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-

delo 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 desee 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 estás 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.

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.

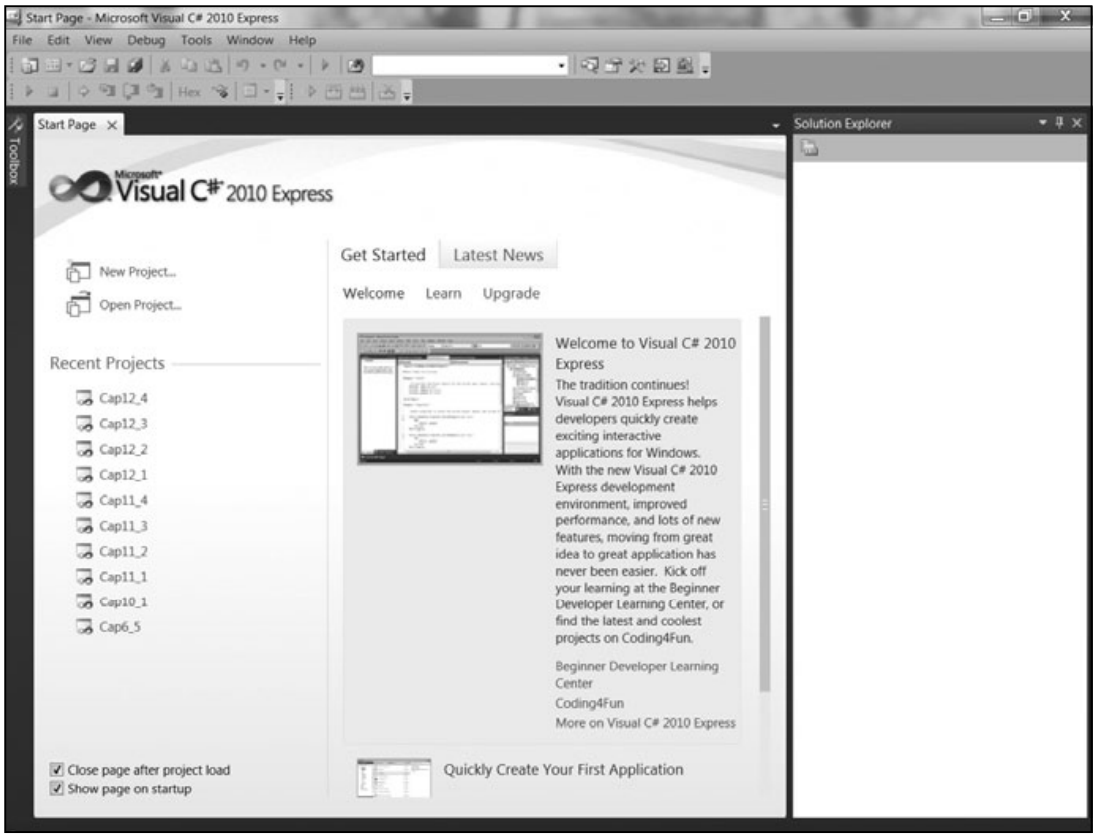

Figura 1. *Aquí vemos la ventana principal de* ***C# Express 2010****. Esta interfaz aparece cada vez que iniciamos el programa.*

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**.

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.

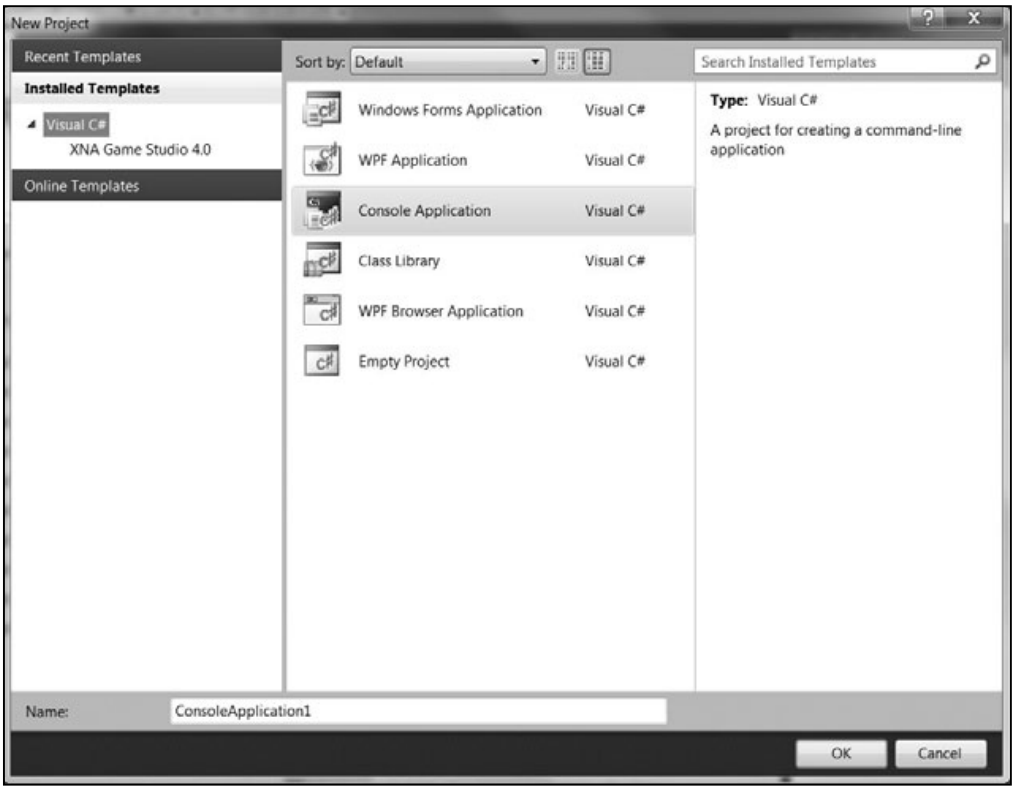

Figura 2. *Éste es el cuadro de dialogo que usamos para crear un nuevo proyecto, gracias a sus opciones podemos generar diferentes tipos de proyectos.*

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.

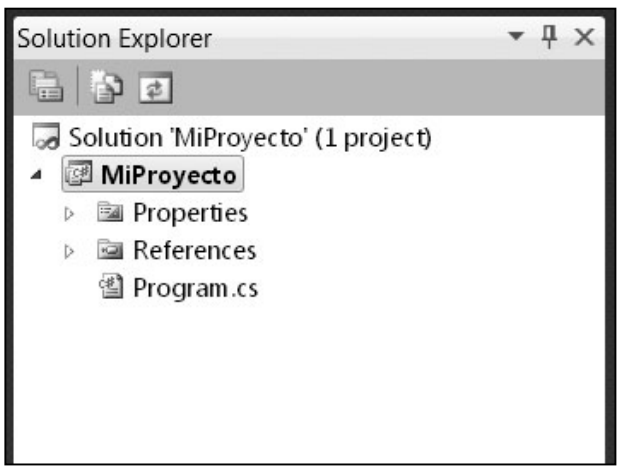

Figura 3. *Esta figura nos muestra el panel derecho denominado* ***Solution Explorer.***

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 namespace 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 **autocompletar** y nos permite trabajar rápidamente y reducir la cantidad de errores de sintaxis. El cuadro de autocompletar 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.

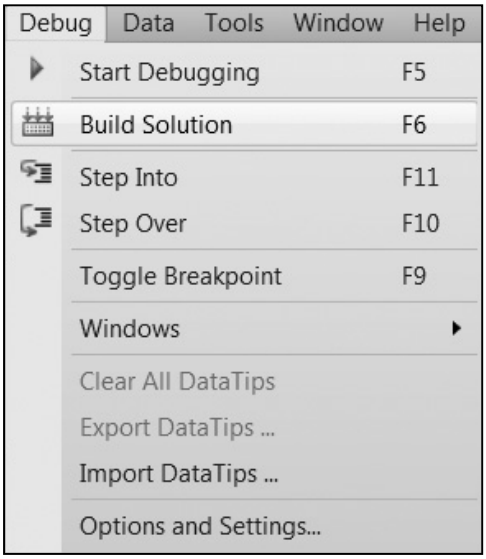

Figura 4. *Para compilar nuestra aplicación, primero debemos construirla, seleccionando la opción adecuada desde el menú.*

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

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.**

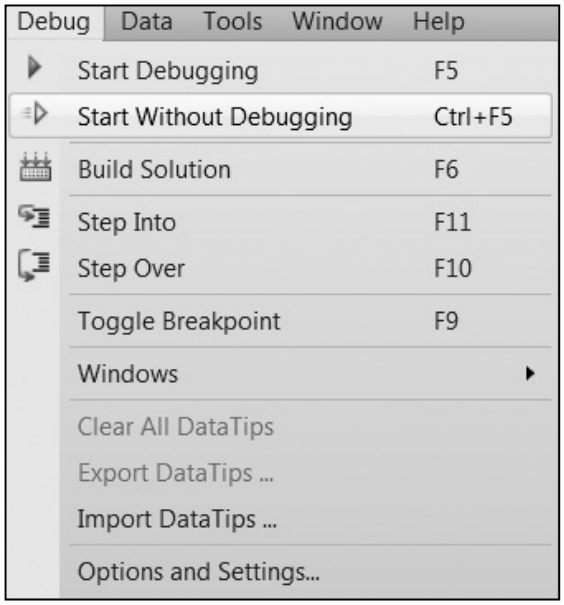

***Figura 5.** Con el menú Debug/Start Without Debbuging nuestro programa se ejecutará.*

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.

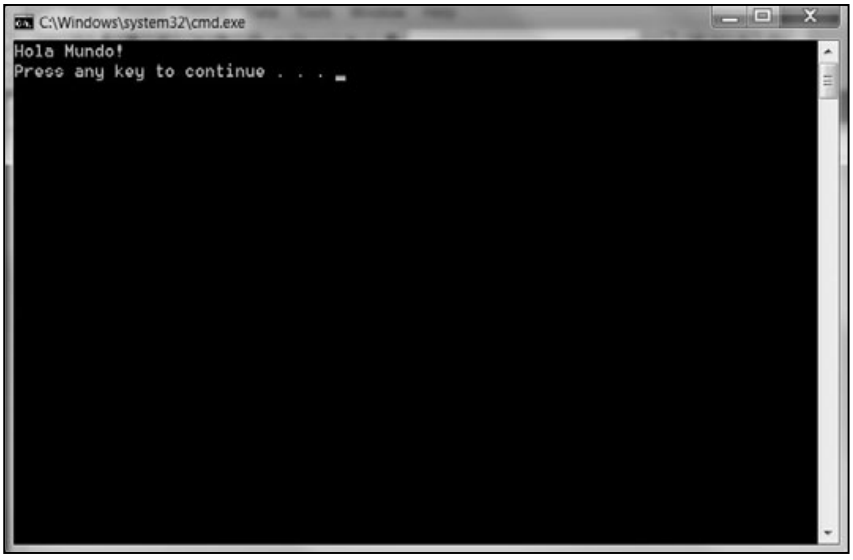

***Figura 6.** Ahora podemos ver nuestra aplicación ejecutándose en la consola.*

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.

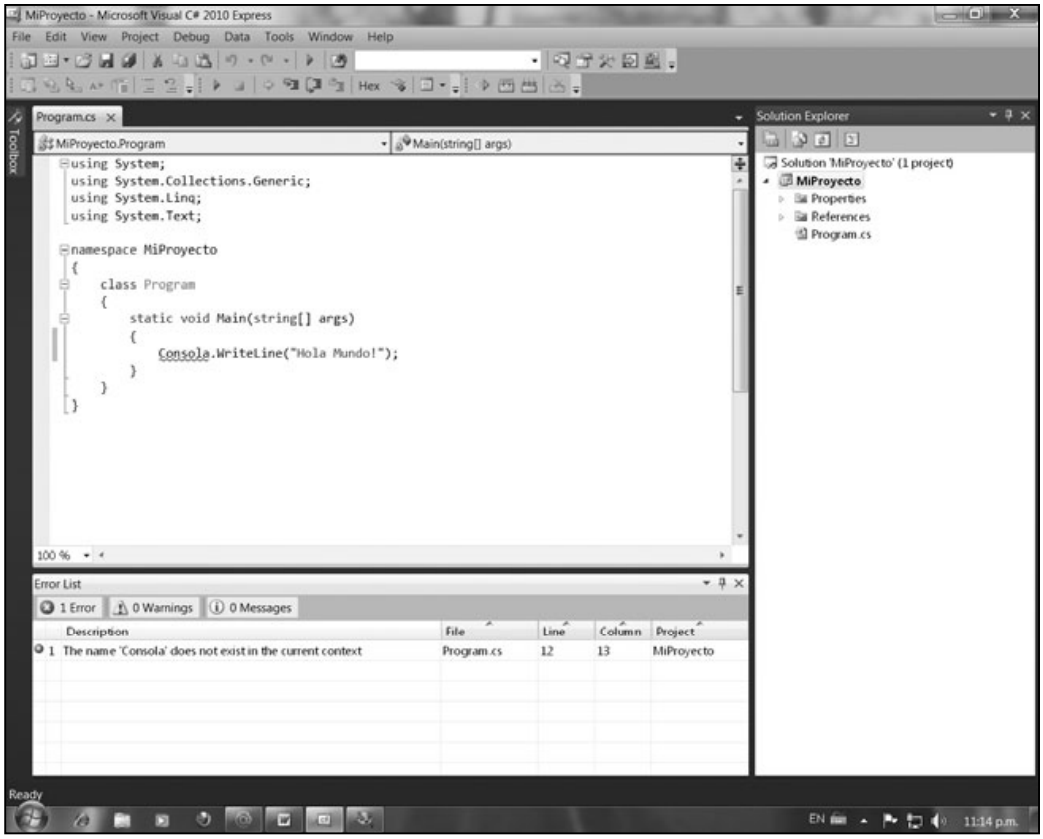

Figura 7. *En esta imagen vemos que el programa tiene un error y por lo tanto no se puede compilar.*

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.

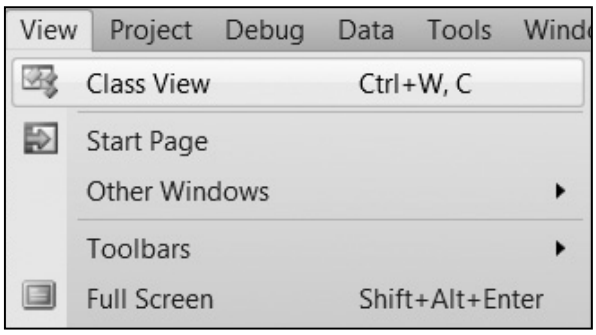

***Figura 8**. Para mostrar la vista de clases debemos de usar el menú **View**.*

La ventana de la vista de clases está divida 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.

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.

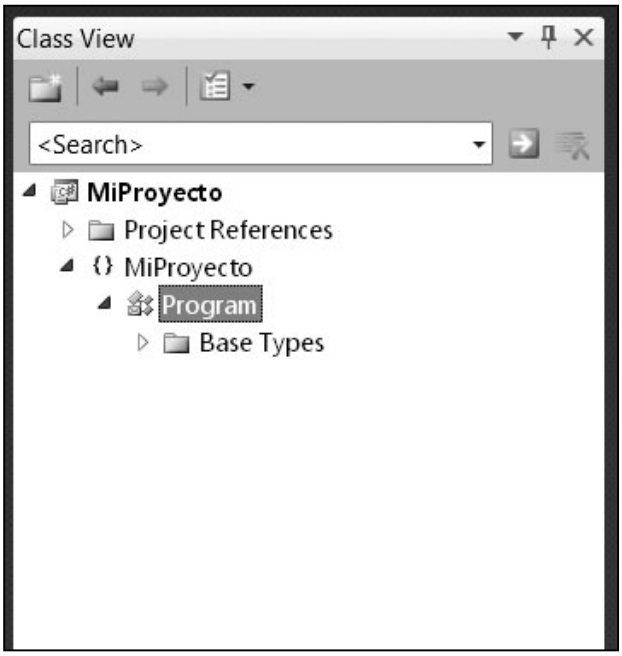

Figura 9. *Aquí podemos observar a la vista de clases en el lado derecho. El árbol está abierto para poder ver los elementos que lo componen.*

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**.

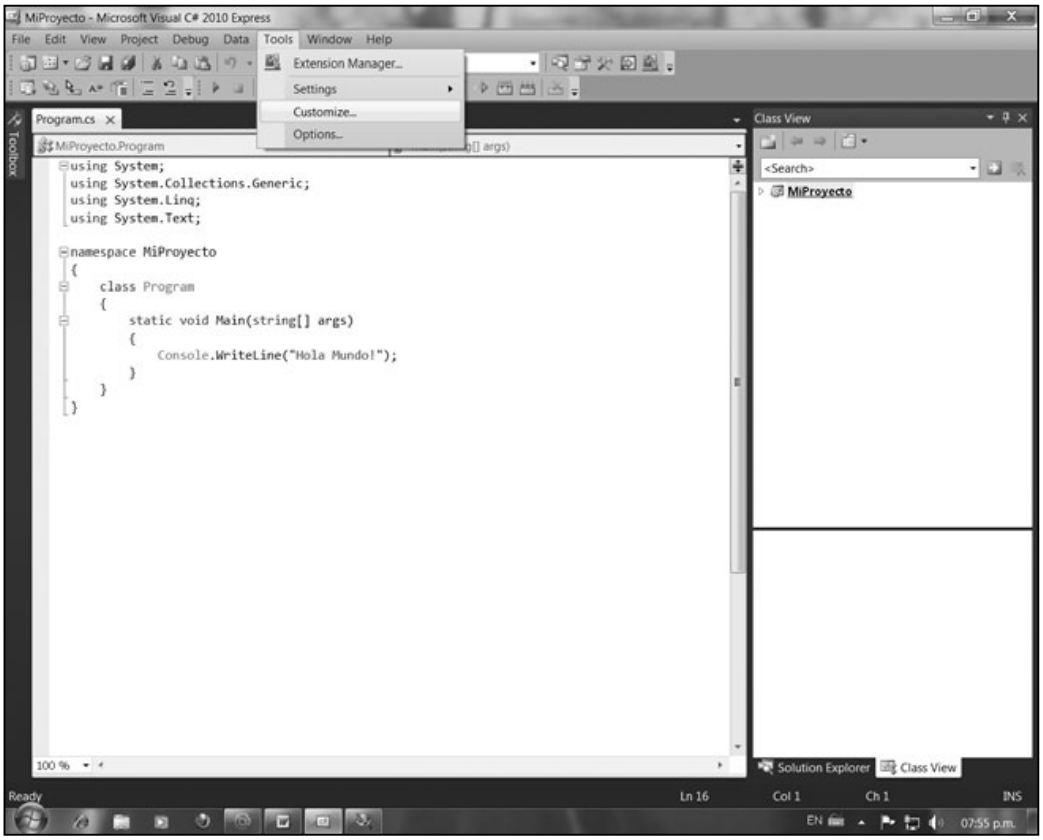

Figura 10. *Es necesario seleccionar la opción de* ***Customize...*** *en el menú* ***Tools.***

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.**

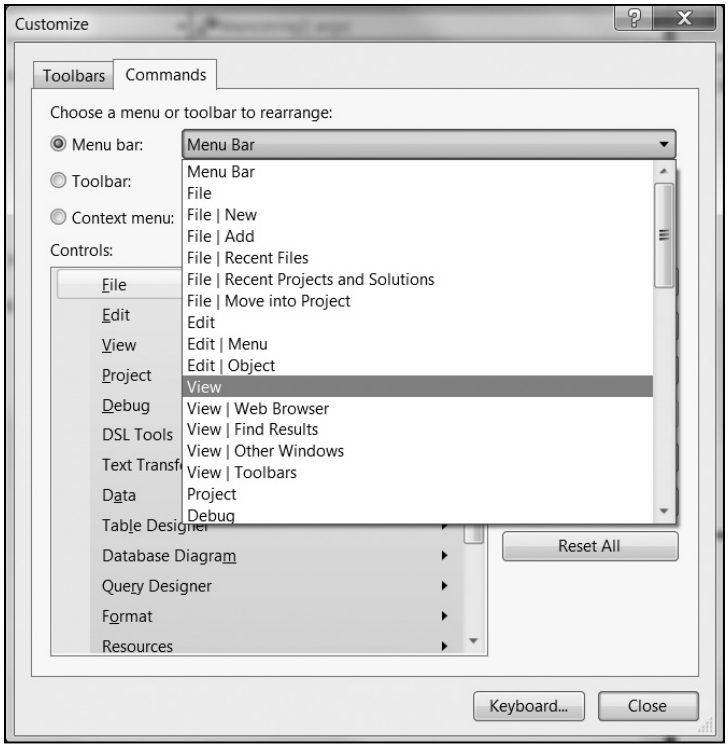

Figura 11. *En esta ventana debemos seleccionar el menú que deseamos modificar.*

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.**

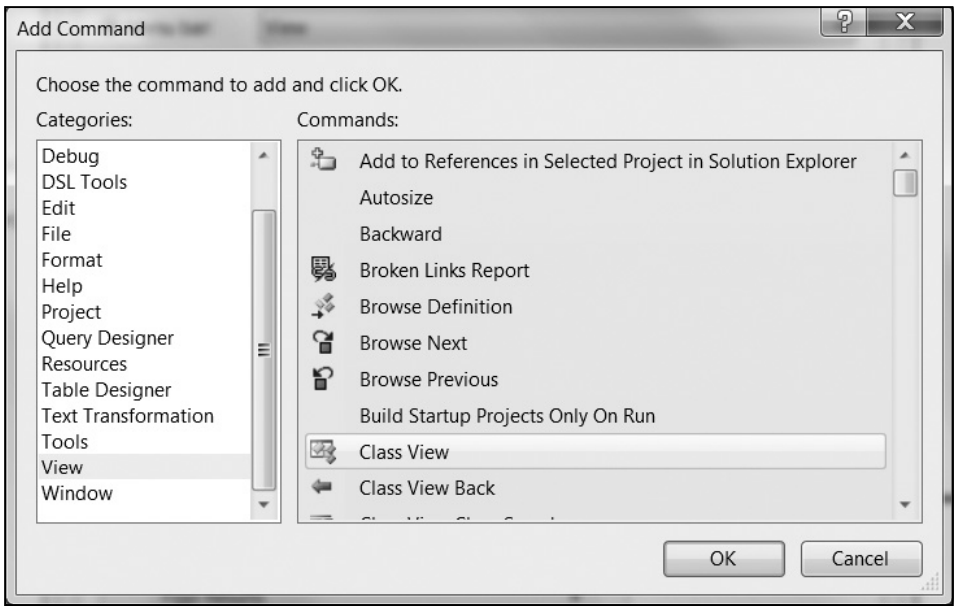

Figura 12. *En esta ventana seleccionamos el comando que deseamos insertar.*

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.