C# para programadores VB6

En este apéndice presentaremos una breve introducción al lenguaje C# dirigida específicamente a aquellos programadores cuya experiencia hasta el presente ha estado centrada en Visual Basic 6.

> *Tenga en cuenta que a lo largo de este apéndice cualquier mención a VB se refiere realmente a VB6. En las pocas ocasiones en que se mencione VB.NET, lo nombraremos explícitamente.*

C# y Visual Basic son lenguajes muy diferentes, tanto en su estilo sintáctico como en los conceptos fundamentales en los que están basados. Esto implica que los programadores de Visual Basic se encontrarán una curva de aprendizaje muy empinada hasta alcanzar la familiaridad con C#, incluso a nivel básico. Este apéndice tiene como objetivo hacer esa curva de aprendizaje más suave, proporcionando una introducción a C# orientada a aquellos que tienen un conocimiento de VB, y se centra en las diferencias conceptuales principales entre los dos lenguajes. Nuestro enfoque a lo largo de casi todo el apéndice se basará en comparar cómo se programaría la solución de un problema en VB y en C#, presentando en paralelo los códigos correspondientes en ambos lenguajes.

Esto significa que nuestro análisis del lenguaje C# se limitará a un nivel básico: no vamos a describir las características más avanzadas del lenguaje – tendrá que leer los primeros capítulos de este libro para eso. El énfasis estará en mostrarle las diferentes metodologías relacionadas con la escritura de código en el lenguaje C#.

Diferencias entre C# y VB

Más allá de las diferencias sintácticas obvias entre los lenguajes, hay dos conceptos fundamentales a los que tendrá que acostumbrarse para pasar de VB a C#:

3. El concepto del flujo completo de ejecución de un programa de principio a fin: Visual Basic oculta al programador este aspecto de los programas, de manera que la única parte de un programa VB que Ud. escribe cuando desarrolla una aplicación son los manejadores de eventos, algunos métodos, etc., en los módulos de clase. C# pone a nuestra disposición el programa completo en el código fuente. La razón para esto tiene que ver con el hecho de que C# puede verse, filosóficamente, como la siguiente generación de C++, y las raíces de C++ se remontan a los años 60. C++ es anterior a las interfaces de usuario basadas en ventanas y los sistemas operativos sofisticados. C++ evolucionó como un lenguaje de bajo nivel, próximo a la máquina y de propósito general. Para crear aplicaciones con interfaz de usuario gráfica en C++, era necesario realizar explícitamente las llamadas al sistema necesarias para crear e interactuar con las ventanas. C# se erige sobre esta herencia, simplificando y modernizando C++, con el objetivo de que puedan obtenerse, aproximadamente, los beneficios de rendimiento derivados del bajo nivel de C++, sin que la programación sea mucho más compleja de lo que es en VB. VB, por otro lado, es un lenguaje joven, diseñado específicamente para el desarrollo rápido de aplicaciones Windows con interfaz de usuario gráfica. Por esta razón, en VB todo el código que se encarga de la gestión de ventanas está oculto, y todo lo que el programador de VB implementa son los manejadores de eventos. En C#, todo ese código de manejo de ventanas se expone como parte del código fuente.

4. Las clases y la herencia en C# son mucho más orientadas a objetos que en VB, exigiendo que todo el código sea parte de una clase. El lenguaje también incluye un amplio soporte para la herencia de implementación. De hecho, la mayoría de los programas C# bien diseñados se apoyarán en esta forma de herencia, completamente ausente de VB.

La mayor parte de este apéndice estará dedicada a desarrollar dos ejemplos, para los cuales escribiremos versiones en VB y C#. El primer ejemplo consiste de un formulario simple, en el que se solicita un número y se visualiza la raíz cuadrada y el signo del número. Comparando detalladamente las versiones VB y C# del ejemplo, aprenderemos la sintaxis básica de C# y también entenderemos los conceptos que subyacen al flujo de ejecución de un programa.

A continuación, presentaremos un módulo de clase de VB, que almacena información sobre empleados, y su equivalente en C#. Con este ejemplo comenzaremos a ver la potencia real de C#, y al añadir características a los ejemplos descubriremos rápidamente que VB simplemente no soporta los conceptos que necesitamos para desarrollar el módulo de clase de acuerdo a los requisitos que hemos establecido, y por lo tanto tendremos que proceder exclusivamente en C#.

Terminaremos el apéndice con un corto recorrido por algunas de las restantes diferencias entre VB y C# no ilustradas por los ejemplos.

Antes de comenzar, tenemos que clarificar algunos conceptos: las clases, la compilación y las clases base .NET.

Clases

A lo largo del apéndice utilizaremos ampliamente las clases C#. Las clases C# representan objetos definidos con gran precisión, que hemos explicado en detalle en los Capítulos 4 y 5. Sin embargo, para nuestros propósitos en este apéndice, sería bueno pensar en ellas como los equivalentes C# de los módulos de clase VB, porque son entidades bastante similares: al igual que un módulo de clase VB, una clase C# implementa propiedades y métodos y contiene variables miembros. Al igual que en el caso de los módulos de clase VB, el programador puede crear objetos de una clase C# dada (instancias de la clase), utilizando al operador `new`. Sin embargo, detrás de estas similitudes hay muchas diferencias. Por ejemplo, un módulo de clase VB es

realmente una clase COM. Las clases C#, por el contrario, no son usualmente clases COM, sino que siempre se integran en la Plataforma .NET. Las clases C# también son más ligeras que sus colegas de VB/COM, en el sentido de que se han diseñado más pensando en el rendimiento y en afectar éste lo menos posible cuando son instanciadas. Sin embargo, estas diferencias no afectarán nuestra discusión del lenguaje C#.

Compilación

Ud. ya sabe que un ordenador nunca ejecuta directamente el código escrito en un lenguaje de alto nivel, ya sea VB, C++, C o cualquier otro lenguaje. En vez de eso, todo el código fuente se traduce a código ejecutable nativo, un proceso conocido como **compilación**. Durante la puesta a punto, VB ofrece la opción de ejecutar el código directamente (en cuyo caso cada línea de código se compilará cuando el ordenador vaya a ejecutar esa línea – en este caso decimos que el código se **interpreta**), o hacer una compilación completa (en cuyo caso primero se traducirá el programa completo a código ejecutable, y sólo después comenzará la ejecución). Realizar una compilación completa permite que cualquier error de sintaxis sea descubierto por el compilador antes de que el programa comience su ejecución. También permite obtener un mayor rendimiento al ejecutar, y es por lo tanto la única opción permitida en C#.

La compilación en C# se realiza en dos fases; inicialmente el código se traduce al llamado Lenguaje Intermedio (Intermediate Language - IL). Esta es la fase a la que nos referiremos informalmente como compilación. La segunda fase, la conversión a código ejecutable nativo, puede hacerse en tiempo de ejecución, pero es una fase mucho más simple que no conduce a problemas de rendimiento significativos. Este proceso es muy diferente de la mera interpretación del código. Fragmentos enteros de código se convierten cada vez de IL a código de máquina, y el código ejecutable nativo resultante se almacena, de modo que no necesite ser compilado nuevamente la próxima vez que se ejecute. Microsoft cree que esto, en combinación con otras optimizaciones, conducirá a un código que tendrá un rendimiento superior al que produce el sistema tradicional de compilar directamente el código fuente a código ejecutable nativo. Aunque la existencia del IL es algo que Ud. deberá tener presente, no afectará para nada nuestra discusión en este apéndice, ya que no afecta la sintaxis del lenguaje C#.

Las clases base .NET

VB no sólo consiste del lenguaje en sí. También incluye un gran número de funciones asociadas, como las funciones de conversión `CInt`, `CStr`, etc., las funciones del sistema de ficheros, las funciones de fecha y hora, y muchas más. VB también se apoya en la presencia de controles ActiveX para proporcionar los controles estándar que el programador coloca en sus formularios – cuadros de listas, botones, cuadros de texto, etc.

C# se apoya también en un extenso soporte en estas áreas de la programación – pero en el caso de C# el soporte proviene de un conjunto de clases muy grande, conocido como las **clases base .NET**. Estas clases proporcionan soporte para casi cualquier aspecto del desarrollo para Windows. Hay clases que representan a cada uno de los controles estándar, clases que realizan conversiones, clases que permiten manipular fechas y horas y acceder al sistema de ficheros, clases para acceder a Internet, y muchas más. Aquí no describiremos en detalle la librería de clases .NET, pero sí nos referiremos frecuentemente a ella. De hecho, C# está tan interrelacionado con las clases base .NET que veremos que varias palabras reservadas de C# son meros envoltorios que encapsulan a ciertas clases base específicas. En particular, todos los tipos de datos básicos de C# que se utilizan para representar enteros, números de punto flotante, cadenas, etc., son realmente clases base.

Una diferencia importante entre VB6 y C# en este sentido, es que las funciones de sistema de VB son específicas de VB, mientras que las clases base .NET pueden ser utilizadas desde cualquier lenguaje de la Plataforma. NET.

Convenios

En este apéndice compararemos frecuentemente código en C# y Visual Basic. Para que sea más fácil de identificar el código en los dos lenguajes, presentaremos el código C# en el mismo formato que se utiliza a lo largo del libro:

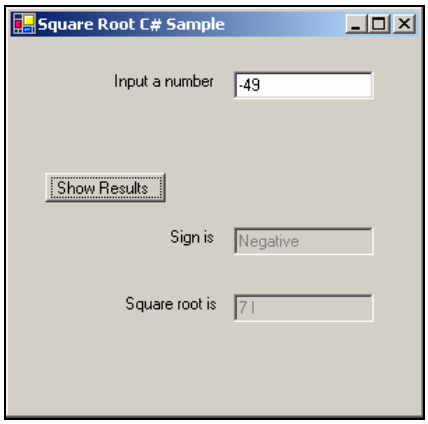
```
// código C# que ya hemos visto
// código C# que queremos destacar o que es nuevo
```

Sin embargo, todo el código de VB se presentará en este formato:

```
' el código VB se presenta sobre fondo blanco
```

Ejemplo: el formulario de la raíz cuadrada

En esta sección vamos a examinar una aplicación sencilla llamada `SquareRoot`, que hemos desarrollado en ambos lenguajes, Visual Basic y C#. La aplicación consiste de un cuadro de diálogo que solicita un número al usuario y, cuando el usuario pulsa un botón, visualiza el signo y la raíz cuadrada del número. Si el número es negativo, la raíz cuadrada debe ser mostrada como un número complejo – lo que implica calcular la raíz cuadrada del número con signo cambiado y añadir '`i`' después del número. La versión C# del ejemplo tiene la apariencia que se muestra a continuación. La versión VB es muy parecida, sólo que tiene un icono estándar de VB, en lugar del icono de formulario de la ventana .NET, en la parte superior izquierda:



La versión VB de SquareRoot

Para hacer que esta aplicación funcione en Visual Basic, tendremos que programar un manejador de eventos para el evento de pulsación sobre el botón. Hemos dado al botón el nombre `cmdShowResults`, y los cuadros de texto tienen los nombres intuitivos `txtNumber`, `txtSign` y `txtResult`. Con estos nombres, el manejador de eventos sería el siguiente:

```
Option Explicit
Private Sub cmdShowResults_Click()
   Dim NumberInput As Single
```

```
   NumberInput = CSng(Me.txtNumber.Text)
   If (NumberInput < 0) Then
      Me.txtSign.Text = "Negative"
      Me.txtResult.Text = CStr(Sqr(-NumberInput)) & " i"
   ElseIf (NumberInput = 0) Then
      txtSign.Text = "Zero"
      txtResult.Text = "0"
   Else
      Me.txtSign.Text = "Positive"
      Me.txtResult.Text = CStr(Sqr(NumberInput))
   End If
End Sub
```

Esa es la única parte del código VB que tendremos que escribir.

La versión C# de SquareRoot

En C# también tendremos que escribir un manejador de eventos para el evento de pulsación del botón. Hemos conservado los mismos nombres para el botón y los cuadros de texto, y el código C# es el siguiente:

```
// Gestor de eventos para la pulsación del botón 'Show Results'.
// Muestra la raíz cuadrada y el signo del número

private void OnClickShowResults(object sender, System.EventArgs e)
{
   float NumberInput = float.Parse(this.txtNumber.Text);
   if (NumberInput < 0)
   {
      this.txtSign.Text = "Negative";
      this.txtResult.Text = Math.Sqrt(-NumberInput).ToString() + " i";
   }
   else if (NumberInput == 0)
   {
      txtSign.Text = "Zero";
      txtResult.Text = "0";
   }
   else
   {
      this.txtSign.Text = "Positive";
      this.txtResult.Text = Math.Sqrt(NumberInput).ToString ();
   }
}
```

Comparando estos dos ejemplos de código, seguramente podrá ver la similitud en la estructura del código, e incluso sin ningún conocimiento de C#, probablemente tendrá una idea de qué está pasando. También es evidente que hay un gran número de diferencias en la sintaxis de los dos lenguajes. En las páginas siguientes vamos a comparar estos dos ejemplos, para ver qué podemos aprender sobre la sintaxis de C# en el proceso. También desvelaremos algunas de las diferencias entre las metodologías básicas de C# y VB.

Sintaxis básica

En esta sección examinaremos los dos programas `SquareRoot`, para ver qué nos enseñan sobre la sintaxis de C#.

C# exige que todas las variables sean declaradas

Si comenzamos con la primera línea del código VB, nos encontramos la instrucción `Option Explicit`. Esta instrucción no tiene contrapartida en C#. La razón es que en C# siempre deben declararse explícitamente las variables antes de utilizarlas. Es como si C# siempre funcionara con `Option Explicit` activado y no permitiera desactivarlo. Por lo tanto, no hay necesidad de declarar `Option Explicit` explícitamente.

El sentido de esta restricción es que C# ha sido diseñado muy cuidadosamente para que resulte difícil introducir accidentalmente errores en el código. Generalmente, en VB se aconseja utilizar `Option Explicit` porque evita errores difíciles de encontrar, motivados por nombres de variables mal escritos. Verá que C# generalmente no permite hacer cosas que tienen un alto riesgo de provocar errores.

Comentarios

Debido a que poner comentarios en el código es siempre importante, lo próximo que hacemos en ambos ejemplos (¡o lo primero en el ejemplo C#!) es añadir un comentario:

```
// Gestor de eventos para la pulsación del botón 'Show Results'.
// Muestra la raíz cuadrada y el signo del número

private void OnClickShowResults(object sender, System.EventArgs e)
{
```

En VB se utiliza un apóstrofo para denotar el inicio de un comentario, y el comentario se extiende hasta el final de la línea. Los comentarios en C# funcionan del mismo modo, excepto que comienzan con dos barras: `//`. Al igual que en el caso de los comentarios de VB, podemos usar una línea completa para un comentario, o añadir un comentario al final de una línea normal de código:

```
    // Esta línea produce el resultado

    int Result = 10*Input;   // calcular el resultado
```

Sin embargo, C# es más flexible en cuanto a comentarios, porque permite otras dos maneras de insertar comentarios en el código, cada una de las cuales tiene un efecto ligeramente diferente.

Un comentario también puede estar delimitado por las secuencias `/*` y `*/`. En otras palabras, si el compilador encuentra una secuencia `/*`, asume que todo el texto siguiente es un comentario, hasta que se encuentre `*/`. Esto permite insertar en el código comentarios que se extienden por varias líneas:

```
/* este es un comentario
muy
muy
muy
largo */
```

Los comentarios cortos dentro de una línea son muy útiles en caso de que sólo se desee intercambiar algo temporalmente en una línea de código durante la puesta a punto:

```
X = /*20*/  15;
```

La tercera forma es muy similar a la primera. Sin embargo, en ella se utilizan tres barras:

```
/// <summary>
/// Gestor de eventos para la pulsación del botón 'Show Results'.
/// Muestra la raíz cuadrada y el signo del número
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>

private void OnClickShowResults(object sender, System.EventArgs e)
```

Si utiliza tres barras en lugar de dos, el comentario se extiende hasta el final de la línea. Sin embargo, este comentario tiene un efecto adicional: el compilador de C# es capaz de utilizar los comentarios que empiezan con tres barras para generar automáticamente documentación para el código fuente, en forma de un fichero XML independiente. Por eso es que el ejemplo anterior parece tener una estructura bastante formal para el texto del comentario: la estructura está lista para ser colocada en un fichero XML. No entraremos en los detalles de este proceso (se analiza en el Capítulo 3). No obstante, sí diremos que esta característica permite que comentando cada método de su código Ud. pueda obtener automáticamente la documentación completa y actualizada de la clase cuando modifique su código. El compilador incluso verificará que la documentación se corresponda con las signaturas de los métodos, etc.

Separación y agrupación de instrucciones

La diferencia más visible para Ud. entre los códigos C# y VB anteriores probablemente será la presencia de los puntos y comas y las llaves en el código C#. Aunque esto puede hacer que el código C# parezca confuso, el principio es realmente muy simple. Visual Basic utiliza los cambios de línea para indicar el final de las instrucciones, mientras que C# utiliza los puntos y comas para el mismo propósito. De hecho, el compilador de C# ignora completamente todos los blancos en el código – incluso los retornos de carro. Estas características de la sintaxis de C# pueden combinarse para tener una gran libertad a la hora de colocar el código. Por ejemplo, lo siguiente (parte del ejemplo anterior, reformateado) también es código C# absolutamente válido:

```
        this.txtSign.Text =

"Negative"; this.txtResult.Text = Math.Sqrt
(-NumberInput) + " i";
```

Obviamente, si desea que otras personas puedan leer su código, optará para el primer estilo, y Visual Studio .NET colocará automáticamente su código utilizando ese estilo.

Las llaves se utilizan para agrupar instrucciones en lo que se conoce como **bloques de instrucciones** (o también como **instrucciones compuestas**). Éste es un concepto que realmente no existe en VB. Se puede agrupar cualquier conjunto de instrucciones encerrándolas entre llaves. Tal grupo tendrá la consideración de una única instrucción de bloque, y podrá utilizarse en cualquier parte del código C# donde pueda colocarse una instrucción.

Los bloques de instrucciones se utilizan mucho en C#. Por ejemplo, en el código anterior no hay ninguna indicación explícita del fin del método (C# utiliza **métodos** donde VB utiliza funciones y subrutinas). VB necesita una instrucción `End Sub` al final de cualquier subrutina, porque éstas pueden contener tantas instrucciones como sea necesario – de manera que un marcador específico es la única manera que tiene VB de saber dónde termina la subrutina. C# trabaja de forma diferente. En C#, un método está formado exactamente por una instrucción compuesta. Debido a esto, el método acaba con la llave cerrada, que se corresponde con la llave abierta que marca el inicio del método.

Esto es algo que encontrará muy frecuentemente en C#: donde Visual Basic utiliza una palabra reservada para marcar el final de un bloque de código, C# simplemente organiza el bloque en una instrucción compuesta. La

instrucción `if` en los ejemplos anteriores ilustra ese mismo aspecto. En VB, es necesaria una instrucción `EndIf` para marcar dónde termina un bloque `If`. En C#, la regla es que una cláusula `if` siempre contiene exactamente **una** instrucción, y la cláusula `else` también contiene una sola instrucción. Si queremos poner más de una instrucción en cualquiera de esas cláusulas, como es el caso en el ejemplo anterior, utilizamos una instrucción compuesta.

Mayúsculas y minúsculas

Otro punto que puede observar en relación con la sintaxis es que todas las palabras reservadas – `if`, `else`, `int`, etc. en el código C# están en minúsculas. A diferencia de VB, C# diferencia entre mayúsculas y minúsculas. Si escribe `If` en lugar de `if`, el compilador no entenderá su código. Una ventaja de distinguir entre mayúsculas y minúsculas es que puede tener dos variables cuyos nombres sólo se diferencien en eso, como `Name` y `name`. Encontraremos esto más adelante, en el segundo ejemplo del apéndice.

> *Si está acostumbrado a no diferenciar entre mayúsculas y minúsculas, puede encontrar bastante extraña la idea de tener nombres de variables que sólo se diferencien en el uso de las mayúsculas y minúsculas. Pero una vez que se haya acostumbrado, encontrará una libertad adicional al nombrar sus variables, que en ocasiones puede resultar muy útil.*

En general, encontrará que todas las palabras reservadas de C# se escriben completamente en minúsculas.

Métodos

Comparemos la sintaxis que utilizan VB y C# para declarar la parte del código que manejará el evento. En VB luce así:

```
Private Sub cmdShowResults_Click()
```

y en C#:

```
private void OnClickShowResults(object sender, System.EventArgs e)
```

Lo primero que debemos destacar es que la versión de VB declara una subrutina (`Sub`), mientras que la versión C# declara un método. En Visual Basic, el código se agrupa tradicionalmente en subrutinas y funciones, con el concepto de que ambos tipos de rutinas constituyen un procedimiento. Adicionalmente, los objetos de clase de VB tienen lo que se conoce como métodos, que para todos los propósitos prácticos significan lo mismo que los procedimientos, sólo que son parte de un módulo de clase.

A diferencia de esto, C# sólo tiene métodos (esto tiene relación con el hecho de que, como veremos después, en C# todo parte de una clase). En C# no hay ningún concepto separado de funciones y subrutinas – esos términos ni siquiera existen en la especificación del lenguaje C#. En VB, la única diferencia real entre una subrutina y una función es que una subrutina nunca devuelve un valor. En C#, si un método no necesita devolver un valor, se declara como que devuelve `void` (como ilustra el método `OnClickShowResults()`).

La sintaxis para declarar un método es similar en los dos lenguajes, al menos en el hecho de que los parámetros siguen al nombre del método entre paréntesis. Sin embargo, observe que en VB indicamos que estamos declarando una subrutina mediante la palabra `Sub`, mientras que en la versión C# no hay ninguna palabra correspondiente. En C#, el tipo de retorno (`void` en este caso), seguido por el nombre del método, seguido por el paréntesis abierto es suficiente para decirle al compilador que estamos declarando un método, debido a que ninguna otra construcción de C# tiene esta sintaxis (en los arrays de C# se utilizan corchetes en lugar de paréntesis, de manera que no hay ningún riesgo de confusión con los arrays).

Al igual que en el `Sub` de VB, la declaración del método C# anterior está precedida por la palabra reservada `private`. Esto tiene aproximadamente el mismo significado que en VB – evita que el código externo pueda ver el método. Más tarde veremos qué significa exactamente 'código externo'.

Hay otras dos diferencias a destacar en cuanto a la declaración de métodos: la versión C# recibe dos parámetros, y tiene un nombre diferente del manejador de eventos VB.

Hablaremos primero sobre el nombre. El nombre del manejador de eventos en VB es proporcionado por el entorno de desarrollo. La razón por la cual VB sabe que esa subrutina es el manejador de eventos que se disparará cuando se haga clic en el botón, es el nombre, `cmdShowResults_Click`. Si renombra la subrutina, ésta no será llamada cuando se pulse el botón. Sin embargo, C# no utiliza el nombre de esta manera. En C#, como veremos pronto, hay otro código que le dice al compilador qué método es el manejador de eventos para este evento. Eso significa que podemos darle el nombre que queramos al manejador. No obstante, es tradicional utilizar un nombre que comience con `On` para los manejadores de eventos, y en C# la práctica recomendada es nombrar los métodos (y la mayoría de los demás elementos del código) utilizando el convenio de nombres **Pascal**, que significa que las palabras se unen poniendo sus primeras letras en mayúsculas, con las demás letras en minúsculas. Utilizar el subrayado en los nombres NO se recomienda en C#. En el ejemplo, hemos elegido un nombre de acuerdo con las pautas anteriores: `OnClickShowResults()`.

En cuanto a los parámetros, no nos preocuparemos por el significado de éstos en este apéndice. Tan sólo diremos que todos los manejadores de eventos en C# tienen que recibir dos parámetros similares a éstos, y que estos parámetros proporcionan información adicional útil sobre el evento que se ha producido (por ejemplo, para un evento de movimiento de ratón, podrían indicar la situación del cursor del ratón).

Variables

El ejemplo `SquareRoot` puede decirnos mucho sobre las diferencias entre las declaraciones de variables en C# y VB. En la versión de VB declaramos un número de punto flotante y le asignamos su valor de la siguiente forma:

```
Dim NumberInput As Single
NumberInput = CSng(Me.txtNumber.Text)
```

La versión C# tiene esta apariencia:

```
float NumberInput = float.Parse(this.txtNumber.Text);
```

Como era de esperar, los tipos de datos de C# no son exactamente los mismos que en VB. `float` es el equivalente C# de `Single`. Probablemente es más fácil entender qué ocurre en este código si dividimos la versión C# en dos líneas equivalentes:

```
float NumberInput;
NumberInput = float.Parse(this.txtNumber.Text);
```

Ahora podemos comparar por separado la declaración e inicialización de variables.

Declaraciones

La diferencia sintáctica obvia entre C# y VB en lo que respecta a las declaraciones de variables es que en C# el tipo de datos precede, en lugar de seguir, al nombre de la variable, sin ninguna otra palabra reservada por medio. Esto da a las declaraciones de C# un formato más compacto que el de sus símiles en VB.

Observará que esta idea de una declaración que consiste sólo de un tipo seguida por un nombre se utiliza

también en otros lugares. Mire de nuevo la declaración de método en C# :

```
private void OnClickShowResults(object sender, System.EventArgs e)
```

El tipo (`void`) precede al nombre del método, sin otra palabra reservada para indicar lo que estamos declarando – eso es obvio del contexto. Lo mismo es cierto también para los parámetros. Los tipos de los parámetros son `object` y `System.EventArgs`. A propósito, el tipo `object` en C# juega un papel similar a `Object` en VB – indica algo para lo que cual deseamos no especificar concretamente el tipo. Sin embargo, el `object` de C# es mucho más poderoso que el `Object` de VB. En C#, `object` también reemplaza al tipo de datos `Variant` de VB. Estudiaremos `object` más tarde, aunque no nos ocuparemos de `System.EventArgs` en este apéndice. Es una clase base .NET, y no tiene equivalente en VB.

En el caso de las variables, la sintaxis de declaración utilizada en C# le permite combinar la declaración con la asignación de un valor inicial para la variable. En el ejemplo, `NumberInput` se inicializa a una expresión aparentemente complicada, que examinaremos pronto. Pero para tomar dos ejemplos más sencillos:

```
int X = 10;      // int es similar a Long en VB
string Message = "Hello World";   // string es similar a String en VB
```

Antes de abandonar el tema de las declaraciones, debemos mencionar otro par de cosas con relación a las variables.

No hay sufijos en C#

VB le permite añadir sufijos a las variables para indicar sus tipos de datos, y se utilizan `$` para `String`, `%` para `Int`, y `&` para `Long`:

```
Dim Message$   ' será una cadena
```

Esta sintaxis no está soportada en C#. Los nombres de las variables pueden contener sólo letras, dígitos y el carácter de subrayado, y siempre se debe indicar explícitamente el tipo de los datos.

No hay valores predefinidos para las variables locales

En el ejemplo de código VB, a la variable `NumberInput` se le asigna el valor predefinido 0 cuando es declarada. Esto realmente es una pérdida de tiempo de procesador, ya que en la siguiente instrucción se le asigna un nuevo valor. C# tiene más en cuenta el rendimiento y no se molesta en asignarle un valor por defecto a las variables locales cuando se declaran. No obstante, exige que inicialice tales variables antes de utilizarlas. El compilador de C# generará un error de compilación si intenta leer el valor de una variable local antes de asignarlo.

Asignación de valores a variables

La asignación de valores a las variables en C# se hace utilizando la misma sintaxis que en VB. Simplemente se pone un signo = después del nombre de la variable, seguido por el valor que se desea asignar. Un aspecto a destacar es que ésta es la única sintaxis que se utiliza en C# . En VB, en algunos casos utilizamos `Let`, mientras para que los objetos siempre se utiliza la palabra reservada `Set`:

```
Set MyListBox = new ListBox
```

C# no utiliza una sintaxis separada por hacer asignaciones a referencias a objetos. El equivalente C# de lo anterior es:

```
MyListBox = new ListBox();
```

Recuerde que en C# siempre se asignan las variables utilizando la sintaxis
<NombreVariable> = <Expresión>;

Clases

Ahora veremos qué ocurre en la expresión que se utiliza para inicializar la variable `NumberInput` en el ejemplo `SquareRoot`. Los ejemplos de C# y VB hacen exactamente lo mismo: toman el texto del cuadro de texto `txtNumber` y lo convierten a número. Pero la sintaxis para esto es diferente en los dos lenguajes:

```
NumberInput = CSng(Me.txtNumber.Text)
```

y:

```
float NumberInput = float.Parse(this.txtNumber.Text);
```

Extraer el valor de los cuadros de texto es bastante similar en ambos casos. La única diferencia para esa parte del proceso es únicamente sintáctica: VB utiliza la palabra reservada `Me`, mientras que C# utiliza la palabra reservada `this`, que tiene exactamente el mismo significado (de hecho, si lo desea, en el ejemplo C# se puede omitir `this`, del mismo modo que se puede omitir `Me` en la versión VB). En C#, podíamos escribir igualmente:

```
float NumberInput = float.Parse(txtNumber.Text)
```

Lo más interesante aquí es la forma en que la cadena recuperada del cuadro de texto se convierte a un `float`, porque esto ilustra un punto fundamental del lenguaje C#, que antes hemos indicado brevemente:

En C# todo es parte de una clase.

En VB, la conversión es llevada a cabo por una función, `CSng`. Sin embargo, C# no tiene funciones de la forma en que las tiene VB. C# es totalmente orientado a objetos y sólo le permitirá declarar métodos que son parte de una clase.

En C# la conversión de la cadena a número de punto flotante es realizada por el método `Parse()`. Sin embargo, como `Parse()` es parte de una clase, tiene que ser precedido por el nombre de la clase. La clase a la que pertenece el método `Parse()` que debemos llamar es `float`. Sí, eso es correcto. Hasta ahora hemos tratado a `float` como el equivalente C# del `Single` de VB. Sin embargo, también es una clase. En C#, **todos** los tipos de datos también son clases, lo cual significa que tipos como `int`, `float` o `string` tienen métodos y propiedades que se pueden llamar (aunque debemos señalar que `int` y `float` son tipos de clases especiales, conocidos en C# como estructuras. La diferencia no es importante para este ejemplo, pero la explicaremos más tarde).

Si estudia cuidadosamente el código anterior, podría notar un problema aparente en cuanto a la analogía con los módulos de clases VB. En VB, los métodos se llaman especificando el

nombre de una variable y no el nombre del módulo de clase, pero hemos llamado a `Parse()` especificando el nombre de la clase, `float`, en lugar del nombre de una variable. `Parse()` es realmente un tipo especial de método, conocido como método estático. No hay ningún equivalente en VB, y un método `static` puede ser llamado sin necesidad de crear una instancia de la clase. En ese caso, se especifica el nombre de la clase, `float`, en lugar del nombre de una variable. A propósito, `static` no tiene el mismo significado en C# que en VB. No hay equivalente en C# para las variables estáticas de VB – no hay necesidad de ellas en la metodología orientada a objetos de C#, porque los campos de C# se pueden utilizar para este propósito.

También, para ser exactos, debemos señalar que el nombre de la clase realmente es `System.Single`, y no `float`. `Single` es una de las clases base .NET, y C# utiliza la palabra reservada `float` para referirse a esta clase.

Las instrucciones if

Ahora llegamos a la parte principal del manejador de eventos: la instrucción `if`. Recuerde que la versión VB se parece a esto:

```
If (NumberInput < 0) Then
    Me.txtSign.Text = "Negative"
    Me.txtResult.Text = CStr(Sqr(-NumberInput)) & " i"
ElseIf (NumberInput = 0) Then
    txtSign.Text = "Zero"
    txtResult.Text = "0"
Else
    Me.txtSign.Text = "Positive"
    Me.txtResult.Text = CStr(Sqr(NumberInput))
End If
```

mientras que ésta es la versión C#:

```
if (NumberInput < 0)
{
   this.txtSign.Text = "Negative";
   this.txtResult.Text = Math.Sqrt(-NumberInput).ToString() + " i";
}
else if (NumberInput == 0)
{
   txtSign.Text = "Zero";
   txtResult.Text = "0";
}
else
{
   this.txtSign.Text = "Positive";
   this.txtResult.Text = Math.Sqrt(NumberInput).ToString();
}
```

De hecho, ya hemos explicado la diferencia sintáctica más grande: que cada parte de la instrucción `if` en C# debe ser una sola instrucción, y por lo tanto, si tenemos que ejecutar condicionalmente más de una instrucción debemos combinarlas en una sola instrucción compuesta. Si sólo hay una instrucción que ejecutar en cualquier rama condicional, no tenemos que formar una instrucción de bloque. Por ejemplo, si se pudiera omitir la asignación del valor inicial al texto del cuadro de texto `txtSign` en el código anterior, podríamos escribir:

```
if (NumberInput < 0)
```

```
         this.txtResult.Text = Math.Sqrt(-NumberInput) + " i";
      else if (NumberInput == 0)
         txtSign.Text = "Zero";
      else
         this.txtResult.Text = Math.Sqrt(NumberInput).ToString();
```

Hay algunas otras diferencias en la sintaxis que debemos comentar. En C#, los paréntesis alrededor de la condición que debe evaluarse en una instrucción `if` son obligatorios. En VB podríamos escribir:

```
If NumberInput < 0 Then
```

Si probamos esto en C#, se producirá un error de compilación inmediatamente. En general, C# es mucho más preciso en cuanto a la sintaxis que VB. Observe también que cuando evaluamos si `NumberInput` es cero, utilizamos dos signos de igual seguidos para la comparación:

```
      else if (NumberInput == 0)
```

En VB, el signo = sirve para los dos propósitos: se utiliza tanto para asignar valores a las variables como para comparar valores. C# los reconoce formalmente como dos tipos de operaciones muy diferentes y utiliza símbolos diferentes: = para la asignación y == para la comparación.

Hay otra diferencia importante de la que debe estar consciente, porque puede sorprenderlo fácilmente al hacer la transición de VB a C#:

> **`else if` son dos palabras separadas en C#, mientras que en VB es una sola: `ElseIf`.**

Cálculo de la raíz cuadrada: otro método de clase

Dados nuestros comentarios anteriores acerca de que en C# todo es miembro de una clase, no se sorprenderá al saber que el equivalente C# de la función `Sqr` de VB, que calcula raíces cuadradas, también es un método que es miembro de una clase. En este caso es el método `Sqrt()`, que es un miembro estático de otra clase base .NET, `System.Math`, que en nuestro código podemos abreviar como `Math`.

También habrá notado en el ejemplo que al comprobar la condición de que el número introducido sea igual a cero, no hemos especificado la palabra reservada `this` en el código C#:

```
         txtSign.Text = "Zero";
         txtResult.Text = "0";
```

y en el código VB correspondiente tampoco hemos especificado explícitamente `Me`. En C#, al igual que en VB, no es necesario utilizar explícitamente `this` a menos que, por alguna razón, el contexto lo requiera. Hemos elegido no hacerlo sólo para ilustrar este aspecto.

Cadenas

Al visualizar la raíz cuadrada de un número negativo, encontramos un primer ejemplo de tratamiento de cadenas de caracteres:

```
        this.txtResult.Text = Math.Sqrt(-NumberInput).ToString() + " i";
```

Notará en este código que en C# la concatenación de cadenas se realiza utilizando el símbolo + en lugar de &. También verá que la conversión de `float` a `String` se realiza llamando a otro método del objeto `float`. El método se llama `ToString()` y no es estático, por lo que debe ser llamado utilizando la misma sintaxis que en VB cuando se llama a métodos de objetos: prefijando el nombre del método con el nombre de la variable que representa al objeto, seguido por un punto. Algo que se debe recordar en C# es que todas las clases (y por lo tanto todos los objetos) implementan el método `ToString()`.

Código adicional en C#

Ahora hemos finalizado la comparación de las rutinas de manejo de eventos en C# y VB. Durante el proceso hemos aprendido mucho sobre las diferencias sintácticas entre los lenguajes. De hecho, hemos aprendido la mayor parte de la sintaxis básica que utiliza C# para encadenar las instrucciones. También hemos tenido nuestro primer contacto con el hecho que en C# todo es una clase. Sin embargo, si ha descargado el código de estos ejemplos del sitio web de Wrox Press, y le echa un vistazo, habrá notado que hemos evitado cualquier discusión sobre la diferencia más obvia entre los ejemplos: en el ejemplo C# hay mucho más código que un simple manejador de eventos. En la versión VB del ejemplo `SquareRoot`, el código del manejador de eventos que hemos presentado aquí representa el código fuente completo del proyecto. Sin embargo, en la versión C# del proyecto, este manejador de eventos es simplemente un método situado dentro de un gran fichero fuente que contiene una gran cantidad de código adicional.

La razón por la que hay tanto código adicional en el proyecto C# tiene que ver con el hecho de que el entorno de desarrollo de Visual Basic oculta en gran medida lo que está ocurriendo en su programa. En Visual Basic, todo lo que necesitamos escribir es el manejador de eventos, pero de hecho el ejemplo hace mucho más. Tiene que iniciarse, visualizar el formulario en pantalla, enviar información a Windows acerca de lo que quiere hacer con los eventos, y cerrar la ventana cuando la ejecución haya terminado. En Visual Basic no se tiene acceso al código que hace todo esto. Por el contrario, C# tiene una filosofía completamente diferente, y deja todo este código a la vista. Podría pensar que esto hace su código fuente más complicado, pero tiene la indudable ventaja de que si el código está disponible, Ud. puede editarlo, y eso significa que ganará mucho en flexibilidad al decidir cómo debe comportarse su aplicación.

De hecho, Visual Basic tiene tanto éxito a la hora de ocultar casi todo lo que va en su programa, que es muy fácil volverse muy productivo y escribir aplicaciones bastante sofisticadas sin tener una comprensión de la estructura completa de un programa. En la próxima sección veremos lo que realmente hay dentro de cualquier programa de ordenador, y entonces estaremos en condiciones para analizar todo el código adicional que Ud. ve en la versión C# de `SquareRoot`.

Lo que ocurre cuando se ejecuta un programa

Cualquier programa determina una secuencia precisa de ejecución de instrucciones. Cuando una aplicación es lanzada, hay un lugar preciso en el código ejecutable que el ordenador reconoce como el lugar por donde debe comenzar la ejecución del código. En otras palabras, la primera instrucción que se ejecutará. Después continuará ejecutando la siguiente instrucción, la siguiente, la siguiente, y así sucesivamente. Algunas de estas instrucciones le dirán al ordenador que salte a una instrucción diferente, quizás en dependencia de los valores contenidos en ciertas variables. Muy a menudo el ordenador saltará atrás y ejecutará de nuevo las mismas instrucciones que ya ha ejecutado antes. Sin embargo, siempre existe esta secuencia continua de ejecución de la próxima instrucción, hasta que el ordenador encuentra una orden que le dice que termine la ejecución del código. Esta secuencia lineal es válida en cualquier programa. Algunos programas pueden tener múltiples hilos, en cuyo caso habrá varias secuencias de ejecución (hilos), pero cada hilo de ejecución seguirá la secuencia antes indicada, desde una instrucción inicial hasta la terminación.

Por supuesto, esta secuencia no es lo que Ud. ve cuando escribe un programa en VB. En VB6, lo que Ud.

escribe es esencialmente un conjunto de manejadores de eventos – un conjunto de subrutinas, cada una de las cuales será llamada cuando el usuario realice una determinada acción. No es visible el punto de entrada del programa - el manejador de eventos `Form_Load` es el punto más cercano del que tenemos conocimiento. Aún así, `Form_Load` es realmente sólo otro manejador de eventos. Es el manejador del evento que se dispara cuando un formulario es cargado, lo cual significa que será el primer evento que se ejecute. De igual forma, si en lugar de un ejecutable está programando un control o un módulo de clase, no tiene un punto de inicio. Ud. simplemente escribe una clase y le añade diversos métodos y propiedades. Cada método o propiedad se ejecutará cuando el código cliente decida llamarlo, si lo hace.

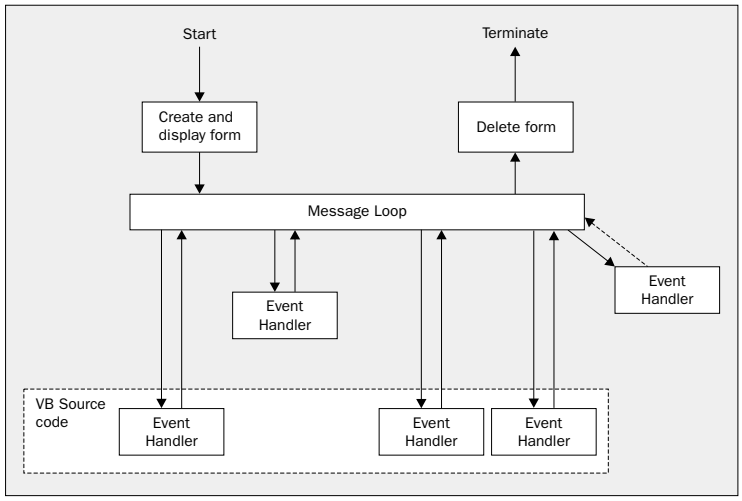
> *Realmente el párrafo anterior no es totalmente cierto. En VB, `Sub Main()` existe y actúa como el punto de entrada de un programa, pero no es muy utilizado. Debido a que estamos comparando un programa C# típico con un programa VB típico, nuestro señalamiento de que los programas VB sólo contienen el código para los eventos es válido.*

Para ver cómo podemos relacionar estas dos ideas de programación, veamos lo que realmente ocurre cuando se ejecuta una aplicación de Visual Basic – o cualquier otra aplicación Windows con interfaz de usuario gráfica, no importa en qué lenguaje esté escrita. Esto es un poco más restrictivo que las aplicaciones que hemos mencionado anteriormente, ya que ahora estamos centrando nuestra atención en las aplicaciones gráficas para Windows (en otras palabras, no tenemos en cuenta las aplicaciones de consola, los servicios, etc.).

Como de costumbre, la ejecución comienza en un punto bien definido. Las instrucciones que se ejecuten probablemente involucrarán la creación de algunas ventanas y controles, y la visualización de esos controles en la pantalla. En este punto, el programa hace algo que se conoce como entrar en un bucle de mensajes. Lo que realmente ocurre es que el programa se pone para dormir y le dice a Windows que lo despierte cuando ocurra algún suceso interesante del cual desea enterarse. Estas cosas 'interesantes' son los eventos para los que Ud. ha escrito manejadores, y también un buen número de eventos para los que Ud. no ha escrito sus propios manejadores, pero para los que el entorno de desarrollo, silenciosamente, haya proporcionado uno. Un buen ejemplo de esto último son los manejadores que tratan con el redimensionamiento de un formulario. Ud. nunca ve el código fuente de VB para esto, pero las aplicaciones VB responden correctamente cuando el usuario intenta redimensionar sus ventanas porque el entorno de desarrollo ha añadido al proyecto, sin informarle de ello, los manejadores de eventos que resuelven correctamente esta situación.

Cada vez que se produzca un evento, Windows despertará a la aplicación y llamará al manejador de eventos pertinente – ahí es donde el código que Ud. ha escrito puede comenzar a ejecutarse. Cuando la subrutina manejadora de eventos termina, la aplicación se pone de nuevo a dormir, diciéndole una vez más a Windows que la despierte cuando se produzca otro evento interesante. Finalmente, suponiendo que nada vaya desastrosamente mal, en algún momento Windows despertará la aplicación y le informará que tiene que terminar. En ese punto, la aplicación tomará alguna acción adecuada – por ejemplo, mostrando un cuadro de mensaje que pregunte al usuario si desea guardar un fichero – y terminará silenciosamente. De nuevo, la mayoría del código necesario para hacer esto ha sido añadida calladamente a su proyecto, detrás de la escena, por el entorno de VB, y Ud. nunca lo verá.

El flujo de la ejecución en una aplicación GUI típica de Windows es más o menos la siguiente:



En este diagrama, el recuadro con el borde discontinuo indica la parte de la ejecución a la que el entorno de desarrollo de VB le permite tener acceso y para la que puede escribir código fuente: algunos de los manejadores de eventos. El resto del código es inaccesible para Ud., aunque puede, en alguna medida, especificarlo mediante su opción de tipo de aplicación, cuando le pide a VB que cree un proyecto específico. Recuerde que cuando crea un nuevo proyecto en VB, aparece un cuadro de diálogo que le pregunta qué tipo de aplicación quiere crear – EXE estándar, EXE ActiveX, DLL ActiveX, etc. Lo que ocurre entonces es que VB utiliza la opción que Ud. ha seleccionado para generar el código adecuado para la parte del programa que está fuera del recuadro discontinuo en el diagrama. El diagrama muestra la situación en que Ud. elige crear un proyecto EXE estándar, y diferirá para otros tipos de proyecto (por ejemplo, una DLL ActiveX no tiene en absoluto un bucle de mensajes, sino que se apoya en los clientes para llamar a los métodos), pero debe darle una idea aproximada de lo qué ocurre.

Anteriormente hemos dicho que C# ofrece acceso a todo el código, pero hay que señalar que todos los pequeños detalles, como qué ocurre dentro del bucle de mensajes, están bien ocultos dentro de varias DLLs que Microsoft ha escrito; no obstante, Ud. consigue ver los métodos de alto nivel que llaman a las diferentes partes del proceso. Por ejemplo, tiene acceso al código que comienza a ejecutar el programa completo, a la llamada al método de librería que hace que su programa entre en el bucle de mensajes y lo pone dormir, y así sucesivamente. También tiene acceso al código fuente que instancia todos los controles que Ud. ha colocado en su formulario, los hace visibles, establece sus posiciones iniciales y tamaños, etc. Un aspecto que debo enfatizar es que Ud. no tendrá que escribir ninguno de estos fragmentos de código. Cuando utilice Visual Studio .NET para crear un proyecto C#, verá un cuadro de diálogo que le preguntará qué tipo de proyecto desea crear; entonces Visual Studio .NET escribirá todo el código de fondo para Ud. La diferencia está en que Visual Studio .NET producirá este código de fondo como código fuente C#, que luego Ud. podrá modificar directamente.

Hacer las cosas de esta manera, como hemos comentado, significa que su código fuente es más largo y más complejo. Sin embargo, la gran ventaja es que tiene mucha más flexibilidad para controlar lo que su programa hace y cómo se comporta. También significa que puede crear muchos más tipos de proyectos en C#. Mientras que en Visual Basic las únicas cosas que puede desarrollar son diferentes tipos de formulario y componentes COM, en C# puede escribir cualquiera de los diferentes tipos de programa que se ejecutan en Windows. Esto incluye, por ejemplo, las aplicaciones de consola (línea de comandos) y las páginas ASP.NET (el sucesor de ASP), que son imposibles de escribir utilizando VB6 (aunque se puede usar VBScript para páginas ASP). Sin embargo, en este apéndice nos concentraremos exclusivamente en las aplicaciones GUI clásicas de Windows.

El código C# para el resto del programa

En esta sección examinaremos el resto del código del ejemplo `SquareRoot`. Durante el proceso aprenderemos un poco más sobre las clases C#.

> *El ejemplo `SquareRoot` de C# fue creado en Visual Studio .NET, y el de VB en el entorno de desarrollo de VB. Sin embargo, el código presentado aquí realmente no es el que Visual Studio .NET generó. Aparte de añadir el manejador de eventos, hemos hecho otro par de modificaciones al código para ilustrar mejor los principios de la programación C#. Sin embargo, le dará una buena idea de la clase de trabajo que hace Visual Studio .NET cuando crea un proyecto para Ud.*

El texto completo del código fuente es bastante largo. Lo presentamos aquí en aras de la integridad, pero probablemente le resulte mejor saltar a la explicación siguiente, y referirse a este código fuente cuando sea necesario:

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

namespace Wrox.ProfessionalCSharp.AppendixC.SquareRootSample
{

   /// <summary>
   /// The Form that forms the main window for the app.
   /// </summary>

   public class SquareRootForm : System.Windows.Forms.Form
   {
      private System.Windows.Forms.TextBox txtNumber;
      private System.Windows.Forms.TextBox txtSign;
      private System.Windows.Forms.TextBox txtResult;
      private System.Windows.Forms.Button cmdShowResults;
      private System.Windows.Forms.Label label1;
      private System.Windows.Forms.Label label2;
      private System.Windows.Forms.Label label3;
      private System.Windows.Forms.Label label4;

      /// <summary>
      /// Required designer variable.
      /// </summary>
      private System.ComponentModel.Container components;
      public SquareRootForm()
      {
         InitializeComponent();
      }
      public override void Dispose()
      {
         base.Dispose();
         if(components != null)
            components.Dispose();
```

```
        }

        #region Windows Form Designer generated code

        /// <summary>
        /// Required method for Designer support - do not modify
        /// the contents of this method with the code editor.
        /// </summary>

        private void InitializeComponent()
        {
           this.txtNumber = new System.Windows.Forms.TextBox();
           this.txtSign = new System.Windows.Forms.TextBox();
           this.cmdShowResults = new System.Windows.Forms.Button();
           this.label3 = new System.Windows.Forms.Label();
           this.label4 = new System.Windows.Forms.Label();
           this.label1 = new System.Windows.Forms.Label();
           this.label2 = new System.Windows.Forms.Label();
           this.txtResult = new System.Windows.Forms.TextBox();
           this.SuspendLayout();

           //
           // txtNumber
           //

           this.txtNumber.Location = new System.Drawing.Point(160, 24);
           this.txtNumber.Name = "txtNumber";
           this.txtNumber.TabIndex = 0;
           this.txtNumber.Text = "";

           //
           // txtSign
           //

           this.txtSign.Enabled = false;
           this.txtSign.Location = new System.Drawing.Point(160, 136);
           this.txtSign.Name = "txtSign";
           this.txtSign.TabIndex = 1;
           this.txtSign.Text = "";

           //
           // cmdShowResults
           //

           this.cmdShowResults.Location = new System.Drawing.Point(24, 96);
           this.cmdShowResults.Name = "cmdShowResults";
           this.cmdShowResults.Size = new System.Drawing.Size(88, 23);
           this.cmdShowResults.TabIndex = 3;
           this.cmdShowResults.Text = "Show Results";
           this.cmdShowResults.Click += new
                              System.EventHandler(this.OnClickShowResults);

           //
           // label3
           //
```

```
            this.label3.Location = new System.Drawing.Point(72, 24);
            this.label3.Name = "label3";
            this.label3.Size = new System.Drawing.Size(80, 23);
            this.label3.TabIndex = 6;
            this.label3.Text = "Input a number";

            //
            // label4
            //

            this.label4.Location = new System.Drawing.Point(80, 184);
            this.label4.Name = "label4";
            this.label4.Size = new System.Drawing.Size(80, 16);
            this.label4.TabIndex = 7;
            this.label4.Text = "Square root is";

            //
            // label1
            //

            this.label1.Location = new System.Drawing.Point(112, 136);
            this.label1.Name = "label1";
            this.label1.Size = new System.Drawing.Size(40, 23);
            this.label1.TabIndex = 4;
            this.label1.Text = "Sign is";

            //
            // label2
            //

            this.label2.Location = new System.Drawing.Point(48, 184);
            this.label2.Name = "label2";
            this.label2.Size = new System.Drawing.Size(8, 8);
            this.label2.TabIndex = 5;

            //
            // txtResult
            //

            this.txtResult.Enabled = false;
            this.txtResult.Location = new System.Drawing.Point(160, 184);
            this.txtResult.Name = "txtResult";
            this.txtResult.TabIndex = 2;
            this.txtResult.Text = "";

            //
            // Form1
            //

            this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
            this.ClientSize = new System.Drawing.Size(292, 269);
            this.Controls.AddRange(new System.Windows.Forms.Control[]
                                   {this.label4,
                                    this.label3,
```

```
                                    this.label2,
                                    this.label1,
                                    this.cmdShowResults,
                                    this.txtResult,
                                    this.txtSign,
                                    this.txtNumber});
            this.Name = "Form1";
            this.Text = "Square Root C# Sample";
            this.ResumeLayout(false);
        }

        #endregion

        /// <summary>
        /// Event handler for user clicking Show Results button.
        /// Displays square root and sign of number
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>

        private void OnClickShowResults(object sender, System.EventArgs e)
        {
            float NumberInput = float.Parse(this.txtNumber.Text);
            if (NumberInput < 0)
            {
                this.txtSign.Text = "Negative";
                this.txtResult.Text = Math.Sqrt(-NumberInput) + " I";
            }
            else if (NumberInput == 0)
            {
                txtSign.Text = "Zero";
                txtResult.Text = "0";
            }
            else
            {
                this.txtSign.Text = "Positive";
                this.txtResult.Text = Math.Sqrt(NumberInput).ToString();
            }
        }
    }
    class MainEntryClass
    {

        /// <summary>
        /// The main entry point for the application.
        /// </summary>

        [STAThread]
        static void Main()
        {
            SquareRootForm TheMainForm = new SquareRootForm();
            Application.Run(TheMainForm);
        }
    }
}
```

Espacios de nombres

La parte principal del código fuente C# de `SquareRoot` comienza con un par de declaraciones de espacios de nombres y una declaración de una clase:

```
namespace Wrox.ProfessionalCSharp.AppendixC.SquareRootForm
{
   public class SquareRootForm : System.Windows.Forms.Form
   {
```

La clase `SquareRootForm` va a contener casi todo el código – todos los métodos, etc., con una pequeña cantidad de código contenida en otra clase llamada `MainEntryClass`. Recuerde que es más fácil pensar en una clase como en un módulo de clase VB – aunque podemos ver aquí que una diferencia es que en C# vemos el código fuente que da inicio a la declaración de la clase. En VB, el entorno de desarrollo simplemente ofrece una ventana separada con el contenido de la clase en ella.

Un espacio de nombres es algo que realmente no tiene análogo en VB, y la manera más fácil de pensar en ellos es como una manera de organizar los nombres de sus clases, casi de la misma manera que un sistema de ficheros organiza los nombres de sus ficheros. Por ejemplo, suponga que tuviera una gran cantidad de ficheros en su disco duro, todos llamados `ReadMe.Txt`. Si ese nombre, `ReadMe.Txt`, fuera la única información que tuviera sobre cada fichero, no tendría ninguna manera de distinguir entre todos ellos. Sin embargo, podrá distinguirlos utilizando el nombre de la ruta completa; por ejemplo, en mi ordenador uno de ellos es `C:\Archivos de programa\HTML Help Workshop\ReadMe.txt`, mientras que otro es `C:\Program Files\ReadMe.txt`.

Los espacios de nombres funcionan de una manera similar, pero sin el coste de tener un sistema de ficheros real – los espacios no son básicamente nada más que etiquetas. No tiene que hacer nada para crear un espacio de nombres, sólo lo declara en su código de la forma en que los hemos hecho anteriormente en nuestro ejemplo. El código presentado anteriormente significa que el nombre completo de la clase que hemos definido no es `SquareRootForm`, sino `Wrox.ProfessionalCSharp.AppendixC.SquareRootForm`. Es sumamente improbablemente que alguien más en el mundo escriba una clase con ese nombre completo. Por otro lado, si no tuviéramos el espacio de nombres, habría más riesgo de confusión, porque cualquier otro programador podría escribir una clase llamada `SquareRootForm`.

Evitar las colisiones de esta forma es importante en C#, porque el ambiente .NET utiliza únicamente estos nombres para identificar a las clases, del mismo modo que los controles ActiveX creados por VB utilizan un complejo mecanismo, que involucra los GUIDs, para evitar las colisiones entre nombres. Microsoft ha optado por el concepto más simple de los espacios de nombres, conscientes de que algunas de las complejidades de COM, como los GUIDs, hacen innecesariamente difícil a los programadores escribir buenas aplicaciones Windows.

Aunque en C# el uso de los espacios de nombres no es estrictamente obligatorio, se recomienda encarecidamente que coloque todas sus clases dentro de un espacio de nombres, para evitar cualquier posible colisión de nombres con otro software. De hecho, es sumamente raro ver código C# que no comience con una declaración de espacio de nombres.

Los espacios de nombres puede anidarse. Por ejemplo, la declaración anterior:

```
namespace Wrox.ProfessionalCSharp.AppendixC.SquareRootSample
{
   public class SquareRootForm : System.Windows.Forms.Form
   {
      // etc.
```

```
    }
}
```

podría haberse escrito de la siguiente forma:

```
namespace Wrox
{
   namespace ProfessionalCSharp
   {
      namespace AppendixC
      {
         namespace SquareRootSample
         {
            public class SquareRootForm : System.Windows.Forms.Form
            {
               // etc.
            }
         }
      }
   }
}
```

En este código hemos mantenido las llaves para enfatizar que siempre tienen que estar en parejas. Las llaves se utilizan para marcar los límites de los espacios de nombres y las clases, de la misma forma que se utilizan para marcar los límites de los métodos y las instrucciones compuestas.

La instrucción using

La parte inicial del código del proyecto `SquareRoot` consiste de instrucciones `using`:

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
```

Estas instrucciones `using` están aquí para simplificar el código. Los nombres completos de las clases, incluidos los espacios de nombres a los que pertenecen, son largos. Por ejemplo, más adelante en este código se definen dos cuadros de texto. Un cuadro de texto es representado por la clase `System.Windows.Forms.TextBox`. Si tuviéramos que escribir todo eso en nuestro código cada vez que quisiéramos hacer referencia a un `TextBox`, nuestro código se haría más complicado. Para evitarlo, la instrucción `using System.Windows.Forms`; le dice al compilador que busque en ese espacio de nombres cualquier clase que no esté en el espacio de nombres actual, y para la cual no hayamos especificado un espacio de nombres. Después de esa declaración, podemos simplemente escribir `TextBox` cada vez que queramos referirnos a esa clase. Es usual comenzar cualquier programa C# con varias instrucciones `using` que pongan en uso todos los espacios de nombres que vamos a utilizar en el código. Los espacios de nombres que aparecen en el ejemplo son todos espacios de nombres que abarcan varias partes de la librería de clases base .NET, permitiéndonos así utilizar convenientemente diversas clases base .NET.

La definición de la clase: la herencia

Veamos ahora la definición de la clase `SquareRootForm`. La propia definición es bastante simple:

```
public class SquareRootForm : System.Windows.Forms.Form
{
```

La palabra reservada `class` le dice al compilador que vamos a definir una clase. La parte interesante son los dos puntos después del nombre de la clase, seguidos por otro nombre, `Form`. Este es el punto en el que tenemos que traer a colación ese otro concepto importante que dijimos que es necesario aprender para entender la programación en C#: la herencia.

Lo que hace la sintaxis anterior es decirle al compilador que nuestra clase `SquareRootForm` hereda de la clase `Form` (realmente `System.Windows.Forms.Form`). Lo que esto significa es que nuestra clase no sólo contiene cualquier método, propiedades, etc. que hayamos definido, sino también todo lo que estaba en `Form`. `Form` es una clase base .NET extraordinariamente poderosa, que ofrece todas las características de un formulario básico. Contiene métodos que hacen que el formulario se visualice a sí mismo, y un gran número de propiedades que incluyen `Height`, `Width`, `DesktopLocation`, `BackColor` (el color de fondo del formulario), y que controlan la apariencia del formulario en pantalla. Al heredar de esta clase, nuestra propia clase obtiene inmediatamente todas estas características, y es por consiguiente un formulario completo. La clase de la que se hereda es conocida como la **clase base**, y la nueva clase es conocida como la **clase derivada**.

Si ha utilizado interfaces anteriormente, la herencia no será nueva para Ud., ya que sabrá que las interfaces pueden heredar unas de otras. Lo que tenemos aquí, sin embargo, es mucho más poderoso que la herencia de interfaces. Cuando una interfaz COM hereda de otra interfaz, todo lo que obtiene son los nombres y signaturas de los métodos y propiedades. Esto es, después de todo, lo que una interfaz contiene. Sin embargo, una clase contiene además todo el código que implementa estos métodos, etc. Esto significa que `SquareRootForm` obtiene todas las implementaciones de `Form`, y no sólo los nombres de los métodos. Este tipo de herencia es conocido como herencia de implementación, y no es nueva en C#; ha sido un concepto clásico y fundamental de la programación orientada a objetos (POO) durante décadas. Los programas C++, en particular, utilizan el concepto ampliamente, pero no está soportado en VB (la herencia de implementación tiene ciertas similitudes con la subclasificación). Cuando se acostumbre a escribir programas C#, encontrará que la arquitectura completa de un programa C# típico está casi invariablemente basada alrededor de la herencia de implementación.

Pero la herencia de implementación es aún más poderosa que eso. Como veremos después, cuando una clase hereda de otra clase, no tiene que aceptar **todas** las implementaciones de todo lo que está en la clase base. Si quiere, puede modificar las implementaciones de métodos y propiedades particulares, utilizando una técnica llamada **redefinición** (**overriding**). Esto significa que se puede crear una clase que sea muy similar a una clase existente, pero que tenga algunas diferencias en cuanto a cómo funciona o a lo que hace. Eso hace muy fácil la reutilización del código que otras personas han escrito, y ahorra mucho tiempo de programación. También es importante entender que no necesita el acceso al código fuente de la clase base para derivar de ella. Por razones comerciales obvias, Microsoft conserva para sí el código fuente de la clase `Form`. El hecho de que la librería compilada esté disponible en forma de una unidad de ensamblaje es suficiente para que podamos heredar de la clase, tomando los métodos que queramos y redefiniendo los que no.

Punto de entrada del programa

Saltaremos ahora casi al final del código del ejemplo, para examinar el punto de entrada del programa principal: la función `Main()`, reproducida a continuación:

```
class MainEntryClass
```

```
    {

        /// <summary>
        /// The main entry point for the application.
        /// </summary>

        [STAThread]
        static void Main()
        {
            SquareRootForm TheMainForm = new SquareRootForm();
            Application.Run(TheMainForm);
        }
    }
```

Esto no parece a primera vista un punto de entrada de programa, pero lo es. La regla en C# es que la ejecución del programa comienza por un método llamado `Main()`. Este método debe definirse como un método estático en alguna clase. Normalmente debe haber sólo un método `Main()` en todas las clases del código fuente que responda a esta descripción en el programa – de lo contrario, el compilador no sabrá cuál elegir. Aquí `Main()` se define como un método que no recibe ningún parámetro y devuelve `void` (en otras palabras, no devuelve nada). Esta no es la única signatura posible para el método, pero es la usual en una aplicación Windows (las aplicaciones de línea de comandos pueden recibir parámetros: los argumentos que se especifiquen en la línea de comandos).

> *Como hemos mencionado, el código VB puede tener un método `Main()`, pero es opcional y raramente se utiliza. En C#, el método `Main()` tiene que estar presente como punto de entrada principal del programa.*

Como `Main()` tiene que estar en alguna clase, lo hemos puesto en una clase llamada `MainEntryClass`. Esta clase no contiene nada más, pero ese no es necesariamente el caso – es totalmente legítimo que la clase que contiene el punto de entrada principal contenga también otros métodos. El hecho de que `Main()` sea un método estático es importante. Recuerde que hemos dicho que los métodos estáticos son métodos especiales que pueden ejecutarse sin tener que crear antes un objeto de la clase. Bien, ya que lo primero que ocurre cuando el programa se ejecuta es la llamada a `Main()`, no hay aún instancias de ninguna clase – todavía no hemos ejecutado ningún código para crearlas. Por eso es que el punto de entrada tiene que ser estático.

Aparte de la palabra reservada `static`, la definición de `Main()` se parece mucho a la definición del método que hemos examinado anteriormente. Sin embargo, está antecedida por la palabra `[STAThread]` entre corchetes. `STAThread` es un ejemplo de atributo – otro concepto que no tiene paralelo en el código fuente de VB.

Un atributo es algo que proporciona información adicional al compilador sobre algún elemento en el código, y siempre toma la forma de una palabra (posiblemente con algunos parámetros, aunque no en este caso) entre corchetes, inmediatamente antes del elemento al que se aplica. Este atributo particular le dice al compilador el modelo de hilos que el código debe utilizar. No entraremos en los detalles sobre los modelos de hilos de ejecución, pero diremos que escribir `[STAThread]` en el código fuente C# tiene un efecto similar a seleccionar el modelo de hilos en las Propiedades del Proyecto en el entorno de desarrollo de VB, aunque en VB sólo puede se hacer esto para proyectos de DLLs y controles ActiveX. Observe además que la analogía es sólo aproximada, porque el atributo de C# especifica un modelo de hilos de .NET, y no de COM.

A propósito, esta comparación muestra una vez más la diferencia filosófica entre C# y VB: en VB, el modelo de hilos está allí y necesita ser especificado, pero está oculto por el entorno de desarrollo de VB, por lo que no puede acceder a él a través del código fuente VB –tiene que acceder a él a través de la configuración del proyecto.

Instanciación de clases

Examinemos ahora el código dentro del método `Main()`. Lo primero que hay que hacer es crear el formulario; en otras palabras, instanciar un objeto `SquareRootForm`. De esto se encarga la primera línea de código:

```
SquareRootForm TheMainForm = new SquareRootForm();
```

Obviamente no podemos comparar esto con el código VB, ya que los comandos correspondientes en VB no están disponibles como código fuente, pero podemos hacer una comparación – si imaginamos que en algún programa VB vamos a crear un cuadro de diálogo. En VB, la manera de hacerlo se parecería a ésta:

```
Dim SomeDialog As MyDialogClass
Set SomeDialog = New MyDialogClass
```

En este código VB, primero declaramos una variable que es una referencia a un objeto: `SomeDialog` hará referencia a una instancia de `MyDialogClass`. A continuación, instanciamos un objeto utilizando la palabra reservada `New` de VB, y hacemos que nuestra variable haga referencia a él.

Eso es exactamente lo que ocurre también en el código C#: declaramos una variable llamada `TheMainForm`, que es una referencia a `SquareRootForm`, y entonces utilizamos la palabra reservada `new` de C# para crear una instancia de `SquareRootForm`, y hacemos que nuestra variable haga referencia a ella. La diferencia sintáctica principal es que C# nos permite combinar ambas operaciones en una instrucción, de la misma manera que pudimos previamente declarar e inicializar la variable `NumberInput` de una sola vez. También observará los paréntesis después de la expresión `new`. Ese es un requisito de C#. Al crear objetos, siempre tendrá que escribir esos paréntesis. La razón es que C# trata de crear un objeto un poco a través de una llamada a un método, hasta el punto de que a veces es posible pasar parámetros en la llamada a `new`, para indicar cómo se desea inicializar el nuevo objeto. En este caso, no pasamos ningún parámetro, pero aún así necesitamos los paréntesis.

Las clases C#

Hasta ahora hemos dicho que las clases C# son similares a los módulos de clase VB. Ya hemos visto una diferencia en el hecho de que las clases C# permiten métodos estáticos. En el código anterior para el método `Main()` ahora resalta otra diferencia: si estuviéramos haciendo algo así en VB, también tendríamos que asignar `Nothing` al objeto creado cuando hayamos terminado con él. Sin embargo, nada parecido a esto aparece en nuestro código C#, porque en C# no es necesario.

La razón de esta diferencia tiene que ver con el hecho de que las clases C# son más eficientes y ligeras que las correspondientes en VB. Los objetos de clase VB son realmente objetos COM, lo cual significa que cada uno incluye algún código sofisticado que verifica cuántas referencias están haciéndose al objeto, para que cada objeto pueda destruirse a sí mismo cuando descubra que ya no es necesario. En VB se considera una mala práctica no asignar `Nothing` a la referencia al objeto cuando se ha terminado de trabajar con él, porque esto significa que el objeto no sabrá que ya no es necesario, y quedará "colgado" en memoria, posiblemente hasta que termine todo el proceso.

Por razones de rendimiento, los objetos C# no realizan este tipo de verificación. En cambio, C# utiliza un mecanismo llamado recolección de basura. Lo que ocurre es que, en lugar de que cada objeto verifique si todavía debe seguir vivo, de vez en cuando el motor de ejecución .NET transfiere el control al **recolector de basura**. El recolector de basura examina el estado de la memoria, utiliza un algoritmo muy eficiente para identificar los objetos a los que su código ya no está haciendo referencia, y los elimina. Gracias a este mecanismo deja de ser importante anular las referencias cuando se haya terminado con ellas – simplemente es suficiente esperar hasta que la variable salga de alcance.

Sin embargo, si desea que las variables de tipos-referencia no hagan referencia a nada, entonces la palabra reservada pertinente es `null`, que significa lo mismo que `Nothing` en VB. Por lo tanto, en VB escribiría:

```
Set SomeDialog = Nothing
```

Y en C# escribiría:

```
TheMainForm = null;
```

Tenga en cuenta, sin embargo, que esto de por sí realmente no logrará mucho en C#, a menos que todavía la variable `TheMainForm` tenga un largo tiempo de vida, debido a que el objeto no será destruido hasta tanto el recolector de basura no sea llamado.

Entrada al bucle de mensajes

Consideraremos ahora la instrucción final del método principal:

```
    Application.Run(TheMainForm);
```

Esta instrucción es la que entra en el bucle de mensajes. Lo que realmente estamos haciendo es llamar a un método estático de la clase `System.Windows.Forms.Application`. El método en cuestión es el método `Run()`. Este método maneja el bucle de mensajes. Pone a la aplicación (o hablando estrictamente, al hilo de ejecución) a dormir y solicita a Windows que la despierte siempre que se produzca un evento interesante. El método `Run()` recibe un parámetro, que es una referencia al formulario que manejará todos los eventos. `Run()` termina cuando se produzca un evento que indique al formulario que debe terminar y este evento sea procesado.

Una vez que el método `Run()` ha terminado, no hay nada más que hacer, por lo que el método `Main()` retorna. Debido a que este método era el punto de entrada del programa, cuando él termina la ejecución de todo el proceso se detiene.

Un elemento de la sintaxis en las instrucciones anteriores que podría encontrar sorprendente es que utilizamos paréntesis al llamar el método `Run()`, aunque no estamos utilizando el valor devuelto por ese método y por lo tanto estamos haciendo lo equivalente a llamar a una subrutina de VB. En esta situación, VB no requiere paréntesis, pero la regla es que en C# siempre se deben utilizar paréntesis al llamar a un método.

> **Utilice siempre paréntesis al llamar a un método en C#, tanto si el método requiere parámetros como si no, y tanto si va a utilizar el valor devuelto como si no.**

La clase SquareRootForm

Hemos visto cómo C# entra en un bucle de mensajes, pero todavía no hemos analizado el proceso de visualizar y crear el propio formulario. También hemos sido bastante vagos en lo que respecta a la llamada a los manejadores de eventos: hemos indicado que Windows llama a los manejadores de eventos, tales como el método `OnClickButtonResults()`. Pero, ¿cómo sabe Windows cuál es el método que debe ser llamado? Podemos encontrar las respuestas a esas preguntas en la definición de la clase `SquareRootForm`, y en su clase base, `Form`.

En primer lugar, notemos que la clase `SquareRootForm` tiene varios campos miembros (**campo miembro** es

la forma de nombrar en C# a una variable que se define como miembro de una clase. Puede pensar en ellos como en variables VB que tienen alcance de formulario, o alternativamente como variables VB que se definen como miembros de un módulo de clase. Cada una de estas variables está asociada con una instancia particular de una clase – un objeto particular – y permanece en alcance mientras tanto el objeto que la contiene permanezca vivo):

```
public class SquareRootForm : System.Windows.Forms.Form
{
   private System.Windows.Forms.TextBox txtNumber;
   private System.Windows.Forms.TextBox txtSign;
   private System.Windows.Forms.TextBox txtResult;
   private System.Windows.Forms.Button cmdShowResults;
   private System.Windows.Forms.Label label1;
   private System.Windows.Forms.Label label2;
   private System.Windows.Forms.Label label3;
   private System.Windows.Forms.Label label4;
```

Cada uno de estos campos corresponde a uno de los controles. Podrá ver claramente los tres cuadros de texto y el botón. Hay también cuatro etiquetas que se corresponden a las áreas de texto en el formulario. No haremos nada en el código con estas etiquetas, por lo que no nos hemos molestado en darles nombres más amistosos.

Sin embargo, cada una de estas variables es sólo una referencia a un objeto, de manera que el hecho que estas variables existan no implica que exista ninguna instancia de esos objetos – los objetos tienen que ser instanciados separadamente. El proceso de instanciar estos controles se hace en un método conocido como **constructor**. Un constructor en C# es un análogo de las subrutinas de VB `Form_Load()`, `Form_Initialize()`, `Class_Load()` y `Class_Initialize()`. Es un método especial que se llama automáticamente cada vez que una instancia de una clase va a ser creada, y contiene el código necesario para inicializar la instancia.

Podemos distinguir el constructor en la clase porque un constructor siempre tiene el mismo nombre que la clase. En este caso, simplemente buscamos un método llamado `SquareRootForm`:

```
   public SquareRootForm()
   {
      InitializeComponent();
   }
```

Observe que debido a que éste es un constructor en lugar de un método corriente, no devuelve ningún tipo. Sin embargo, tiene paréntesis después de su nombre, al igual que todos los métodos. Puede utilizar estos paréntesis para especificar los parámetros que van a ser pasados al constructor (recuerde que anteriormente hemos dicho que es posible pasar parámetros en los paréntesis que siguen a la cláusula `new`, al crear una variable – pues bien, esos serán los parámetros que recibirá el constructor, en caso de que los haya). La definición del constructor indica si se necesita algún parámetro para crear una instancia del objeto. Aquí no tenemos ningún parámetro – veremos ejemplos de constructores que reciben parámetros en el ejemplo `Employee`, más adelante en este apéndice.

En este caso, el constructor sólo incluye una llamada al método `InitializeComponent()`. Esto realmente se debe a Visual Studio .NET. Visual Studio .NET ofrece las mismas posibilidades que el entorno de desarrollo de VB6 para manipular gráficamente los controles. Sin embargo, ahora en C# las definiciones de todos los controles se asignan en el código fuente, y Visual Studio .NET tiene que ser capaz de leer el código fuente para averiguar qué controles están presentes en el formulario. Y esto lo hace buscando el método `InitializeComponent()`, y viendo qué controles están instanciados allí.

`InitializeComponent()` es un método largo y no lo estudiaremos completo, pero comienza así:

```
        private void InitializeComponent()
        {
            this.txtNumber = new System.Windows.Forms.TextBox();
            this.txtSign = new System.Windows.Forms.TextBox();
            this.cmdShowResults = new System.Windows.Forms.Button();
            this.label3 = new System.Windows.Forms.Label();
            this.label4 = new System.Windows.Forms.Label();
            this.label1 = new System.Windows.Forms.Label();
            this.label2 = new System.Windows.Forms.Label();
            this.txtResult = new System.Windows.Forms.TextBox();
```

El fragmento de código anterior contiene un conjunto de llamadas para instanciar todos los controles del formulario. Este fragmento realmente no contiene ninguna sintaxis nueva de C# que ya no hayamos encontrado. El siguiente fragmento asigna valores a las propiedades de los controles:

```
            //
            // txtNumber
            //

            this.txtNumber.Location = new System.Drawing.Point(160, 24);
            this.txtNumber.Name = "txtNumber";
            this.txtNumber.TabIndex = 0;
            this.txtNumber.Text = "";

            //
            // txtSign
            //

            this.txtSign.Enabled = false;
            this.txtSign.Location = new System.Drawing.Point(160, 136);
            this.txtSign.Name = "txtSign";
            this.txtSign.TabIndex = 1;
            this.txtSign.Text = "";
```

Este código asigna las coordenadas y el texto inicial de dos de los controles, el cuadro de texto de entrada y el cuadro de texto que visualiza el signo del número introducido. Algo nuevo en el código es que la localización de los controles (relativa al extremo superior izquierdo de la pantalla) se especifica utilizando la clase `Point`. `Point` es una clase base .NET (más exactamente, es una estructura) que almacena un par de coordenadas `x` e `y`. La sintaxis de las dos líneas anteriores que asignan `Location` es instructiva. La propiedad `TextBox.Location` es sólo una referencia a `Point`, y para asignarle un valor necesitamos crear e inicializar un objeto `Point` que contenga las coordenadas correctas. Esta es la primera vez que vemos un constructor que recibe parámetros – en este caso las coordenadas horizontal y vertical del control. Si quisiéramos traducir una de estas líneas a VB, y suponiendo que hubiéramos definido un módulo de clase VB llamado `Point`, y que tuviéramos una clase que tuviera esa propiedad, podríamos escribir algo similar a esto:

```
Dim Location As Point
Set Location = New Point
Location.X = 160
Location.Y = 24
SomeObject.Location = Location
```

Esto sería el equivalente de este código C#:

```
someObject.Location = new System.Drawing.Point(160, 24);
```

¡Es obvio lo compacto de la notación C#!

Veamos ahora las mismas instrucciones para el botón. Encontraremos las asignaciones a los mismos tipos de propiedades, pero en este caso hay algo adicional: tenemos que decirle a Windows que llame a nuestro manejador de eventos cuando se haga clic en el botón. La línea que hace esto está resaltada:

```
            this.cmdShowResults.Name = "cmdShowResults";
            this.cmdShowResults.Size = new System.Drawing.Size(88, 23);
            this.cmdShowResults.TabIndex = 3;
            this.cmdShowResults.Text = "Show Results";
            this.cmdShowResults.Click += new
                                    System.EventHandler(this.OnClickShowResults);
```

Lo que ocurre aquí es lo siguiente. El botón, al que se hace referencia mediante la referencia `cmdShowResults`, contiene un evento, `Click`, que se disparará cuando el usuario haga clic en el botón. Tenemos que agregar este evento a nuestro manejador de eventos. Ahora bien, C# no nos permite pasar nombres de métodos directamente – en vez de eso, tenemos que envolverlos en algo conocido como un **delegado**. Los detalles de esto están más allá del alcance de este apéndice – se cubren completamente en el Capítulo 6 de este libro, pero diremos que los delegados permiten garantizar la seguridad de tipos, y ésta es la razón para la creación de la instancia de `System.EventHandler()` en el código. Una vez que hemos encapsulado el nombre del manejador de eventos, lo asociamos al evento utilizando el operador `+=`, que discutiremos a continuación.

Operadores de asignación aritmética

El símbolo `+=` representa lo que se conoce en C# como el operador de suma-asignación. Provee una abreviatura conveniente para los casos en que se desea añadir una cantidad a otra. Digamos que se han declarado dos enteros `A` y `B` en un programa VB, y que desea escribir:

```
B = B + A
```

En C#, el tipo equivalente es `int`, y se puede escribir exactamente lo mismo:

```
B = B + A;
```

Sin embargo, en C# hay una abreviatura alternativa para esto:

```
B += A;
```

`+=` realmente significa añadir la expresión de la derecha a la variable de la izquierda, y funciona para todos los tipos de datos numéricos, no sólo para `int`. También hay otros operadores similares, como `*=`, `/=`, y `-=`, que proporcionan atajos similares para la multiplicación, división y resta, respectivamente. Por ejemplo, para dividir un número entre 2, y asignar el resultado a `B`, escribiría:

```
B /= 2;
```

Aunque no entraremos en detalles en este apéndice, C# también ofrece otros operadores para la manipulación de bits, así como uno que produce el resto de la división – y para todos éstos existen los operadores de asignación correspondientes. Los detalles sobre estos operadores están en el Capítulo 3.

En el ejemplo `SquareRootForm`, hemos aplicado el operador de suma-asignación a un evento; la línea:

```
            this.cmdShowResults.Click += new
                                  System.EventHandler(this.OnClickShowResults);
```

simplemente significa 'añada este manejador al evento'. Podrá estar un poco sorprendido al ver que un operador como += puede aplicarse a algo que no es un tipo numérico simple como `int` o `float`, pero esto ilustra un aspecto importante acerca de los operadores en C#:

> **Los operadores como +, -, *, etc. en VB sólo tienen sentido cuando se aplican a tipos de datos numéricos. Pero en C# pueden aplicarse a cualquier tipo de objetos.**

La frase anterior debe explicarse un poco más. Para poder aplicar los operadores a otros tipos de objetos, primero es necesario decirle al compilador lo que estos operadores significan para esos otros tipos de objetos – un proceso conocido como **sobrecarga de operadores**. Suponga que desea escribir una clase que permita representar vectores matemáticos. Algo que en VB codificaría como un módulo de clase, y que le permitiría escribir:

```
Dim V1 As Vector
Set V1 = New Vector
```

En matemáticas es posible sumar vectores, y aquí es donde aparecería la sobrecarga de operadores. Pero como VB6 no soporta la sobrecarga de operadores, en vez de ello definirá un método, `Add` como éste:

```
' V1, V2 y V3 son de tipo Vector
Set V3 = V1.Add(V2)
```

En VB, esto es lo mejor que se podría hacer. Sin embargo, en C# se puede definir una clase `Vector` y sobrecargar el operador + para la clase. La sobrecarga del operador es básicamente un método con el nombre `operator +`, y que el compilador llamará si ve + aplicado a una instancia de `Vector`. Eso significa que en C# podría escribir:

```
// V1, V2 y V3 son instancias de Vector

V3 = V1 + V2;
```

Obviamente, no querrá definir sobrecargas de operadores para todas las clases. Para la mayoría de las clases que Ud. escribe no tendría sentido hacer cosas como sumar o multiplicar objetos. Sin embargo, para las clases que tenga sentido hacerlo, la sobrecarga de operadores puede ser una excelente forma de hacer su código más fácil de leer. Eso es lo que ha ocurrido con los eventos. Ya que tiene sentido hablar de añadir un manejador a un evento, se ha proporcionado una sobrecarga del operador + para permitirnos hacer esto utilizando una sintaxis intuitiva basada en los operadores + (y +=). También puede utilizar – ó -= para "desconectar" un manejador de un evento.

Resumen

Hemos ido tan lejos como hemos podido con el ejemplo `SquareRootForm`. Hay mucho más código C# que no hemos examinado en la versión C# de esta aplicación, pero este código adicional tiene que ver con la inicialización de los demás controles del formulario, y no introduce ningún principio nuevo, por lo que no vamos a examinarlo.