Ediciones ENI

C# 7 y Visual Studio 2017

Los fundamentos del lenguaje

Colección
Recursos Informáticos

Extracto del Libro

Capítulo 4
Las bases del lenguaje

1. Introducción

Como todos los lenguajes de programación, C# impone ciertas reglas al desarrollador. Estas reglas se ponen de manifiesto a través de la sintaxis del lenguaje, aunque cubren el amplio espectro funcional propuesto por C#. Antes de explorar en profundidad las funcionalidades del lenguaje en el siguiente capítulo, estudiaremos las nociones esenciales y fundamentales de C#: la creación de los datos que puede utilizar una aplicación y el procesamiento de estos datos.

2. Las variables

Los datos que se utilizan en un programa C# se representan mediante variables. Una variable es un espacio de memoria reservado al que se asigna arbitrariamente un nombre y cuyo contenido es un valor con un tipo fijado. Es posible manipular ese contenido en el código utilizando el nombre de la variable.

2.1 Nomenclatura de las variables

La especificación del lenguaje C# establece algunas reglas que deben tenerse en cuenta cuando se asigna el nombre a una variable:

– El nombre de una variable puede contener únicamente cifras, caracteres del alfabeto latino –acentuados o no–, el carácter ç o los caracteres especiales _ y µ.
– El nombre de una variable no puede, en ningún caso, empezar por una cifra. No obstante, puede incluirlas en cualquier otra posición.
– El lenguaje es sensible a mayúsculas y minúsculas: las variables `unavariable` y `unaVariable` son, por tanto, variables distintas.
– La longitud de un nombre de variable es prácticamente ilimitada: ¡el compilador no muestra ningún error hasta que tenga 30.000 caracteres! Se recomienda no utilizar nombres de variables demasiado largos, la longitud máxima, en la práctica, no suele superar un máximo de treinta caracteres.
– El nombre de una variable no puede ser una palabra clave del lenguaje. Es posible, no obstante, prefijar una palabra clave con un carácter autorizado o con el carácter @ para utilizar un nombre de variable similar.

Los nombres de variables siguientes son válidos en el compilador C#:

– `miVariable`
– `miVariableNúmero1`
– `@void` (`void` es una palabra clave de C#)
– `µn4_VàRïãbl3`

De manera general, es preferible utilizar nombres de variables explícitos, es decir, que permitan saber a qué se corresponde el valor almacenado en la variable, como `nombreCliente`, `importeCompra` o `edadDelEmpleado`.

© Editions ENI - All rights reserved

2.2 Tipo de las variables

Una de las características de C# es la noción de tipado estático: cada variable se corresponde con un tipo de datos y no puede cambiar. Además, este tipo debe poder determinarse en tiempo de compilación.

2.2.1 Tipos valor y tipos referencia

Los distintos tipos que pueden utilizarse en C# se dividen en dos familias: los tipos valor y los tipos referencia. Esta noción puede resultar algo desconcertante a primera vista, pues una variable representa exactamente un dato y, por tanto, un valor. Este concepto está, de hecho, vinculado con la manera en que se almacena la información en memoria.

Cuando se utiliza una variable de tipo valor, se accede directamente a la zona de memoria que almacena el dato. En el momento de la creación de una variable de tipo valor, se reserva una zona de memoria equivalente al tipo. Cada byte de esta zona se inicializa automáticamente con el valor binario 00000000. Nuestra variable tendrá, a continuación, un valor igual a 0 como valor binario.

En el caso de una variable de tipo referencia, el comportamiento es diferente. La zona de memoria asignada a nuestra variable contiene una dirección de memoria en la que se almacena el dato. La dirección de memoria se inicializa con el valor especial `null`, que no apunta a nada, mientras que la zona de memoria que contiene los datos no se ha inicializado. Se inicializará cuando se instancie la variable. Del mismo modo, la dirección de memoria almacenada en nuestra variable se actualizará.

Una variable de tipo referencia podrá, entonces, no contener ningún dato, mientras que una variable de tipo valor tendrá obligatoriamente un valor correspondiente a una secuencia de 0 binaria.

Esta diferencia en el comportamiento tiene una consecuencia importante: la copia de la variable se realiza por valor o por referencia, lo que significa que las variables de tipo valor se copiarán mientras que en las variables de tipo referencia se copiará la dirección que contiene la variable y será posible actuar sobre el dato real desde cualquiera de las variables que apunten sobre el mismo espacio de memoria.

2.2.2 Tipos integrados

El framework .NET incluye miles de tipos diferentes que pueden reutilizarse en los desarrollos. Entre estos tipos tenemos una quincena que podríamos considerar como fundamentales: son los tipos integrados (también llamados tipos primitivos). Son los tipos básicos sobre los que se construyen los demás tipos de la BCL así como los que crea el desarrollador en su propio código. Permiten definir variables que contienen datos muy simples.

Estos tipos tienen la particularidad de que disponen, cada uno, de un alias integrado en C#.

Tipos numéricos

Estos tipos permiten definir variables numéricas enteras o decimales. Cubren rangos de valores distintos y cada uno tiene una precisión específica. Algunos tipos estarán mejor adaptados para cálculos enteros, otros para cálculos en los que la precisión decimal resulte importante, como por ejemplo los cálculos financieros.

A continuación se enumeran los distintos tipos numéricos con sus alias así como los rangos de valores que cubren.

Tipo	Alias C#	Rango de valores cubierto	Tamaño en memoria
`System.Byte`	`byte`	0 a 255	1 byte
`System.SByte`	`sbyte`	-128 a 127	1 byte
`System.Int16`	`short`	-32768 a 32767	2 bytes
`System.UInt16`	`ushort`	0 a 65535	2 bytes
`System.Int32`	`int`	-2147483648 a 2147483647	4 bytes
`System.UInt32`	`uint`	0 a 4294967295	4 bytes
`System.Int64`	`long`	-9223372036854775808 a 9223372036854775807	8 bytes
`System.UInt64`	`ulong`	0 a 18446744073709551615	8 bytes
`System.Single`	`float`	±1,5e-45 a ±3,4e38	4 bytes

© Editions ENI - All rights reserved

Tipo	Alias C#	Rango de valores cubierto	Tamaño en memoria
`System.Double`	`double`	±5,0e-324 a ±1,7e308	8 bytes
`System.Decimal`	`decimal`	±1,0e-28 a ±7,9e28	16 bytes

Los tipos numéricos primitivos son todos de tipo valor. Una variable de tipo numérico y no inicializada por el desarrollador tendrá un valor por defecto igual a 0.

Observación

.NET 4 incluye el tipo `System.Numerics.BigInteger` que permite manipular valores enteros de un tamaño arbitrario. Este tipo es también de tipo valor, pero no forma parte de los tipos integrados.

Los valores numéricos que se utilizan en el código se pueden definir a partir de tres bases numéricas:

– **Decimal** (base 10): es la base numérica que se utiliza por defecto.

  Por ejemplo: 280514

– **Hexadecimal** (base 16): se utiliza mucho en el mundo Web para la definición de colores, y más generalmente para codificar valores numéricos en un formato más comprimido. **Un valor hexadecimal se escribe con el prefijo 0x**.

  Por ejemplo: 0x447C2

– **Binario** (base 2): permite acercarse más al modo en que la máquina interpreta el código compilado. El uso de binarios en C# puede resultar valioso en el ámbito de las optimizaciones (compensaciones de bits en lugar de multiplicaciones por 2, almacenamiento de múltiples datos en una misma variable, etc.). **Los valores binarios se escriben con el prefijo 0b**.

  Por ejemplo: 0b01000100011111000010

La lectura de valores numéricos puede ser difícil en algunos casos, sobre todo en los que se representan en formato binario. El equipo encargado del desarrollo de C# ha integrado, en la séptima versión del lenguaje, la posibilidad de agregar separadores en los valores numéricos para mejorar la legibilidad global. El carácter utilizado para ello es:

- 280_514
- 0x04_47_C2
- 0b0100_0100_0111_1100_0010

Tipos textuales

Existen, en la BCL, dos tipos que permiten manipular caracteres Unicode y cadenas de caracteres Unicode: `System.Char` y `System.String`. Estos tipos tienen, respectivamente, los alias `char` y `string`.

El tipo `char` es un tipo valor que encapsula los mecanismos necesarios para procesar un carácter Unicode. Por ello, una variable de tipo `char` se almacena en memoria utilizando dos bytes.

Los valores de tipo `char` deben delimitarse por los caracteres `' '`: `'a'`.

Algunos caracteres tienen un significado particular para el lenguaje y deben utilizarse, obligatoriamente, con el carácter de escape \ para que se interpreten correctamente. Existen otros caracteres que no tienen representación gráfica y deben declararse utilizando secuencias específicas. La siguiente tabla resume las secuencias que pueden utilizarse.

Secuencia de escape	Carácter asociado
\'	Comilla simple '
\"	Comilla doble "
\\	Backslash \
\a	Alerta sonora
\b	Marcha atrás
\f	Salto de página
\n	Salto de línea

© Editions ENI - All rights reserved

Ediciones ENI

C# 7

Desarrolle aplicaciones Windows con Visual Studio 2015

Colección
Expert IT

Extracto del Libro

Capítulo 11
Creación de controles de usuario

1. Introducción

El desarrollo de aplicaciones se basa principalmente en los controles, que proporcionan las distintas funcionalidades en forma visual y permiten al usuario interactuar con ellos. Todos estos controles provienen, de una manera más o menos directa, de la clase base `System.Windows.Forms.Control`. Visual Studio ofrece la integración de controles de terceros, añadiéndolos a la caja de herramientas. Pero si la necesidad es muy específica, es posible crear sus propios controles.

La clase base de los controles, `Control`, proporciona las funcionalidades básicas que son necesarias, fundamentalmente para la entrada de datos del usuario a través del teclado y el ratón. Esto implica propiedades, métodos y eventos comunes a todos los controles. Sin embargo, esta clase base no proporciona la lógica de visualización del control.

Existen tres modos de creación del control:

– Los controles personalizados.

– La herencia de controles.

– Los controles de usuario.

La creación de controles sigue el principio de reutilización del código. La lógica se crea en un único sitio y se puede usar varias veces. La ventaja es muy importante en términos de mantenimiento de la aplicación, ya que para cambiar el comportamiento de este control, solo será necesario modificar un archivo.

2. Los controles personalizados

Estos controles ofrecen las mayores posibilidades de personalización, tanto a nivel gráfico como lógico. Un control personalizado hereda directamente de la clase `Control`. Por lo tanto, es necesario escribir toda la lógica de visualización lo que, según el resultado esperado, puede resultar una fase muy larga y complicada. El desarrollador también debe definir los métodos, propiedades y eventos.

La clase base `Control` expone el evento `Paint`. Es el que se produce cuando se genera el control e implica la ejecución del controlador del evento por defecto `OnPaint`. Este método recibe un único argumento de tipo `PaintEventArgs`, que contiene la información necesaria en la superficie de dibujo del control. El tipo `PaintEventArgs` tiene dos propiedades, `Graphics`, del tipo `System.Drawing.Graphics` y `ClipRectangle`, del tipo `System.Drawing.Rectangle`. Para añadir la lógica de diseño al control hay que sobrecargar el método `OnPaint` y añadirle el código de diseño:

```
protected override void OnPaint
                        (System.Windows.Forms.PaintEventArgs e)
{
    // Código de diseño del control
}
```

La propiedad `Graphics` del objeto `PaintEventArgs` representa la superficie del control, mientras que la propiedad `ClipRectangle` representa su área de dibujo. Durante la primera representación del control, la propiedad `ClipRectangle` representa los límites del control. Estos límites se pueden modificar, por ejemplo, si un control por debajo oculta una parte, de tal manera que sea necesario volver a dibujar el control. La parte `ClipRectangle` representara el área a modificar.

© Editions ENI - All rights reserved

Cree una carpeta **Controles** en la raíz del proyecto y añada una nueva clase llamada **CustomControl** definida de la siguiente manera:

```
using System.Drawing;

namespace SelfMailer.Controls
{
    public class CustomControl: System.Windows.Forms.Control
    {
        protected override void OnPaint
                        (System.Windows.Forms.PaintEventArgs e)
        {
            Rectangle R = new Rectangle(0, 0,
                                  this.Size.Width, this.Size.Height);
            e.Graphics.FillRectangle(Brushes.Green, R);
        }
    }
}
```

En el constructor del formulario **MailServerSettings**, añada el código de instanciación del control personalizado:

```
Controls.CustomControl C = new Controls.CustomControl();
C.Localization = new System.Drawing.Point(0, 0);
C.Size = this.Size;
this.Controls.Add(C);
```

Este código instancia un nuevo control de tipo `CustomControl`, lo ubica en la parte superior izquierda y ajusta su tamaño al del formulario. Para terminar, el control se añade a la colección de controles del formulario.

Ejecute la aplicación ([F5]) y abra el formulario de los argumentos del servidor de mail para ver que el control, que representa un sencillo rectángulo verde, rellena el formulario con un color de fondo.

Observación

Las posibilidades de dibujo con GDI+ se abordarán más adelante en este libro, en la sección El diseño con GDI+, del capítulo Para llegar más lejos.

Para terminar, basta con añadir los miembros necesarios para la lógica del control.

3. La herencia de controles

Si el objetivo es extender las funcionalidades de un control existente, ya sea un control del Framework .NET o de un editor de terceros, la manera más rápida es heredar de este control. El nuevo control tiene de esta manera todos los miembros y la representación visual de su clase padre. Solo hay que añadir la lógica de procesamiento. De la misma manera que los controles personalizados, es posible sobrecargar el método `OnPaint` para modificar el aspecto visual del control.

Si una aplicación tiene varios formularios que necesitan un email como campo de entrada, es mejor crear un control heredando de la clase `TextBox`, implementar la lógica de validación y después añadir este control a los formularios de manera que no sea necesario repetir el código de validación de cada uno de ellos.

La creación de un control heredado se hace de la misma manera que un control personalizado, creando una clase que va a heredar del control que tiene el comportamiento base requerido. Cree la clase `EmailTextBox` en la carpeta **Controles** y herede de la clase `TextBox`:

```
public class EmailTextBox: System.Windows.Forms.TextBox
{
}
```

Añada una sobrecarga del método `OnValidating` para realizar las comprobaciones de formato y añada el código del método `FromEmail_Validating` del formulario **MailServerSettings**:

```
protected override void
          OnValidating(System.ComponentModel.CancelEventArgs e)
{
    base.OnValidating(e);
    string pattern = @"^([a-zA-Z0-9_\-\.]+)@((\[[0-9]{1,3}\." +
                     @"[0-9]{1,3}\.[0-9]{1,3}\.)|" +
                     @"(([a-zA-Z0-9\-]+\.)+))" +
                     @"([a-zA-Z]{2,4}|[0-9]{1,3})(\]?)$";
    Regex reg = new Regex(pattern);
    if (!reg.IsMatch(this.Text))
    {
         this.BackColor = Color.Bisque;
         e.Cancel = true;
    }
```

© Editions ENI - All rights reserved

```
    else
        this.BackColor = this.PreviousBackColor;
}
```

Es preciso hacer cambios puesto que ya no se accede a la propiedad `Text` del control a partir del formulario. Por tanto, hay que sustituir:

```
this.FromEmail.Text
```

por un acceso directo:

```
this.Text
```

La primera instrucción:

```
base.OnValidating(e);
```

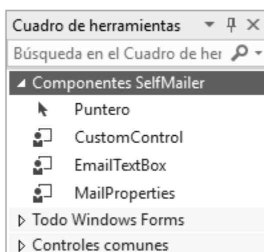
permite llamar al método de validación de la clase base. De esta manera, existe una primera validación en la clase base y después hay una validación del control por parte del código adicional.

El último aspecto destacable es que el componente **ErrorProvider** ya no está al mismo nivel. Para indicar al usuario que el campo es inválido, en lugar de mostrar un icono, se modifica el color de fondo del control. La propiedad `PreviousBackColor`, que ha sido en principio inicializada con el color de fondo de base en el constructor, permite conservarlo para reasignarlo al control en caso de que la validación tenga éxito:

```
protected Color PreviousBackColor { get; set; }

public EmailTextBox()
{
    this.PreviousBackColor = this.BackColor;
}
```

El controlador del evento `FromEmail_Validating` es, en este caso, irrelevante, ya que la validación se realiza dentro del control. Además, puede eliminar el control de tipo `TextBox` para la entrada de datos del email de la persona que envía el correo electrónico y sustituirlo por un control de tipo `EmailTextBox`, que Visual Studio ha añadido en la caja de herramientas dentro del grupo **Componentes SelfMailer** (**SelfMailer** representa el nombre del proyecto).

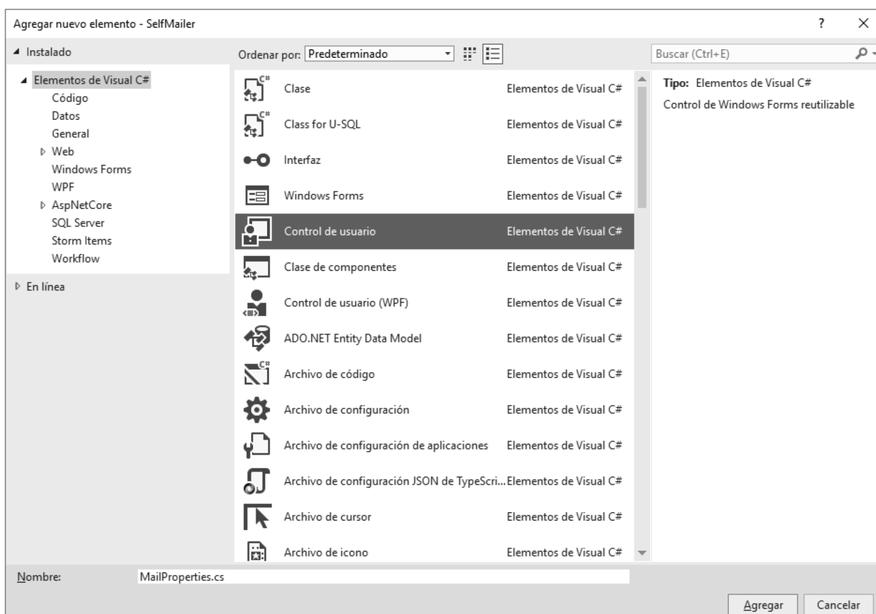


Ejecute la aplicación ([F5]) para probar la funcionalidad.

4. Los controles de usuario

El objetivo de un control de usuario es agrupar de manera lógica los controles para obtener una entidad reutilizable. La creación se hace añadiendo al proyecto un **Control de usuario** desde la ventana de adición de un nuevo elemento.

Añada un control de usuario llamado **MailProperties** en la carpeta **Elementos de Visual C#** del proyecto:



© Editions ENI - All rights reserved