiere de mucha experimentación, por lo que faremos varias pruebas para conocer sus características principales.

Empecemos con un programa sencillo. Usaremos el ciclo para mostrar un mensaje, que únicamente imprimirá el valor de la variable de control del ciclo. Para saber cuál es el trabajo que realiza el ciclo, colocaremos un mensaje antes y después del ciclo.

Para iniciar con facilidad, queremos que el ciclo se lleve a cabo diez veces. Primero veamos el programa y luego analicemos cada una de las partes del ciclo. Nuestro programa queda de la siguiente forma:

```

using System;
using System.Collections.Generic;
using System.Text;

namespace AplicacionBase
{
    class Program
    {
        // Esta es la función principal del programa
        // Aquí inicia la aplicación
        static void Main(string[] args)
        {
            // Variables necesarias
            int n = 0; // variable de control

            Console.WriteLine("-- Antes del ciclo --");

            for (n = 1; n <= 10; n = n + 1)
                Console.WriteLine("{0}", n);

            Console.WriteLine("-- Después del ciclo --");
        }
    }
}

```

Antes de ejecutar el programa, veamos cómo está construido. En primer lugar declaramos nuestra variable de control. Ésta se llama **n**. El programa imprimirá los números del **1** al **10** y **n** cambiará su valor según se repita el ciclo.

Adentro del ciclo for encontramos inmediatamente que a **n** se le asigna el valor **1**. Esto quiere decir que nuestro conteo empezará con el número **1**. Si deseáramos iniciar en otro valor, en esta parte es dónde lo asignaríamos. Luego tenemos la condición. La condición limita hasta donde contaremos con **n**. En este caso es hasta **10**. Mientras **n** tenga un valor menor o igual a **10** el mensaje se muestra.

Como en este caso **1 <= 10** se cumple, ya que **n** vale **1**, entonces el mensaje se escribe. El mensaje es muy sencillo ya que simplemente muestra el valor contenido en la variable **n**. Después de esto, vamos al incremento. El incremento aumenta el valor

de **n** en uno. Después del primer incremento **n** valdrá **2**. Esto se repite y se incrementa en **1** el valor de **n** cada vez que se repite el ciclo. En el último incremento, **n** tendrá el valor de **11** y en este caso la condición ya no se cumplirá y se termina con el ciclo. Veamos la ejecución del programa y observaremos cómo **n** cambia de valor y efectivamente el ciclo se lleva a cabo solamente **10** veces.

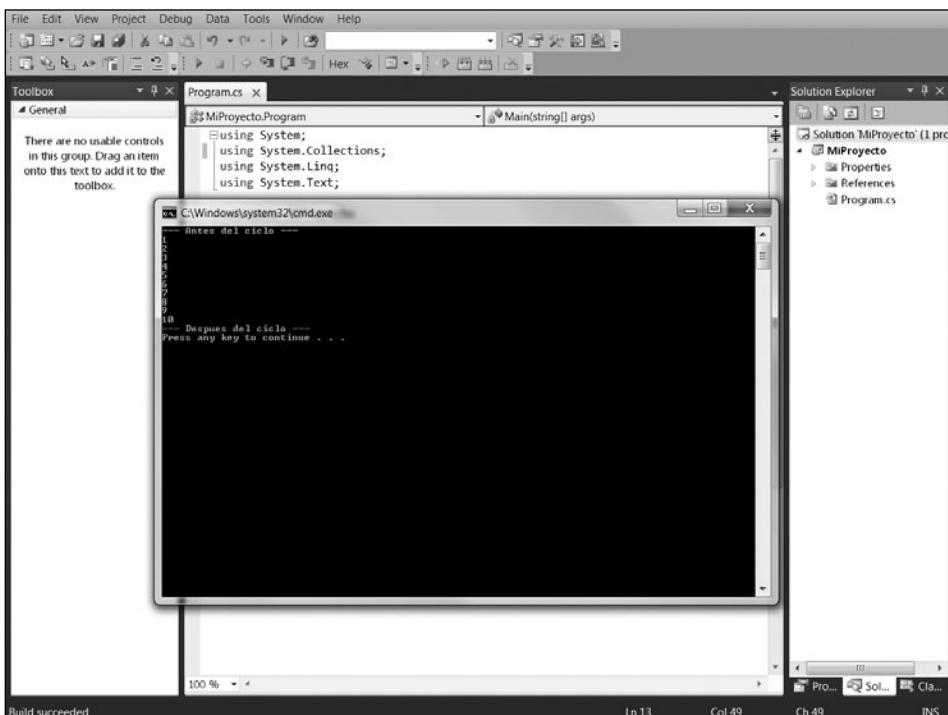


Figura 3. Aquí podemos observar cómo el ciclo se repitió y el valor de **n** fue modificado en cada vuelta.

El valor de inicio

Ahora ya podemos empezar a experimentar con el ciclo. Lo primero que haremos será colocar el valor de inicio del ciclo y ver cómo se modifica la ejecución del



CICLOS QUE EMPIEZAN EN CERO

En C# veremos que en muchos casos tenemos estructuras con índice cero. Esto quiere decir que su primer elemento se considera en la posición **0**, no en la posición **1**. Aunque nos parezca extraño empezar a contar desde cero, en la computadora es muy común que esto ocurra, por lo que nos conviene acostumbrarnos a hacer ciclos de esta forma.

programa. Esto es útil ya que no siempre es necesario empezar a contar a partir de **1**, a veces necesitamos empezar a contar a partir de otros números.

Supongamos que ahora tenemos que contar del **3** al **10**. Para lograr esto, simplemente modificamos el valor de inicio en la sección de inicialización del ciclo:

```
for (n = 3; n <= 10; n = n + 1)
    Console.WriteLine("{0}", n);
```

Ejecutemos el programa.

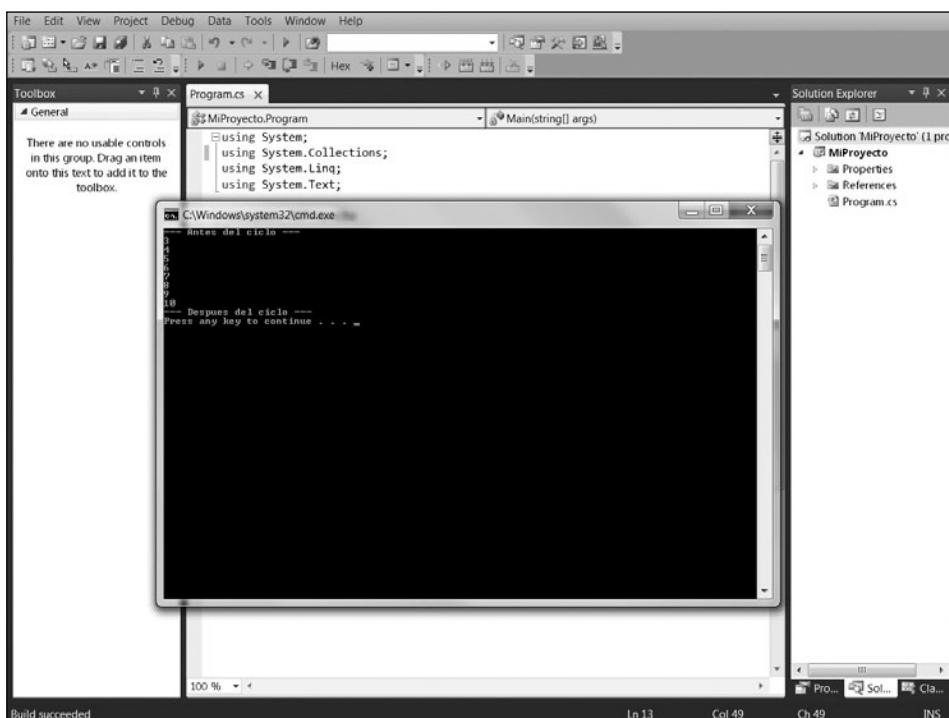


Figura 4. Éste es el resultado. Efectivamente, la variable de control empieza con 3. Desde luego, el número de repeticiones es menor.

Es posible que la variable de control tenga un valor negativo en la inicialización. Esto es útil cuando necesitamos contar desde un número negativo. Modifiquemos el código del ciclo for de la siguiente manera:

```
for (n = -10; n <= 10; n = n + 1)
    Console.WriteLine("{0}", n);
```

La ejecución nos da el resultado que observamos en la siguiente figura.

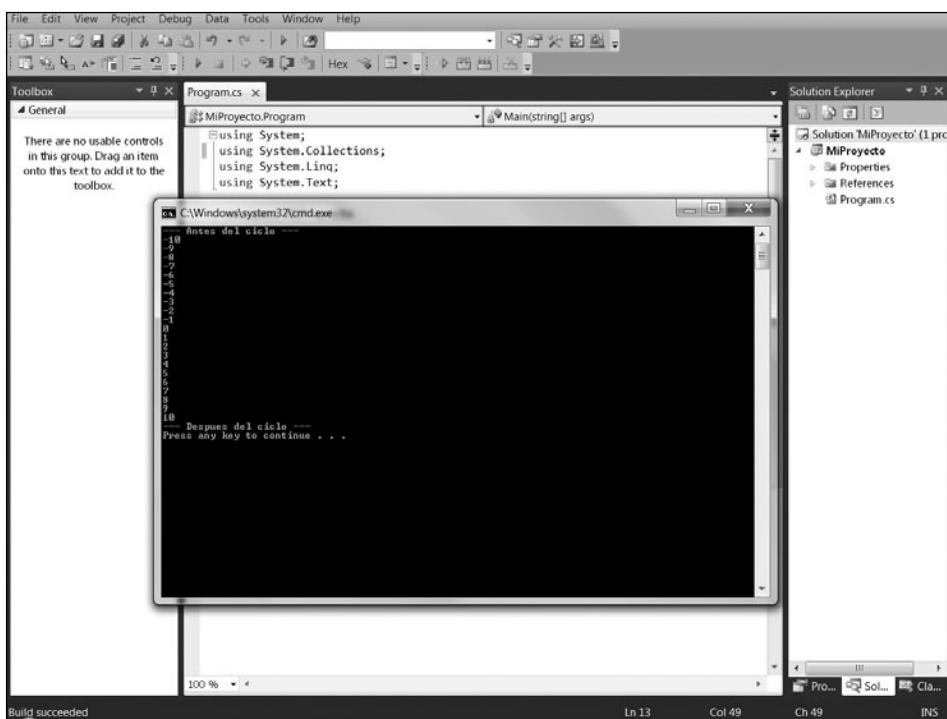


Figura 5. Podemos ver que el recorrido inicia en **-10**. No hay que pasar por alto que la variable de control también pasa por el valor de cero.

El límite de conteo del ciclo

También podemos tener un control de hasta qué número contar. Veamos nuestro código para realizar el conteo del 1 al 10.

```
for (n = 1; n <= 10; n = n + 1)
    Console.WriteLine("{0}", n);
```

En este ejemplo pudimos ver que el límite de conteo está siendo controlado en la condición y que no hay una única forma de escribirla. Podemos ver que la siguiente condición **n <= 10** también podría ser escrita como **n < 11** sin que se afecte en lo más mínimo la ejecución del ciclo.

```
for (n = 1; n < 11; n = n + 1)
    Console.WriteLine("{0}", n);
```

Veamos el resultado de este cambio en la figura a continuación.

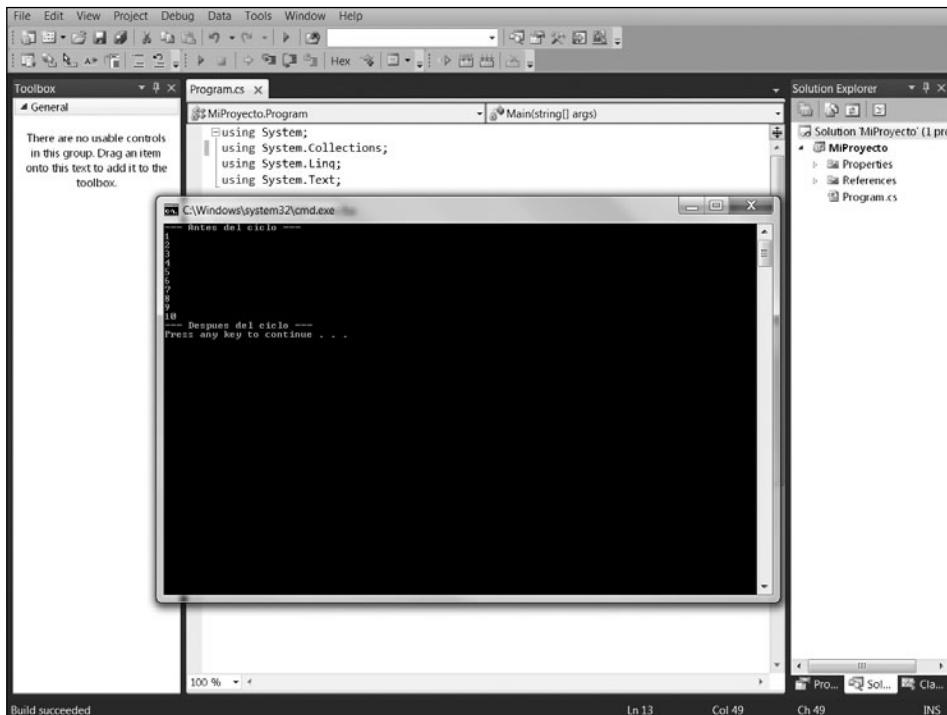


Figura 6. Podemos observar que aunque la condición es diferente, aún se cumple lo que deseamos.

Esto hay que tenerlo en cuenta para evitar confusiones en el futuro, en especial cuando veamos código de otras personas que utilicen un estilo diferente al nuestro. Ahora modifiquemos el ciclo para que cuente del **1** al **15**, en este caso usaremos el operador **<** en lugar de **<=**.

```
for (n = 1; n < 16; n = n + 1)
    Console.WriteLine("{0}", n);
```

III PROBLEMAS CON EL CICLO FOR

Uno de los problemas más comunes con el ciclo **for** es equivocar la cantidad de repeticiones necesarias. Algunas veces sucede que el ciclo da una vuelta más que las que deseamos. Si esto ocurre, lo primero que debemos revisar es la condición. Una condición mal escrita hará que no se produzcan las repeticiones deseadas. Otro problema puede ser el colocar ; al final de for.

Ejecutemos el programa y veamos su comportamiento.

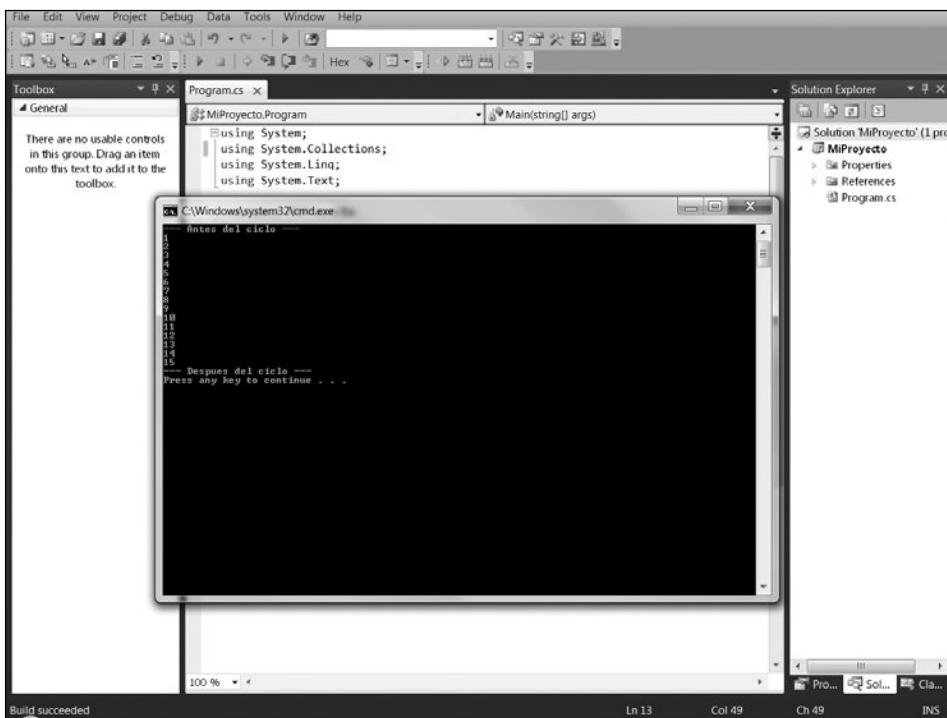


Figura 7. En este caso hemos incrementado el rango del ciclo.

Control del incremento

Ahora que ya sabemos cómo colocar el rango del ciclo **for** desde su valor de inicio hasta el valor al que contará, podemos empezar a aprender cómo hacer uso del incremento. Nuestro incremento ha sido de uno en uno. Sin embargo, podemos hacer que el conteo sea de dos en dos, de tres en tres o de cualquier otro valor.

Para lograr esto, simplemente tenemos que modificar el incremento e indicar cómo se incrementaría nuestra variable de control. El valor del incremento puede ser un valor colocado explícitamente o el valor que se encuentra adentro de una variable. Por ejemplo, hagamos que nuestro ciclo avance de dos en dos.

```
for (n = 1; n <16; n = n + 2)
    Console.WriteLine("{0}", n);
```

Como vemos, ahora hemos colocado **n = n + 2**. Con esto indicamos que se sumará **2** al valor de **n** con cada vuelta del ciclo. Esto es más fácil de entender si vemos la ejecución del programa y vemos el desplegado de los valores de **n**.

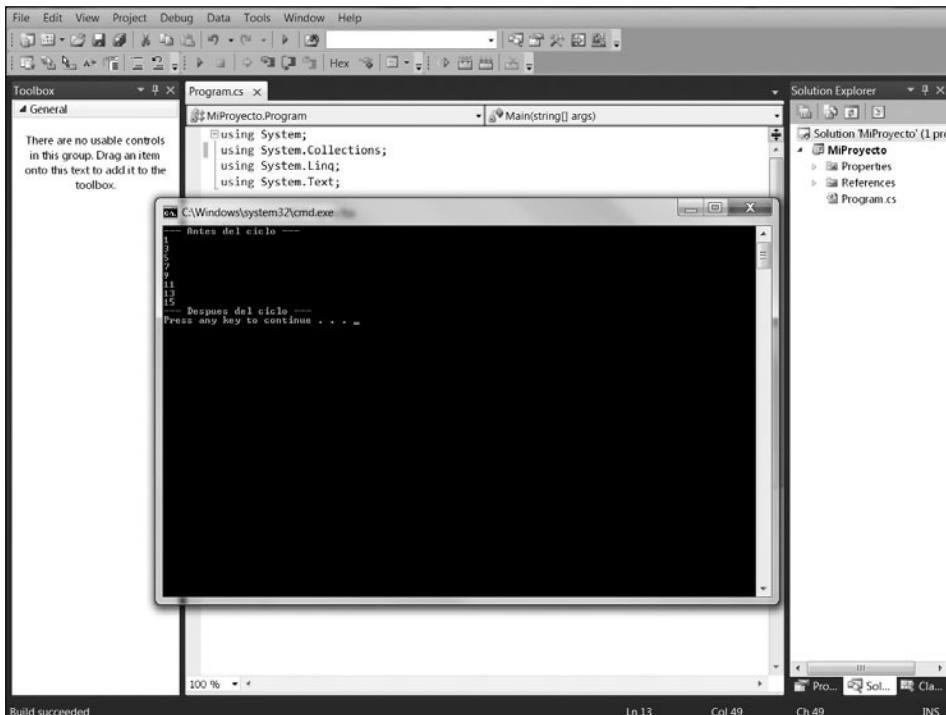


Figura 8. Podemos observar que efectivamente el valor de n se incrementa de dos en dos por cada vuelta del ciclo.

Los ciclos que hemos utilizado siempre han contado de manera **progresiva**, es decir del menor valor al valor más alto, pero también es posible hacer un ciclo **regresivo**. Por ejemplo, podríamos hacer que el ciclo cuente del 10 al 1. Para esto necesitamos modificar no solamente el incremento sino que también es necesario colocar las expresiones correctas en la inicialización y la condición.

Veamos cómo lograr esto con la siguiente sentencia:

```
for (n = 10; n>=1; n = n - 1)
```

III INICIALIZACIÓN DE VARIABLES EN CICLOS

Si llevamos a cabo la declaración e inicialización de una variable adentro del bloque de código de un ciclo, la variable será inicializada cada vez que se repita el ciclo y no conservará su valor entre vueltas del ciclo. Este detalle también nos puede llevar a errores de lógica, por lo que hay que decidir bien dónde se declara la variable.

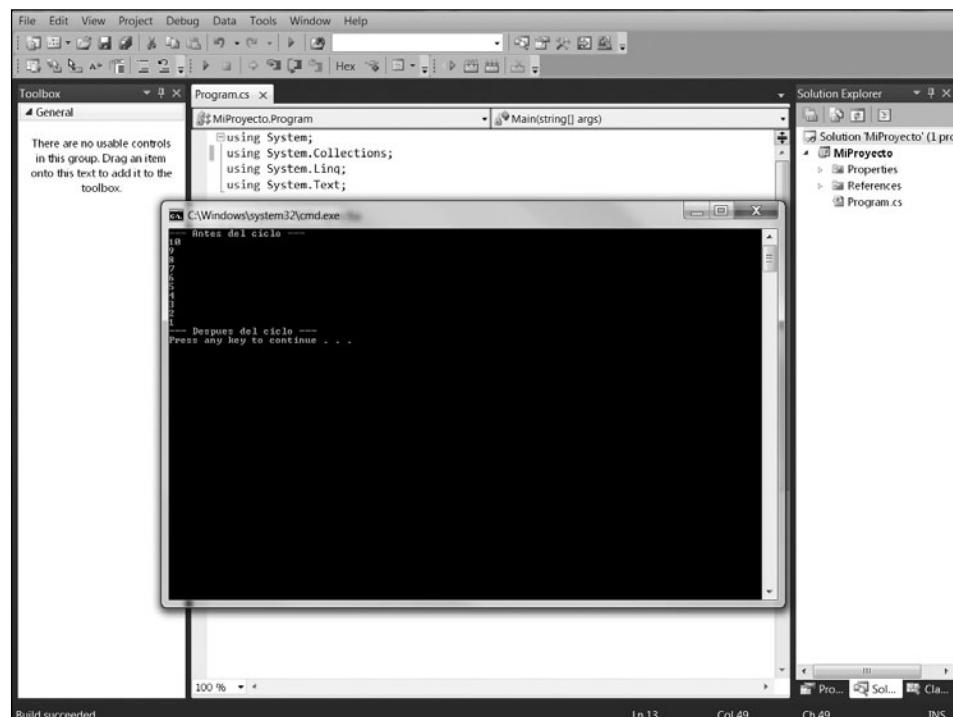


Figura 9. Aquí podemos observar cómo el valor de *n* se decremente de uno en uno.

El contador y el acumulador

Ahora aprenderemos dos conceptos nuevos. Existen dos categorías de variables dependiendo de cómo guardan la información, en primer lugar tenemos el **contador**. El contador es una variable que será incrementada o disminuirá su valor de uno en uno. En la mayoría de los ejemplos anteriores *n* ha funcionado como contador. Por su parte, el **acumulador** es una variable que puede incrementar o disminuir su valor en cualquier número.

Aún no hemos hecho uso de un acumulador, pero lo veremos cuando debamos resolver el problema de los promedios de los alumnos.

Por el momento, veamos un pequeño ejemplo de estas dos clasificaciones de variables explicadas hasta aquí:

```
using System;
using System.Collections.Generic;
using System.Text;
```

```

namespace AplicacionBase
{
    class Program
    {
        // Esta es la función principal del programa
        // Aquí inicia la aplicación
        static void Main(string[] args)
        {
            // Variables necesarias
            int n = 0; // variable de control
            int contador = 0, acumulador = 0;

            Console.WriteLine("-- Antes del ciclo --");

            for (n = 10; n >= 1; n = n - 1)
            {
                contador = contador + 1;
                acumulador = acumulador + contador;
                Console.WriteLine("{0}, {1}", contador, acumulador);
            }

            Console.WriteLine("-- Después del ciclo --");
        }
    }
}

```

Hemos creado dos variables: **contador** y **acumulador**. La variable **contador** incrementa su valor de uno en uno. Por su parte, la variable **acumulador** incrementará el valor en base al número contenido en **contador**.

A continuación, para comprender mejor lo explicado, ejecutemos el programa y veamos el funcionamiento de las dos variables:

III PROBLEMAS CON VARIABLES EN LOS CICLOS

Si llevamos a cabo la declaración de una variable adentro del bloque de código del ciclo, ésta únicamente puede ser usada en ese ciclo. Si tratamos de utilizarla después del ciclo, no será reconocida por el programa. Si éste fuera el caso, lo mejor es declarar la variable al inicio de la función para que sea conocida en ella.

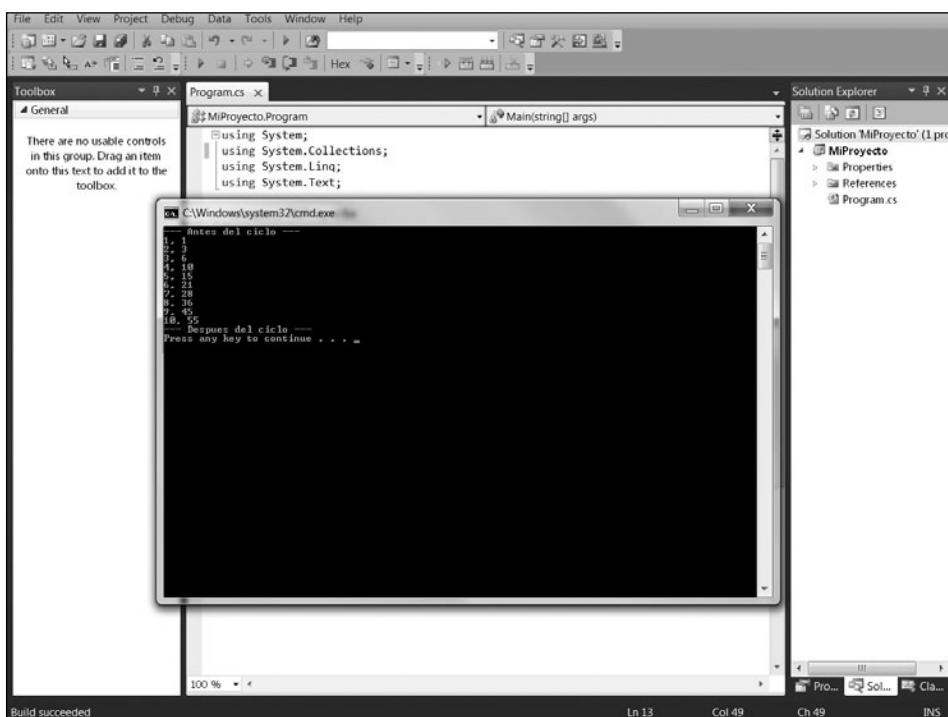


Figura 10. Vemos cómo cambian los valores de las variables con cada vuelta del ciclo.

Incrementos y decrementos

Cuando trabajamos el C# con el ciclo **for** es muy común que tengamos que incrementar o disminuir siempre de uno en uno. Para facilitarnos esto, tenemos operadores nuevos que son el **operador de incremento** y el **operador de decremento**.

OPERADOR	SIGNO
Incremento	++
Decremento	--

Tabla 1. Este tipo de operadores nos permiten cambiar el valor de la variable en 1.

Estos operadores pueden ser usados como **sufijos** o **prefijos** en la variable. En el caso de los sufijos lo escribimos como **variable++** y para el prefijo como **++variable**. El valor contenido en la variable en ambos casos se incrementará en uno. Lo que cambia es cómo se evalúa la expresión que contienen los operadores.

Cuando tenemos el caso del sufijo, se evalúa la expresión con el valor actual de la variable y luego se incrementa a la variable. En el caso del prefijo, se incrementa primero el valor de la variable y posteriormente se evalúa la expresión.

Quizás esto no quede muy claro en la primer explicación, pero no debemos preocuparnos por ello. Para poder entender esto, veremos un ejemplo a continuación:

```
using System;
using System.Collections.Generic;
using System.Text;

namespace AplicacionBase
{
    class Program
    {
        // Esta es la función principal del programa
        // Aquí inicia la aplicación
        static void Main(string[] args)
        {
            int numero = 5;

            Console.WriteLine("Valor inicial {0}", numero);

            // incrementamos
            numero++;

            Console.WriteLine("Después del incremento {0}", numero);

            // Decrementamos
            numero--;

            Console.WriteLine("Después del decremento {0}", numero);

            // Incremento en la sentencia
            Console.WriteLine("Incremento en la sentencia {0}", numero++);

            Console.WriteLine("Valor después de la sentencia {0}",
                numero);

            // Incremento en la sentencia como prefijo
            Console.WriteLine("Incremento en la sentencia {0}", ++numero);

            Console.WriteLine("Valor después de la sentencia {0}",
                numero);
        }
    }
}
```

Podemos ejecutar el código escrito y ver qué es lo que ha ocurrido, y la razón por la que obtenemos esos valores desplegados.

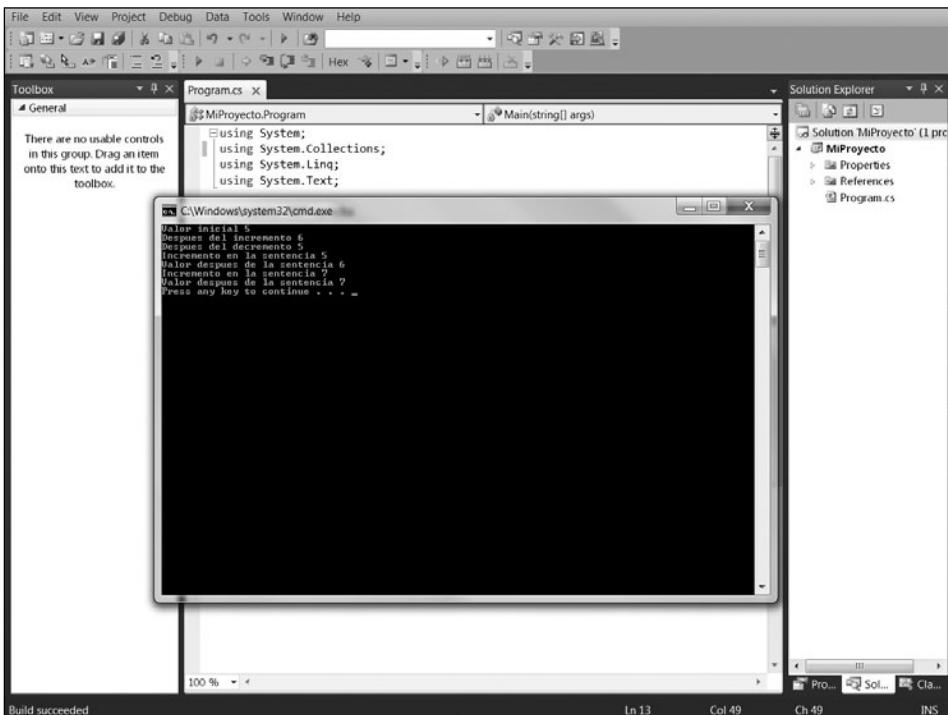


Figura 11. Aquí vemos los valores obtenidos por los operadores y la diferencia en usarlos como sufijos o prefijos.

Empezamos con una variable llamada **numero** a la que le asignamos el valor inicial de **cinco**. Desplegamos su valor en la pantalla. Luego usamos el operador e incrementamos el valor de la variable. En este caso vale **6**, ya que el incremento es en uno. Esto lo comprobamos al imprimir el valor de la variable. Después llevamos a cabo el decremento. En este caso, **numero** guarda el valor de **5**, ya que lo hemos decrementado en uno. La impresión del mensaje nos muestra que estamos en lo correcto. Ahora que ya hemos visto cómo funciona, podemos tratar de colocar el operador adentro de una sentencia. La sentencia será el mismo mensaje que deseamos mostrar con el fin de observar cómo se evalúa y las diferencias entre el uso como sufijo y prefijo. No olvidemos que el último valor en **numero** es **5**. Empezamos por colocar el incremento adentro de la sentencia y lo usamos como sufijo. Esto incrementa el valor a **6**, pero si vemos la ejecución del programa, el valor que aparece es **5**. ¿Por qué? La razón ya la conocemos: cuando se usa como sufijo, se evalúa la expresión con el valor actual y luego se incrementa. La siguiente impresión del valor de la variable nos muestra cómo efectivamente la variable sí fue incrementada después de la sentencia anterior. Ahora podemos experimentar con el sufijo. Nuestro valor actual es de **6** e ingresamos la

sentencia. En este caso, primero se incrementa el valor y tenemos **7** en la variable **número** y luego la imprimimos. Por eso, en este caso obtenemos **7** en la pantalla. Con esto ya hemos visto el comportamiento de estos operadores y podemos integrarlos en nuestro ciclo for. En el caso de incremento:

```
for (n = 1; n <16; n++)
    Console.WriteLine("{0}", n);
```

O para decrementar decrementar su valor:

```
for (n = 10; n>=1; n--)
    Console.WriteLine("{0}", n);
```

Existen otros operadores que también podemos utilizar cuando deseamos calcular un valor con la variable y guardar el mismo valor en la variable.

OPERADOR	EJEMPLO	EQUIVALE A
+=	numero+=5	numero=numero+5
-=	numero-=5	numero=numero-5
=	numero=5	numero=numero*5
/=	numero/=5	numero=numero/5

Tabla 2. Estos operadores también nos permiten modificar el valor de la variable.

Es bueno conocer estos operadores ya que los veremos y deberemos utilizar frecuentemente en muchos programas de cómputo. Por ejemplo, para hacer el incremento de cinco en cinco en un ciclo for, realizamos lo siguiente:

```
for (n = 1; n <16; n+=5)
    Console.WriteLine("{0}", n);
```

Ejemplos con el ciclo for

Ahora ya podemos resolver el problema sobre el cálculo del promedio. Sabemos que es lo que se debe repetir y la cantidad de veces que debemos hacerlo. En este caso, lo que se debe repetir es pedir la calificación del alumno y hacer una suma de estas calificaciones. La cantidad de veces que se tiene que repetir depende de la cantidad de alumnos. Esta suma de las calificaciones puede hacerse con la variable acumulador. El promedio final se calcula dividiendo el acumulador entre la cantidad de

alumnos y éste se presenta en la pantalla. Veamos el diagrama de flujo de nuestro algoritmo para resolver este problema.

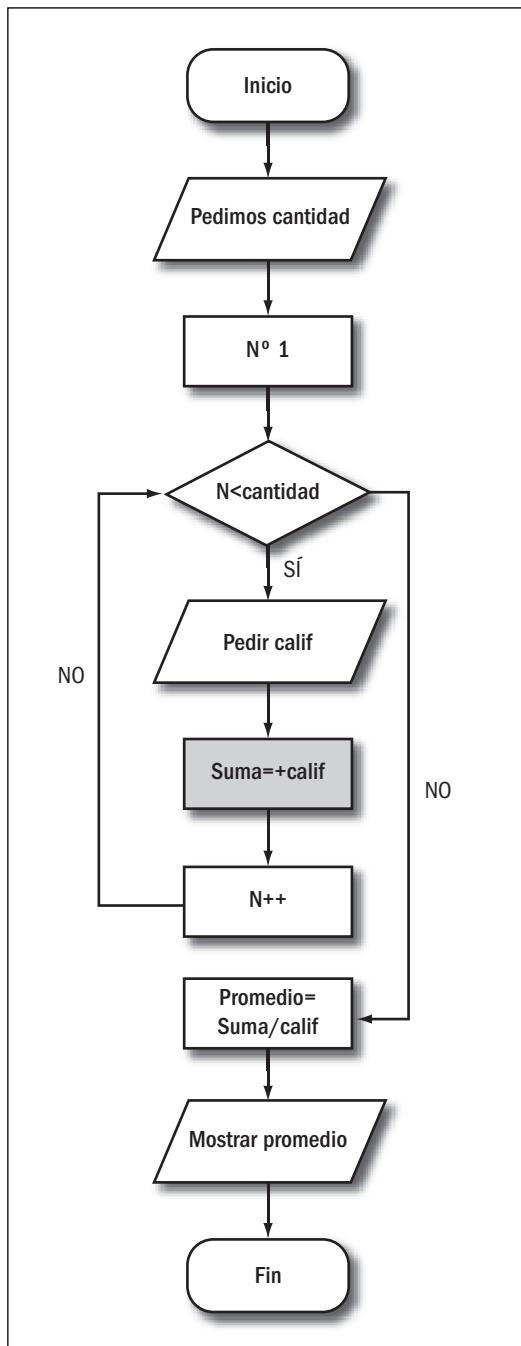


Figura 12. Podemos observar fácilmente el ciclo for y lo que se realiza adentro de él.

Ahora podemos ver cómo queda el código del programa:

```
using System;
using System.Collections.Generic;
using System.Text;

namespace AplicacionBase
{
    class Program
    {
        // Esta es la función principal del programa
        // Aquí inicia la aplicación
        static void Main(string[] args)
        {
            // Variables necesarias
            int n = 0;    // Control ciclo
            int cantidad; // Cantidad alumnos
            float calif = 0.0f; // Calificación del alumno
            float suma = 0.0f; // Sumatoria de calificaciones
            float promedio = 0.0f; // Promedio final
            string valor = "";

            Console.WriteLine("Dame la cantidad de alumnos:");
            valor = Console.ReadLine();
            cantidad = Convert.ToInt32(valor);

            // Ciclo para la captura de calificaciones
            for (n = 1; n <= cantidad; n++)
            {
                Console.WriteLine("Dame la calificación del alumno");
                valor = Console.ReadLine();
                calif = Convert.ToSingle(valor);

                // Llevamos a cabo la suma de calificaciones
                suma += calif;
            }

            // Calculamos el promedio
            promedio = suma / cantidad;
```

```

        // Mostramos el promedio
        Console.WriteLine("El promedio es {0}",promedio);

    }
}
}

```

Si ejecutamos el programa obtenemos la salida que se muestra en la siguiente figura.

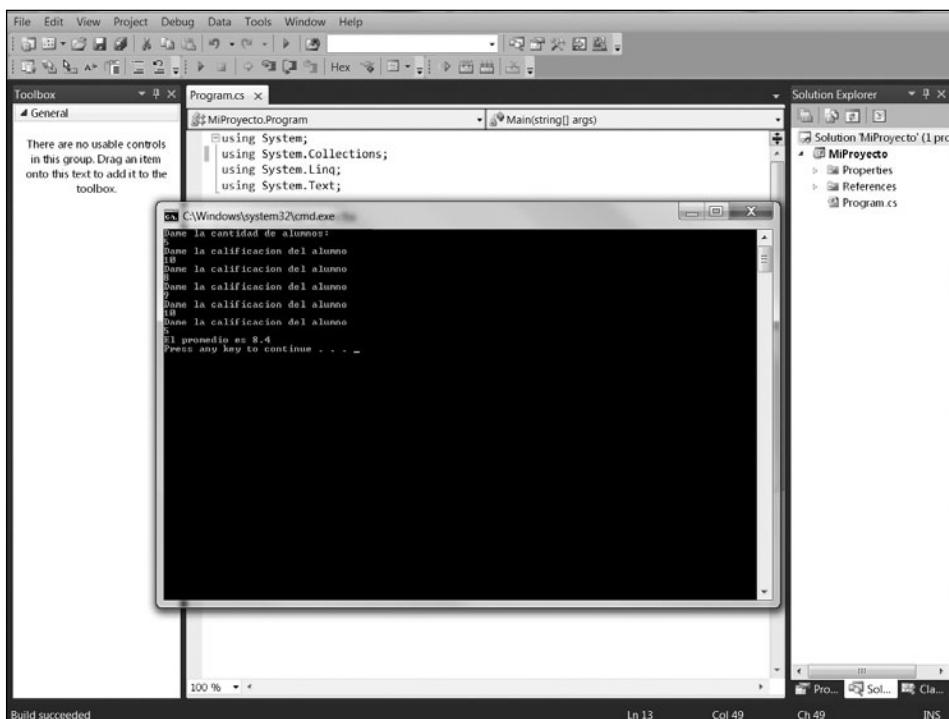


Figura 13. Vemos cómo la petición
de la calificación se repite el número de veces indicado.

Un punto importante sobre este programa, en comparación al primer intento que hicimos al inicio del capítulo, es que puede funcionar con cualquier cantidad de alumnos. No importa si son 5, 10 ó 5000, el programa llevará a cabo su cometido. Vemos cómo el uso del ciclo no solamente nos da flexibilidad, sino que también evita que tengamos que repetir mucho código.

Veamos otro ejemplo. Podemos utilizar el ciclo **for** en cualquier problema que necesite algo que se repita: una operación, un conteo, algún proceso. Pero también el número de repeticiones debe ser conocido, ya sea por medio de un valor colocado explícitamente o un valor colocado adentro de una variable.

Ahora tenemos que calcular el **factorial** de un número. Por ejemplo, el factorial de 5 es $5*4*3*2*1$ que da **120**. De igual forma, debemos calcular el valor factorial de cualquier número dado por el usuario. Si observamos el problema y cómo se resuelve, podemos apreciar que hay un ciclo y este ciclo es con conteo regresivo. Podemos resolver fácilmente con un ciclo **for** que lleve a cabo la multiplicación con la variable de control como producto.

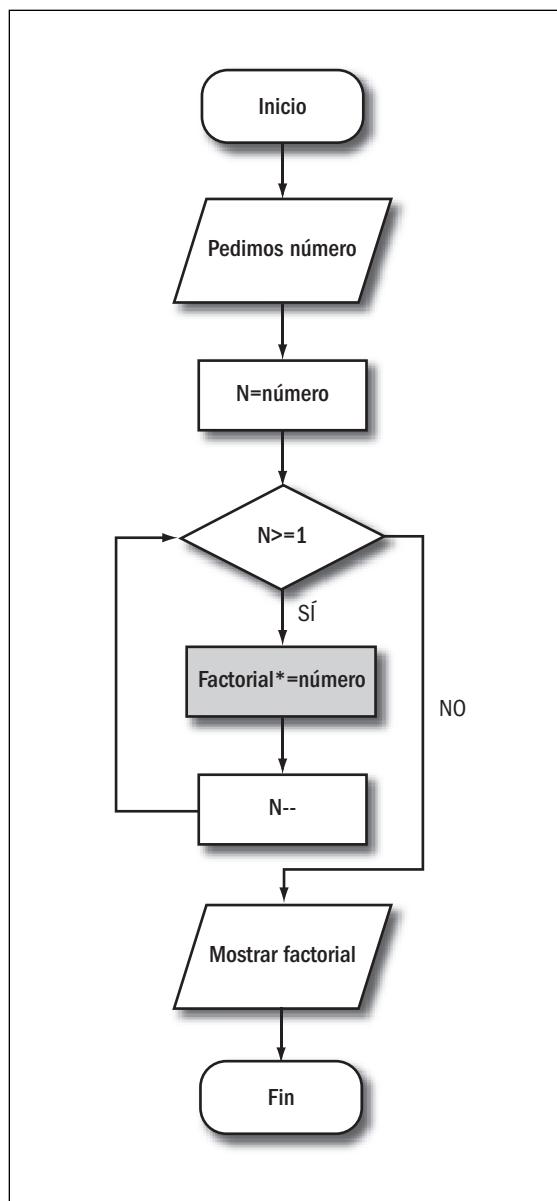


Figura 14. Éste es el diagrama de flujo del algoritmo para calcular el factorial de un número.

El código del programa queda de la siguiente forma:

```
using System;
using System.Collections.Generic;
using System.Text;

namespace AplicacionBase
{
    class Program
    {
        // Esta es la función principal del programa
        // Aquí inicia la aplicación
        static void Main(string[] args)
        {
            // Variables necesarias
            int n = 0; // Variable de control
            int numero = 0; // Número al que sacamos factorial
            int factorial = 1; // Factorial calculado
            string valor = "";

            // Pedimos el numero
            Console.WriteLine("Dame el número al que se le saca el
                factorial:");
            valor = Console.ReadLine();

            numero = Convert.ToInt32(valor);

            // Calculamos el factorial en el ciclo
            for (n = numero; n >= 1; n--)
                factorial *= n;

            // Mostramos el resultado
            Console.WriteLine("El factorial de {0} es {1}", numero,
                factorial);
        }
    }
}
```

Cuando ejecutamos el programa, nos encontramos con que podemos llevar a cabo el cálculo factorial sin problema alguno.

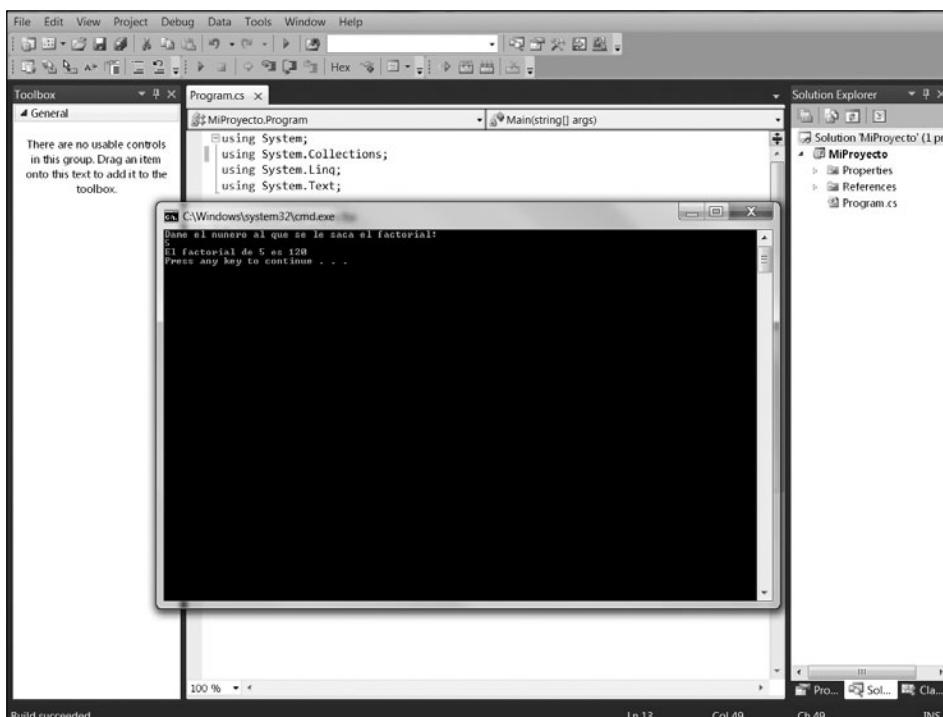


Figura 15. El factorial del número es calculado adecuadamente.

El ciclo do while

Hemos visto que el ciclo **for** es muy útil para repetir algo cuando sabemos el número de repeticiones previamente. Sin embargo, algunas veces no es posible saber el número de repeticiones que tendremos. Pensemos que una carpintería nos ha contratado para hacer un programa que transforme de pies y pulgadas a centímetros, ya que ellos lo usan para calcular el tamaño de las tablas. El programa es muy sencillo, pero no podemos dejarlo así ya que sería muy incómodo para ellos ejecutar el programa cada vez que necesitan hacer un corte.

Pensamos que esto se puede resolver con un ciclo **for**. ¿De cuántas vueltas el ciclo? Si colocamos **10** vueltas podemos pensar que es suficiente. Pero hay días que necesitan **15** conversiones y es necesario ejecutar el programa **dos** veces. Y los días que sólo necesitan **5** conversiones tienen que escribir **5** más tan sólo para finalizar el programa. Podemos deducir del problema que efectivamente necesitamos un ciclo. Sin embargo, no sabemos el número de repeticiones previamente, por lo que el ciclo **for** no es adecuado. Necesitamos un ciclo que pueda ser controlado por una condición, y la evaluación de esta condición dependerá del estado del programa en un momento dado. El ciclo **do while** nos permite hacer esto. Permite que cierto código se repita mientras una condición se evalúe como verdadera. El valor de la evaluación dependerá del estatus del programa en un momento dado.

El ciclo **do while** se codifica de la siguiente manera:

```
do {
    Código
}(condición);
```

Si observamos el diagrama de flujo de la figura 16, podemos ver el funcionamiento interno del ciclo **do while** de una manera simple y clara.

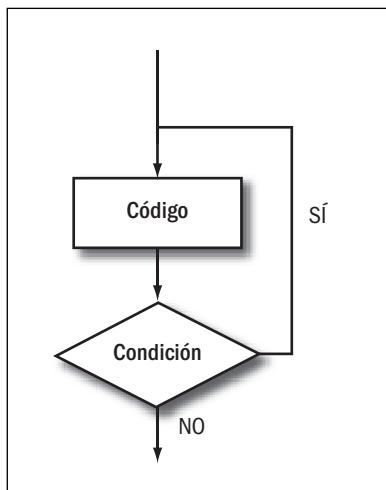


Figura 16. Éste es el diagrama del ciclo *do while*.

Podemos observar su comportamiento interno.

Empecemos a recorrer el ciclo. En primer lugar encontramos el código que hay que llevar a cabo. Este código es definido adentro de un bloque de código aunque sea solamente una sentencia. Generalmente, en el código se modificará de alguna forma el valor de la variable o las variables que usamos para la expresión en la condición. Esta modificación de valor puede llevarse a cabo por medio de un cálculo, proceso o incluso una petición al usuario. Después tenemos la condición a evaluar. En la condición colocamos una expresión lógica o relacional. Si al evaluarse la condición obtenemos el valor **true**, entonces se repite el código. En caso de que la condición del valor de **false** ya no se repite y se continúa con el programa. Hay que tener en cuenta que a veces sucede que la condición se evalúa como falsa desde la primera vez. En este caso, solamente se habrá ejecutado el código una vez.

Veamos un ejemplo. Crearemos el programa para la carpintería y usaremos el ciclo **do while** para que el programa se repita el número de veces necesarias, aun sin saber cuántas veces son. Para lograr esto tenemos que pensar bien en nuestra condición y cuál es el estado del programa que evaluaremos en ella.

La forma más sencilla de hacerlo es preguntarle al usuario si desea hacer otra conversión. Si lo afirma, se repite el código para convertir. En caso de que el usuario no lo desee, el programa finaliza. De esta forma, es controlada la repetición del ciclo pudiéndola repetir, aun sin saber cuántas veces hay que hacerlo.

Nuestro programa tendrá el siguiente diagrama de flujo.

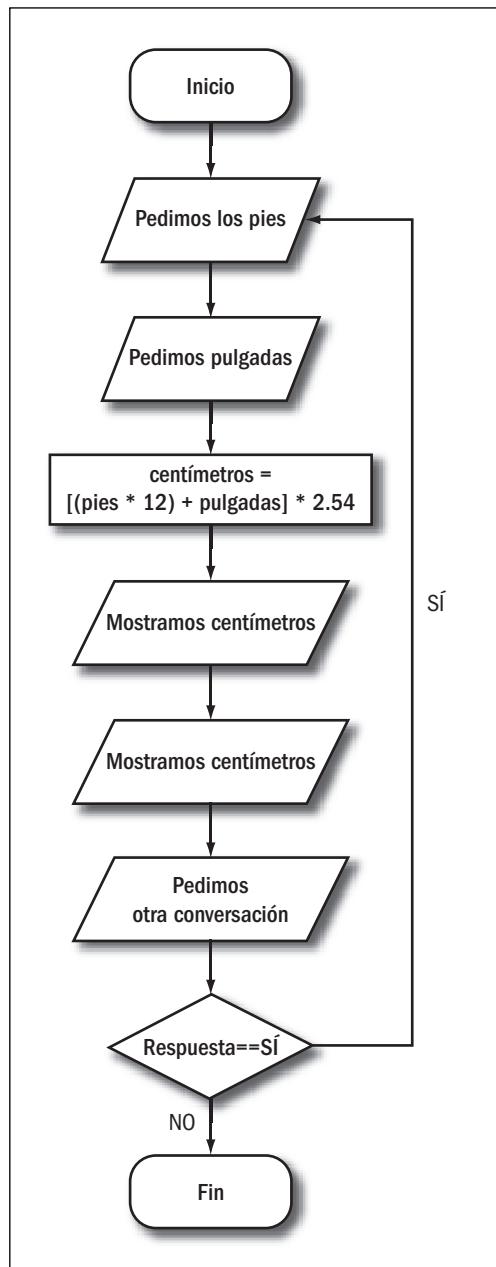


Figura 17. Aquí podemos observar el programar y distinguir el ciclo `do while`.

El programa quedará de la siguiente manera:

```
using System;
using System.Collections.Generic;
using System.Text;

namespace AplicacionBase
{
    class Program
    {
        // Esta es la función principal del programa
        // Aquí inicia la aplicación
        static void Main(string[] args)
        {
            // Variables necesarias
            float pies = 0.0f; // Cantidad de pies
            float pulgadas = 0.0f; // Cantidad de pulgadas
            float centimetros = 0.0f; // Resultado en centímetros
            string respuesta = ""; // Respuesta para otro cálculo
            string valor = "";

            do
            {
                // Pedimos los pies
                Console.WriteLine("Cuántos pies:");
                valor = Console.ReadLine();
                pies = Convert.ToSingle(valor);

                // Pedimos las pulgadas
                Console.WriteLine("Cuántas pulgadas:");
                valor = Console.ReadLine();
                pulgadas = Convert.ToSingle(valor);

                // Convertimos a centímetros
                centimetros = ((pies * 12) + pulgadas) * 2.54f;

                // Mostramos el resultado
                Console.WriteLine("Son {0} centímetros", centimetros);

                // Preguntamos si otra conversión
            }
        }
    }
}
```

```

Console.WriteLine("Deseas hacer otra conversión
    (si/no)?");
respuesta = Console.ReadLine();

} while (respuesta == "si");

}
}
}

```

Ejecutemos el programa.

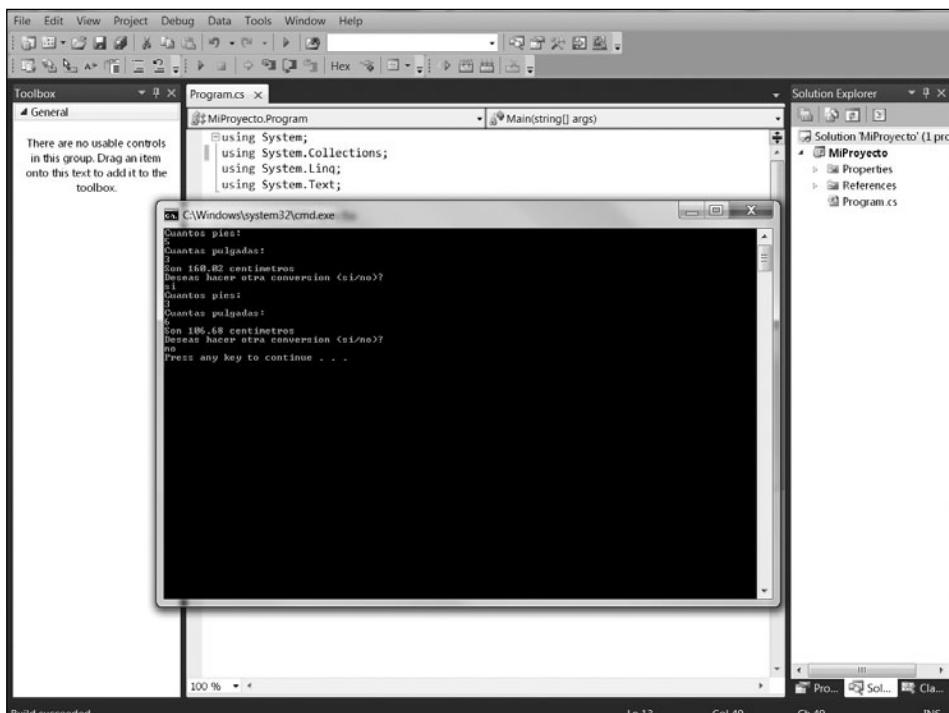


Figura 18. En la ejecución podemos verificar que tenemos un ciclo, pero puede terminar cuando lo deseemos.

Podemos observar que hemos colocado dentro del ciclo la parte de conversión de pies y pulgadas a centímetros. Después de mostrar el resultado le preguntamos al usuario si desea realizar otra conversión. En caso afirmativo, repetimos el ciclo. Si el usuario no desea hacer otra conversión, el ciclo finaliza. Esto nos permite que a veces se repita cinco veces, cuando sea necesario diez, y así dependiendo de las necesidades del usuario. Hemos resuelto el problema.

El ciclo **do while** nos da flexibilidad extra, pero siempre es necesario usar el tipo de ciclo adecuado al problema que tenemos.

Veamos otro ejemplo donde podemos utilizar este ciclo. En uno de los programas anteriores le preguntábamos al usuario qué operación deseaba realizar y luego los operandos. El programa tal y como está solamente se ejecuta una vez, pero si usamos el ciclo **do while** y colocamos una nueva opción en el menú, entonces el usuario puede realizar las operaciones que sean necesarias o solamente una como se usaba anteriormente. Este tipo de comportamiento es muy útil y fácil de implementar con el ciclo **do while**.

Si modificamos el programa quedará de la siguiente manera:

```
using System;
using System.Collections.Generic;
using System.Text;

namespace AplicacionBase
{
    class Program
    {
        // Esta es la función principal del programa
        // Aquí inicia la aplicación
        static void Main(string[] args)
        {
            // Variables necesarias
            float a = 0.0f;
            float b = 0.0f;
            float resultado = 0.0f;
            string valor = "";
            int opcion = 0;

            // Tenemos el ciclo
            do
            {
                // Mostramos el menú
                Console.WriteLine("1- Suma");
                Console.WriteLine("2- Resta");
                Console.WriteLine("3- División");
                Console.WriteLine("4- Multiplicación");
                Console.WriteLine("5- Salir");
            }
        }
}
```

```
Console.WriteLine("Que operación deseas hacer: ");
valor = Console.ReadLine();
opcion = Convert.ToInt32(valor);

if (opcion != 5)
{
    // Pedimos el primer número
    Console.WriteLine("Dame el primer número:");
    valor = Console.ReadLine();
    a = Convert.ToSingle(valor);

    // Pedimos el segundo número
    Console.WriteLine("Dame el segundo número:");
    valor = Console.ReadLine();
    b = Convert.ToSingle(valor);

    switch (opcion)
    {
        // Verificamos para suma
        case 1:
            resultado = a + b;
            break;

        // Verificamos para resta
        case 2:
            resultado = a - b;
            break;

        // Verificamos para división
        case 3:
            if (b != 0) // este if esta anidado
                resultado = a / b;
            else // Este else pertenece al segundo if
                Console.WriteLine("Divisor no válido");
            break;

        // Verificamos para la multiplicación
        case 4:
            resultado = a * b;
            break;
    }
}
```

```

        // Si no se cumple ninguno de los casos
        // anteriores
    default:
        Console.WriteLine("Opción no válida");
        break;

    }

    // Mostramos el resultado
    Console.WriteLine("El resultado es: {0}",
                      resultado);
}
} while (opcion != 5);

}

}
}

```

El ciclo while

Habiendo comprendido el ciclo **Do While**, estamos en condiciones de ver otro tipo de ciclo. Este ciclo se conoce como **while** y en cierta forma se asemeja al anterior, pero tiene sus propias características que debemos conocer para utilizarlo correctamente. El ciclo **while** también puede ser utilizado cuando tenemos algo que se debe repetir pero no conocemos el número de repeticiones previamente. La repetición del ciclo tiene que ver con el cumplimiento de una condición. A diferencia del ciclo **do while**, este ciclo puede no ejecutarse ni siquiera una vez. Su estructura es la siguiente:

```

while(condicion) {
    Código
}

```

III VERIFICAR CAMBIOS EN UN PROGRAMA

En el programa de las operaciones matemáticas hemos agregado un ciclo **do while**. Sin embargo, nuestro trabajo no termina ahí. Tenemos que ejecutarlo para ver si se comporta de la forma deseada. Por eso fue necesario que adicionáramos un **if**, ya que de lo contrario nos pediría los operadores aun cuando lo que deseáramos fuera salir.

Para comprender cómo funciona, veamos el diagrama de flujo.

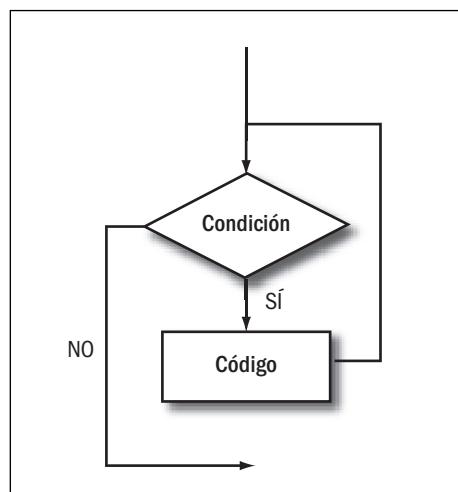


Figura 19. Este ciclo inicia con una condición y esto lo debemos de tener en cuenta en nuestros algoritmos.

Si observamos el diagrama de flujo lo primero que encontramos es una condición. Adentro de esta condición colocaremos una expresión lógica o relacional. Si la condición se evalúa como verdadera, entonces se procede a ejecutar el código. Después del código se regresa a la condición. Si la condición no se cumple, entonces no se ejecuta el código y se continúa con el resto del programa.

El tener la condición al inicio del ciclo nos lleva a un punto importante. Si la condición no se cumple desde el inicio, entonces el ciclo nunca se lleva a cabo. Si nuestro algoritmo está bien diseñado, esto es deseable.

Ahora podemos hacer un ejemplo de programa que utilice el ciclo **while**. En este ejemplo utilizaremos la característica del ciclo que puede repetirse o no repetirse ni siquiera una vez. Imaginemos que tenemos que hacer un programa de control para enfriar una caldera. La caldera debe ser enfriada a 20 grados centígrados.

El ciclo **while** será usado para reducir la temperatura de uno en uno para cada vuelta del ciclo hasta que lleguemos a 20 grados centígrados. La ventaja que nos da este ciclo

III GARANTIZAR LA EJECUCIÓN DEL CICLO WHILE

Algunos programadores usan el ciclo **while** y para garantizar su ejecución les asignan valores a las variables de forma tal que la condición se cumpla al inicio. Esto es recomendable en pocos casos. Si necesitamos un ciclo que garantice la entrada es mejor hacer uso de **do while** en lugar de forzar nuestra lógica con **while**.

es que si la temperatura es menor a 20 grados, ni siquiera se entra al ciclo y no se lleva a cabo ningún enfriamiento. En este caso, aprovechamos las propiedades del ciclo. Veamos el código de este programa:

```
using System;
using System.Collections.Generic;
using System.Text;

namespace AplicacionBase
{
    class Program
    {
        // Esta es la función principal del programa
        // Aquí inicia la aplicación
        static void Main(string[] args)
        {
            // Variables necesarias
            int temperatura = 0;
            string valor = "";

            // Pedimos la temperatura
            Console.WriteLine("Dame la temperatura actual:");
            valor = Console.ReadLine();

            temperatura = Convert.ToInt32(valor);

            // El ciclo reduce la temperatura
            while (temperatura > 20)
            {
                // Disminuimos la temperatura
                temperatura--;
            }
        }
    }
}
```

III CICLOS ANIDADOS

Al igual que con los **if**, es posible tener ciclos anidados. Como adentro de los ciclos podemos colocar cualquier código válido, entonces también podemos colocar otro ciclo. Cuando un ciclo se encuentra adentro de otro, decimos que están anidados. Hay que tener mucho cuidado con las variables de control y las condiciones para evitar problemas de lógica.

```

        Console.WriteLine("Temperatura->{0}",
                           temperatura);
    }

    // Mostramos la temperatura final

    Console.WriteLine("La temperatura final es {0}",
                           temperatura);
}
}
}

```

Para comprobar que efectivamente el ciclo actúa como hemos dicho, debemos correr el programa dos veces. La primera vez colocaremos la temperatura en valor mayor a 20°C. Esto debe hacer que el ciclo se lleve a cabo. La segunda vez colocaremos un valor menor que 20. En este caso, el programa no deberá entrar al ciclo.

Veamos el resultado de la primera ejecución:

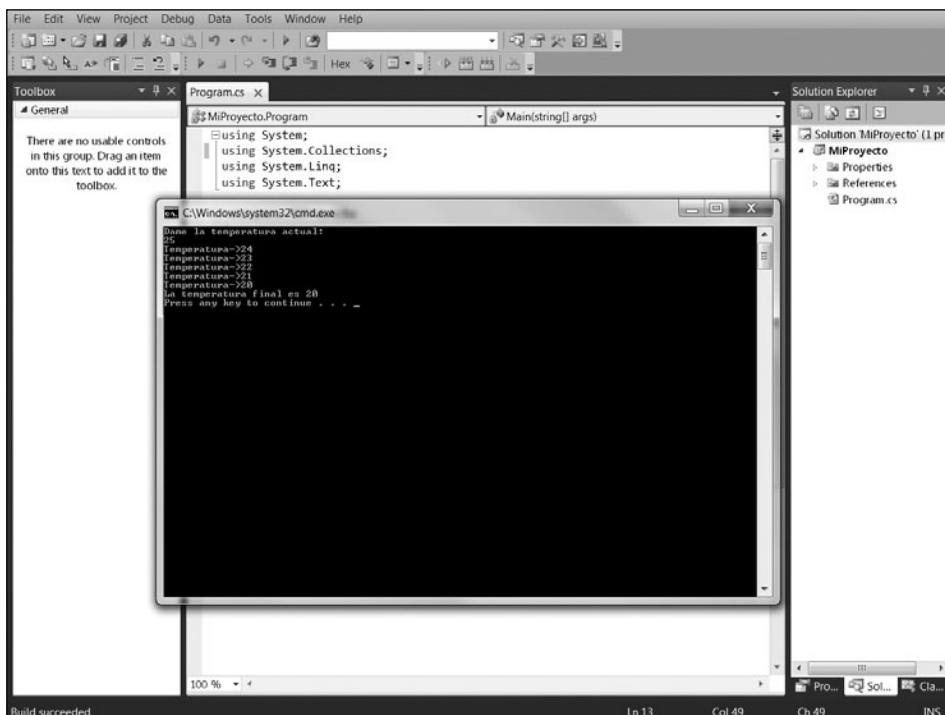


Figura 20. Podemos observar que se ha entrado al ciclo y que se lleva a cabo mientras la temperatura es mayor a 20°.

Si llevamos a cabo la segunda ejecución obtenemos lo siguiente:

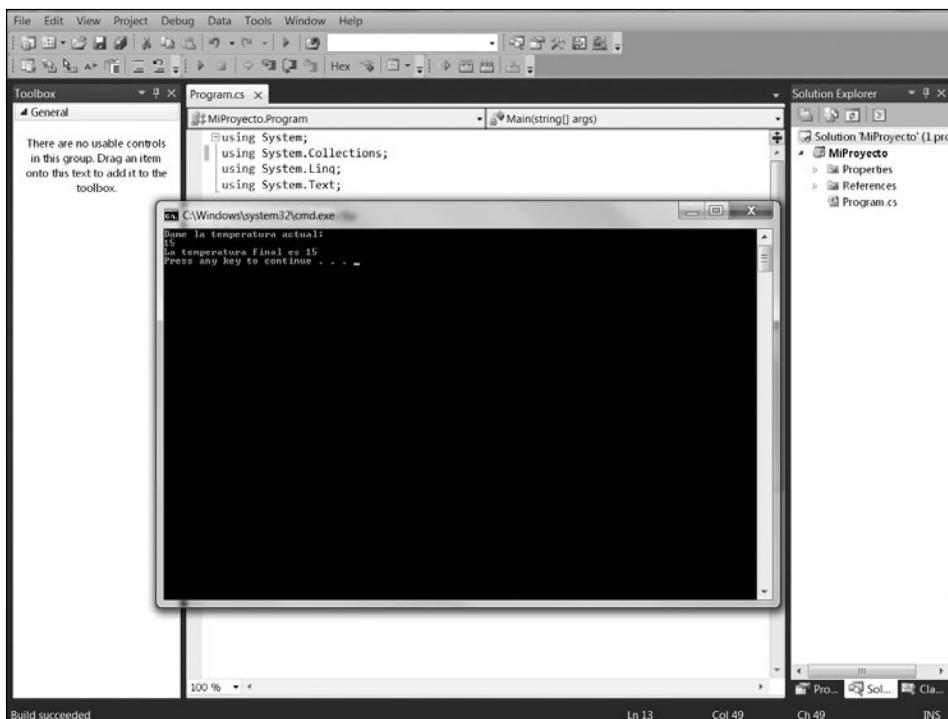


Figura 21. En este caso no se entra al ciclo en la ejecución del programa.

Con esto hemos visto los tres ciclos principales disponibles en C# y de qué forma podemos utilizarlos dentro de nuestros desarrollos.



RESUMEN

Los ciclos nos permiten repetir la ejecución de cierto código. El ciclo for es usado para repetir algo un número determinado de veces. Este ciclo necesita de una variable de control con valor inicial, una condición y un incremento. El ciclo do while nos permite repetir cierto código un número de veces desconocido y se ejecuta al menos una vez. El número de veces depende de una condición. El ciclo while también nos permite repetir el código un número desconocido de veces, pero en este caso el ciclo puede o no puede ejecutarse dependiendo de su condición.



ACTIVIDADES

TEST DE AUTOEVALUACIÓN

1 ¿Qué es un ciclo?

2 ¿Cuáles son las partes del ciclo **for**?

3 ¿Cómo colocamos el valor inicial de conteo en un ciclo **for**?

4 ¿Cómo colocamos el valor final de conteo en un ciclo **for**?

5 ¿Cómo se lleva a cabo el incremento en un ciclo **for**?

6 ¿Cómo funciona el ciclo **do while**?

7 ¿Por qué el ciclo **while** se lleva a cabo al menos una vez?

8 ¿Se necesita punto y coma al finalizar el ciclo **do while**?

9 ¿Cómo funciona el ciclo **while**?

10 ¿Cuántas veces se puede repetir el ciclo **while**?

11 ¿Qué tipo de condición podemos colocar en el ciclo **while**?

12 ¿Se coloca un bloque de código en el ciclo **while**?

EJERCICIOS PRÁCTICOS

1 Hacer un programa que muestre la tabla de multiplicar del 1 al 10 de cualquier número.

2 Hacer un programa que calcule el resultado de un número elevado a cualquier potencia.

3 Hacer el diagrama de flujo del programa para reducir la temperatura de la caldera.

4 Hacer un programa que encuentre los números primos que existen entre el 1 y el 1000.

5 Hacer un programa que calcule el promedio de edad de un grupo de personas y diga cuál es la de edad más grande y cuál es la más joven.

Funciones y métodos

Las funciones y los métodos son elementos muy importantes de programación. En C# estaremos utilizándolos frecuentemente, en especial cuando avancemos hacia la programación orientada a objetos. Las funciones nos dan muchas ventajas y nos permiten desarrollar de forma más rápida y más ordenada nuestras aplicaciones.

Las funciones	148
Funciones que ejecutan código	150
Funciones que regresan un valor	156
Funciones que reciben valores	159
Funciones que reciben parámetros y regresan un valor	163
Optimizar con funciones	171
Paso por copia y paso por referencia	178
Uso de parámetros de default	183
Resumen	185
Actividades	186

LAS FUNCIONES

Empecemos a conocer las funciones. Veamos un ejemplo de un problema y luego la forma cómo la utilización de la función nos puede ayudar. La función es un elemento del programa que contiene código y se puede ejecutar, es decir, lleva a cabo una operación. La función puede **llamarse** o **invocarse** cuando sea necesario y entonces el código que se encuentra en su interior se va a ejecutar. Una vez que la función termina de ejecutarse el programa continúa en la sentencia siguiente de donde fue llamada.

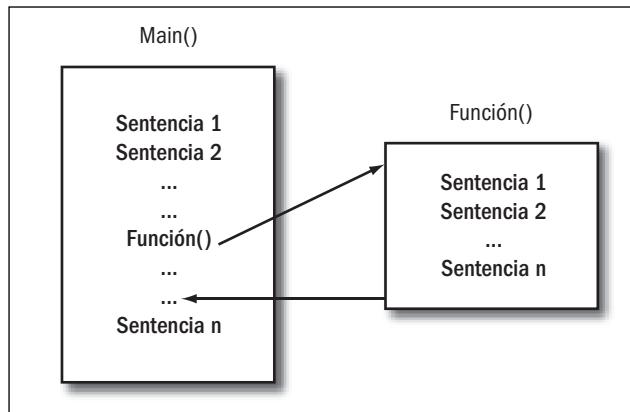


Figura 1. Podemos observar cómo la ejecución del programa se dirige a la función y luego regresa a la sentencia siguiente de donde fue invocada.

Las funciones, para que sean útiles, deben estar especializadas. Es decir que cada función debe hacer solamente una cosa y hacerla bien. Nosotros ya hemos utilizado una función, es la función **Main()** y podemos crear más funciones conforme las necesitemos. Dentro de la programación orientada a objetos las clases tienen código y este código se encuentra adentro de funciones llamadas métodos. Por ejemplo, cuando hemos convertido de cadena a un entero, hacemos uso de una función llamada **ToIn32()** que se encuentra adentro de la clase **Convert**.

Las funciones constan de cinco partes:

```

modificador tipo Nombre(parámetros)
{
  código
}
  
```

Veremos ahora las cinco partes, pero conforme estudiemos a las funciones sabremos cómo son utilizadas. Las funciones pueden regresar información y esta información

puede ser cadena, entero, flotante o cualquier otro tipo. En la sección de **tipo** tenemos que indicar precisamente la clase de información que regresa. Si la función no regresa ningún valor entonces tenemos que indicar a su tipo como **void**.

Todas las funciones deben identificarse y lo hacemos por medio de su **nombre**. Las funciones que coloquemos adentro de las clases deben de tener un nombre único. El nombre también es utilizado para invocar o ejecutar a la función.

Las funciones pueden necesitar de **datos** o **información** para poder trabajar. Nosotros le damos esta información por medio de sus **parámetros**. Los parámetros no son otra cosa que una lista de variables que reciben estos datos. Si la función no necesita usar a los parámetros, entonces simplemente podemos dejar los paréntesis vacíos. Nunca debemos olvidar colocar los paréntesis aunque no haya parámetros.

El código de la función se sitúa adentro de un **bloque de código**. En esta sección podemos colocar cualquier código válido de C#, es decir, declaración de variables, ciclos, estructuras selectivas e incluso invocaciones a funciones.

Las funciones al ser declaradas pueden llevar un **modificador** antes del tipo. Los modificadores cambian la forma como trabaja la función. Nosotros estaremos utilizando un modificador conocido como **static**. Este modificador nos permite usar a la función sin tener que declarar un objeto de la clase a la que pertenece.

Tenemos cuatro tipos básicos de funciones: las que solo ejecutan código, las que reciben parámetros, las que regresan valores y las que reciben parámetros y reciben valores. Durante este capítulo iremos conociéndolas.

Ahora que ya conocemos los elementos básicos de las funciones, podemos ver un ejemplo de donde su uso sería adecuado. Supongamos que tenemos un programa en el cual en diferentes secciones tenemos que escribir el mismo código, pero debido a su lógica no podemos usar un ciclo, ya que está en diferentes partes del programa. En este caso nuestra única solución hasta el momento sería simplemente escribir nuevamente el código. Otra forma de resolverlo y que nos evitaría tener código repetido es usar la función. El código que se va a utilizar repetidamente se coloca adentro de la función, y cada vez que necesitemos usarlo, simplemente se invoca a la función. En nuestro ejemplo del carpintero, el cual analizamos en el capítulo anterior, nos

III EL NOMBRE DE LA FUNCIÓN

Es importante seleccionar correctamente el nombre de la función. El nombre debe hacer referencia al tipo de trabajo que lleva a cabo la función; en muchos casos podemos hacer uso de verbos. El nombre de la variable puede llevar números, pero debe empezar con una letra.

encargamos de realizar conversiones de pies y pulgadas a centímetros. La conversión se hacía con una fórmula sencilla pero un poco larga. Si ahora nuestro carpintero necesita un programa para transformar las medidas de una mesa completa a centímetros, entonces tendríamos que usar la fórmula adecuada en varios lugares. Para no copiar el mismo código varias veces, lo más conveniente sería colocar la fórmula en una función y simplemente invocarla donde fuera necesario. Empecemos a conocer cómo programar e invocar a las funciones.

Funciones que ejecutan código

El primer tipo de funciones que vamos a conocer son las que ejecutan código. Estas funciones no reciben datos y no regresan ningún dato. Solamente llevan a cabo alguna operación. Aunque en este momento no parecen muy útiles, sí lo son.

Para poder utilizar la función, debemos **declararla**. La declaración se debe hacer adentro del bloque de código correspondiente a una clase. Para los ejemplos analizados en este libro, estamos usando la clase denominada **Program**. Nuestra función **Main()** se encuentra también adentro de esta clase.

En este momento tomaremos una estrategia de programación, que aún no pertenece a la programación estructurada, en el futuro veremos las técnicas orientadas a objetos. En esta técnica usamos a la función **Main()** como administradora de la lógica y la mayor parte del proceso se llevará a cabo en las funciones.

Crearemos una aplicación y le colocaremos cada uno de los tipos de funciones que vamos a aprender, dejando a **Main()** como nuestra administradora del programa. El programa va a ser conceptualmente sencillo, ya que lo que nos interesa es comprender el funcionamiento de las funciones.

El programa simplemente se encargará de preguntarle al usuario el tipo de operación aritmética que queremos llevar a cabo y luego la realizará con los datos dados que le proporcionemos. Este programa ya fue resuelto anteriormente, pero en este caso será implementado agregando el uso de las funciones.

En el diagrama de flujo indicamos a la función por medio de un rectángulo que tiene dos franjas a los lados. Cuando lo veamos, sabemos que tenemos que ir a la función y ejecutar su código. Veamos cómo quedaría el diagrama de flujo de la aplicación.

III PROBLEMAS AL DECLARAR LAS FUNCIONES

Las funciones no pueden ser declaradas adentro de otra función. Esto nos lleva a un error de sintaxis y de lógica que debe ser corregido lo antes posible. Es importante mantener las funciones ordenadas y tener cuidado de no desbalancear los **{ }** cuando se crean nuevas funciones. Hay que recordar que deben ir adentro de una clase.

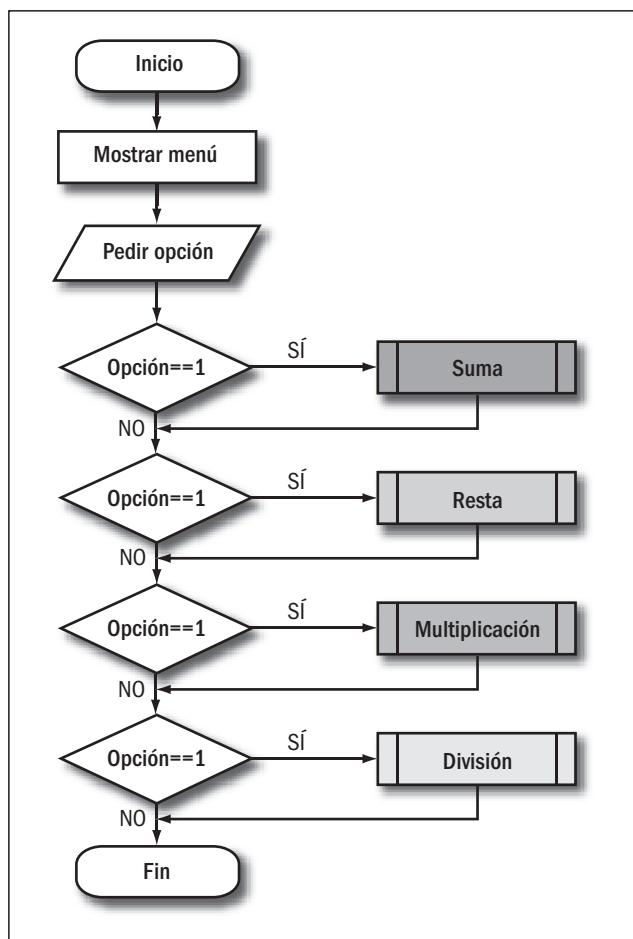


Figura 2. Aquí podemos ver el diagrama de flujo y encontrar los lugares donde se invocan a las funciones.

Vemos que nuestro diagrama es muy sencillo y también podemos localizar inmediatamente el lugar donde invitamos a las funciones. Cada una de las operaciones tendrá su propia función. En este momento no programaremos todo, iremos haciendo crecer el programa poco a poco.

Empecemos por construir el programa, dejando la administración de la lógica a **Main()**, tal y como aparece en el diagrama de flujo.

```

using System;
using System.Collections.Generic;
using System.Text;

namespace AplicacionBase
  
```

```
{  
    class Program  
    {  
        // Esta es la funcion principal del programa  
        // Aqui inicia la aplicacion  
        static void Main(string[] args)  
        {  
            // Variables necesarias  
            int opcion = 0;  
            string valor = “”;  
  
            // Mostramos el menu  
            Console.WriteLine(“1-Suma”);  
            Console.WriteLine(“2-Resta”);  
            Console.WriteLine(“3-Multiplicacion”);  
            Console.WriteLine(“4-Division”);  
  
            // Pedimos la opcion  
            Console.WriteLine(“Cual es tu opcion:”);  
            valor = Console.ReadLine();  
            opcion = Convert.ToInt32(valor);  
  
            // Checamos por la suma  
            if (opcion == 1)  
            {  
            }  
  
            // Checamos por la resta  
            if (opcion == 2)  
            {  
            }  
  
            // Checamos por la multiplicacion  
            if (opcion == 3)  
            {  
            }  
  
            // Checamos por la division
```

```

        if (opcion == 4)
        {
        }

    }
}

```

Como vemos en nuestra función **Main()** tenemos la lógica principal del programa. Hemos dejado en este momento los bloques de código de los **if** vacíos para ir colocando el código correspondiente según vallamos avanzando.

Empecemos a crear la función para la suma. Como esta función no va a recibir datos ni a regresar valores, todo se llevará a cabo en su interior. Será responsable de pedir los operandos, realizar la suma y mostrar el resultado. Entonces colocamos la declaración de la función. Esto lo hacemos después de la función **Main()**, hay que recordar tener cuidado con los **{ }**. También les debemos crear su diagrama de flujo y resolverlas de la forma usual. El diagrama de flujo es el siguiente.

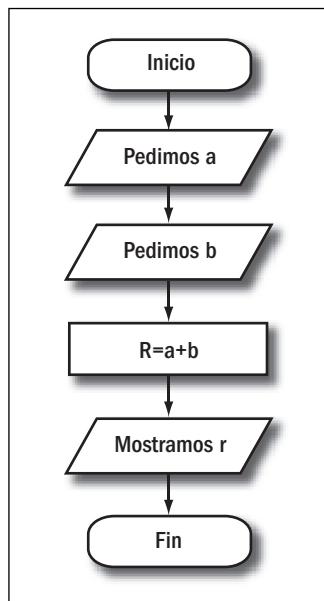


Figura 3. Éste es el diagrama de flujo de la función, también debemos hacerlos.

Nuestra función queda de la siguiente forma:

```
static void Suma()
```

```

{
    // Variables necesarias
    float a=0;
    float b=0;
    float r=0;
    string numero="";

    // Pedimos los valores
    Console.WriteLine("Dame el primer numero");
    numero = Console.ReadLine();
    a = Convert.ToSingle(numero);

    Console.WriteLine("Dame el segundo numero");
    numero = Console.ReadLine();
    b = Convert.ToSingle(numero);

    // Calculamos el resultado
    r = a + b;

    // Mostramos el resultado
    Console.WriteLine("El resultado es {0}",r);
}

}

```

Observamos que la función es **static**. Como la función no regresa ningún valor su tipo es **void**, el nombre de la función es **Suma**. Escogemos este nombre porque describe la actividad que lleva a cabo la función. Como no recibe ningún dato del programa principal no colocamos ningún parámetro y los paréntesis están vacíos. En el código tenemos todos los pasos para resolver la suma y ya los conocemos. Como podemos observar es posible declarar variables. Debemos saber que las variables que se declaran adentro de la función se conocen como **variables locales** y

III NOMBRE EN LA INVOCACIÓN

Un error muy común cuando se empieza a programar las funciones es equivocarse al escribir el nombre de la función de forma diferente durante la invocación. Por ejemplo, si nuestra función se declara como **Suma()** cuando la invocamos no podemos usar **suma()**. La invocación debe usar el nombre de la función exactamente como fue declarado.

solamente serán conocidas adentro de esa función, de esta forma solo podrán ser invocadas dentro de la función correspondiente.

Con esto la función está lista, sin embargo, si ejecutamos el programa nada diferente sucede. Esto se debe a que no hemos invocado la función. Cuando invocamos a la función su código se ejecuta. Para invocar a una función que solamente ejecuta código simplemente colocamos su nombre seguido de () y punto y coma en el lugar donde queremos que se ejecute. En nuestro caso sería el bloque de código del if para la suma, de la siguiente manera:

```
// Checamos por la suma

if(opcion==1)
{
    Suma();
}
```

Ahora si ejecutamos el programa y seleccionamos la opción de suma, veremos que efectivamente se ejecuta el código propio de la función.

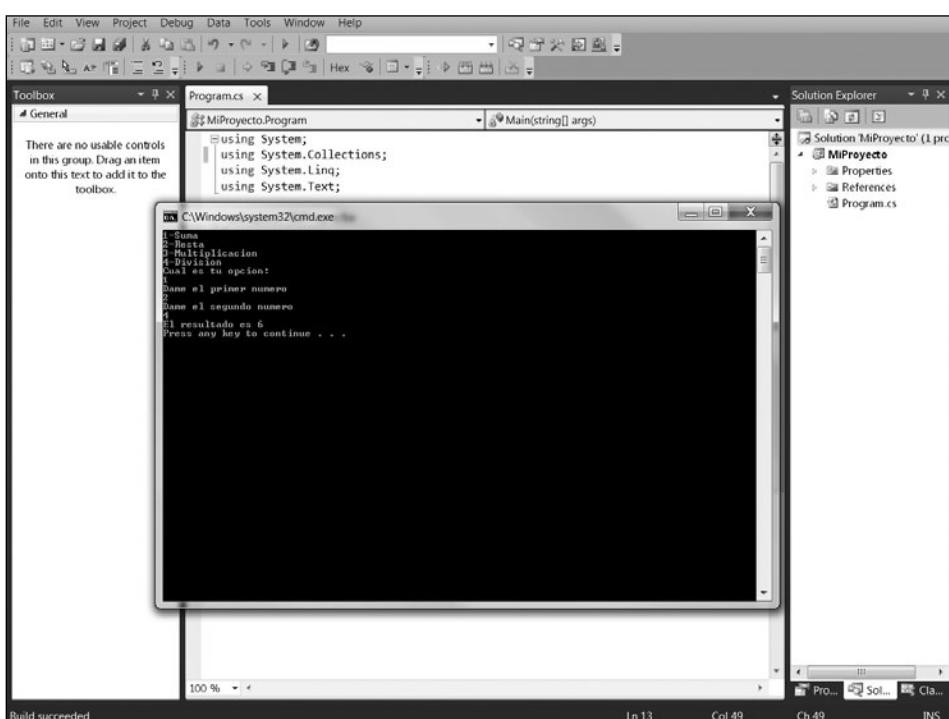


Figura 4. En la ejecución observamos cómo se está ejecutando el código de la función.

Funciones que regresan un valor

Nuestro siguiente tipo de función puede regresar un valor. Esto significa que cuando la función es invocada va a llevar a cabo la ejecución de su código. El código va a calcular un valor de alguna manera y este valor calculado por la función será regresado a quien haya invocado a la función. La invocación puede haber sido hecha por la función **Main()** o algún otra función.

Como la función va a regresar un valor, necesitamos indicar su **tipo**. El tipo va a depender del valor devuelto. Si el valor es un entero, entonces el tipo de la función es **int**. Si el valor es un flotante, la función tendrá tipo **float** y así sucesivamente. La función puede regresar cualquiera de los tipos definidos en el lenguaje y también tipos definidos por el programador y objetos de diferentes clases.

La función va a utilizar un comando especial para regresar el valor, este comando se conoce como **return** y se usa de la siguiente manera:

```
return variable;
```

En cuanto la ejecución del programa encuentra un **return**, la función es finalizada aun si no ha llegado a su fin. Al mismo tiempo que finaliza la función el valor colocado después del **return** es regresado a quien hizo la invocación. El valor puede ser colocado con una variable o explícitamente con un valor en particular. No hay que olvidar colocar el punto y coma al finalizar la sentencia. La finalización de la función se lleva a cabo aunque tengamos código escrito después del **return**.

Como la función regresa un valor, del lado del invocador necesitamos tener alguien que pueda recibir el valor regresado. Generalmente usaremos una variable, pero en algunos casos puede ser una expresión que será evaluada con el valor regresado por la función. Supongamos que nuestra función regresa un valor entero. Entonces podemos tener un código como el siguiente.

```
int n;  
n=función();
```

De esta forma el valor regresado por la función queda guardado en la variable **n** y podemos hacer uso de él. El código lo entendemos de la siguiente forma. Tenemos una variable entera **n**. Luego tenemos una asignación para **n**. Si recordamos, la asignación siempre se lleva a cabo de derecha a izquierda. Se evalúa la expresión y la función se ejecuta y calcula un valor, el cual es regresado por medio de **return**. Este valor se considera como la evaluación de la expresión y es asignado a **n**. A partir de este momento podemos usar el valor según lo necesitemos.

Ahora ya podemos empezar a usar este tipo de funciones en nuestra aplicación. Para la operación de la resta haremos uso de ella. El código en la parte del **Main()** no será tan sencillo, ya que no solamente necesitamos una invocación, debemos de recibir un valor y hacer algo con él. De esta forma en el **if** para la resta, obtendremos el valor regresado por la función y luego lo presentaremos al usuario. La función **Resta()** tiene el siguiente diagrama de flujo.

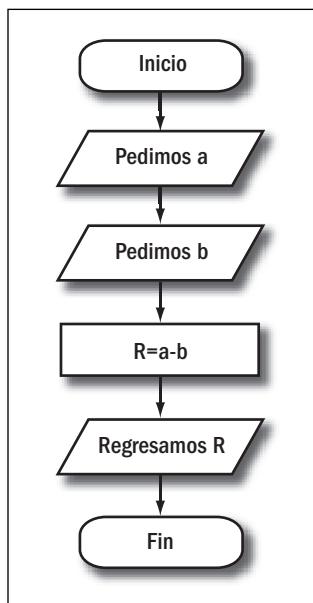


Figura 5. Podemos observar la forma cómo se crea el diagrama de esta función, no hay que olvidar el retorno del valor.

El código de la función queda de la siguiente manera:

```

static float Resta()
{

```

III

CUIDADO CON EL TIPO DE RETORNO

Es importante que la variable que recibe el valor de regreso de la función tenga el mismo tipo que la función. Si no es así podemos tener desde problemas de compilación hasta perdida de precisión en valores numéricos. En algunos casos si los tipos son diferentes será necesario llevar a cabo conversiones entre los tipos.

```

    // Variables necesarias
    float a=0;
    float b=0;
    float r=0;
    string numero="";

    // Pedimos los valores
    Console.WriteLine("Dame el primer numero");
    numero = Console.ReadLine();
    a = Convert.ToSingle(numero);

    Console.WriteLine("Dame el segundo numero");
    numero = Console.ReadLine();
    b = Convert.ToSingle(numero);

    // Calculamos el resultado
    r = a - b;

    // Retornamos el resultado
    return r;
}

```

Podemos observar que la función **Resta()** es de tipo **float** y al final hemos colocado el **return** indicando la variable cuyo valor deseamos regresar, en nuestro caso **r**. Pero no solamente debemos adicionar la función, también es necesario colocar nuevo código en el **if** que corresponde a la resta.

```

    // Checamos por la resta
    if(opcion==2)
    {

```

III LA POSICIÓN DE RETURN EN LA FUNCIÓN

No es estrictamente necesario que el **return** se ubique al final de la función, aunque generalmente lo encontraremos ahí. También es posible tener varios **return** escritos, por ejemplo uno en cada caso de una escalera de **if**. Lo que no debemos olvidar es que cuando la ejecución encuentra el **return**, la función finaliza.

```

    // Variable para nuestro resultado
    float resultado=0;

    // Invocamos y obtenemos el resultado
    resultado=Resta();

    // Mostramos el resultado
    Console.WriteLine("El resultado de la resta es
                      {0}",resultado);
}

```

Ejecutemos el programa. Podemos observar que la función actúa como se esperaba.

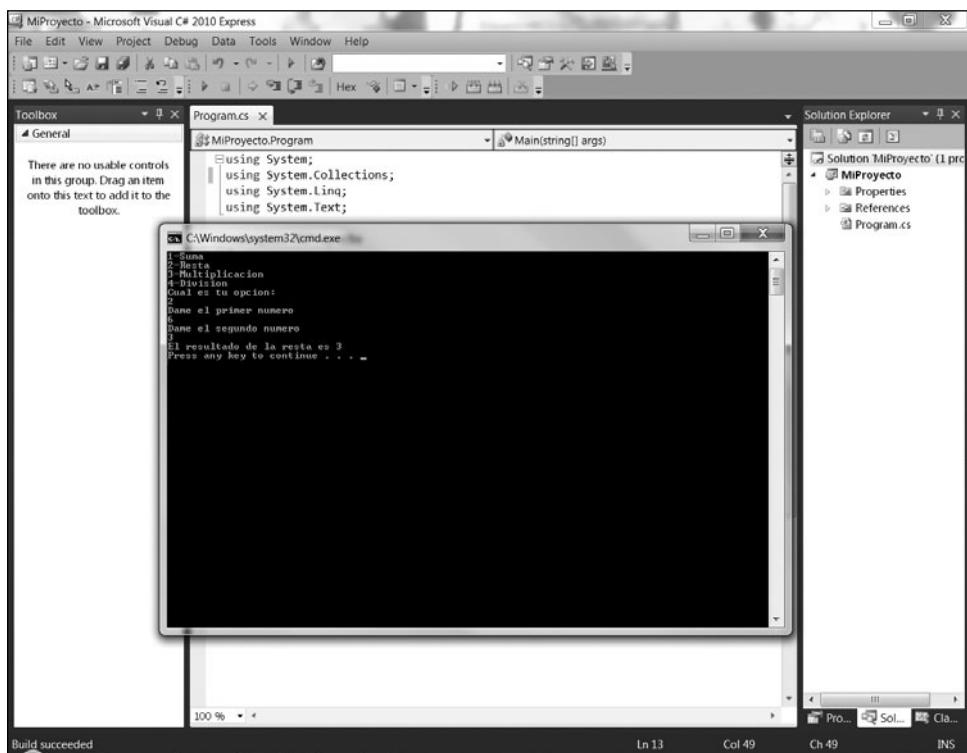


Figura 6. Vemos cómo el valor regresado por la función es desplegado en quien lo invocó, en nuestro caso `Main()`.

Funciones que reciben valores

Hasta el momento las funciones que hemos utilizado piden directamente al usuario los valores que necesitan para trabajar. Sin embargo, las funciones también

pueden recibir valores en el momento que son invocadas. De esta forma trabajarán con los valores pasados por el programa en lugar de pedirlos al usuario. Estos valores son conocidos como parámetros. Los parámetros pueden ser de cualquier tipo, ya sea de los tipos nativos de C# como entero, flotante, cadena o de tipos definidos por el programador como clases y estructuras.

Los parámetros deben llevar su tipo y su nombre. El nombre nos permite acceder a los datos que contiene y de hecho van a trabajar como si fueran variables locales a la función. Adentro de la función los usamos como variables normales. Los parámetros se definen en la declaración de la función. Adentro de los paréntesis de la función los listamos. La forma de listarlos es colocando primero en tipo seguido del nombre. Si tenemos más de un parámetro para la función, entonces debemos separarlos por medio de comas.

La invocación de la función es muy sencilla, simplemente debemos colocar el nombre de la función y, entre los paréntesis, los datos que vamos a enviar como parámetros. Los datos pueden ser situados por medio de variables o un valor colocado explícitamente.

Podemos hacer ahora la función que se encargará de la multiplicación. Esta función recibirá los operandos desde **Main()** por medio de los parámetros, realizará el cálculo y lo mostrará al usuario. Como la función **Main()** manda la información, entonces será responsabilidad de ella pedirlos a los usuarios.

El diagrama de flujo de la función **Multiplicacion()** se muestre a continuación.

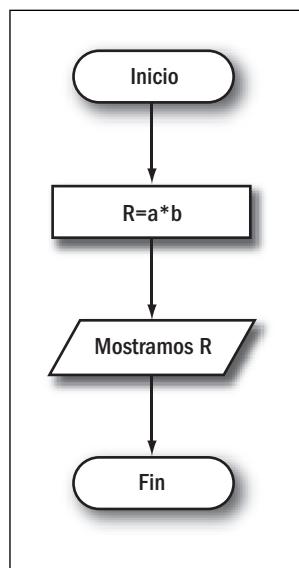


Figura 7. La función **Multiplicacion()**
es más sencilla, pues no pide los datos directamente al usuario.

El código de la función es el siguiente:

```
static void Multiplicacion(float a, float b)
{
    // Variables necesarias
    float r;

    // Calculamos el valor
    r=a*b;

    // Mostramos el resultado
    Console.WriteLine("El resultado es {0}",r);
}
```

Como podemos observar la función va a tener dos parámetros. Los parámetros son de tipo flotante y se llaman a y b. No olvidemos que están separados por una coma. Los parámetros a y b funcionarán como variables y su contenido serán los valores pasados en la invocación.

El otro lugar donde debemos de adicionar código es la función **Main()**, en este caso el **if** para la multiplicación.

```
// Checamos por la multiplicacion
if(opcion==3)
{
    // Variables necesarias
    float n1=0;
    float n2=0;
    string numero="";
}
```

III ORDEN DE LOS PARÁMETROS

Durante la invocación de la función los parámetros deben colocarse en el mismo orden en el cual fueron declarados. No hacerlo así puede llevarnos a que los parámetros reciban datos equivocados. Éste es uno de los puntos que debemos cuidar. También es bueno nombrar a los parámetros de tal manera que su nombre nos recuerde la información que va a colocársele.

```

// Pedimos los valores
Console.WriteLine("Dame el primer numero");
numero = Console.ReadLine();
n1 = Convert.ToSingle(numero);

Console.WriteLine("Dame el segundo numero");
numero = Console.ReadLine();
n2 = Convert.ToSingle(numero);

// Invocamos a la funcion
Multiplicacion(n1, n2);

}

```

En el interior del **if** creamos dos variables flotantes y le pedimos al usuario los valores con los que vamos a trabajar. Estos valores quedan guardados dentro de las variables **n1** y **n2**. Veamos ahora la invocación de la función. Como la función fue definida para recibir parámetros entonces debemos de pasar datos en su invocación. Los parámetros son colocados entre los paréntesis. De la forma como lo tenemos el parámetro **a** recibirá una copia del valor contenido en **n1** y el parámetro **b** recibirá una copia del valor contenido en **n2**. Ya con esto la función puede llevar a cabo su trabajo. Las variables usadas en la invocación no necesitan llamarse igual que los parámetros declarados en la función. Si lo necesitáramos la invocación podría llevar un valor colocado explícitamente, por ejemplo:

```

// Invocamos a la funcion
Multiplicacion(n1, 4.0f);

```

Cuando la función se invoca el parámetro **a** recibe una copia del valor contenido en **n1** y el parámetro **b** recibe el valor de **4**. Veamos cómo funciona.

III TIPOS DE LOS PARÁMETROS

Los datos que enviamos deben de ser compatibles con los tipos que la función espera recibir como parámetros y de ser posible del mismo tipo. Podemos hacer uso de conversiones antes de enviar el dato hacia la función. Con datos no compatibles tendremos problemas de compilación, en datos compatibles, pero no del mismo tipo, tendremos problemas de perdida de precisión.

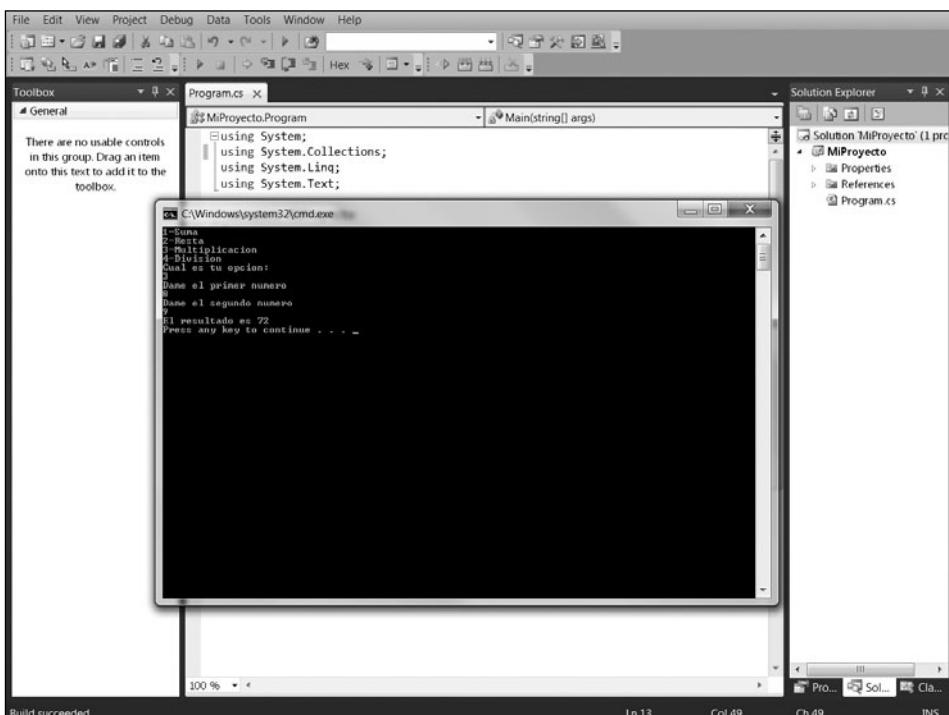


Figura 8. Vemos que la función imprime el resultado correcto, lo que nos muestra que está recibiendo la información pasada en los parámetros.

Funciones que reciben parámetros y regresan un valor

Ya hemos visto tres tipos diferentes de funciones, y seguramente ya hemos comprendido cómo funcionan y podemos fácilmente imaginar lo que hará este tipo de función. Esta función va a recibir de quien la invoque información. La información es pasada por medio de parámetros. La función lleva a cabo alguna acción o cálculo y obtiene un valor que va a ser regresado. El valor regresado es recibido en el lugar donde se invocó a la función y se puede trabajar con él.

Los parámetros serán usados como variables y los declaramos adentro de los paréntesis de la función. Tenemos que indicar el tipo que va a recibir seguido de su nombre, si tenemos más de un parámetro, entonces debemos de separarlos por comas. Como la función regresa un valor, es necesario indicar su tipo en la declaración. La variable que va a recibir el valor retornado deberá tener de preferencia el mismo tipo que la función o cuando menos un tipo que sea compatible o posible de convertir. Como siempre haremos uso de **return** para poder regresar el valor calculado. De hecho este tipo de función utiliza todas las partes de las que consiste una función. En este momento debemos programar la función que lleva a cabo la división. La función recibirá dos valores flotantes, verificará que no llevemos a cabo la división entre cero, realiza el cálculo y regresa el valor.

Si pasamos el diagrama de flujo a código, encontraremos algo interesante:

```
static float Division(float a, float b)
{
    // Variables necesarias
    float r=0;

    // Verificamos por division entre cero
    if(b==0)
    {
        Console.WriteLine("No es posible dividir
                           entre cero");
        return 0.0f;
    }
    else

    {
        r=a/b;
        return r;
    }
}
```

Adentro de la función tenemos dos **return**. ¿Esto significa que vamos a regresar dos valores? En realidad no. Únicamente podemos regresar un valor con este tipo de funciones. Lo que sucede es que cada uno de nuestros **return** se encuentra en una ruta de ejecución diferente. Observemos que una se lleva a cabo cuando el **if** se cumple y el otro se ejecuta cuando el **if** no se cumple. Para la función, solamente uno de ellos podrá ser ejecutado. También podemos notar que uno de los **return** está regresando un valor colocado explícitamente, en este caso **0.0**. Si colocamos valores de retorno explícitos, éstos deben de ser del mismo tipo que el tipo de la función. Éste es un concepto importante que nos da flexibilidad en nuestro código. Podemos regresar bajo diferentes condiciones, siempre y cuando solamente tengamos un **return** por ruta de ejecución. Si nuestra función debe regresar valores y tenemos varias rutas de ejecución, por ejemplo, varios **if** o **if** anidados, entonces debemos de colocar un **return** en cada ruta. No es posible dejar una ruta sin la posibilidad de regresar un valor. Debemos tomar en cuenta estas características para evitar problemas de lógica dentro de nuestro programa.

En el lado de la función **Main()** también debemos adicionar código. El código lo colocaremos en el bloque de código que corresponde al **if** de la división.

```

// Checamos por la division
if(opcion==4)
{
    // Variables necesarias
    float n1=0.0f;
    float n2=0.0f;
    float resultado=0.0f;
    string numero="";

    // Pedimos los valores
    Console.WriteLine("Dame el primer numero");
    numero = Console.ReadLine();
    n1 = Convert.ToSingle(numero);

    Console.WriteLine("Dame el segundo numero");
    numero = Console.ReadLine();
    n2 = Convert.ToSingle(numero);

    // Invocamos a la funcion
    resultado=Division(n1, n2);

    // Mostramos el resultado
    Console.WriteLine("El resultado es
        {0}",resultado);
}

```

Al igual que en casos anteriores, primero declaramos las variables necesarias y pedimos los valores. Luego invocamos a la función. En la invocación pasamos como parámetros a nuestras variables **n1** y **n2**. Se lleva a cabo la ejecución de la función y recibimos de regreso un valor. Este valor es asignado a la variable resultado y luego utilizado por **Main()** al mostrar el resultado.

III PARA CONOCER LAS FUNCIONES

El ambiente de desarrollo .NET nos permite ver las funciones o métodos que tiene una clase en particular. Cuando se escribe el nombre del objeto de dicha clase seguido del operador punto nos aparece una lista de ellas. Esto lo podemos hacer con la clase **Console**. El investigar estas funciones nos permite aprender más sobre .NET y lo que podemos hacer con él.

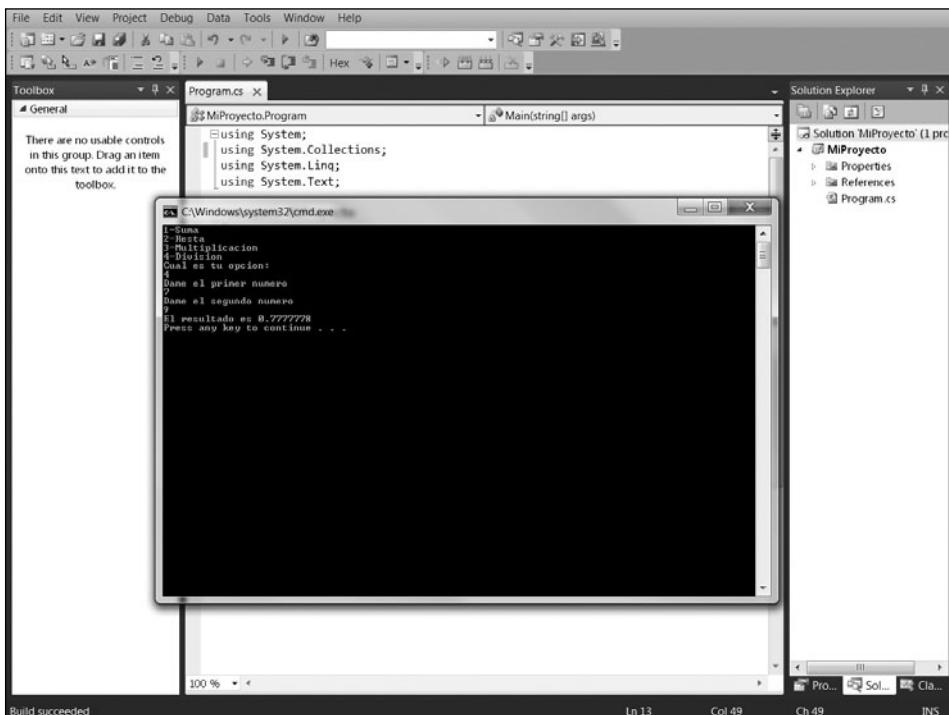


Figura 10. La división se lleva a cabo y el valor de regreso es utilizado para mostrar el resultado.

Con esto ya tenemos el programa completo.

```
using System;
using System.Collections.Generic;
using System.Text;

namespace AplicacionBase
{
    class Program
    {
        // Esta es la funcion principal del programa
        // Aqui inicia la aplicacion
        static void Main(string[] args)
        {

            // Variables necesarias
            int opcion = 0;
            string valor = "";
        }
    }
}
```

```
// Mostramos el menu
Console.WriteLine("1-Suma");
Console.WriteLine("2-Resta");
Console.WriteLine("3-Multiplicacion");
Console.WriteLine("4-Division");

// Pedimos la opcion
Console.WriteLine("Cual es tu opcion:");
valor = Console.ReadLine();
opcion = Convert.ToInt32(valor);

// Checamos por la suma
if (opcion == 1)
{
    Suma();
}

// Checamos por la resta
if (opcion == 2)

{
    // Variable para nuestro resultado
    float resultado = 0;

    // Invocamos y obtenemos el resultado
    resultado = Resta();

    // Mostramos el resultado
    Console.WriteLine("El resultado de la resta es
{0}", resultado);
}

// Checamos por la multiplicacion
if (opcion == 3)
{
    // Variables necesarias
    float n1 = 0;
    float n2 = 0;
```

```
        string numero = "";

        // Pedimos los valores
        Console.WriteLine("Dame el primer numero");
        numero = Console.ReadLine();
        n1 = Convert.ToSingle(numero);

        Console.WriteLine("Dame el segundo numero");
        numero = Console.ReadLine();
        n2 = Convert.ToSingle(numero);

        // Invocamos a la funcion
        Multiplicacion(n1, n2);
    }

    // Checamos por la division
    if (opcion == 4)
    {
        // Variables necesarias
        float n1 = 0.0f;
        float n2 = 0.0f;

        float resultado = 0.0f;
        string numero = "";

        // Pedimos los valores
        Console.WriteLine("Dame el primer numero");
        numero = Console.ReadLine();
        n1 = Convert.ToSingle(numero);

        Console.WriteLine("Dame el segundo numero");
        numero = Console.ReadLine();
        n2 = Convert.ToSingle(numero);

        // Invocamos a la funcion
        resultado = Division(n1, n2);

        // Mostramos el resultado
        Console.WriteLine("El resultado es {0}",
            resultado);
    }
}
```

```
}

} // Cierre de Main

static void Suma()
{
    // Variables necesarias
    float a = 0;
    float b = 0;
    float r = 0;
    string numero = "";

    // Pedimos los valores
    Console.WriteLine("Dame el primer numero");
    numero = Console.ReadLine();
    a = Convert.ToSingle(numero);

    Console.WriteLine("Dame el segundo numero");
    numero = Console.ReadLine();
    b = Convert.ToSingle(numero);

    // Calculamos el resultado
    r = a + b;

    // Mostramos el resultado
    Console.WriteLine("El resultado es {0}", r);
}

static float Resta()
{
    // Variables necesarias
    float a = 0;
    float b = 0;
    float r = 0;
    string numero = "";

    // Pedimos los valores
    Console.WriteLine("Dame el primer numero");
```

```
numero = Console.ReadLine();
a = Convert.ToSingle(numero);

Console.WriteLine("Dame el segundo numero");
numero = Console.ReadLine();
b = Convert.ToSingle(numero);

// Calculamos el resultado
r = a - b;

// Retornamos el resultado
return r;
}

static void Multiplicacion(float a, float b)
{
    // Variables necesarias
    float r = 0;

    // Calculamos el valor

    r = a * b;

    // Mostramos el resultado
    Console.WriteLine("El resultado es {0}", r);
}

static float Division(float a, float b)
{
    // Variables necesarias
    float r = 0;

    // Verificamos por division entre cero
    if (b == 0)
    {
        Console.WriteLine("No es posible dividir entre cero");
        return 0.0f;
    }
}
```

```

    else
    {
        r = a / b;
        return r;
    }
}

}

```

Optimizar con funciones

Ya conocemos cómo utilizar los diferentes tipos de funciones y ahora aprenderemos cómo podemos optimizar un programa haciendo uso de ellas. En el programa anterior podemos pensar que hemos utilizado todas las funciones que son necesarias. Pero nosotros sabemos que una buena oportunidad para hacer uso de las funciones es cuando tenemos código que se repite constantemente. Si observamos nuestro programa, podemos observar que esto ocurre.

En nuestro programa tenemos algo que sucede ocho veces. Se piden los valores a utilizar al usuario en varios lugares del programa. Cada vez que lo pedimos usamos tres líneas de código y la forma cómo se pide es muy similar entre ellas.

Cuando tenemos estos casos hay que llevar a cabo un poco de análisis. Lo primero que nos preguntamos es: ¿qué función lleva a cabo este código? La respuesta es pedir un valor flotante. Luego nos tenemos que preguntar si la función necesita de algún dato o datos para poder trabajar.

En una primera vista parece que no, ya que el usuario es el que va a introducir el dato. Sin embargo, si observamos mejor veremos que en realidad sí necesitamos un dato. La petición se lleva a cabo por medio de un mensaje y este mensaje puede ser diferente en cada caso. Podemos tener como dato el mensaje a mostrar por el usuario. Este va a entrar como parámetro en nuestra función. La última pregunta que nos debemos hacer es: ¿necesita la función regresar algo? En este caso es muy fácil deducir que sí. Lo que necesitamos regresar es el valor que el usuario nos ha dado.

Ya con esta información podemos construir nuestra función. Si la función fuera más complicada necesitaríamos hacer el diagrama de flujo correspondiente. Afortunadamente nuestra función es muy sencilla y queda de la siguiente manera:

```

static float PideFlotante(string mensaje)

{
    // Variables necesarias
    float numero=0.0f;
    string valor="";

    // Mostramos el mensaje
    Console.WriteLine(mensaje);

    // Obtenemos el valor
    valor = Console.ReadLine();
    numero = Convert.ToSingle(valor);

    // Regresamos el dato
    return numero;
}

```

Empecemos a ver cómo trabaja esta función que hemos creado.

El nombre es **PideFlotante** ya que como sabemos el nombre de la función tiene que indicar una referencia al tipo de trabajo que realiza. Como nuestra función regresa un valor de tipo flotante, entonces colocamos como su tipo a **float**.

La función tiene un único parámetro que es el mensaje que deseamos mostrar al usuario en la petición. Como es un mensaje de texto entonces el tipo que más conviene usar es **string**. A nuestro parámetro le nombramos **mensaje**.

En el interior de la función declaramos dos variables. Una es el número que vamos a recibir del usuario y la otra variable es la cadena que usamos con **ReadLine()**. Luego mostramos el mensaje de petición. De la forma habitual obtenemos el valor dado por el usuario. Ya para finalizar simplemente regresamos el número.

Para invocar a esta función, lo haremos de la siguiente manera:

```
n1 = PideFlotante("Dame el primer numero");
```

Ahora vemos que en lugar de utilizar las tres líneas de código por petición, simplemente hacemos una invocación a nuestra función. Pero este no es el único cambio que tenemos que hacer. En el programa original habíamos colocado variables de trabajo para apoyarnos en la petición del valor, pero ya no son necesarias.

Si cambiamos el programa para hacer uso de nuestra nueva función quedará como en el ejemplo que vemos a continuación:

```
using System;
using System.Collections.Generic;
using System.Text;
namespace AplicacionBase
{
    class Program
    {
        // Esta es la funcion principal del programa
        // Aqui inicia la aplicacion
        static void Main(string[] args)
        {

            // Variables necesarias
            int opcion = 0;
            string valor = "";

            // Mostramos el menu
            Console.WriteLine("1-Suma");
            Console.WriteLine("2-Resta");
            Console.WriteLine("3-Multiplicacion");
            Console.WriteLine("4-Division");

            // Pedimos la opcion
            Console.WriteLine("Cual es tu opcion:");
            valor = Console.ReadLine();
            opcion = Convert.ToInt32(valor);

            // Checamos por la suma
            if (opcion == 1)
            {
                Suma();
            }

            // Checamos por la resta
            if (opcion == 2)
            {
                // Variable para nuestro resultado
                float resultado = 0;

                // Invocamos y obtenemos el resultado
            }
        }
}
```

```
resultado = Resta();  
  
    // Mostramos el resultado  
    Console.WriteLine("El resultado de la resta es  
                      {0}", resultado);  
}  
  
    // Checamos por la multiplicacion  
    if (opcion == 3)  
    {  
        // Variables necesarias  
        float n1 = 0;  
        float n2 = 0;  
  
        // Pedimos los valores  
        n1 = PideFlotante("Dame el primer numero");  
        n2 = PideFlotante("Dame el segundo numero");  
  
        // Invocamos a la funcion  
  
        Multiplicacion(n1, n2);  
    }  
  
    // Checamos por la division  
    if (opcion == 4)  
    {  
        // Variables necesarias  
        float n1 = 0.0f;  
        float n2 = 0.0f;  
        float resultado = 0.0f;  
  
        // Pedimos los valores  
        n1 = PideFlotante("Dame el primer numero");  
        n2 = PideFlotante("Dame el segundo numero");  
  
        // Invocamos a la funcion  
        resultado = Division(n1, n2);  
  
        // Mostramos el resultado  
        Console.WriteLine("El resultado es {0}",
```

```
        resultado);
    }

} // Cierre de Main

static void Suma()
{
    // Variables necesarias
    float a = 0;
    float b = 0;
    float r = 0;

    // Pedimos los valores
    a = PideFlotante("Dame el primer numero");
    b = PideFlotante("Dame el segundo numero");

    // Calculamos el resultado
    r = a + b;

    // Mostramos el resultado

    Console.WriteLine("El resultado es {0}", r);
}

static float Resta()
{
    // Variables necesarias
    float a = 0;
    float b = 0;
    float r = 0;

    // Pedimos los valores
    a = PideFlotante("Dame el primer numero");
    b = PideFlotante("Dame el segundo numero");

    // Calculamos el resultado
    r = a - b;

    // Retornamos el resultado
    return r;
```

```
}

static void Multiplicacion(float a, float b)
{
    // Variables necesarias
    float r = 0;

    // Calculamos el valor
    r = a * b;

    // Mostramos el resultado
    Console.WriteLine("El resultado es {0}", r);
}

static float Division(float a, float b)
{
    // Variables necesarias

    float r = 0;

    // Verificamos por division entre cero
    if (b == 0)
    {
        Console.WriteLine("No es posible dividir entre cero");
        return 0.0f;
    }
    else
    {
        r = a / b;
        return r;
    }
}

static float PideFlotante(string mensaje)
{
    // Variables necesarias
    float numero = 0.0f;
    string valor = "";
}
```

```

    // Mostramos el mensaje
    Console.WriteLine(mensaje);

    // Obtenemos el valor
    valor = Console.ReadLine();
    numero = Convert.ToSingle(valor);

    // Regresamos el dato
    return numero;

}

}

```

Si observamos el código, podemos notar que es más fácil de leer y luce más ordenado. También hemos reducido los lugares donde podemos tener un error de sintaxis o lógica. Ejecutemos el programa para verificar si trabaja correctamente. Podemos seleccionar cualquier operación.

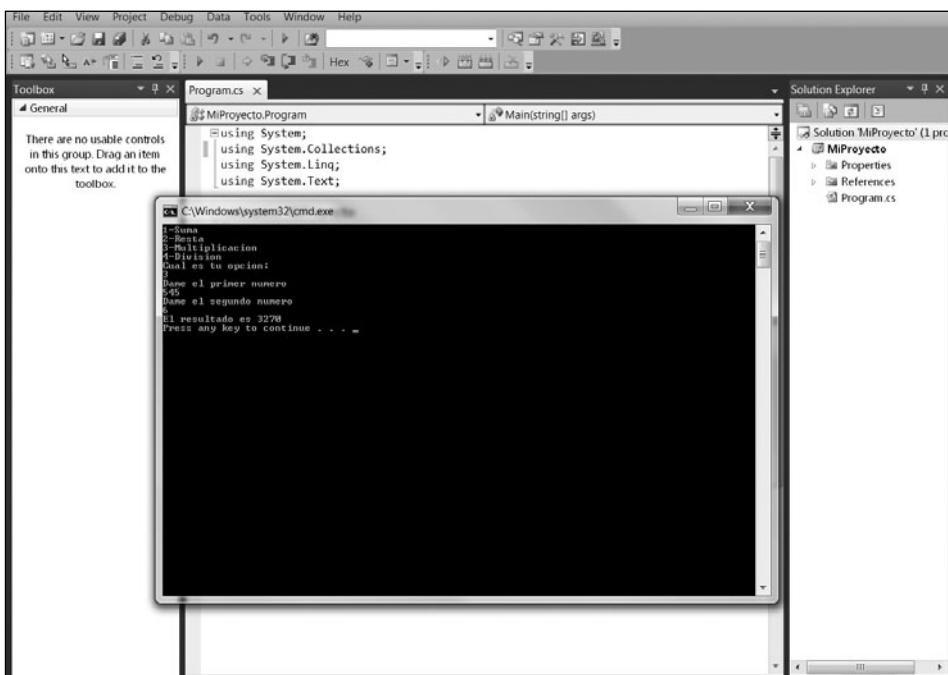


Figura 11. El programa se ejecuta correctamente con nuestra función para pedir flotantes al usuario.

Otro punto que debemos de tener en cuenta es que las funciones están invocando a nuestra función sin problema. Las funciones pueden invocar a las funciones.

Paso por copia y paso por referencia

Tenemos que aprender un concepto importante sobre los parámetros y las funciones. La mejor forma de hacerlo es por medio de un experimento sencillo. Ya anteriormente hemos comentado dos conceptos. El primero es que las variables tienen **ámbito**, es decir que las partes del programa donde se pueden utilizar depende de donde fueron declaradas. El otro concepto es que cuando invocamos a una función y pasamos parámetros, una copia del valor del parámetro es pasada a la función. Empecemos con nuestro experimento. Vamos a crear un programa con el siguiente código.

```
using System;
using System.Collections.Generic;
using System.Text;

namespace AplicacionBase
{
    class Program
    {
        // Esta es la funcion principal del programa
        // Aqui inicia la aplicacion
        static void Main(string[] args)
        {
            // Variables necesarias
            int numero = 5;

            // Valor antes de funcion
            Console.WriteLine("Valor antes de funcion {0}", numero);

            // Invocamos funcion
            Cambiar(numero);

            // Valor despues de funcion
            Console.WriteLine("Valor despues de funcion {0}", numero);
        }
        static void Cambiar(int numero)
        {
            // Cambiamos el valor
```

```

        numero = 17;
    }
}

```

En el programa tenemos dos funciones, la función **Main()** y la función **Cambiar()**. En **Main()** estamos declarando una variable y le asignamos el valor de **5**. Luego imprimimos un mensaje que muestra el valor de nuestra variable antes de invocar a la función. En seguida invocamos a la función y después de ésta imprimimos otro mensaje que nos muestra el valor de la variable.

Esto lo hacemos para poder verificar si la variable se ve afectada por la función. La función es muy sencilla, simplemente recibe un parámetro y le asigna un nuevo valor. Este programa podría parecer que presentaría un valor diferente antes y después de la función, pero ejecutémoslo y veamos qué ocurre.

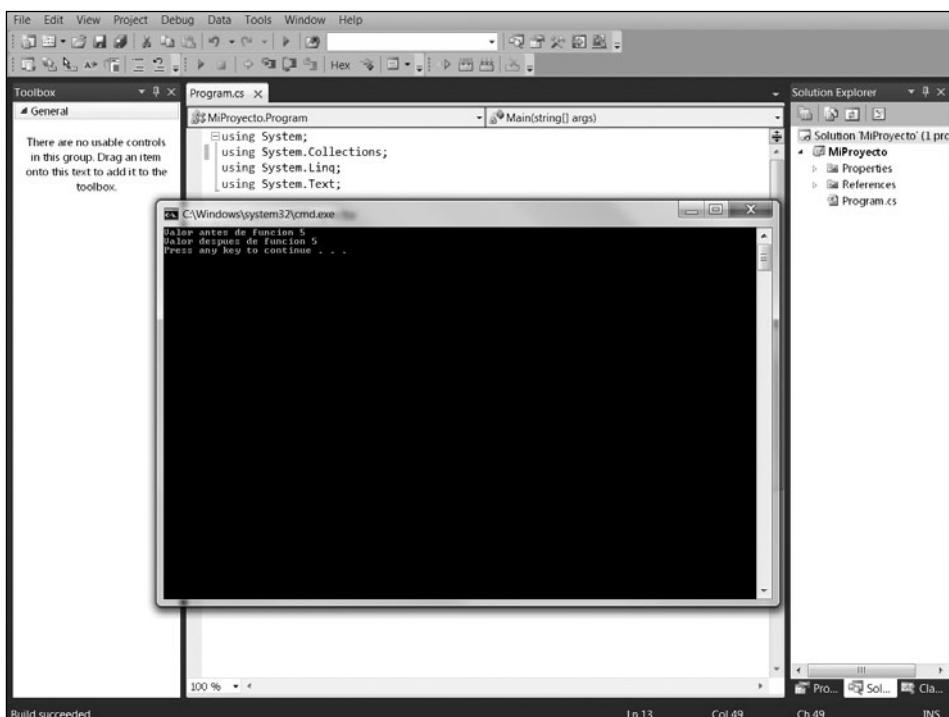


Figura 12. El valor de la variable es el mismo antes y después de la ejecución de la función.

Al ejecutar notamos que el valor de la variable es igual antes y después de invocar a la función. Es decir que la función no alteró el valor de la variable, aunque parecía que lo iba a hacer. Veamos la razón de esto.

La primera razón de por qué esto sucede es que la variable fue declarada en la función **Main()**, es decir que solamente esta función la conoce y puede utilizarla directamente. **Main()** es el ámbito de la variable. Luego tenemos que cuando el parámetro se pasa de esta forma, en realidad no es la variable lo que se está pasando, sino una copia del valor de ella. Por eso cuando nuestra función trata de cambiar el valor, cambia a la copia y no a la variable original.

Cuando pasamos los parámetros de esta forma decimos que pasan por **copia**. Pero existe otra forma de pasar los parámetros y se conoce como **paso por referencia**. Cambiemos nuestro programa para que quede de la siguiente manera:

```
using System;
using System.Collections.Generic;
using System.Text;
namespace AplicacionBase
{
    class Program
    {
        // Esta es la funcion principal del programa
        // Aqui inicia la aplicacion
        static void Main(string[] args)
        {
            // Variables necesarias
            int numero = 5;

            // Valor antes de funcion
            Console.WriteLine("Valor antes de funcion {0}",
                numero);

            // Invocamos funcion
            Cambiar(ref numero);

            // Valor despues de funcion
            Console.WriteLine("Valor despues de funcion {0}",
                numero);
        }

        static void Cambiar(ref int numero)
        {
            // Cambiamos el valor
            numero = 17;
        }
    }
}
```

```

        }
    }
}

```

Ejecutemos primero el programa y luego veamos qué ocurre.

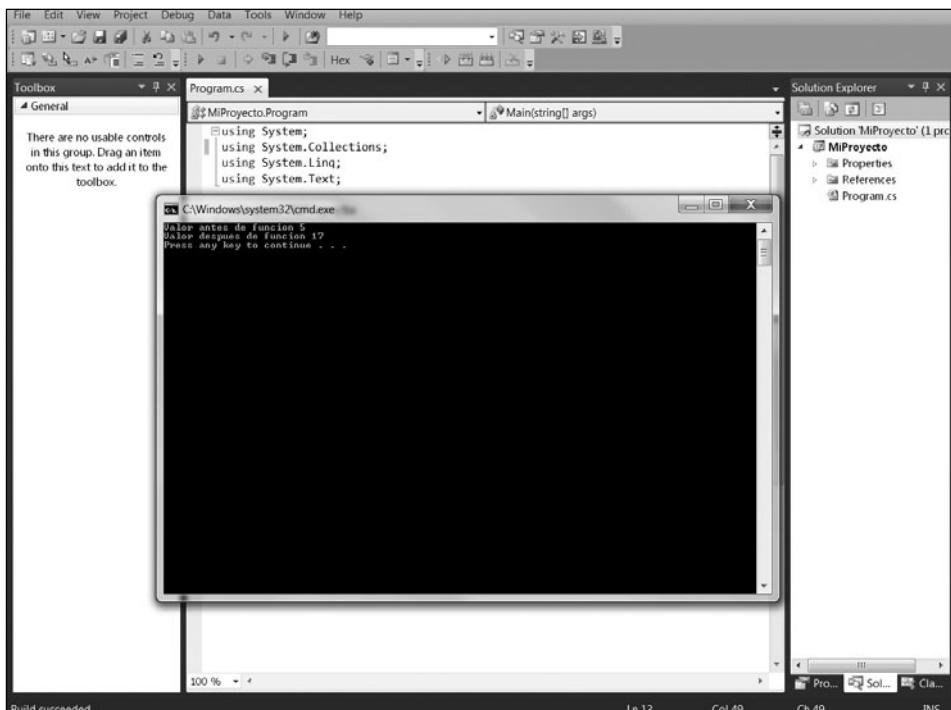


Figura 13. En este caso el valor de la variable si es cambiado. Esto se debe a que se pasó la referencia a la variable.

En este caso estamos pasando el parámetro por referencia. Cuando esto sucede, en lugar de pasar una copia del valor, lo que pasamos es una referencia al lugar donde se encuentra la variable en memoria. De esta forma cualquier asignación a la variable se realiza sobre la variable original y su valor puede ser cambiado aunque haya sido definida en otro ámbito.

Para poder indicar que un parámetro necesita pasarse por referencia es necesario usar la instrucción **ref**, tanto en la invocación de la función como en su declaración. Podemos tener más de una variable pasada por referencia en una función.

El paso por referencia es muy útil, en especial para casos cuando es necesario que una función regrese más de un valor. Como sabemos, **return** solamente puede regresar un valor, pero al usar referencias podemos tener una función que regrese más de un valor por medio de la modificación de las variables pasadas.

Veamos un ejemplo de esto. Vamos a crear una función que permita intercambiar los valores de dos variables. Esto problema puede parecer sencillo, pero sin el uso de las referencias es imposible resolverlo con una sola función. A continuación veremos cómo queda nuestra programa.

```
using System;

using System.Collections.Generic;
using System.Text;

namespace AplicacionBase
{
    class Program
    {
        // Esta es la funcion principal del programa
        // Aqui inicia la aplicacion
        static void Main(string[] args)
        {

            // Variables necesarias
            int a = 5;
            int b = 3;

            // Valor antes de funcion
            Console.WriteLine("Valores antes de funcion a={0},
                b={1}", a,b);

            // Invocamos funcion
            Cambiar(ref a, ref b);

            // Valor despues de funcion

            Console.WriteLine("Valores despues de funcion a={0},
                b={1}", a, b);
        }

        static void Cambiar(ref int x, ref int y)
        {
            // Variable de trabajo
            int temp = 0;
```

```

    // Cambiamos el valor
    temp = x;
    x = y;
    y = temp;
}

}

```

Si lo ejecutamos, obtenemos lo siguiente:

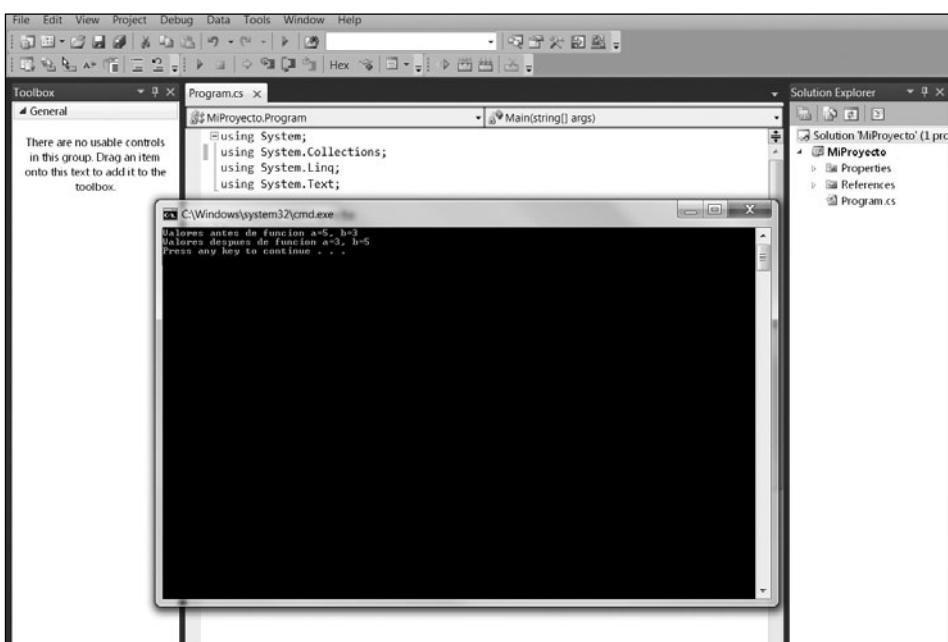


Figura 14. Los valores han sido intercambiados por la función.

Como vemos, hacemos uso de una variable de trabajo para ayudarnos en el intercambio de los valores. El paso por referencia nos permite modificar ambas variables.

Uso de parámetros de default

Una de las características nuevas que encontramos en la última versión de C# es el uso de **parámetros de default** en las funciones. Con las funciones que hemos estado utilizando es necesario proveer a la función de todos los parámetros con los que fue declarada. Es decir, si la función tiene dos parámetros, cuando la invocamos debemos darle dos valores para que trabaje con ella.

Con los parámetros de default es posible colocar un valor predeterminado al parámetro, de tal forma que si en la invocación de la función no se da explícitamente el valor, entonces se usa el valor predeterminado. Esto puede resultar útil cuando tenemos parámetros que en la gran mayoría de los casos usan el mismo valor. De esta forma cuando invocamos a la función lo hacemos usando solamente los valores que cambien, lo cual nos permite escribir menos y solamente cuando es necesario colocamos el valor en el parámetro.

Esto lo podemos ver en el siguiente ejemplo. El cual tiene una función con parámetros de default y es invocada de forma tradicional y luego haciendo uso del parámetro de default. Cuando hace uso del parámetro de default impuesto usa el valor de **0.16**.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Cap5_5
{
    class Program
    {

        static void Main(string[] args)
        {
            double costo = 50.0;
            double imp = 0.0;
            double total = 0.0;

            // Hacemos uso de la función de manera tradicional

            imp = CalculaImpuesto(costo, 0.25);
            total = costo + imp;

            // Imprimimos resultado
            Console.WriteLine("El total es ${0}", total);

            // Hacemos uso de la función con parámetro de default
            // Hay que notar que solamente pasamos un parámetro,
            el otro usa

            // el valor predeterminado
```

```
    imp = CalculaImpuesto(costo);
    total = costo + imp;

    // Imprimimos resultado
    Console.WriteLine("El total es ${0}", total);

}

public static double CalculaImpuesto(double cantidad,
    double impuesto = 0.16f)
{
    double impuestoCalculado;

    impuestoCalculado = cantidad * impuesto;
    return impuestoCalculado;
}
}
```

Con esto hemos visto los principios y los usos fundamentales de las funciones. Las funciones son muy útiles y las continuaremos utilizando frecuentemente.



RESUMEN

Las funciones nos permiten tener secciones especializadas de código en nuestra aplicación, reducir la cantidad de código necesario y reutilizar el código de manera eficiente. Las funciones constan de cinco partes y hay cuatro tipos de ellas. La función puede recibir información por medio de los parámetros y regresar información por medio de return. Es posible pasar los parámetros por copia o por referencia.



ACTIVIDADES

TEST DE AUTOEVALUACIÓN

1 ¿Qué son las funciones?

2 ¿Qué es invocar una función?

3 ¿Cuáles son las partes de las funciones?

4 ¿Para qué nos sirve el modificador static?

5 ¿Cuáles son los cuatro tipos de funciones?

6 Cuando la función no regresa un valor,
¿cuál es su tipo?

7 ¿Qué tipos de valores pueden regresar las
funciones?

8 ¿Para qué sirve return?

9 ¿Cómo se colocan los parámetros?

10 ¿Cómo podemos usar las funciones para
optimizar nuestro programa?

11 ¿Qué es el paso por copia?

12 ¿Qué es el paso por referencia?

EJERCICIOS PRÁCTICOS

1 Hacer una función para transformar de
grados a radianes.

2 Hacer una función para transformar de
grados centígrados a grados Fahrenheit.

3 Hacer una función que calcule el
perímetro de cualquier polígono regular
dando el número de lados y sus
dimensiones.

4 Hacer una función que calcule el factorial
de un número.

5 Hacer una función que dado un número
nos regrese una cadena donde se
encuentre escrito en palabras.

Los arreglos

En los capítulos anteriores hemos utilizado las variables. Colocamos información en éstas para luego procesarlas. Nuestros programas pueden tener muchas variables, en especial cuando la cantidad de información a guardar es mucha. Los **arreglos** nos brindan una forma de poder administrar la información fácilmente, y son especialmente útiles cuando necesitamos muchas variables y el trabajo que se realiza sobre ellas es el mismo.

Los arreglos	188
Declaración de los arreglos de una dimensión	189
Asignación y uso de valores	191
Arreglos de dos dimensiones	196
Arreglos de tipo jagged	205
Los arreglos como parámetros a funciones	212
Resumen	215
Actividades	216

LOS ARREGLOS

Pensemos en un problema donde los arreglos nos pueden ser útiles. En el **Capítulo 4** creamos un programa donde se calculaba el promedio de un grupo de alumnos. En él usamos un ciclo **for**, mediante el cual se pedía la calificación constantemente. Para la calificación siempre se usaba la misma variable. Pensemos ahora, que nos encontramos con la necesidad de que nuestro programa no solamente muestre el promedio, sino también la calificación más alta y más baja del salón. Estos cálculos requieren ser utilizados en diferentes partes del programa, por lo que no es posible colocar todo adentro del ciclo **for** con el que fue diseñado en un principio.

El problema al que nos estamos enfrentando es que, como sólo hemos hecho uso de una variable para las calificaciones, cuando necesitemos procesar nuevamente la información desde nuestro software, ya no tendremos almacenado el valor de la calificación anterior en la variable, sólo tendremos la última calificación capturada. ¿Cómo podemos resolver esto? Una forma de hacerlo sería crear y utilizar muchas variables como: **calif1**, **calif2**, **calif3**..., etcétera.

Al principio esta idea puede parecer correcta, pero manejar muchas variables de forma independiente, luego de un tiempo puede tornarse engoroso, y también podremos encontrar un caso en el que no sabemos cuántas variables necesitaremos. Sin embargo, esto nos puede dar una idea sobre lo que necesitamos.

Si observamos el nombre de las variables, vemos que todas llevan en su nombre **calif**, es decir que todas se refieren a la calificación. Podemos pensar que están conceptualmente agrupadas para la calificación. Otro punto que notamos es que las variables están numeradas, como si tuvieran un **índice** que nos sirve para identificarlas.

Los arreglos son similares a estos conceptos ya que son grupos de variables y estas variables serán referenciadas por el mismo nombre. Para poder acceder a una variable del arreglo usaremos un número de índice, ya que todas las variables adentro de un arreglo serán de un mismo tipo. Un punto muy importante que no debemos olvidar cuando trabajemos con los arreglos es que éstos están basados en **índice cero**, esto quiere decir que el primer elemento del arreglo se encuentra en la posición **0**, no en la posición **1** como podríamos pensar. No olvidar este punto nos evitará muchos problemas de lógica en el futuro.

III EL TAMAÑO DEL ARREGLO

En un arreglo es conveniente colocar la cantidad correcta de elementos, ya que una vez creado, no puede crecer a menos que utilicemos una función especial. Si hacemos lo último en forma constante, esto indica que nuestro programa tiene problemas de diseño. En el próximo capítulo veremos estructuras que permiten variar la cantidad de elementos a guardar.

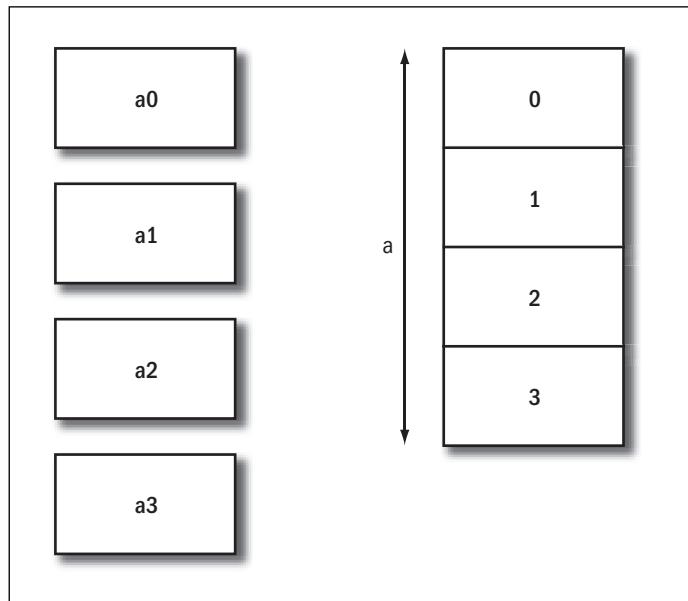


Figura 1. En esta figura vemos la comparación entre tener varias variables y un arreglo.

Declaración de los arreglos de una dimensión

Los arreglos pueden tener diferentes **dimensiones**, y si el arreglo se parece a una simple lista, como la lista de calificaciones que tenemos, entonces decimos que es de una dimensión. Estos arreglos también se conocen como **monodimensionales** o **unidimensionales**. Si el arreglo es como una tabla con varios renglones y varias columnas, entonces es un arreglo de dos dimensiones o **bidimensional**. Primero trabajaremos con los arreglos de una dimensión y luego pasaremos a arreglos de dos dimensiones o más. Para poder trabajar un arreglo, primero es necesario declararlo. En la declaración nosotros indicamos su tipo, su nombre y su tamaño.

```
tipo[] nombre = new tipo[tamaño];
```

III CUIDADO CON EL RANGO DEL ARREGLO

Los arreglos tienen una cantidad finita de elementos y ésta se indica en el momento en que se declaran. Cuando intentamos acceder a uno de sus elementos debemos colocar un valor de índice válido y que no exceda el tamaño del arreglo. Un error común con los arreglos es intentar acceder a elementos más allá de los que tiene.

La declaración puede parecer un poco extraña, por lo que a continuación haremos un ejemplo más claro y real, y lo explicaremos a fondo.

```
float[] calificaciones= new float[10];
```

En C# los arreglos son **objetos**, y deberemos usar **new** al declararlos. El arreglo será de tipo flotante, y usaremos **[]** para indicar su declaración. Luego debemos colocar el nombre con el que lo identificaremos. En nuestro ejemplo se llama: **calificaciones**. Del lado derecho de la sentencia tenemos la **instanciación** del arreglo. Indicaremos entre **[]** la cantidad de elementos que deseamos tener. La cantidad puede ser colocada de forma explícita, tal como está en el ejemplo, o por medio del contenido de una variable. Podemos exemplificar esto de la siguiente forma:

```
int n = 10;  
  
float[] calificaciones = new float[n];
```

En este caso se tendrá la cantidad de elementos igual al valor guardado en la variable **n**. En algunas ocasiones podemos conocer los valores que colocaremos adentro del arreglo, por lo que podemos declararlo y asignarle sus valores en la misma sentencia. Esto lo hacemos indicando primero el tipo y los **[]** seguidos del nombre del arreglo y en el lado derecho de la sentencia colocamos entre **{ }** los elementos que se le desean asignar al arreglo. Estos elementos deberán estar separados por comas. Veámoslo exemplificado de manera más clara:

```
float[] valores = { 1.5f, 3.78f, 2.1f };
```

En este caso hemos creado un arreglo llamado **valores** y tendrá tres elementos con los valores colocados. La siguiente figura nos muestra cómo quedaría.

III COLOCAR EL TAMAÑO CORRECTAMENTE

La cantidad de datos debe ser un valor válido. No podemos colocar números negativos, ni un tamaño de cero ya que no tendría sentido. Si la cantidad de elementos a crear se pasará por medio de una variable, ésta debe de ser de tipo entera. No tener en cuenta esto traerá problemas al compilar la aplicación.

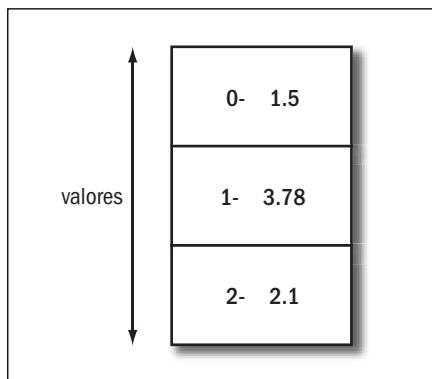


Figura 2. El arreglo tiene tres elementos con los índices 0, 1 y 2.

Asignación y uso de valores

Ya hemos visto cómo declarar el arreglo. Ahora tenemos que aprender cómo poder colocar información en su interior y hacer uso de ésta.

Para poder asignarle un valor a alguno de los elementos del arreglo necesitamos hacer uso del índice del elemento que queremos utilizar, y como dijimos antes, no debemos olvidar que el primer elemento se encuentra en la posición **0**.

Supongamos que queremos asignarle la calificación **8.5** al tercer alumno.

```
calificaciones[2] = 8.5f;
```

Lo primero es usar el nombre del arreglo, tal y como con las variables, pero entre **[]** colocamos el índice del elemento al que se lo queremos asignar. El tercer alumno se encuentra en el índice **2**. Esto se debe a que empezamos a contar desde cero: **0, 1, 2**. Luego del lado derecho de la asignación colocamos el valor a asignar. El control del índice también se puede hacer por medio de una variable de tipo entero. El valor contenido en la variable será usado para acceder al elemento del arreglo.

```
calificaciones[indice] = 8.5f;
```

Para el índice, tener la capacidad de utilizar una variable es muy útil, ya que esto nos permitirá recorrer el arreglo con alguno de los ciclos estudiados en los primeros capítulos del libro. Los valores contenidos adentro del arreglo pueden usarse como variables normales, por ejemplo en un cálculo.

```
impuesto = costo[n] * 01.5f;
```

Y el desplegado es igualmente fácil.

```
Console.WriteLine("El valor es {0}", costo[n]);
```

Ahora que ya tenemos los conceptos básicos de los arreglos podemos hacer nuestro programa, donde se pedirá la cantidad de alumnos y las calificaciones. El programa encontrará el promedio, la calificación más alta y la calificación más baja. Para demostrar que los datos no se pierden al ser guardados en el arreglo haremos cada cálculo en un ciclo diferente, y para comprender mejor esto, veamos a continuación el diagrama de flujo correspondiente:

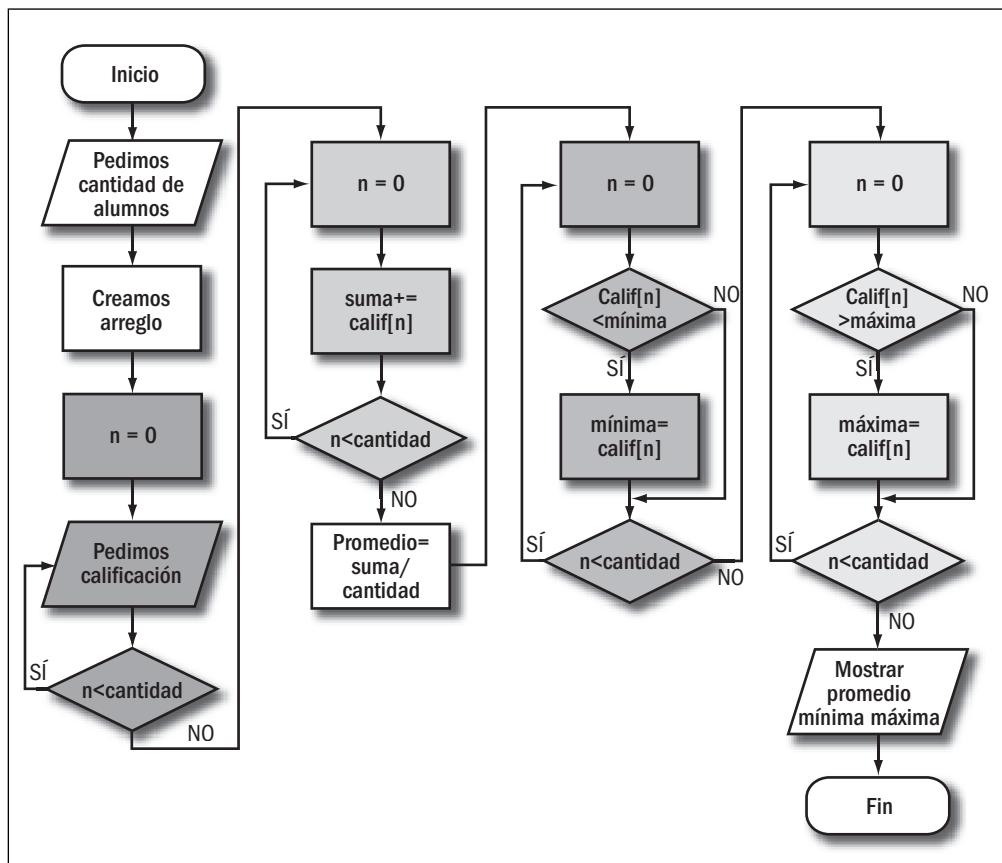


Figura 3. El diagrama nos muestra las diferentes partes donde usamos la información del arreglo.

```
using System;
using System.Collections.Generic;
```

```
using System.Text;
namespace AplicacionBase
{
    class Program
    {
        // Esta es la función principal del programa
        // Aquí inicia la aplicación
        static void Main(string[] args)
        {
            // Variables necesarias
            int cantidad = 0;      // Cantidad de alumnos
            int n = 0;              // Variable de control de ciclo
            string valor = "";

            // Variables para el promedio
            float suma = 0.0f;
            float promedio = 0.0f;

            float minima = 10.0f; // Variable para la
                                  // calificación mínima
            float máxima = 0.0f; // Variable para la calificación
                                  // máxima

            // Pedimos la cantidad de alumnos
            Console.WriteLine("Dame la cantidad de alumnos");
            valor = Console.ReadLine();
            cantidad = Convert.ToInt32(valor);

            // Creamos el arreglo
            float[] calif = new float[cantidad];

            // Capturamos la información
            for (n = 0; n < cantidad; n++)
            {
                Console.Write("Dame la calificación ");
                valor = Console.ReadLine();
                calif[n] = Convert.ToSingle(valor);
            }

            // Encontramos el promedio
```

```

        for (n = 0; n < cantidad; n++)
    {
        suma += calif[n];
    }

    promedio = suma / cantidad;

    // Encontramos la calificación mínima
    for (n = 0; n < cantidad; n++)
    {
        if (calif[n] < minima)
            minima = calif[n];
    }

    // Encontramos la calificación máxima
    for (n = 0; n < cantidad; n++)
    {
        if (calif[n] > maxima)
            máxima = calif[n];
    }

    // Desplegamos los resultados
    Console.WriteLine("El promedio es {0}", promedio);
    Console.WriteLine("La calificación mínima es {0}", minima);
    Console.WriteLine("La calificación máxima es {0}", máxima);

} // Cierre de main

} // Cierre de la clase
}

```

Inicialmente declaramos las variables que necesitará la aplicación. Sin embargo, aún no hemos declarado el arreglo. Esto se debe a que el tamaño del arreglo dependerá de la cantidad de alumnos. Una vez que tenemos esa cantidad creamos el arreglo. Luego, por medio de un ciclo **for** capturamos la información. Debemos notar que el ciclo inicia en **0**. Esto se debe a que el primer elemento del arreglo se encuentra en el índice **0**. El ciclo se recorre tantas veces como elementos tenga el arreglo. Para calcular el promedio, recorremos el arreglo de nuevo por medio de un ciclo **for** con la variable **suma** para obtener la sumatoria de todas las calificaciones. Luego simplemente calculamos el promedio, dividiendo la suma entre la cantidad de alumnos.

Para poder calcular la calificación menor declaramos una variable llamada **minima**. A esta variable la inicializamos con el valor más alto posible para las calificaciones (la razón de esto tendrá sentido enseguida). Creamos nuevamente un ciclo **for** y recorremos el arreglo. Para cada elemento del arreglo comparamos su valor con el valor de nuestra variable. Si el valor del elemento del arreglo es menor que **minima**, eso significa que hemos encontrado un valor menor, por lo que **minima** se actualiza con este nuevo valor menor. Esto se repite hasta que hemos terminado con todos los elementos del arreglo y **minima** en cada caso tendrá el valor más pequeño encontrado hasta esa vuelta del ciclo.

De igual forma encontramos la calificación más alta del arreglo pero en este caso creamos una variable llamada **máxima**. Esta variable se inicializa con el valor más pequeño que puede tener una calificación. Hacemos un ciclo y recorremos todo el arreglo, y a cada elemento contenido en éste lo comparamos con el valor de **máxima**. Si es mayor eso significa que hemos encontrado un nuevo máximo, por lo que actualizamos la variable. Si lo anterior se cumple, al finalizar el ciclo tendremos la calificación más grande para ese grupo de alumnos.

Para finalizar el programa, simplemente mostramos en la consola o línea de comandos, los resultados que hemos obtenido.

Ejecutemos y probemos cómo funciona la teoría hasta aquí explicada.

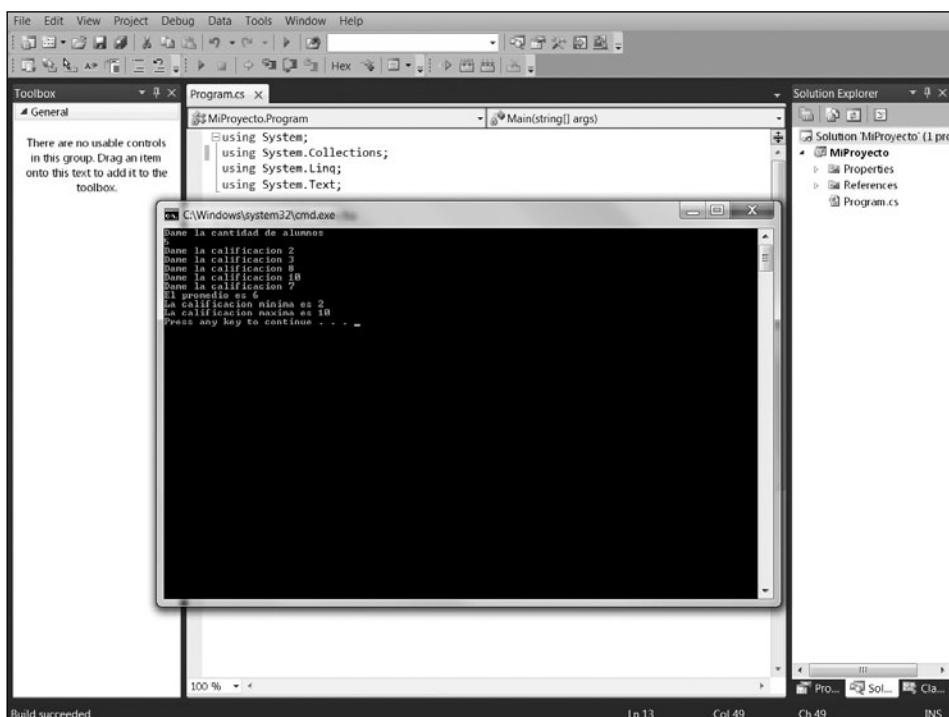


Figura 4. El programa guarda la información en un arreglo y podemos utilizarla cuando sea necesario.

Arreglos de dos dimensiones

Hasta aquí hemos visto que los arreglos nos ayudan a guardar información y a trabajar de una manera más cómoda con ella. Cuando tenemos mucha información que almacenar es más fácil manipular un arreglo que muchas variables. Los arreglos que vimos son similares a una lista, pero no todos los problemas se pueden resolver con este esquema, y veamos por qué. Supongamos que ahora debemos hacer un nuevo programa para una fábrica que produce automóviles y desea tener la información de las unidades producidas a diario. La información será procesada por semana, con el promedio semanal de vehículos producidos. Hasta el momento el problema sería muy similar al anterior. Podríamos crear un arreglo de 7 elementos llamado semana y estaría resuelto, pero ahora también desean la información por mes. El mes tiene cuatro semanas, por lo que podemos pensar en tener cuatro arreglos, uno que se corresponda con cada semana, pero en realidad existe una forma mejor de solucionar este problema.

Si pensamos un poco más en el problema veremos que podemos guardar la información en una tabla, cada **columna** sería una semana y cada **renglón** representaría un día. Esto se conoce como **matriz**, y es un arreglo de dos dimensiones. En los arreglos de dos dimensiones tenemos que utilizar dos índices. Uno controlará el renglón y el otro la columna. Con la creación de estos dos índices es posible que accedamos a cualquier **celda** ubicada dentro de la matriz.

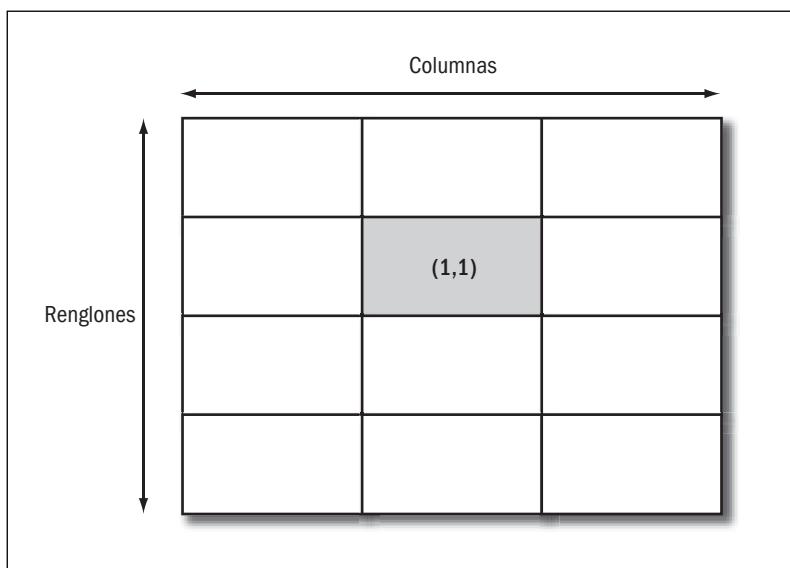


Figura 5. El diagrama nos muestra un arreglo de dos dimensiones y cómo podemos localizar una celda por medio de su renglón y de su columna.

Al igual que con los arreglos de una sola dimensión, los índices están basados en **0**. Esto es tanto para los renglones como para las columnas.

Declaración de los arreglos de dos dimensiones

La declaración es similar al arreglo de una dimensión, pero indicamos la cantidad de elementos en cada dimensión.

```
float[,] tabla = new float[5,3];
```

En este caso, hemos creado una matriz de valores flotantes llamada **tabla**. Hay que notar que hemos colocado una coma entre los **[]** para indicar que serán dos dimensiones. En el lado derecho indicamos el tamaño de cada dimensión. Para este ejemplo tenemos cinco columnas y tres renglones.

También es posible declarar la matriz por medio de variables.

```
float[,] tabla = new float[n, m];
```

Acceso a un arreglo de dos dimensiones

La utilización del arreglo de dos dimensiones es muy sencilla, podemos acceder a la información si indicamos el renglón y la columna de la celda que nos interesa. Por ejemplo, si deseamos hacer uso del valor que se encuentra almacenado en la celda **(3, 2)** hacemos lo siguiente:

```
impuesto = producto[3,2] * 0.15;
```

En este caso se toma el valor de la celda **(3,2)** del arreglo **producto**, se multiplica por **0.15** y el valor resultante se asigna a **impuesto**.

De igual forma podemos hacer la asignación al arreglo.

```
producto[3,2] = 17.50;
```

Para mostrar el valor en la consola hacemos lo siguiente:

```
Console.WriteLine("El costo es {0}", producto[3,2]);
```

No debemos olvidar que los índices siempre se inician en cero, y que no es posible colocar índices más grandes que la cantidad de elementos contenidos en el arreglo. Con este conocimiento podemos comenzar a trabajar en un ejemplo. Tomaremos

el código del ejemplo de la escuela y ahora haremos posible que trabaje para varios salones. Para este caso pediremos la cantidad de salones y la cantidad de alumnos. Cada columna representará el salón y los renglones guardarán la información de los alumnos. De esta forma podemos capturar toda la información de la escuela y calcular el promedio, y la calificación mínima y máxima de toda la escuela.

Para que este programa funcione de forma eficiente utilizaremos la sentencia **for** y también un tipo de ciclo mencionado en un capítulo anterior. Éstos son los **ciclos enlazados**, o simplemente, un ciclo dentro de otro ciclo. La forma de utilizarlo es la siguiente: tendremos un ciclo para recorrer cada uno de los salones y dentro de éste tendremos otro ciclo que se dedicará a recorrer los estudiantes. De esta forma podemos cubrir todos los salones y todos los estudiantes de cada salón.

A continuación un ejemplo práctico de ciclos enlazados.

```
using System;
using System.Collections.Generic;
using System.Text;

namespace AplicacionBase
{
    class Program
    {
        // Esta es la función principal del programa
        // Aquí inicia la aplicación
        static void Main(string[] args)
        {
            // Variables necesarias
            int n = 0; // Ciclo exterior
            int m = 0; // Ciclo interior

            // Ciclos enlazados
            for (n = 0; n < 3; n++)
            {
                for (m = 0; m < 5; m++)
                {
                    Console.WriteLine("Ciclo exterior {0}, ciclo interior
{1}", n, m);
                }
            }
        }
    }
}
```

```
}
```

Compilemos y ejecutemos el programa.

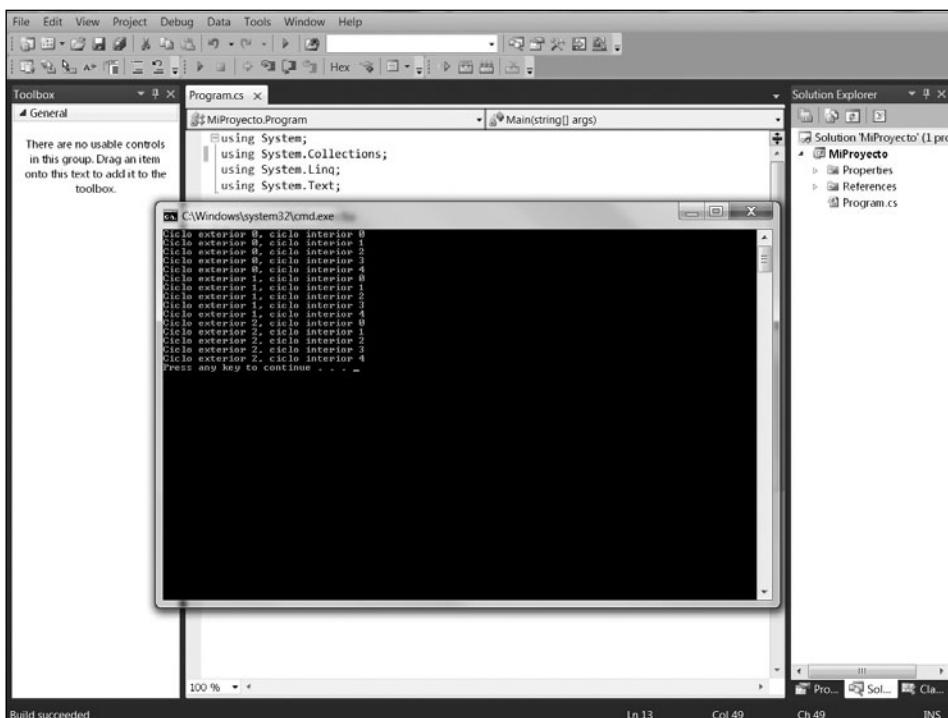


Figura 6. Vemos cómo se recorren los ciclos. Para cada paso del ciclo exterior tenemos una vuelta completa del ciclo interior.

Al primer ciclo lo llamaremos **ciclo exterior** y al ciclo que se coloca adentro lo llamaremos **ciclo interior**. Cada vez que el ciclo exterior avanza un paso, el ciclo interior da una vuelta completa. Este tipo de comportamiento es el que nos

III | EL MÉTODO CLEAR()

Los arreglos tienen un método conocido como **Array.Clear()**. Este método limpia el arreglo y al hacerlo coloca todos sus elementos en **0**, **false** o **null** dependiendo del tipo del arreglo. El método tiene tres parámetros. El primero es el arreglo a borrar. Luego debemos colocar el índice a partir de donde deseamos borrar y por último la cantidad de elementos a limpiar.

permite recorrer toda la tabla. Para cada columna recorremos todos los renglones. Veamos en el diagrama de flujo del programa cómo sería:

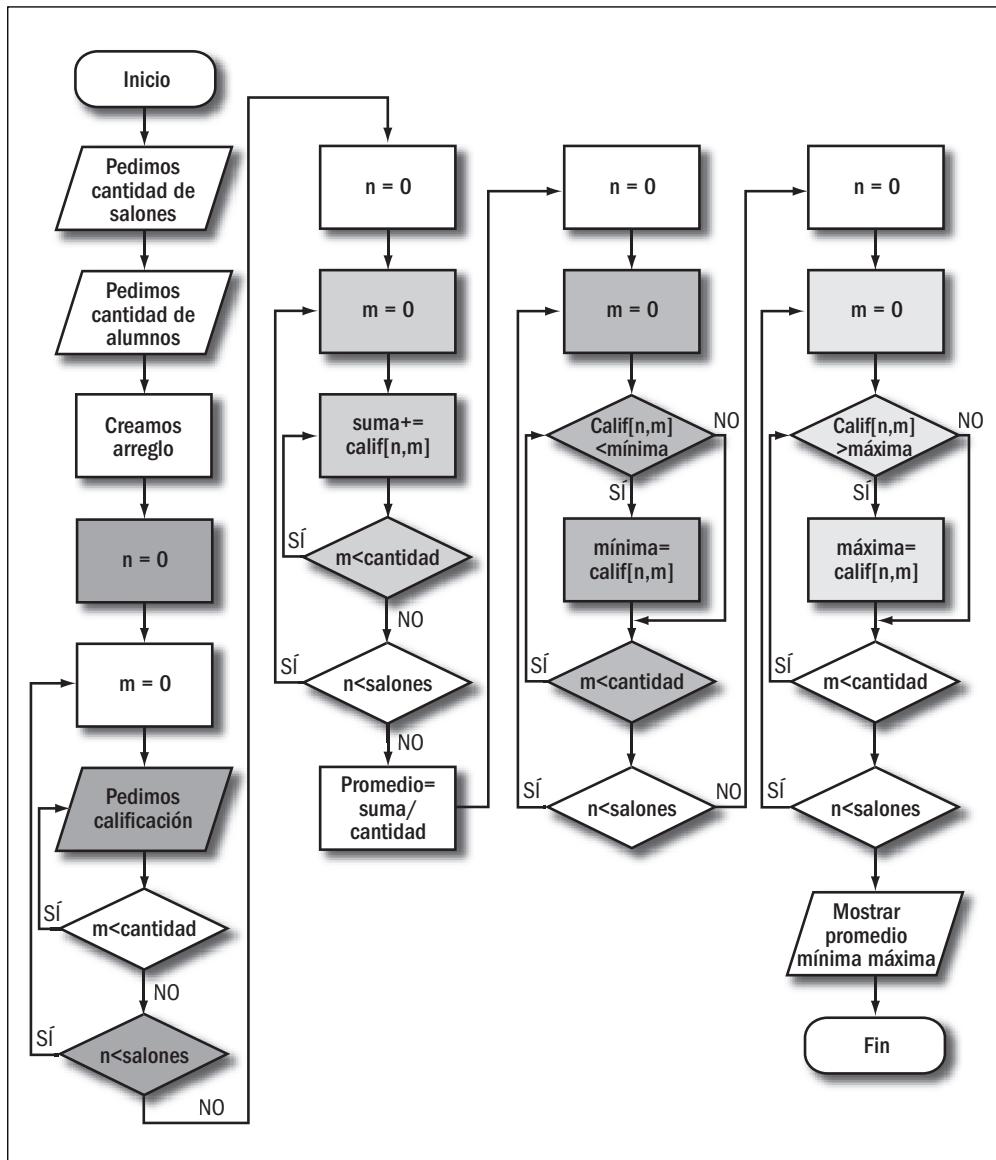


Figura 7. El diagrama muestra cómo podemos localizar los ciclos enlazados fácilmente.

Nuestro programa queda constituido de la siguiente manera:

```

using System;
using System.Collections.Generic;
    
```

```

using System.Text;

namespace AplicacionBase
{
    class Program
    {
        // Esta es la función principal del programa
        // Aquí inicia la aplicación
        static void Main(string[] args)
        {

            // Variables necesarias
            int cantidad = 0;      // Cantidad de alumnos
            int salones = 0;       // Cantidad de salones
            int n = 0;             // Variable de control de ciclo salones
            int m = 0;             // Variable de control del ciclo alumnos
            string valor = "";

            // Variables para el promedio
            float suma = 0.0f;
            float promedio = 0.0f;

            float minima = 10.0f; // Variable para la calificación mínima
            float máxima = 0.0f; // Variable para la calificación máxima

            // Pedimos la cantidad de salones
            Console.WriteLine("Dame la cantidad de salones");
            valor = Console.ReadLine();
            salones = Convert.ToInt32(valor);

            // Pedimos la cantidad de alumnos

```



ARREGLOS DE MÁS DIMENSIONES

Podemos tener arreglos de más de dos dimensiones. El mecanismo para declararlos y utilizarlos es el mismo. Simplemente debemos tener cuidado con el manejo de los índices para cada dimensión y si el arreglo es de tres dimensiones necesitaremos tres índices, si es de cuatro, cuatro índices y así sucesivamente.

```

Console.WriteLine("Dame la cantidad de alumnos por salón");
valor = Console.ReadLine();
cantidad = Convert.ToInt32(valor);

// Creamos el arreglo
float[,] calif = new float[salones, cantidad];

// Capturamos la información
for (n = 0; n < salones; n++) // Ciclo salones

{
    Console.WriteLine("Salón {0}", n);
    for (m = 0; m < cantidad; m++) // Ciclo alumnos
    {
        Console.Write("Dame la calificación ");
        valor = Console.ReadLine();
        calif[n, m] = Convert.ToSingle(valor);
    }
}

// Encontramos el promedio
for (n = 0; n < salones)           // Ciclo salones
{
    for (m = 0; m < cantidad; m++) // Ciclo alumnos
    {
        suma += calif[n, m];
    }
}
promedio = suma / (cantidad * salones);

// Encontramos la calificación mínima

```

III CRECIMIENTO DE LA MEMORIA USADA POR LOS ARREGLOS

Aunque no parezca, cuando usamos arreglos de muchas dimensiones la cantidad de memoria necesaria para guardarlos puede crecer rápidamente sin notarlo. Un arreglo de tres dimensiones de enteros con un tamaño de 256x256x256 no puede parecer mucho, pero necesita 67,108,864 bytes de memoria.

```

        for (n = 0; n < salones; n++)      // Ciclo salones
    {
        for (m = 0; m < cantidad; m++) // Ciclo alumnos
        {
            if (calif[n, m] < minima)
                minima = calif[n, m];
        }
    }

    // Encontramos la calificación maxima
    for (n = 0; n < salones; n++) // Ciclo salones
    {
        for (m = 0; m < cantidad; m++) // Ciclo alumnos
        {
            if (calif[n, m] > maxima)
                máxima = calif[n, m];
        }
    }

    // Desplegamos los resultados
    Console.WriteLine("El promedio es {0}", promedio);
    Console.WriteLine("La calificación mínima es {0}", minima);
    Console.WriteLine("La calificación máxima es {0}",
                      máxima);

} // Cierre de main

}
}

```

Lo primero que necesitamos es obtener las dimensiones de nuestra matriz. Para esto debemos saber cuántos salones o cuántas columnas y qué cantidad de alumnos o renglones necesitaremos. Como podemos observar, en el código tenemos los ciclos enlazados. Un ciclo recorre los salones y el otro recorre los alumnos. Cuando hacemos la petición de la información, es posible asignarla a la celda adecuada por medio de las variables **m** y **n**. De esta forma capturamos toda la información y queda guardada en las celdas del arreglo.

Teniendo esta información, podemos procesarla. Como trabajamos sobre toda la tabla, nuevamente usamos los ciclos anidados, desplegando al final sólo los resultados. Ejecutemos el programa y veamos cómo funciona:

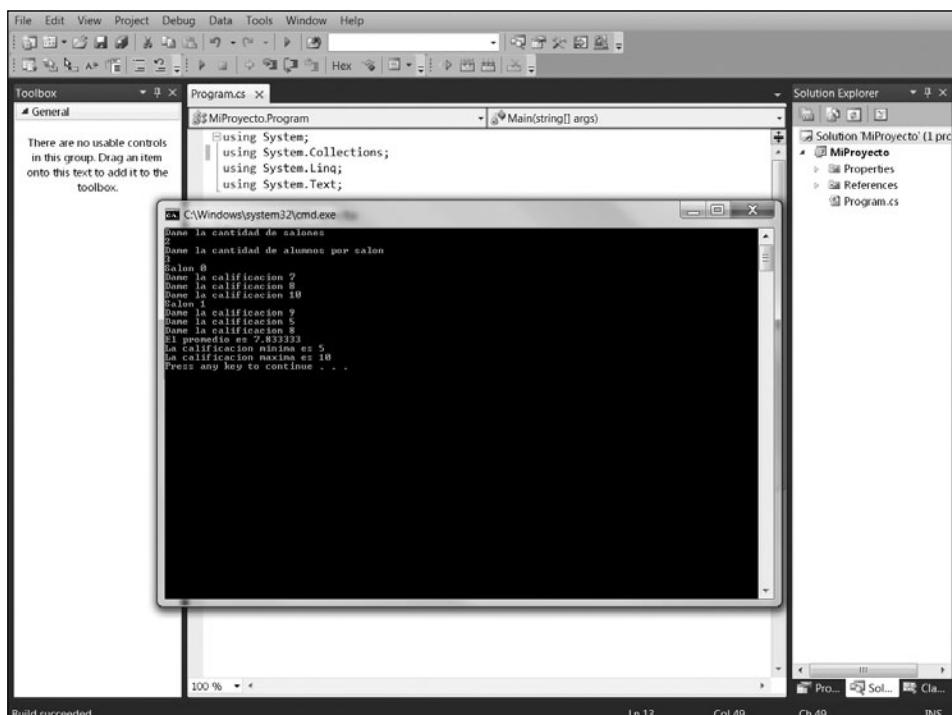


Figura 8. Observemos que capturamos la información y luego la procesamos en diferentes partes del programa para obtener los resultados finales.

Es posible que nosotros hayamos procesado una columna en particular. Para esto simplemente mantenemos fijo el valor de la columna y recorremos los renglones con un ciclo. Un ejemplo de esto sería el siguiente:

```

for (n = 0; n < 10; n++)
{
    Console.WriteLine("El valor es {0}", producto[5,n]);
}

```

En este caso recorremos la columna **5** de la matriz *producto*. El índice del renglón es colocado por la variable de control del ciclo.

También es posible mantener fijo el renglón y recorrer todas las columnas. Para esto nuevamente es necesario un solo ciclo.

```

for (n = 0; n < 10; n++)
{
}

```

```
Console.WriteLine("El valor es {0}", producto[n,3]);
}
```

En el caso anterior recorrimos todas las columnas del renglón 3.

Arreglos de tipo jagged

En el ejemplo anterior todas las columnas tienen la misma cantidad de renglones, es decir, que todos los salones tendrían la misma cantidad de alumnos, sin embargo, en la práctica esto no sucederá. Tendremos salones con diferentes cantidades de alumnos, y los arreglos que hemos visto hasta el momento no permiten tener columnas con diferente cantidad de renglones. Esto puede ser una limitación, y puede significar que tengamos renglones sin utilizar en varias columnas. Esto nos lleva a desperdiciar memoria. Una solución podría ser tener un arreglo que nos permita lograr esto. Para hacerlo tenemos que crear un arreglo de arreglos. Esto se conoce como **arreglo jagged**. En lugar de usar una matriz, lo que haremos es crear un arreglo, pero cada elemento de este arreglo será a su vez otro arreglo. Así podemos controlar de forma independiente la cantidad de renglones en cada arreglo. Estos arreglos son más flexibles que los tradicionales, pero requieren que seamos más cuidadosos con ellos.

La siguiente figura nos muestra cómo está constituido un arreglo jagged.

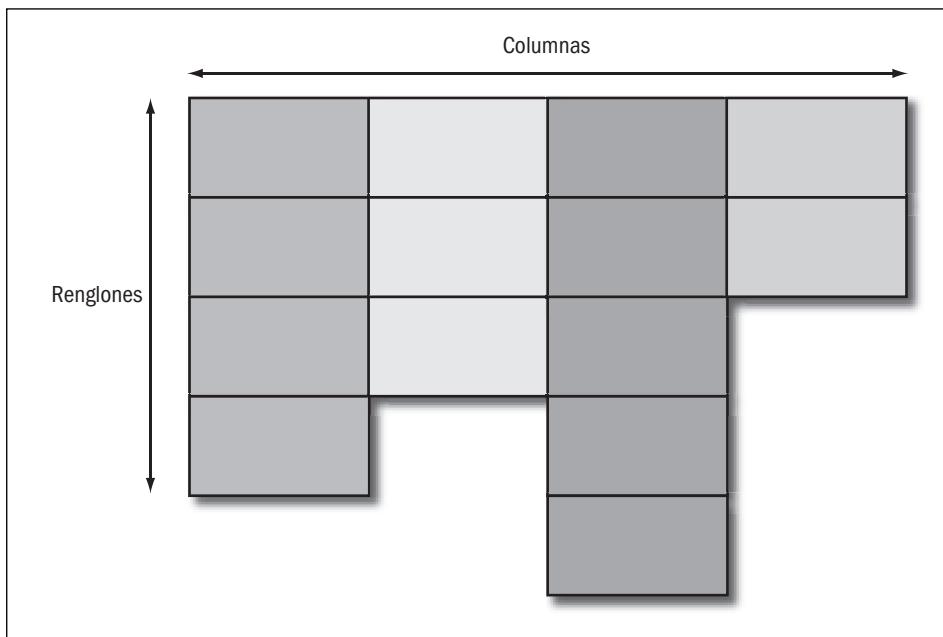


Figura 9. Tenemos un arreglo y cada elemento del arreglo es a su vez otro arreglo.

Al arreglo que contiene los demás arreglos lo llamaremos **arreglo contenedor** para que podamos reverenciarnos a él fácilmente.

Declaración de un arreglo jagged

La declaración de los arreglos jagged es ligeramente más complicada que la de los tradicionales. Cuando los declaramos debemos declarar en primer lugar el arreglo contendor y luego cada uno de los arreglos independientes que tiene.

Veamos un primer ejemplo de declaración:

```
// Declaramos arreglo contenedor
int[][] Costos = new int[3][];
// Declaramos los arreglos
Costos[0] = new int[15];
Costos[1] = new int[20];
Costos[2] = new int[10];
```

Lo que declaramos es un arreglo jagged de tipo entero que se llamará **Costos**. Como observamos, junto al **int** tenemos **[][]**. Esto indica que es un arreglo de arreglos y no una matriz. En el lado derecho de la asignación tenemos algo similar. Aquí indicamos que nuestro arreglo tiene 3 columnas, pero inmediatamente dejamos **[]** vacíos. Esto es lo que nos permite tener la flexibilidad de darle un tamaño propio a cada renglón. Ya tenemos inicializado el arreglo contenedor. Ahora necesitamos inicializar cada uno de los arreglos internos. Primero indicamos cuál es la columna a inicializar. En el ejemplo tenemos **Costos[0]**, es decir, que inicializaremos el arreglo que se encuentra en la primera columna. Del lado derecho está **int[15]**, con lo que indicamos que tendrá quince elementos, es decir, 15 renglones para la columna 0.

De forma similar la columna 1 tendrá 20 renglones y la columna 2 tendrá 10 renglones. Podemos ver que ahora tenemos diferente cantidad de renglones para cada columna. Al igual que con los arreglos tradicionales, la cantidad de elementos que tendrá el arreglo puede ser colocada de forma explícita o por medio de

III ADMINISTRAR LOS ARREGLOS JAGGED

Como en un arreglo de tipo jagged nosotros podemos tener diferente cantidad de renglones, es necesario tener un mecanismo que nos permita saber cuántos renglones tenemos, ya que de lo contrario corremos el riesgo de colocar índices indebidos que provocarán errores de lógica. Estos errores pueden ser difíciles de corregir, por lo que es mejor estar preparados antes de utilizarlos.

una variable. Si conocemos con anterioridad la información que tendrá el arreglo jagged podemos hacer lo siguiente:

```
int[][] valores = new int[3][];
valores[0] = new int[] { 9, 3, 1, 7, 2, 4 };
valores[1] = new int[] { 2, 9 };
valores[2] = new int[] { 3, 5, 2, 9 };
```

En este caso creamos un arreglo jagged de tres columnas y luego, al momento de crear cada uno de los arreglos internos, los instanciamos colocando los valores directamente. En este caso cada columna tendrá la cantidad de renglones según la cantidad de valores utilizados en su instanciación. La columna **0** tendrá seis renglones, la columna **1** tendrá dos renglones y la columna **2** tendrá cuatro renglones.

A continuación se muestra otra forma para inicializar este tipo de arreglos cuando conocemos previamente los elementos que utilizaremos adentro de él:

```
int[][] valores = new int[][] {
{
    new int[] { 9, 3, 1, 7, 2, 4 },
    new int[] { 2, 9 },
    new int[] { 3, 5, 2, 9 }
};
```

En el ejemplo listado no indicamos directamente la cantidad de columnas. C# encontrará este valor dependiendo de la cantidad de arreglos que se declaren adentro del bloque de código. En este caso declaramos tres arreglos internos, por lo que la cantidad de columnas es de tres. Al igual que en el caso anterior, la cantidad de renglones por columnas dependerá de la cantidad de elementos que declaramos en cada inicialización, entonces tendremos las mismas cantidades que en el ejemplo anterior.

Acceder a un arreglo jagged

Para acceder a los elementos guardados adentro de un arreglo jagged también necesitamos utilizar índices. Un índice será aplicado para indicar cuál elemento del arreglo contenedor utilizaremos, es decir el número de columna. El otro índice entonces nos indicará el elemento del arreglo interno que queremos acceder.

Por ejemplo, para asignar un valor realizamos lo siguiente:

```
productos[6][7] = 5.7f;
```

En este ejemplo vemos que se selecciona el elemento **6** del arreglo contenedor, o si lo preferimos la columna **6**. Adentro de esa columna seleccionamos el elemento **7** y ahí es dónde se coloca el valor **5.7**. Si lo que necesitamos es mostrar el contenido de un elemento, el esquema es similar a lo que ya conocemos.

```
Console.WriteLine("El valor es {0}", productos[5][n]);
```

Ahora modificaremos algunas partes del programa de la escuela para usar el arreglo de tipo jagged. Lo que haremos es tener salones con diferente cantidad de alumnos y luego simplemente mostraremos las calificaciones de cada salón.

El diagrama es el siguiente:

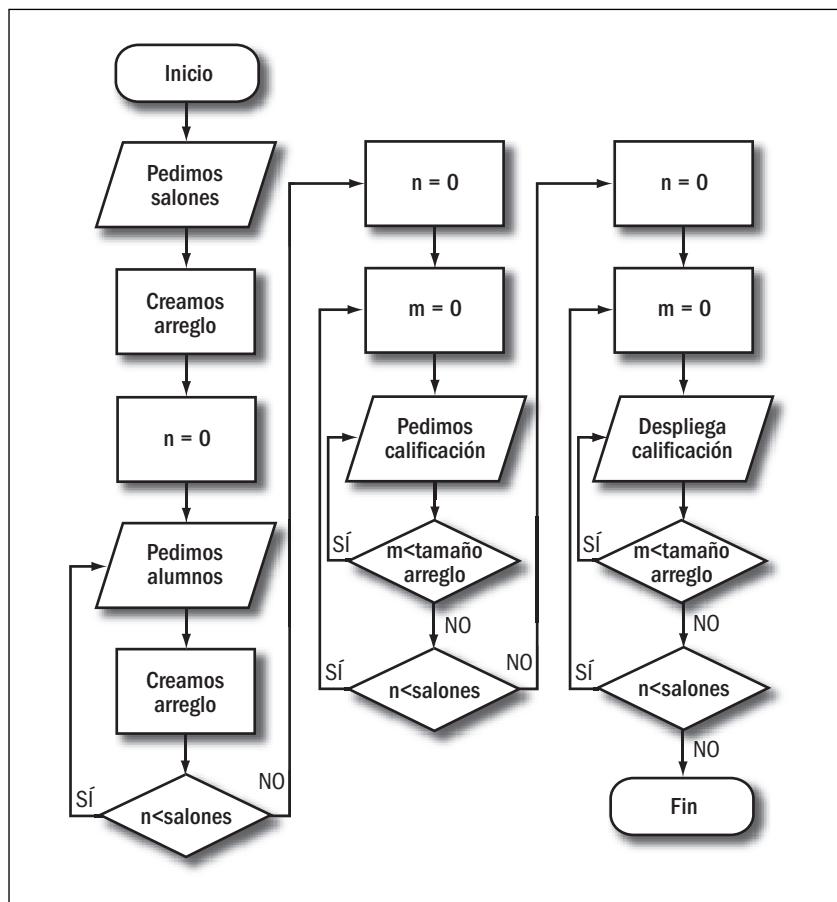


Figura 10. Hemos agregado un ciclo para la creación de cada uno de los arreglos internos.

El código del programa quedará de la siguiente forma:

```
using System;
using System.Collections.Generic;
using System.Text;

namespace AplicacionBase
{
    class Program
    {
        // Esta es la función principal del programa
        // Aquí inicia la aplicación
        static void Main(string[] args)
        {
            // Variables necesarias
            int cantidad=0;           // Cantidad de alumnos
            int salones=0; // Cantidad de salones
            int n=0;           // Variable de control de ciclo
            salones
            int m=0;           // Variable de control del ciclo
            alumnos
            string valor="";

            // Variables para el promedio
            float suma=0.0f;
            float promedio=0.0f;

            float minima=10.0f; //Variable para la
            calificación
            mínima
            float maxima=0.0f; //Variable para la calificación
            maxima

            // Pedimos la cantidad de salones
            Console.WriteLine("Dame la cantidad de salones");
            valor=Console.ReadLine();
            salones=Convert.ToInt32(valor);

            // Creamos el arreglo
            float[][] calif= new float [salones][];

            // Pedimos los alumnos por salón
```

```

        for(n=0;n<salones;n++) // Ciclo salones
        {
            Console.WriteLine("Dame la cantidad de
                alumnos
                para el salon {0}",n);
            valor=Console.ReadLine();
            cantidad=Convert.ToInt32(valor);

            // Instanciamos el arreglo
            calif[n]=new float[cantidad];

        }

        // Capturamos la información
        for(n=0;n<salones;n++) // Ciclo salones
        {
            Console.WriteLine("Salon {0}",n);
            for(m=0;m<calif[n].GetLength(0);m++) // 
                Ciclo
                alumnos
            {
                Console.Write("Dame la calificación ");
                valor=Console.ReadLine();
                calif[n][m]=Convert.ToSingle(valor);
            }
        }

        // Desplegamos la información
        Console.WriteLine("— Información —");
        for (n = 0; n < salones; n++) // Ciclo salones
        {
            Console.WriteLine("Salon {0}, n");
            for (m = 0; m < calif[n].GetLength(0); m++) // Ciclo
                alumnos
            {
                Console.WriteLine("El alumno {0} tiene {1} ", m,
                    calif[n][m]);
            }
        }
    }
}

```

```
    }  
}  
}
```

Podemos observar que como primera medida hemos creado el arreglo contenedor de acuerdo con la cantidad de salones que tendremos, luego, por medio de un ciclo, recorremos cada uno de los salones preguntando en cada uno la cantidad de alumnos, entonces ahí creamos el arreglo interno con el tamaño obtenido. Debemos tener cuidado y prestar mucha atención en el uso de la sintaxis para procesar cada arreglo que manejaremos.

Veamos que del lado derecho de la sentencia tenemos el índice del arreglo que crearemos y en el lado izquierdo indicamos el tamaño. De esta forma, quedará creado un arreglo de ese tamaño en tal posición del arreglo contenedor.

Ya creado el arreglo **jagged** procedemos a colocar la información en su interior. Al igual que antes usamos ciclos enlazados, y solamente deberemos cambiar la asignación de acuerdo con la sintaxis del arreglo jagged. Lo mismo ocurre cuando queremos imprimir los contenidos de arreglo jagged en la consola.

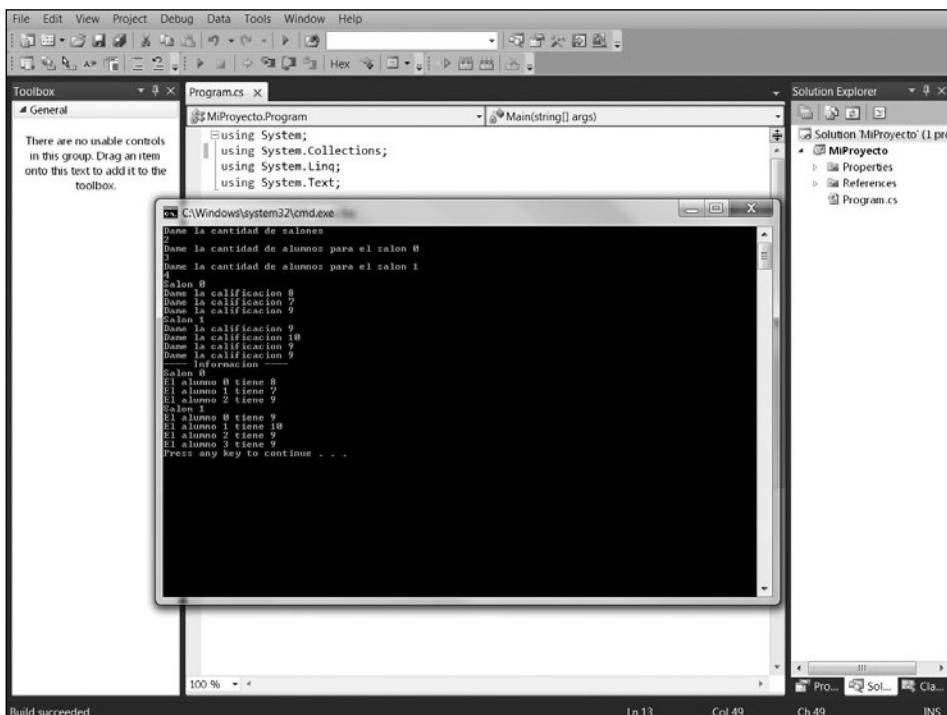


Figura 11. Podemos observar que efectivamente cada salón tiene un número diferente de alumnos.

Para verificar que efectivamente podemos crear columnas con diferentes tamaños cada vez, ejecutemos el programa de nuevo con otros valores.

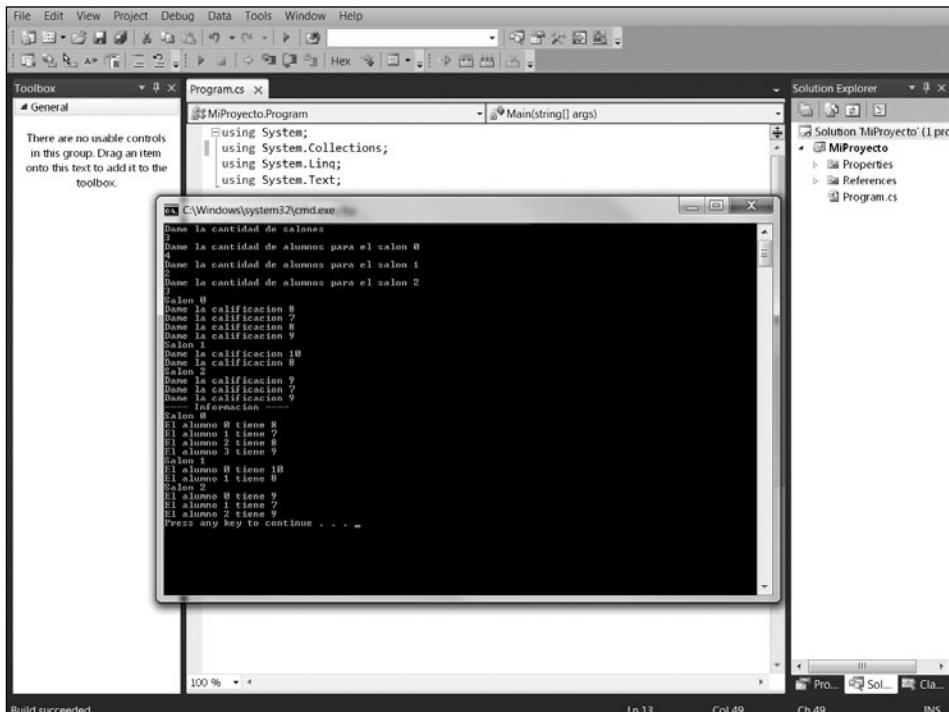


Figura 12. En esta ocasión hemos creado un arreglo jagged de diferente tamaño.

Los arreglos como parámetros a funciones

Aprendimos a usar los arreglos. Hasta el momento han sido utilizados adentro de la función **Main()**. A medida que nuestros programas sean más grandes y especializados, tendremos que usar funciones y éstas necesitarán procesar la información contenida en los arreglos. Al igual que con las variables, es posible pasar un arreglo como parámetro. De esta forma podremos encargarnos de enviar la información desde el arreglo hacia la función correspondiente.

III LOS ARREGLOS COMO PARÁMETROS

Cuando enviamos un arreglo como parámetro a una función, contemplemos los siguientes factores. El primero es que el arreglo enviado y el arreglo del parámetro deben ser el mismo tipo de dato, el segundo es que deben tener las mismas dimensiones, es decir ambos son como lista o como matriz, y el último es que ambos deben ser del mismo estilo: normal o jagged.

Nuestra función debe estar definida de forma similar, como se muestra a continuación:

```
static void Imprime(int[] arreglo)
{
    ...
}
```

Como podemos ver, en la lista de parámetros indicamos que el parámetro es un arreglo al colocar `[]` y es de tipo `int`. El nombre del parámetro en este caso es `arreglo` y lo podremos utilizar internamente dentro de la función como variable local.

La invocación se llevaría a cabo de la siguiente forma:

```
int[] numeros = new int[5];
...
...
Imprime(numeros);
```

Vemos que tenemos un arreglo de enteros que se llama `números`. En el momento de invocar la función lo hacemos por medio del nombre de la función y colocamos como parámetro solamente el nombre del arreglo, es decir, pasamos el arreglo pero sin colocarle `[]`. Esto se debe a que pasamos un arreglo completo y no solamente un elemento que se encuentra adentro del arreglo. Ahora veremos un ejemplo donde podamos hacer uso de esto. En la función `Main()` crearemos un arreglo y capturaremos los datos necesarios. Luego tendremos una función especializada en imprimir los contenidos del arreglo.

Nuestro código queda de la siguiente forma:

```
using System;
using System.Collections.Generic;
using System.Text;

namespace AplicacionBase
{
    class Program
    {
```

```
// Esta es la función principal del programa
// Aquí inicia la aplicación
static void Main(string[] args)
{
    // Variables necesarias
    int[] numeros = new int[5];
    int n = 0;
    string valor = "";

    // Pedimos los números
    for (n = 0; n < 5; n++)
    {
        Console.Write("Dame un número ");
        valor = Console.ReadLine();
        numeros[n] = Convert.ToInt32(valor);
    }

    // Invocamos a la función
    Imprime(numeros);
}

// Esta es la función de impresión
static void Imprime(int[] arreglo)
{
    int n = 0;

    for (n = 0; n < 5; n++)
    {
        Console.WriteLine("El número es {0}", arreglo[n]);
    }
}
```

Vemos que es muy sencillo llevar a cabo el paso del arreglo como parámetro y la función **Imprime()** puede usar la información que contiene sin problemas.

Si compilamos y ejecutamos la aplicación obtendremos el siguiente resultado.

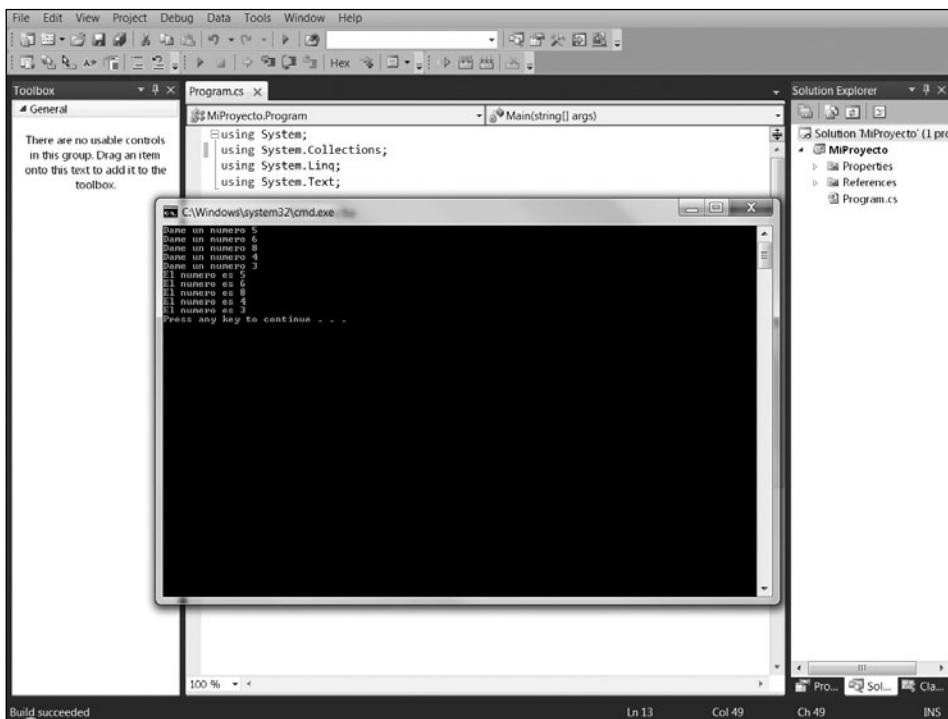


Figura 13. Podemos observar que la información capturada efectivamente es pasada a la función y ésta la utiliza.

Con esto hemos visto los puntos más importantes de los arreglos y cómo utilizarlos.

... RESUMEN

Los arreglos nos permiten guardar información adentro de un grupo de variables que son todas del mismo tipo y a las que se puede acceder por un nombre en común. Para acceder a un elemento en particular necesitamos usar su número de índice. En los arreglos el índice está basado en cero, es decir que el primer elemento se encuentra en la posición cero, a diferencia de los arreglos, que pueden tener varias dimensiones dependiendo de nuestras necesidades. Los arreglos de tipo jagged en realidad son arreglos de arreglos. Éstos nos permiten tener columnas con diferente cantidad de renglones. Al ser clases, los arreglos tienen funciones de apoyo que nos facilitan su uso y pueden ser pasados como parámetros a funciones.



ACTIVIDADES

TEST DE AUTOEVALUACIÓN

- 1** ¿Qué son los arreglos?
- 2** ¿Cómo se declara un arreglo?
- 3** ¿Qué es el índice cero?
- 4** ¿Cómo se accede a un arreglo?
- 5** ¿Cómo se declara un arreglo de dos dimensiones?
- 6** ¿Qué relación hay entre la cantidad de índices y la cantidad de dimensiones?
- 7** ¿Qué sucede si le ponemos un índice mayor al tamaño del arreglo?
- 8** ¿Qué es un arreglo de tipo jagged?
- 9** ¿Cuántas formas de declarar un arreglo jagged existen?
- 10** ¿Cómo declaramos una función para que reciba un arreglo?
- 11** ¿Cómo invocamos una función para que procese un arreglo?
- 12** ¿Qué sucede si intento indexar un arreglo jagged con la sintaxis de un arreglo tradicional?

EJERCICIOS PRÁCTICOS

- 1** Agregar el cálculo del promedio para el programa con arreglo jagged.
- 2** Agregar el cálculo de la menor calificación para el programa con arreglo jagged.
- 3** Agregar el cálculo de la mayor calificación para el programa con arreglo jagged.
- 4** Modificar el programa de la escuela para que utilice funciones.
- 5** Crear un programa que pase un arreglo jagged como parámetro a una función.

Las colecciones

Hemos aprendido a utilizar los arreglos para guardar grupos de información y trabajar más fácilmente. Si bien los arreglos son muy útiles y sencillos de usar, en algunas ocasiones están limitados en su funcionamiento y son poco flexibles para guardar grupos de datos dinámicos. En este capítulo aprenderemos a utilizar las colecciones. Éstas nos brindan muchos de los beneficios de los arreglos y nos dan la flexibilidad necesaria para guardar datos dinámicos.

Las colecciones más importantes	218
El ArrayList	218
El Stack	232
El Queue	241
El Hashtable	249
Resumen	253
Actividades	254

LAS COLECCIONES MÁS IMPORTANTES

Las colecciones son **estructuras de datos** que nos permiten guardar en su interior cualquier tipo de información. Existen diferentes tipos de colecciones y la forma como se guarda, se accede y se elimina la información en cada una de ellas es distinta. En los arreglos nosotros teníamos que indicar la cantidad de elementos que el arreglo debía tener. En las colecciones esto no es necesario, ya que es posible agregar elementos **dinámicamente**. Esto quiere decir que cuando el programa se está ejecutando podemos adicionar o borrar sus elementos.

En otros lenguajes de programación cuando se desea tener este tipo de estructuras generalmente es necesario programarlas antes de poder usarlas. Sin embargo C# nos provee de las colecciones más importantes y podemos utilizarlas directamente sin tener que hacer ningún tipo de desarrollo previo.

Las colecciones que aprenderemos en este capítulo son: **ArrayList**, **Hashtable**, **Queue**, y **Stack**. También aprenderemos un nuevo tipo de ciclo que nos facilitará la utilización de estas colecciones.

El ArrayList

La primera colección que aprenderemos se conoce como **ArrayList**, que guarda la información como si fuera una lista. Y sobre esta lista es posible realizar diferentes actividades con los elementos almacenados. Entendemos al **ArrayList** como un arreglo que puede cambiar su tamaño según lo necesitemos.

Puede guardar cualquier tipo de dato, por lo que lo podemos usar para enteros, cadenas, flotantes o incluso para tipos definidos por nosotros mismos. **ArrayList** es una clase en C#, por lo que va a tener métodos o funciones que nos permitirán trabajar con los datos.

El **ArrayList** tiene una propiedad que llamamos **capacidad**, que indica el tamaño que ocupa la lista. También tenemos el **conteo**, el cual nos dice cuántos elementos está guardando en su interior.

Para entender cómo funciona **ArrayList** crearemos una pequeña aplicación y en ella realizaremos las operaciones más importantes.

III EL NAMESPACE DE LAS COLECCIONES

Para poder utilizar las colecciones necesitamos que nuestro programa utilice el namespace de **System.Collections**. Sin este namespace las colecciones no serán reconocidas y tendremos problemas al compilar el programa. Hay que recordar que los namespace a utilizar se colocan al inicio del programa antes de nuestro código.

Declaración de un ArrayList

En nuestro programa podemos tener tantos **ArrayList** como sean necesarios, pero es necesario declararlos primero. La declaración se lleva a cabo de la siguiente manera:

```
ArrayList datos = new ArrayList();
```

Lo primero que necesitamos es indicar **ArrayList**, ya que éste es el nombre de la clase. Luego colocamos el nombre que va a tener, en nuestro caso es datos. Posteriormente pasamos a la instanciación, la cual se lleva a cabo por medio de **new**. En este ejemplo el constructor de la clase no recibe ningún parámetro.

Si bien el **ArrayList** aumenta su tamaño dinámicamente, es posible instanciar el arreglo con algún valor de capacidad inicial. Esto es útil si sabemos inicialmente cuantos elementos puede contener el **ArrayList**. para hacerlo simplemente colocamos la capacidad inicial entre los paréntesis de la siguiente forma:

```
ArrayList datos = new ArrayList(32);
```

Aquí, datos tiene una capacidad inicial de 32, aunque se encuentre vacío.

Adición de información

Nosotros podemos adicionar cualquier tipo de información al **ArrayList**. Hacerlo es muy sencillo y requiere que usemos un método conocido como **Add()**.

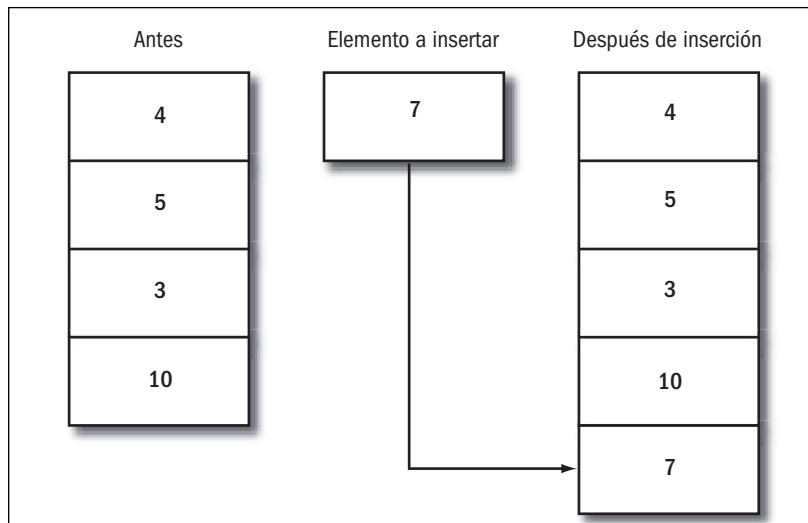


Figura 1. Aquí podemos ver cómo el nuevo elemento es colocado al final del *ArrayList* existente.

Siempre que se adiciona un elemento al **ArrayList**, este nuevo elemento se agrega al final. Si incorporamos otro elemento, se colocará después del anterior. La adición siempre se lleva a cabo después del último elemento que se encuentre en el **ArrayList**.

El método **Add()** va a necesitar de un único parámetro y este es el dato que queremos guardar. Por ejemplo para guardar un dato de tipo entero podemos hacer lo siguiente

```
datos.Add(7);
```

El dato a guardar también puede ser pasado por medio de una variable:

```
datos.Add(n);
```

Si quisiéramos guardar una cadena, se puede hacer de la siguiente manera:

```
palabras.Add("Hola");
```

El uso de foreach

En la programación orientada a objetos existe un patrón conocido como iterador o iterator, según su sintaxis en el idioma inglés. La principal característica del iterador es permitirnos realizar un recorrido por todos los elementos que existen en una estructura de datos. El recorrido lo hace de forma secuencial, uno por uno. Esto resulta muy útil cuando sabemos que debemos recorrer todos los elementos de la estructura, como un **ArrayList**, y hacer algo con ellos.

En C# encontramos un iterador en el **foreach**. Con él es posible que recorramos los elementos, luego ejecutamos alguna acción con ellos y finalizamos cuando la colección ya no tiene más elementos que entregarnos.

La sintaxis es la siguiente:

III LA CAPACIDAD Y EL CONTEO

En varias ocasiones vamos a encontrar que la capacidad del **ArrayList** es mayor que su conteo, pero nunca menor. La capacidad tiende a ser mayor para que las operaciones de inserción de datos sean rápidas. El arreglo guarda los datos hasta que llega a su capacidad y si la cantidad de datos pasa cierto límite, el arreglo crece en su capacidad dejando espacio libre para nuevas inserciones.

```
foreach (tipo identificador in expresión)
    sentencia
```

El **tipo** está relacionado con la clase de información que guarda la colección. Si nuestra colección guarda enteros, entonces el tipo debe de ser **int** y así sucesivamente para cada tipo de dato. Luego tenemos que tener una variable que represente al elemento que se encuentra en la lista, esta variable es el **identificador** y luego por medio de esta variable identificador podremos hacer algo con la información de ese elemento.

La **expresión** simplemente es la colección o el arreglo que vamos a recorrer con el ciclo. La **sentencia** es el código que se va a repetir para cada vuelta del ciclo. La sentencia puede ser sencilla o se puede colocar un bloque de código en caso de ser necesario. Como ya conocemos bien los arreglos, veamos cómo podemos utilizar este ciclo para mostrar los contenidos de un arreglo. Supongamos que tenemos un arreglo de enteros que se llama **costo** y al que hemos introducido valores. Para mostrarlo con este ciclo, haremos lo siguiente:

```
foreach(int valor in costo)
{
    Console.WriteLine("El valor es {0}", valor);
}
```

En este caso estamos indicando que el tipo de dato con el que vamos a trabajar es entero. La variable **valor** cambiará su contenido dependiendo del elemento del arreglo que estemos recorriendo en ese momento. Luego simplemente indicamos que el arreglo a recorrer es **costo**. Adentro del ciclo mostramos un mensaje indicando el contenido de **valor** para esa vuelta del ciclo.

Ahora es posible continuar nuestro aprendizaje sobre C#, más específicamente seguiremos revisando las características de las colecciones.



EL ÍNDICE DEL NUEVO ELEMENTO

El método **Add()** regresa un valor de tipo entero. Este valor indica el índice donde fue guardado un nuevo elemento. Muchas veces ignoramos este valor, pero en algunos casos puede resultarnos útil. Hay que recordar que el índice en el cual se encuentra un elemento puede variar, ya que es posible eliminar elementos del **ArrayList** también.

Cómo acceder a la información de un ArrayList

La colección **ArrayList** nos permite acceder a sus elementos por medio de un índice, algunas colecciones no lo permiten, pero **ArrayList** sí. Esto es bueno, pues podemos trabajarla de forma similar a un arreglo, pero con la flexibilidad de ser dinámica.

Si tenemos un **ArrayList** llamado datos y deseamos imprimir el elemento **2**, lo podemos hacer de la siguiente manera:

```
Console.WriteLine("El dato es {0}", datos[2]);
```

De igual forma podemos utilizar el valor del elemento en una expresión:

```
impuesto = datos[2] * 0.15f;
```

Y podemos también asignar un valor determinado:

```
datos[2] = 5;
```

Cómo obtener la cantidad de elementos en un ArrayList

En muchas ocasiones es útil saber cuántos elementos tenemos en el **ArrayList**. A diferencia del arreglo tradicional, éste puede cambiar su tamaño durante la ejecución de la aplicación y conocer el tamaño nos evita problemas con los valores de los índices al acceder el arreglo.

Afortunadamente es muy sencillo hacerlo, simplemente tenemos que leer el valor de la propiedad **count** del **ArrayList**. Esta propiedad es un valor entero.

```
elementos = datos.Count;
```

En este caso la variable **elementos** tendrá la cantidad de elementos en el arreglo **datos**. No debemos olvidar que si **elementos** tiene el valor de **5**, los índices irán de **0** a **4**.

Insertar elementos

Hemos visto que podemos adicionar elementos y que éstos son colocados al final del **ArrayList**, sin embargo también es posible llevar a cabo una inserción. La inserción permite colocar un elemento nuevo en cualquier posición válida del arreglo. Para lograr esto usamos el método **Insert()**. Este método necesita de dos parámetros, el primer parámetro es el índice donde deseamos insertar el elemento y

el segundo parámetro es el elemento a insertar. La figura que se muestra a continuación, se encarga de mostrar cómo trabaja la inserción.

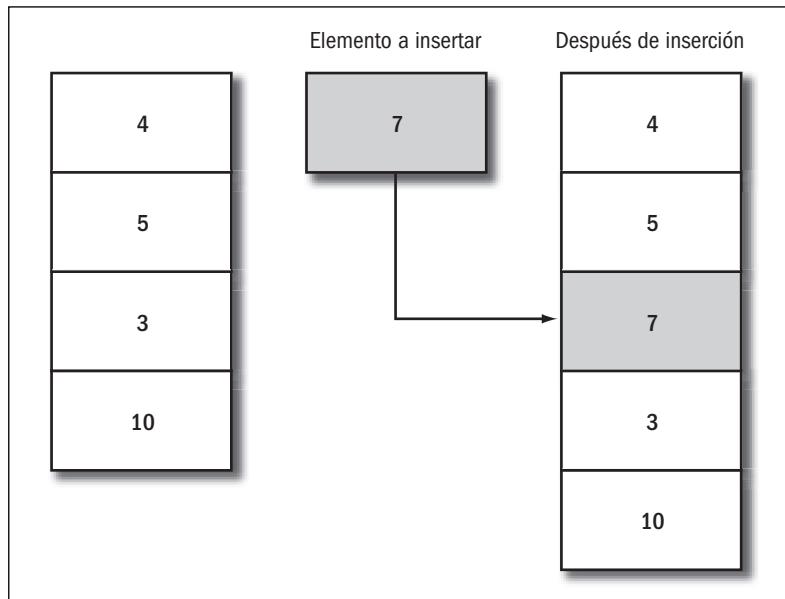


Figura 2. Podemos observar cómo se lleva a cabo la inserción del elemento, debemos notar cómo algunos elementos cambian de índice.

Por ejemplo, si deseamos insertar el valor de **5** en el índice **2**, hacemos lo siguiente:

```
datos.Insert(2, 5);
```

El **ArrayList** crecerá su tamaño según sea necesario.

Para eliminar un elemento

Es posible eliminar cualquier elemento del **ArrayList** y hacerlo de forma muy sencilla. Lo único que necesitamos es conocer el índice del elemento a eliminar. El

III LA CAPACIDAD DEL ARRAYLIST

Si en alguna de nuestras aplicaciones necesitáramos saber cuál es la capacidad del **ArrayList**, es posible hacerlo al leer el valor de propiedad **Capacity**. Esta propiedad es de tipo entero. Si lo que deseamos es reducir la capacidad del **ArrayList**, se puede usar el método **TrimToSize()**, pero debemos tener cuidado para no perder información útil.

índice es un valor entero y debe ser válido. El método que se encarga de llevar a cabo la eliminación es **RemoveAt()**.

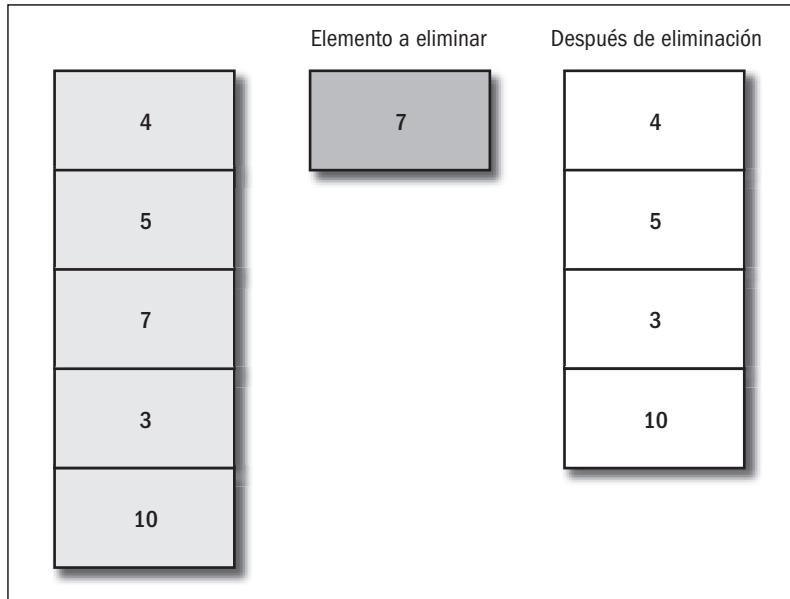


Figura 3. Al eliminar el elemento, desaparece del *ArrayList*. Los demás elementos se reorganizan y sus índices pueden modificarse.

Este método solamente necesita de un parámetro, que es el índice del objeto que deseamos eliminar. Por ejemplo si queremos eliminar el elemento que se encuentra en el índice 7, podemos hacer lo siguiente:

```
datos.RemoveAt(7);
```

Para encontrar un elemento

Con los **ArrayList** es posible saber si un elemento en particular se encuentra adentro de él. Para lograr esto hacemos uso del método **IndexOf()**. Este método requiere de un solo parámetro que es el objeto a buscar adentro del **ArrayList**. El método nos regresa un valor entero.

Este valor es el índice donde se encuentra la primera ocurrencia del elemento, esto es debido a que podemos tener el elemento guardado en diferentes posiciones. Si el elemento no se encuentra en el **ArrayList**, entonces simplemente recibimos el valor de **-1**.

Si lo que deseamos es buscar el índice donde se encuentra el elemento **7** en nuestro **ArrayList**, hacemos lo siguiente:

```
índice = datos.IndexOf(7);
```

Ahora la variable índice tiene la locación donde se encuentra el elemento 7.

Ejemplo con ArrayList

Ahora podemos construir un ejemplo que use el **ArrayList** y los diferentes métodos que es posible usar con él. Empecemos por declarar nuestras variables y colocar lo necesario para crear el **ArrayList** y adicionar unos datos inicialmente.

```
static void Main(string[] args)
{
    // Variables necesarias
    int indice=0;
    int cantidad=0;

    // Declaramos el ArrayList
    ArrayList datos = new ArrayList();

    // Añadimos valores al ArrayList
    datos.Add(7);
    datos.Add(5);
    datos.Add(1);

    Console.WriteLine("Tenemos inicialmente los
                      datos:");
    Imprime(datos);
}
```

Para que nuestra aplicación muestre el arreglo crearemos una pequeña función llamada **Imprime**. Esta función recibe el **ArrayList** a imprimir y hace uso del **foreach** para mostrar cada elemento del **ArrayList**.

```
static void Imprime(ArrayList arreglo)
{
    foreach (int n in arreglo)
```

```
Console.WriteLine(" {0},",n);
```

```
Console.WriteLine("\n-----");
```

```
}
```

Si ejecutamos la aplicación obtendremos lo siguiente:

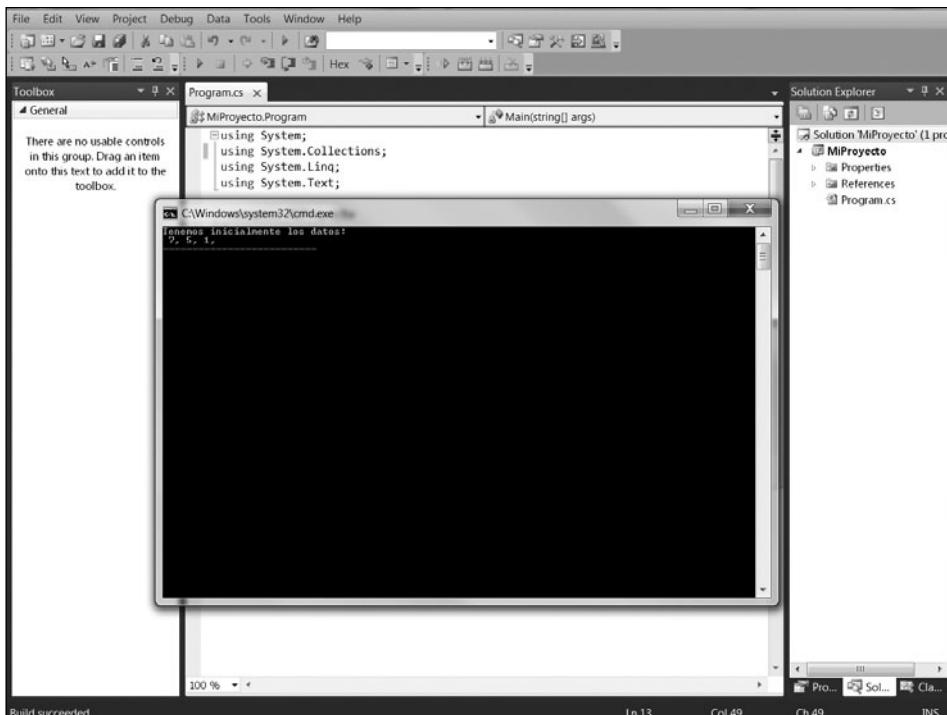


Figura 4. Podemos observar cómo se han mostrado los elementos del ArrayList.

Vamos a experimentar un poco y haremos crecer al arreglo insertando tres nuevos valores. Para el último valor obtendremos el índice donde fue colocado.

```
// Hacemos crecer el ArrayList
datos.Add(4);
datos.Add(5);

// Obtenemos el indice
indice=datos.Add(10);

Console.WriteLine("Despues de hacerlo crecer:");
```

```

    Imprime(datos);
    Console.WriteLine("El ultimo elemento tiene el
                      indice {0}",indice);
    Console.WriteLine("\n-----");

```

Ejecutemos el programa y veremos el siguiente resultado.

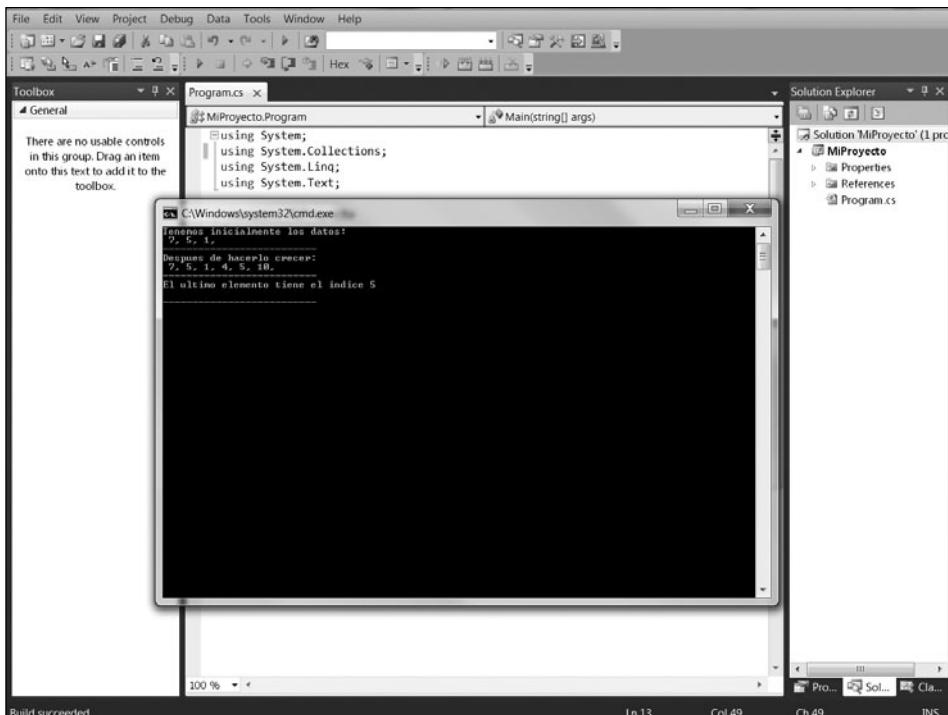


Figura 5. Podemos verificar que el *ArrayList* ha crecido y vemos cómo hemos obtenido el índice de la última inserción.

Ahora utilizaremos el índice para leer el valor de un elemento y modificar el valor de otro. En este caso la sintaxis es muy similar a la de los arreglos.

```

    // Imprimimos un elemento en particular
    Console.WriteLine("El valor en el indice 2 es
                      {0}",datos[2]);
    Console.WriteLine("\n-----");

    // Modificamos un dato
    datos[3]=55;

```

```
Console.WriteLine("Despues de la modificacion:");
Imprime(datos);
```

Si compilamos este ejemplo, podremos observar cómo se llevó a cabo el cambio del elemento adentro del arreglo correspondiente.

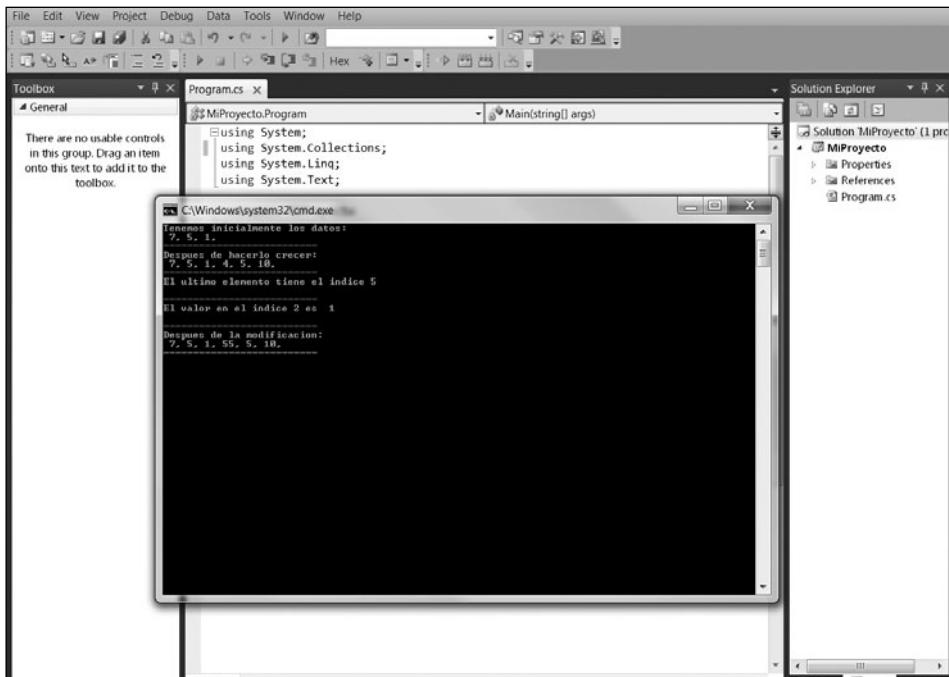


Figura 6. Hemos obtenido el contenido de un elemento y modificado otro.

En este momento obtendremos la cantidad de elementos que contiene el arreglo y la desplegaremos en la pantalla.

```
// Obtenemos la cantidad
```

III PROBLEMAS CON LOS KEY DE LAS HASHTABLE

Cuando hacemos uso de las **Hashtable**, cada elemento que coloquemos debe de tener un **key** único. El **key** no se debe repetir, si lo hacemos podemos experimentar problemas con nuestro programa. La razón de esto es que el **key** es usado para encontrar la posición en el **Hashtable** del elemento y una repetición crearía conflicto al tener dos elementos diferentes en la misma posición.

```

cantidad=datos.Count;
Console.WriteLine("La cantidad de elementos es
{0}",cantidad);
Console.WriteLine("\n-----");

```

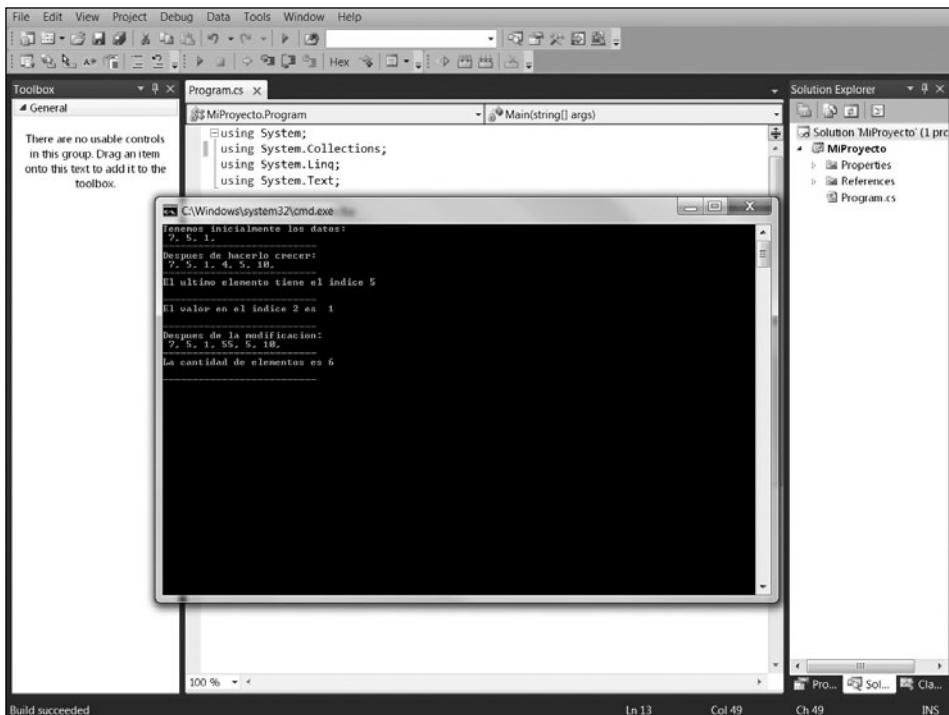


Figura 7. Hemos obtenido la cantidad de elementos en el **ArrayList**.

Si deseamos insertar un elemento en alguna posición en particular es posible hacerlo.

```

// Insertamos un elemento
datos.Insert(2,88);

```

III CUIDADO CON EL RANGO

Al igual que con los arreglos tradicionales, a **ArrayList** le debemos tener cuidado cuando lo accedamos por medio de un índice ya que si intentamos acceder a un elemento con un índice inexistente obtendremos un error al ejecutar el programa. No debemos usar índices negativos ni mayores a la cantidad de elementos que se tienen. Los índices también están basados en cero.

```
Console.WriteLine("Despues de la insercion:");
Imprime(datos);
```

Al ejecutar el programa podemos ver que fue insertado en el lugar de adicionado.

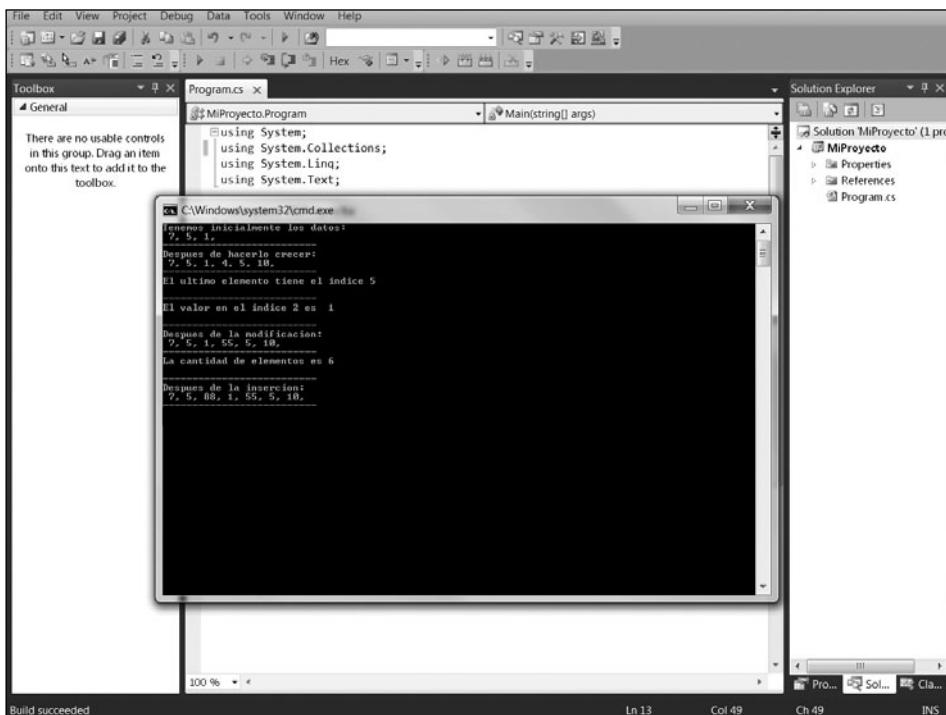


Figura 8. La inserción puede llevarse a cabo en cualquier lugar del *ArrayList*, la adición solo al final.

Si lo que necesitamos es eliminar un elemento lo hacemos de la siguiente manera:

```
// Eliminamos un elemento
```

III OTRA FORMA DE TOMAR EL ELEMENTO

El método **Pop()** tomó el elemento que se encuentra en la parte superior del **Stack** y lo regresa al exterior, pero el elemento ya no pertenece al **Stack**. Hay otro método conocido como **Peek()** que lee el valor del elemento en la parte superior del **Stack** y lo regresa al exterior, pero no elimina el elemento del **Stack**.

```
datos.RemoveAt(4);
Console.WriteLine("Despues de la eliminacion:");
Imprime(datos);
```

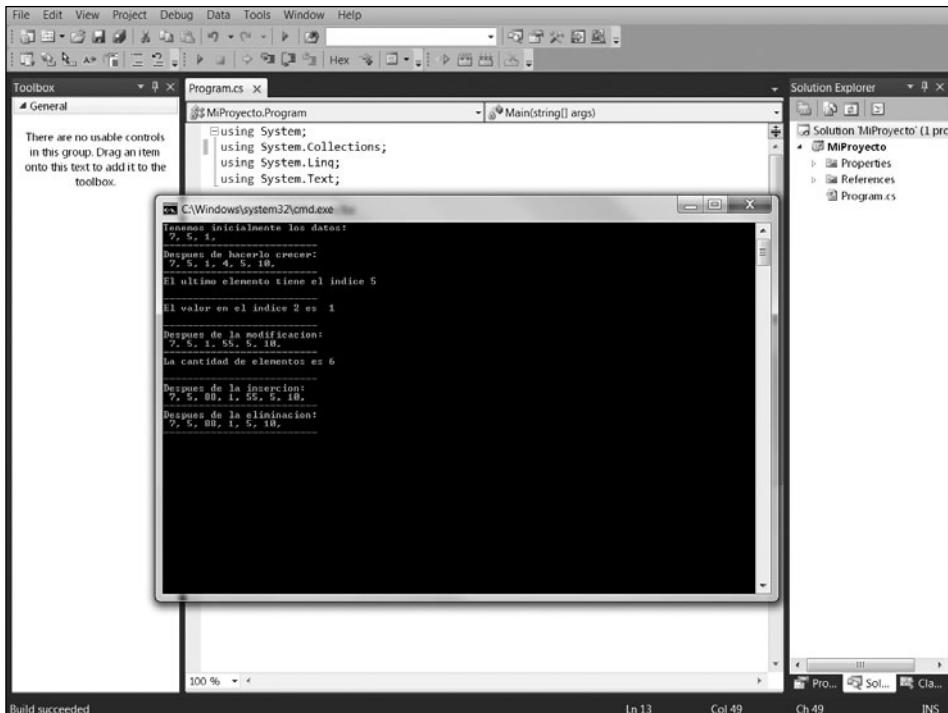


Figura 9. El elemento ha desaparecido del *ArrayList*.

Lo último que nos falta en la generación de nuestra aplicación, es dotarla de la capacidad de encontrar el índice donde se encuentra un elemento en particular. Para ello debemos agregar el código que se muestra a continuación.

```
// Encontramos el indice donde se encuentra el
// primer 5
indice=datos.IndexOf(5);
Console.WriteLine("El primer 5 se encuentra en
{0}",indice);
Console.WriteLine("\n-----");
```

Luego de agregar el trozo de código propuesto en el bloque anterior, nuestra aplicación será capaz de encontrar el índice; y como ya se halla completa lucirá similar al ejemplo que podemos ver en la imagen mostrada a continuación:

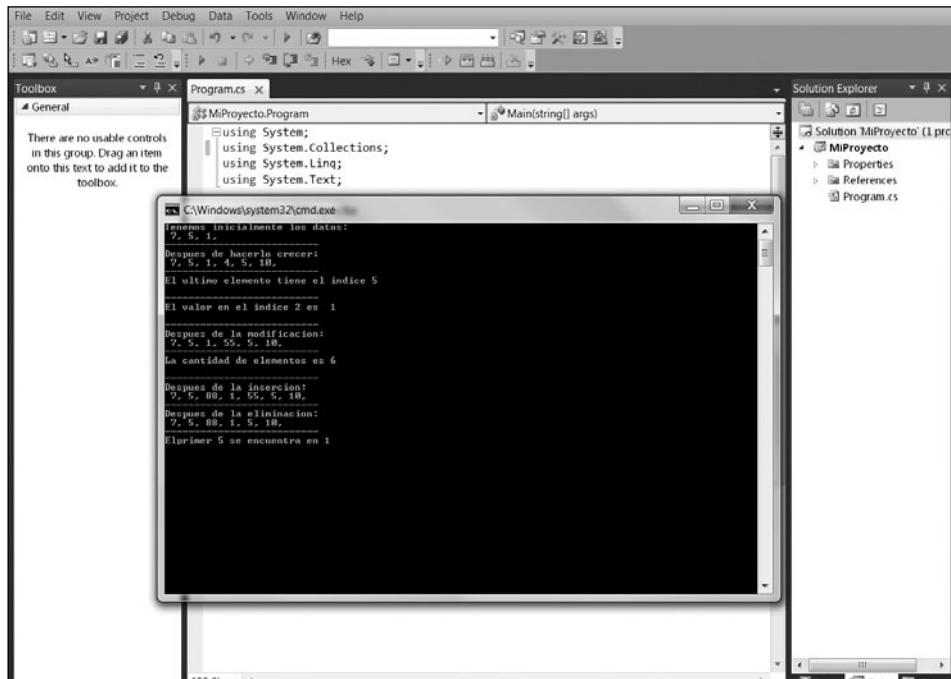


Figura 10. La primera ocurrencia del valor de 5 es en el índice 1.

El Stack

Ahora empezaremos a conocer otro tipo de colección. A esta colección se la conoce como **Stack** o **pila**, nos permite guardar elementos y cambia su tamaño de forma dinámica, sin embargo trabaja en forma diferente al arreglo y al **ArrayList**.

El **Stack** es una estructura de tipo **LIFO**. LIFO es el acrónimo en inglés para *Last-in-first-out*, es decir el primero que entra, el último que sale. Para entender su funcionamiento podemos imaginar una pila de platos. El primer plato que colocamos queda hasta la base, el siguiente se colocará encima y así sucesivamente. Como el primer plato queda hasta abajo, no lo podemos sacar directamente pues la pila se derrumbaría. Al sacar los platos, debemos tomar el que se encuentre hasta arriba de la pila primero y así continuar.

III LA CAPACIDAD DEL STACK

El **Stack** al igual que el **ArrayList** tiene una capacidad y ésta crece dinámicamente. Cuando instanciamos el **Stack** adquiere la capacidad de default. Si necesitamos crear un **Stack** con determinada capacidad, la forma de hacerlo es colocando el valor de capacidad entre los paréntesis del constructor al momento de instanciarlo.

El efecto de colocar nuevos elementos en la parte superior del **Stack** se conoce como **Push**. Esto se muestra en la siguiente figura:

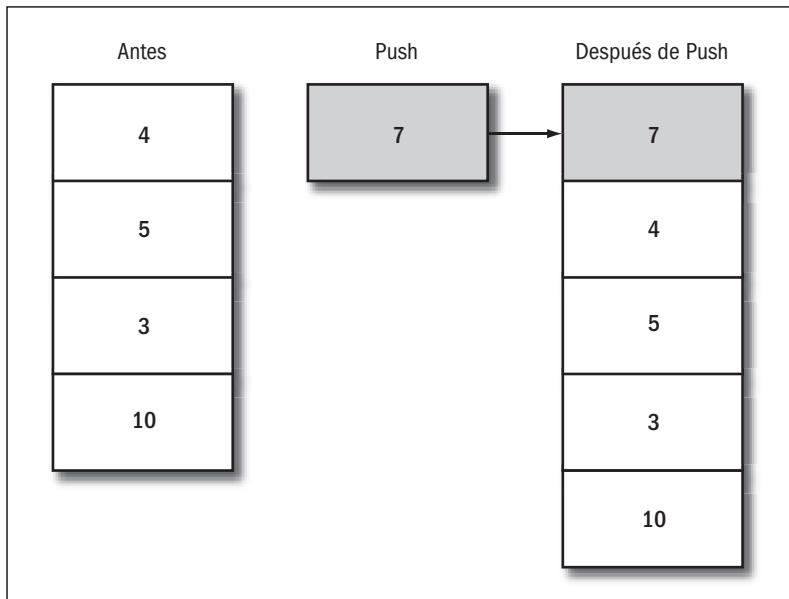


Figura 11. Podemos observar cómo **Push** inserta nuevos elementos en la parte superior.

Cuando tomamos un elemento de la parte superior del **Stack** se conoce como **Pop**. En la figura podemos observar cómo esto sucede.

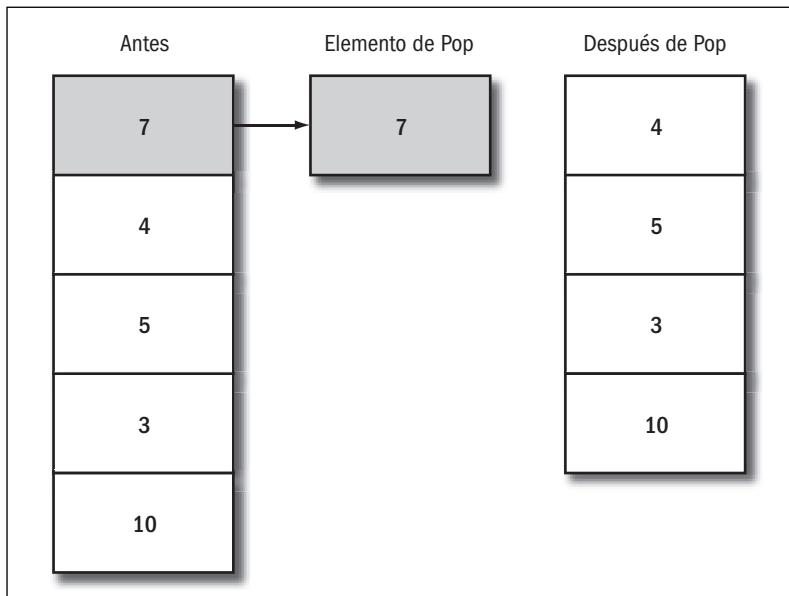


Figura 12. Pop toma el elemento que se encuentra en la parte superior del Stack.

Debido a este comportamiento con **Push** y **Pop** podemos entender cómo el primer objeto que se coloca adentro del Stack es el último en poder salir.

Como crear el Stack

Como cualquier otra colección el **Stack** necesita ser instanciado para poderlo utilizar. C# nos provee de la clase **Stack** y adentro de esta clase encontramos todos los métodos necesarios para trabajar con él. La instanciación simplemente será la creación de un objeto de esta clase. Si deseamos crear un **Stack** hacemos lo siguiente:

```
Stack miPila = new Stack();
```

Hemos creado un **Stack** llamado **miPila**. Nosotros podemos usar cualquier nombre que sea válido. Una vez instanciado podemos empezar a colocar información en él.

Cómo introducir información al Stack

Para introducir información al **Stack** usamos el método **Push()**. Este método coloca el nuevo elemento en la parte superior del **Stack**. El método necesita únicamente de un parámetro que es el elemento que deseamos insertar. Podemos utilizar el método **Push()** cuantas veces sea necesario para colocar la información.

```
miPila.Push(7);
miPila.Push(11);
miPila.Push(8);
```

En este ejemplo se introduce primero el elemento **7** y luego sobre él se coloca el elemento **11**. En seguida se coloca un nuevo elemento en la parte superior, que en este caso es **8**. El Stack resultante se muestra en la siguiente figura:

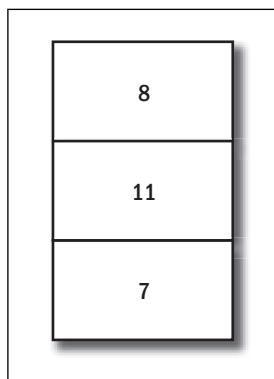


Figura 13. El Stack resultante nos muestra como el elemento 7 ha quedado en la base.

Cómo obtener información del Stack

Si lo que necesitamos es obtener la información que está contenida en el **Stack** lo podemos hacer al tomar el elemento que se encuentra en la parte superior del **Stack**. Para lograr esto hacemos uso del método **Pop()**. Este método no necesita ningún parámetro y regresa el elemento correspondiente.

Por ejemplo, si deseamos tomar el elemento de nuestro **Stack**:

```
int valor = 0;
...
valor = (int)miPila.Pop();
```

Siguiendo el ejemplo anterior la variable **valor** ahora tendrá en su interior **8**. Esto lo vemos en la siguiente figura:

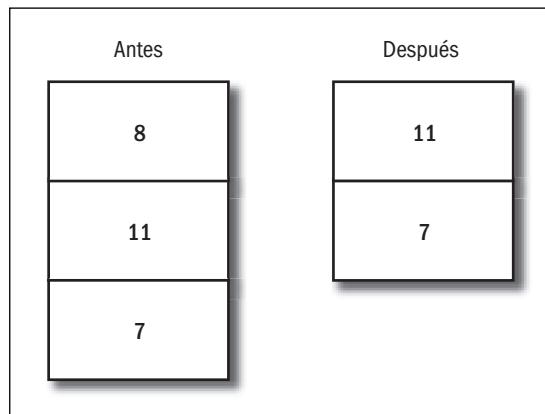


Figura 14. Se ha tomado el elemento de la parte superior del **Stack**. Hay que notar que el elemento no continúa siendo parte de él.

Uso de **foreach** para recorrer el **Stack**

Si necesitamos recorrer el **Stack**, es posible hacerlo por medio del uso de **foreach**. El uso de esta alternativa es similar al del **ArrayList**.

III EL CASTING

Existe una clase a partir de la cual descienden todas las demás clases en C#, esta clase se conoce como **Object**. Muchos métodos regresan objetos de tipo **Object**, por lo que es necesario indicar el tipo correcto bajo el cual debemos de usarlo. Para hacer esto hacemos uso del casting. Por ejemplo para hacer un casting a entero hacemos lo siguiente: **Valor = (int)fila.Dequeue();**

```
foreach( int n in miPila)
    Console.WriteLine("{0}",n);
```

Podemos usar el **foreach**, generalmente recorreremos el **Stack** por medio de **Pop()**.

Para obtener la cantidad de elementos del Stack

Es posible conocer la cantidad de elementos que tiene el **Stack** y hacerlo es muy sencillo. Debemos leer la propiedad **Count** del **Stack**. Esta propiedad nos regresa un valor entero con la cantidad de elementos. El uso es equivalente al del **ArrayList**.

```
cantidad = miPila.Count;
```

Para limpiar los contenidos del Stack

Si deseamos eliminar todos los elementos del **Stack** de forma rápida lo podemos hacer al usar el método **Clear()**. Este método no necesita de ningún parámetro y solamente debe ser usado cuando sepamos que debemos borrar los elementos del **Stack**.

```
miPila.Clear();
```

Para saber si el Stack tiene un elemento

Si deseamos saber si adentro del **Stack** se encuentra un elemento en particular, podemos hacer uso del método **Contains()**. Este método necesita de un parámetro que es el objeto a encontrar adentro del **Stack**. El método regresa un valor de tipo **bool**. Si el objeto se encuentra el valor es **true**, pero si no se encuentra es **false**.

```
bool enStack = false;
...
enStack = miPila.Contains(7);
```

Creación de una aplicación

Podemos hacer un programa que muestre como usar los métodos del **Stack**. Para esto crearemos una aplicación que presente las operaciones del **Stack**. El contenido se mostrará para ver los cambios que se tienen debidos al **Push** y al **Pop**.

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

```
// Variables necesarias

int opcion=0;
string valor="";
int numero=0;
bool encontrado=false;

// Creamos el stack

Stack miPila= new Stack();

do
{
    // Mostramos menu
    Console.WriteLine("1- Push");
    Console.WriteLine("2- Pop");
    Console.WriteLine("3- Clear");
    Console.WriteLine("4- Contains");
    Console.WriteLine("5- Salir");
    Console.WriteLine("Dame tu opcion");
    valor=Console.ReadLine();
    opcion=Convert.ToInt32(valor);

    if(opcion==1)
    {
        // Pedimos el valor a introducir
        Console.WriteLine("Dame el valor a
                           introducir");
        valor=Console.ReadLine();
        numero=Convert.ToInt32(valor);

        // Adicionamos el valor en el stack
        miPila.Push(numero);
    }

    if(opcion==2)
    {

        // Obtnemos el elemento
        numero=(int)miPila.Pop();
```

```
// Mostramos el elemento
Console.WriteLine("El valor obtenido
es: {0}",numero);
}

if(opcion==3)
{
    // Limpiamos todos los contenidos
    // del stack
    miPila.Clear();
}

if(opcion==4)
{
    // Pedimos el valor a encontrar
    Console.WriteLine("Dame el valor a
    encontrar");
    valor=Console.ReadLine();
    numero=Convert.ToInt32(valor);

    // Vemos si el elemento esta
    encontrado=miPila.Contains(numero);

    // Mostramos el resultado
    Console.WriteLine("Encontrado -
{0}",encontrado);
}

// Mostramos la informacion del stack
Console.WriteLine("El stack tiene {0}
elementos",miPila.Count);
foreach(int n in miPila)
    Console.Write(" {0},", n);

Console.WriteLine("");
Console.WriteLine("____");

}while(opcion!=5);
}
```

```

    }
}

```

El programa es muy sencillo, en primer lugar declaramos las variables necesarias. Una variable es para la opción seleccionada por el usuario. Otra variable llamada **numero** será usada para los casos en los que tengamos que obtener o colocar un valor numérico en el **Stack**. La variable llamada **encontrado** es de tipo **bool** y tendrá el valor de **true** o **false** dependiendo si el elemento a buscar se encuentra o no en el **Stack**. Como siempre tenemos una variable de trabajo de tipo cadena que nos ayuda a obtener el valor dado por el usuario. Luego simplemente creamos un **Stack** llamado **miPila**.

Tenemos un ciclo **do** que estará repitiéndose hasta que el usuario decida finalizar el programa. En el ciclo lo primero que hacemos es mostrar el menú con las posibles acciones que podemos hacer con el **Stack**. Luego de acuerdo a cada opción manipulamos el **Stack**. Después de la opción hacemos uso del **foreach** para mostrar los contenidos del **Stack** y poder verificar los cambios realizados. También aprovechamos para mostrar la cantidad de elementos que existen en el **Stack**. Compilemos y ejecutemos la aplicación. Lo primero que haremos es insertar algunos valores en el **Stack**, para esto seleccionaremos la opción **1**. Coloquemos los valores **5, 7, 3**.

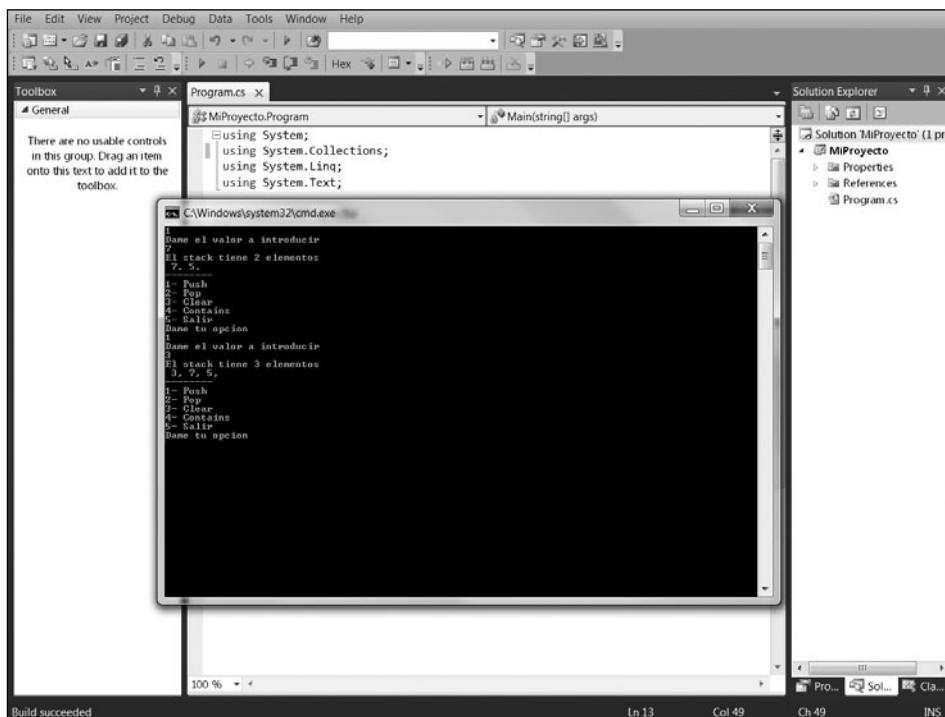


Figura 15. Podemos observar cómo el último valor introducido se encuentra al inicio del **Stack**, el primer valor se encuentra al final.

Si en este momento hacemos un **Pop**, el **Stack** debe de regresarnos el elemento que se encuentra en la parte superior del **Stack**, en nuestro ejemplo debe ser **3**.

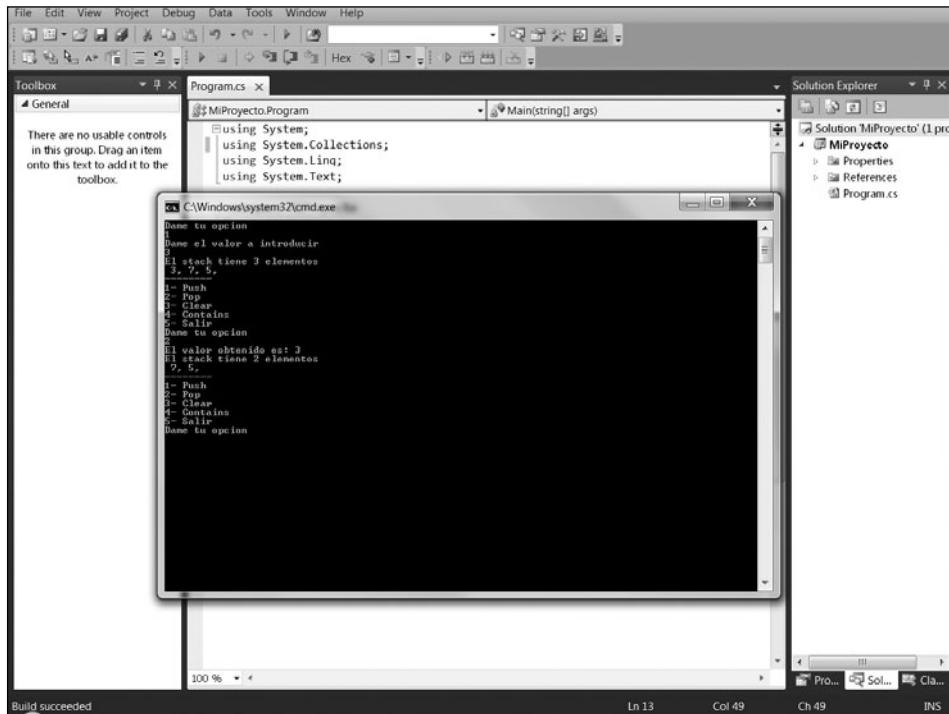


Figura 16. Vemos que el valor obtenido es **3** y el **Stack** ahora solamente tiene dos elementos.

Ahora podemos ver si el **Stack** tiene un elemento en particular. Para realizar esta operación, nos encargaremos de seleccionar la opción número **4** y para continuar tendremos que preguntar por el elemento número **9**.

Si ahora preguntamos por el elemento **5** obtendremos una respuesta diferente y la variable **encontrado** deberá tener el valor **true**. Lo último que nos queda por hacer es limpiar el **Stack** y para esto seleccionamos la opción **3**.

III EL CONSTRUCTOR DEL QUEUE

Nosotros podemos utilizar el constructor de default del **Queue** el cual no necesita de ningún parámetro, sin embargo, es buena idea consultar MSDN para conocer otros constructores. Por ejemplo tenemos una versión que nos permite colocar la capacidad inicial del **Queue** y otros que nos pueden resultar útiles en nuestras aplicaciones.

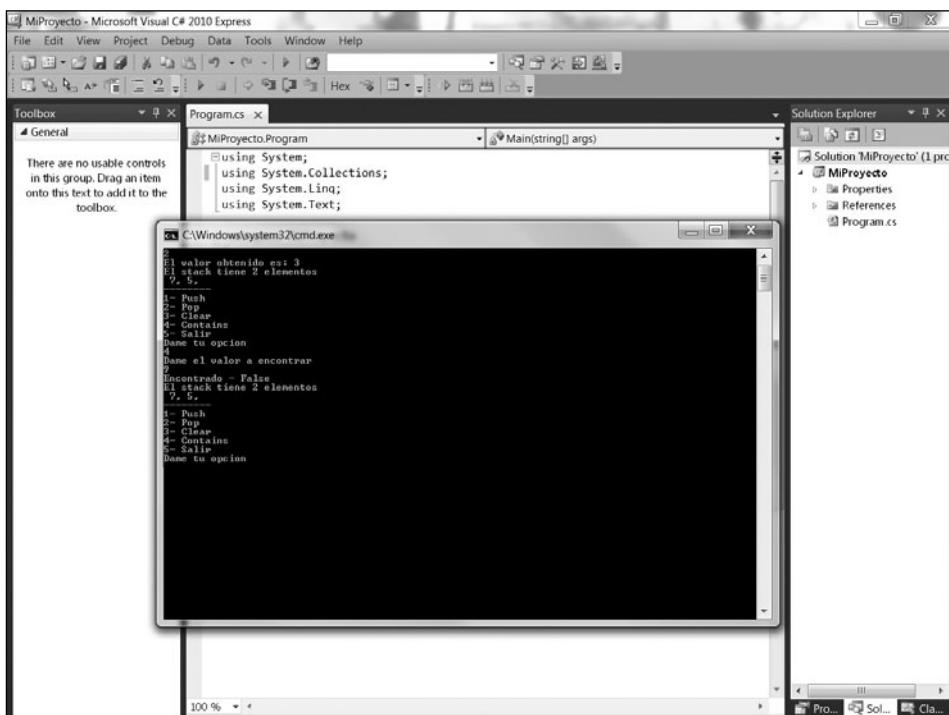


Figura 17. El **Stack** nos regresa el valor de *false* en la variable *encontrado* pues el elemento no se halla en el **Stack**.

El Queue

La siguiente colección que aprenderemos se conoce como **Queue**, algunas veces también denominada **cola**. Al igual que en el **Stack**, nosotros ya no necesitamos programarla, pues C# nos provee una clase con toda la funcionalidad necesaria para poder utilizarla. La clase que debemos de utilizar para poder tener una cola es **Queue**.

La forma como trabaja es diferente al **Stack**, ya que el **Queue** es una estructura de tipo **FIFO**. Su comportamiento es diferente al **Stack**. Para entender el **Queue**, podemos imaginar una fila para comprar los boletos o las entradas al cine. La primera persona en llegar se coloca junto a la taquilla y conforme van llegando otras perso-

III EL COMPORTAMIENTO DE FIFO

Cuando nosotros trabajamos con una estructura de datos que funcione bajo el esquema de **FIFO** debemos tomar en cuenta su comportamiento. **FIFO** es el acrónimo para **First In First Out**, que en español significa: Primero en Entrar, Primero en Salir. En estas estructuras el primer elemento que colocamos, es el primer elemento que podemos extraer.

nas se colocan atrás de ella y así sucesivamente. Sin embargo, en cuanto se habré la taquilla, la primera persona en pasar es la primera que se encuentra en la fila. Esto nos indica que cuando insertamos elementos en el **Queue**, éstos se colocan atrás del último elemento insertado o en la parte posterior de éste. Al momento de leer elementos, se toma el que se encuentre en la parte superior del **Queue**. Esto lo podemos ver más fácilmente en la siguiente figura:

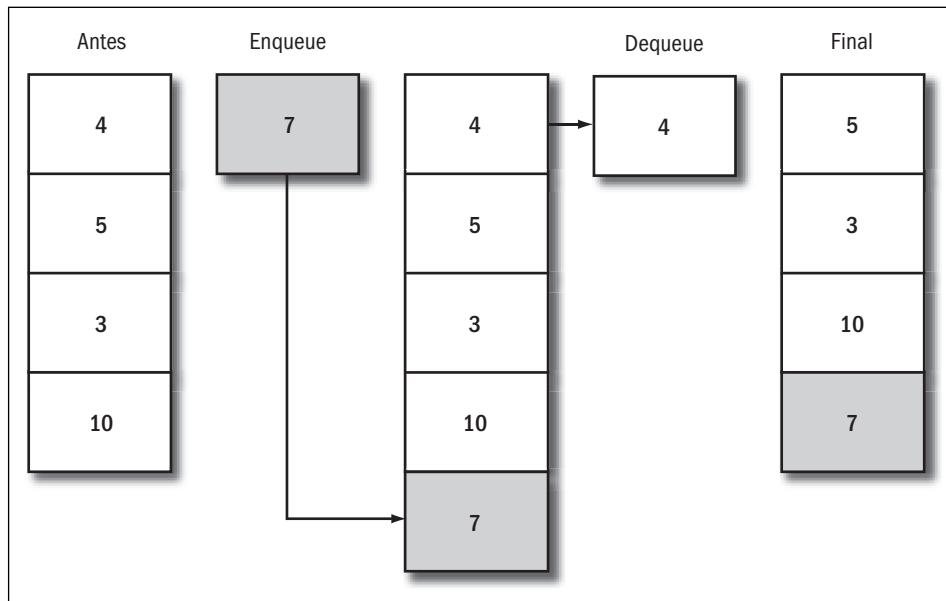


Figura 18. Aquí podemos observar el proceso de adición de un elemento y también qué es lo que sucede cuando se extrae un elemento.

El proceso por medio del cual nosotros insertamos un elemento nuevo al **Queue** se conoce como **Enqueue** y no debemos olvidar que lo hace al final. El acto de extraer un elemento del **Queue** se llama **Dequeue** y siempre toma el elemento que se encuentra en la parte superior.

Declaración del Queue

Para utilizar el **Queue**, lo primero que debemos hacer es crear una instancia de la clase **Queue**. El nombre de la instancia puede ser cualquier nombre válido de variable en C#. Si deseamos crear un **Queue** llamado fila, haremos lo siguiente:

```
Queue fila = new Queue();
```

Una vez instanciado ya es posible hacer uso de fila y también de los métodos que nos provee la clase correspondiente para trabajar con él.

Adición de elementos al Queue

Nosotros podemos adicionar elementos al **Queue** en cualquier momento que lo necesitemos. El tamaño del **Queue** se modificará dinámicamente por lo que no debemos de preocuparnos por él. Siempre que se adiciona un elemento, este elemento se coloca al final o en la parte baja del **Queue**.

Para poder hacer esto debemos utilizar el método **Enqueue()**, el cual pertenece a la clase **Queue** de C#. Este método es muy sencillo, ya que solamente requiere de un parámetro. En el parámetro colocamos el elemento que deseamos añadir, este método no regresa ningún valor.

Por ejemplo, si deseamos añadir el elemento **7** a nuestro **Queue** llamado **fila**, será necesario que escribamos el código que se muestra a continuación:

```
fila.Enqueue(7);
```

Otra alternativa posible es colocar variables en el parámetro adecuado, si realizamos esto, una copia del valor se ubicará en el **Queue**.

Cómo extraer un elemento del Queue

Al igual que podemos colocar un elemento, también es posible extraerlo. Sin embargo, la extracción se lleva a cabo de acuerdo a las reglas del **Queue**. Cuando extraemos un elemento, el elemento que recibimos es el que se encuentra en la parte superior o al inicio del **Queue**. Debemos recordar que no es posible la extracción de los elementos que se encuentren en otras posiciones.

Para llevar a cabo la extracción tenemos que usar un método de la clase **Queue** que se llama **Dequeue()**. El método no necesita de ningún parámetro y regresa el elemento correspondiente. Es importante tener una variable que reciba al elemento o una expresión que haga uso de él.

Si lo que deseamos es extraer un elemento del **Queue**, podremos agregar el código que se muestra en el siguiente bloque:



PÉRDIDA DE PRECISIÓN

Algunos tipos de datos, en especial los numéricos, pueden llegar a ser compatibles entre sí. Pero debemos tener algo en cuenta. Cuando convertimos de un tipo mayor a uno menor, por ejemplo de **double** a **float**, es posible que tengamos perdida de información. Esto se debe a que el tipo menor no puede guardar tantos dígitos como el mayor.

```
int valor = 0;
...
...
valor = fila.Dequeue();
```

Para observar un elemento del Queue

Cuando hacemos uso del método **Dequeue()**, el elemento es extraído y deja de encontrarse adentro del **Queue** después de esa operación. Sin embargo, podemos observar el elemento. En este caso recibimos el valor del elemento, pero éste no es eliminado del **Queue** después de ser leído.

Para llevar a cabo la observación hacemos uso del método **Peek()**. Este método no necesita de ningún parámetro y regresa el elemento observado. Al igual que con **Dequeue()** debe de existir una variable o una expresión que reciba este valor.

Un ejemplo de su uso puede ser:

```
int valor = 0;
...
...
valor = fila.Peek();
```

Para saber si el Queue tiene determinado elemento

Algunas veces podemos necesitar saber si en el interior del **Queue** se guarda un elemento en particular. Esto es posible hacerlo gracias a un método conocido como **Contains()**. Para usar este método, necesitamos pasar como parámetro el elemento que queremos determinar. El método regresará un valor de tipo **bool**. Si el elemento existe adentro del **Queue**, el valor regresado será **true**. En el caso de que el elemento no exista en el interior del **Queue** el valor regresado será **false**. Necesitamos tener una variable o una expresión que reciba el valor del método **Contains()**.

III EL CONSTRUCTOR DE LA CLASE

Las clases tienen un método especial llamado **constructor**. Este método es invocado automáticamente en el momento que un objeto de esa clase se instancia. Los constructores pueden o no recibir parámetros y siempre se nombran igual que la clase, se usan principalmente para inicializar correctamente al objeto.

Por ejemplo, si deseamos saber si el elemento **7** existe, podemos hacer lo siguiente:

```
if(miFila.Contains(7) == true)
    Console.WriteLine("El elemento si existe");
```

Para borrar los contenidos del Queue

Si necesitamos eliminar todos los contenidos del **Queue**, es sencillo de hacer por medio del método **Clear()**. Este método ya es conocido por nosotros de colecciones que previamente hemos estudiado. Su utilización es muy sencilla ya que no necesita de ningún parámetro. Solamente debemos tener cuidado, ya que si lo utilizamos en un momento inadecuado, perderemos información que todavía puede ser útil.

Para eliminar todos los elementos del **Queue**, colocamos lo siguiente:

```
miFila.Clear();
```

Para conocer la cantidad de elementos que tiene el Queue

Es conveniente saber la cantidad de elementos que contiene el **Queue**, especialmente antes de llevar a cabo algunas operaciones sobre éste. Cuando deseamos saber la cantidad de elementos existentes, debemos utilizar la propiedad de **Count**. Esta propiedad corresponde a un valor de tipo entero.

La podemos utilizarla de la siguiente manera:

```
int cantidad = 0;
...
...
cantidad = miFila.Count;
```

Si queremos aprovecharla para hacer una operación en particular y evitar algún error con el **Queue**, puede ser algo como lo siguiente:

```
if(miFila.Count>0)
    valor = (int)miFila.Dequeue();
```

Para recorrer los elementos del Queue

Si necesitamos recorrer los elementos del **Queue**, por ejemplo, para mostrarlos o imprimirlos lo que nos conviene usar es un ciclo **foreach**. Su utilización es muy sencilla y similar a lo que hemos usado con otras colecciones.

```
foreach( int n in miFila)
    Console.WriteLine("{0}",n);
```

Ahora podemos ver un ejemplo que haga uso del **Queue**. En este ejemplo llevaremos a cabo las operaciones más comunes sobre éste. Mostraremos al usuario un menú y dependiendo de la opción seleccionada será la operación a realizar.

```
using System;
using System.Collections;
using System.Text;

namespace AplicacionBase
{
    class Program
    {
        // Esta es la funcion principal del programa
        // Aqui inicia la aplicacion
        static void Main(string[] args)
        {

            // Variables necesarias

            int opcion=0;
            string valor="";
            int numero=0;
            bool encontrado=false;

            // Creamos el Queue

            Queue  miFila= new Queue();
```

III OPERACIONES INVÁLIDAS

Si el **Queue** se encuentra vacío, es decir, sin elementos; cuando intentemos hacer una operación como **Dequeue()** o **Peek()** se generará un error. Una forma de evitar esto, es llevar a cabo estas operaciones solamente cuando la cantidad de elementos sea mayor que cero. Un simple **if** nos puede ayudar con esto y evitar que el programa ocasione problemas.

```

do
{
    // Mostramos menu
    Console.WriteLine("1- Enqueue");
    Console.WriteLine("2- Dequeue");
    Console.WriteLine("3- Clear");
    Console.WriteLine("4- Contains");
    Console.WriteLine("5- Salir");
    Console.WriteLine("Dame tu opcion");

    valor=Console.ReadLine();
    opcion=Convert.ToInt32(valor);

    if(opcion==1)
    {
        // Pedimos el valor a introducir

        Console.WriteLine("Dame el valor a
                           introducir");
        valor=Console.ReadLine();
        numero=Convert.ToInt32(valor);

        // Adicionamos el valor en el queue
        miFila.Enqueue(numero);

    }

    if(opcion==2)
    {
        // Obtnemos el elemento
        numero=(int)miFila.Dequeue();
    }
}

```



OTRO ESTILO DE COLECCIONES

Existe otro estilo de colecciones que son funcionalmente equivalentes a las colecciones que hemos visto, pero que proveen mejor desempeño y más seguridad en el manejo de los tipos de los elementos. Estas colecciones utilizan tipos genéricos para los elementos y se encuentran en el namespace **System.Collections.Generics**

```
// Mostramos el elemento
Console.WriteLine("El valor obtenido
es: {0}",numero);
}

if(opcion==3)
{
    // Limpiamos todos los contenidos
    // del Queue
    miFila.Clear();
}

if(opcion==4)
{
    // Pedimos el valor a encontrar
    Console.WriteLine("Dame el valor a
encontrar");
    valor=Console.ReadLine();
    numero=Convert.ToInt32(valor);

    // Vemos si el elemento esta
    encontrado=miFila.Contains(numero);

    // Mostramos el resultado
    Console.WriteLine("Encontrado -
{0}",encontrado);
}

// Mostramos la informacion del stack
Console.WriteLine("El Queue tiene {0}
elementos",miFila.Count);
foreach(int n in miFila)
    Console.Write(" {0},", n);

Console.WriteLine("");
Console.WriteLine("____");

}while(opcion!=5);
```

```

    }
}

}

```

El Hashtable

La última colección que aprenderemos en este capítulo se conoce como **Hashtable**. Es una estructura de datos un poco compleja de implementar, pero afortunadamente C# nos provee de una clase que nos da toda la funcionalidad necesaria y podemos utilizarla tan fácilmente como las colecciones anteriores.

El **Hashtable** es una colección indexada. Es decir que vamos a tener un índice y un valor referenciado a ese índice. Sin embargo, la indexación no se lleva a cabo como en el arreglo o el **ArrayList**. El lugar adentro del **Hashtable** donde se coloca el elemento va a depender de un valor conocido como **key** o **llave**. El valor contenido en **key** es usado para calcular la posición del elemento en el **Hashtable**. El elemento que vamos a colocar se conoce como **value**.

En el **Hashtable** siempre utilizaremos una pareja de **key** y **value** para trabajar con él. La forma como trabaja el **Hashtable** puede parecer extraña, pero en realidad es una estructura muy eficiente al momento de leer la información. El comprender el funcionamiento interno del **Hashtable** pertenece a un libro de nivel más avanzado, por lo que no tocaremos este tema aquí.

La clase se llama **Hashtable** y pertenece al igual que las demás colecciones al namespace **System.Collections**.

Declaración del Hashtable

La declaración del **Hashtable** es muy sencilla y similar a la declaración de las otras colecciones. Para hacerlo, simplemente creamos una instancia de la clase **Hashtable** y después podemos hacer uso de las operaciones de la colección por medio de los métodos que nos provee.



OTRAS COLECCIONES

Tenemos otras colecciones que no veremos en el capítulo pero resultan interesantes y útiles de aprender. El conocer cómo trabajan otras colecciones puede ahorrarnos mucho código y tiempo ya que posiblemente existe lo que estamos tratando de hacer. Para poder conocerlas, como siempre, es recomendable visitar MSDN. Algunas de estas colecciones son: **BitArray**, **Dictionary** y **SortedList**.

Si lo que deseamos es declarar e instanciar un **Hashtable** llamado **miTabla**, es posible realizarlo prosiguiendo de la siguiente manera:

```
Hashtable miTabla = new Hashtable();
```

El nombre del **Hashtable** puede ser cualquier nombre válido de variable para C#. Ahora ya podemos aprender a utilizarlo.

Adición de elementos al **Hashtable**

Nosotros podemos adicionar cualquier cantidad de elementos al **Hashtable**. Hacerlo es tan sencillo como con las otras colecciones, pero debemos tomar en cuenta la forma cómo el **Hashtable** indexa su información.

Para insertar un elemento usamos el método **Add()**. Este método a diferencia de los de las otras colecciones, va a necesitar de dos parámetros. En el primer parámetro colocamos el **key** que será usado para indexar al elemento. Este **key** puede ser de cualquier tipo, pero generalmente se utilizan cadenas. El segundo parámetro será el valor o elemento a insertar, también puede ser de cualquier tipo. Hay que recordar que cuando hagamos uso de esta colección siempre trabajamos la información en parejas **key-value**.

Por ejemplo, podemos usar el **Hashtable** para guardar un producto con su costo. El nombre del producto puede ser el **key** y su costo será el **value**. Si deseamos insertar algunos elementos, quedará de la siguiente forma:

```
miTabla.Add("Pan", 5.77);  
miTabla.Add("Soda", 10.85);  
miTabla.Add("Atun", 15.50);
```

El lugar donde quedaron colocados estos elementos adentro del **Hashtable** depende del algoritmo de **hashing** que se use. Realmente a nosotros no nos interesa, ya que cualquier acceso que necesitemos lo haremos por medio de **key**.

III ÍNDICES VÁLIDOS

Para el **ArrayList** un índice se considera válido si no es menor que cero y es menor que la cantidad de elementos que contiene el **ArrayList**. Si se coloca un índice fuera de este rango se producirá un error. Si estos errores no se manejan adecuadamente, podemos tener pérdida de información o la finalización inesperada de la aplicación. Lo mejor es evitarlos.

Recorriendo el Hashtable

Para poder recorrer el **Hashtable**, haremos uso del ciclo **foreach**. Si queremos obtener la pareja **key-value**, nos apoyaremos en una clase conocida como **DictionaryEntry**. El diccionario también guarda parejas de datos.

Por ejemplo, si deseamos proseguir con la impresión de los contenidos del Hashtable, podemos colocar el código que se muestra a continuación:

```
foreach(DictionaryEntry datos in miTabla)
    Console.WriteLine("Key - {0}, Value -{1}", datos.Key,
        datos.Value);
```

Si lo deseamos podemos extraer solamente los valores y colocar una copia de ellos en una colección. Esto nos permitiría trabajar con los valores de una forma más parecida a lo que hemos aprendido anteriormente.

```
ICollection valores = miTabla.Values;
...
...
foreach(double valor in valores)
    Console.WriteLine("El valor es {0}",valor);
```

ICollection es una interfase usada para implementar las colecciones, de tal forma que valores puede actuar como cualquier colección válida que tengamos, en este caso la colección que representa los valores extraídos del **Hashtable**.

Para obtener un elemento del Hashtable

Si deseamos leer un elemento en particular del **Hashtable**, es posible que lo hagamos por medio de su propiedad **Item**. Para utilizarla, simplemente tenemos que colocar como índice el **key** que corresponde al valor que queremos leer en el momento. Debemos tener en cuenta que es posible que necesitemos hacer un type cast para dejar el valor correspondiente en el tipo necesario.

```
float valor;
...
...
valor = (float)miTabla.Item["Pan"];
```

Para borrar los contenidos del Hashhtable

Todos los elementos guardados adentro del **Hashtable** pueden ser borrados al utilizar el método **Clear()**. Este método no necesita de ningún parámetro.

```
miTabla.Clear();
```

Para encontrar si ya existe un key

Como no podemos repetir el mismo **key** para dos elementos, es necesario poder saber si ya existe determinado **key** en el **Hashtable**. Poder hacerlo es fácil, pues C# nos provee del método **Contains()**. Este método necesita de un único parámetro que es el **key** a buscar y regresa un valor de tipo **bool**. Si el valor regresado es **true** significa que el **key** ya existe, y si es **false** que no se encuentra en el Hashtable.

```
bool exsite;
...
...
existe = miTabla.Contains("Pan");
```

Para encontrar si ya existe un value

Igualmente podemos buscar adentro del **Hashtable** por un **value** en particular. En este caso, usaremos el método **ContainsValue()**. Como parámetro colocaremos el valor a buscar y el método regresará un **bool**. Si el valor regresado es **true**, el **value** existe en el Hashtable y el valor de **false** nos indica que no se encuentra.

```
bool existe;
...
...
existe = miTabla.ContainsValue(17.50);
```

Para conocer la cantidad de parejas en el Hashtable

Si deseamos saber cuantas parejas **key-value** existen adentro de nuestro **Hashtable**, podemos usar la propiedad de **Count**. No hay que olvidar que el valor regresado es un entero que dice el número de parejas.

```
int cantidad;
...
...
```

```
cantidad = miTabla.Count;
```

Para eliminar un elemento del Hashtable

En el Hashtable no solamente podemos adicionar elementos, también podemos eliminarlos. Al eliminarlos removemos la pareja **key-value** del **Hashtable**. Para poder llevar a cabo la eliminación, necesitamos conocer el **key** del valor a eliminar y utilizar el método **Remove()**. Este método necesita de un solo parámetro que es el **key** del elemento a borrar.

```
miTabla.Remove("Pan");
```

Con esto finalizamos los métodos principales del **Hashtable** y las colecciones más usadas en C#.



RESUMEN

Las colecciones nos permiten guardar elementos, a diferencia de los arreglos, éstas son dinámicas y pueden modificar su tamaño según sea necesario. Las colecciones tienen diferentes comportamientos, según la estructura de datos en la que estén basadas. Las que hemos aprendido en este capítulo son: ArrayList, Stack, Queue y Hashtable. Cada colección tiene su propia clase y necesitamos instanciarla para poder utilizarla. También nos proveen de diversos métodos para poder llevar a cabo las operaciones necesarias con los elementos a guardar.



ACTIVIDADES

TEST DE AUTOEVALUACIÓN

1 ¿Qué son las colecciones?

2 ¿Cómo funciona ArrayList?

3 ¿Cómo introducimos un elemento en el ArrayList?

4 ¿Cómo funciona el ciclo foreach?

5 ¿Cómo funciona el Stack?

6 ¿Qué hace la operación Push?

7 ¿Qué hace la operación Pop?

8 ¿Para qué sirve la propiedad Count?

9 ¿Cómo funciona el Queue?

10 ¿Cómo funciona la operación Enqueue?

11 ¿Cómo funciona la operación Dequeue?

12 ¿Cómo funciona el Hashtable?

EJERCICIOS PRÁCTICOS

1 Hacer el programa que calcula el promedio, calificación máxima y mínima de un salón de clases usando el ArrayList.

2 Hacer un programa que funcione como un diccionario, con palabra y definición, usando el Hashtable.

3 Hacer un programa que simule un sistema de atención a clientes, donde el cliente que llega primero es atendido primero.

4 Hacer un programa que muestre la suma de los gastos ocurridos durante el mes, y que los muestre en orden cronológico descendente.

5 Hacer un programa que funcione como una agenda telefónica y que guarde el nombre de la persona y su número telefónico.

Las cadenas

Las cadenas son un elemento muy importante en la programación de software, ya que nos permiten guardar y manipular información escrita. Algunos lenguajes no tienen un tipo de cadena predefinido, pero C# nos da una clase con diversos métodos que nos ayudan a trabajar con ellas.

El uso de las cadenas	256
Cómo declarar la cadena	256
El método ToString()	256
Cómo convertir y formatear fechas a cadenas	257
Para darles formato a valores numéricos	259
Cómo concatenar cadenas	260
Uso de StringBuilder	261
Comparación de cadenas	263
Para encontrar una subcadena	264
Para obtener una subcadena	265
Para determinar si una cadena finaliza en una subcadena	266
Cómo copiar una parte de la cadena	267
Cómo insertar una cadena	268
Para encontrar la posición de una subcadena	269
Justificación del texto	270
Para eliminar caracteres de la cadena	271
Cómo reemplazar una subcadena	271
Cómo dividir la cadena	272
Intercambiar mayúsculas y minúsculas	273
Cómo podar la cadena	275
Resumen	277
Actividades	278

El uso de las cadenas

En los capítulos anteriores hemos trabajado con cadenas. Las utilizamos por ejemplo, cuando necesitamos mandarle mensajes al usuario. Sin embargo, las cadenas tienen mucha más funcionalidad que eso. Si necesitamos guardar cualquier información de tipo **alfanumérica** las podemos utilizar. También son utilizadas para guardar nombres, direcciones, etcétera.

Para poder utilizar cadenas necesitamos hacer uso de la clase **String**, que nos da todo lo necesario para poder trabajar con ellas. En su interior la cadena se guarda como una colección y cada carácter es un elemento. En este capítulo aprenderemos a utilizarlas y conoceremos los métodos más importantes.

Cómo declarar la cadena

Declarar una cadena es muy sencillo. De hecho, ya lo hemos llevado a cabo varias veces. Cuando necesitamos declararla, creamos una instancia de la clase **String** y le asignamos el valor que contendrá. Este valor deberá estar colocado entre comillas. Podemos declarar la variable y asignarle una cadena vacía.

```
String miCadena = "Hola a todos";
String dato = "";
```

En este caso la variable **miCadena** contendrá la frase **"Hola a todos"** y la variable **dato** contendría una cadena vacía.

Siempre representaremos las cadenas entre comillas. Podemos imaginarlas como se muestra en la **Figura 1**. Esto hace fácil poder visualizarlas y entender las operaciones que podemos realizar sobre ellas.

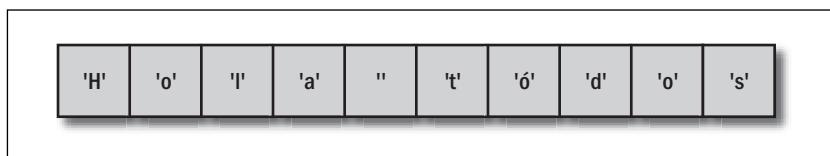


Figura 1. Una cadena puede ser representada de esta forma y se puede ver cada uno de sus elementos.

El método **ToString()**

Un método común para muchas clases que existen en C# y .NET es **ToString()**. Éste es usado para convertir el tipo de dato o su representación a una cadena y es muy útil cuando necesitamos desplegar esa información para que el usuario la pueda leer.

Si nosotros hacemos nuestras propias clases también lo podemos implementar. El método siempre regresará una cadena, que puede ser formateada para nuestras necesidades. Imaginemos que tenemos una variable de tipo entero llamada **valor** y queremos desplegársela al usuario. Para esto podemos convertir la cadena agregando el código que se muestra a continuación:

```
String Cadena = valor.ToString();
```

Cómo convertir y formatear fechas a cadenas

Uno de los usos más comunes de las cadenas es el de poder formatear información, que puede ser de cualquier tipo. Uno bastante común que no hemos utilizado hasta el momento es el de las fechas. Para utilizar la fecha y la hora, lo haremos mediante una clase conocida como **DateTime**. Esta clase provee todos los elementos necesarios para poder trabajar con la información relacionada con el tiempo, como son la fecha y las horas del día. Si deseamos tener un objeto de este tipo y que tenga la hora actual de la máquina en el momento en que se instanció podemos hacerlo de la siguiente manera:

```
DateTime miTiempo = DateTime.Now;
```

La propiedad **Now** de **DateTime** contiene el tiempo actual del sistema en su interior. Ahora lo único que necesitamos es poder pasar esa información a un formato útil y práctico de trabajar para nosotros. Podemos hacer uso del método **Format()** de **String**. Este método nos permite reemplazar los contenidos de la cadena por una cadena con un formato en particular.

Para llevar a cabo un formato debemos usar los **especificadores de formato**. **DateTime** contiene un conjunto de estos especificadores.

A continuación veamos los más importantes de estos, presentados en la **tabla 1** de una forma clara de leer y comprender.

III EL CONSTRUCTOR DE LA CADENA

Es importante visitar MSDN y obtener más información sobre la clase **String**. Una de las cosas que podemos descubrir es que tiene muchas sobrecargas a su constructor, lo que nos permite tener un control muy preciso sobre la forma y las características con las que será creada la cadena a utilizar. En este libro usaremos la forma más sencilla.

ESPECIFICADOR	DESCRIPCIÓN
d	Día del mes.
dd	Día del mes, pero los primeros días se representan con 01, 02, 03, etcétera.
ddd	Presenta el nombre del día abreviado.
dddd	Presenta el nombre del día completo.
h	Hora en el rango de 1 a 12.
hh	Hora en el rango de 1 a 12. Las primeras horas se presentan como 01, 02, 03, etcétera.
H	Hora en el rango de 0 a 23.
HH	Hora en el rango de 0 a 23. Las primeras horas se presentan como 01, 02, 03, etcétera.
m	Minuto.
mm	Minuto. Los primeros minutos se muestran como 01, 02, 03, etcétera.
M	Mes.
MM	Mes. Los primeros meses se presentan como 01, 02, 03, etcétera.
MMM	Nombre del mes abreviado.
MMMM	Nombre del mes completo.
s	Segundos.
ss	Segundos. Los primeros segundos se presentan como 01, 02, 03, etcétera.
t	Despliega A o P según AM o PM.
tt	Despliega AM o PM.
yyyy	Muestra el año.

Tabla 1. Lista de los especificadores más importantes para *DateTime*. Una vez que conocemos los especificadores simplemente debemos indicar el formato deseado para que la información de *DateTime* quede guardada en nuestra cadena.

Por ejemplo, si deseamos tener una cadena en la que se muestre la información de la fecha de forma tal que primero esté escrito el día de la semana, seguido del año y por último el mes en número, podemos hacerlo de la siguiente manera:

```
String formato;
formato=String.Format("La fecha es: {0:dddd yyyy M}",DateTime.Now);
```

III LA INFORMACIÓN ALFANUMÉRICA

Nosotros podemos guardar información alfanumérica adentro de las cadenas. Este tipo de información se refiere a los caracteres como letras, números y signos. Lo único que debemos tener en cuenta es que se guardan como caracteres dentro de la cadena, no como valores de otro tipo. Esto es especialmente importante con los números.

Adentro de **Format** colocamos la cadena de formato y como solamente debemos desplegar un valor, hacemos uso de **{0}**, pero este valor tendrá un formato en particular. Ponemos el formato al colocar : y los especificadores necesarios adentro de **{}**. De esta forma, ese valor tendrá el formato correspondiente. Nosotros podemos usar los especificadores que queramos siempre y cuando sean válidos para el tipo que deseamos formatear en la cadena.

Para darles formato a valores numéricos

Al igual que con la información de la fecha y hora, es posible darles formato a los valores numéricos que deseemos mostrar. Para éstos también tendremos una serie de especificadores que podemos utilizar.

ESPECIFICADOR	DESCRIPCIÓN
#	Dígito
.	Punto decimal
,	Separador de miles
%	Porcentaje
E0	Notación científica
;	Separador de secciones

Tabla 2. Éstos son los especificadores de formato para los valores numéricos más importantes.

Cuando especificamos un formato podemos tener diferentes secciones. Por ejemplo, podemos utilizar un formato en particular para cuando los números sean positivos y otro formato para cuando los números sean negativos. Estas secciones se separan por medio del carácter ;.

Si usamos dos secciones, el formato colocado en la primera será utilizado para los números positivos y el cero. El formato que coloquemos en la segunda sección será el que se utilizará para los números negativos. En el caso que definamos tres secciones, el formato colocado en la primera sección será utilizado para los números positivos. En la segunda sección colocaremos el formato que se

III FORMATO A VALORES NUMÉRICOS

Si deseamos aprender más sobre la forma de utilizar los especificadores, el mejor lugar para hacerlo es MSDN. En él podemos encontrar toda la documentación necesaria y ejemplos útiles que usaremos como guía para nuestros propios programas. Dar formato no es complicado, pero requiere de experimentación para aprenderlo correctamente.

utilizará cuando el valor sea negativo. La tercera sección tendrá el formato usado para cuando el valor sea cero.

Por ejemplo, podemos colocar el siguiente formato para un valor numérico:

```
Console.WriteLine(String.Format("{0:$#,##0.00;Negativo  
$#,##0.00;Cero}",  
valor));
```

En este caso, si el número contenido adentro de valor es positivo se presentará normalmente y con un signo de moneda antes de él, pero si el valor es negativo aparecerá con la palabra Negativo y el signo de moneda antes de él. Si el contenido de valor fuera cero, aparecería la palabra Cero.

Cómo concatenar cadenas

Otra de las manipulaciones que podemos hacer es la **concatenación**, que en C# a diferencia de otros lenguajes es muy natural e intuitiva. Esto se debe a que podemos utilizar una función especializada para esto o aprovechar la sobrecarga del operador + para la clase **String**. Si deseamos llevar a cabo la concatenación por medio del operador + podemos hacerlo de la siguiente manera:

```
String nombre = "Juan";  
String apellido = "Lopez";  
String NombreCompleto = "";  
NombreCompleto = nombre + " " + apellido;
```

De esta forma, obtendremos una cadena que tendrá en su interior “**Juan Lopez**”. Al hacer la concatenación, hemos incluido otra cadena. En este caso contiene un espacio. De no hacerlo, el nombre y el apellido hubieran quedado juntos y la cadena sería: “**JuanLopez**”.

III OPTIMIZAR LA CONCATENACIÓN

Cuando llevamos a cabo la concatenación por medio del operador +, puede suceder que muchas cadenas se guarden en la memoria y algunas de ellas no sean necesarias, como en el caso de las cadenas que insertan espacios. Para mejorar el uso de la memoria se puede utilizar **StringBuilder**, una clase que nos da funcionalidades para facilitar la concatenación.

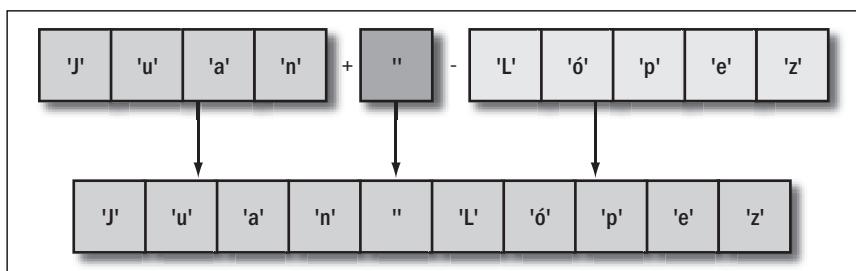


Figura 2. Aquí vemos una representación de la concatenación y la cadena resultante al final.

Otra forma de concatenar es hacer uso del método **Concat()**. Este método pertenece a **String**, es muy sencillo de utilizar y solamente requiere de dos parámetros, que serán las cadenas que deseamos concatenar. El método regresa una cadena, que es la cadena resultante de la concatenación de los parámetros.

Veamos un ejemplo de cómo podemos concatenar:

```
NombreCompleto = String.Concat(nombre, apellido);
```

El método **Concat()** es estático, por eso es posible usarlo sin tener que declarar un objeto de la clase **String**. Para llevar a cabo una concatenación múltiple, podemos hacerlo de la siguiente manera:

```
NombreCompleto = String.Concat(nombre, String.Concat(" ", apellido));
```

Uso de **StringBuilder**

StringBuilder es una clase que provee .NET y podemos utilizar con C#. Nos permite construir cadenas de forma eficiente y a su vez podemos utilizarla en lugar de las concatenaciones si fuera necesario. Aquí sólo veremos los métodos más importantes. Invitamos a los lectores a investigar el resto de los métodos con la ayuda de MSDN. Su constructor tiene varias versiones. Podemos utilizar el constructor de **default** que no necesita de ningún parámetro. Hay una versión en la que podemos pasar como parámetro una cadena con la que podemos inicializar el objeto **StringBuilder**.

La clase tiene varias propiedades que podemos usar, como la propiedad **Capacity**, que nos indica la capacidad actual, que puede o no ser igual a la cantidad de caracteres que tenemos guardados en su interior, aunque generalmente será mayor.

Otra opción es la propiedad **Chars**, mediante la que podemos obtener o modificar un carácter en particular. La modificación se hace indicando el índice donde se encuentra ese carácter. Lo podemos hacer de la siguiente manera:

```
StringBuilder sb = new StringBuilder("Hola a todos");
...
...
sb.Chars[6]='o';
```

Si leemos la propiedad de **Length** obtenemos la longitud que tiene el **StringBuilder**. Si colocamos un valor en esta propiedad entonces el **StringBuilder** tendrá el tamaño que indicamos. Si el nuevo tamaño es menor que el actual, entonces se trunca el contenido del **StringBuilder**.

Para llevar a cabo una concatenación, debemos usar el método **Append()**. Este método tiene muchas sobrecargas ya que no solamente puede concatenar cadenas. El método necesita solamente un parámetro y es el valor a concatenar en la cadena. Hay que recordar que **Append()** es eficiente.

```
StringBuilder sb = new StringBuilder("Hola a todos");
...
...
sb.Append(" Hola mundo");
```

En **sb** ahora tendremos el contenido de “**Hola a todos Hola mundo**”.

También podemos agregar una cadena de formato construida de forma similar a la forma como trabajamos con los formatos de **WriteLine()**. Para esto usamos el método **AppendFormat()**. Este método necesitará dos parámetros, pero puede llevar más si fuera necesario. En el primero colocamos la cadena de forma y en el segundo y/o los siguientes la lista de variables a utilizar. Por ejemplo:

```
StringBuilder sb = new StringBuilder();
int valor = 5;
...
...
sb.AppendFormat("El valor es {0}", valor);
```

Un método importante en esta clase es **ToString()**, que permitirá convertir **StringBuilder** en una cadena, y de esta manera poder utilizar los contenidos creados por medio de esta clase. Para usarlo simplemente hacemos una asignación:

```
String cadena = ";
```

```
cadena = sb.ToString();
```

De esta forma, la cadena tendrá en su interior la cadena creada por medio del **StringBuilder**. Existen otros métodos más que pertenecen a esta clase y pueden llegar a sernos útiles en el futuro.

Comparación de cadenas

En algunas de nuestras aplicaciones es posible que lleguemos a necesitar llevar a cabo una comparación entre algunas cadenas. Esto nos puede servir para determinar si una cadena en especial ha sido introducida por el usuario. Pero la comparación también nos puede ayudar a organizar alfabéticamente una serie de cadenas que nosotros seleccionemos, o también, simplemente para resolver algún problema que requiera saber si una cadena es mayor, igual o menor que otra.

La comparación de cadenas es una herramienta muy importante para solucionar diversos problemas y lograr resultados que de otra forma serían muy complejos.

La clase **String** nos da el método **Compare()**. Este método es estático, por lo que podemos hacer uso de éste sin necesidad de declarar un objeto de tipo **String**. El método necesitará dos parámetros, que son las cadenas a comparar. Los llamaremos en nuestro ejemplo **Cadena1** y **Cadena2**. El método regresará luego del análisis, un valor entero y el valor de este entero será el que se encargue de indicarnos la relación que existe entre las cadenas que acabamos de comparar.

Si el valor regresado es igual a **0**, esto significa que **Cadena1** y **Cadena2** son iguales. En el caso de que el valor sea menor que **0**, es decir, un número negativo, **Cadena1** es menor que **Cadena2**. Si se recibe un número mayor que **0** significa que **Cadena1** es mayor que **Cadena2**.

A continuación veamos la forma utilizar este método:

```
int comparación=0;
String nombre="Juan";
...
...
comparación=String.Compare("Pedro",nombre);

if(comparación==0)
    Console.WriteLine("Los nombres son iguales");
else
    Console.WriteLine("Los nombres son diferentes");
```

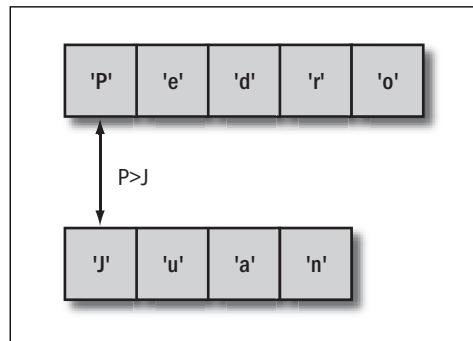


Figura 3. En este ejemplo P es mayor que J, por lo que “Pedro” es más grande que “Juan”.

Otra forma que podemos utilizar para saber si dos cadenas son iguales es hacer uso del método **Equals()**, que solamente dirá si dos cadenas son iguales, no realiza ninguna otra comparación. Existen varias sobrecargas del método **Equals()**, por lo que es conveniente aprender el resto con la ayuda de MSDN. La versión que veremos aquí es estática y requiere dos parámetros. Los parámetros son las cadenas a comparar. El método regresa un **bool** con el valor de **true** si las cadenas son iguales y el valor de **false** si son diferentes.

Podemos usarlo de la siguiente manera:

```
String cadena1 = "Hola";
String cadena2 = "Todos!";
...
...
if( String.Equals(cadena1,cadena2) == true)
    Console.WriteLine("Las cadenas son iguales");
```

Para encontrar una subcadena

La cadena puede contener una frase con muchas palabras y para algunas aplicaciones necesitamos saber si determinada palabra se encuentra en esa cadena. Es decir que buscaremos una **subcadena** adentro de otra cadena. Esta subcadena es una cadena que queremos saber si se encuentra en la cadena principal.

Esto puede servirnos en búsquedas o cuando necesitamos llevar a cabo determinada lógica si se encuentra una palabra o frase en particular. El método que usaremos se llama **Contains()** y debe ser invocado por la cadena a la que se le realizará la búsqueda. Esto es importante y no debemos olvidarlo ya que la invocación de **Contains()** en la cadena incorrecta nos puede llevar a errores de lógica.

El método **Contains()** sólo necesita un parámetro. Este parámetro es la subcadena a buscar. Ésta puede ser dada explícitamente o por medio de una variable de tipo **String** que la contenga. El método regresa un **bool**. Si la subcadena se encontró, el valor regresado es **true** y si no se encontró el valor regresado es **false**.

El uso de **Contains()** puede ser como se muestra a continuación:

```
String NombreCompleto = "Juan Pedro Lopez";
String NombreBuscar = "Pedro";
...
if (NombreCompleto.Contains(NombreBuscar) == true)
    Console.WriteLine("El nombre se encuentra");
```

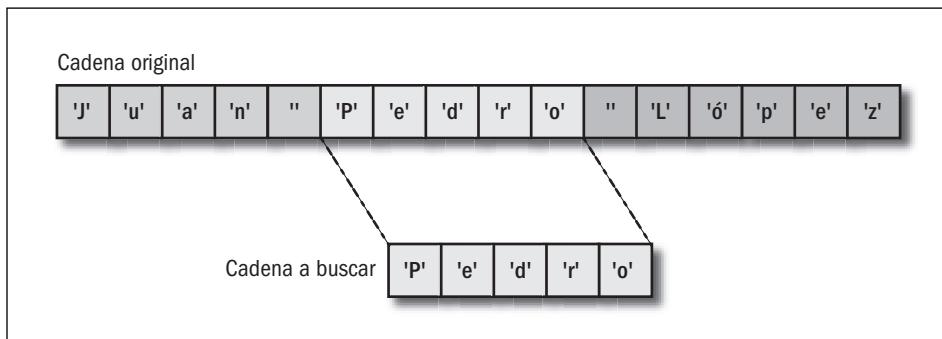


Figura 4. Vemos cómo la subcadena es encontrada en la cadena original.

Para obtener una subcadena

Ahora que ya tenemos una forma de saber si existe una subcadena, también podemos extraer una subcadena. Esto nos puede servir para los casos en los que necesitamos tener solamente una parte de la cadena original. La forma cómo lo hacemos es por medio del medio del método **Substring()**.

III PARA EVITAR PROBLEMAS CON CONTAINS()

Para evitar problemas de lógica debemos tener en cuenta puntos relacionados con **Contains()**. El primero es que una cadena vacía como "" siempre regresará **true**. Otro punto importante es que la búsqueda que realiza **Contains()** es sensible a mayúsculas y minúsculas. Buscar "Hola" no es lo mismo que buscar "Hola".

Para poder usar este método necesitamos contar con dos parámetros, que son valores de tipo entero. El primero indica el índice adentro de la cadena original donde inicia la subcadena que nos interesa obtener y el segundo es la cantidad de caracteres que tiene la subcadena. El método regresa una cadena que contiene a la subcadena que hemos obtenido.

El siguiente ejemplo nos muestra cómo podemos usar este método:

```
String cadena="Hola mundo redondo";
String resultado="";
...
...
resultado=cadena.Substring(5,5);
```

Ahora la cadena resultado tendrá en su interior a “**mundo**”, que es la subcadena que hemos obtenido de la variable **cadena**.

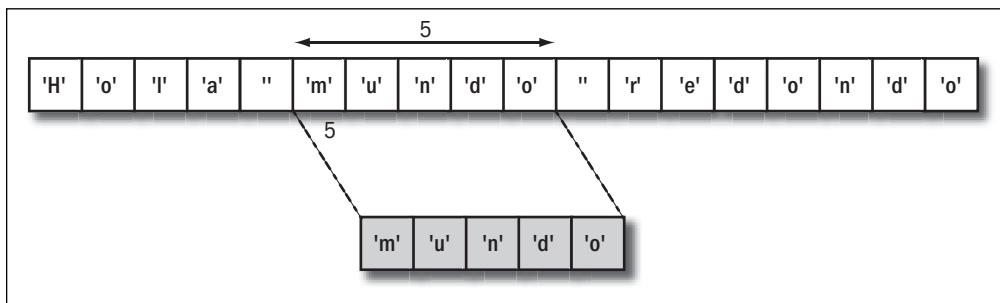


Figura 5. A partir del índice y la cantidad de caracteres se extrae la subcadena.

Para determinar si una cadena finaliza en una subcadena

Es posible saber si una cadena finaliza en una subcadena en particular. Por ejemplo, si analizamos la cadena, y ésta termina en punto, o si la cadena termina en una palabra en particular.

Al igual que en los casos anteriores, tenemos un método que nos permite llevar a cabo esto. El método es **EndsWith()** y necesita un parámetro, que será la subcadena que deseamos verificar en la terminación de la cadena principal. La función regresa un valor de tipo **bool**, si es **true** la cadena termina en la subcadena en caso contrario el valor regresado es de **false**.

Un ejemplo de cómo podemos usar esta función es el siguiente:

```
String cadena1 = "Juan Pedro Lopez";
```

```

String cadena2 = "Lopez";
...
...
if(cadena1.EndsWith(cadena2) == true)
    Console.WriteLine("La cadena finaliza en Lopez");

```

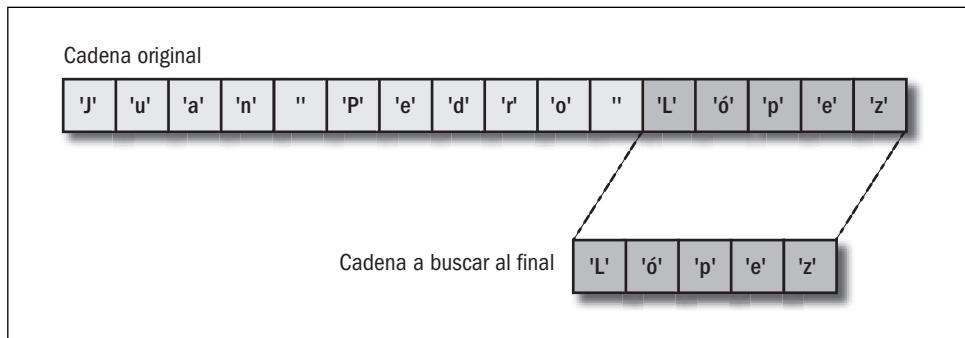


Figura 6. La búsqueda de la subcadena se hace al final de la cadena.

Cómo copiar una parte de la cadena

En muchas aplicaciones es necesario obtener o copiar una parte en particular de una cadena, y si la cadena contiene una frase, podemos extraer una palabra en particular. Cuando necesitamos procesar cadenas es muy útil hacer esto.

El método que usaremos se conoce como **CopyTo()** y aunque es un poco más complejo que los anteriores es fácil de utilizar ya que necesita el mismo de cuatro parámetros. Antes de ver los parámetros es necesario comentar que la cadena extraída será colocada en un arreglo de tipo **char**. El método no regresa ningún valor.

El primer parámetro es el índice adentro de la cadena principal desde donde se empezará a copiar. El segundo parámetro es la cadena de tipo **char** donde se colocará la cadena a extraer. El tercer parámetro es el índice del arreglo a partir de donde se empezarán a colocar los caracteres de la cadena copiada. Esto es útil ya que podemos hacer la copia en cualquier posición del arreglo, no solamente al inicio. El cuarto parámetro indica la cantidad de caracteres a copiar. Con estos parámetros podemos indicar sin problema cualquier sección a copiar.

Veamos un ejemplo de esto último a continuación:

```

char[] destino = new char[10];
String cadena = "Hola a todos mis amigos";
...

```

```
...
cadena.CopyTo(7, destino, 0, 5);
```

En este caso hemos extraído la cadena “**todos**” y se ha guardado en el arreglo destino.

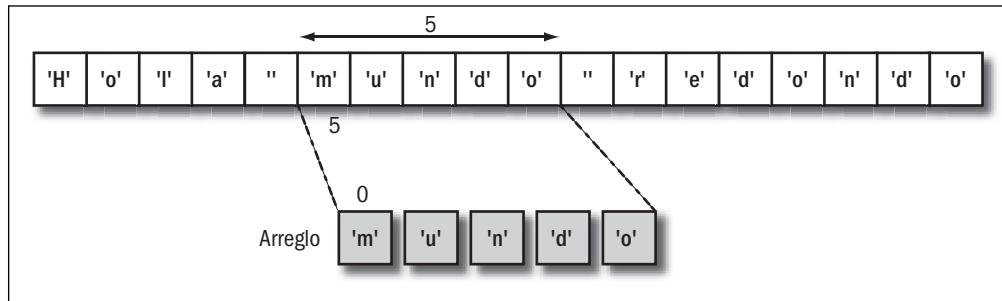


Figura 7. Los caracteres son copiados al arreglo a partir del índice indicado.

Cómo insertar una cadena

Ya hemos visto una manera fácil de cómo podemos extraer una subcadena, sin embargo, también es posible llevar a cabo la inserción de una cadena en otra. Esto nos ayuda cuando necesitamos colocar información nueva en la cadena, pero ya no es posible hacerlo por medio de concatenación o formato.

Para poder hacer esto, recurriremos a utilizar al método **Insert()**, que nos brinda C#. Este método coloca la cadena dentro de la cadena principal. Este también necesitará dos parámetros. El primero indicará el índice dentro de la cadena principal donde se va a insertar la nueva cadena. Este valor debe de ser de tipo entero y nunca podemos pasarle un valor negativo. El segundo parámetro es la cadena que deseamos insertar, que puede ser colocada explícitamente o mediante una variable de tipo **String**.

El método **Insert()** regresa un valor de tipo **String**. Este valor regresado sería la instancia de la nueva cadena que ya contiene la inserción. Un ejemplo para llevar a cabo la inserción sería de la siguiente forma:

III PARA EVITAR ERRORES CON EL MÉTODO INSERT()

Si necesitamos llevar a cabo inserciones de cadenas debemos tener cuidado con los parámetros para no tener problemas con nuestro programa. El índice donde se inserta la cadena nunca debe ser negativo o tener un valor más grande que el tamaño de la cadena principal. La variable de la cadena a insertar debe contener una cadena válida y no un valor **null**.

```

String cadena1 = "Hola a todos";
String cadena2 = "hola ";
String resultado = "";
...
...
resultado = cadena1.Insert(5, cadena2);

```

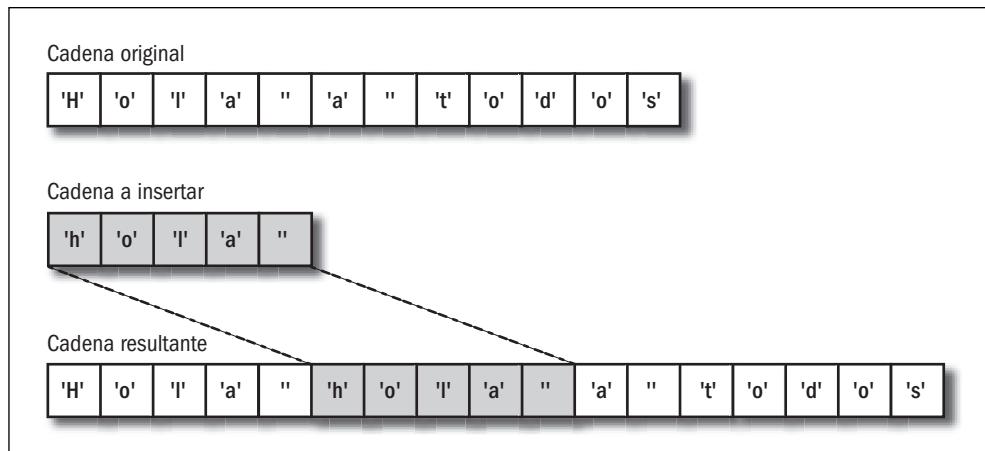


Figura 8. En esta figura podemos ver la inserción de una cadena.

Para encontrar la posición de una subcadena

En algunas situaciones necesitamos encontrar dónde se encuentra determinada subcadena para poder llevar a cabo alguna operación en ese índice en particular. Para poder hacer esto hacemos uso del método **LastIndexOf()**. Un punto importante a tener en cuenta es que este método nos regresa el índice de la última posición encontrada, es decir que si la cadena principal tiene dos repeticiones de esa cadena, nos da el índice de la segunda repetición.

Este método tiene varias sobrecargas, pero la que veremos sólo necesita un parámetro y es la subcadena a encontrar. Como siempre la cadena puede ser dada de forma explícita o por medio de una variable de tipo **String**. El método regresa un valor de tipo entero que contiene el índice de la última ocurrencia de la subcadena.

La forma para usar este método es la siguiente:

```

int indice = 0;
String cadena = "Hola a todos. Hola";
...
...

```

```
índice=cadena.LastIndexOf("Hola");
```

Justificación del texto

Hasta el momento hemos visto métodos para manipular las cadenas dentro de C#, pero no hemos hablado aún que la clase **String**, que nos provee de varios más. A continuación, veremos algunos que nos permiten hacer otras modificaciones.

Si necesitamos justificar el texto hacia la derecha es posible hacerlo. La forma cómo funciona esto es la siguiente. Supongamos que tenemos una cadena de **10** caracteres como “**0123456789**”. Para justificarla necesitamos saber el tamaño de la cadena resultante con la justificación incluida. Supongamos que la cadena resultante será de **25** caracteres. Esto nos daría **15** caracteres del que tenemos que insertar del lado izquierdo para obtener “**0123456789**” que se encuentra justificada hacia la derecha. Para esto necesitamos un método que nos permita empotrar esos caracteres de espacio. El método **PadLeft()** se encarga de esto. Requiere de un parámetro que es la cantidad de caracteres de la cadena final. Éste es un valor entero y representa los caracteres originales más los espacios en blanco. Regresa la cadena final justificada.

Un ejemplo de uso sería el que vemos a continuación:

```
String cadena="Hola";
String resultado="";
...
...
resultado=cadena.PadLeft(10);
```

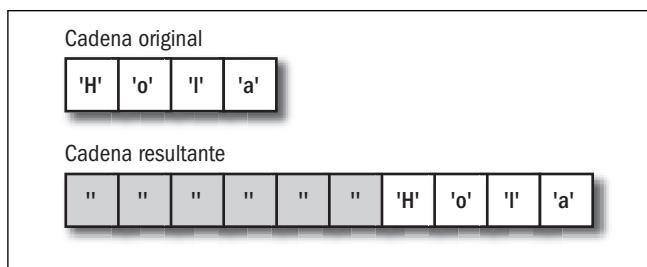


Figura 9. El texto ha sido justificado a la derecha al insertar los espacios del lado izquierdo.

De forma similar, podemos hacer una justificación hacia la izquierda. En este caso, los caracteres en blanco serán insertados a la derecha de la cadena. Se insertarán tantos caracteres como sean necesarios hasta llegar al tamaño deseado.

Para esto usaremos un método equivalente a **PadLeft()**, pero hacia la derecha. Este método es conocido como **PadRight()**. El método necesita un parámetro, que es un valor entero que indica la cantidad total de caracteres de la cadena final. Este total debe ser igual a los caracteres de la cadena original más los caracteres vacíos. El método regresa la cadena justificada.

Para eliminar caracteres de la cadena

Otra manipulación que podemos hacer sobre la cadena es la eliminación de caracteres. Es posible borrar determinada parte de la cadena, según sea lo que necesitemos en la lógica del programa.

El método que podemos utilizar se conoce como **Remove()**. Este método está sobrecargado, pero veremos la versión que es más flexible. La eliminación de los caracteres puede hacerse en cualquier parte de la cadena, sólo debemos tener cuidado de no generar ningún error.

El método **Remove()** necesita dos parámetros, ambos de valores enteros. El primer parámetro se usa para indicar el índice a partir del cual se empezarán a eliminar los caracteres de la cadena, y en el segundo parámetro colocamos la cantidad de caracteres a eliminar. El método regresa una cadena, que es la cadena resultante de la eliminación.

Para poder eliminarla realicemos el siguiente ejercicio:

```
String cadena="Hola mundo, hola a todos";
String resultado="";
...
...
resultado=cadena.Remove(12,4);
```

Cómo reemplazar una subcadena

Reemplazar una parte de la cadena principal con otra cadena puede ser un proceso que toma tiempo programar. Esto se debe a que necesitamos encontrar la subcadena a eliminar, luego eliminarla y al final introducir los caracteres de la nueva cadena. Como siempre, la clase **String** nos provee de un método que nos facilita la manipulación de la cadena para el reemplazo. Éste es el método **Replace()**. Existen dos versiones del método **Replace()**, una de ellas funciona con caracteres y la que aprenderemos en esta ocasión hace uso de cadenas.

Esto nos puede permitir reemplazar una palabra en particular contenida dentro de la cadena por otra palabra. El método llevará a cabo el reemplazo en todas las ocurrencias de la palabra que tengamos.

El método es sencillo de utilizar y necesita dos parámetros. El primero es la cadena que simboliza la palabra que deseamos reemplazar. El segundo es la cadena con la que la reemplazaremos. El método regresa una cadena, que es la resultante con los reemplazos ya llevados a cabo.

Veamos un ejemplo de la utilización de este método:

```
String cadena1="Hola a todos. Hola mundo";
String cadena2="Adios";
String resultado="";
...
...
resultado=cadena1.Replace("Hola",cadena2);
```

Por el ejemplo, vemos que es posible colocar el parámetro ya sea de forma explícita o por medio de una variable.

Cómo dividir la cadena

Otro problema clásico con la manipulación de las cadenas es la subdivisión cuando se encuentra algún carácter en particular. Por ejemplo, si la cadena contiene un párrafo de un texto, podemos dividirla en subcadenas, cada una de ellas delimitada por un **signo de puntuación**. Para lograr esto necesitamos tener un arreglo de caracteres que contenga los caracteres que tomaremos como **delimitadores**. Cada vez que el método encuentre uno de estos caracteres, llevará a cabo el corte de la subcadena.

El método que usaremos se conoce como **Split()**, éste sólo requiere de un parámetro, que es el arreglo de caracteres delimitadores. El método regresará un arreglo de tipo **String**. Cada uno de los elementos presentes en el arreglo regresado será una de las subcadenas que se habrán recortado. Después debemos proceder a hacer uso de las cadenas en el arreglo y procesarlas o utilizarlas según sea necesario en la aplicación que estamos desarrollando.

III POSIBLES PROBLEMAS CON EL REEMPLAZO

No colocar valores adecuados en los parámetros puede traer problemas al momento de utilizar el método **Replace()**. Esto puede suceder cuando el valor del parámetro se pasa por medio de una variable y ésta no contiene un valor adecuado. El método puede ocasionar problemas cuando el primer parámetro recibe **null** o una cadena vacía del tipo "".

Un ejemplo de cómo podemos usar este método es el siguiente:

```
String cadena1="Hola a todos. Este es un ejemplo, de lo que podemos
hacer.");
String resultado[] = cadena1.Split(new char[] { ',', '.', ';' });
```

Como podemos ver en el ejemplo, usamos los caracteres punto, coma y punto y coma como delimitadores. Adentro de **resultado** estarán las cadenas que han sido recortadas de **cadena1**.

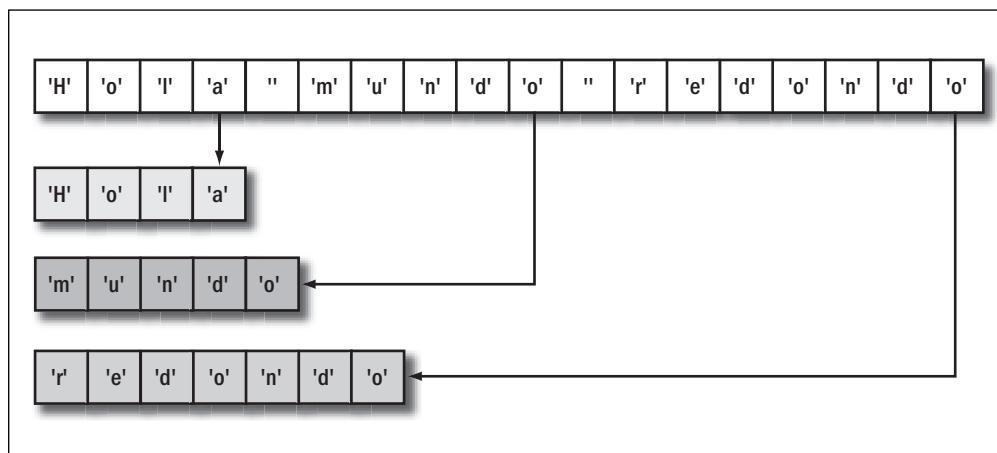


Figura 10. La cadena ha sido subdividida. Al ser el delimitador, el espacio no aparece en las cadenas resultantes.

Intercambiar mayúsculas y minúsculas

En muchas ocasiones tendremos cadenas que estarán escritas con letras en mayúscula y minúscula mezcladas, pero puede suceder que para facilitar la lógica de aplicación debamos tener la cadena exclusivamente en mayúscula o minúscula.



PARA PODAR OTROS CARACTERES

Si necesitamos podar otros caracteres en lugar del espacio en blanco, es posible hacerlo. Para esto tenemos que usar la versión sobrecargada de **Trim()**. Esta versión necesitará un parámetro. En él colocamos un arreglo de caracteres. Este arreglo tendrá aquellos caracteres a podar del inicio y fin de la cadena original.

Esto nos puede ayudar a reducir la cantidad de comparaciones o búsquedas que necesitamos hacer. Si lo que deseamos hacer es convertir la cadena a minúscula, entonces debemos hacer uso del método **ToLower()**. Este método no necesita ningún parámetro. La fuente de información para hacer la conversión es la misma cadena que lo invoca, pero en esta ocasión nos daremos cuenta de que el método sí se encargará de regresar un valor, el valor que se devuelve será la cadena convertida totalmente a letras minúsculas.

Por ejemplo, podemos tener lo siguiente:

```
String cadena="Hola Hola";
String resultado="";
...
...
resultado=cadena.ToLower();
```

Al finalizar el código la variable resultado tendrá en su interior a la cadena **"hola hola"**. De forma similar, podemos pasar la cadena a mayúscula. La forma de hacer esto es con el método **ToUpper()**, que toma la cadena y pasa todas sus letras a mayúscula. El método no necesita ningún parámetro, ya que al igual que **ToLower()**, toma la información directamente de la cadena que lo invoca y regresa una cadena, que es la resultante con todas las letras en mayúscula.

Veamos un ejemplo de cómo aplicar esto:

```
String cadena="Hola Hola";
String resultado="";
...
...
resultado=cadena.ToUpper();
```

III PUNTOS IMPORTANTES SOBRE EL MÉTODO SPLIT()

Cuando usamos el método **Split()** es bueno conocer que ninguno de los caracteres que usamos como delimitadores se copiarán a las subcadenas. Si la cadena no contiene ninguno de los delimitadores, lo que obtendremos será un arreglo con un solo elemento que es la misma cadena. Si el arreglo de delimitadores es **null** o está vacío, el carácter espacio será usado como delimitador.

La variable resultado tendrá en su interior “**HOLA HOLA**”.

Los dos métodos tienen otra versión que tiene en cuenta las reglas culturales para poder llevar a cabo la conversión. En ese caso tendríamos que pasar como parámetro el identificador de la cultura que se usará como base para convertir los caracteres.

En MSDN podemos encontrar la información sobre los diferentes identificadores de cultura que puede manejar .NET. Para hacer uso del identificador de cultura debemos usar un objeto de tipo **CultureInfo**. En su constructor debemos pasar el ID de la cultura correspondiente. Por ejemplo, para pasar a mayúscula con las reglas de la cultura en México podemos colocar lo siguiente:

```
String cadena="Hola Hola";
String resultado="";
...
...
resultado=cadena.ToUpper(new CultureInfo("es-MX"));
```

Cómo podar la cadena

Cuando trabajamos con las cadenas podemos encontrarnos con situaciones como cuando la cadena tiene exceso de espacios, ya sea al inicio o al final. Algunas veces esto puede ser útil, como cuando justificamos, pero otras veces estos espacios extras pueden ser indeseados. Los espacios extras al inicio o al final pueden deberse a operaciones realizadas sobre la cadena o simplemente a entradas erróneas del usuario, y para librarnos de estos caracteres tenemos diferentes opciones. En primer lugar conoceremos un método que elimina los espacios en blanco extras tanto al inicio como al final de la cadena. Este método se conoce como **Trim()**.

El uso de **Trim()** es muy sencillo ya que no necesita ningún parámetro, simplemente trabajará sobre la cadena que lo invoca. Este método regresará una cadena nueva, que es la cadena original sin los espacios extras.

Es necesario que tengamos en cuenta que la cadena original no se modifica, por esta razón recibiremos una cadena completamente nueva.

III VALORES NUMÉRICOS EN LAS CADENAS

Hay que tener cuidado cuando se guardan valores numéricos en las cadenas. No es lo mismo **54** que **“54”**. Cuando se guarda adentro de la cadena, no queda como un número sino como carácter **“5”** seguido de carácter **“4”**, que es diferente. Si necesitamos pasar la cadena a una variable que pueda contener números, debemos hacer uso de la clase **Convert**.

Veamos un ejemplo de cómo poder utilizar este método:

```
String cadena="      Hola a todos.      ";
String resultado="";
...
...
resultado=cadena.Trim();
```

En la cadena resultado tendremos “**Hola a todos**”, que es la cadena sin los espacios extras. El método **Trim()** poda los espacios tanto del inicio como del final de la cadena. Sin embargo, puede haber ocasiones en las que necesitemos podar únicamente el inicio de la cadena. El método para lograr esto es **TrimStart()**. Este método es un poco más complejo que **Trim()** ya que necesita un parámetro. Este parámetro es un arreglo de caracteres y en él tenemos que colocar los caracteres que nos interesa extraer del inicio de la cadena. El método regresará otra cadena, que es la resultante de la cadena original sin los caracteres podados a su inicio.

Es posible que creemos el arreglo de caracteres o también que lo coloquemos explícitamente. Es conveniente pensar bien cuáles serán los caracteres a podar para evitar la eliminación de caracteres que sí pueden ser útiles.

Veamos cómo podemos utilizar este método:

```
String cadena="x x x x x x Hola a todos.      ";
String resultado="";
...
...
resultado=cadena.TrimStart(' ', 'x');
```

En este ejemplo se podan los caracteres espacio y **x**. La cadena final resultante es “**Hola a todos**”. Debemos notar que los caracteres al final no han sido podados, ya que solamente trabaja sobre los caracteres al inicio de la cadena.

Si lo que necesitamos es podar el final de la cadena, entonces tenemos que usar el método **TrimEnd()**. Este método es equivalente a **TrimStart()**, pero funciona únicamente al final de la cadena.

Recordemos que el método necesita un parámetro. El parámetro es un arreglo de caracteres. En este arreglo es necesario que coloquemos los caracteres que deseamos eliminar del final de la cadena con la que trabajamos. El método correspondiente se encargará de regresar una cadena, está será el resultado sin los caracteres que fueron eliminados al final de la cadena original.

Veamos un ejemplo del uso de este método:

```
String cadena="x x x x x x Hola a todos.           ";
String resultado="";
...
...
resultado=cadena.TrimEnd(' ', 'x');
```

En la cadena de resultado tendremos “**x x x x x x Hola a todos**”, el espacio fue eliminado al final ya que se encuentra dentro de la lista de caracteres a podar. Con esto hemos visto el uso y la manipulación de las cadenas, así como los métodos más importantes que nos permiten trabajar con ellas.



RESUMEN

En este capítulo hemos aprendido muchas cosas sobre las cadenas y su manipulación. Aprendimos cómo obtener la hora y fecha del sistema y colocar esa información con formato adentro de una cadena. También aprendimos cómo darles formato a los valores numéricos que podemos presentarle al usuario. La concatenación nos sirve para unir dos cadenas o más y tenemos varias formas de poder llevarla a cabo. `StringBuilder` es una clase que nos da apoyo para la creación de cadenas. Podemos comparar las cadenas por medio del método `Compare()` o `Equals()`. Podemos saber si una subcadena se encuentra adentro de la cadena principal y también obtener una subcadena extraída de la cadena principal, la justificación del texto es posible, así como su contraparte que consiste en eliminar espacios extras al inicio o al final de la cadena. Es posible pasar la cadena enteramente a mayúscula o minúscula.



ACTIVIDADES

TEST DE AUTOEVALUACIÓN

1 ¿Qué es una cadena?

2 ¿Qué clase se usa para obtener la fecha y hora de la computadora?

3 ¿Cómo se les da el formato a la fecha y la hora?

4 ¿Cómo se le da el formato a un valor numérico?

5 ¿Qué es la concatenación?

6 ¿De qué forma nos ayuda StringBuilder?

7 ¿De qué manera se lleva a cabo la comparación de cadenas?

8 ¿Cómo podemos obtener una parte de la cadena?

9 ¿Cómo podemos saber si una palabra se encuentra adentro de una cadena?

10 ¿Cómo podemos justificar una cadena a la derecha?

11 ¿De qué forma podemos eliminar los espacios extras al final de la cadena?

12 ¿De qué forma podemos colocar una cadena totalmente en mayúscula?

EJERCICIOS PRÁCTICOS

1 Hacer un programa que le pida al usuario una cadena y muestre cada una de las palabras de la cadena en una línea.

2 Hacer un programa que le pida al usuario una cadena y la forma de justificarla.

3 Hacer un programa que muestre la hora del día en formato AM/FM seguida del año, el día y el mes actual.

4 Hacer un programa que calcule sumas y muestre los resultados negativos entre paréntesis.

5 Hacer un programa que le solicite al usuario dos cadenas y luego las muestre en orden alfabético.

Estructuras y enumeraciones

En este capítulo aprenderemos un tema muy útil. Empezaremos por cómo utilizar los tipos definidos por el programador, es decir, los tipos de datos definidos por nosotros. Estos tipos nos darán la flexibilidad de poder guardar lo que nosotros necesitemos en nuestra aplicación y poder utilizarlos de forma similar a las variables.

Las estructuras	280
Cómo definir una estructura	280
Cómo crear una variable del nuevo tipo	282
Cómo acceder a los campos de la estructura	283
Cómo mostrar los campos de la estructura	283
Creación de un constructor para la estructura	286
Estructuras enlazadas	299
Las enumeraciones	309
Resumen	315
Actividades	316

Las estructuras

Las estructuras son tipos definidos por el programador y son un conjunto de datos agrupados. Supongamos que programamos una agenda telefónica. Desde luego, la agenda necesita guardar mucha información, por lo que ya hemos pensado en usar arreglos. Pero también necesitamos pensar en los datos que debe contener como: nombre, edad, teléfono. Esto nos lleva a tener tres arreglos. Si necesitamos más datos, entonces más arreglos serán necesarios. Al final es posible que sea un poco complicado administrar tantos arreglos y esto reduce la flexibilidad de nuestro software. Lo mejor sería poder agrupar esa información, de forma tal que solamente tengamos que administrar un solo arreglo. La forma de poder agrupar la información es por medio del uso de las estructuras.

Cuando creamos una estructura, definimos un nuevo tipo y adentro de este tipo podremos colocar datos. Estos datos se conocen como **campos**. Cada campo está representado por una variable, que puede ser de cualquier tipo. Luego que hemos terminado de definir la estructura podemos crear variables de esta estructura y guardar la información que necesitemos en ellas.

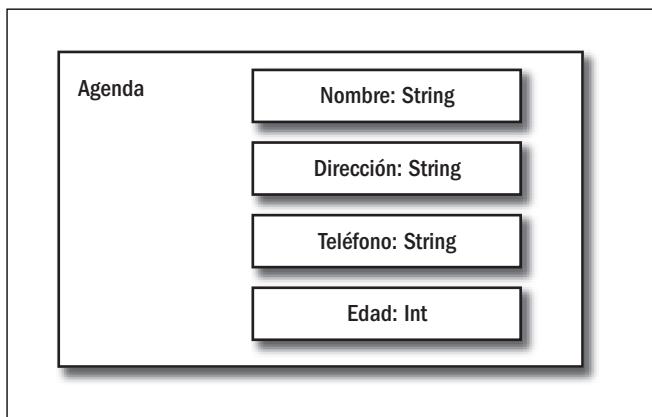


Figura 1. Ésta es una representación de una estructura y sus campos.

Cómo definir una estructura

Definir una estructura es sencillo. Sin embargo, el primer paso no se lleva a cabo en la computadora. Lo primero que tenemos que hacer es encontrar cuáles son los campos y el tipo que debe guardar la estructura. En el programa la definiremos utilizando el código que comentamos en el siguiente bloque:

```

acceso struct nombre
{
  
```

```

acceso tipo campo1;
...
acceso tipo campoN;

}

```

El **acceso** indica si la estructura puede verse por afuera del ámbito donde ha sido definida o no. Si el acceso es de tipo **público**, se puede acceder a la estructura por afuera del ámbito. Para esto pondremos como acceso **public**. Si no deseamos que el exterior pueda acceder a la estructura, entonces la colocamos como **privada**. Para esto colocamos el acceso como **private**. En este libro siempre trabajaremos con acceso de tipo **public**. Para indicar que definimos una estructura usamos **struct** seguido del nombre de la estructura. Éste puede ser cualquier nombre válido en C#. No hay que confundir el nombre de la estructura con la o las variables que usaremos de ella. Con este nombre identificaremos el nuevo tipo.

Luego tenemos que crear un bloque de código. Dentro de este bloque definiremos los campos que necesitamos. Éstos se definen igual que las variables, pero es necesario colocar el acceso. En este libro utilizaremos un acceso de tipo **public**, de forma tal que podamos leer la información contenida adentro de la estructura.

Por ejemplo, la estructura de nuestra agenda puede ser definida de la siguiente forma:

```

public struct Agenda
{
    public String Nombre;
    public int Edad;
    public String Telefono;
}

```

Así de sencilla es la definición. Ya tenemos una estructura llamada **Agenda** que contiene los campos: **Nombre**, **Edad** y **Telefono**.

III SELECCIÓN ADECUADA DE LOS CAMPOS

El hacer un análisis correcto de los campos y los tipos que debe tener una estructura nos será de mucha ayuda. Esto nos ahorra cambios que pueden ser muy costosos en tiempo y mantenimiento de nuestro programa. La lista de campos surge de manera similar a la lista de variables que hemos aprendido en el **Capítulo 2**.

Nosotros podemos definir tantas estructuras como sean necesarias y también cada estructura puede tener la cantidad necesaria de campos.

Cómo crear una variable del nuevo tipo

Ya que tenemos definida la estructura, es necesario poder definir sus variables para guardar información. Como hemos realizado la declaración de un nuevo tipo, entonces nos será posible crear variables de ésta, y la forma de hacer esto es similar a la de cualquier otro tipo.

Para definir la variable sólo deberemos poner el nombre del tipo seguido del nombre que le daremos a nuestra variable. No debemos olvidar colocar ; (punto y coma) al final de la sentencia. Supongamos que deseamos crear una variable cuyo nombre será **amigo** y es del tipo agenda.

```
Agenda amigo;
```

Ahora, adentro del nuevo tipo de variable declarada llamada **amigo**, se encuentran los campos **Nombre**, **Edad** y **Telefono**. Es posible declarar varias variables de **Agenda**.

```
Agenda amigo1, amigo2, amigo3;
```

En el caso de necesitar hacer uso de éste, podemos crear un arreglo de la estructura. Este arreglo puede ser del tamaño que necesitemos para nuestra aplicación, y es entonces de esta forma que para la estructura agenda podemos tener un arreglo donde guardamos la información de los amigos, y otro arreglo donde se guardará la información de los clientes.

```
Agenda []amigos=new Agenda[15];
Agenda []clientes=new Agenda[5];
```

III CREACIÓN DE VARIABLES DE ESTRUCTURA

Podemos definir una variable de la estructura de la forma tradicional, pero también podemos hacerlo por medio **new**. Con ésta última es posible colocar un constructor adentro de la estructura y facilitarnos la introducción de información. Si se define de forma tradicional deberemos inicializar cada campo de la estructura manualmente.

Cómo acceder a los campos de la estructura

Ya tenemos variables de la estructura y sabemos que adentro de éstas se encuentran los campos que guardarán la información. Tenemos que tener acceso a los campos para poder guardar, leer o modificar los datos. El acceso al campo se lleva a cabo de la siguiente manera:

```
VariableEstructura.Campo
```

Esto quiere decir que primero debemos indicar cuál es la variable con la que deseamos trabajar. Luego necesitamos utilizar el operador **punto** y seguido de él colocamos el nombre del campo a acceder. Veamos un ejemplo de esto:

```
amigo.Edad = 25;
```

En este caso decimos que al campo **Edad** de la variable **amigo** le colocamos en su interior el valor **25**. Podemos utilizar los campos en cualquier tipo de expresión válida, tal y como una variable:

```
if(amigo.Edad>18)
...
diasVividos=amigo.Edad * 365;
...
Console.WriteLine("El nombre es {0}",amigo.Nombre);
```

Cómo mostrar los campos de la estructura

Algo que necesitaremos constantemente es presentar los campos de la estructura. La manera más evidente de hacerlo es con el método **WriteLine()**. Simplemente mostramos el contenido del campo como si fuera otra variable del programa.



CAMPOS OLVIDADOS

Es posible que tengamos una estructura y que sólo utilicemos determinados campos, sin embargo, los demás campos continúan existiendo. Algunos de éstos pueden inicializarse automáticamente a **0** o **null** si usamos **new** para crear la variable. Pero si se hace de forma tradicional y no se han inicializado, entonces ese campo no tiene instancia y originará problemas si se intenta usar.

```
Console.WriteLine("La edad es {0}",amigo.Edad);
```

Esta forma es sencilla y útil en los casos en los que solamente necesitamos mostrar un campo de la estructura. Sin embargo, de necesitar mostrar todos los campos, deberemos colocar **WriteLine()** para cada campo, o uno solo con una cadena de formato larga. En cualquier caso resulta incómodo.

La mejor forma de mostrar todos los datos contenidos en los campos sería mediante la conversión de la estructura a una cadena. Sabemos que existe un método llamado **ToString()** que hemos usado con las variables numéricas. Sin embargo, C# no se lo puede dar directamente a nuestra estructura porque no puede saber cuáles son los campos que contiene ni cómo los deseamos mostrar. Por esto, cae en nuestra responsabilidad programar el método **ToString()** de nuestra estructura.

Veamos esto en el siguiente ejemplo:

```
public struct Agenda
{
    public String Nombre;
    public int Edad;
    public String Telefono;

    public override String ToString()
    {
        StringBuilder sb = new StringBuilder();
        sb.AppendFormat("Nombre: {0}, Edad: {1},
                        Telefono: {2}", Nombre, Edad,
                        Telefono);
        return (sb.ToString());
    }
}
```

Podemos observar que adentro del bloque de código de la estructura hemos colocado nuestra versión del método **ToString()**. El acceso debe ser público para que se pueda invocar desde el exterior de la estructura. Este método debe regresar un objeto de tipo **String** y no necesita ningún parámetro.

En el interior del método simplemente colocamos el código necesario para darle formato a la cadena y en nuestro caso hacemos uso de **StringBuilder**, aunque es válido usar cualquier otro método. Al finalizar ya tenemos la cadena con la información de nuestros campos y la regresamos por medio de **return**.

Ahora que ya tenemos implementado **ToString()** mostrar la información contenida será tan sencillo como:

```
Console.WriteLine(amigo.ToString());
```

Veamos un ejemplo completo de estos conceptos:

```
public struct Agenda
{
    public String Nombre;
    public int Edad;
    public String Telefono;

    public override String ToString()
    {
        StringBuilder sb = new StringBuilder();
        sb.AppendFormat("Nombre: {0}, Edad: {1},
                        Telefono: {2}", Nombre, Edad,
                        Telefono);
        return (sb.ToString());
    }
}

static void Main(string[] args)
{
    //
    // TODO: agregar aquí código para iniciar la aplicación
    //

    Agenda []amigos=new Agenda[5];

    amigos[1].Edad=25;
    amigos[1].Nombre="Juan";
    amigos[1].Telefono="(555) 123-4567";

    Console.WriteLine(amigos[1].ToString());
}
```

Como parte del ejemplo creamos un arreglo de la estructura **Agenda** que se llama **amigos** y tiene **5** elementos, luego, para el elemento en el índice **1** colocamos los datos. Para finalizar simplemente imprimimos el elemento **1** del arreglo **amigos**, pero usamos el método **ToString()** programado por nosotros. Si compilamos y ejecutamos el programa obtendremos lo siguiente:

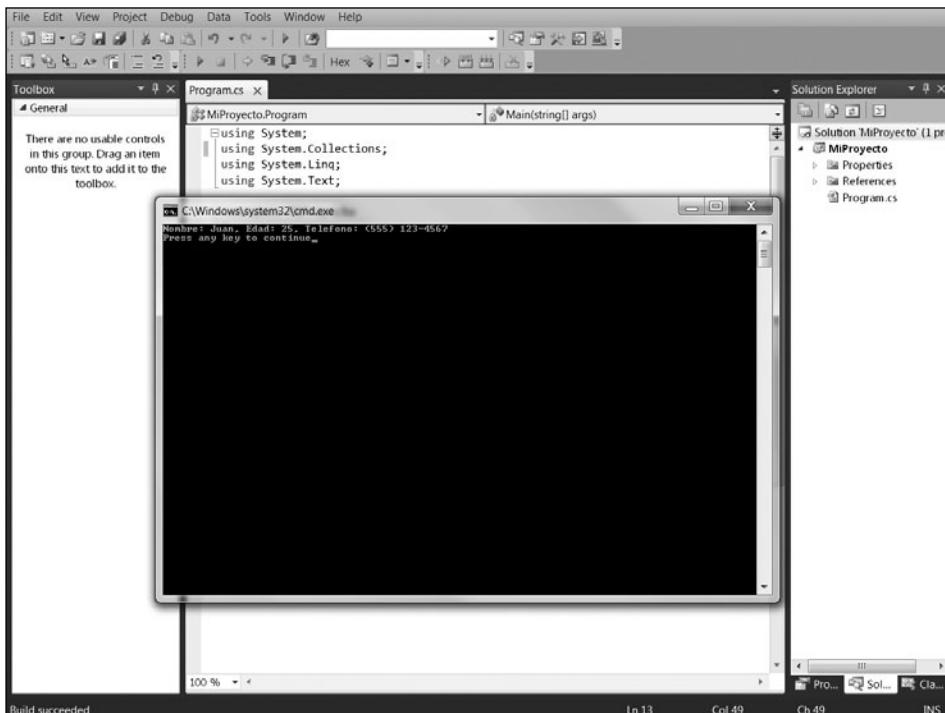


Figura 2. Podemos observar cómo por medio de **ToString()** obtenemos la cadena con la información de la estructura y la mostramos en la consola.

Creación de un constructor para la estructura

En el ejemplo anterior hemos visto la manera de cómo cada uno de los campos que componen la estructura ha tenido que inicializarse en forma individual. Esto es correcto y no presenta ningún problema, sin embargo, hay una forma de poder inicializar los campos más fácilmente, sin tantas complicaciones.

Para hacer esto podemos hacer uso de un **constructor**. El constructor no es otra cosa que un método que nos permitirá llevar a cabo la inicialización de los campos. Sin embargo, este método tiene algunas características especiales. La primera característica es que siempre se llamará igual que la estructura a la que pertenece. La segunda es muy importante: el constructor se invoca automáticamente cuando llevamos a cabo la instanciación de la variable de la estructura. La última característica del constructor es que no tiene tipo. No sólo no regresa nada, no tiene tipo.

Adentro del constructor podemos colocar cualquier código válido de C#, pero es evidente que colocaremos código dedicado a la inicialización de los campos. Veamos un primer ejemplo de cómo podemos crear un constructor. El constructor siempre va adentro del bloque de código de la estructura.

```
public Agenda(String pNombre, int pEdad, String
    pTelefono)
{
    // Llevamos a cabo la asignación
    Nombre=pNombre;
    Edad=pEdad;
    Telefono=pTelefono;
}
```

En el código del constructor vemos que el acceso es público. Esto es necesario y siempre debemos dejarlo así. Si observamos luego se coloca directamente el nombre del constructor. El nombre es **Agenda**, ya que pertenece a la estructura **Agenda**. A continuación tenemos la lista de parámetros. Los valores pasados como parámetros serán asignados a los campos correspondientes. En la declaración de la variable lo tendremos que usar de la siguiente forma:

```
Agenda amigo=new Agenda("Juan",25,"(555) 123-4567");
```

Aquí vemos que declaramos la variable **amigo** que es de tipo **Agenda**. Al hacer la instanciación por medio de new vemos que pasamos los parámetros. Éstos serán los datos que quedarán guardados en los campos de la variable **amigo**. La cadena “**Juan**” quedaría adentro del campo **Nombre**, el valor de **25** adentro del campo **Edad** y la cadena “**(555) 123-4567**” en el campo **Telefono**.

Veamos cómo quedaría un programa completo con el constructor:



EL USO DE OVERRIDE

El método **ToString()** se definió en la clase **Object**, de la cual desciende todo en C#, pero al estar definiendo una versión nueva para nuestra estructura usamos **override**, que le indica al compilador que creamos una nueva implementación que usará nuestra estructura. Así, al invocar **ToString()** se usará la versión implementada por nosotros en lugar de la versión base.

```

public struct Agenda
{
    public String Nombre;
    public int Edad;
    public String Telefono;

    public Agenda(String pNombre, int pEdad, String
                  pTelefono)
    {
        // Llevamos a cabo la asignación
        Nombre=pNombre;
        Edad=pEdad;
        Telefono=pTelefono;
    }

    public override String ToString()
    {
        StringBuilder sb = new StringBuilder();
        sb.AppendFormat("Nombre: {0}, Edad: {1},
                        Telefono: {2}", Nombre, Edad,
                        Telefono);
        return (sb.ToString());
    }
}

static void Main(string[] args)
{
    //
    // TODO: agregar aquí código para iniciar la aplicación
    //
}

```



TUTORÍA DE ESTRUCTURAS

En la dirección web que listaremos a continuación, encontraremos una pequeña tutoría sobre la utilización de las estructuras en C#. Recordemos siempre buscar información, tutoriales y cualquier otro material que nos sea de utilidad, para poder ampliar nuestros conocimientos: www.devjoker.com/asp/ver_contenidos.aspx?co_contenido=161.

```

Agenda amigo=new Agenda("Juan",25,"(555) 123-4567");

Console.WriteLine(amigo.ToString());
}

```

Ejecutemos el programa y veamos el resultado que obtenemos:

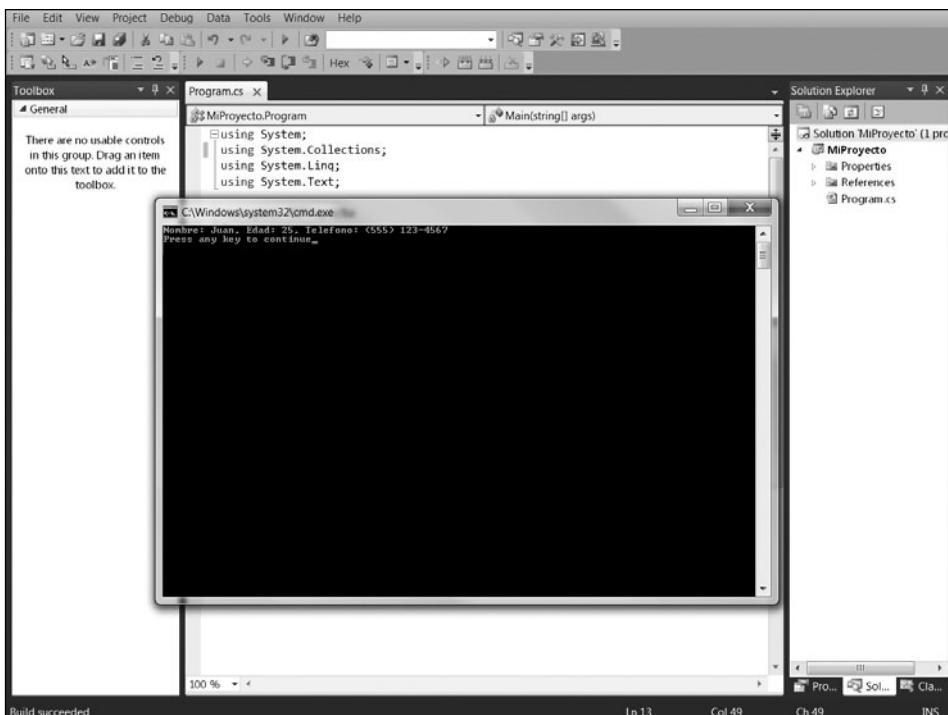


Figura 3. Vemos cómo efectivamente los datos han sido colocados en los campos de la variable por medio del constructor.

Cómo usar el constructor para validar información

El constructor no solamente puede ser utilizado para colocar la información adentro de los campos, una de las grandes ventajas que éste nos da es la posibilidad de poder validar la información antes de que sea asignada a uno de sus campos. Es posible que existan ciertas reglas sobre lo que se considera información válida para nuestros campos, y por supuesto, podemos implementar estas mismas en el constructor.

Supongamos que el teléfono sólo se considera válido si tiene más de **8** caracteres. En caso de que sea menor es inválido y no debe ser asignado para evitar problemas. Esta simple regla es fácil de programar en el constructor. En nuestro caso podría quedar de la siguiente forma:

```

public Agenda(String pNombre, int pEdad, String
              pTelefono)
{
    // Llevamos a cabo la asignación
    Nombre=pNombre;
    Edad=pEdad;

    if(pTelefono.Length>8)
        Telefono=pTelefono;
    else
        Telefono="Teléfono no válido";
}

```

Como vemos, hacemos uso de **if** y verificamos la longitud de la cadena **pTelefono**. Si es mayor a **8** entonces le asignamos **pTelefono** al campo **Telefono**. En caso contrario, colocamos un mensaje que dice que el teléfono no es válido.

El programa completo es el siguiente:

```

public struct Agenda
{
    public String Nombre;
    public int Edad;
    public String Telefono;

    public Agenda(String pNombre, int pEdad, String
                  pTelefono)
    {
        // Llevamos a cabo la asignación
        Nombre=pNombre;

```



TUTORÍA DE ENUMERACIÓN

El mismo sitio web que nombramos anteriormente presenta una tutoría sobre el uso de la enumeración en C# que complementa los conceptos que hemos aprendido en este capítulo: www.devjoker.com/contenidos/Tutorial-C/164/Enumeraciones.aspx.

```

        Edad=pEdad;

        if(pTelefono.Length>8)
            Telefono=pTelefono;
        else
            Telefono="Telefono no valido";
    }

    public override String ToString()
    {
        StringBuilder sb = new StringBuilder();
        sb.AppendFormat("Nombre: {0}, Edad: {1},
                        Telefono: {2}", Nombre, Edad,
                        Telefono);
        return (sb.ToString());
    }

}

static void Main(string[] args)
{
    //
    // TODO: agregar aquí código para iniciar la aplicación
    //

    Agenda amigo=new Agenda("Juan",25,"(555) 123-4567");
    Agenda amigo1=new Agenda("Pedro",32,"(555)");
    Console.WriteLine(amigo.ToString());
    Console.WriteLine(amigo1.ToString());
}

```



TUTORÍA SOBRE EL CONSTRUCTOR EN LAS ESTRUCTURAS

Si deseamos ampliar los conceptos y conocimientos sobre el uso del constructor en las estructuras, podemos visitar las tutorías del sitio web Devjoker: www.devjoker.com/contenidos/Tutorial-C/163/Constructores-de-estructuras.aspx.

}

Si compilamos y ejecutamos el programa tendremos el siguiente resultado:

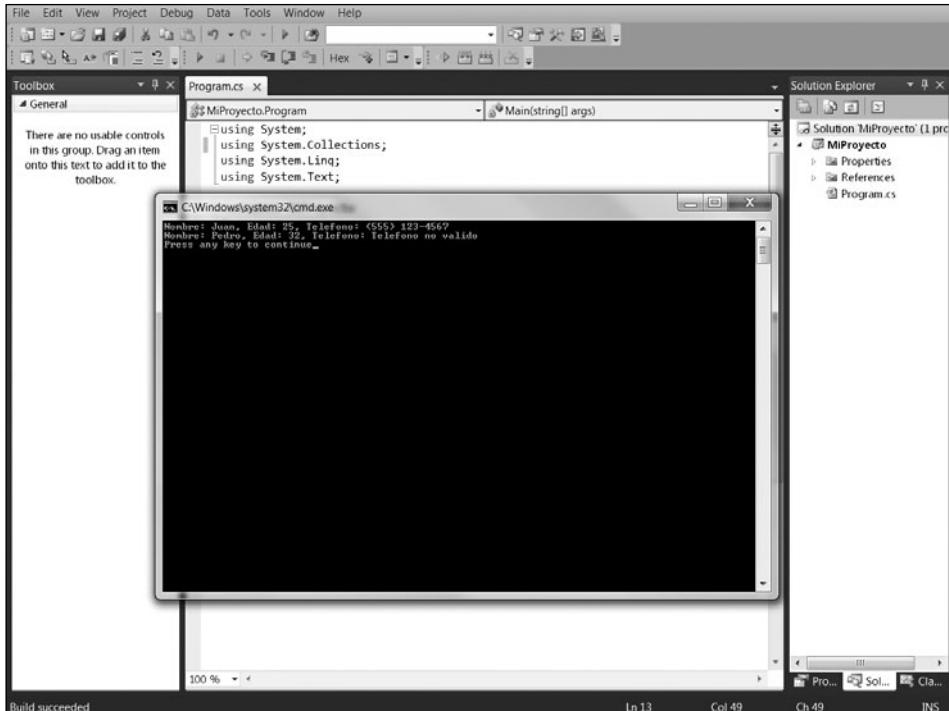


Figura 4. Podemos ver cómo el constructor no sólo ha asignado el valor sino que también ha validado la información.

La sobrecarga del constructor

El constructor puede ser sobrecargado y esta habilidad nos brinda mucha flexibilidad para poder utilizarlo. Supongamos que tenemos amigos que tienen teléfono y otros que no. Si no tienen teléfono, no tiene sentido utilizar un constructor que lo solicite. Lo mejor sería tener una segunda versión del constructor que solamente

III LA SOBRECARGA

Sobrecarga es una técnica de programación que permite tener varias versiones de una función o un método. El compilador selecciona la versión a utilizar basándose en la cantidad de parámetros y los tipos. Esto nos permite agrupar métodos diferentes bajo un mismo nombre o concepto, facilitando la programación, solamente recordando un nombre, y no los nombres para cada versión.

necesite el nombre y la edad, y podemos hacer que esta versión coloque un mensaje que indique que no tiene teléfono.

Para hacer la sobrecarga colocamos la segunda versión debajo de la que ya tenemos.

Por ejemplo, nuestra segunda versión queda de la siguiente forma:

```
public Agenda(String pNombre, int pEdad)
{
    // Llevamos a cabo la asignación
    Nombre=pNombre;
    Edad=pEdad;

    Telefono="Sin teléfono";
}
```

Como podemos ver, solamente recibimos dos parámetros, pero aun así los tres campos son asignados con un valor. Ningún campo se queda sin asignar. Para declarar la variable le pasamos los 2 parámetros necesarios.

```
Agenda amigo2=new Agenda("Luis",28);
```

Veamos un ejemplo que utilice la sobrecarga del constructor de la estructura:

```
public struct Agenda
{
    public String Nombre;
    public int Edad;
    public String Telefono;

    public Agenda(String pNombre, int pEdad, String
    pTelefono)
    {
        // Llevamos a cabo la asignación
        Nombre=pNombre;
        Edad=pEdad;
        if(pTelefono.Length>8)
```

```
        Telefono=pTelefono;
    else
        Telefono="Teléfono no válido";
    }

    public Agenda(String pNombre, int pEdad)
    {
        // Llevamos a cabo la asignación
        Nombre=pNombre;
        Edad=pEdad;

        Telefono="Sin teléfono";
    }

    public override String ToString()
    {

        StringBuilder sb = new StringBuilder();
        sb.AppendFormat("Nombre: {0}, Edad: {1},
                        Telefono: {2}", Nombre, Edad,
                        Telefono);
        return (sb.ToString());
    }

}

static void Main(string[] args)
{
    //
    // TODO: agregar aquí código para iniciar la aplicación
    //

    Agenda amigo=new Agenda("Juan",25,"(555) 123-4567");
    Agenda amigo1=new Agenda("Pedro",32,"(555)");
    Agenda amigo2=new Agenda("Luis",28);

    Console.WriteLine(amigo.ToString());
    Console.WriteLine(amigo1.ToString());
    Console.WriteLine(amigo2.ToString());
}
```

Si compilamos la aplicación, veremos el resultado en la siguiente figura:

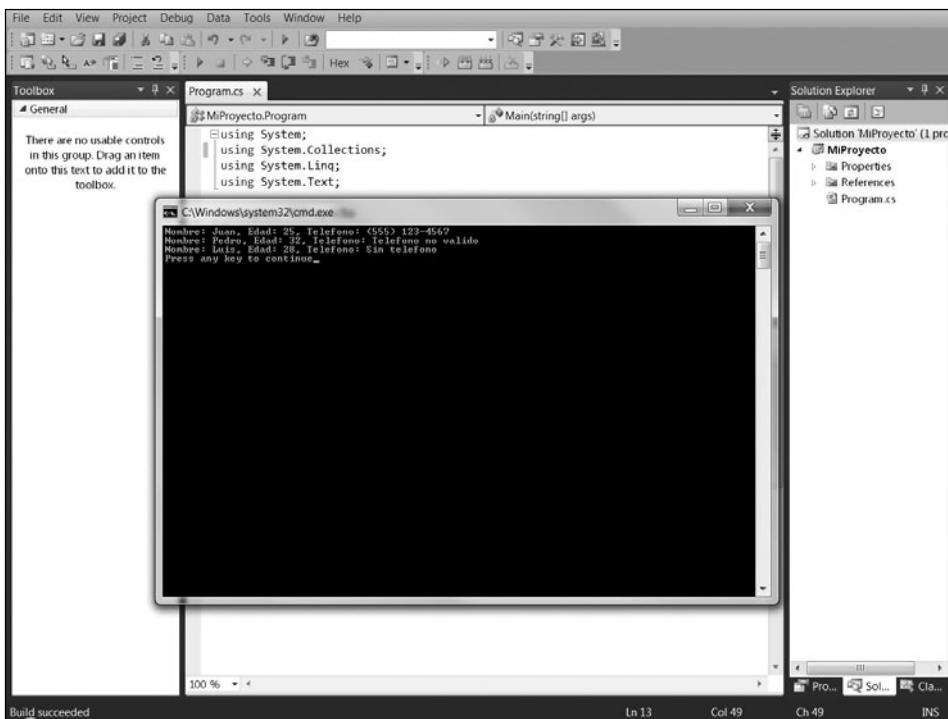


Figura 5. En esta figura podemos observar cómo el constructor sobrecargado también ha asignado la información necesaria en los campos.

Incluso podemos hacer que el constructor le solicite la información directamente al usuario. Hay que recordar que podemos colocar cualquier código válido en su interior. Esto lo logramos al crear una nueva sobrecarga.

Veamos cómo puede quedar esta nueva versión del constructor:

```
public Agenda(String pNombre)
{
    // Asignamos el nombre
    Nombre=pNombre;

    // Pedimos la edad
    Console.WriteLine("Dame la edad:");
    Edad=Convert.ToInt32(Console.ReadLine());
```

```
// Pedimos el teléfono
Console.WriteLine("Dame el teléfono:");
Telefono=Console.ReadLine();

if(Telefono.Length<8)
    Telefono="Sin teléfono";

}
```

Lo primero que hacemos es asignar el nombre que hemos recibido por parámetro. Luego de la forma usual pedimos la edad, notando cómo está colocada la sentencia. Sabemos que **ToInt32()** necesita un parámetro que es la cadena a convertir y **ReadLine()** regresa una cadena. De esta forma, aprovechamos y colocamos todo el código adentro de una sola línea. Otra opción que podríamos haber utilizado es una función que pida el dato, tal y como lo vimos en un capítulo anterior. Enseguida solicitamos el teléfono y aprovechamos para validar la información.

El programa completo queda de la siguiente manera:

```
public struct Agenda
{
    public String Nombre;
    public int Edad;
    public String Telefono;

    public Agenda(String pNombre, int pEdad, String
                  pTelefono)
    {
        // Llevamos a cabo la asignación
        Nombre=pNombre;
        Edad=pEdad;
    }
}
```

III CONSTRUCTOR EN LA ESTRUCTURA

Cuando utilizamos el constructor con nuestra estructura o cuando tenemos sobrecargas, tenemos que tener cuidado con la cantidad de parámetros. Dentro de las estructuras el constructor necesita tener al menos un parámetro ya que no es válido colocar un constructor sin parámetros. Si olvidamos colocar los parámetros, tendremos un error durante la compilación del programa.

```

        if(pTelefono.Length>8)
            Telefono=pTelefono;
        else
            Telefono="Teléfono no valido";
    }

public Agenda(String pNombre, int pEdad)
{
    // Llevamos a cabo la asignación
    Nombre=pNombre;
    Edad=pEdad;

    Telefono="Sin teléfono";
}

public Agenda(String pNombre)
{
    // Asignamos el nombre
    Nombre=pNombre;

    // Pedimos la edad
    Console.WriteLine("Dame la edad:");
    Edad=Convert.ToInt32(Console.ReadLine());

    // Pedimos el teléfono
    Console.WriteLine("Dame el teléfono:");
    Telefono=Console.ReadLine();

    if(Telefono.Length<8)
        Telefono="Sin teléfono";
}

```

III EL MODIFICADOR OVERRIDE

Si deseamos conocer más sobre el modificador **override** en C#, podemos hacerlo directamente en el sitio MSDN. Este sitio de Microsoft siempre es una muy buena referencia y podemos encontrar toda la información que necesitemos. La dirección electrónica es: [http://msdn2.microsoft.com/es-es/library/ebca9ah3\(VS.80\).aspx](http://msdn2.microsoft.com/es-es/library/ebca9ah3(VS.80).aspx).

```

    }

    public override String ToString()
    {
        StringBuilder sb = new StringBuilder();
        sb.AppendFormat("Nombre: {0}, Edad: {1},
                        Teléfono: {2}", Nombre, Edad,
                        Teléfono);
        return (sb.ToString());
    }

}

static void Main(string[] args)
{
    //
    // TODO: agregar aquí código para iniciar la aplicación
    //

    Agenda amigo=new Agenda("Juan",25,"(555) 123-4567");
    Agenda amigo1=new Agenda("Pedro",32,"(555)");
    Agenda amigo2=new Agenda("Luis",28);
    Agenda amigo3=new Agenda("Maria");

    Console.WriteLine(amigo.ToString());
    Console.WriteLine(amigo1.ToString());
    Console.WriteLine(amigo2.ToString());
    Console.WriteLine(amigo3.ToString());
}

```

Compilemos la aplicación y veamos cómo se comporta.



INFORMACIÓN SOBRE ESTRUCTURAS EN MSDN

En MSDN también encontraremos información sobre las estructuras, sus constructores, campos y demás elementos que las pueden constituir. También encontramos una referencia a su guía de programación. La encontraremos en la siguiente dirección:

[http://msdn2.microsoft.com/es-es/library/ah19swz4\(VS.80\).aspx](http://msdn2.microsoft.com/es-es/library/ah19swz4(VS.80).aspx).

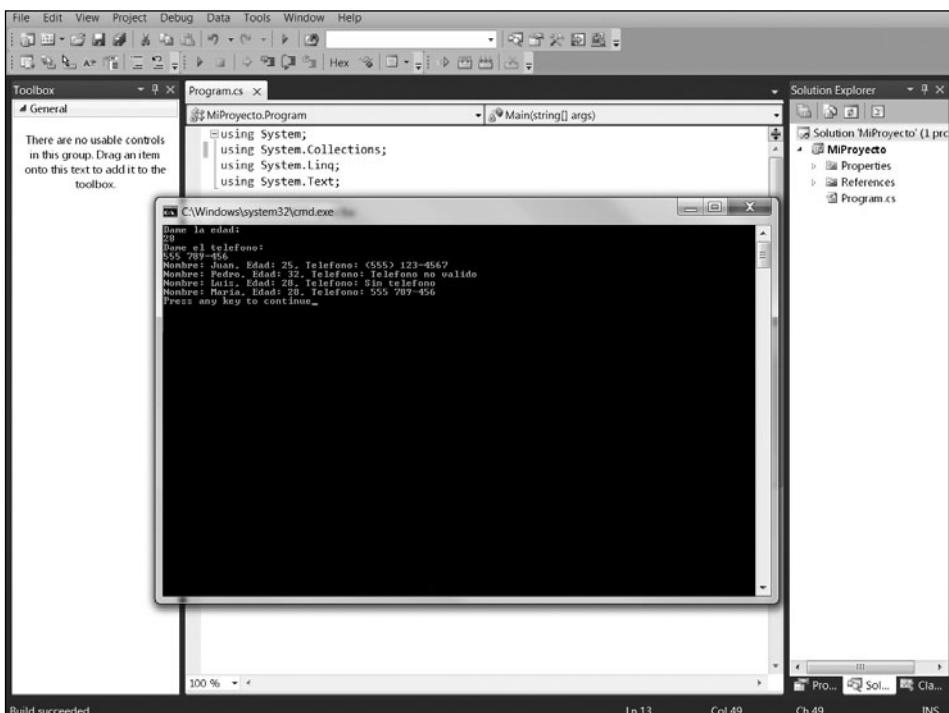


Figura 6. Vemos que el constructor efectivamente nos pide los datos por medio de la consola y luego son asignados.

Estructuras enlazadas

Las estructuras nos dan la facilidad de poder agrupar información que está relacionada adentro de un mismo elemento. Ya vimos que se puede colocar como dato a cualquier tipo que sea válido en C#. Estos tipos válidos también corresponden a tipos definidos por el programador, es decir otras estructuras.

Tenemos una estructura que se llama **Agenda**, pero esta agenda está un poco incompleta, debido a que nos hace falta colocar la dirección o el domicilio de la persona. Esto lo podemos resolver de una manera práctica, colocando dos campos nuevos como la calle y el número de la casa, pero también podríamos pensar en otra estructura llamada **Direccion** que contenga esta información.

Por ejemplo, la estructura podría quedar de la siguiente forma:

```
public struct Direccion
{
    public String Calle;
```

```
public int Numero;

public Direccion(String pCalle, int pNumero)
{
    Calle=pCalle;
    Numero=pNumero;
}
}
```

Como vemos, los campos son **Calle** y **Numero**. Se ha creado un constructor para ayudarnos a colocar la información fácilmente cuando se instancie una variable de este nuevo tipo. Ahora que ya tenemos definido el nuevo tipo **Direccion**, podemos usarlo en nuestra estructura **Agenda**, pero serán necesarios algunos cambios.

Primero veamos cómo agregar un campo de tipo **Direccion** a la estructura:

```
public struct Agenda
{
    public String Nombre;
    public int Edad;
    public String Telefono;
    public Direccion Domicilio;
```

Agregar un campo es muy sencillo, simplemente definimos la variable. Sin embargo, si dejamos la estructura con ese cambio tendremos problemas, ya que el campo **Domicilio** no estará instanciado. Esta instanciación se puede llevar a cabo cuando se asignan los datos, por ejemplo, en el constructor.

Cambiemos el primer constructor para instanciar y asignarle datos al campo **Domicilio**:

```
public Agenda(String pNombre, int pEdad, String
              pTelefono, String pCalle, int pNumero)
{
    // Llevamos a cabo la asignación
    Nombre=pNombre;
    Edad=pEdad;

    if(pTelefono.Length>8)
```

```

    Telefono=pTelefono;
else
    Telefono="Teléfono no valido";

Domicilio=new Direccion(pCalle, pNumero);

}

```

Vemos que hemos llevado a cabo algunos cambios. En primer lugar, el constructor de **Agenda** recibe todos los datos necesarios, incluidos los de la estructura **Direccion**. En el interior del constructor asignamos los datos que le corresponden a **Agenda** y en la parte final hacemos la instanciación de **Domicilio** por medio de **new** y le pasamos a su constructor los datos que le corresponden. También puede suceder que el constructor de **Agenda** no reciba los datos de **Domicilio**. En ese caso, debemos instanciar también al campo domicilio y pasar algunos datos de default al constructor.

```

public Agenda(String pNombre, int pEdad)
{
    // Llevamos a cabo la asignación
    Nombre=pNombre;
    Edad=pEdad;

    Telefono="Sin teléfono";

    Domicilio=new Direccion("Sin dirección",0);
}

```

En este caso, vemos que la instanciación se lleva a cabo, pero simplemente pasamos algunos parámetros de default. Si continuamos con estos cambios, el programa quedaría de la siguiente manera:

```

public struct Direccion
{
    public String Calle;
    public int Numero;

    public Direccion(String pCalle, int pNumero)
    {

```

```

        Calle=pCalle;
        Numero=pNumero;
    }

}

public struct Agenda
{
    public String Nombre;
    public int Edad;
    public String Telefono;
    public Direccion Domicilio;

    public Agenda(String pNombre, int pEdad, String
                  pTelefono, String pCalle, int pNumero)
    {
        // Llevamos a cabo la asignación
        Nombre=pNombre;
        Edad=pEdad;

        if(pTelefono.Length>8)
            Telefono=pTelefono;
        else
            Telefono="Teléfono no valido";

        Domicilio=new Direccion(pCalle, pNumero);
    }

    public Agenda(String pNombre, int pEdad)
    {
        // Llevamos a cabo la asignación
        Nombre=pNombre;
        Edad=pEdad;

        Telefono="Sin teléfono";

        Domicilio=new Direccion("Sin dirección",0);
    }

    public Agenda(String pNombre)

```

```
{  
    // Asignamos el nombre  
    Nombre=pNombre;  
  
    // Pedimos la edad  
    Console.WriteLine("Dame la edad:");  
    Edad=Convert.ToInt32(Console.ReadLine());  
  
    // Pedimos el teléfono  
    Console.WriteLine("Dame el teléfono:");  
    Telefono=Console.ReadLine();  
  
    if(Telefono.Length<8)  
        Telefono="Sin teléfono";  
  
    Domicilio=new Direccion("Sin dirección",0);  
  
}  
  
public override String ToString()  
{  
    StringBuilder sb = new StringBuilder();  
    sb.AppendFormat("Nombre: {0}, Edad: {1},  
    Teléfono: {2}", Nombre, Edad,  
    Telefono);  
    return (sb.ToString());  
}  
  
}  
  
static void Main(string[] args)  
{  
    //  
    // TODO: agregar aquí código para iniciar la aplicación  
    //  
  
    Agenda amigo=new Agenda("Juan",25,"(555) 123-4567","Av.  
    Principal",105);
```

```

Agenda amigo1=new Agenda("Pedro",32,"(555)", "Calle
Bolivar",350);
Agenda amigo2=new Agenda("Luis",28);
Agenda amigo3=new Agenda("Maria");

Console.WriteLine(amigo.ToString());
Console.WriteLine(amigo1.ToString());
Console.WriteLine(amigo2.ToString());
Console.WriteLine(amigo3.ToString());
}

```

Podemos observar que en las variables **amigo** y **amigo1** ya hemos colocado previamente los parámetros necesarios por el constructor modificado. Sólo queda que compilemos y ejecutemos el programa.

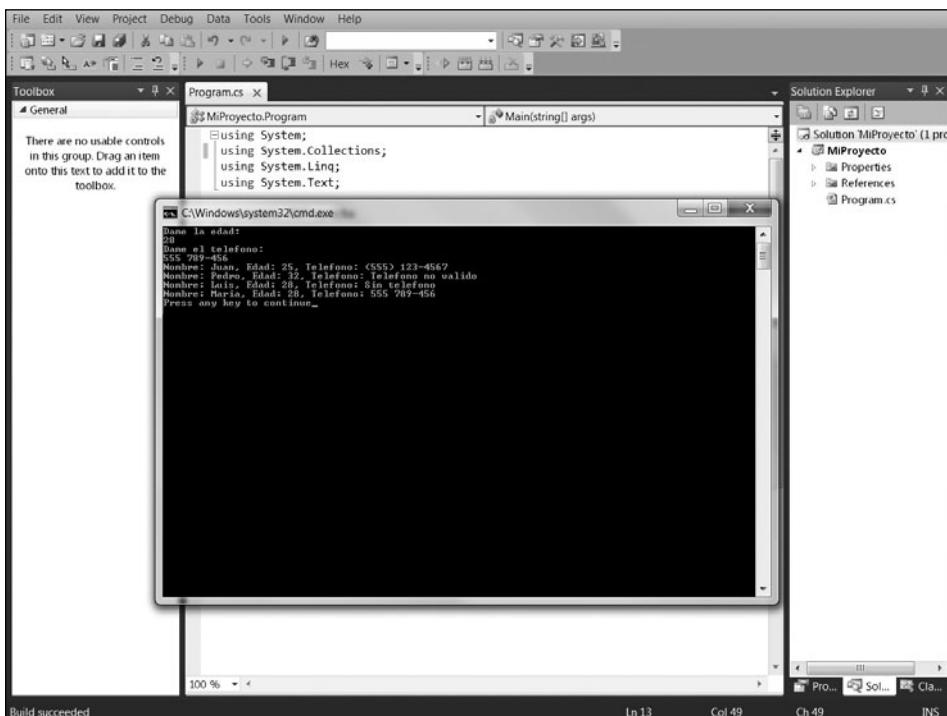


Figura 7. Vemos que el resultado del programa es el mismo que el anterior.

Al parecer tenemos un problema ya que obtenemos el mismo resultado que el programa anterior a pesar de los cambios. Esto sucede debido a que no hemos actualizado el método **ToString()** de **Agenda** ni creado el propio de **Direccion**.

Para **Direccion**:

```
public override String ToString()
{
    StringBuilder sb = new StringBuilder();
    sb.AppendFormat("Direccion: {0} #{1}", Calle,
    Numero);
    return (sb.ToString());
}
```

Para **Agenda**:

```
public override String ToString()
{
    StringBuilder sb = new StringBuilder();
    sb.AppendFormat("Nombre: {0}, Edad: {1},
    Telefono: {2}", Nombre, Edad,
    Telefono);

    // Adicionamos la cadena que viene de
    // Domicilio
    sb.Append(Domicilio.ToString());
    return (sb.ToString());
}
```

Vemos que adentro del **ToString()** de **Agenda** agregamos la cadena que nos regresa el **ToString()** de **Domicilio**. De esta forma, cuando se impriman los contenidos de **Agenda** también se imprimirán los del campo **Domicilio**. El programa completo queda de la siguiente forma:

```
public struct Direccion
{
    public String Calle;
    public int Numero;

    public Direccion(String pCalle, int pNumero)
    {
        Calle=pCalle;
        Numero=pNumero;
```

```
    }

    public override String ToString()
    {
        StringBuilder sb = new StringBuilder();
        sb.AppendFormat("Direccion: {0} #{1}", Calle,
                       Numero);
        return (sb.ToString());
    }
}

public struct Agenda
{
    public String Nombre;
    public int Edad;
    public String Telefono;
    public Direccion Domicilio;

    public Agenda(String pNombre, int pEdad, String
                  pTelefono, String pCalle, int pNumero)
    {
        // Llevamos a cabo la asignación
        Nombre=pNombre;
        Edad=pEdad;

        if(pTelefono.Length>8)
            Telefono=pTelefono;
        else
            Telefono="Teléfono no valido";

        Domicilio=new Direccion(pCalle, pNumero);
    }

    public Agenda(String pNombre, int pEdad)
    {
        // Llevamos a cabo la asignación
        Nombre=pNombre;
        Edad=pEdad;

        Telefono="Sin teléfono";
    }
}
```

```
Domicilio=new Direccion("Sin dirección",0);
}

public Agenda(String pNombre)
{
    // Asignamos el nombre
    Nombre=pNombre;

    // Pedimos la edad
    Console.WriteLine("Dame la edad:");
    Edad=Convert.ToInt32(Console.ReadLine());

    // Pedimos el telefono
    Console.WriteLine("Dame el teléfono:");
    Telefono=Console.ReadLine();

    if(Telefono.Length<8)
        Telefono="Sin teléfono";

    Domicilio=new Direccion(" Sin dirección");
}

public override String ToString()
{
    StringBuilder sb = new StringBuilder();
    sb.AppendFormat("Nombre: {0}, Edad: {1},
                    Telefono: {2}", Nombre, Edad,
                    Telefono);

    // Adicionamos la cadena que viene de
    // Domicilio
    sb.Append(Domicilio.ToString());
    return (sb.ToString());
}

static void Main(string[] args)
```

```

{
    //
    // TODO: agregar aquí código para iniciar la aplicación
    //

    Agenda amigo=new Agenda("Juan",25,"(555) 123-4567","Av.
        Principal",105);
    Agenda amigo1=new
        Agenda("Pedro",32,"(555)","Bolivar",350);
    Agenda amigo2=new Agenda("Luis",28);
    Agenda amigo3=new Agenda("Maria");

    Console.WriteLine(amigo.ToString());
    Console.WriteLine(amigo1.ToString());
    Console.WriteLine(amigo2.ToString());
    Console.WriteLine(amigo3.ToString());
}

```

Ahora ejecutemos el programa y veamos el resultado.

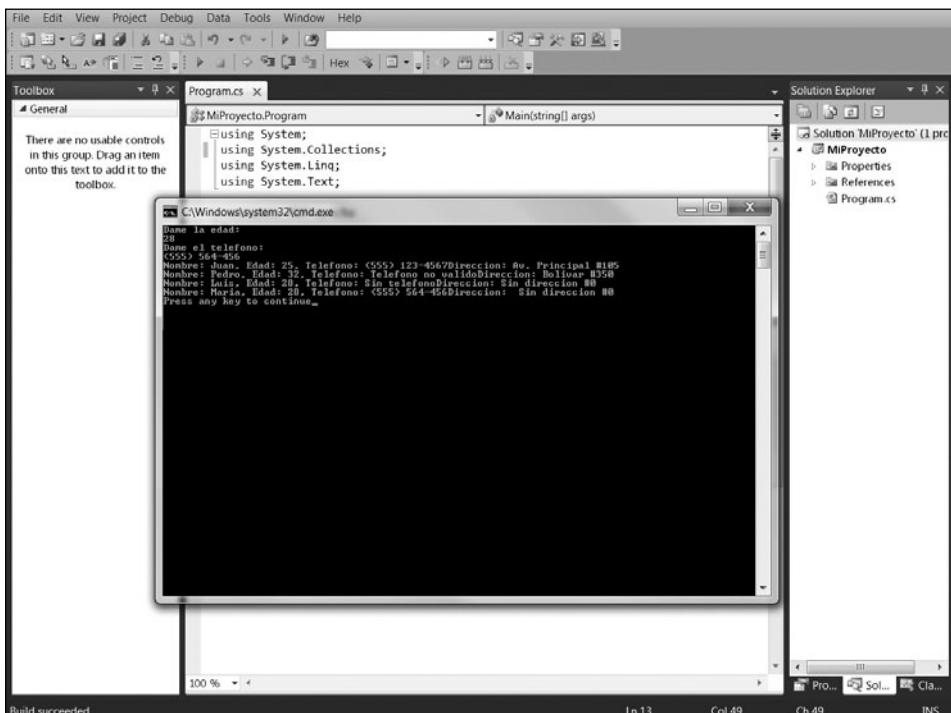


Figura 8. Observemos que en este caso la información se presenta completa.

Las enumeraciones

Ya hemos aprendido las estructuras y ahora conoceremos otro nuevo tipo que puede ser definido por el programador. Este nuevo tipo se conoce como enumeraciones. Éstas son tipos que nosotros podemos definir, pero tendrán un número finito de valores posibles, es decir, que podemos definir el tipo y los valores a guardar. Las enumeraciones son útiles cuando tenemos información que sabemos que solamente puede tener uno de una serie de posibles valores. Por ejemplo, sabemos que la semana tiene siete días. Entonces, un día no puede tener un valor que no sea el de un día de la semana. La semana sería una enumeración.

Esto lo podemos utilizar para cualquier otro ejemplo donde sepamos los valores posibles a utilizar. Otro ejemplo, supongamos que estamos haciendo un programa para una empresa que vende quesos. Tendremos un número finito de posibles tipos de queso. Entonces, podemos colocarlos adentro de una enumeración.

Al parecer, las enumeraciones son muy sencillas, pero podemos correr el riesgo de pensar que no son muy útiles. Sin embargo, nos dan grandes beneficios. También nos sirven para reducir la cantidad de errores que podemos tener en nuestra aplicación al asignar y utilizar valores. Como beneficio, reducen la lógica que debemos utilizar e incluso mejoran el desempeño de nuestra aplicación. Por eso es útil aprender a usarlas correctamente.

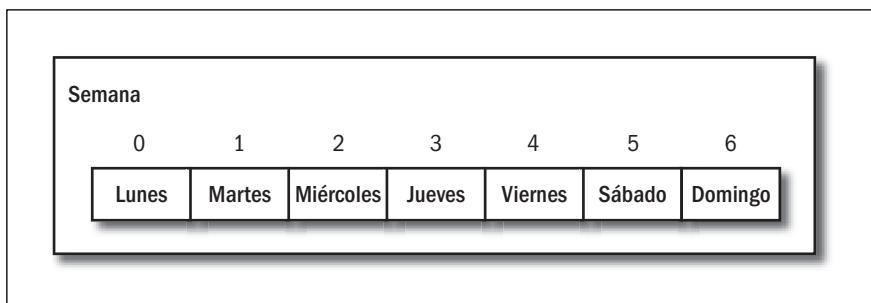


Figura 9. Aquí vemos una representación de la enumeración y sus posibles valores.

Veamos un ejemplo donde podemos utilizar la enumeración para mejorar nuestra aplicación. Si nuestro programa necesita guardar por algún motivo el día de la semana podemos pensar en diferentes opciones. La primera puede ser la utilización de una variable de tipo entero, ya que cada día de la semana puede ser definido por un número del 1 al 7. Sin embargo, esto nos presenta algunos problemas. El primer problema es que usamos un tipo que puede guardar miles de valores posibles, pero sólo usamos 7, entonces tenemos un desperdicio de memoria.

El segundo problema es que el tipo, al poder guardar muchos valores, puede guardar datos que no correspondan a la realidad como el día 8 de la semana. Para evitar esto necesitamos agregar lógica de control con el uso de uno o varios **if**. De

esta manera, se complica nuestra aplicación. Otro problema que esto nos presenta es el despliegue de la información. Cada vez que queremos mostrar el día tenemos que poner lógica para transformar **1** a “**Lunes**”, **2** a “**Martes**”, y así sucesivamente para poder listar cada día de la semana.

Quizás otra solución sería el uso de una cadena para guardar el día de la semana, pero también presenta problemas. Un problema que podemos tener es que si nuestra lógica funciona con “**Lunes**”, cuando el usuario introduzca un valor como “**lunes**” ya no funcionará. Para corregir esto podemos usar el método **ToUpper()**, pero agrega más lógica y proceso extra a nuestra aplicación que no es necesario. También gastamos memoria de más ya que guardar una cadena requiere más memoria que guardar un simple número. Desde luego, la impresión sería más sencilla, pero la administración de la información más complicada.

Nuestra mejor opción es el uso de las enumeraciones. Ahora aprenderemos cómo usarlas y cómo el problema anterior se resolvería fácilmente con ellas.

Declaración y asignación de valores

Para declarar una enumeración usamos el siguiente formato:

```
enum nombre {valor1, valor2, ..., valorn};
```

En primer lugar, tenemos que colocar la palabra clave **enum** seguida del nombre con el que identificaremos al nuevo tipo de enumeración. Luego entre **{}** debemos colocar los valores que puede tener este tipo. Hay que recordar que los valores son finitos.

Veamos cómo declarar una enumeración para nuestro ejemplo de la semana:

```
enum semana {Lunes, Martes, Miercoles, Jueves, Viernes, Sabado, Domingo};
```

En este caso, hemos declarado la enumeración con el nombre de semana. Cada uno de los días que hemos listado entre **{}** serán los posibles valores que puede tener. Con

III LOS VALORES DE LAS ENUMERACIONES

Cuando declaramos la enumeración llevamos a cabo el listado de todos los valores posibles. Podemos pensar que internamente se guardan con un número de identificación. El primer elemento tendrá el valor de cero, el segundo de uno y así en forma consecutiva. Sin embargo, también es posible indicar cuál será el valor para el primer elemento de la enumeración.

esto, el siguiente paso sería simplemente la creación de una variable de tipo semana. La declaración y asignación de la variable la hacemos de la forma tradicional.

```
semana miDia;
```

Aquí declaramos la variable **miDia** que es de tipo **semana**, por lo que solamente podrá tener los valores listados en la enumeración. Si lo hubiéramos deseado, también podríamos haber declarado e inicializado la variable con algún valor.

```
semana miDia=semana.Martes;
```

La asignación del valor en esta variable se puede llevar a cabo en cualquier momento, dentro de nuestra aplicación, después de su declaración, como en cualquier otra variable, pero con uno de los valores contenidos en semana.

```
miDia=semana.Viernes;
```

Para imprimir el valor de una enumeración

Si deseamos mostrar el valor de una variable de tipo enumeración, lo único que necesitamos es utilizar el método **ToString()**. Éste nos dará una cadena con el valor correspondiente y podemos utilizarla dentro de nuestra lógica para desplegar valores. El método **ToString()** regresa una cadena y esa cadena será el valor correspondiente. Por ejemplo, si en la enumeración listamos un valor como **Lunes**, entonces la cadena regresada será “**Lunes**”.

```
String mensaje="El dia es ";
...
...
mensaje=mensaje+miDia.ToString();
```

Otro mecanismo que podemos utilizar para desplegar la información de una variable de tipo enumeración es la cadena de formato.

En este caso, simplemente colocamos la variable en nuestra lista de variables:

```
Console.WriteLine("El dia es {0}", miDia);
```

Para pasar de enumeración a valor numérico

Si necesitamos pasar el valor de la enumeración a una variable numérica, lo único que necesitamos hacer es colocar un **type cast**. En nuestro ejemplo, **Lunes** al ser el primer valor definido, se pasará como **0** a la variable numérica, **Miercoles** será el valor **2** y así sucesivamente.

```
int valor=0;  
...  
...  
valor=(int)miDia;
```

Para iniciar la enumeración con un valor definido

Como vimos antes, el primer valor recibe el índice **0**, pero en algunos casos es posible que necesitemos que nuestra enumeración tenga un rango de valores especiales. La forma como resolvemos esto es dándole al primer elemento de la enumeración el valor que necesitamos y todos los demás elementos tendrán valores sucesivos a éste.

Por ejemplo, supongamos que para nuestros propósitos en la enumeración de la semana, el día **Lunes** debe tener el valor **3**. Para esto colocamos lo siguiente:

```
enum semana {Lunes=3, Martes, Miercoles, Jueves, Viernes, Sabado, Domingo};
```

De esta forma, **Lunes** tendrá un valor de **3**, **Martes** de **4** y así sucesivamente hasta **Domingo** con un valor de **9**.

La enumeración en expresiones

Como nosotros trabajamos con una variable de tipo enumeración, podemos utilizarla en cualquier tipo de expresión. Sólo debemos ser cuidadosos con los tipos con los que interactuamos en la expresión. Si fuera necesario, podemos hacer uso de **type cast** para que la expresión quede correctamente escrita.

III IMPORTANCIA EN EL APRENDIZAJE

Puede que en muchas oportunidades, a lo largo de los capítulos del libro, hayamos encontrado algunas trabas, o ejercicios con cierta complejidad. el caso de este capítulo puede llegar a ser uno. La recomendación es volver a repasar cada uno de los temas en los que hemos encontrado estas trabas para lograr una mejor comprensión.

Por ejemplo, para una expresión aritmética podemos realizar lo siguiente:

```
int salario=0;
int pagoDia=200;
...
...
salarion=pagoDia*((int)miDia);
```

En las expresiones aritméticas es muy común que usemos **type cast**. Si lo deseamos, podemos hacer uso de los paréntesis para ayudarnos a leer la expresión. En una expresión relacional también podemos hacer uso de la enumeración. Si comparamos hacia el mismo tipo no es necesario hacer uso de un type cast.

```
if(miDia == semana.Martes)
```

Pero en el caso que necesitemos realizar una comparación hacia otro tipo, entonces sí será conveniente hacer utilización de type cast.

```
if((int)miDia==3)
```

Dentro de una expresión lógica podemos hacer uso de algo similar:

```
if((miDia>semana.Lunes && (int)miDia<5)
```

Ejemplo de aplicación

Ahora que ya conocemos los conceptos más importantes de las enumeraciones, es momento de hacer una aplicación. Nuestra aplicación será sencilla y simplemente probará los conceptos que hemos aprendido.

Nuestra aplicación puede quedar de la siguiente manera:

```
// Declaramos la enumeración
enum semana {Lunes, Martes, Miercoles, Jueves, Viernes,
             Sabado, Domingo};
enum colores { Rojo = 1, Verde, Azul, Amarillo };

static void Main(string[] args)
```

```
{  
    int numerico = 0;  
    // Creamos una variable de tipo semana  
  
    semana miDia;  
  
    // Asignamos un valor  
    miDia = semana.Lunes;  
  
    // Pasamos de enumeración a entero  
    numerico = (int)miDia;  
  
    // Mostramos la información  
    Console.WriteLine("El dia es {0} con valor {1}", miDia,  
        numerico);  
  
    // Creamos una variable de color  
    colores miColor = colores.Rojo;  
  
    // Pasamos a número entero  
    numerico = (int)miColor;  
  
    // Mostramos la información  
    Console.WriteLine("El color es {0} con valor {1}", miColor,  
        numerico);  
  
}
```

En primer lugar, declaramos dos enumeraciones. La primera es **semana** y **Lunes** tendrá como valor **0**. La segunda enumeración se llama **colores** y la iniciaremos con un valor inicial **Rojo** que será **1**.

Dentro de la función **Main()** declaramos una variable de tipo entero, que usaremos para nuestros experimentos de convertir las enumeraciones a valores numéricos. Posteriormente, declaramos una variable de tipo **semana** llamada **miDia**. En la línea siguiente le damos el valor de **Lunes** contenido en la enumeración **semana**. A continuación convertimos **miDia** a entero y se lo asignamos a la variable **numérico**. Luego mostramos los valores de las variables. Creamos una variable de tipo **colores** e inmediatamente le asignamos el valor de **Rojo** de la enumeración **colores**. La convertimos a valor numérico y luego la mostramos. Si ejecutamos el programa, obtendremos la siguiente salida en la consola:

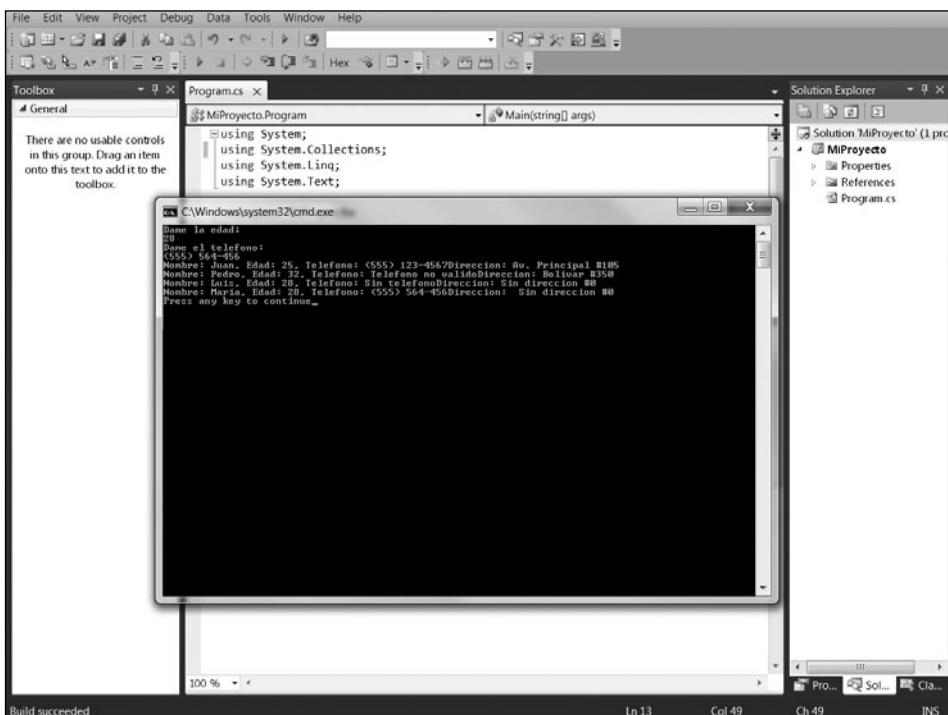


Figura 10. Podemos ver cómo la variable tiene el valor de la enumeración y su equivalente numérico.

Con esto ya hemos aprendido a lo largo de este capítulo, en complemento con otros, las bases del manejo de las estructuras y las enumeraciones. Su uso es recomendable y debemos utilizarlas cuando tengamos programas grandes ya que nos ayudan a administrar la información y reducir la cantidad de errores.

RESUMEN

Las estructuras nos permiten crear un tipo nuevo y agrupar información que está relacionada adentro de una unidad. Los datos adentro de la estructura se llaman campos. También es posible crear variables del tipo estructura y poder asignarles información a sus campos. El constructor nos ayuda a introducir la información original y a verificar que sea válida. Las estructuras pueden estar enlazadas. En las enumeraciones nosotros creamos un tipo que tendrá un número finito de valores posibles. Podemos darle un valor inicial a la enumeración e incluso convertirla a valores numéricos.



ACTIVIDADES

TEST DE AUTOEVALUACIÓN

1 ¿Qué es una estructura?

2 ¿Cómo accedemos a un campo de la estructura?

3 ¿Cómo creamos un constructor para la estructura?

4 ¿Para qué sirve un constructor?

5 ¿Cómo se usa la sobrecarga del constructor?

6 ¿Qué son las estructuras enlazadas?

7 ¿Cómo se define una variable de la estructura?

8 ¿Cómo podemos usar el método ToString() en la estructura?

9 ¿Qué es una enumeración?

10 ¿Cómo se declara una enumeración?

11 ¿Cómo creamos una variable de enumeración?

12 ¿De qué forma se le coloca un valor al elemento de la enumeración?

EJERCICIOS PRÁCTICOS

1 Crear una estructura para guardar los productos de una tienda.

2 Crear una estructura para guardar la información de una cuenta bancaria.

3 Crear estructuras enlazadas para guardar la información de una mascota y su dueño.

4 Crear una enumeración para los diferentes tipos de neumáticos.

5 Crear una enumeración para los diferentes tipos de estrellas.

Cómo crear nuestras propias clases

En este capítulo empezaremos a ver temas más avanzados. Hasta ahora hemos utilizado C# para la programación estructurada, pero ahora iremos hacia otro paradigma de programación, el de la orientación a objetos. C# es un lenguaje orientado a objetos, por lo que se puede obtener su mayor poder y beneficio si programamos con ese paradigma. Parte fundamental de la programación orientada a objetos es la clase.

La programación orientada a objetos	318
Las clases	318
Cómo declarar la clase y los datos	320
Cómo crear la instancia de nuestra clase	324
Cómo asignarles valores a datos públicos	324
Cómo invocar métodos de un objeto	325
Cómo imprimir un dato público	325
Protección de datos y creación de propiedades	329
Cómo acceder a las propiedades	333
Métodos públicos y privados	333
Convertir un objeto a cadena	334
El constructor en las clases	336
Sobrecarga del constructor	339
Resumen	347
Actividades	348

La programación orientada a objetos

La programación estructurada es muy sencilla e intuitiva. En ella simplemente tenemos un problema y lo subdividimos en problemas cada vez más pequeños. Usamos funciones para colocar zonas especializadas de código o código que se usa constantemente en su interior.

El paradigma estructurado es correcto, pero tiene algunas limitaciones y algunos problemas cuando el programa a realizar es grande. Unos de estos problemas se conoce como la corrupción de información. Supongamos que tenemos una variable y ésta es utilizada en diversas partes del programa. Una forma sencilla para poder tener acceso a la variable es declararla de forma global, así todo el programa la conoce. Al hacer esto cualquier parte del programa no sólo puede leerla sino también modificarla. Debido a este esquema, una parte del programa puede cambiarse sin que otra sepa de este cambio y así se producen errores en la información. Para evitar esto es posible pasarle la variable como parámetro a quien la necesite, pero esto complica su administración.

Los programas estructurados muy grandes también son difíciles de extender y mantener, y llevar a cabo un cambio puede ser costoso. Generalmente, un cambio en una parte del programa produce cambios en otras partes que quizás no estén relacionadas directamente. En general, los programas estructurados son poco flexibles.

La programación estructurada es buena para los programas pequeños y para aprender las bases de programación. Sin embargo, el mundo actual pide el desarrollo orientado a objetos. Como recomendación, después de este libro, podemos aprender análisis y diseño orientado a objetos y luego continuar aprendiendo más C# con todas las técnicas orientadas a objetos. Esto ya sería un nivel avanzando - experto de programación.

En la programación orientada a objetos tomamos una idea diferente en la resolución de los problemas. En lugar de subdividirlos, lo que hacemos es ver cuáles son los componentes u objetos que componen el problema y la forma cómo interactúan. Nosotros programaremos estos objetos y haremos que se comuniquen entre sí. Cuando los objetos hagan su labor y se comuniquen, el problema estará resuelto.

Las clases

El componente principal de la programación orientada a objetos es la **clase**. Nosotros podemos pensar en la clase como si fuera un plano por medio del cual podemos crear objetos, por ejemplo, pensemos en una casa. Para hacer una casa lo primero que hacemos es pensar en los cuartos que deseamos que tenga y luego diseñamos un plano. El plano no es la casa, ya que no podemos vivir ni actuar en él. Sin embargo, éste nos proporciona las características de la casa. Por medio del plano podemos construir la casa y en esta construcción sí podemos llevar a cabo nuestras actividades.

El plano sería equivalente a la clase y la casa construida al **objeto**. La clase es un pláno, una descripción, y el objeto tiene esas características y puede llevar a cabo trabajo concreto. Si necesitásemos hacer otra casa igual, no sería necesario hacer un nuevo pláno, simplemente tomaríamos el pláno ya realizado y crearíamos otra ca-sa. Una clase nos puede servir para crear muchos objetos independientes pero que tienen las mismas características. El proceso de crear un objeto a partir de una cla-se es lo que conocemos como **instanciación**.

Adentro de la clase, nosotros colocaremos información y más importante aún, los **métodos** que trabajan sobre esta información, es decir, que los **datos** y los métodos que los procesan están contenidos dentro de una sola unidad. A esto lo llamamos **encapsulamiento**. Al encapsular datos y métodos los protegemos contra la corrupción de información.

Los objetos se comunicarán por medio del uso de **mensajes**. En estos mensajes es po-sible solicitarle un dato a otro objeto, pedirle que lleve a cabo un proceso, etcétera.

Los datos

Los datos son la información con la que trabajará la clase. La clase solamente debe tener los datos que necesita para poder llevar a cabo su trabajo. Declarar un dato es muy similar a declarar una variable, pero al igual que en las estructuras, necesita-mos indicar el acceso ya que tenemos básicamente tres tipos de acceso: **público**, **pri-vado** y **protector**. Cuando nosotros tenemos un dato con acceso público cualquier elemento del exterior, como la función **Main()** o algún otro objeto, puede acceder al dato, leerlo y modificarlo. Cuando tenemos el acceso privado solamente los mé-todos definidos dentro de la clase podrán leerlo o modificarlo. El acceso protegido es un poco más avanzado y está por afuera de los límites de este libro.

Un punto muy importante con relación a los datos que no debemos olvidar es que los datos definidos en la clase son conocidos por todos los métodos de la misma cla-se. Es decir, actúan como si fueran globales para la clase. Cualquier método puede acceder a ellos directamente sin necesidad de que los pasemos como parámetro.

En algunos casos podremos colocar el acceso a nuestros datos como público, aun-que preferentemente no lo haremos. Si nos excedemos o usamos el acceso público en un mal diseño, corremos el riesgo de corrupción de información. Por lo general,



LA HERENCIA

Es una característica de la programación orientada a objetos. Ésta permite crear una nueva clase que hereda las características (**datos y métodos**) de otra clase, de forma tal que solamente tengamos que agregar los elementos necesarios para la nueva. Es una gran forma de reutilización de código si se usa en forma adecuada.

nuestros datos serán privados, aunque esto puede parecer un problema ya que si el exterior necesita alguna información calculada por el objeto no podrá tener acceso a ella. Para resolver esto hacemos uso de las funciones de interfaz.

Una **función de interfaz** es aquella que puede ser invocada desde el exterior y que regresa una copia del valor de algún dato dentro del objeto. También podemos usarla para colocar un valor determinado en un dato. La ventaja que nos da la función de interfaz es que podemos administrar el acceso a nuestra información, y podemos colocar dentro de ésta código de seguridad que verifique o valide la información que entra o sale. De esta forma evitamos la corrupción de información.

Los métodos

Los métodos son las funciones que llevan a cabo el proceso o la lógica de la clase, y crear un método dentro de la clase es muy parecido a la forma que hemos utilizado anteriormente. Los métodos también tendrán un tipo de acceso, al igual que los datos. Trabajarán sobre los datos de la clase. No hay que olvidar que todos los métodos conocen todos los datos definidos dentro de la clase, y pueden recibir parámetros y regresar valores. A un dato definido dentro de la clase no necesitamos pasarlo como parámetro ya que el método lo conoce. Solamente los métodos que necesiten ser invocados desde el exterior deben tener acceso público. Si el método sólo se invoca desde el mismo interior de la clase su acceso debe ser privado. Esto lo hacemos con fines de seguridad y para mantener el encapsulamiento correctamente.

Cómo declarar la clase y los datos

La declaración de la clase es un proceso sencillo. Las clases se declaran dentro de un **namespace** y cualquiera que tenga acceso a ese namespace puede crear objetos de la clase. No olvidemos que la clase es como el plano y los objetos son realmente los que usamos para llevar a cabo el trabajo.

Para declarar la clase tendremos un esquema como el siguiente:

```
class nombre
{
    // datos
    ...
    ...
    // métodos
    ...
    ...
}
```

```
}
```

El **nombre** de la clase puede ser cualquier nombre válido dentro de C#. El nombre debe ser único para el namespace, es decir no podemos tener dos clases que se llamen igual adentro del mismo namespace. La clase necesita un **bloque de código** y en su interior llevamos a cabo las declaraciones de los elementos que la componen. Generalmente, declaramos primero los datos. Esto nos permite tener un programa organizado y luego facilita la lectura. Además, es posible declarar los métodos implementados antes.

La mejor forma de hacer esto es por medio de un ejemplo. Supongamos que crearemos un programa que calcula el área y el volumen de cubos y prismas rectangulares. Como en esta ocasión lo hacemos vía programación orientada a objetos, lo primero que hacemos es pensar en los objetos que componen el problema.

Los objetos son: **cubo** y **prisma rectangular**. Ahora que ya conocemos los objetos, debemos pensar en los datos y las operaciones que se realizan sobre éstos.

Para el cubo necesitaremos como datos primordiales la longitud de su lado, el área y su volumen. En el caso del prisma requerimos aún más datos, que son el ancho, el alto, el espesor, el área y el volumen.

Enseguida debemos pensar en las operaciones que se realizan sobre estos datos. El cubo es más sencillo de resolver, ya que solamente necesitamos dos métodos, uno llamado **CalculaArea()** y el otro llamado **CalculaVolumen()**. Para comprender mejor las clases y por motivos ilustrativos el prisma necesitará tres métodos. El primero se llamará **CalculaArea()**, el segundo **CalculaVolumen()** y el tercero **AreaRectangulo()**. Como el área del prisma es igual a la suma de los rectángulos de sus caras nos apoyaremos en este método para calcularla.

Podemos diagramar las clases si hacemos uso de **UML**. La clase del cubo quedará como se muestra en la siguiente figura:

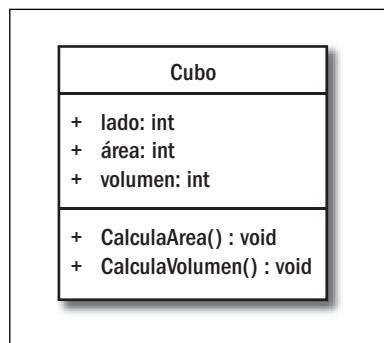


Figura 1. Éste es el diagrama de clases para el cubo.

Podemos ver todos los elementos que lo conforman.

De igual forma es posible crear el diagrama de clases correspondiente al prisma:

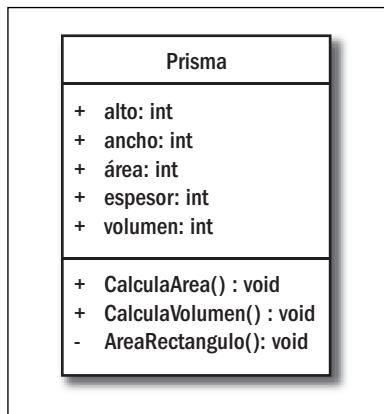


Figura 2. Esta figura nos muestra el diagrama de clases para el prisma.

Los diagramas anteriores se pueden leer fácilmente. Hay un rectángulo que representa a la clase. El rectángulo está dividido en tres secciones. La sección superior, que es utilizada para colocar el nombre de la clase, la sección intermedia, que se usa para indicar los datos que tendrá la clase y la sección inferior, que es para indicar cuáles son los métodos a utilizar.

El acceso se muestra por medio de los signos + o -. Cuando usamos el signo más estamos indicando que ese dato o método tiene un acceso del tipo público. En el caso de que coloquemos el signo menos, el acceso es del tipo privado.

El formato de los datos lleva primero el acceso seguido del nombre del dato. Luego se coloca : y el tipo que tiene este dato. El tipo puede ser nativo o definido por el programador. En el caso de los métodos, primero indicamos el acceso seguido del nombre del método. Como ninguno de nuestros métodos necesita parámetros, entonces los dejamos vacíos. El tipo del método, es decir el tipo de valor que regresa, se indica por medio de : seguido del tipo. En nuestro ejemplo hasta el momento ningún método regresa algo, por lo que todos son de tipo **void**.

Ahora podemos comenzar a declarar nuestras clases:

III EL USO DE UML

UML es un lenguaje unificado de modelado, un lenguaje visual que nos sirve para llevar a cabo diagramas y modelado de sistemas. Resulta muy útil en la programación y el diseño orientado a objetos, ya que facilita el diseño y la depuración de la aplicación aun antes de que se escriba una línea de código. Es recomendable buscar información adicional sobre éste y aprenderlo.

```

class cubo
{
    // Declaramos los datos
    public int lado;
    public int area;
    public int volumen;
}

class prisma
{
    // Declaramos los datos
    public int ancho;
    public int alto;
    public int espesor;
    public int area;
    public int volumen;
}

```

Como podemos observar, declaramos dos clases. En cada una hemos declarado los datos que le corresponden. La declaración de los datos es muy similar a la declaración de variables, y si prestamos atención, podemos notar que ambas clases tienen nombres que se repiten en sus datos como **area** y **volumen**. Esto es posible porque cada clase es una entidad diferente, y estos nombres nunca se confundirán entre sí. Continuemos con nuestra clase cubo. Ahora lo que necesitamos hacer es colocar los métodos de la clase. Las operaciones son muy sencillas. Nuestra clase cubo quedará como se muestra en el siguiente bloque de código:

```

class cubo
{
    // Declaramos los datos
    public int lado;
    public int area;
    public int volumen;

    // Método para calcular el área
    public void CalculaArea()
    {
        area = (lado * lado) * 6;
    }
}

```

```
// Método para calcular el volumen
public void CalculaVolumen()
{
    volumen = lado * lado * lado;
}
```

La declaración de los métodos se hace adentro del bloque de código de la clase y cada uno de ellos tiene su propio bloque de código. Como podemos observar, usan los datos directamente. Esto se debe a que cualquier dato declarado en la clase es conocido por todos los métodos de esa clase. Más adelante continuaremos con la clase **prisma**.

Cómo crear la instancia de nuestra clase

Ya tenemos la clase cubo que es el plano donde se indica qué datos tiene y qué información utilizará con ellos. Ahora tenemos que construir el objeto, que es quien llevará a cabo el trabajo en realidad. Para instanciar un objeto de la clase cubo debemos utilizar el código que se muestra a continuación:

```
cubo miCubo = new cubo();
```

Quien realiza realmente la instanciación es **new** y esa instancia queda referenciada como **miCubo**. A partir de **miCubo** podremos empezar a trabajar con él.

Cómo asignarles valores a datos públicos

Como ya tenemos la instancia, ya podemos comenzar a trabajar con ella. Lo primero que haremos será asignarle un valor al dato **lado**. Haremos la asignación por medio del operador de asignación **=** (igual), pero también deberemos indicar a cuál de todos los datos de la clase vamos a acceder. Esto lo hacemos con el

III EL ACCESO PROTEGIDO

El acceso privado es utilizado cuando hacemos uso de la herencia. Si declaramos un dato como protegido, la propia clase y las clases que hereden de ellas podrán acceder a él, leerlo y modificarlo. Todas las demás clases lo verán como si su acceso fuera privado y no podrán acceder a él directamente.

operador punto. Por ejemplo, asignemos el valor de **5** al **lado**, realizando esto como vemos en el código a continuación:

```
miCubo.lado = 5;
```

Cómo invocar métodos de un objeto

Cuando invocamos el método de un objeto, éste ejecuta el código que tiene en su interior. Desde el exterior de la clase solamente podemos invocar métodos que sean públicos. La invocación del método es muy similar a lo que aprendimos en el **Capítulo 3**, con todos los casos que vimos. En este caso sólo tenemos que indicar con qué objetos trabajaremos, seguido el operador **.** y el nombre del método con sus parámetros, si los necesita.

Invoquemos los métodos para calcular el área y el volumen del cubo:

```
// Invocamos los métodos
miCubo.CalculaArea();
miCubo.CalculaVolumen();
```

Cómo imprimir un dato público

Como ya tenemos los valores calculados, ahora los podemos mostrar. Para esto los usaremos como cualquier variable normal, pero debemos indicar el objeto con el que trabajamos, seguido del operador **.** y en nombre del dato.

```
// Desplegamos los datos
Console.WriteLine("Area={0}, Volumen={1}", miCubo.area,
    miCubo.volumen);
```

Nuestro programa queda de la siguiente manera:

```
class cubo
{
    // Declaramos los datos
    public int lado;
    public int area;
    public int volumen;
```

```
// Método para calcular el área
public void CalculaArea()
{
    area = (lado * lado) * 6;
}

// Método para calcular el volumen
public void CalculaVolumen()
{
    volumen = lado * lado * lado;
}

}

class prisma
{
    // Declaramos los datos
    public int ancho;
    public int alto;
    public int espesor;
    public int area;
    public int volumen;
}

class Program
{
    static void Main(string[] args)
    {
        // Instanciamos a la clase cubo
        cubo miCubo = new cubo();

        // Asignamos el valor del lado
        miCubo.lado = 5;

        // Invocamos los métodos
        miCubo.CalculaArea();
        miCubo.CalculaVolumen();

        // Desplegamos los datos
        Console.WriteLine("Área={0}, Volumen={1}", miCubo.area,
            miCubo.volumen);
    }
}
```

}

Si ejecutamos el programa obtendremos lo siguiente:

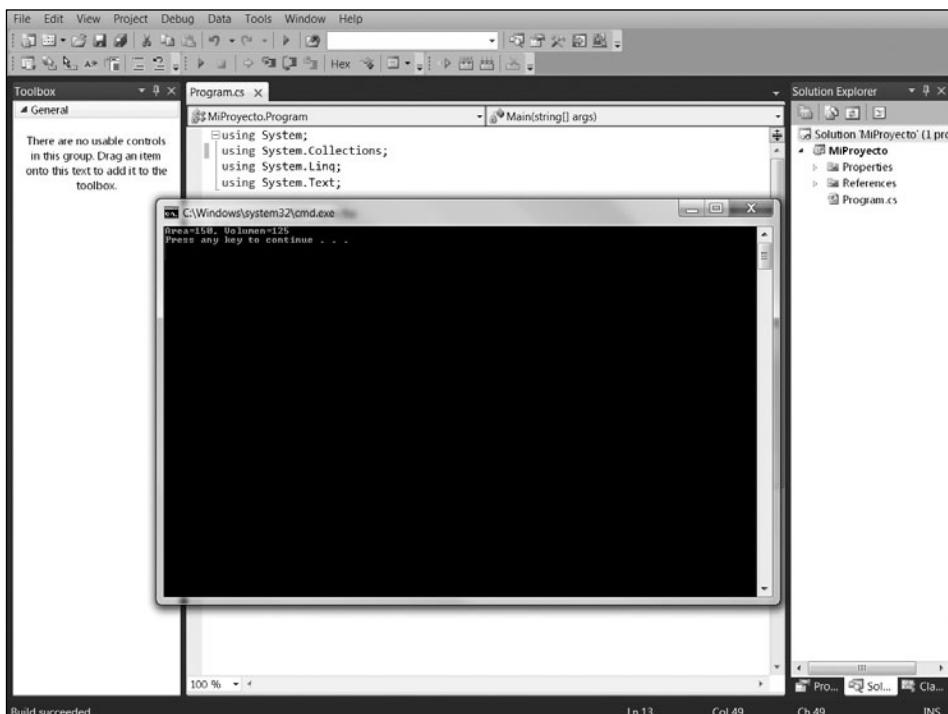


Figura 3. Vemos el resultado de la ejecución de nuestro programa.

Una de las ventajas que tiene la programación orientada a objetos es la reutilización de código. Si necesitáramos dos cubos o más, simplemente creamos nuevas instancias. Cada una de ellas tendría en su interior sus propias variables y podría llevar a cabo los cálculos que fueran necesarios.

Por ejemplo, modifiquemos el programa para que se tengan dos cubos. El segundo cubo estará en la instancia **tuCubo** y tendrá un valor de **lado** de **8**.

```
static void Main(string[] args)
{
    // Instanciamos a la clase cubo
    cubo miCubo = new cubo();
    cubo tuCubo = new cubo();
```

```

    // Asignamos el valor del lado
    miCubo.lado = 5;
    tuCubo.lado = 8;

    // Invocamos los métodos
    miCubo.CalculaArea();
    miCubo.CalculaVolumen();
    tuCubo.CalculaArea();
    tuCubo.CalculaVolumen();

    // Desplegamos los datos
    Console.WriteLine("Mi cubo Area={0}, Volumen={1}",
        miCubo.area, miCubo.volumen);
    Console.WriteLine("Tu cubo Area={0}, Volumen={1}",
        tuCubo.area, tuCubo.volumen);
}

```

Este cambio solamente fue necesario en **Main()**, ya que todo el comportamiento que necesitamos se encuentra en la clase. Ejecutemos el programa y veamos el resultado.

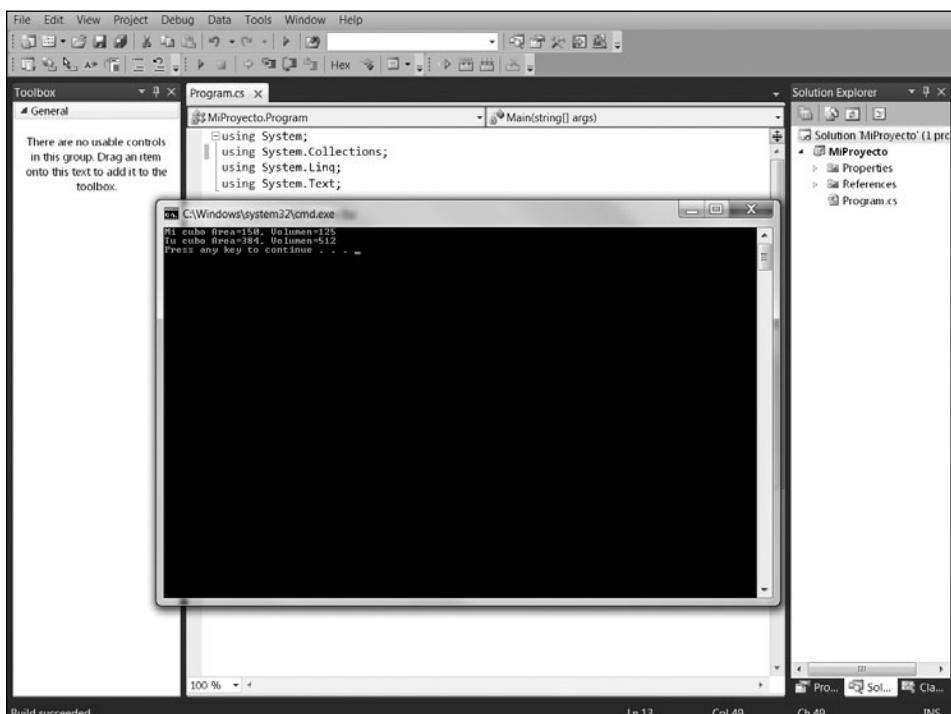


Figura 4. Aquí observamos los resultados calculados por ambas instancias.

Protección de datos y creación de propiedades

Con la clase cubo hemos visto cómo crear una clase sencilla que ya tiene funcionalidad. Sin embargo, presenta un problema. El primero es que todos sus datos son públicos, lo que nos puede llevar a corrupción de información. Para proteger los datos tenemos que hacerlos privados y proveer una función de interfaz a aquellos a los que se necesita acceder por el exterior.

Para tener una comparación, trabajaremos sobre nuestra clase prisma. Lo primero que hacemos es cambiar el acceso de los datos a privado.

```
class prisma
{
    // Declaramos los datos
    private int ancho;
    private int alto;
    private int espesor;
    private int area;
    private int volumen;
}
```

Las propiedades son funciones de interfaz. Nos permiten acceder a los datos privados de una manera segura y controlada, pero van más allá de simples funciones ya que también nos brindan una forma de acceso intuitiva y sencilla.

La propiedad puede ser de varios tipos: **lectura**, **escritura** y la combinación de ambas(**lectura-escritura**). Una propiedad de lectura solamente nos permite leer el dato, pero no podemos agregarle información. Una propiedad que es de tipo escritura sólo nos permite colocar información en el dato, pero no podemos leerlo. La propiedad de **lectura-escritura** permite llevar a cabo ambas acciones.

Para lograr esto, la propiedad tendrá dos métodos. El método relacionado con la lectura se conoce como **get** y el relacionado con la escritura es **set**. Dependiendo de cuál método coloquemos en la propiedad, será su tipo.

La propiedad de éste tiene la siguiente forma de declaración:

```
public tipo nombre {
    get
    {
        ...
        ...
        return x;
    }
}
```

```

    }

set
{
...
...
x=value;
}

}

```

Las propiedades son públicas para poder llamarlas desde el exterior de la clase. El tipo está referenciado al tipo del valor que leerá o colocará, ya sea entero, flotante, doble, etcétera. En su interior tenemos **get**, donde colocamos el código para sacar un valor de la clase por medio de **return**, y a **set**, donde ponemos el código necesario para introducir un valor en la clase.

Empecemos por crear propiedades para la clase prisma. Lo primero que tenemos que preguntarnos es a qué datos se necesita acceder por el exterior y qué tipo de acceso requieren. Podemos ver que los datos **ancho**, **alto** y **espesor** necesitarán de escritura, pero también de lectura. Esto es en caso de que necesitemos saber las dimensiones. A sus propiedades las llamaremos: **Ancho**, **Alto** y **Espesor**.

Los otros datos que necesitan tener una propiedad son **area** y **volumen**, pero en este caso solamente necesitamos leerlos. No tiene sentido escribir sobre esos datos, ya que la clase calculará sus propios valores.

Ahora que ya sabemos cuáles son las propiedades necesarias podemos decidir si es necesario algún tipo de validación. Sabemos que no podemos tener prismas con cualquiera de sus lados con valor de **0** o negativos y ése es un buen punto para validar. Si el usuario diera un valor incorrecto, entonces colocaremos por **default** el valor **1**. Hay que recordar que esto lo hacemos como ejemplo y cada aplicación puede tener sus propias reglas de validación para la información.

Cuando usemos el método set tendremos una variable previamente definida por el lenguaje que se llama **value**. Esta variable representa el valor que el usuario asigna y podemos usarlo en la lógica que necesitemos.

Nuestras propiedades de lectura y escritura quedan de la siguiente forma:

```

// Definimos las propiedades
public int Ancho
{

```

```
get
{
    return ancho;
}

set
{
    if (value <= 0)
        ancho = 1;
    else
        ancho = value;
}

public int Alto
{
    get
    {
        return alto;
    }

    set
    {
        if (value <= 0)
            alto = 1;
        else
            alto = value;
    }
}
```

III EL USO DE LAS PROPIEDADES

En ocasiones, dentro de los métodos **get** y **set** de la propiedad, tendremos un **return** o una asignación. Aunque esto es correcto, no debemos olvidar que en las propiedades podemos colocar cualquier lógica válida de C# que nos permita validar la información que sale o entra. Esto hará que nuestro código sea más seguro y evitaremos problemas con nuestra información.

```
}

public int Espesor
{
    get
    {
        return espesor;
    }

    set
    {
        if (value <= 0)
            espesor = 1;
        else
            espesor = value;

    }
}
```

Ahora crearemos las propiedades de sólo lectura para el área y el volumen:

```
public int Area
{
    get
    {
        return area;
    }
}

public int Volumen
{
    get
    {
        return volumen;
    }
}
```

Como estas propiedades son de sólo lectura únicamente llevan el método **get**.

Cómo acceder a las propiedades

Acceder a las propiedades es muy sencillo ya que únicamente necesitamos colocar el objeto con el que queremos trabajar seguido del operador . y el nombre de la propiedad. La asignación se lleva a cabo por medio del operador =.

Por ejemplo, si deseamos indicar que el ancho tiene 5 unidades, hacemos lo siguiente:

```
miPrisma.Ancho=5;
```

Y si deseamos imprimir el valor de la propiedad **Volumen**:

```
Console.WriteLine("El volumen es {0}",Volumen);
```

Métodos públicos y privados

Como mencionamos anteriormente, los métodos pueden ser públicos o privados. Los primeros pueden ser invocados desde el exterior del objeto y los privados solamente desde su interior. Al programar la única diferencia es el tipo de acceso que colocamos. En nuestro ejemplo necesitamos dos métodos públicos para invocar al cálculo del área y el volumen, y un método privado que nos apoyará en el cálculo del área. Veamos cómo programarlos:

```
public void CalculaVolumen()
{
    volumen = ancho * alto * espesor;
}

public void CalculaArea()
{
    int a1 = 0, a2 = 0, a3 = 0;

    a1 = 2 * CalculaRectangulo(ancho, alto);
    a2 = 2 * CalculaRectangulo(ancho, espesor);
    a3 = 2 * CalculaRectangulo(alto, espesor);

    area = a1 + a2 + a3;
}

private int CalculaRectangulo(int a, int b)
```

```

{
    return (a * b);
}

```

Los métodos son muy sencillos, lo importante es notar que el método **CalculaRectangulo()** tiene acceso privado, por lo que nadie del exterior puede invocarlo. Sin embargo, **CalculaArea()** lo invoca sin problemas ya que pertenece a la misma clase.

Convertir un objeto a cadena

Tenemos varias opciones para imprimir la información que guarda un objeto en la consola. La primera consiste en leer la propiedad e imprimir de la forma tradicional. Otra opción puede ser crear un método dentro del objeto que se especialice en la impresión de la información. Esto puede ser útil si deseamos imprimir tan solo uno o algunos datos.

La última opción y la que aprenderemos a usar es la de programar el método **ToString()** para la clase. Esto ya lo hemos hecho en el capítulo anterior para las estructuras. El mecanismo es similar, simplemente tenemos que implementar una versión del método **ToString()** para nuestra clase. Este método regresa una cadena que contiene la información en el formato que deseamos y tampoco necesita recibir ningún parámetro.

Este método se implementa adentro de la clase y cada clase en la aplicación lo puede tener de ser necesario. Entonces, cuando necesitemos imprimir los contenidos del objeto simplemente lo invocaremos e imprimiremos la cadena resultante. El método debe tener acceso público ya que es necesario que el exterior pueda invocarlo.

En nuestro caso, el método quedaría de la siguiente manera:

```

public override string ToString()
{
    String mensaje = "";
    mensaje += "Ancho " + ancho.ToString() + " Alto " +
               alto.ToString() + " Espesor " + espesor.ToString();
    mensaje += " Area " + area.ToString() + " Volumen " +
               volumen.ToString();
    return mensaje;
}

```

En este ejemplo hacemos uso de la concatenación para poder generar la cadena que el método regresará. Hacemos esto para comparar con el uso de **StringBuilder**, que se

ha utilizado en el capítulo donde hablamos de las estructuras. La concatenación suele utilizarse para agrupar registros obtenidos de una base de datos. Podemos hacer uso de cualquiera de estos métodos según necesitemos de ellos. La impresión de los contenidos del objeto apoyándonos en este método puede ser de la siguiente forma:

```
Console.WriteLine(miPrisma.ToString());
```

Si lo deseamos, podemos probar a éste en nuestro programa, y ver que obtendremos el resultado de la siguiente figura:

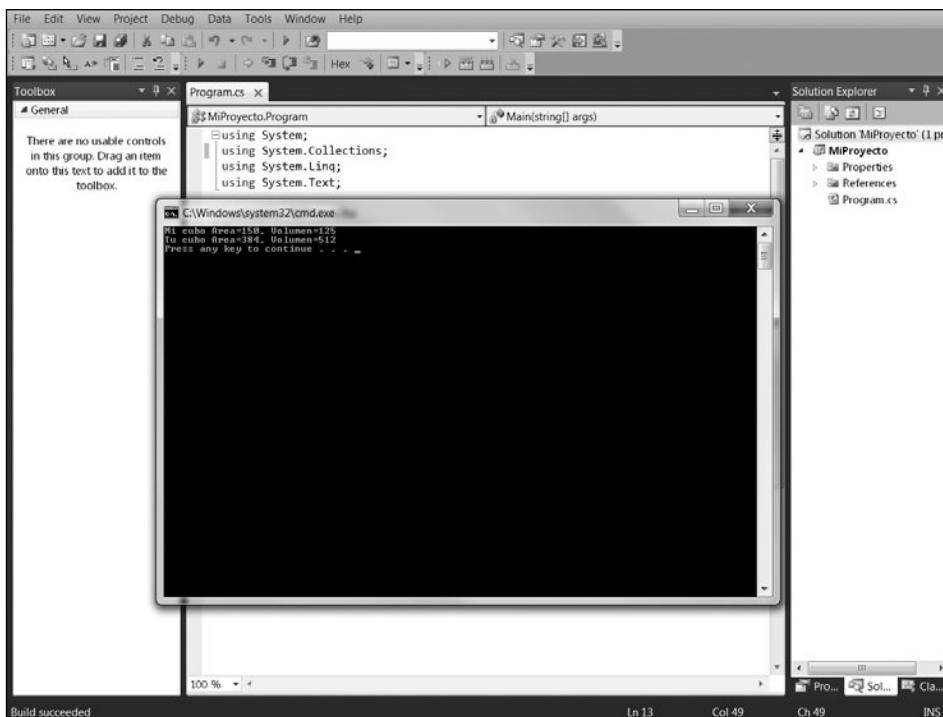


Figura 5. Podemos observar cómo hemos impreso los contenidos del objeto apoyándonos en el método `ToString()`.

Para imprimir solamente algunos datos

Si solamente necesitamos imprimir algunos datos, entonces es necesario crear un método especializado para ello. Este método deberá tener acceso público para que pueda ser invocado desde el exterior, y en el caso de llegar a necesitarlo, puede prepararse el método para recibir parámetros, aunque esto no es necesario. Supongamos que deseamos tener disponible un método que sólo imprima los resultados para el área y el volumen.

El trozo de código para ello, es el que mostramos en el siguiente bloque:

```
public void ImprimeResultado()
{
    Console.WriteLine("El área es {0}, el volumen es {1}", area,
                      volumen);
}
```

Como observamos, el código del método es muy sencillo. Igualmente podemos probarlo en nuestra aplicación y obtener el siguiente resultado en la consola.

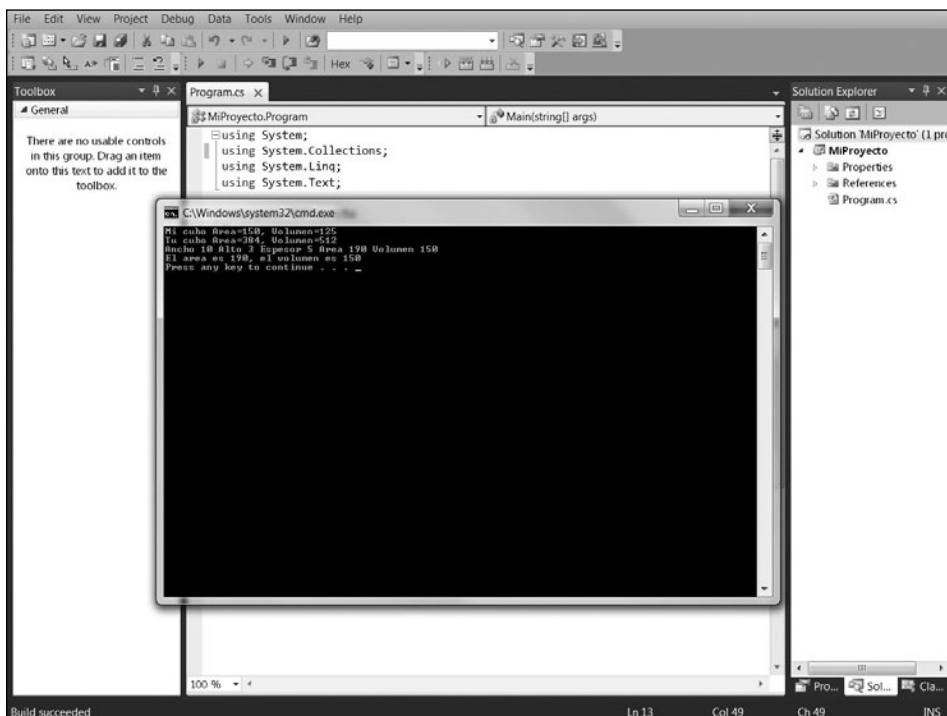


Figura 6. Ahora podemos comparar la impresión con la ayuda de `Tostring()` y del método especializado.

El constructor en las clases

El constructor es un método especial que podemos utilizar con las clases. Éste generalmente es usado para inicializar los valores de los datos con los que trabajará el objeto. La forma como lo utilizamos con las clases es equivalente a la forma como lo utilizamos con las estructuras en el capítulo anterior. El constructor es un método especial y tiene ciertas características que lo distinguen de los demás métodos.

Su primera característica es que tiene el mismo nombre de la clase y su segunda característica más importante es que no tiene **tipo**, es decir, que no solamente no regresa nada, sino que no tiene tipo alguno.

El constructor es invocado en forma automática cuando el objeto es instanciado, ya que esto nos da la oportunidad de llevar a cabo cosas en el instante que se instancia el objeto, como por ejemplo, hacer inicializaciones. El constructor puede tener en su interior cualquier código válido de C# y también puede tener parámetros o no. Si utilizamos los parámetros tendremos que pasar los valores necesarios en el momento en el que instanciamos el objeto.

Veamos un ejemplo de constructor para nuestra clase prisma. En este constructor no utilizaremos parámetros, veamos el siguiente ejemplo:

```
public prisma()
{
    // Datos necesarios
    String valor="";

    // Pedimos los datos
    Console.WriteLine("Dame el ancho");
    valor=Console.ReadLine();
    ancho=Convert.ToInt32(valor);

    Console.WriteLine("Dame el alto");
    valor=Console.ReadLine();
    alto=Convert.ToInt32(valor);

    Console.WriteLine("Dame el espesor");
    valor=Console.ReadLine();
    espesor=Convert.ToInt32(valor);
}
```

III NO HAY QUE CONFUNDIR EL CONSTRUCTOR

Mucha gente confunde el constructor y cree erróneamente que es el encargado de construir el objeto. Esto es falso. El constructor no se encarga de instanciar el objeto, sólo se invoca en forma automática en el momento en que el objeto se instancia. No debemos tener esta confusión.

Como podemos observar dentro del ejemplo, el constructor tiene acceso de tipo público. Esto es importante, ya que como se invoca automáticamente, el exterior necesitará tener acceso a él.

El constructor se encargará de solicitarle al usuario los datos necesarios. Cuando llevemos a cabo la instanciación, será de la siguiente forma:

```
prisma miPrisma = new prisma();
```

Si compilamos el programa, obtendremos el siguiente resultado:

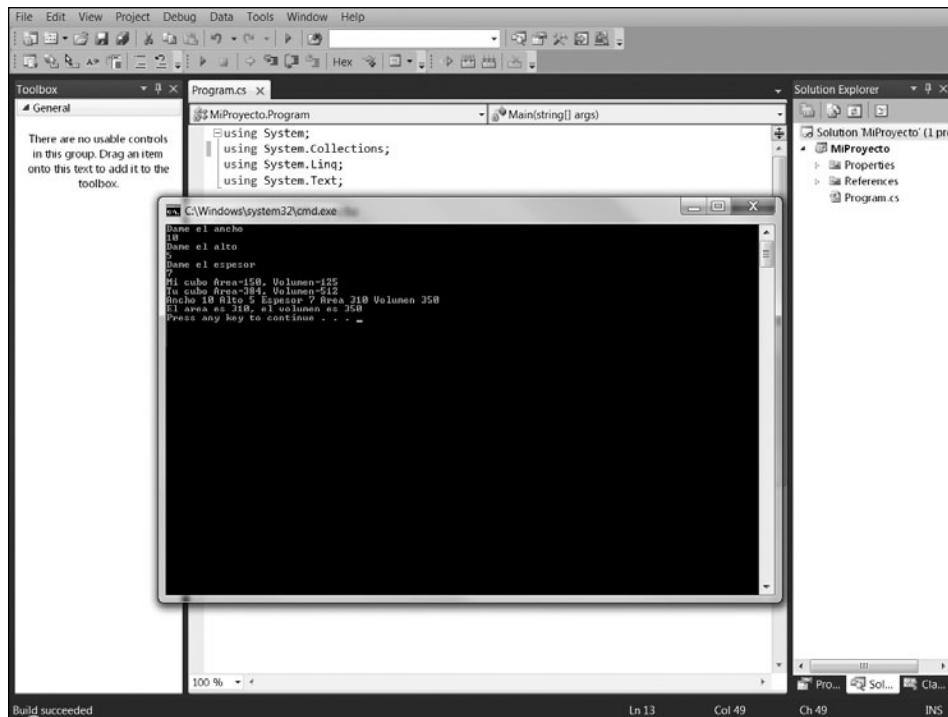


Figura 7. Podemos observar cómo se ejecuta el constructor y nos solicita los datos necesarios.

III CONSTRUCTORES PRIVADOS

En las técnicas avanzadas de programación, como por ejemplo la programación de patrones, podemos encontrar constructores privados. Uno de los patrones que lo utilizan se conoce como **singleton**. Generalmente, haremos uso de constructores públicos hasta que aprendamos este tipo de técnicas avanzadas.

Sobrecarga del constructor

El constructor puede ser sobrecargado, es decir, podemos tener más de una versión del constructor. Esto resulta útil ya que podemos seleccionar cómo se inicializarán los datos del objeto dependiendo del tipo de constructor que utilicemos.

El compilador seleccionará automáticamente el tipo de constructor dependiendo de los tipos y la cantidad de parámetros.

Ya tenemos un constructor que nos pide directamente los datos, pero ahora podemos hacer un constructor que recibe los valores con los que se inicializará el prisma en el momento de la instancia. Para esto tenemos que colocar parámetros en la segunda versión del constructor.

```
public prisma(int pancho, int palto, int pespesor)
{
    // Asignamos los valores
    ancho=pancho;
    alto=palto;
    espesor=pepesor;
}
```

Cuando instanciamos el objeto con este constructor, lo usaremos así:

```
prisma miPrisma3 = new prisma(5,3,7);
```

Observemos cómo los valores fueron pasados al momento de la instancia. El valor **5** será colocado en el dato **ancho**, **3** en el dato **alto** y **7** en el dato **espesor**.

Si tenemos el dato contenido en una variable, también es posible utilizarla cuando instanciamos el objeto. Solamente debemos asegurarnos de que el tipo de la variable sea el mismo que el tipo del parámetro que colocamos. Si no fueran del mismo tipo, lo recomendable es utilizar **type cast**. Supongamos que tenemos lo siguiente:

```
int miNumero=11;
...
...
prisma miPrisma4 = new prisma(5,3,miNumero);
```

Como vemos, pasamos una copia del valor de **miNúmero** y ahora en el interior del objeto **miPrisma4** en dato **espesor** tendrá el valor de **11**.

Podemos continuar llevando a cabo más sobrecargas del constructor, tantas como sean necesarias. La cantidad de sobrecargas dependerá del análisis y las necesidades del programa. No debemos exagerar en las sobrecargas, debemos colocar solamente aquellas que sean realmente necesarias.

El programa completo que tenemos luce de la siguiente manera:

```
class cubo
{
    // Declaramos los datos
    public int lado;
    public int area;
    public int volumen;

    // Método para calcular el área
    public void CalculaArea()
    {
        area = (lado * lado) * 6;
    }

    // Método para calcular el volumen
    public void CalculaVolumen()
    {
        volumen = lado * lado * lado;
    }
}

class prisma
{
    // Declaramos los datos
    private int ancho;
    private int alto;
    private int espesor;
    private int area;
    private int volumen;

    // Definimos las propiedades
    public int Ancho
    {
        get
    }
}
```

```
{  
    return ancho;  
}  
  
set  
{  
    if (value <= 0)  
        ancho = 1;  
    else  
        ancho = value;  
}  
  
}  
  
public int Alto  
{  
    get  
    {  
        return alto;  
    }  
  
set  
{  
    if (value <= 0)  
        alto = 1;  
    else  
        alto = value;  
}  
  
}  
  
}  
  
public int Espesor  
{  
    get  
    {  
        return espesor;  
    }  
}
```

```
    set
    {
        if (value <= 0)
            espesor = 1;
        else
            espesor = value;

    }

}

public int Area
{
    get
    {
        return area;
    }
}

public int Volumen
{
    get
    {
        return volumen;
    }
}

// Definimos los constructores

public prisma()
{
```

III LOS NOMBRES DE LAS PROPIEDADES

Las propiedades pueden tener cualquier nombre válido de C#. El nombre de las propiedades debe reflejar de alguna forma el tipo de dato sobre el cual actúa. Si nuestro dato es **costo**, entonces la propiedad se puede llamar **Costo**. El dato y la propiedad no pueden tener el mismo nombre, pero recordemos que C# distingue entre mayúsculas y minúsculas.

```

// Datos necesarios
String valor = "";

// Pedimos los datos
Console.WriteLine("Dame el ancho");
valor = Console.ReadLine();
ancho = Convert.ToInt32(valor);

Console.WriteLine("Dame el alto");
valor = Console.ReadLine();
alto = Convert.ToInt32(valor);

Console.WriteLine("Dame el espesor");
valor = Console.ReadLine();
espesor = Convert.ToInt32(valor);

}

// Version sobrecargada
public prisma(int pancho, int palto, int pespesor)
{

    // Asignamos los valores
    ancho = pancho;
    alto = palto;
    espesor = pespesor;
}

// Definimos los métodos

```



PROBLEMAS CON LAS PROPIEDADES

Tenemos que decidir oportunamente el tipo de acceso que permitirá la propiedad. Si tratamos de asignarle un valor a una propiedad de sólo lectura, el compilador nos indicará un error. Lo mismo sucede si tratamos de leer una propiedad de sólo escritura. Por eso lo mejor es planificar durante la etapa de análisis el acceso que les colocaremos a las propiedades de nuestra clase.

```
public void CalculaVolumen()
{
    volumen = ancho * alto * espesor;
}

public void CalculaArea()
{
    int a1 = 0, a2 = 0, a3 = 0;

    a1 = 2 * CalculaRectangulo(ancho, alto);
    a2 = 2 * CalculaRectangulo(ancho, espesor);
    a3 = 2 * CalculaRectangulo(alto, espesor);

    area = a1 + a2 + a3;
}

private int CalculaRectangulo(int a, int b)
{
    return (a * b);
}

public override string ToString()
{
    String mensaje = "";
    mensaje += "Ancho " + ancho.ToString() + " Alto " +
               alto.ToString() + " Espesor " + espesor.ToString();
    mensaje += " Area " + area.ToString() + " Volumen " +
               volumen.ToString();
    return mensaje;
}

public void ImprimeResultado()
{
    Console.WriteLine("El área es {0}, el volumen es {1}", area,
                      volumen);
}

class Program
```

```
{  
    static void Main(string[] args)  
    {  
        // Instanciamos a la clase cubo  
        cubo miCubo = new cubo();  
        cubo tuCubo = new cubo();  
  
        // Instanciamos el prisma  
        prisma miPrisma = new prisma();  
  
        // Instanciamos con la versión sobrecargada  
        prisma miPrisma2 = new prisma(3, 5, 7);  
  
        // Asignamos el valor del lado  
        miCubo.lado = 5;  
        tuCubo.lado = 8;  
  
        // Invocamos los métodos  
        miCubo.CalculaArea();  
        miCubo.CalculaVolumen();  
        tuCubo.CalculaArea();  
        tuCubo.CalculaVolumen();  
  
        // Asignamos los valores al prisma  
        // Quitar comentarios para versión sin constructor  
        //miPrisma.Ancho = 10;  
        //miPrisma.Alto = 3;  
        //miPrisma.Espesor = 5;  
  
        // Invocamos los métodos del prisma  
        miPrisma.CalculaArea();
```



APRENDER MÁS SOBRE LAS CLASES

Como siempre, podemos consultar MSDN para conocer más sobre los elementos que aprendemos en este libro. Si deseamos conocer más sobre las clases en C# podemos consultar el sitio web de Microsoft para interiorizarnos o profundizar conocimientos en el tema: [http://msdn2.microsoft.com/es-es/library/0b0thckt\(VS.80\).aspx](http://msdn2.microsoft.com/es-es/library/0b0thckt(VS.80).aspx).

```

        miPrisma.CalculaVolumen();

        miPrisma2.CalculaArea();
        miPrisma2.CalculaVolumen();

        // Desplegamos los datos
        Console.WriteLine("Mi cubo Area={0}, Volumen={1}",
                           miCubo.area, miCubo.volumen);
        Console.WriteLine("Tu cubo Area={0}, Volumen={1}",
                           tuCubo.area, tuCubo.volumen);
        Console.WriteLine(miPrisma.ToString());
        miPrisma.ImprimeResultado();

        Console.WriteLine(miPrisma2.ToString());
    }
}

```

Hemos escrito bastante código para el funcionamiento de esta aplicación. Ejecutemos el programa y nuestro resultado es el siguiente:

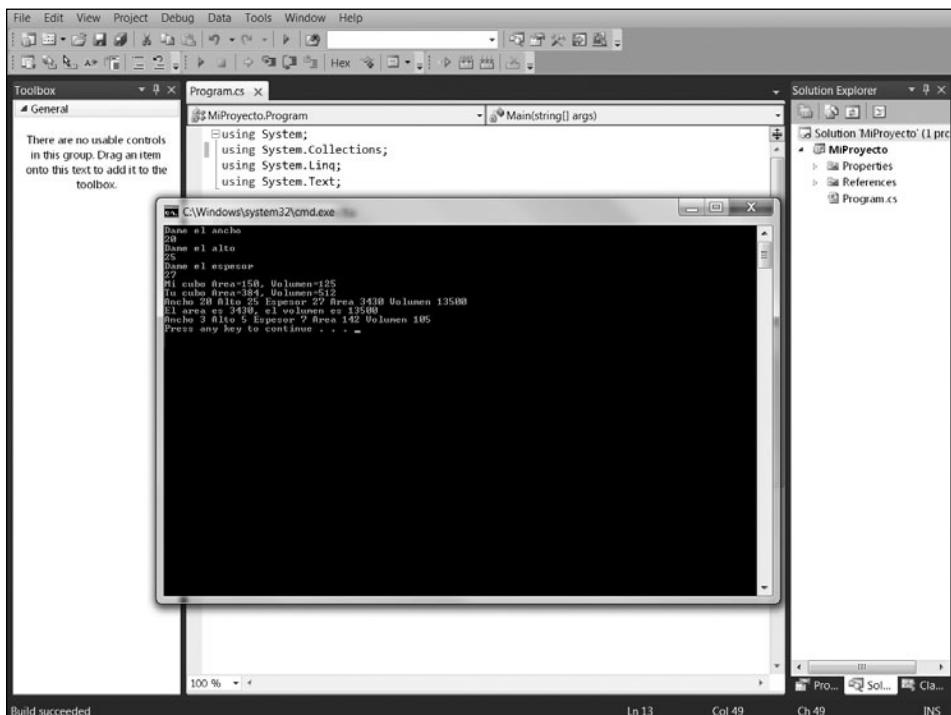


Figura 8. Cada objeto usa el constructor que le corresponde.

Si revisamos el código que hemos presentado en el bloque anterior, podemos darnos cuenta del funcionamiento de las sobrecargas del constructor que hemos agregado, es importante reafirmar la idea que ya explicábamos en los párrafos anteriores, tenemos la posibilidad de agregar cuantas sobrecargas sean necesarias pero siempre debemos tener en cuenta la necesidad de no exagerar.

A través de este capítulo hemos analizado la importancia del uso de clases para la generación de nuestros programas, utilizando C#. Con esto hemos visto el inicio de los elementos para programar clases. Aún queda mucho por aprender sobre la programación orientada a objetos y su programación en C#, pero éstos serán temas de un libro más avanzado y especializado en la programación del experto en el lenguaje C#.



RESUMEN

La programación orientada a objetos es un paradigma de programación diferente a la programación estructurada. Tenemos que reconocer los objetos que componen el sistema así como la comunicación que tienen entre ellos. La clase es el elemento fundamental de este tipo de programación y actúa como el plano sobre el que los objetos son construidos. Los objetos o las instancias son los que realmente llevan a cabo el trabajo. Los datos y los métodos pueden tener diferentes tipos de acceso: público, privado o protegido. Las propiedades nos sirven como funciones de interfaz para poder acceder a los datos privados de forma segura. Las clases pueden tener un constructor que nos ayuda a inicializar la información. Es posible hacer uso de la sobrecarga en el constructor.



ACTIVIDADES

TEST DE AUTOEVALUACIÓN

1 ¿Qué es la programación orientada a objetos?

2 ¿Cómo definimos una clase?

3 ¿Cuál es la función de un objeto?

4 ¿Cómo se lleva a cabo la instanciación de un objeto?

5 ¿Qué es el acceso público?

6 ¿Cuál es el riesgo de corrupción de información?

7 ¿Para qué sirve una propiedad?

8 ¿Cuáles son los diferentes tipos de propiedades?

9 ¿Cómo se crean los métodos en una clase?

10 ¿Cómo se invoca un método?

11 ¿Qué es el constructor?

12 ¿Cómo se sobrecarga el constructor?

EJERCICIOS PRÁCTICOS

1 Crear el diseño de una clase para llevar el inventario de una tienda.

2 Crear una clase para llevar la información de los estudiantes de una escuela.

3 Crear una clase para polígonos con sobrecarga del constructor.

4 Crear propiedades para la clase polígono que solamente permitan colocar valores válidos.

5 Programar el método `ToString()` para la clase polígono.

Flujos y archivos

A lo largo del libro hemos realizado muchos programas, y cada vez que usamos un programa, debemos introducirle información. Esto no ha sido un gran problema en los ejemplos realizados hasta aquí, porque generalmente manipulamos pocos datos. Sin embargo, en muchas aplicaciones reales necesitaremos tener alguna metodología para poder guardar la información y recuperarla cuando la aplicación se ejecute nuevamente.

Para resolver esto, tenemos acceso a los archivos y los flujos.

Los flujos	350
Los stream en la memoria	351
El uso de archivos	363
Resumen	371
Actividades	372

LOS FLUJOS

Los **flujos** también son conocidos como **streams** por su nombre en inglés. Se los llama de esta forma porque nos recuerdan como fluye el agua, pero en este caso se tratará de flujos de información. El stream es utilizado para poder mover información de un lugar a otro. A veces, moveremos la información de la memoria a otra parte de la memoria, pero generalmente lo que haremos será mover la información de la memoria a un dispositivo de almacenamiento como el disco duro o del dispositivo nuevamente a la memoria.

Cuando hacemos uso de los flujos, la información no es enviada en un solo movimiento, sino que se envía **byte** por **byte** de forma ordenada. Como el envío es de esta forma, tenemos que tener cuidado sobre qué **byte** procesamos en ese momento.

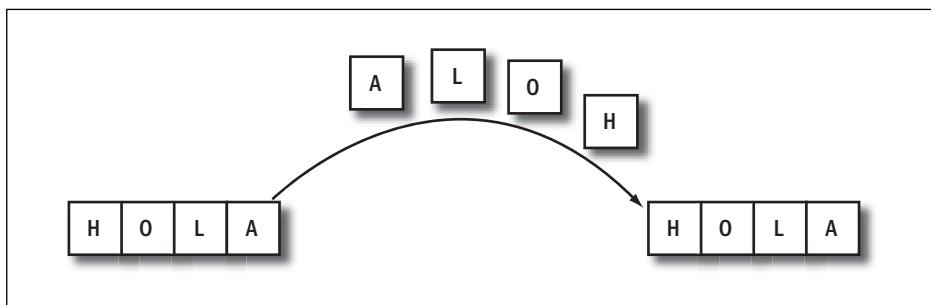


Figura 1. Los flujos nos permiten mover la información de un lugar a otro de la memoria, **byte** por **byte**.

Para poder tener control sobre el **byte** a enviar, imaginaremos que tenemos un **apuntador** o **indicador** en nuestro flujo. Este indicador siempre nos señala cuál es el siguiente **byte** a enviar. Cada vez que enviamos un **byte** a su nuevo lugar por medio del flujo, el indicador se actualiza y nos señala el siguiente **byte**. Esta forma de trabajo funciona muy bien si el envío de datos es secuencial, pero también debemos tener un mecanismo que nos permita seleccionar a nosotros mismos el **byte** a enviar. Para hacer esto tenemos que indicar de alguna manera el **byte** y esto solamente lo podemos llevar a cabo si tenemos un punto de referencia dentro del flujo.

En el flujo encontramos tres puntos de referencia. El primero es el **inicio** del flujo. Si queremos colocarnos en un **byte** en particular para enviar, simplemente tenemos que decir a cuántos **bytes** de **distancia** desde el inicio se encuentra el **byte** que deseamos procesar. El segundo punto de referencia será el **final** del flujo. De manera similar tenemos que decir a cuántos **bytes** de distancia se encuentra el **byte** a procesar desde el final del flujo. El tercer punto de referencia es la **posición actual** dentro del flujo, de igual forma a los casos anteriores, debemos dar la distancia desde nuestra posición actual al **byte** que nos interesa procesar. En todos los casos, este **byte** se volverá nuestro nuevo **byte** actual.

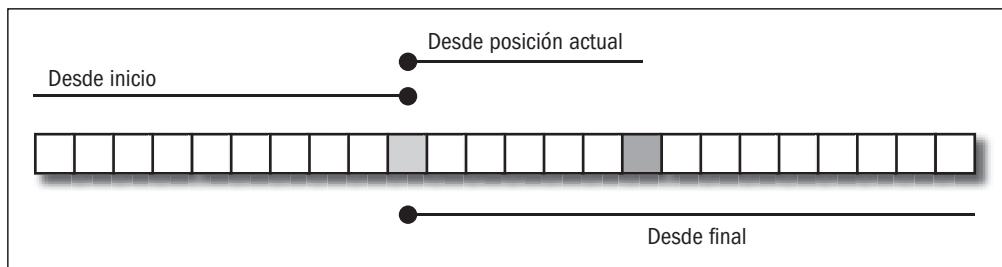


Figura 2. Aquí vemos cómo un mismo byte del stream puede estar referenciado de formas diferentes.

Los stream en la memoria

Empecemos a trabajar con el stream. Éste simplemente llevará la información de un lugar a otro. La forma más sencilla de aprenderlo es haciendo streams en memoria. Aquí veremos las operaciones básicas que podemos realizar con ellos y después este conocimiento puede ser llevado a streams que se comuniquen con dispositivos de almacenamiento masivo como discos duros.

Para trabajar con streams en la memoria nos apoyaremos de una clase conocida como **MemoryStream**. Cuando necesitemos usar esta clase debemos agregar el **namespace** al que pertenece. Para esto colocamos el siguiente código en la parte superior de nuestro programa, antes de las declaraciones de las clases.

```
// Adicionamos para el uso de los stream
using System.IO;
```

Esta clase crea un stream, pero el lugar donde guarda la información es un sitio en memoria. La información es guardada como un arreglo de **bytes sin signo**. La clase **MemoryStream** tiene sobrecargado su constructor y seguramente podemos encontrar una versión de acuerdo con nuestras necesidades. El constructor puede crear el arreglo en la memoria vacío o puede inicializarlo a un tamaño en particular. Hay que recordar que el tamaño está en **bytes**.

III DISTANCIAS EN LOS FLUJOS

Como hemos visto, para seleccionar un **byte** dentro del flujo debemos indicar su distancia desde el punto de referencia. Si es el mismo punto de referencia el que deseamos enviar, la distancia será cero. Las distancias se miden de izquierda a derecha, y negativas de derecha a izquierda. Por ello cuando usamos como referencia el final del flujo las distancias llevan el signo menos.

Nosotros usaremos la versión en la que podemos indicar el tamaño inicial del arreglo y para instanciar **MemoryStream** podemos hacerlo de la siguiente manera:

```
// Creamos el stream en memory
// La iniciamos con una capacidad de 50 bytes
MemoryStream ms = new MemoryStream(50);
```

El objeto se llama **ms**, pero puede tener cualquier nombre válido de C#. El tamaño inicial que le asignamos es **50** bytes.

Cómo obtener información sobre el stream

Nosotros podemos obtener ciertos datos sobre el stream. Estos datos nos pueden servir en la lógica de nuestro programa y resulta útil poder conocerlos. Los datos que podemos obtener son: **la capacidad, la longitud y la posición**.

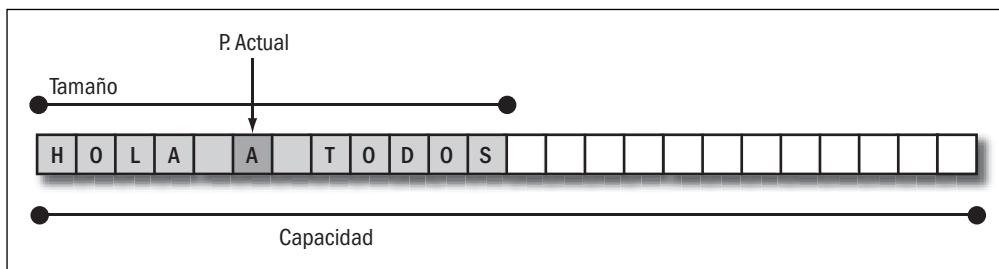


Figura 3. En este diagrama podemos observar las tres propiedades del stream.

La capacidad nos indica cuántos **bytes** puede almacenar el stream. En el ejemplo anterior es de **50**, ya que le hemos colocado ese tamaño en su instanciación. El valor de la capacidad se guarda en la propiedad **Capacity** y ésta es de tipo entero. Si deseamos obtener la capacidad lo podemos hacer de la siguiente forma:

```
int capacidad = 0;
```

III CUIDADO CON EL TAMAÑO

Los streams en memoria que se inicializan con un arreglo de **bytes** sin signo no pueden modificar su tamaño. Esto hace importante que seleccionemos adecuadamente el tamaño a utilizar. Podemos hacer una estimación del tamaño máximo y colocarla en el constructor.

```

...
...
capacidad = ms.Capacity;

```

El siguiente dato que podemos obtener es la **longitud**. Ésta nos indica el tamaño de la información actual que tiene el stream. El tamaño está dado en **bytes**. La propiedad relacionada con esta información es **Length**. Esta propiedad es de tipo **long**. A continuación veamos un ejemplo de cómo podemos obtener este dato:

```

long longitud = 0;
...
...
longitud = ms.Length;

```

Por último tenemos la posición. Este dato es sumamente importante ya que nos indica el lugar donde se encuentra el **byte** actual, es decir el siguiente a ser procesado. Esta posición está referenciada con relación al inicio del stream. La propiedad para este dato se llama **Position** y al igual que la anterior también es de tipo **long**. La forma de obtener estos datos es similar a las anteriores.

```

long posicion = 0;
...
...
posicion = ms.Position;

```

Cómo colocar la posición en el stream

Una de las actividades más importantes y frecuentes que realizaremos con los streams es colocar la posición actual en un sitio determinado. Para esto hacemos uso de un método conocido como **Seek()**. Éste necesita dos parámetros. El primero es la distancia desde el punto de referencia donde quedará la nueva posición actual en el stream. El segundo indica el punto de referencia que utilizaremos.

El punto de referencia tiene que ser de tipo **SeekOrigin**. Este tipo tiene tres valores posibles: **Begin**, **Current** y **End**. El valor **Begin** es usado para indicar que estamos referenciados con relación al origen del stream. Cuando se utiliza el valor **Current** nuestra posición actual será usada como punto de referencia. Si usamos **End**, la parte final del stream será el punto de referencia para encontrar nuestra nueva posición. Por ejemplo, si deseamos colocarnos en el inicio del stream, podemos hacerlo de la siguiente manera:

```
ms.Seek(0, SeekOrigin.Begin); // nos colocamos a 0 distancia desde el  
inicio
```

En este caso, indicamos que nos encontramos a **0** distancia desde el inicio del stream. Si lo que deseamos es encontrarnos a **10** bytes de distancia desde el inicio, la forma de usar **Seek()** es la siguiente:

```
ms.Seek(10, SeekOrigin.Begin);
```

Pero también podemos colocarnos desde otro punto de referencia. Por ejemplo, para colocarnos a **10** bytes de distancia desde el final del stream usaremos:

```
ms.Seek(-10, SeekOrigin.End);
```

Hay que observar que el valor usado es **-10**, ya que, como vimos anteriormente, en la placa de la página 251, las distancias que se miden hacia la izquierda del punto de referencia son negativas.

Por último, si lo que deseamos es movernos con relación a nuestra posición actual, por ejemplo avanzar **5** bytes, lo que usaremos es:

```
ms.Seek(5, SeekOrigin.Current);
```

Y si lo que deseamos es retroceder **10** bytes desde nuestra posición actual, usaremos el código que veremos a continuación:

```
ms.Seek(-10, SeekOrigin.Current);
```

III PROBLEMAS CON EL USO DE SEEK()

Si no utilizamos correctamente el método **Seek()**, tendremos problemas con nuestro programa. Un problema común es tratar de colocar una posición actual que está antes del punto de inicio o después del final del stream. Debemos tener lógica para evitar estas acciones, y para lograrlo, tenemos que hacer uso de las propiedades del stream como posición, longitud y capacidad.

Cómo leer datos del stream

Ahora que ya nos hemos posicionado en algún lugar en particular del stream, podemos proceder a leer información, que se lee **byte** por **byte**. Cuando hacemos una lectura, el **byte** que se lee es el que se encuentra en la posición actual. Inmediatamente después de leer, la posición actual se actualiza y nuestra nueva posición actual será la del **byte** consecutivo al que acabamos de leer. Esta lectura se lleva a cabo de forma automática y nosotros no la controlamos.

Para realizar la lectura podemos usar el método **Read()**. Este método necesita tres parámetros. El primero es un arreglo de bytes. Este arreglo es necesario porque se rá el lugar donde se guarde la información leída por el stream. El segundo nos da la capacidad de colocar un **offset** para el arreglo de bytes. Generalmente usaremos el valor cero en este parámetro, ya que queremos que la información leída se coloque desde el inicio en el arreglo. El tercero es la cantidad de bytes a leer.

El arreglo de bytes para guardar la información es colocado de la siguiente manera:

```
byte [] buffer=new byte[50];
```

En este caso lo llamamos búfer y tendrá capacidad para 50 bytes.

Supongamos que deseamos leer cinco bytes a partir de la posición actual donde nos encontramos. Para ello hacemos uso de **Read()** de la siguiente manera:

```
ms.Read(buffer, 0, 5);
```

Después de ejecutarse esta línea, **buffer** contendrá los cinco bytes leídos y los podemos encontrar al inicio de **buffer**.

Cómo escribir información el stream

No solamente es posible leer información del stream, también podemos agregarla o escribirla. Para esto tendremos un método especializado que se llama **Write()**. Para



EVITAR ERRORES CON LA LECTURA

Es posible tener problemas con la lectura de un stream, como el olvidar crear el búfer. Otro problema es tratar de leer de un stream que ya se cerró o no ha sido abierto. Un error menos frecuente es dar un valor negativo en el segundo o tercer parámetro. Si los valores se controlan por variables debemos colocar una lógica que impida poner valores inválidos en los parámetros.

el uso de este método utilizaremos tres parámetros. En el primer parámetro tenemos que colocar el búfer o el arreglo de bytes desde el que tomaremos la información para colocarla en el stream. El segundo parámetro es la posición en el stream desde donde empezaremos a escribir. Generalmente utilizaremos el valor de cero, y de esta forma empezará a escribirse desde el inicio del stream. El último parámetro es la cantidad máxima de bytes que se escribirán.

Un ejemplo práctico de esto es el siguiente:

```
// Escribimos los datos en la cadena
ms.Write(miBuffer, 0, 15);
```

Cómo cerrar el stream

Algo que no debemos olvidar hacer es cerrar el stream. Cuando éste está cerrado, los recursos que se hayan necesitado se liberan. Si el stream está cerrado no es posible llevar a cabo ningún tipo de operación sobre él. El cierre del stream se lleva a cabo por medio del método **Close()**, que no necesita ningún parámetro. La forma de utilizar el método se muestra a continuación:

```
ms.Close();
```

Programa de ejemplo

Ahora podemos crear un pequeño programa de ejemplo que utilice lo que hemos realizado con los streams. El programa será sencillo, ya que únicamente nos interesa conocer y experimentar con el código. Nuestro programa le pedirá al usuario una cadena que será colocada en el stream y luego podremos leer diferentes partes del stream con los métodos que hemos aprendido.

Como siempre, tenemos que empezar por definir las variables que son necesarias.

```
// Creamos el stream en memory
// La iniciamos con una capacidad de 50 bytes
MemoryStream ms = new MemoryStream(50);

// Cadena con información
String informacion = "";

// Variables necesarias
int capacidad = 0;
```

```
long longitud = 0;
long posicion = 0;
byte [] buffer=new byte[50];
```

En primer lugar tenemos un objeto llamado **ms** que es de tipo **MemoryStream** y que puede guardar hasta **50** bytes en su interior. Luego creamos la cadena que utilizaremos para guardar la frase escrita por el usuario. Como experimentaremos con el stream creamos tres variables. Cada una de ellas se usa para guardar la capacidad, la longitud y la posición, respectivamente. Al final creamos un arreglo de bytes llamado **buffer** y de tamaño **50**.

Pedir los datos será de la forma usual y no necesitamos explicarlo en este momento.

```
// El usuario da los datos
Console.WriteLine("Dame la cadena para el flujo");
informacion = Console.ReadLine();
```

Ahora que ya tenemos una cadena, podemos escribirla adentro del stream.

```
ms.Write(ASCIIEncoding.ASCII.GetBytes(informacion), 0, informacion.Length);
```

Lo primero que hacemos es obtener los bytes de la cadena, pero éstos estarán codificados como ASCII. Luego indicamos que escribiremos en el stream desde su inicio. Por último, indicamos la cantidad de bytes a escribir, que lo obtenemos por medio de la longitud de la cadena. Ahora mostraremos la información que podemos obtener sobre el stream.

```
// Obtenemos información de la cadena
capacidad = ms.Capacity;
longitud = ms.Length;
posicion = ms.Position;

// Mostramos la información
Console.WriteLine("Capacidad {0}, longitud {1}, posicion {2}",
    capacidad, longitud, posicion);
```

Después de obtener la información, simplemente la mostramos en la consola.

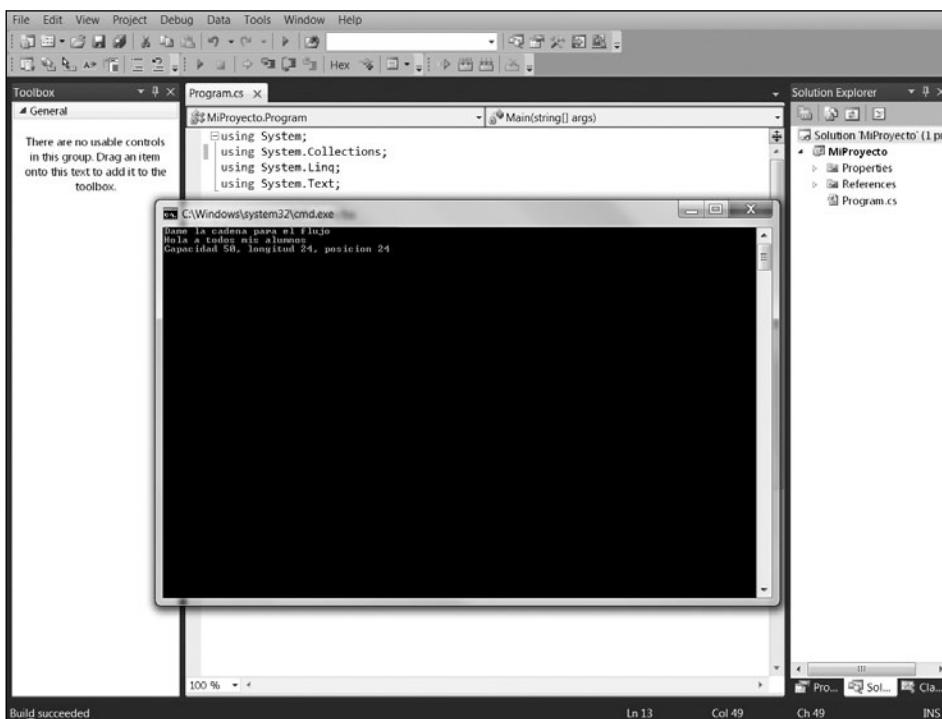


Figura 4. El programa ya tiene el stream y hemos obtenido su información.

Ahora podemos empezar a experimentar con stream. Nos colocaremos en diferentes posiciones y leeremos información desde ahí. Empecemos por lo más sencillo. Leeremos los primeros cinco bytes que se encuentran en el stream.

```
// Colocamos y leemos datos basandonos en el inicio
    ms.Seek(0, SeekOrigin.Begin); // nos colocamos a 0 distancia
                                // desde el inicio
    ms.Read(buffer, 0, 5); // desde donde nos encontramos, 5
                          // caracteres

    // Mostramos la informacion
    Console.WriteLine(ASCIIEncoding.ASCII.GetString(buffer));
```

Lo primero que hacemos es usar el método **Seek()**. Con este método decimos que queremos encontrarnos a distancia cero desde el inicio del stream. Evidentemente ésta es la primera posición disponible. Como nuestra posición actual ya está definida, procedemos a hacer uso del método **Read()**. En este caso leemos cinco bytes desde donde nos encontramos. Los bytes leídos desde el stream ahora se encuentran guardados en el arreglo de bytes que llamamos búfer.

Para mostrar lo que acabamos de obtener usamos el método **WriteLine()**, y como solamente tenemos una colección de bytes, debemos codificarlas adecuadamente con **GetString()**. La codificación que deseamos es ASCII. En la consola aparecen los primeros cinco caracteres de la frase que escribimos.

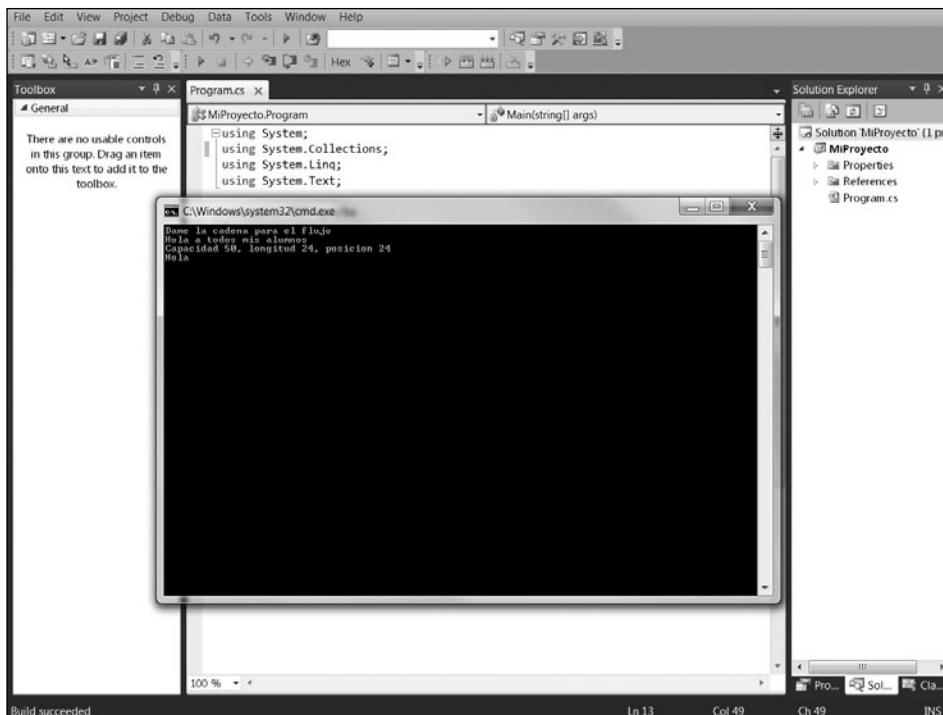


Figura 5. Podemos observar que los primeros cinco caracteres son mostrados.

En el código que veremos a continuación, haremos una prueba, realizando otra lectura en una posición diferente.

```
// Otra lectura desde el inicio
ms.Seek(10, SeekOrigin.Begin); // nos colocamos a 10 distancia
// desde el inicio
ms.Read(buffer, 0, 5); // desde donde nos encontramos, 5
// caracteres
Console.WriteLine(ASCIIEncoding.ASCII.GetString(buffer));
```

En este caso, por medio del método **Seek()** nos posicionamos a **10** bytes de distancia desde que es iniciado el stream. Nuevamente procedemos a leer cinco bytes desde esa posición que serán colocados en el **buffer**. Para poder comprobar que esto es óptimo y que funciona en forma adecuada lo mostramos en la consola.

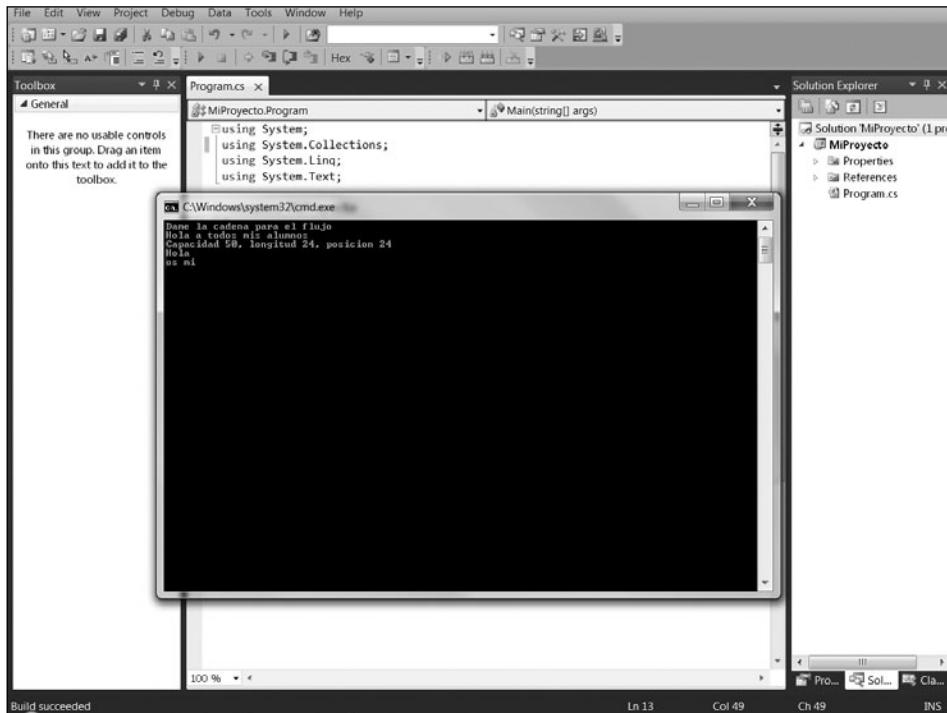


Figura 6. En este caso podemos observar los bytes leídos desde la posición donde nos encontramos.

Pero sabemos que no solamente podemos utilizar el inicio del stream como referencia para indicar la posición actual. Ahora realizaremos otra prueba con el final del stream como punto de referencia.

```
// Lectura relativa al final del flujo
    ms.Seek(-10, SeekOrigin.End); // nos colocamos a 10 distancia
                                // desde el final
    ms.Read(buffer, 0, 5); // desde donde nos encontramos, 5
                          // caracteres

Console.WriteLine(ASCIIEncoding.ASCII.GetString(buffer));
```

Para este ejemplo usamos nuevamente el método **Seek()**, pero con la diferencia que ahora indicaremos que el punto de referencia es el final del stream. Nuestra nueva posición actual se encontrará a una distancia de diez bytes desde el final del stream. Por esta razón, vamos a colocar el valor **-10** en el primer parámetro. Ya colocados en esta distancia simplemente procedemos a leer sólo cinco bytes. Al final de la lectura mostramos lo que hemos obtenido en la consola.

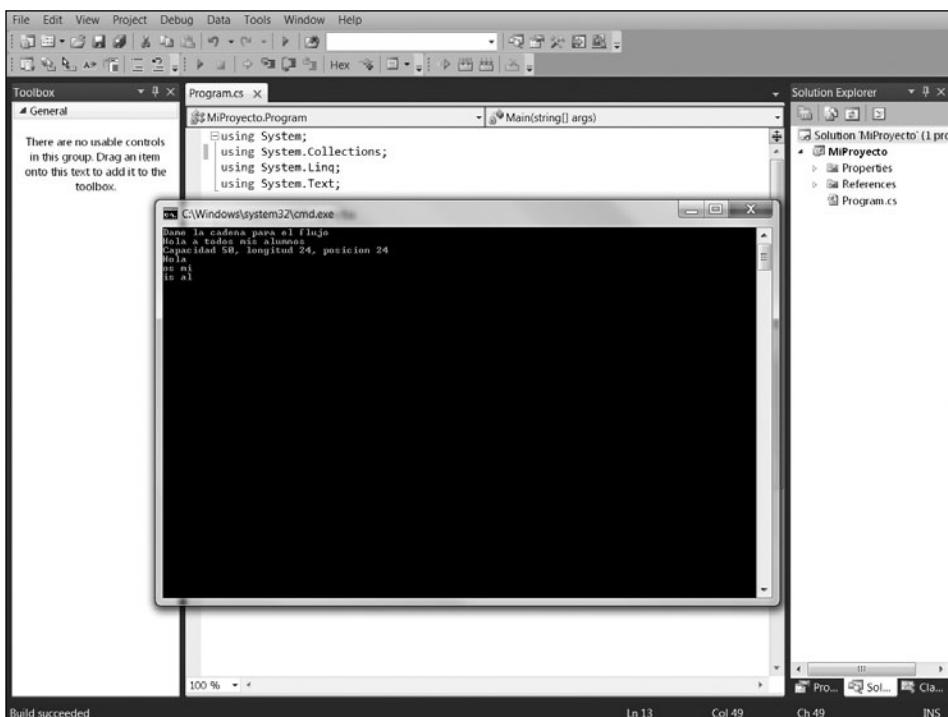


Figura 7. Ahora observamos los bytes leídos con referencia al punto final del stream.

Debemos recordar que cuando se lleva a cabo una lectura o escritura en el stream, la posición actual se modifica. La última posición colocada explícitamente se encontraba a diez bytes del final del stream, pero después de la lectura se ha modificado en cinco bytes.

Comprobemos esto de forma sencilla:

```
// Obtenemos nuestra posición actual
posición = ms.Position;
Console.WriteLine("La posición es {0}", posición);
```

III

PARA EVITAR PROBLEMAS CON LOS STREAM

Un problema con el que nos podemos encontrar cuando empezamos a utilizar los streams es olvidar indicar el **namespace** necesario. Coloquemos la siguiente línea en la parte superior de nuestro código: **using System.IO;**. Si no lo hacemos, el programa no podrá reconocer las clases y los métodos relacionados con los stream y tendremos problemas de compilación.

Simplemente obtenemos nuestra posición actual y la mostramos en la consola. Esto lo podemos hacer en cada operación y verificar cómo se altera la posición actual.

Desde nuestra nueva posición actual podemos continuar leyendo.

```
// Lectura relativa desde la posición actual
ms.Read(buffer, 0, 5); // desde donde nos encontramos, 5
                      caracteres
Console.WriteLine(ASCIIEncoding.ASCII.GetString(buffer));
```

Ahora no hemos utilizado el método **Seek()** ya que deseamos continuar leyendo desde donde nos encontramos.

Por último, no debemos olvidar cerrar el stream.

```
// Cerramos el flujo, no olvidar
ms.Close();
```

A continuación, ejecutemos el programa para ver su funcionamiento:

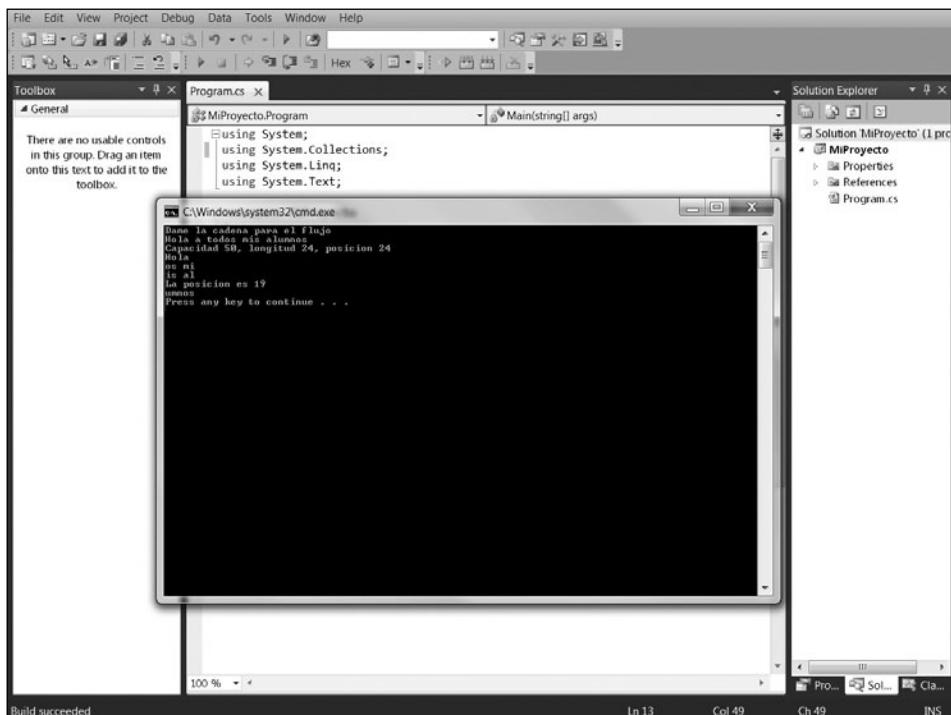


Figura 8. Esta figura nos muestra la ejecución completa del programa.

El uso de archivos

Como hemos aprendido, los streams no solamente funcionan en la memoria, sino que también nos pueden servir para mover información de la memoria a un dispositivo de almacenamiento masivo. Este dispositivo generalmente será el disco duro. Nosotros podemos crear, escribir y leer **archivos** en el disco duro apoyándonos en los streams. En esta sección del libro haremos dos programas, uno que escriba datos en el disco, y otro que los lea y los presente en la consola.

Todas las operaciones que hemos aprendido para los streams serán válidas y veremos que es sencillo llevar esto a cabo.

Cómo crear un stream a archivo

Lo primero que tenemos que hacer es crear un stream a archivo. En lugar de crear el stream en la memoria, éste utilizará un medio de almacenamiento masivo. Para poder llevar a cabo esto, necesitamos utilizar la clase **FileStream**. Esta clase nos provee de toda la funcionalidad que necesitamos.

El constructor de esta clase necesita dos parámetros. El primero tiene que ser una cadena que contenga el nombre del archivo con el que trabajaremos. Es útil que el nombre del archivo también tenga su extensión.

El segundo parámetro es más interesante. En este parámetro colocaremos el modo del archivo, que indica cómo funcionará y se manipulará el archivo. El valor colocado debe ser del tipo de la enumeración **FileMode**. Los valores posibles son: **Append**, **Create**, **CreateNew**, **Open**, **OpenOrCreate**, **Truncate**.

Es importante conocer lo que significa cada uno de estos modos, por lo que lo comentaremos aquí. El modo **Create** nos permite crear un nuevo archivo. En caso de que el archivo ya existiera, simplemente se sobrescribe. El modo **CreateNew** también nos permite crear un archivo, pero si el archivo ya existe, se produce una excepción. Las excepciones y su manejo lo veremos en un capítulo posterior.

El modo **Open** nos permite abrir un archivo. Si el archivo que intentamos abrir no existe, entonces se produce una excepción. En el modo **OpenOrCreate** si el archivo existe, se abre, pero en el caso de que el archivo no exista, se crea.

En el modo **Append**, si el archivo existe será abierto, y la posición actual será colocada al final del archivo, de forma tal que cualquier información escrita, sea



III COLOCAR EL NOMBRE DEL ARCHIVO

El nombre del archivo no necesariamente necesita ser colocado explícitamente. Podemos usar una variable de tipo cadena para contenerlo. También es posible colocar toda la ruta al archivo en este nombre. Si no colocamos la ruta del archivo, éste será creado en el mismo directorio que el ejecutable de nuestra aplicación.

agregada al archivo sin modificar lo anterior. En el caso de que el archivo no exista, será creado un nuevo archivo.

El modo **Truncate** es especial y debemos tener cuidado con él ya que abre el archivo. Entonces, los contenidos se eliminan hasta que el archivo tenga una longitud de 0 bytes. Por ejemplo, podemos crear un stream a disco en modo de creación, como podemos ver en el código a continuación:

```
FileStream fs = new FileStream("miTexto.txt", FileMode.Create);
```

El archivo a crear se llama **miTexto.txt** y será creado en el mismo directorio que el ejecutable de nuestra aplicación. El nombre del stream es **fs** y ya podemos llevar a cabo las operaciones necesarias sobre él.

Cómo escribir información en el archivo

Ya que tenemos el stream, es posible empezar a trabajar con él, y al igual que con los streams de la memoria, podemos llevar a cabo diferentes operaciones, entre ellas la de escritura. Para escribir el archivo usaremos el método **Write()**, que pertenece al stream. Por ejemplo, si deseamos escribir una cadena al archivo, podemos hacerlo de la siguiente forma:

```
// Escribimos al stream la cadena capturada  
fs.Write(ASCIIEncoding.ASCII.GetBytes(cadena), 0, cadena.Length);
```

El método necesita tres parámetros, el primer parámetro es el arreglo de bytes que escribiremos, el segundo es la posición a partir de donde empezaremos a escribir con respecto a nuestra posición actual, y el último parámetro nos sirve para indicar la cantidad de bytes que colocaremos. Es importante no olvidar que esta posición se actualiza después de hacer uso del método **Write()**.

En nuestro ejemplo a realizar vamos a suponer que una cadena será escrita al archivo. A esta cadena la tenemos que colocar como un arreglo de bytes, pero éstos deben estar codificados como **ASCII**, para lo cual usaremos la clase **ASCIIEncoding**. Para poder obtener los bytes de la cadena deberemos hacer uso del método **GetBytes()** por el que se pasa la cadena escrita como parámetro.

La cantidad de bytes que deseamos colocar será la longitud de la cadena, que obtenemos al hacer uso de la propiedad **Length**.

Cómo cerrar el archivo

El cierre del archivo es muy sencillo ya que únicamente debemos cerrar el stream. Sabemos que esto lo realizamos por medio del método **Close()**.

```
// Cerramos el stream
fs.Close();
```

Ejemplo de un programa para escribir cadenas a disco

Ahora que ya conocemos los elementos necesarios para utilizar los streams a disco, podemos hacer un ejemplo sencillo. Haremos un programa que le preguntará al usuario cadenas, y cada una de éstas serán escritas a disco. Cuando el usuario dé una cadena vacía, es decir un **return**, el archivo se cerrará y se terminará la aplicación. Luego podremos utilizar un programa, como el **Bloc de Notas**, para poder leer el archivo que creamos.

Empecemos por definir las variables necesarias y crear el archivo:

```
// Variables necesarias
String cadena = "";

// Creamos el stream al archivo
// Experimentar con append y create
FileStream fs = new FileStream("miTexto.txt",
    FileMode.Create);
```

Luego, podemos hacer uso de un ciclo **do while**, que se repetirá constantemente hasta que el usuario dé una cadena vacía. En el interior del ciclo llevaremos a cabo la petición de la cadena y la escritura de ésta a disco.

```
// Capturamos cadenas
do
{
    // Leemos la cadena del usuario
    cadena = Console.ReadLine();

    // Escribimos al stream la cadena capturada
    fs.Write(ASCIIEncoding.ASCII.GetBytes(cadena), 0,
        cadena.Length);

} while (cadena != "");
```

Para finalizar simplemente cerramos el archivo.

```
// Cerramos el stream  
fs.Close();
```

El programa completo queda de la siguiente manera:

```
using System;  
using System.Collections.Generic;  
using System.Text;  
using System.IO;  
  
namespace Cap11_2  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            // Variables necesarias  
            String cadena = “”;  
  
            // Creamos el stream al archivo  
            // Experimentar con append y create  
            FileStream fs = new FileStream(“miTexto.txt”,  
                FileMode.Create);  
  
            // Capturamos cadenas  
            do  
            {  
                // Leemos la cadena del usuario  
                cadena = Console.ReadLine();  
  
                // Escribimos al stream la cadena capturada  
                fs.Write(ASCIIEncoding.ASCII.GetBytes(cadena), 0,  
                    cadena.Length);  
  
            } while (cadena != “”);  
  
            // Cerramos el stream  
            fs.Close();
```

```

        }
    }
}

```

Una vez escrito el programa, sólo nos queda compilar a ambos y ejecutarlos. Es posible escribir varias cadenas y luego simplemente presionar la tecla **ENTER** para finalizar. Toda la información que se ha escrito en el programa será almacenada en el archivo. Luego con el **Bloc de Notas** podemos abrir el archivo y ver que efectivamente se encuentra la información que escribimos.

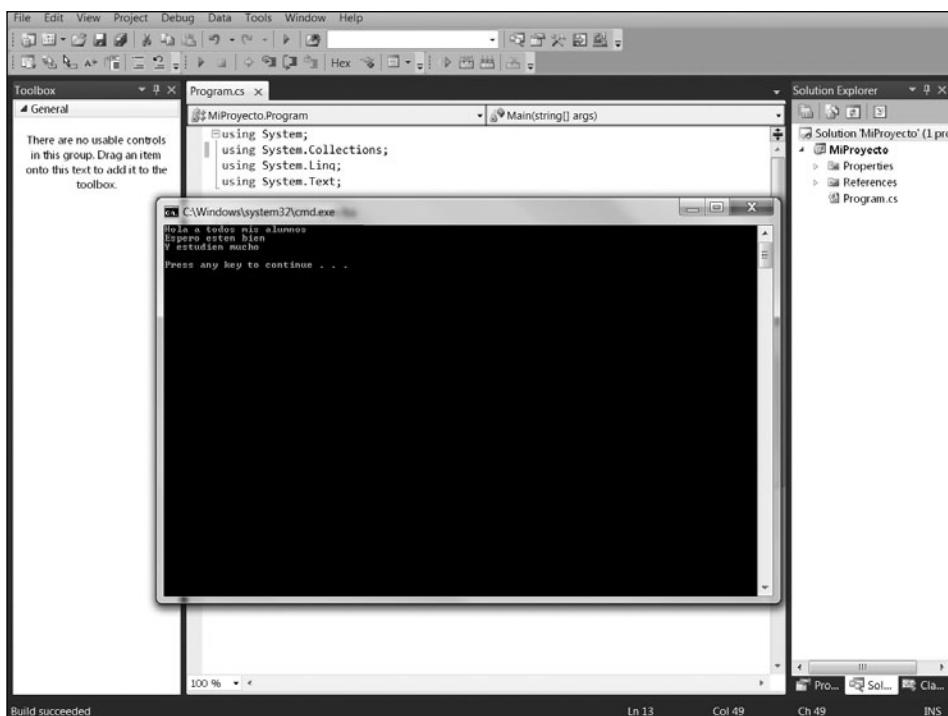


Figura 9. En el programa hemos escrito las frases y se han guardado en el archivo.



CÓMO ENCONTRAR NUESTRO ARCHIVO

Dentro de la carpeta que contiene nuestro proyecto o solución creado, debemos buscar otra carpeta con el mismo nombre. En su interior ésta última debe haber otra carpeta con el nombre **bin**. En ésta se encuentran dos carpetas, **Debug** y **Release**. En el interior de alguna de ellas estará el archivo creado por nuestro programa.

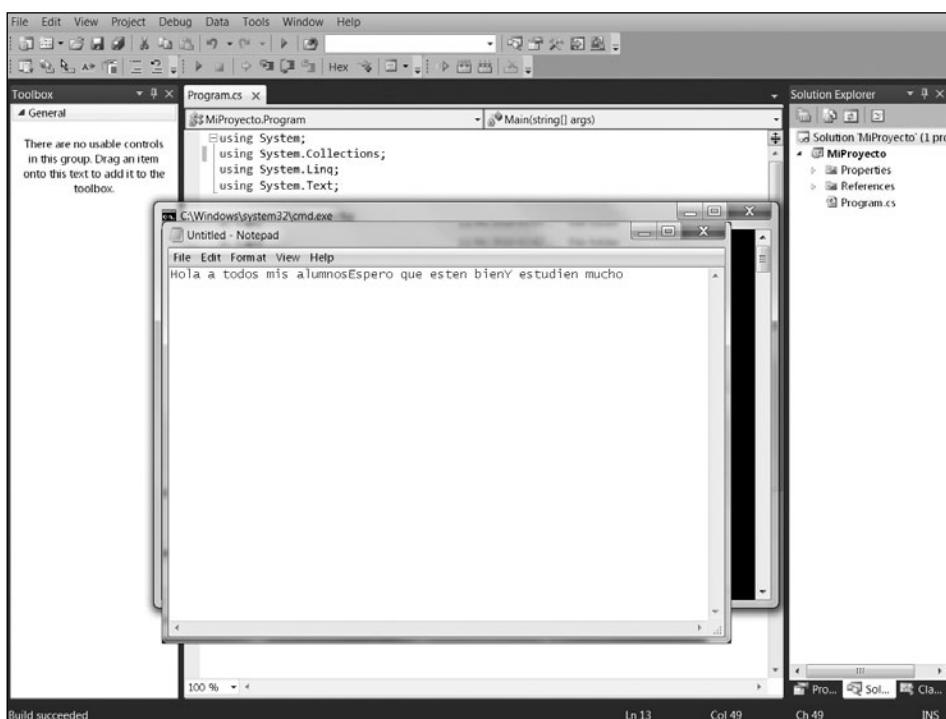


Figura 10. Con el bloc de notas podemos verificar que la información ha sido escrita.

Cómo hacer una aplicación que lee archivos

Como ya tenemos el código para escribir información a un archivo, ahora sería bueno poder tener una aplicación que lea esa información. El proceso es sencillo ya que conocemos los streams. Básicamente, lo que tenemos que llevar a cabo es la apertura de un stream a archivo y luego proceder a leer los contenidos. Debemos recordar que estamos leyendo bytes, por lo que es necesario procesarlos o convertirlos a un tipo adecuado para nuestro programa. En el programa anterior usamos cadenas ya que ahora los bytes leídos serán colocados en una cadena. Desde luego, cuando hemos finalizado con la lectura debemos cerrar el stream del archivo. Para crear esta aplicación empezemos por definir un arreglo de bytes. La finalidad de este arreglo de bytes es recibir la información que proviene del archivo vía el stream. El tamaño del arreglo debe ser lo suficientemente grande para contener la cantidad de información. Otra opción es leer el stream en grupos de bytes del mismo tamaño que nuestro arreglo, pero esto requiere de lógica extra que controle el proceso. Con fines de experimentación, colocaremos el tamaño del arreglo en 100 bytes ya que en nuestro programa anterior no hemos colocado cadenas muy grandes. El arreglo queda de la siguiente forma:

```
// Variables necesarias
```

```
byte[] infoArchivo = new byte[100];
```

Luego podemos proceder a abrir nuestro stream:

```
// Creamos el stream de lectura
FileStream fs = new FileStream("miTexto.txt", FileMode.Open);
```

Para esto creamos una instancia de la clase **FileStream** y la nombramos **fs**. En el constructor indicamos que nuestro archivo es **miTexto.txt**. Debemos notar que el modo del stream en este caso es **Open** ya que consideramos que el archivo existe y lo abriremos para trabajar con él. Como el archivo fue creado por el programa anterior, deberemos copiarlo de la carpeta donde se encuentra a la carpeta correspondiente de este proyecto. La ruta es similar, solamente que está referenciada a nuestro proyecto actual. Si no existe la carpeta **Debug** o **Release**, deberemos compilar nuestra aplicación para que sean creadas. Hay que copiar el archivo antes de ejecutar el programa. Otra solución es colocar el nombre del archivo a abrir con su ruta completa. Si el archivo que queremos abrir no existe o no está donde indicamos, el programa generará una excepción.

El siguiente paso consiste en leer los contenidos del archivo:

```
// Leemos el contenido del archivo
fs.Read(infoArchivo, 0, (int)fs.Length);
```

El método que utilizamos para la lectura es **Read()**. Este método pertenece al stream. El primer parámetro que colocamos es el arreglo de bytes donde se guardarán los bytes leídos. En el segundo parámetro indicamos que leeremos desde el inicio del stream y por último indicamos la cantidad de bytes a leer. En este caso leemos la totalidad de los bytes en el archivo, por eso indicamos como valor la longitud del stream.

Con esto, ahora nuestro arreglo contiene una copia de los bytes del stream y desplegaremos la información en la consola como cadena.

```
// Mostramos el contenido leido
Console.WriteLine(ASCIIEncoding.ASCII.GetString(infoArchivo));
```

Para mostrar la información tenemos que colocar los bytes como cadena y esta cadena tendrá el formato ASCII. Para esto nos apoyamos nuevamente en **ASCIIEncoding** y el método **GetString()**, y pasamos nuestro arreglo de bytes como parámetro. La escritura en consola es como siempre, por medio de **WriteLine()**. Para finalizar, debemos cerrar el stream de la forma habitual con el uso del método **Close()**.

```
// Cerramos el stream  
fs.Close();
```

Nuestro programa completo queda de la siguiente manera:

```
using System;  
using System.Collections.Generic;  
using System.Text;  
using System.IO;  
  
namespace Cap11_3  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            // Variables necesarias  
            byte[] infoArchivo = new byte[100];  
  
            // Creamos el stream de lectura  
            FileStream fs = new FileStream("miTexto.txt", FileMode.Open);  
  
            // Leemos el contenido del archivo  
            fs.Read(infoArchivo, 0, (int)fs.Length);  
  
            // Mostramos el contenido leido  
            Console.WriteLine(ASCIIEncoding.ASCII.GetString(infoArchivo));  
  
            // Cerramos el stream  
            fs.Close();  
        }  
    }  
}
```

Ahora ya podemos compilar y ejecutar la aplicación. En la consola podemos leer las cadenas que guardamos con el otro programa.

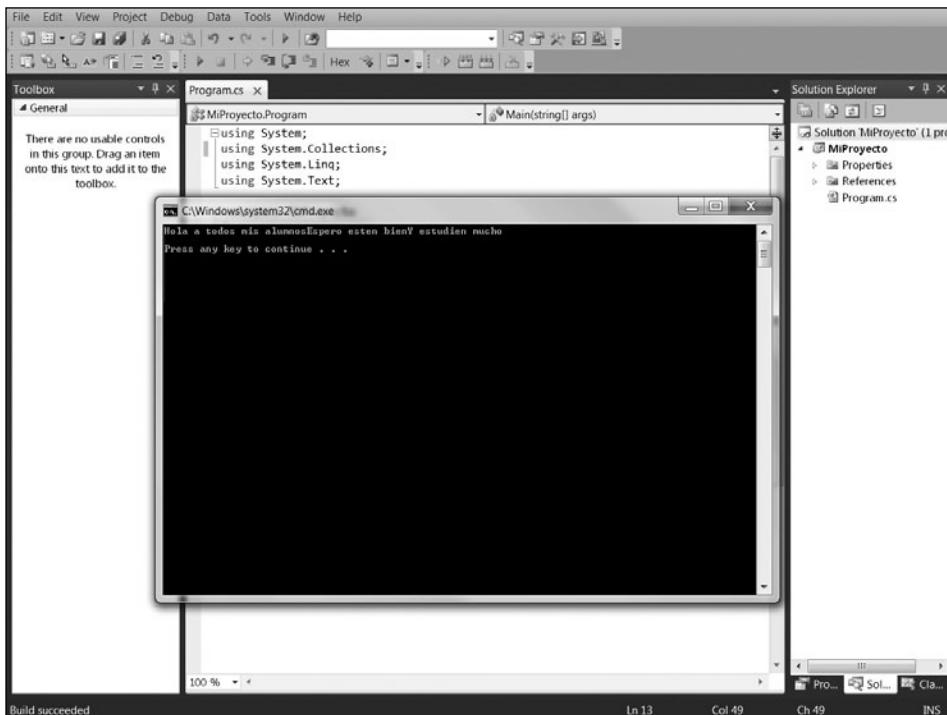


Figura 11. Podemos observar que nuestro programa efectivamente ha leído la información del archivo.

Con esto hemos aprendido el trabajo básico de los flujos y los archivos en C#. También hemos llevado a cabo una serie de manipulaciones sobre los archivos.



RESUMEN

Los flujos o streams permiten mover información de un lugar a otro. Puede moverse entre diferentes partes de la memoria o entre la memoria y los dispositivos de almacenamiento. Para ello necesitamos el namespace System.IO para poder utilizarlo. Para trabajar correctamente con el stream podemos colocarnos en cualquier parte de éste referente al inicio, fin o posición actual. Una vez en la posición adecuada podemos llevar a cabo la lectura de los bytes en el stream. Una vez abierto el stream, y después de trabajar con él, es necesario cerrarlo para liberar cualquier recurso utilizado. Los streams también se pueden utilizar para el acceso al disco con los archivos, podemos abrir el archivo de diferentes formas y llevar a cabo manipulaciones de los archivos como borrarlos, renombrarlos, copiarlos o moverlos.



ACTIVIDADES

TEST DE AUTOEVALUACIÓN

1 ¿Qué es un flujo?

2 ¿Qué namespace es necesario para poder trabajar con streams?

3 ¿Cómo abrimos un stream?

4 ¿Qué método se utiliza para colocar nuestra posición en el stream?

5 ¿Qué puntos de referencia tenemos para colocar nuestra posición en el stream?

6 ¿Por qué en algunos casos se colocan valores negativos cuando indicamos nuestra posición en el stream?

7 ¿Por qué debemos cerrar el stream?

8 ¿Cómo cerramos el stream?

9 ¿Cómo abrimos un archivo?

10 ¿Cuántos modos para abrir un archivo tenemos?

11 ¿Cómo leemos información desde el archivo?

12 ¿Cómo escribimos información hacia el archivo?

EJERCICIOS PRÁCTICOS

1 Hacer un programa que genere un stream en la memoria y coloque la información de una cadena escrita por el usuario. Luego leer esa cadena en orden inverso y mostrarla. Es decir “hola” se mostrará como “aloh”.

2 Hacer un programa que le pida al usuario el nombre y la ruta de un archivo de texto a leer y muestre sus contenidos en la consola.

3 Hacer un programa que le permita al usuario copiar un archivo, y le pregunte el nombre y la ruta del archivo a copiar y la ruta dónde quedará copiado.

4 Hacer un programa que le permita al usuario borrar un archivo. Si el archivo existe, deberá preguntarle al usuario si realmente desea borrarlo y únicamente en caso afirmativo proceder a eliminarlo.

5 Hacer un programa que le permita al usuario cambiar el nombre de cualquier archivo, pero el nuevo nombre siempre debe tener más de cuatro letras.

Depuración

Hemos llegado al último capítulo del libro y en este capítulo aprenderemos las técnicas básicas de depuración.

La depuración es utilizada para corregir problemas en el programa. Algunas veces los programas tienen errores de sintaxis, pero principalmente de lógica.

Al depurarlos podemos observar su comportamiento de mejor forma y detectar el comportamiento que está dando problemas.

Cómo empezar a depurar un programa	374
Para corregir los errores de compilación	374
Cómo corregir los errores en tiempo de ejecución	379
Cómo manejar los errores	383
Resumen	389
Actividades	390

CÓMO EMPEZAR A DEPURAR UN PROGRAMA

Para depurar un programa nosotros tenemos varias herramientas, pero es importante en primer lugar conocer los posibles errores que podemos tener. Básicamente podemos tener dos clases de errores bien diferenciados: por un lado los **errores de compilación** y por otro los **errores en tiempo de ejecución**.

Los errores de compilación son aquellos que impiden que el programa logre compilarse y generalmente son más fáciles de resolver. Muchos de estos errores se deben a problemas de sintaxis, es decir, son responsabilidad del usuario.

Los errores en tiempo de ejecución son aquellos que suceden cuando el programa se está ejecutando. Estos errores pueden deberse a problemas de lógica, es decir, que el algoritmo no fue diseñado correctamente. Pero también pueden ocurrir debido a un mal control de la información, por ejemplo, podemos tratar de leer más allá de los límites correspondientes a un arreglo determinado.

Para corregir los errores de compilación

La mejor forma de poder aprender a depurar programas es con la práctica. Para esto crearemos un pequeño programa de ejemplo. El siguiente programa tiene varios errores para poder identificar y corregir. En el siguiente bloque de código, los errores han sido colocados en forma deliberada.

```
using System;
using System.Collections.Generic;
using System.Text;

namespace Cap12_1
{
    class Program
    {
        static void Main(string[] args)
```

III LOS PROBLEMAS DE SINTAXIS

Los problemas de sintaxis son sencillos de corregir, se deben a que se ha escrito algo de forma incorrecta. Algunas veces los nombres de variables se han escrito erróneamente, o hemos olvidado colocar ; al final de la sentencia. Una revisión rápida de la línea donde está el error nos permite encontrarlo, con la práctica también se reduce la incidencia de estos errores.

```

{
    // Variables necesarias
    int a = 5

    int b = 10;
    int c = 0;
    int r = 0;

    // Hacemos la division
    r = b / c;

    // Mostramos el resultado
    Console.WriteLine("El resultado es {},r);

    // Mostramos el resultado 5 veces
    for (int n = 0; n < 5; n++)
    {
        Console.WriteLine("El resultado es {0}", r);
    }

    // Invocamos la funcion
    MuestraValor();

}

static void MuestraValor(int n)
{
    Console.WriteLine("El resultado es {0}", n);

}
}

```

III PARA EVITAR ERRORES CON LOS BLOQUES DE CÓDIGO

Cuando trabajamos con bloques de código un error común es olvidar cerrarlo. Para evitar esto, es buena costumbre cerrar el bloque de código inmediatamente después de abrirlo y luego colocar el código que va a llevar en su interior. Si abrimos y colocamos el código, puede suceder que olvidemos cerrarlo.

El programa tiene varios errores, algunos de ellos muy comunes, por lo que es importante aprender a reconocerlos y resolverlos.

Para comenzar, lo primero que hacemos es llevar a cabo una compilación. Si hay un error de compilación, inmediatamente aparece una ventana. Esta ventana se conoce como lista de errores. Podemos observar esta ventana en la siguiente figura.

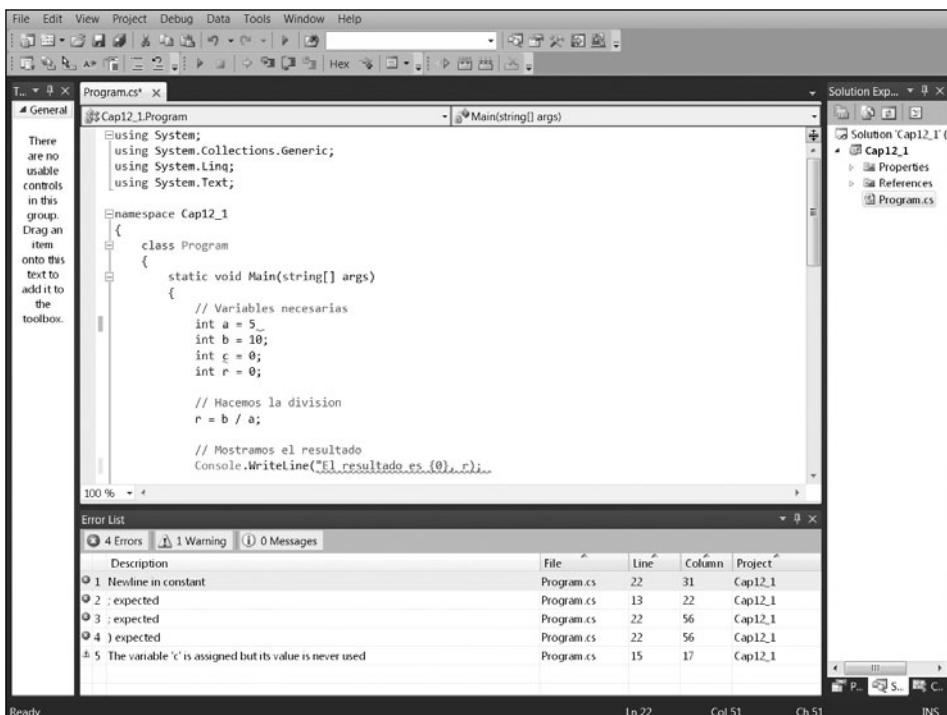


Figura 1. Despu s de compilar el programa, podemos encontrar f cilmente la lista de errores.

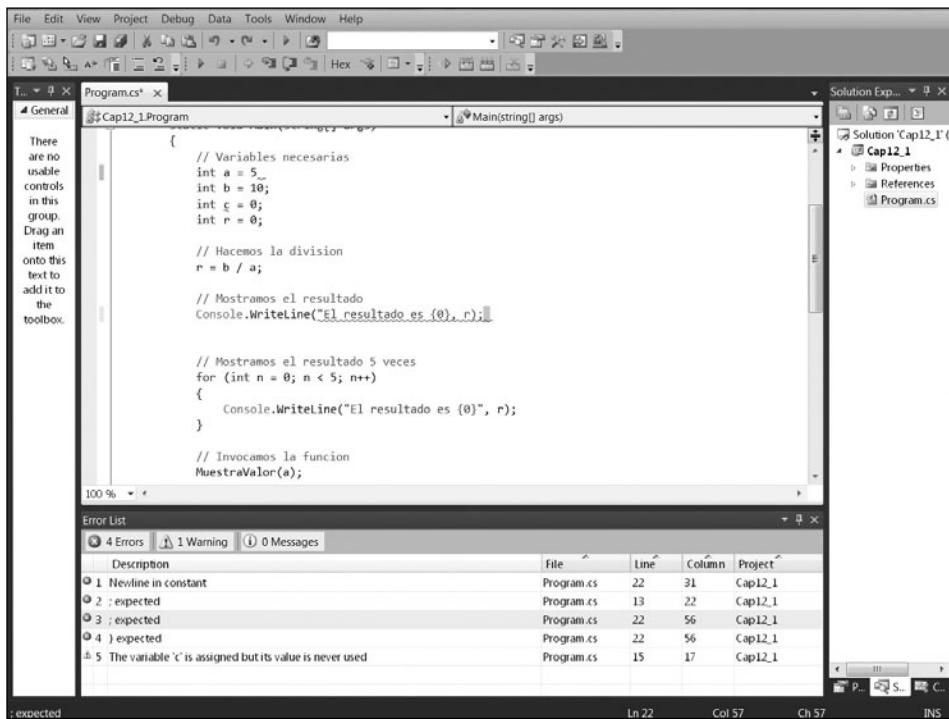
La ventana de la lista de errores contiene informaci n importante que nos permite encontrar el error r pidamente. La lista tiene varias columnas. La primera columna nos indica si el elemento listado es un **error** o una **advertencia**. Esto depende del icono mostrado. Un icono con señal amarilla es una advertencia y el icono rojo un error. En la segunda columna tenemos el **n mero secuencial** de error.

La tercera l nea es muy importante, ya que es una **descripci n** del error. Esta descripci n nos da pistas sobre la posible causa del error. La siguiente columna indica el **documento** en el cual ocurri  el error. Esto es \'til cuando tenemos programas que se constituyen en varios documentos. Las dos siguientes l neas nos dicen el **lugar** donde est  el posible error indicando la l nea y la columna. Un punto importante a tomar en cuenta es que la posici n es solamente un indicador. Nuestra \'ltima columna indica el **proyecto** en el cual ocurri  el error, esto llega a ser \'til cuando tenemos aplicaciones con m ltiples proyectos.

Si observamos el editor, veremos que hay texto que aparece con un subrayado rojo. Esto indica el texto que está relacionado con la lista de errores y facilita su localización. Empecemos a corregir los errores. La depuración es un proceso iterativo, por lo que tenemos que ir corrigiendo errores hasta que no quede ninguno. En algunas ocasiones, cuando corrijamos un error aparecerán listados nuevos errores. No debemos preocuparnos por esto y tenemos que continuar corrigiendo los errores listados hasta que no quede ninguno.

Cuando corregimos los problemas de nuestro programa, debemos referirnos a la lista de errores. Siempre tenemos que empezar con el primer problema. Esto se debe a que a veces un error en la parte superior del programa puede ocasionar que líneas que sean correctas parezcan que tienen problemas.

En la lista debemos dar doble clic al primer error. Esto nos lleva automáticamente al código y señala la parte con el problema. Esto lo podemos observar en el ejemplo que se muestra en la siguiente figura.



The screenshot shows the Microsoft Visual Studio IDE interface. The main window displays a C# code editor with the file 'Program.cs' open. The code contains several errors, notably at the end of a string constant and a missing closing brace. The 'Error List' window at the bottom left shows five errors and one warning. The first error is highlighted with a red underline and a yellow circle icon, indicating it is the current focus. The error details are: '1 Newline in constant' at line 22, column 31; '2 : expected' at line 13, column 22; '3 ; expected' at line 22, column 56; '4) expected' at line 22, column 17; and '5 The variable 'c' is assigned but its value is never used' at line 15, column 17. The 'Solution Explorer' window on the right shows the project structure for 'Cap12_1'.

Description	File	Line	Column	Project
1 Newline in constant	Program.cs	22	31	Cap12_1
2 : expected	Program.cs	13	22	Cap12_1
3 ; expected	Program.cs	22	56	Cap12_1
4) expected	Program.cs	22	56	Cap12_1
5 The variable 'c' is assigned but its value is never used	Program.cs	15	17	Cap12_1

Figura 2. Podemos observar que la línea donde se encuentra el problema ha sido señalada.

El problema señalado indica que se tiene una nueva línea en una constante. Esto generalmente ocurre cuando usamos una cadena y olvidamos cerrarla. Si observamos la línea veremos que efectivamente la cadena de formato no se ha cerrado. Procedamos a cerrar la cadena. De tal forma que la línea quede de la siguiente forma:

```
// Mostramos el resultado  
Console.WriteLine("El resultado es {}",r);
```

Siempre que corregimos un problema, es bueno realizar una compilación. Esto actualizará la lista de errores y podemos proceder con el siguiente. Después de la compilación nuestra lista de errores se ha reducido a únicamente dos.

El siguiente error que corregiremos es muy común. Si vemos la descripción nos indica que se estaba esperando `}`. Este error ocurre cuando hemos olvidado cerrar un bloque de código. Puede ser a nivel de **namespace**, clase, método, ciclo, etc. La mejor forma de resolverlo es verificar nuestros bloques de código y adicionar el cierre donde corresponde. En muchas ocasiones el cierre no necesariamente es donde lo señala el editor, por lo que se necesita revisar correctamente dónde colocar el cierre. En nuestro caso es muy sencillo y lo adicionamos al final.

Ya cerrado el bloque de código, podemos compilar y ver la lista de errores. Ahora únicamente nos aparece un error en la lista. Inicialmente teníamos cinco, hemos corregido dos y solo queda uno. Estos cambios en la cantidad de errores son muy comunes. Esto se debe a que los errores pueden suceder en cascada, como comentábamos, un error en la parte superior puede crear errores fantasma en otras partes.

El error que tenemos que corregir en este momento es muy sencillo y ocurre cuando olvidamos terminar la sentencia con `;`. Simplemente lo adicionamos en la línea que corresponde. Al parecer ya hemos terminado de corregir los problemas. Compilemos y veamos qué sucede.

Después de corregir el último error, han aparecido dos errores más. Como mencionábamos la depuración es iterativa y podemos pensar que el compilador hace varias pasadas. En este caso se nos presentan dos problemas en los métodos.

El primer error que tenemos nos indica que no se tiene una definición para **WriteLine**. Este tipo de error ocurre generalmente en dos casos. El primero es cuando estamos intentando usar algo, ya sea método, variable, objeto, etc. que no hemos definido. Para corregirlo simplemente definimos lo que necesitamos usar. El segundo caso es más común y se relaciona a un error de escritura, es decir que hemos escrito mal el nombre de lo que queremos utilizar y al no reconocerlo el compilador nos marca este error.

III PARA EVITAR ERRORES DE ESCRITURA

Una forma de evitar errores de escritura para variables y métodos es apoyarnos en la función de auto completar del editor. Esta función nos muestra el elemento que podemos colocar cuando escribimos las primeras letras del nombre. Al dar la tecla de **ENTER**, lo escribe por nosotros. Hacer una buena selección de nombres también ayuda a reducir los errores de escritura.

En nuestro ejemplo, el error está referenciado a un problema de escritura. Simplemente debemos ir al código y escribir **WriteLine** de la forma correcta. Compilemos y continuamos con el siguiente error.

Nuevamente nos queda un error. Este error nos dice que no hay un método sobre-cargado. Este error suele pasar cuando tenemos una función o método y estamos pasando una cantidad errónea de parámetros o los parámetros son de un tipo no correcto. En nuestro programa vemos que el método **MuestraValor()** necesita de un parámetro, pero en la invocación no se está pasando ningún valor y esto genera el problema. Para resolverlo simplemente debemos colocar el valor a pasar. Por ejemplo podemos colocar la línea de la siguiente forma:

```
// Invocamos la funcion
MuestraValor(a);
```

Ahora podemos compilar nuevamente. La ventana de salida nos indica que se ha compilado exitosamente la aplicación. Esto significa que ya no tenemos errores de compilación, pero aún es posible tener errores en tiempo de ejecución.

Cómo corregir los errores en tiempo de ejecución

Los errores en tiempo de ejecución se hacen notar cuando el programa está corriendo. El error puede provocar que el programa deje de funcionar o se termine repentinamente. En otros casos el programa no da resultados correctos.

Cuando el programa no da resultados correctos, esto puede deberse a que nuestro algoritmo no está adecuadamente implementado. En este caso, lo mejor es primero revisar el análisis y tratar de encontrar el error a nivel de algoritmo. Si encontramos el error en el algoritmo, simplemente debemos corregirlo y luego modificar el programa para que se ejecute de acuerdo con el algoritmo correcto.

En el caso de que nuestro algoritmo esté correcto y el programa nos provee con resultados erróneos entonces es posible que la implementación, es decir la forma como pasamos el algoritmo a programa, este equivocada. La mejor forma de corregir esto es ir depurando paso a paso el programa, viendo cómo cambian los valores y de esta forma encontrar dónde falló la implementación. Las técnicas para llevar a cabo esto las veremos un poco más adelante.

Ahora podemos ver los errores en tiempo de ejecución que provocan la finalización inesperada del programa. Muchos de estos errores se deben a que hemos realizado una operación indebida con nuestro programa. A los errores de este tipo se les llama **excepciones** y cuando ocurren se dice que se han **levantado o aventado**.

Veamos cómo ocurren estos errores. Simplemente ejecutemos el programa anterior. El programa está listo para levantar una excepción.

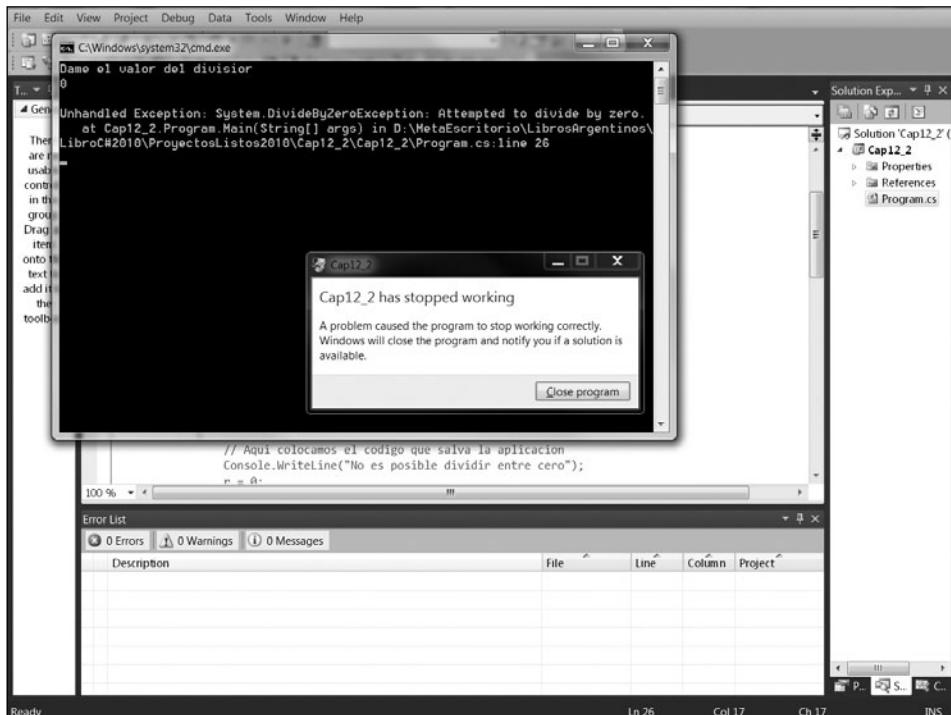


Figura 3. En cuanto el programa levanta una excepción, finaliza su ejecución de forma inesperada y nos presenta un diálogo con información.

Como vemos el programa ha levantado una excepción. Para poder depurar el programa debemos ejecutarlo en modo de depuración. Para esto seleccionamos el menú de **Debug** o **Depurar** y presionamos **Start Debugging**.

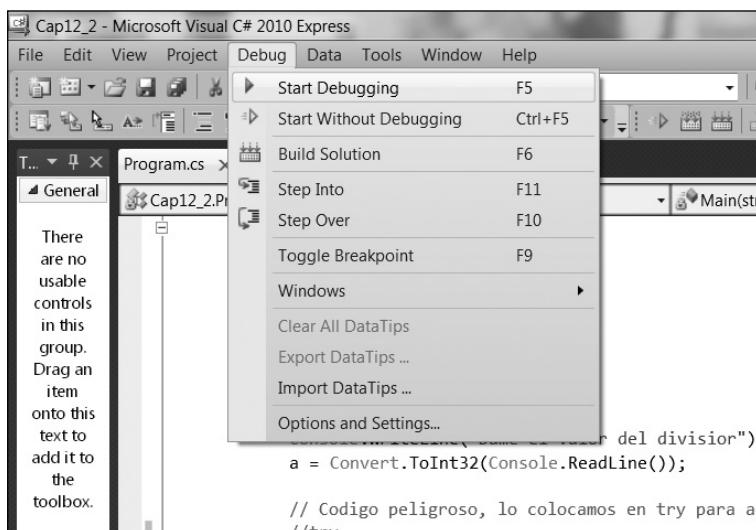


Figura 4. Aquí podemos seleccionar dónde llevar a cabo la depuración.

Ya que nos encontramos en el depurador, el lugar donde se generó la excepción aparece marcado en forma muy clara, y junto a él tendremos un cuadro de diálogo que contiene la información correspondiente.

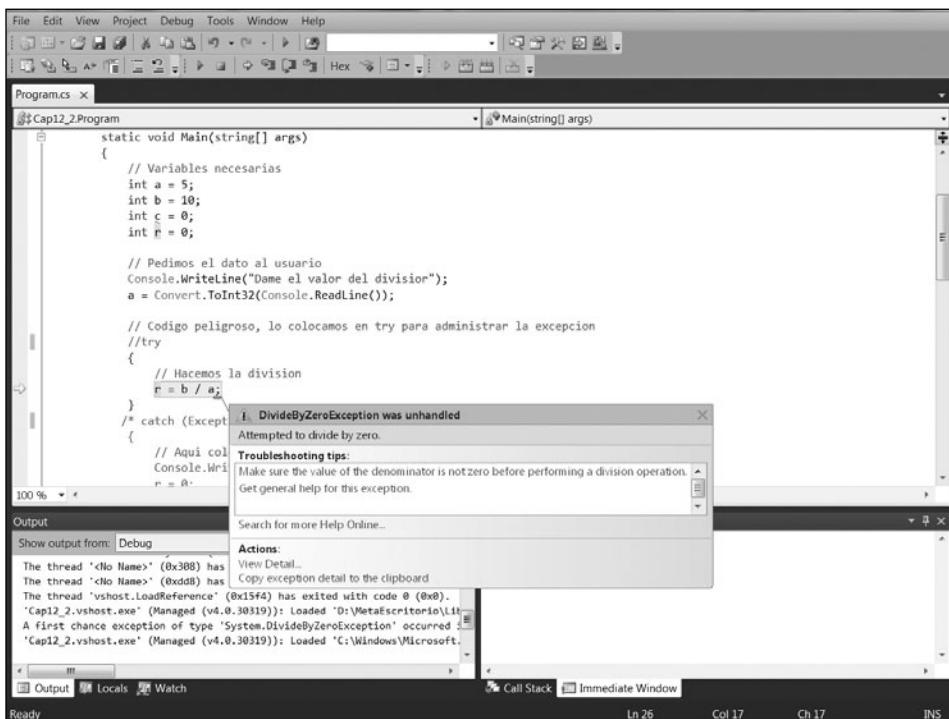


Figura 5. El cuadro de diálogo nos indica el tipo de excepción que se levantó.

Si observamos el cuadro de diálogo veremos que la excepción, es decir el error, fue provocado por un intento de dividir entre cero. La división entre cero no está definida y por eso es una operación inválida. Ahora que ya sabemos qué provoca el problema, simplemente podemos corregir el código.

Podemos pensar que la corrección es sencilla y simplemente dividir entre la variable que tiene un valor de cero, puede ser algo como lo que se muestra en el siguiente bloque de código, a modo de ejemplo:

III LA DESCRIPCIÓN DEL ERROR

Debemos acostumbrarnos a leer la descripción del error, es un punto muy importante. Con el tiempo nos acostumbraremos a la descripción y sabremos inmediatamente qué hacer para resolver el problema. Muchos programadores con malos hábitos solamente van a la línea del error, pero no leen la descripción del error. Esto los lleva a tiempos más largos para resolverlo.

```
// Hacemos la division
r = b / a;
```

Si procedemos con la compilación, no aparece ningún problema, pero veamos qué sucede cuando ejecutemos nuevamente el programa.

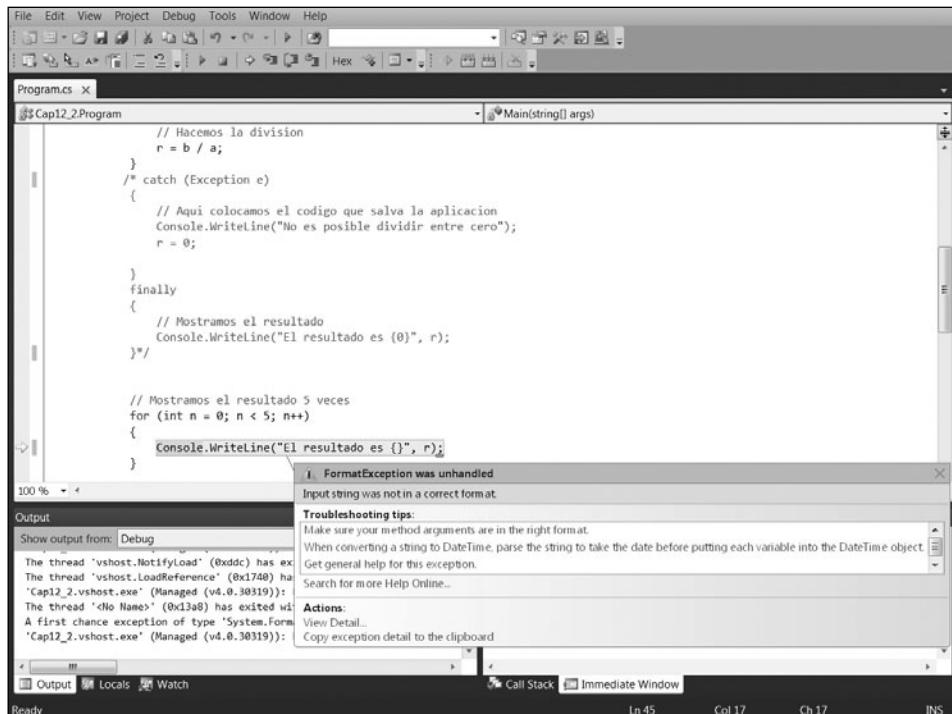


Figura 6. Una nueva excepción ha ocurrido en el programa.

El programa nuevamente tiene problemas de ejecución. Si observamos la información que nos provee el depurador inmediatamente nos damos cuenta de que la excepción ha ocurrido porque no estamos colocando el formato correcto en la cadena. Al ver la línea de código podemos identificar que efectivamente el formato es incorrecto, la línea con el formato adecuado debe ser como la siguiente:

```
// Mostramos el resultado
Console.WriteLine("El resultado es {0}", r);
```

Después de corregir, nuevamente debemos compilar y ejecutar. Todo debe estar bien y el programa puede ejecutarse sin problema. El resultado lo vemos en el ejemplo que se muestra en la siguiente figura:

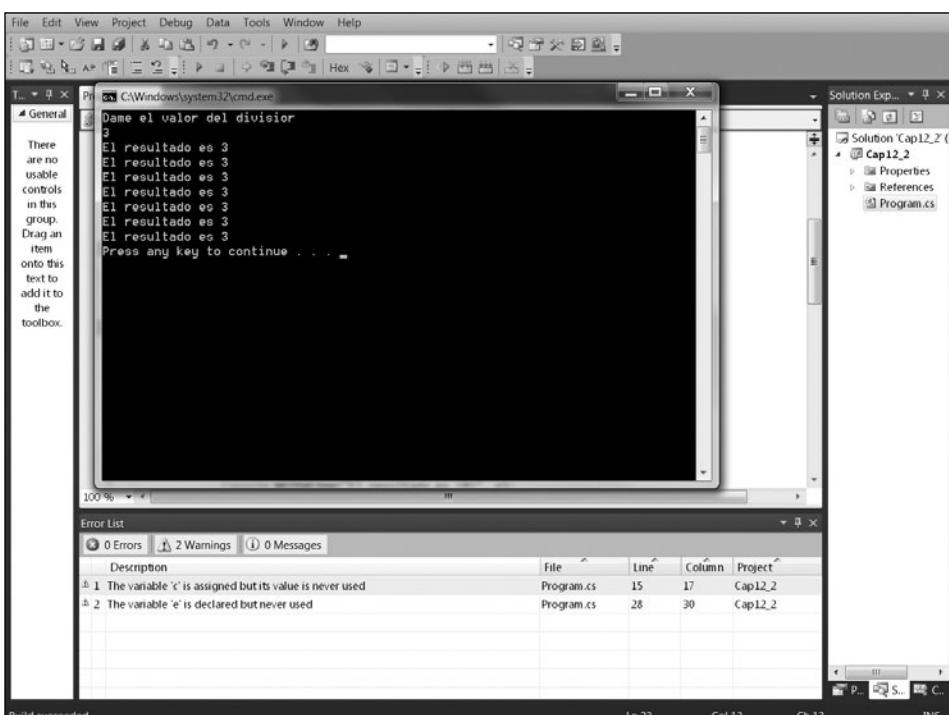


Figura 7. El programa por fin se ejecuta y podemos ver en la consola los resultados.

Ya vimos cómo localizar el código que nos genera excepciones. Pero lo mejor sería poder evitarlas o controlarlas de alguna manera.

Cómo manejar los errores

No es posible evitar las excepciones completamente. Esto se debe a que no siempre tenemos control sobre la información o el manejo que va a tener el usuario en nuestro programa. Por ejemplo, en el programa anterior, la excepción se eliminó al dividir entre la variable que no tiene el valor de cero. Pero imaginemos que el usuario es el que debe colocar el valor del divisor. En este caso no podemos forzar a una variable con un valor diferente de cero. En algunas ocasiones el usuario colocará un valor adecuado, pero también es posible que dé el valor de cero. En esos casos, el programa terminará por el error.

Como no podemos prever todas las excepciones que pueden ocurrir, lo mejor que podemos hacer es tener una forma de manejar o administrar los errores. Esto permitirá que el programa detecte la excepción y entonces puede hacer algo para salvar la ejecución de programa.

C# nos permite hacer esto ya que provee un sistema de manejo estructurado de excepciones. Para ver cómo poder lograr esto, modificaremos el programa y le daremos el manejo de las excepciones. El programa modificado queda así:

```
using System;
using System.Collections.Generic;
using System.Text;

namespace Cap12_2
{
    class Program
    {
        static void Main(string[] args)
        {
            // Variables necesarias
            int a = 5;
            int b = 10;
            int c = 0;
            int r = 0;

            // Pedimos el dato al usuario
            Console.WriteLine("Dame el valor del divisor");
            a = Convert.ToInt32(Console.ReadLine());

            // Hacemos la division
            r = b / a;

            // Mostramos el resultado
            Console.WriteLine("El resultado es {0}", r);

            // Mostramos el resultado 5 veces
            for (int n = 0; n < 5; n++)
            {
                Console.WriteLine("El resultado es {0}", r);
            }

            // Invocamos la funcion
            MuestraValor(a);

        }

        static void MuestraValor(int n)
        {
```

```

        Console.WriteLine("El resultado es {0}", n);

    }
}

}

```

Veamos cómo este programa puede ejecutarse correctamente a veces y con problemas en otros casos. Si ejecutamos el programa y damos como valor 5, el programa se ejecuta correctamente. Cuando ejecutemos el programa nuevamente, si el valor que damos es cero, entonces la excepción ocurrirá. Esto lo vemos claramente en las dos figuras que se mostrarán a continuación.

Es sencillo saber qué métodos pueden generar excepciones. Simplemente debemos ir a MSDN y buscar la documentación del método a utilizar. Dentro de esta documentación hay una sección que describe las posibles excepciones que pueden ocurrir. De esta forma hacemos que nuestro programa las evite.

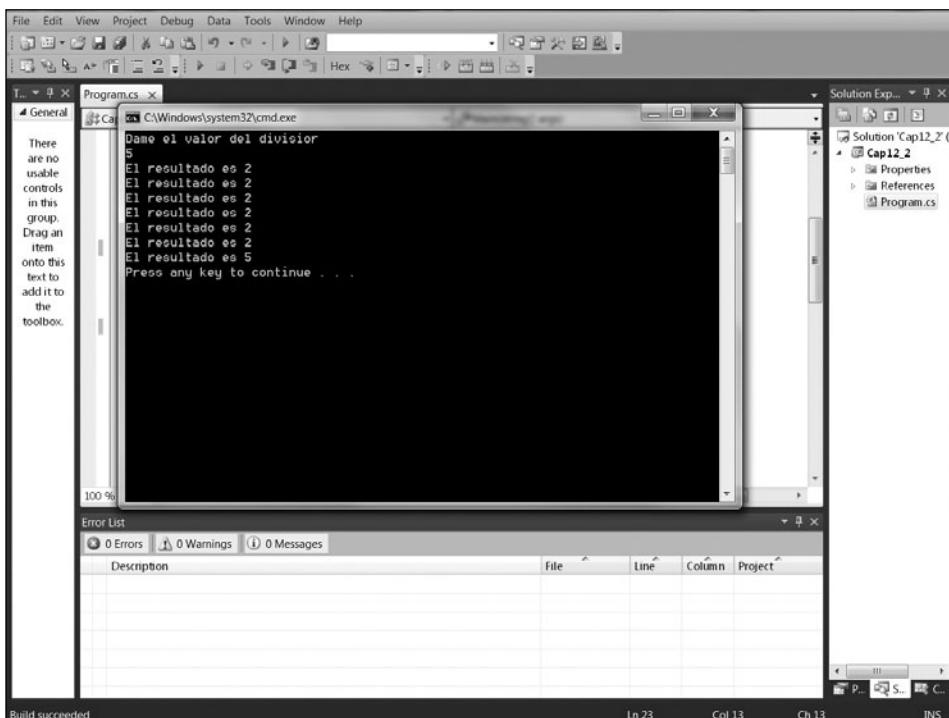


Figura 8. En esta figura observamos el caso en el cual la aplicación funciona correctamente.

Para llevar a cabo la administración de excepciones vamos a tener tres bloques de código conocidos como: **try**, **catch** y **finally**.

Lo primero que tenemos que hacer es detectar dónde se encuentra el código peligroso, es decir, dónde puede ocurrir la excepción. En nuestro caso es fácil de identificar ya que sabemos que es la división.

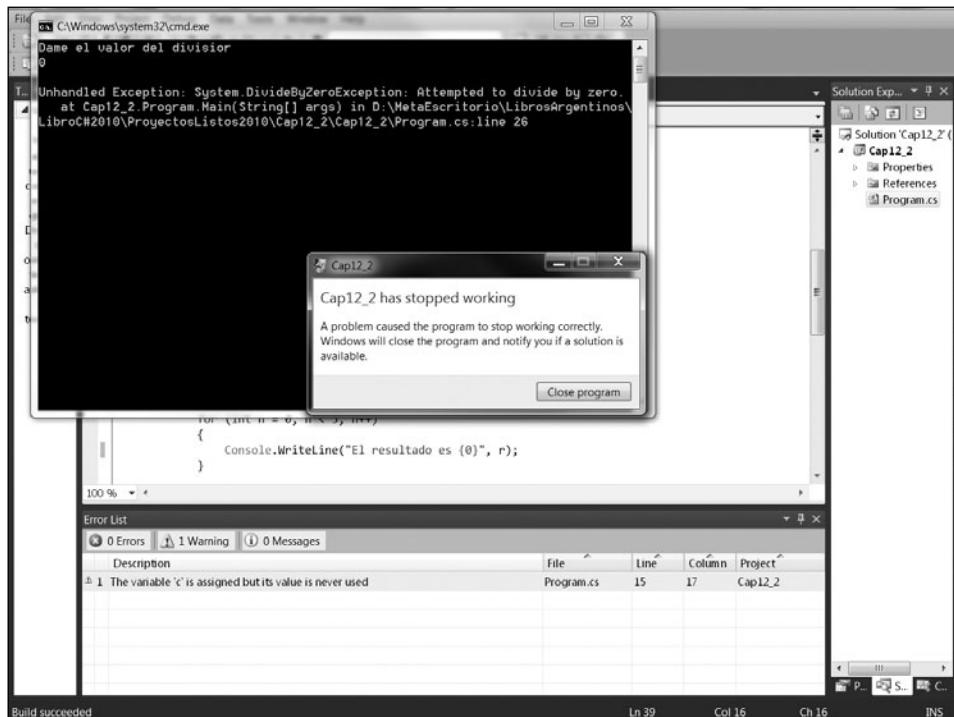


Figura 9. Ahora podemos observar cómo la excepción se ha levantado y el programa finaliza su ejecución.

El código que tiene el riesgo de levantar la excepción debe ser colocado en el bloque de **try**. Por ejemplo, en nuestro caso colocaremos lo siguiente:

```
// Código peligroso, lo colocamos en try para
// administrar la excepción
```

III NO EXAGERAR CON LAS EXCEPCIONES

Existen muchos métodos y sentencias que pueden generar excepciones. Pero no es bueno exagerar. Una gran cantidad de excepciones se evitan simplemente al usar correctamente los métodos y hacer buen uso de nuestras variables y sus valores. Solamente debemos implementar la administración de excepciones en las partes del programa que realmente lo necesiten.

```

try
{
    // Hacemos la division
    r = b / a;
}

```

El código adentro de **try** tiene dos opciones: levantar o no la excepción. Si la excepción se levanta entonces en el bloque de **catch** podemos colocar el código que se encargue de evitar que el programa termine inesperadamente. Aquí podemos poner código que corrija lo ocurrido.

En nuestro caso, no hay mucho que hacer, simplemente podemos enviar un mensaje al usuario y dar un valor adecuado a la variable de resultado.

```

catch (Exception e)
{
    // Aqui colocamos el codigo que salva la apliacion
    Console.WriteLine("No es posible dividir entre cero");
    r = 0;

}

```

Vemos que este bloque tiene un parámetro llamado *e*. Adentro de este parámetro podemos encontrar información relacionada con la excepción. Lo que coloquemos en el interior del bloque depende de la lógica de nuestra aplicación. Si no hubiera excepción el código adentro del bloque **catch** no se ejecuta.

Tenemos otro bloque llamado **finally**. Este bloque es opcional. El código que se coloque en el bloque **finally** se ejecuta siempre sin importar si la excepción se llevó a cabo o no. Siempre se ejecuta una vez que **try** o **catch** han finalizado de ejecutarse. **Finally** puede ser utilizado para limpiar o liberar cualquier recurso que se haya utilizado en **try** o **catch**. En nuestro caso puede utilizarse simplemente para darle al usuario el valor del resultado.

```

finally
{
    // Mostramos el resultado
    Console.WriteLine("El resultado es {0}", r);
}

```

Nuestro programa completo con la administración de la excepción es el siguiente:

```
using System;
using System.Collections.Generic;
using System.Text;

namespace Cap12_2
{
    class Program
    {
        static void Main(string[] args)
        {
            // Variables necesarias
            int a = 5;
            int b = 10;
            int c = 0;
            int r = 0;

            // Pedimos el dato al usuario
            Console.WriteLine("Dame el valor del divisor");
            a = Convert.ToInt32(Console.ReadLine());

            // Código peligroso, lo colocamos en try para
            // administrar la excepción
            try
            {
                // Hacemos la división
                r = b / a;
            }
            catch (Exception e)
            {
                // Aquí colocamos el código que salva la aplicación
                Console.WriteLine("No es posible dividir entre cero");
                r = 0;
            }
            finally
            {
                // Mostramos el resultado
                Console.WriteLine("El resultado es {0}", r);
            }
        }
    }
}
```

```

    }

    // Mostramos el resultado 5 veces
    for (int n = 0; n < 5; n++)
    {
        Console.WriteLine("El resultado es {0}", r);
    }

    // Invocamos la función
    MuestraValor(a);

}

static void MuestraValor(int n)
{
    Console.WriteLine("El resultado es {0}", n);

}
}
}

```

Veamos si esto funciona adecuadamente. Compilemos el programa y al ejecutarlo vamos a dar el valor de cero. El programa ahora no deberá generar problemas, pues hemos capturado la excepción y ejecutado código que evita que el programa finalice de forma no adecuada.



RESUMEN

La depuración nos permite corregir los problemas que pueda tener nuestro programa. Los errores de compilación impiden que el programa pueda ser compilado y muchas veces son simples errores de sintaxis. Tenemos una ventana que nos muestra la lista de errores, siempre debemos corregir el primer error de la lista, ya que muchos errores se generan en cascada. La depuración es un proceso iterativo. Si tenemos código peligroso que puede generar excepciones, es recomendable administrar ese código para evitar que la excepción termine con nuestro programa, cuando administraremos la excepción, podemos colocar código que salve al programa.



ACTIVIDADES

TEST DE AUTOEVALUACIÓN

1 ¿Qué es la depuración?

2 ¿Cuántos tipos de errores tenemos?

3 ¿Por qué algunos programas no pueden compilar?

4 ¿Qué es un error de sintaxis?

5 ¿En dónde podemos ver los errores que tenemos?

6 ¿Qué son los errores en tiempo de ejecución?

7 ¿Qué es un error de lógica?

8 ¿Qué es una excepción?

9 ¿Qué es la administración de excepciones?

10 ¿Cuáles son los bloques de código para administrar la excepción?

11 ¿Qué es un punto de interrupción?

12 ¿Cómo podemos depurar paso a paso?

EJERCICIOS PRÁCTICOS

1 Usar la depuración paso a paso para observar cómo cambia el valor de la variable en el programa del factorial.

2 Utilizar el método **WriteLine()** de la clase **Debug** para que las funciones nos indiquen cuando entramos y salimos de ellas.

3 Buscar en MSDN cuáles son las excepciones que pueden ocurrir con los arreglos.

4 Buscar en MSDN cuáles son las excepciones que pueden ocurrir con los streams.

5 Buscar en MSDN cuáles son las excepciones que pueden ocurrir con el método **WriteLine()**.

C#

Servicios al lector

En este último apartado encontraremos un índice temático con las palabras más significativas de esta obra.

Una guía que nos servirá para encontrar con mayor facilidad el contenido que necesitamos.

ÍNDICE TEMÁTICO

#			
.NET	15	Catch	387
16 bits	16	Char	44
		Chars	261
		Ciclos	111
		Ciclos enlazados	198
	A		
Acceso privado	319	CIL	16, 17
Acceso protegido	319	Clase	23
Acceso público	319	Clases	318
Acumulador	123	Clear()	236, 245
Add()	219	CLR	15
Algoritmo	33	CLS	16
API	14	Colecciones	218
Append	363	COM	15
Append()	262	Comentarios	45, 46
AppendFormat()	262	Compare()	263
ArrayList	218	Compilador	15
Arreglos de dos dimensiones	196	Concatenar	260
Arreglos de una dimensión	189	Console	27, 35
Arreglos	188	Console Application	20
ASCII	364	Constructor	238
ASCIIEncoding	364	Contador	123
Assembly	16, 17	Contains()	236, 244, 264
		ContainsValue()	252
	B		
Basic	14	Convert	57
Begin	353	CopyTo()	267
Bloque de código	34	Count	245
Bloque de comentarios	46	Cpp	21
Bool	44	Create	363
Buffer	359	CreateNew	363
Build	24	CultureInfo	274
Byte	44/350	Current	353
Bytes	350		
	C		
C++	14	D.O.S.	25
Cadenas	256	DateTime	257
Case	105	Datos públicos	324
		Debug	25
		Decimal	44
	D		

Decrementos	125	H	
Depuración	373	Hashtable	218, 249
Depurar	25	Herencia	319
Dequeue	242, 244	I	
Diagnostics	389	If	76
Diagrama de flujo	61	If anidados	89
Disyunción	98	If-else	84
Do while	134	Incrementos	125
Double	44, 51	Indentación	92
E		IndexOf()	224
Else	84	Insert()	222, 268
Encapsulamiento	319	Instanciación	319
End	353	Int	43, 44
EndsWith()	266	J	
Enqueue	242	Jagged	205
Enumeraciones	309	Java	14
Equals()	264	Jitter	17
Errores de lógica	38	L	
Errores de sintaxis	38	LastIndexOf()	269
Estructuras enlazadas	299	Length	262
Excepciones	380	LIFO	232
Explorador de soluciones	21	Línea de comandos	25
Expresiones lógicas	96	Long	45
F		M	
FIFO	241	Main()	23, 28
FileMode	363	Math	22
FileStream	369	Matriz	196
Finally	387	MemoryStream	351
Float	45, 51	Métodos	147
Flujos	350	Mono	18
For	112	MSDN	57
Foreach	225	Multiplataforma	16
Format	257	N	
Framework .NET	15	Namespace	22
Funciones	147, 148, 156	Not	96
G		Now	257
Get	329		
GetBytes()	364		

O			
Open	363	Split()	272
OpenOrCreate	363	Stack	232
Operador	49	Static	149, 154
Operador de asignación	45	Streams	350
Operadores	73	String	45, 256
Orientado a objetos	34	StringBuilder	261
Override	287, 297	SubString()	265
OWL	14	Switch	105
		System.IO	364
P		T	
PadLeft()	270	Tipo	43
PadRight()	271	ToInt32()	57
Paso por copia	178	ToLower()	273
Paso por referencia	178	ToSingle()	58
Passport	18	ToString()	256
Peek()	244	ToUpper()	274
Pop	233, 235	Trim()	275
Private	281	TrimEnd()	276
Proceso	62	TrimStart()	276
Protección de datos	329	Truncate	363
Proyecto	19	Try	387
Public	281	Type Cast	312
Push	233		
R		U	
ReadLine()	56, 172	Uint	45
Recolector de basura	17	Uling	45
Recursión	183	UML	321
RemoveAt()	223	Ushort	45
Replace()	271	Using	22
Runtime	14		
S		V	
Sbyte	44	Variable	42
Seek()	354	Vista de clases	28
SeekOrigin	353	Visual Basic	14
Set	329	Visual Studio 2008	18
SetValue()	239		
Singleton	338	W	
Solución	20	While	141
		Write()	41
		WriteLine	35

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 hasta un CD-ROM, desde un Servicio de Atención al Lector hasta la posibilidad de leer el sumario en la Web para evaluar con tranquilidad la compra, o la presencia de adecuados índices temáticos, todo suma al valor de un buen libro.

5 VERIFIQUE EL IDIOMA

No sólo el del texto; también revise que las pantallas incluidas en el libro estén en el mismo idioma del programa que usted utiliza.

6 REVISE LA FECHA DE PUBLICACIÓN

Está en letra pequeña en las primeras páginas; si es un libro traducido, la que vale es la fecha de la edición original.



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.

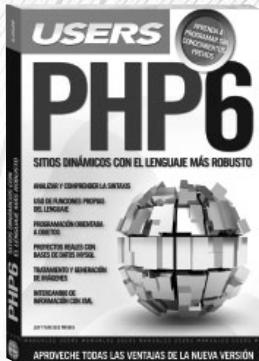
TAMBIÉN PUEDE CONSEGUIR NUESTROS LIBROS EN KIOSCOS O PUESTOS DE PERIÓDICOS, LIBRERÍAS, CADENAS COMERCIALES, SUPERMERCADOS Y CASAS DE COMPUTACIÓN.



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

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

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



PHP 6

Este libro es un completo curso de programación en PHP en su versión 6.0. Un lenguaje que se destaca tanto por su versatilidad como por el respaldo de una amplia comunidad de desarrolladores lo cual lo convierte en un punto de partida ideal para quienes comienzan a programar.

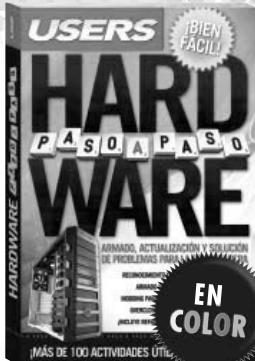
→ COLECCIÓN: MANUALES USERS
→ 368 páginas / ISBN 978-987-663-039-9



200 Respuestas: Blogs

Esta obra es una completa guía que responde a las preguntas más frecuentes de la gente sobre la forma de publicación más poderosa de la Web 2.0. Definiciones, consejos, claves y secretos, explicados de manera clara, sencilla y didáctica.

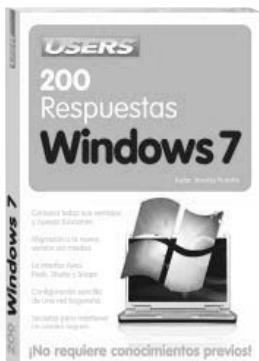
→ COLECCIÓN: 200 RESPUESTAS
→ 320 páginas / ISBN 978-987-663-037-5



Hardware paso a paso

En este libro encontraremos una increíble selección de actividades que abarcan todos los aspectos del hardware. Desde la actualización de la PC hasta el overclocking de sus componentes, todo en una presentación nunca antes vista, realizada íntegramente con procedimientos paso a paso.

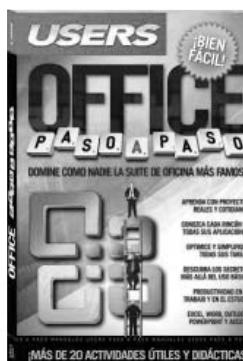
→ COLECCIÓN: PASO A PASO
→ 320 páginas / ISBN 978-987-663-034-4



200 Respuestas: Windows 7

Esta obra es una guía básica que responde, en forma visual y práctica, a todas las preguntas que necesitamos conocer para dominar la última versión del sistema operativo de Microsoft. Definiciones, consejos, claves y secretos, explicados de manera clara, sencilla y didáctica.

→ COLECCIÓN: 200 RESPUESTAS
→ 320 páginas / ISBN 978-987-663-035-1



Office paso a paso

Este libro presenta una increíble colección de proyectos basados en la suite de oficina más usada en el mundo. Todas las actividades son desarrolladas con procedimientos paso a paso de una manera didáctica y fácil de comprender.

→ COLECCIÓN: PASO A PASO
→ 320 páginas / ISBN 978-987-663-030-6



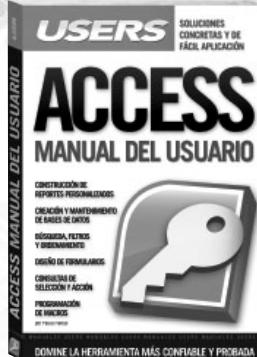
101 Secretos de Hardware

Esta obra es la mejor guía visual y práctica sobre hardware del momento. En su interior encontraremos los consejos de los expertos sobre las nuevas tecnologías, las soluciones a los problemas más frecuentes, cómo hacer overclocking, modding, y muchos más trucos y secretos.

→ COLECCIÓN: MANUALES USERS
→ 352 páginas / ISBN 978-987-663-029-0

iLéalo antes Gratis!

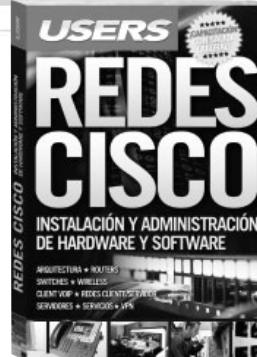
En nuestro sitio, obtenga GRATIS un capítulo del libro de su elección antes de comprarlo.



Access

Este manual nos introduce de lleno en el mundo de Access para aprender a crear y administrar bases de datos de forma profesional. Todos los secretos de una de las principales aplicaciones de Office, explicados de forma didáctica y sencilla.

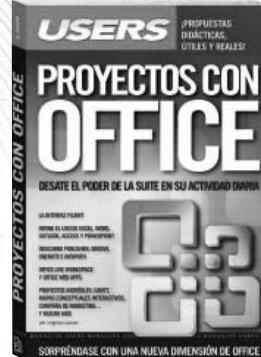
- COLECCIÓN: MANUALES USERS
→ 320 páginas / ISBN 978-987-663-025-2



Redes Cisco

Este libro permitirá al lector adquirir todos los conocimientos necesarios para planificar, instalar y administrar redes de computadoras. Todas las tecnologías y servicios Cisco, desarrollados de manera visual y práctica en una obra única.

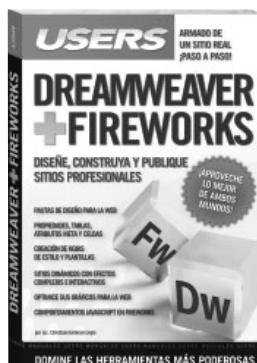
- COLECCIÓN: MANUALES USERS
→ 320 páginas / ISBN 978-987-663-024-5



Proyectos con Office

Esta obra nos enseña a usar las principales herramientas de Office a través de proyectos didácticos y útiles. En cada capítulo encontraremos la mejor manera de llevar adelante todas las actividades del hogar, la escuela y el trabajo.

- COLECCIÓN: MANUALES USERS
→ 352 páginas / ISBN 978-987-663-023-8



Dreamweaver y Fireworks

Esta obra nos presenta las dos herramientas más poderosas para la creación de sitios web profesionales de la actualidad. A través de procedimientos paso a paso, nos muestra cómo armar un sitio real con Dreamweaver y Fireworks sin necesidad de conocimientos previos.

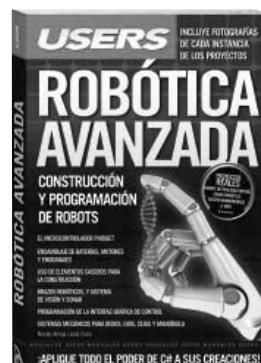
- COLECCIÓN: MANUALES USERS
→ 320 páginas / ISBN 978-987-663-022-1



Excel revelado

Este manual contiene una selección de más de 150 consultas de usuarios de Excel y todas las respuestas de Claudio Sánchez, un reconocido experto en la famosa planilla de cálculo. Todos los problemas encuentran su solución en esta obra imperdible.

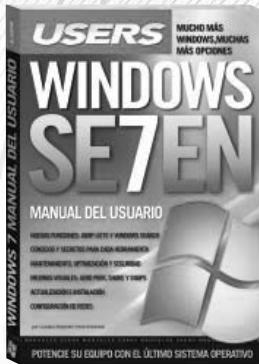
- COLECCIÓN: MANUALES USERS
→ 336 páginas / ISBN 978-987-663-021-4



Robótica avanzada

Esta obra nos permitirá ingresar al fascinante mundo de la robótica. Desde el ensamblaje de las partes hasta su puesta en marcha, todo el proceso está expuesto de forma didáctica y sencilla para así crear nuestros propios robots avanzados.

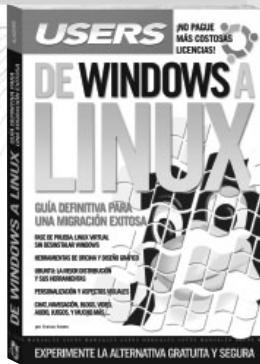
- COLECCIÓN: MANUALES USERS
→ 352 páginas / ISBN 978-987-663-020-7



Windows 7

En este libro, encontraremos las claves y los secretos destinados a optimizar el uso de nuestra PC tanto en el trabajo como en el hogar. Aprenderemos a llevar adelante una instalación exitosa y a utilizar todas las nuevas herramientas que incluye esta versión.

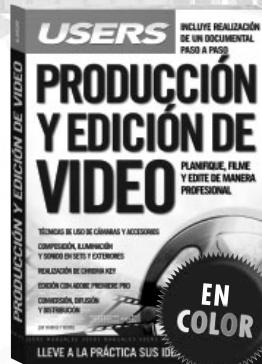
→ COLECCIÓN: MANUALES USERS
→ 320 páginas / ISBN 978-987-663-015-3



De Windows a Linux

Esta obra nos introduce en el apasionante mundo del software libre a través de una completa guía de migración, que parte desde el sistema operativo más conocido: Windows. Aprenderemos cómo realizar gratuitamente aquellas tareas que antes hacíamos con software pago.

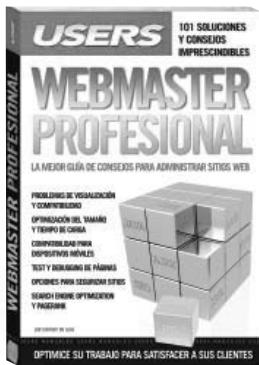
→ COLECCIÓN: MANUALES USERS
→ 336 páginas / ISBN 978-987-663-013-9



Producción y edición de video

Un libro ideal para quienes deseen realizar producciones audiovisuales con bajo presupuesto. Tanto estudiantes como profesionales encontrarán cómo adquirir las habilidades necesarias para obtener una salida laboral con una creciente demanda en el mercado.

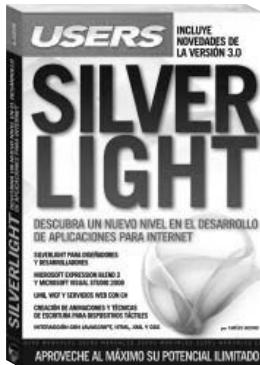
→ COLECCIÓN: MANUALES USERS
→ 336 páginas / ISBN 978-987-663-012-2



Webmaster profesional

Este obra explica cómo superar los problemas más frecuentes y complejos que enfrenta todo administrador de sitios web. Ideal para quienes necesiten conocer las tendencias actuales y las tecnologías en desarrollo que son materia obligada para dominar la Web 2.0.

→ COLECCIÓN: MANUALES USERS
→ 336 páginas / ISBN 978-987-663-011-5



Silverlight

Este manual nos introduce en un nuevo nivel en el desarrollo de aplicaciones interactivas a través de Silverlight, la opción multiplataforma de Microsoft. Quien consiga dominarlo creará aplicaciones visualmente impresionantes, acordes a los tiempos de la incipiente Web 3.0.

→ COLECCIÓN: MANUALES USERS
→ 352 páginas / ISBN 978-987-663-010-8



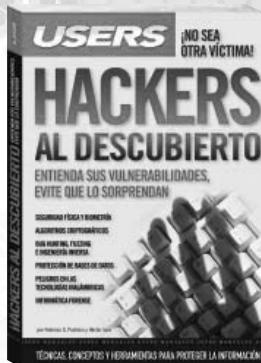
Flash extremo

Este libro nos permitirá aprender a fondo Flash CS4 y ActionScript 3.0 para crear aplicaciones web y de escritorio. Una obra imperdible sobre uno de los recursos más empleados en la industria multimedia, que nos permitirá estar a la vanguardia del desarrollo.

→ COLECCIÓN: MANUALES USERS
→ 320 páginas / ISBN 978-987-663-009-2

iLéalo antes Gratis!

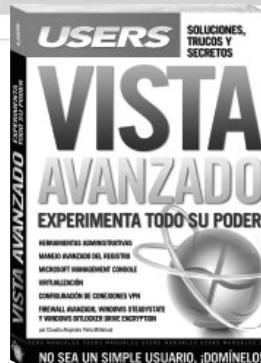
En nuestro sitio, obtenga GRATIS un capítulo del libro de su elección antes de comprarlo.



Hackers al descuberto

Esta obra presenta un panorama de las principales técnicas y herramientas utilizadas por los hackers, y de los conceptos necesarios para entender su manera de pensar, prevenir sus ataques y estar preparados ante las amenazas más frecuentes.

- COLECCIÓN: MANUALES USERS
→ 352 páginas / ISBN 978-987-663-008-5



Vista avanzado

Este manual es una pieza imprescindible para convertirnos en administradores expertos de este popular sistema operativo. En sus páginas haremos un recorrido por las herramientas fundamentales para tener máximo control sobre todo lo que sucede en nuestra PC.

- COLECCIÓN: MANUALES USERS
→ 352 páginas / ISBN 978-987-663-007-8



101 Secretos de Excel

Una obra absolutamente increíble, con los mejores 101 secretos para dominar el programa más importante de Office. En sus páginas encontraremos un material sin desperdicios que nos permitirá realizar las tareas más complejas de manera sencilla.

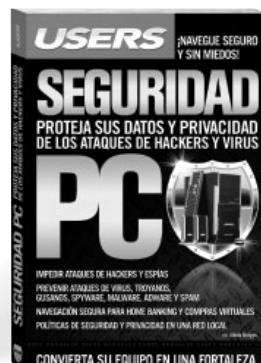
- COLECCIÓN: MANUALES USERS
→ 336 páginas / ISBN 978-987-663-005-4



Electrónica & microcontroladores PIC

Una obra ideal para quienes desean aprovechar al máximo las aplicaciones prácticas de los microcontroladores PIC y entender su funcionamiento. Un material con procedimientos paso a paso y guías visuales, para crear proyectos sin límites.

- COLECCIÓN: MANUALES USERS
→ 368 páginas / ISBN 978-987-663-002-3



Seguridad PC

Este libro contiene un material imprescindible para proteger nuestra información y privacidad. Aprendere-mos cómo reconocer los síntomas de infección, las medidas de prevención por tomar, y finalmente, la manera de solucionar los problemas.

- COLECCIÓN: MANUALES USERS
→ 336 páginas / ISBN 978-987-663-004-7



Hardware desde cero

Este libro brinda las herramientas necesarias para entender de manera amena, simple y ordenada cómo funcionan el hardware y el software de la PC. Está destinado a usuarios que quieran independizarse de los especialistas necesarios para armar y actualizar un equipo.

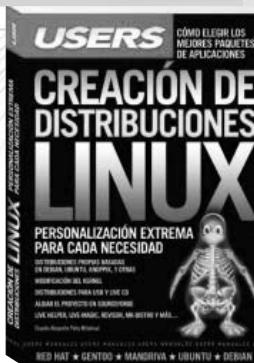
- COLECCIÓN: MANUALES USERS
→ 320 páginas / ISBN 978-987-663-001-6



200 Respuestas: Photoshop

Esta obra es una guía que responde, en forma visual y práctica, a todas las preguntas que necesitamos contestar para conocer y dominar Photoshop CS3. Definiciones, consejos, claves y secretos, explicados de manera clara, sencilla y didáctica.

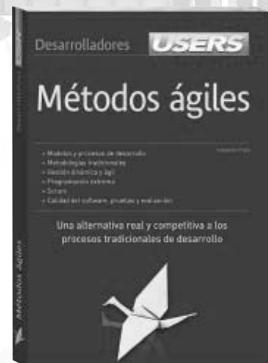
→ COLECCIÓN: 200 RESPUESTAS
→ 320 páginas / ISBN 978-987-1347-98-8



Creación de distribuciones Linux

En este libro recorremos todas las alternativas para crear distribuciones personalizadas: desde las más sencillas y menos customizables, hasta las más avanzadas, que nos permitirán modificar el corazón mismo del sistema, el kernel.

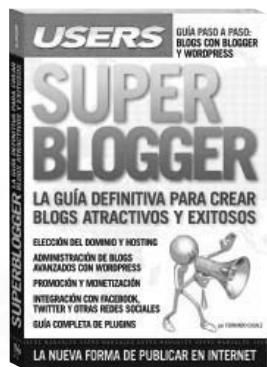
→ COLECCIÓN: MANUALES USERS
→ 336 páginas / ISBN 978-987-1347-99-5



Métodos ágiles

Este libro presenta una alternativa competitiva a las formas tradicionales de desarrollo y los últimos avances en cuanto a la producción de software. Ideal para quienes sientan que las técnicas actuales les resultan insuficientes para alcanzar metas de tiempo y calidad.

→ COLECCIÓN: DESARROLLADORES
→ 336 páginas / ISBN 978-987-1347-97-1



SuperBlogger

Esta obra es una guía para sumarse a la revolución de los contenidos digitales. En sus páginas, aprenderemos a crear un blog, y profundizaremos en su diseño, administración, promoción y en las diversas maneras de obtener dinero gracias a Internet.

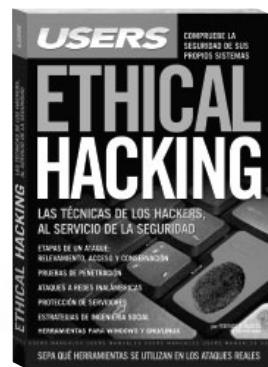
→ COLECCIÓN: MANUALES USERS
→ 352 páginas / ISBN 978-987-1347-96-4



UML

Este libro es la guía adecuada para iniciarse en el mundo del modelado. Conoceremos todos los constructores y elementos necesarios para comprender la construcción de modelos y razonarlos de manera que reflejen los comportamientos de los sistemas.

→ COLECCIÓN: DESARROLLADORES
→ 320 páginas / ISBN 978-987-1347-95-7



Ethical Hacking

Esta obra expone una visión global de las técnicas que los hackers maliciosos utilizan en la actualidad para conseguir sus objetivos. Es una guía fundamental para obtener sistemas seguros y dominar las herramientas que permiten lograrlo.

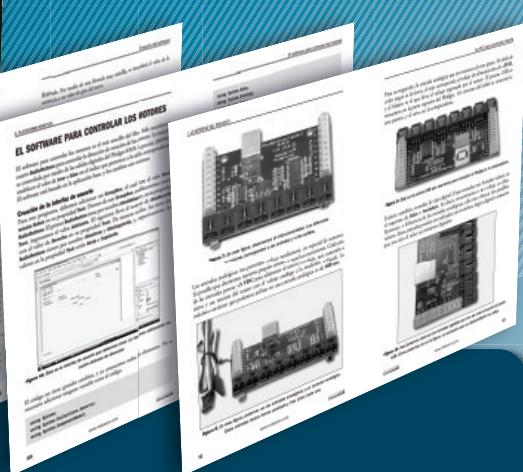
→ COLECCIÓN: MANUALES USERS
→ 320 páginas / ISBN 978-987-1347-93-3

DISEÑO, ARMADO Y PROGRAMACIÓN DE ROBOTS COMPLEJOS



Esta obra nos permitirá ingresar al fascinante mundo de la robótica. Desde el ensamblaje de las partes hasta su puesta en marcha, todo el proceso está expuesto de forma didáctica y sencilla para así crear nuestros propios robots avanzados.

» HARDWARE / DESARROLLO
» 352 PÁGINAS
» ISBN 978-987-663-020-7



LLEGAMOS A TODO EL MUNDO VÍA  * Y  **

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

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

CONTENIDO

1 | C# Y .NET

Programación para Windows / Comprender .NET / Compilador de C# / El ambiente de desarrollo

2 | LOS ELEMENTOS BÁSICOS DE UN PROGRAMA

Lenguajes / Errores en los programas / Las variables / Comentarios / Precedencia de operadores / Pedido de datos

3 | EL PROGRAMA TOMA DECISIONES

La toma de decisiones / Expresiones y operadores relacionales / El uso de if y else / If anidados / Escalera de if-else / Expresiones lógicas / El uso de switch

4 | CREACIÓN DE CICLOS

El ciclo for / El valor de inicio / Límite de conteo / Control de incremento / El contador y el acumulador / Incrementos y decrementos / El ciclo while y do while

5 | FUNCIONES Y MÉTODOS

Las funciones / Tipos de funciones / Optimizar con funciones / Pasaje de parámetros por copia y paso por referencia

6 | LOS ARREGLOS

Declaración de arreglos / Asignación y uso de valores / Acceso a arreglos / Arreglos de tipo jagged

7 | LAS COLECCIONES

ArrayList / Declaración / Agregar información / El ciclo foreach / Stack / Queue / Hashtable

8 | LAS CADENAS

El método ToString() / StringBuilder / Justificación del texto / Intercambio entre mayúsculas y minúsculas

9 | ESTRUCTURAS Y ENUMERACIONES

Definición / Crear variables del nuevo tipo / Acceso a la estructura / Creación de un constructor

10 | CÓMO CREAR NUESTRAS PROPIAS CLASES

Programación orientada a objetos / Clases / Datos / Métodos / Protección de datos / Métodos públicos y privados

CAPÍTULO 11: FLUJOS Y ARCHIVOS

Los flujos / Streams en memoria / Uso de archivos

CAPÍTULO 12: DEPURACIÓN

Cómo depurar un programa / Errores de compilación / Errores en tiempo de ejecución / Manejo de errores / Errores de lógica / Detener la ejecución

NIVEL DE USUARIO

PRINCIPIANTE

INTERMEDIO

AVANZADO

EXPERTO

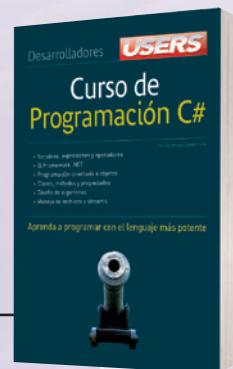
C# GUÍA TOTAL DEL PROGRAMADOR

Este libro es un completo curso de programación en C#, actualizado a la versión 4.0, incluida en Visual Studio 2010. Es ideal tanto para quienes desean migrar a este potente lenguaje, como para quienes quieran aprender a programar, incluso, sin tener conocimientos previos.

C# es el lenguaje más utilizado en la actualidad, por ser poderoso, flexible y robusto. En esta obra cubriremos desde las bases de la programación estructurada hasta los fundamentos de la orientada a objetos, sin dejar de aprender el manejo de estructuras complejas, como los archivos y las clases.

Nicolás Cosio aporta su vasta experiencia y la didáctica comprobada de sus obras previas, para facilitar al lector la tarea de comprender cómo se logra desarrollar código optimizado destinado a obtener aplicaciones poderosas y eficientes.

Una obra imprescindible para fundar bases sólidas en el desarrollo y aprender a programar de la mejor manera.



Versión completamente mejorada,
actualizada y ampliada de la obra
"Curso de programación C#"

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

C# 2010

This book is a full C# learning course. Aimed for those who want to migrate to this language, as well as for those who want to learn programming, even if they do not have previous knowledge on the subject.



MANUALES USERS MANUALES USERS MANUAL

EXPERIMENTE TODO EL PODER DE VISUAL STUDIO 2010

www.FreeLibros.me