

# Utilizar los formularios

Los formularios representan la interacción del usuario con una aplicación. Se utilizan tanto para la presentación como para la introducción de datos. El diseño de la interfaz de usuario es una etapa importante, ya que una interfaz visualmente coherente y lógica, y por tanto usable, es más sencilla de utilizar.

## 1. Añadir formularios al proyecto

Durante la creación del proyecto de aplicación Windows se crea un formulario por defecto, **Form1**. Este formulario representa la clase que se instanciará durante la ejecución. Esta clase parcial está compuesta por dos archivos **Form1.designer.cs** y **Form1.cs**:

- **Form1.designer.cs**: los archivos de formularios con extensión **.designer.cs** se generan automáticamente en el diseñador de pantallas. Cuando se ubica un control en el formulario, Visual Studio inserta el código de inicialización y los parámetros por defecto en este archivo. El diseñador de pantallas es, en este caso, un mecanismo para crear código de manera visual.
- **Form1.cs**: este archivo permite escribir la lógica del formulario, los tratamientos de las acciones de usuario y todo el código necesario para el correcto funcionamiento lógico del formulario.

Antes de crear un nuevo formulario vamos a crear una nueva carpeta llamada **Forms** en la raíz del proyecto para almacenar en ella todos los formularios. Separar los elementos dentro de un proyecto permite tener una visualización de conjunto mejor y encontrar con mayor facilidad el elemento deseado, sobre todo en caso de mantenimiento.

## 2. Modificar el formulario de inicio

El formulario de inicio es el que se abre por defecto cuando se ejecuta la aplicación. Se trata de la clase que se instancia en el archivo **Program.cs**.

Para modificar el formulario de inicio, que actualmente es **Form1**, hay que empezar creando uno nuevo en la carpeta **Forms** y llamarlo **Main.cs**. En el archivo **Program.cs**, basta con sustituir la instrucción:

```
Application.Run(new Form1());
```

por

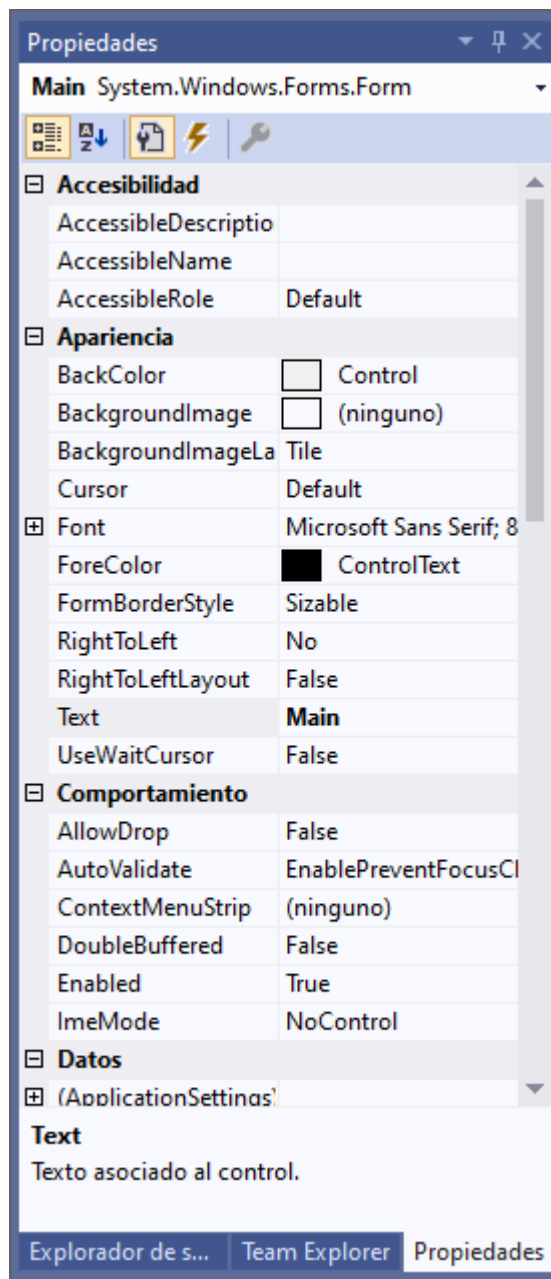
```
Application.Run(new Forms.Main());
```

Como el formulario **Form1** ya no es necesario, se puede eliminar.

Ejecutando la depuración ([F5]) comprobamos que se muestra el formulario **Main**.

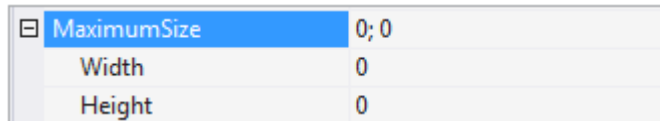
### 3. Las propiedades de los formularios

En modo diseñador de pantallas, Visual Studio permite definir las propiedades de los objetos gracias a la ventana **Propiedades** ([Alt][Entrar]).



Las propiedades se clasifican en grupos lógicos. Cada una de ellas afecta al aspecto o al comportamiento del formulario. Algunas propiedades, como la propiedad **MaximumSize**, son

estructurales. Haciendo clic en la flecha que hay al lado, es posible modificar cada propiedad:



Las propiedades también se pueden modificar a nivel de código:

```
this.Text = "Argumentos del servidor mail";
```

Esta línea, ubicada en el constructor del formulario, tendrá el mismo efecto que modificar la propiedad en la ventana de propiedades del diseñador de pantallas, con la diferencia de que Visual Studio habrá situado esta instrucción en el archivo designer. En el archivo con extensión .designer.cs, se transcriben todas las modificaciones realizadas en el constructor de formularios en código por Visual Studio.

La asignación de las propiedades sigue la misma sintaxis que para otros miembros de la clase. El operador de asignación (=) se utiliza para asignar un valor a la propiedad referenciada con su nombre.

Asigne las siguientes propiedades al formulario:

Propiedades	Valor	Efecto
FormBorderStyle	FixedToolWindow	El formulario ya no se redimensiona. Los botones para agrandar o reducir la ventana y el icono ya no se ven más.
ShowInTaskbar	False	Cuando el formulario se abre, no se ve ninguna pestaña en la barra de tareas de Windows.
Size	355; 205	Fija el tamaño del formulario.
StartPosition	CenterParent	El formulario se muestra centrado en relación a su padre.
Text	Parámetros del servidor mail	Se modifica el texto en la barra del título del formulario.

Con la asignación de los valores anteriores en el diseñador de pantallas, Visual Studio completa el archivo designer con las líneas siguientes:

```
this.ClientSize = new System.Drawing.Size(355, 205);
this.FormBorderStyle =
    System.Windows.Forms.FormBorderStyle.FixedToolWindow;
this.ShowInTaskbar = false;
this.StartPosition =
    System.Windows.Forms.FormStartPosition.CenterParent;
this.Text = "Argumentos del servidor mail";
```

Todo lo que se hace en el diseñador de pantallas, repercute en el código del archivo designer.

## 4. Los métodos de los formularios

Los formularios derivan de la clase base `System.Windows.Forms.Form` y heredan varios métodos que permiten gestionar la visualización de los formularios.

Una vez instanciado, el objetivo de un formulario es permitir al usuario interactuar con él. Los métodos `Show` y `ShowDialog` de un formulario hacen que el formulario se pueda ver. El método `Show` muestra el formulario en la pantalla y le asigna el foco. La propiedad `Visible` del formulario toma automáticamente el valor `true`. El método `Show` no crea el formulario. Si existe en memoria pero no es visible, es decir si su propiedad `Visible` es `false`, la llamada del método cambia este valor a `true`. El método `ShowDialog` es idéntico, excepto que el formulario es modal, es decir, se debe cerrar antes de que el foco se pueda asignar a otro formulario de la aplicación. Este método obliga al usuario a realizar una acción.

Después de que el usuario ha realizado las acciones que desee en el formulario, éste se debe cerrar. Existen dos maneras de cerrar un formulario. La primera consiste en ocultarlo con el método `Hide`. Esto equivale a asignar el valor `false` a la propiedad `Visible` del formulario. Con este método, el formulario siempre está en memoria y la llamada del método `Show` lo vuelve visible. La segunda manera de cerrar un formulario es con el método `Close`, que elimina la instancia en memoria y libera los recursos. Es imposible llamar al método `Show` de un formulario después llamar al método `Close`, pues el formulario ya no existe y se debe instanciar de nuevo. La llamada al método `Close` para el formulario de inicio implica cerrar la aplicación.

Estos métodos se usan durante la implementación de administradores de eventos.

## 5. Los eventos de los formularios

Un formulario, como cualquier objeto, tiene un ciclo de vida y momentos clave que desencadenan eventos específicos.

El evento `Load` corresponde a la creación del formulario en memoria. Se produce cuando se llama por primera vez al método `Show`, o `ShowDialog`. Si el formulario ya está en memoria, la llamada del método `Show` no desencadenará una segunda vez el evento `Load`. Existe un controlador destinado al evento `Load` que permite inicializar las variables del formulario a partir de una base de datos o de un archivo, por ejemplo.

Los eventos Activated y Deactivate se desencadenan cuando el formulario recibe o pierde el foco. Son eventos que se pueden desencadenar varias veces durante la vida del formulario. Estos eventos solo se desencadenan si el foco del formulario se modifica dentro de la aplicación, esto significa que si el usuario pasa el foco a otra aplicación y después le devuelve el foco, no se desencadenará ninguno de los dos eventos.

El proceso de cierre de un formulario implica que se desencadenen dos eventos distintos: FormClosing y FormClosed.

FormClosing es el evento que se produce cuando se cierra el formulario, se desencadena con la llamada al método Close o cuando el usuario pulsa el botón de cierre del formulario. Se asigna un controlador al evento FormClosing que permite comprobar y validar los datos del usuario. Si fuera necesario es posible anular el cierre del formulario y conservarlo abierto, utilizando la propiedad Cancel del objeto FormClosingEventArgs que se pasa como parámetro al controlador del evento:

```
private void FormClosing(object sender, FormClosingEventArgs e)
{
    e.Cancel = true;
}
```

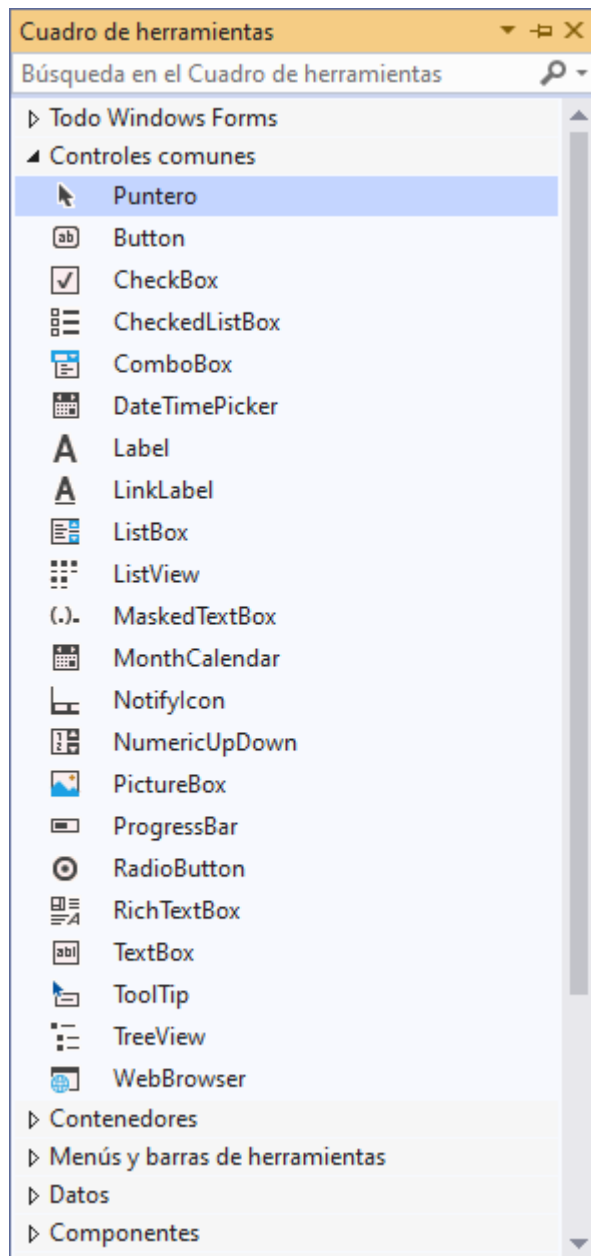
Cuando se cierra el formulario se desencadena el evento FormClosed, después FormClosing y, después la ejecución de los controladores de eventos, FormClosing.

Estos eventos se usan durante la implementación de los controladores de eventos.

# Usar los controles

## 1. Los tipos de controles

El Framework .NET ofrece una serie de controles y Visual Studio los integra en el **Cuadro de herramientas** ([Ctrl][Alt] X) del diseñador de pantallas.



La caja de herramientas se subdivide en grupos de controles: los controles comunes, los contenedores, los menús e incluso los datos, entre otros.

Cada grupo tiene varios tipos de controles: esto sirve principalmente al usuario para ejecutar una acción como los **Button**, permitir introducir datos como los **TextBox** o dar información al usuario como los **Label**.

Otros tipos de controles, llamados componentes, tienen la particularidad de no ser visibles en la interfaz del usuario, como el control **Timer**. Los controles de este tipo se muestran en la barra de componentes del formulario, en la parte inferior del diseñador de pantallas:

Los extenders son un tipo de componente particular cuyo objetivo es extender las funcionalidades de los controles. Cuando se añade un extender a un formulario, todos los controles muestran una o varias propiedades nuevas. El control **ErrorProvider** es un ejemplo.

## 2. Añadir controles a los formularios

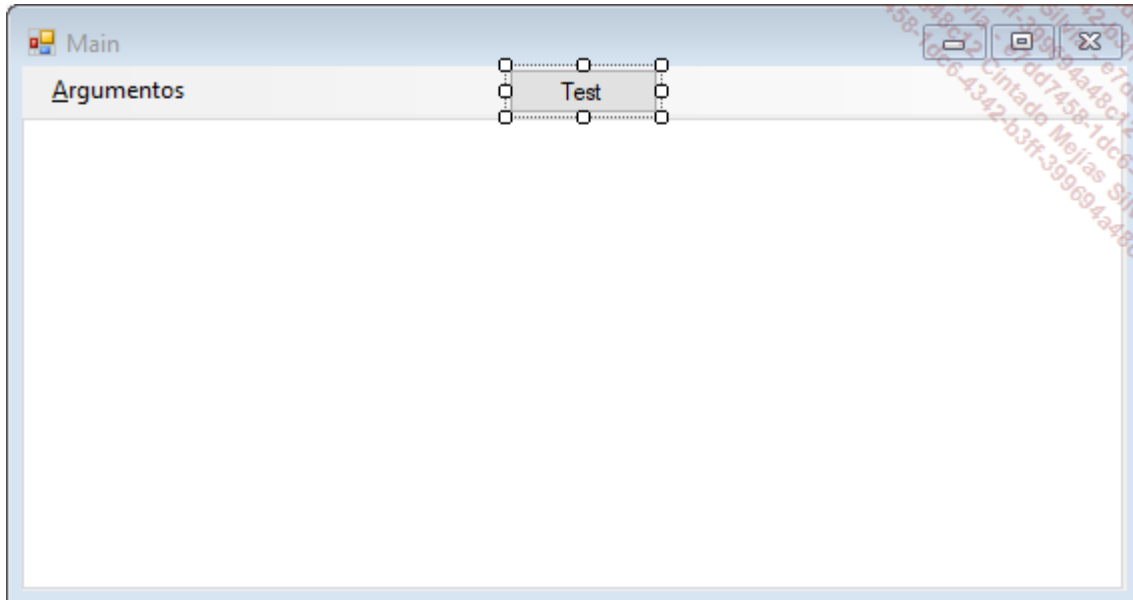
Para añadir un control al formulario, es suficiente con arrastrarlo desde la caja de herramientas y ubicarlo en el formulario. El diseñador de pantallas se encarga de escribir el código para instanciar el control, situarlo e inicializar sus variables.

Añada los controles al formulario para obtener lo siguiente:

También es posible crear un control desde el código, instanciando el objeto y definiendo sus propiedades. Después se añade a la colección de controles del formulario. El código siguiente crea un control de tipo **Button** y lo añade al formulario actual:

```
Button newBtn = new Button();
```

```
newBtn.Name = "btnTest";  
newBtn.Text = "Test";  
newBtn.Location = new System.Drawing.Point(25, 25);  
newBtn.Size = new System.Drawing.Size(100, 25);  
this.Controls.Add(newBtn);
```



Cualquier contenedor, incluso los formularios, tiene una colección de controles hijos accesibles mediante la propiedad `Controls`. Utilizando esta colección, es posible añadir controles, modificarlos o eliminarlos de manera dinámica durante la ejecución, como se muestra en el ejemplo anterior.

### 3. Las propiedades de los controles

Al igual que los formularios, los controles tienen propiedades que permiten definirlos visualmente. Basta con seleccionar un control en el formulario para que la ventana muestre las propiedades del dicho control.

Es posible modificar las propiedades de varios controles al mismo tiempo arrastrando el ratón alrededor de los mismos o manteniendo pulsada la tecla `[Ctrl]` y pulsando en los diferentes controles. La ventana de propiedades muestra solo aquellas propiedades comunes a todos los controles seleccionados.

Observe que la propiedad `UseSystemPasswordChar` de la clase `TextBox` transforma un campo clásico de entrada de valores en un campo de contraseña usando los parámetros de sistema para sustituir los caracteres:

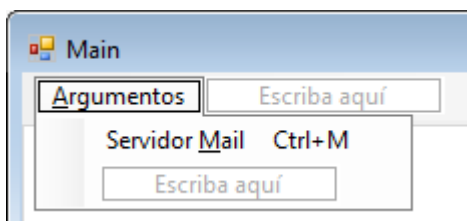


La propiedad PasswordChar tiene el mismo efecto, en el sentido de que el control **TextBox** se convierte en un campo de entrada de valores de tipo contraseña pero, en lugar de utilizar el carácter por defecto del sistema sobre el que se ejecuta la aplicación para sustituir los caracteres reales, es posible especificar este carácter. La propiedad UseSystemPasswordChar es prioritaria sobre PasswordChar, por lo que es necesario que UseSystemPasswordChar tenga valor false para que la propiedad PasswordChar se pueda tener en cuenta.

## 4. Los menús

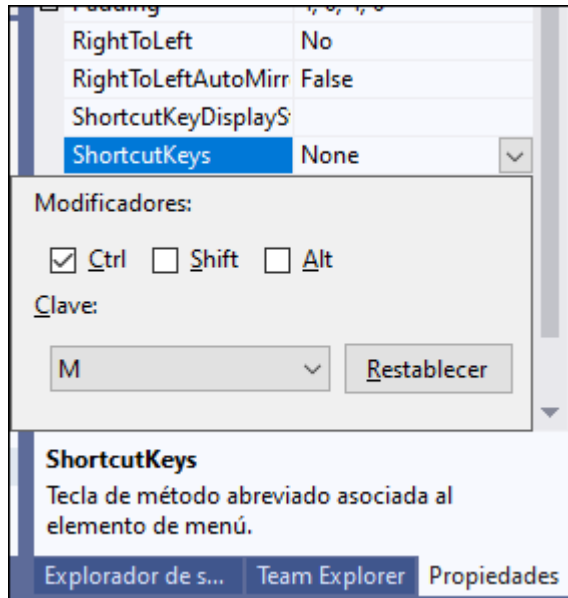
Cualquier aplicación está formada por varios formularios y tiene que implementar una interfaz de usuario que permita navegar entre ellos. Los menús representan la manera más sencilla para organizar y permitir al usuario acceder a las funcionalidades de la aplicación.

La creación de un menú se hace con el control **MenuStrip**. Éste contiene una colección de controles **ToolStripMenuItem**. Para ello se añade un control **MenuStrip** al formulario **Main**, que se acopla, por defecto, en la parte superior del formulario y ocupa todo el ancho del mismo. La propiedad Dock del control permite obtener este comportamiento. Para añadir elementos de menú en el diseñador de pantallas, basta con pulsar en el mensaje **Escriba aquí** e introducir el texto del menú:



El carácter & antes de un carácter del título del menú permite especificar la tecla de acceso rápido que el usuario podrá combinar con la tecla [Alt] para acceder al menú. Añadiendo el elemento de menú que tenga el texto **&Parámetros** al formulario **Main**, durante de la ejecución de la aplicación, el usuario podrá abrir este menú pulsando la combinación de teclas [Alt] **P**. De manera visual se reconoce este carácter, ya que se subraya.

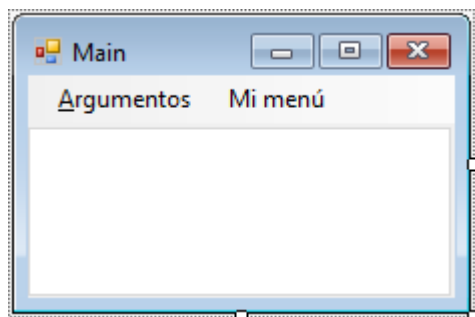
Las teclas de acceso rápido permiten acceder directamente a los menús sin tener que pasar por el menú padre, como sucede por ejemplo con la tecla de acceso rápido. Es posible asignar una combinación de teclas con la propiedad `ShortcutKeys` del elemento de menú. La ventana de propiedades muestra un cuadro de diálogo intuitivo que permiten definir la combinación del acceso rápido:



Añada un elemento de menú, llamado **Servidor &Mail** al menú **Parámetros** y asígnele la combinación [Ctrl] **M** como tecla de acceso rápido. El diseñador de pantallas muestra la combinación a la derecha del elemento de menú. Es posible hacer que desaparezca este texto, asignando el valor `false` a la propiedad `ShowShortcutKeys`. También se puede modificar, insertando el texto a mostrar en la propiedad `ShortcutKeyDisplayString` del elemento de menú.

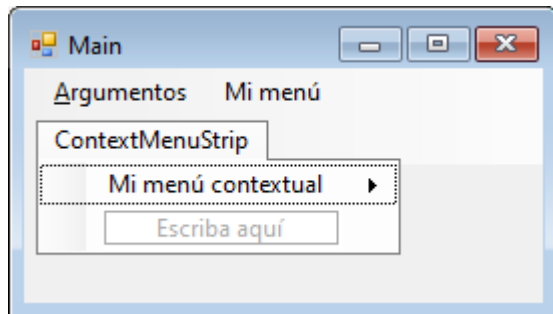
También es posible añadir los menús de manera dinámica durante la ejecución para aportar una interacción adicional al usuario. Basta con instanciar un objeto del tipo `ToolStripMenuItem` y añadirla a la colección `Items` del menú padre:

```
ToolStripMenuItem newMenu = new ToolStripMenuItem("Mi menú ");
this.MainMenu.Items.Add(newMenu);
```



Existe otro tipo de menú llamado menú contextual (**ContextMenuStrip** en la caja de herramientas). El usuario puede ver estos menús cuando pulsa con el botón derecho del ratón sobre un elemento de la aplicación. Los controles tienen la propiedad

ContextMenuStrip, que permite definir el menú contextual que se va a mostrar. Además de esta diferencia, los menús contextuales funcionan de la misma manera que los menús clásicos, tanto a nivel del diseñador de pantallas como de código. Es posible activar las teclas de acceso directo pero no las teclas de acceso rápido.



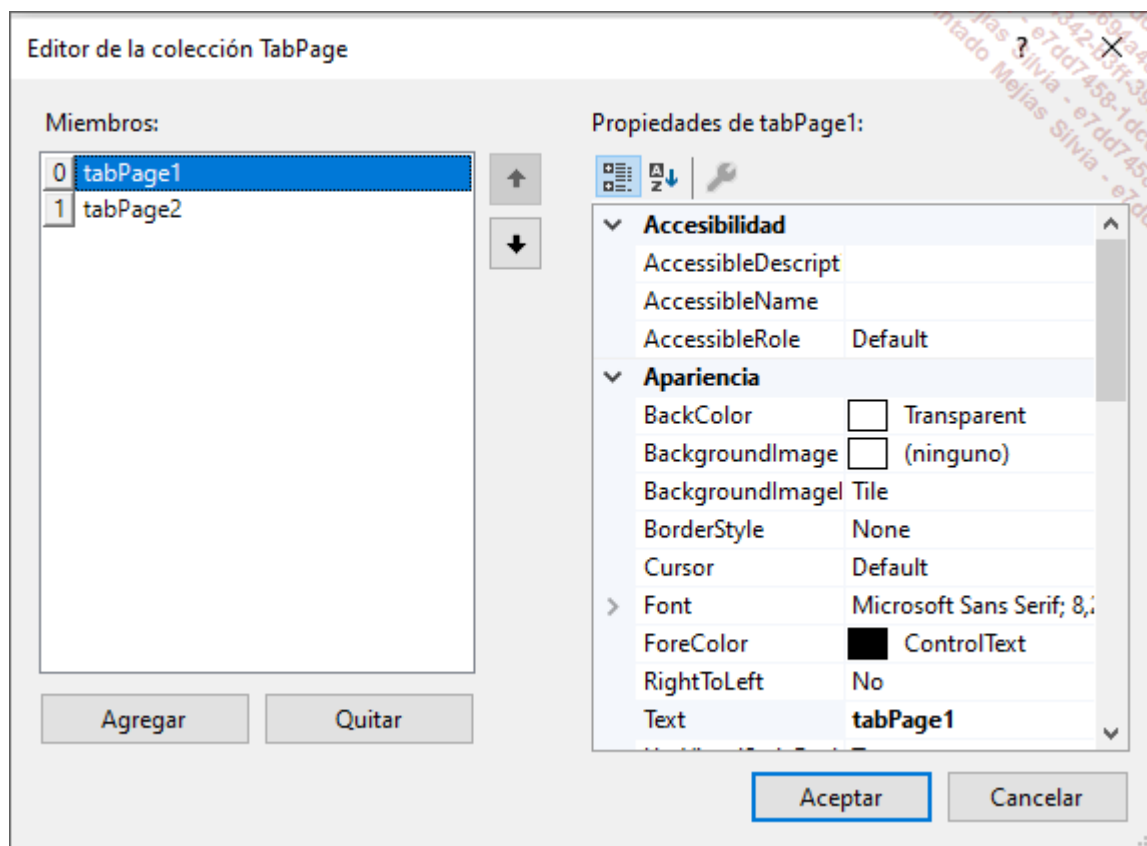
## 5. Los contenedores

La funcionalidad de algunos controles, igual que sucede con los formularios, es agrupar otros controles. Se llaman contenedores. El hecho de modificar las propiedades de un contenedor puede implicar la modificación de los controles que contiene. Por ejemplo, si la propiedad Enabled de un control **Panel** se define a false, todos los controles en el contenedor **Panel** se desactivan.

El control **TabControl** es un contenedor cuya función es agrupar otros controles dentro de pestañas en la colección **TabPage**. Un **TabPage** es un contenedor que se parece a un control **Panel**. Es posible activar las barras de desplazamiento con la propiedad AutoScroll.

Para añadir controles a un **TabPage** basta con seleccionar la pestaña y arrastrar los controles desde la caja de herramientas.

Añada un control **TabControl** en el formulario **Main** y asigne a la propiedad Dock el valor Fill. El control se extiende para ocupar todo el espacio disponible en el formulario. Pulsando en el botón ... de la propiedad **TabPage**, el diseñador de pantallas muestra un formulario que permite gestionar los objetos **TabPage**.



Para la aplicación de demostración, los **TabPage** se añaden al control de forma dinámica en el código. Por tanto, puede eliminar los dos **TabPage** por defecto.

## 6. La usabilidad

La usabilidad es un punto muy importante de un formulario. La información se debe presentar de manera clara e intuitiva. Visual Studio dispone de herramientas que permiten mejorar la experiencia de usuario.

El orden de tabulación representa la secuencia de controles que van a recibir el foco, pulsando la tecla [Tab]. Para definir el orden hay que definir la propiedad `TabIndex` de cada control, empleando por 0. Para impedir que el control reciba el foco con la tabulación hay que asignar el valor `false` a la propiedad `TabStop`. Cuantos más controles tenga el formulario, más complicado será definir la propiedad `TabIndex` de cada uno de ellos.

Por este motivo, la herramienta **Orden de tabulación** en el menú **Ver** es muy útil. Activando los controles del formulario se muestran los números que representan el valor de su propiedad `TabIndex`:

Argumentos del servidor de mail

Nombre Exp.: 0

Mail Exp.: 1

Host: 2

Usuario: 3

Contraseña: 4

5 Validar 6 Cancelar

Para definir un orden diferente al orden por defecto, basta con pulsar en los controles, unos a continuación de otros, para obtener una secuencia diferente. De manera predeterminada, la propiedad TabIndex se define en el orden en que se añaden los controles en el formulario.

Un formulario que normalmente es un formulario de parametrización con un botón de validación y un botón de anulación y que se abrirá con el método ShowDialog, debe utilizar las propiedades AcceptButton y CancelButton del formulario.

Estas propiedades aceptan como valor el nombre de un botón del formulario y cuando se pulsa sobre este botón el formulario devuelve respectivamente los valores DialogResult.OK o DialogResult.Cancel, que se podrán recuperar en el formulario padre para realizar acciones específicas:

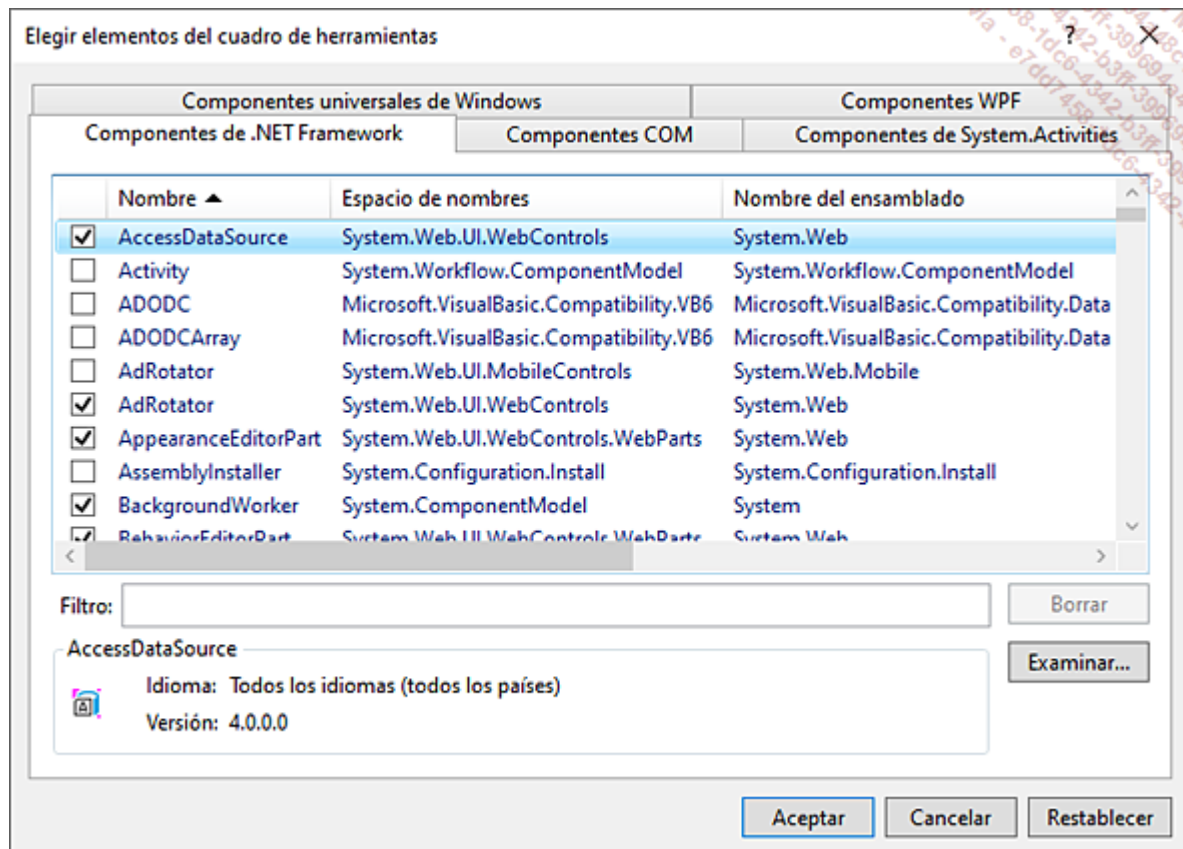
```
if (new Form1()..ShowDialog() == DialogResult.OK)
{
}
```

## 7. Añadir controles a la caja de herramientas

La caja de herramientas no es estática, es posible añadir nuevos controles desarrollados por terceros o en el ámbito de otra solución.

Para añadir un control a la caja de herramientas, haga clic con el botón derecho del ratón sobre el grupo al que desea añadirlo y despliegue el menú contextual.

Seleccione **Elegir los elementos...** y Visual Studio abrirá una ventana de selección:



Si no encontrara el control en su sistema, pulsando el botón **Examinar...** puede seleccionar una librería o un ejecutable que contenga el control a añadir. A continuación basta con marcar las opciones de los controles a añadir y validar el formulario.

Los controles creados en un proyecto se añaden automáticamente en la caja de herramientas, en una pestaña que se llama igual que el proyecto.

# Implementación del administrador de eventos

## Introducción

Los eventos ocurren durante la ejecución de la aplicación. Cada control, incluidos los formularios, pueden desencadenar varios eventos como respuesta a una acción del usuario. Por ejemplo, cuando se pulsa sobre un control de tipo Button, se desencadena un evento Click. Si los métodos llamados controladores de eventos están suscritos a este evento, se ejecutan.

Cada control tiene un evento por defecto. Haciendo doble clic sobre el control, Visual Studio crea el controlador del evento asociado. Para la mayor parte de los controles, se trata del evento Click.

## La creación de controladores de eventos

Haga doble clic en el elemento de menú **Servidor Mail** del formulario **Main**. Visual Studio abre el archivo de código asociado con la implementación base del controlador de eventos:


```
private void MailServerMenu_Click(object sender, EventArgs e)
{
}
```

El método se marca con la palabra reservada `private`. Como primer argumento admite un objeto de tipo `object`, que representa el objeto que desencadena el evento y, como segundo argumento, un objeto de tipo `EventArgs`, que contiene información adicional del evento.

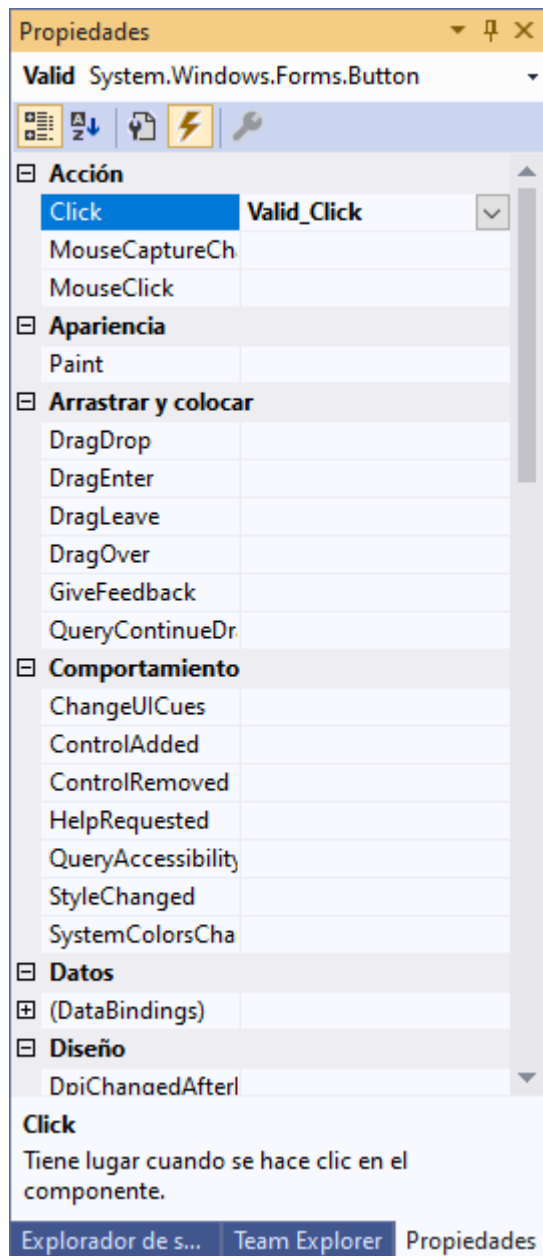
Añada el siguiente código al controlador de eventos:

```
new Forms1().ShowDialog();
```

Esta instrucción crea una nueva instancia del objeto y la muestra de manera modal. Para probar el controlador de eventos basta con ejecutar la aplicación ([F5]) y pulsar en el menú **Servidor Mail**. Si utiliza el acceso rápido de teclado o la tecla de acceso directo que se ha definido, el evento también se desencadena.

Los controles tienen muchos eventos. Es posible acceder a ellos desde la ventana de propiedades, pulsando en el icono **Eventos** .

Basta con pulsar dos veces en el evento que desee, para que Visual Studio genere el código asociado. Cree un controlador de eventos para el botón **Validar** del formulario **MailServer-Settings**.



Visual Studio genera un método privado:

```
private void Valid_Click(object sender, EventArgs e)
{
}
```

## 1. La mecánica de un evento

Un controlador de eventos es un método que se invoca a través de un delegado. Un delegado es un puntero hacia un método. Permite pasar la referencia del punto de entrada de este método para que se pueda invocar de manera implícita.

Para asociar un método a un evento hay que crear una nueva instancia del delegado que apunte al método y asociarla al evento.



Cuando se desencadene el evento, se invocará a los delegados suscritos, lo que equivale a una llamada al método hacia el que apunta el delegado.

La creación de los dos controladores de eventos anteriores no solo tiene como efecto crear métodos privados. El diseñador de pantallas también ha añadido las instrucciones que permiten suscribir esos métodos a los eventos en el método `InitializeComponent` de los archivos **.designer.cs** de los formularios correspondientes:

- Se añade la siguiente instrucción al archivo **Main.designer.cs**:

```
this.MailServerMenu.Click +=  
    new System.EventHandler(this.MailServerMenu_Click);
```

- Se añade la siguiente instrucción al archivo **MailServerSettings.designer.cs**:

```
this.Valid.Click += new System.EventHandler(this.Valid_Click);
```

## 2. La adición dinámica de un controlador de eventos

La declaración de los eventos se puede hacer de manera dinámica en tiempo de ejecución. Cuando se crea un evento en el diseñador de pantallas, Visual Studio lo inserta en el archivo designer del formulario. Respecto a la inicialización de los controles, se inserta una línea de código que asocia el evento con el delegado que apunta al método que tendrá la función de controlador de eventos.

El operador `+=` permite realizar la asociación de un controlador a un evento. Este método se ha usado anteriormente en el constructor de la clase `Project`:

```
public Project()  
{  
    this.ProjectSettings = new ProjectSettings();  
    this.ProjectSettings.Changed +=  
        new EventHandler<ChangedEventArgs>(ChildChanged);  
    this.MailServerSettings = new MailServerSettings();  
    this.MailServerSettings.Changed +=  
        new EventHandler<ChangedEventArgs>(ChildChanged);  
}
```

Añadir un controlador a un evento es sintácticamente idéntico a añadir un método destino a un delegado.

### 3. La eliminación dinámica de un controlador de eventos

Igual que con la adición, es posible eliminar la asociación de un método a un evento en tiempo de ejecución. El código que permite eliminar esta asociación se parece al que sirve para añadirla. Esto se hace con el operador -=. La referencia a un delegado apuntando hacia el método a eliminar del controlador es:

```
this.Valid.Click -= new EventHandler(this.Valid_Click);
```

Eliminar un controlador de un evento es sintácticamente idéntico a eliminar un método destino de un delegado.

La eliminación de una asociación que no existe no implica la generación de un error.

## Los controladores de eventos avanzados

### 1. Un controlador para varios eventos

Un controlador se puede suscribir a varios eventos diferentes. Es útil cuando varios objetos o controles deben desencadenar la misma acción y esto evita tener que crear varios controladores idénticos.

La asociación se hace, normalmente, asociando al evento un delegado que apunta hacia el método, salvo que el delegado sea idéntico para los dos eventos:

```
this.Valid.Click += new EventHandler(this.Valid_Click);  
this.Cancel.Click += new EventHandler(this.Valid_Click);
```

Puede empezar declarando e instanciando el delegado y después asociarlo a varios eventos:

```
EventHandler newHandler = new EventHandler(this.Valid_Click);  
this.Valid.Click += newHandler;  
this.Cancel.Click += newHandler;
```

Cuando varios controles desencadenan el mismo evento, es posible distinguir las acciones que se deben realizar utilizando el argumento sender que se pasa al controlador de eventos:

```
private void Valid_Click(object sender, EventArgs e)  
{  
    MessageBox.Show((sender as Control).Name);  
}
```

## 2. Varios controladores para un evento

De manera inversa, es posible asociar varios controladores para un mismo evento. La asociación se hace de manera habitual con el operador +=:

```
this.Valid.Click += new System.EventHandler(this.Valid_Click1);  
this.Valid.Click += new System.EventHandler(this.Valid_Click1);  
this.Valid.Click += new System.EventHandler(this.Valid_Click2);
```

El orden de llamada de los controladores cuando el evento se desencadena es la misma que el orden de asociación. De esta manera, cuando un evento se asocia con los controladores A, B y C en este orden, el desencadenamiento del evento implicará la llamada de los controladores en este mismo orden A, después B y, por último, C.

En el código anterior, hay una asociación de tres controladores para el mismo evento. Los dos primeros delegados son idénticos. Apuntan hacia el método Valid\_Click1. Cuando el evento Click se desencadena, se llama el método Valid\_Click1 sucesivamente dos veces y después el método Valid\_Click2. Si el método Valid\_Click2 elimina la asociación entre el evento y el delegado que está duplicada, como se muestra a continuación:

```
private void Valid_Click2(object sender, EventArgs e)  
{  
    this.Valid.Click -= new System.EventHandler(this.Valid_Click1);  
}
```

Se elimina solo una de las dos asociaciones entre el evento Click y el método Valid\_Click1, la primera. Por tanto permanecerá una y el evento se asocia, a partir de ahora, a dos controladores en lugar de a tres.

# Validar los datos introducidos

## Introducción

El principio general de cualquier aplicación es recopilar datos en entrada y suministrar datos transformados de salida. Estos datos pueden provenir de diferentes fuentes y, antes de poder tratarlos, hay que validarlos para asegurar su correcto formato y, de esta manera, mejorar la fiabilidad de la aplicación.

En caso de los formularios y de los datos introducidos por un usuario, la validación y el control se van a poder hacer a nivel de los campos de entrada y así, a nivel de formulario. Además de la validación y, por tanto, de la comprobación de los errores de entrada de datos, hay que devolver mensajes explicativos al usuario para que pueda corregir sus datos.

# La validación a nivel de campos

## 1. Las propiedades de validación

La validación a nivel de campos verifica los datos introducidos por el usuario. Algunas propiedades de los campos van a permitir limitar la entrada de datos del usuario como, por ejemplo, la propiedad `MaxLength` de los controles de tipo `TextBox`.

Esta propiedad, una vez definida, impide al usuario introducir más caracteres que el número permitido. Por tanto, tenemos una primera validación nativa de los controles de tipo `TextBox` que evita escribir código adicional para detectar el número de caracteres introducidos.

Las propiedades de los controles permiten implementar limitaciones sencillas en la entrada de datos, sin gestionar todos los casos. Si un campo debe tener como entrada un código postal, la propiedad `MaxLength` del campo de tipo `TextBox` para la entrada de datos de este dato permitirá limitar al usuario la entrada de datos a cinco caracteres como máximo, pero nada le impedirá introducir menos o incluso caracteres no numéricos. Los eventos relacionados con el teclado son útiles para validar la entrada de datos.

## 2. Los eventos de validación

Los eventos de teclado como `KeyDown`, `KeyUp` o `KeyPress` se desencadenan en el control que tiene el foco durante la entrada de datos y, según el caso, cuando una tecla se pulsa, se suelta o se pulsa y después se suelta. Cuando estos eventos se desencadenan, el objeto de tipo `KeyEventArgs`, o `KeyPressEventArgs` para el evento `KeyPress`, que se pasa como parámetro al controlador del evento permite obtener información sobre la tecla que ha desencadenado el evento.

### a. `KeyDown` y `KeyUp`

Los eventos `KeyDown` y `KeyUp` sirven principalmente para determinar si las teclas `[Alt]`, `[Ctrl]`, `[Supr]` o `[Mayús]` se han pulsado o soltado. El objeto `KeyEventArgs` que se pasa al controlador expone propiedades booleanas para determinar la combinación de teclas que han desencadenado el evento. La propiedad `KeyCode` permite determinar el código de la tecla que ha desencadenado el evento y, de esta manera, realizar una acción en función de ésta.

### b. `KeyPress`

El evento `KeyPress` lo desencadenan las teclas correspondientes a los caracteres alfabéticos, así como algunos caracteres especiales, tales como las teclas `[Enter]` o `[Del]`. El principio es que si una tecla genera un valor ASCII, ésta desencadena el evento. Por este

motivo, teclas como [Ctrl] o [Alt] no desencadenan este evento, a diferencia de los eventos KeyDown y KeyUp.

La propiedad KeyChar del objeto KeyPressEventArgs que se pasa al controlador del evento KeyPress permite determinar el carácter que se ha rellenado. Para determinar si la tecla pulsada es una cifra, basta con probar este carácter en el controlador de eventos relacionado.

El tipo Char contiene los métodos estáticos que permiten determinar el tipo de carácter que se pasa como parámetro:

```
private void TextBox_KeyPress(object sender, KeyPressEventArgs e)
{
    if (!Char.IsDigit(e.KeyChar))
        MessageBox.Show("El carácter pulsado no es una cifra");
}
```

### **c. Validating y Validated**

Algunos datos se deben verificar en su conjunto y no solo carácter a carácter, ya que no tiene sentido si no están completos. Los eventos Validating y Validated son los eventos que se desencadenan cuando el control pierde el foco y la entrada de datos se considera terminada.

Cada control capaz de recibir una entrada de usuario, ya sea por teclado o por ratón, puede recibir el foco. El control que tiene el foco es el que recibe las entradas de usuario. Solo puede haber uno al mismo tiempo en una aplicación. Una vez que el control pierde el foco y antes de pasarlo al control siguiente, se desencadena el evento Validating. Este evento se da únicamente si el control que va a recibir el foco tiene la propiedad CausesValidation definida a true (valor por defecto). El evento Validating permite realizar la validación del control una vez que el usuario ha terminado de introducir los datos.

En caso de que el dato de control no cumpla las condiciones, el controlador del evento puede utilizar la propiedad Cancel del objeto CancelEventArgs que se pasa como argumento para anular la pérdida de foco del control:

```
private void TextBox_Validating(object sender, CancelEventArgs e)
{
    e.Cancel = true;
}
```

Esta técnica se debe utilizar con precaución porque el foco permanecerá en el control mientras que la entrada de datos no sea correcta. De esta manera, se debe conocer la prueba que permite definir si la entrada de datos es válida o no, para no bloquear indefinidamente el foco en este control.

En caso de que la validación realizada mediante el controlador del evento Validating sea correcta, desencadena el evento Validated, permitiendo realizar cualquier acción con los datos validados.

En el formulario el control de tipo TextBox para la dirección de email de la persona que envía el correo electrónico, debe tener un formato concreto.

Añada un controlador para el evento Validating de este control y haga una prueba para determinar si la dirección de correo electrónico es sintácticamente correcta:

```
void FromEmail_Validating(object sender, CancelEventArgs 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.FromEmail.Text))
    {
        MessageBox.Show("El formato del email es incorrecto.");
        e.Cancel = true;
    }
}
```

Este método utiliza expresiones regulares (en el espacio de nombres System.Text.RegularExpressions), que se detallarán más adelante en este libro, para determinar si el formato de entrada corresponde a una dirección de email.

## La validación a nivel de formulario

La validación a nivel de formulario consiste en probar los datos introducidos una única vez, antes de cerrar el formulario. La idea es recorrer todos los controles para determinar los errores y mostrarlos al usuario una única vez. Además, algunos datos de un formulario solo tienen sentido vistos en conjunto, y no de manera aislada los unos de los otros. Además, un campo que no puede permanecer sin valor y que no haya recibido el foco podría quedar sin valor cuando se cierre el formulario.

Un formulario que tenga un botón de validación permite realizar los controles necesarios para validar los datos en un controlador que responda al evento Click del botón de validación. El formulario contiene el campo que permite especificar el host del servidor de correo, que no puede estar vacío.

Añada el código de validación de este campo al controlador del evento Click del botón **Valid**, creado anteriormente:

```
void Valid_Click(object sender, EventArgs e)
{
    if (string.IsNullOrEmpty(this.Host.Text))
        MessageBox.Show("El campo Host se debe rellenar.");
}
```

Este código no valida los campos uno a uno. Si el foco no entra nunca en el control de entrada de datos del email de la persona que envía, su método de validación, FromEmail\_Validating, nunca se ejecutará. Para lanzar la validación de los controles hijos del formulario, añada el siguiente código al método Valid\_Click:

```
else if (this.ValidateChildren())
{
    // Código para guardar los datos
    this.DialogResult = DialogResult.OK;
}
```

La llamada al método ValidateChildren del formulario devuelve el evento Validating de todos los controles hijo del formulario. Devuelve true si ninguno de los controles contiene algún error de entrada de datos. En caso de entrada de datos correcta, habrá que guardar los datos introducidos y después cerrar el formulario, asignando el valor DialogResult.OK a la propiedad DialogResult del formulario. Esta asignación indica al formulario que se debe cerrar y éste será el valor que devuelva el método ShowDialog que haya abierto el formulario.

Antes de cerrar el formulario, los datos se deben conservar de manera que la próxima vez que se abra el formulario los campos estén rellenos.

Añada un miembro estático de tipo Project a la clase Program:

```
public static Library.Project Project;
```

Este campo es accesible desde cualquier parte del proyecto y este objeto es el que contendrá las modificaciones del usuario. En primer lugar es preciso inicializarlo dentro del método Main de la clase Program:

```
Project = new Library.Project();
```

Para guardar los valores del formulario, añada los siguientes miembros a la clase :

```
protected string fromName;
protected string fromEmail;
protected string host;
protected string username;
protected string password;

public string FromName
{
    get { return this.fromName; }
    set
    {
        if (this.fromName != value)
        {
            this.fromName = value;
            this.HasChanged = true;
        }
    }
}
```

```

    }
}
public string FromEmail
{
    get { return this.fromEmail; }
    set
    {
        if (this.fromEmail != value)
        {
            this.fromEmail = value;
            this.HasChanged = true;
        }
    }
}
public string Host
{
    get { return this.host; }
    set
    {
        if (this.host != value)
        {
            this.host = value;
            this.HasChanged = true;
        }
    }
}
public string Username
{
    get { return this.username; }
    set
    {
        if (this.username != value)
        {
            this.username = value;
            this.HasChanged = true;
        }
    }
}
public string Password
{
    get { return this.password; }
    set
    {
        if (this.password != value)
        {
            this.password = value;
            this.HasChanged = true;
        }
    }
}

```



```
}  
}  
}
```

Ahora es posible guardar en el objeto `Project` los valores introducidos:

```
Program.Project.MailServerSettings.FromName = FromName.Text;  
Program.Project.MailServerSettings.FromEmail = FromEmail.Text;  
Program.Project.MailServerSettings.Host = Host.Text;  
Program.Project.MailServerSettings.Username = Username.Text;  
Program.Project.MailServerSettings.Password = Password.Text;
```

La última acción para completar la mecánica del formulario **MailServerSettings** consiste en restaurar los datos durante la apertura del formulario. Añada el siguiente código en el constructor del formulario:

```
FromName.Text = Program.Project.MailServerSettings.FromName;  
FromEmail.Text = Program.Project.MailServerSettings.FromEmail;  
Host.Text = Program.Project.MailServerSettings.Host;  
Username.Text = Program.Project.MailServerSettings.Username;  
Password.Text = Program.Project.MailServerSettings.Password;
```

Puede probar el formulario ejecutando la aplicación ([F5]).

## Los métodos de retorno al usuario

Cuando un dato es incorrecto, hay que informar al usuario de su error para que lo pueda corregir. Si el error es lo suficientemente explícito, centrar la atención del usuario sobre un control del formulario cambiando su color a rojo puede ser una buena técnica. En los ejemplos anteriores, se ha utilizado el método `MessageBox.Show` para mostrar un mensaje al usuario. Esto permite describir el error y ser más preciso. El componente `ErrorProvider` permite señalar los errores de una manera más elegante.

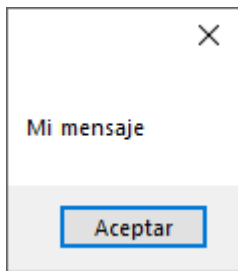
### 1. MessageBox

El tipo `MessageBox`, que contiene el método estático `Show`, es la manera más sencilla de mostrar un mensaje al usuario. Este método muestra un cuadro de diálogo modal, lo que provoca que se detenga la ejecución de la aplicación; el usuario debe realizar una acción para cerrarla.

La manera más sencilla de mostrar un mensaje consiste en utilizar el método `Show` con un único argumento de tipo `string`:

```
MessageBox.Show("Mi mensaje.");
```

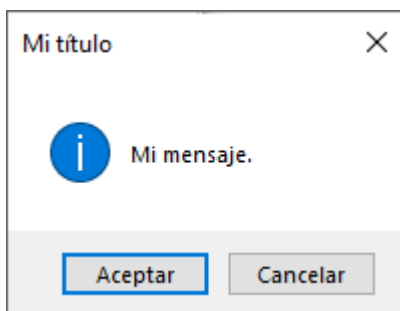
El cuadro de diálogo que se muestra será:



El método Show tiene veinte sobrecargas que permiten personalizar el mensaje que se muestra. De esta manera es posible modificar el título, mostrar un icono en función del tipo de mensaje que se envía, elegir los botones que se muestran y definir el botón por defecto, entre otros argumentos posibles:

```
MessageBox.Show("Mi mensaje.",  
    "Mi título",  
    MessageBoxButtons.OKCancel,  
    MessageBoxIcon.Information,  
    MessageBoxDefaultButton.Button1);
```

El cuadro de diálogo será:



## 2. ErrorProvider

El componente ErrorProvider es un elemento más elegante que el tipo MessageBox para mostrar los mensajes al usuario. Permite definir un mensaje de error en caso de que se produzca una entrada de datos no válida, para cada uno de los controles del formulario. El mensaje de error se presenta como un icono situado junto al control y muestra un tooltip cuando se pasa el ratón sobre él, que detalla el error.

Arrastre desde la caja de herramientas (en el grupo **Componente**) un control **ErrorProvider** y ubíquelo en el formulario. El componente aparecerá en la librería de componentes, en la parte inferior del diseñador de pantallas. Renómbrelo como errorProvider. A partir de este momento, todos los controles del formulario tienen nuevas propiedades, ya que el componente **ErrorProvider** es un extender, es decir, extiende el campo de acción de los controles, asignándole nuevas propiedades y/o métodos en función del componente.

El componente **ErrorProvider** no contiene muchas propiedades, y éstas se configuran rápidamente con el diseñador de pantallas. La propiedad `Icon` permite especificar la imagen que se mostrará junto al control que tenga un error. Las propiedades `BlinkRate` y `BlinkStyle` permiten definir de qué manera se va a resaltar la imagen y con qué velocidad.

Defina la propiedad `BlinkStyle` del componente `errorProvider` con el valor `NeverBlink`, para que el icono se muestre como un error no resaltado.

El componente **ErrorProvider** ha añadido tres nuevas propiedades a los controles que son reconocibles, ya que tienen el nombre del extender:

- `Error` sobre `errorProvider`: esta propiedad contiene la descripción del error para el control. Si se define, se muestra el icono de error.
- `IconAlignment` sobre `errorProvider`: esta propiedad permite definir la posición del icono de error en relación al control.
- `IconPadding` sobre `errorProvider`: el valor de esta propiedad define el espacio entre el control y el icono de error.

La asignación de un error a un control en tiempo de ejecución se hace de una manera específica al componente **ErrorProvider**. Hay que usar su método `SetError`, pasando como argumentos el nombre del control y el texto descriptivo del error.

En el método `FromEmail_Validating` del formulario **MailServerSettings** modifique el método de visualización del mensaje de error por:

```
this.errorProvider.SetError(this.FromEmail,
    "El formato del email es incorrecto.");
```

Haga lo mismo para la visualización del mensaje de error en el método `Valid_Click`:

```
this.errorProvider.SetError(this.Host,
    "El campo host debe estar relleno.");
```

Ejecute la aplicación ([F5]) para observar el resultado:

