



Programació amb C# .NET

Tema 5 (a): Colecciones

Jordi Linares i Pellicer



Introducción

- La utilización de estructuras de datos dinámicas resulta especialmente interesante en muchas aplicaciones.
- En System.Collections.Generic disponemos de una implementación de las estructuras de datos más útiles:
 - Listas (List, LinkedList)
 - Pilas (Stack) y Colas (Queue)
 - Tablas de dispersión/hash (Dictionary)
- En C# y con el uso de los genéricos no es difícil implementar nuestras propias estructuras (árboles, grafos etc.)

Ejemplo: una pila dinámica

```
public class Pila<T>
{
    private class Nodo<T>
    {
        private T info;
        private Nodo<T> sig;

        public Nodo<T> Sig
        {
            get { return sig; }
            set { sig = value; }
        }

        public T Info
        {
            get { return info; }
            set { info = value; }
        }
    }

    private Nodo<T> cabeza = null;

    public void push(T valor)
    {
        Nodo<T> n = new Nodo<T>();
        n.Info = valor;
        n.Sig = cabeza;
        cabeza = n;
    }

    public void pop()
    {
        cabeza = cabeza.Sig;
    }

    public T top()
    {
        return cabeza.Info;
    }

    public Boolean isEmpty()
    {
        return cabeza == null;
    }
}
```

```
Pila<double> p = new Pila<double>();

p.push(3.1);
p.push(3.14);
p.push(3.141);
p.push(3.1415);
p.push(3.14159);

while (!p.isEmpty())
{
    Console.WriteLine(p.top());
    p.pop();
}
```

List, Queue y Stack

- Implementados internamente con un array, redimensionando éste cuando se insertan nuevos elementos.
- El array interno se redimensiona de forma que siempre haya posiciones libres donde insertar (diferencia entre las propiedades Capacity y Count)



List<T>

- Métodos más comunes: Add, Contains, Insert, Remove, Clear, IndexOf, Max, Min, Sort ...
- Se puede acceder con [] a sus components (primera componente 0)
- Ejemplo:

```
List<double> lista = new List<double>();  
  
lista.Add(3.141); lista.Add(3.1415); lista.Add(3.14159);  
  
foreach(double x in lista) Console.WriteLine(x);  
  
lista.Clear();
```



Stack<T>

- Métodos más comunes: Push, Pop (devuelve y elimina el top), Contains, Clear...
- Puede ser enumerada, foreach, sin perder su información.
- Ejemplo:

```
Stack<double> pila = new Stack<double>();  
  
pila.Push(3.141); pila.Push(3.1415); pila.Push(3.14159);  
  
foreach(double x in pila) Console.WriteLine(x);  
  
while (pila.Count > 0) Console.WriteLine(pila.Pop());
```



Queue<T>

- Métodos más comunes: Enqueue, Dequeue (devuelve y elimina la cabeza de la cola), Contains, Clear...
- Puede ser enumerada, foreach, sin perder su información.
- Ejemplo:

```
Queue<string> cola = new Queue<string>();  
  
cola.Enqueue("1"); cola.Enqueue("2"); cola.Enqueue("3");  
  
foreach(double x in cola) Console.WriteLine(x);  
  
while (cola.Count > 0) Console.WriteLine(cola.Dequeue());
```



LinkedList<T>

- Lista totalmente dinámica (doblemente enlazada)
- Métodos más comunes: AddAfter, AddBefore, AddFirst, AddLast, Remove, RemoveFirst, RemoveLast ...
- Se puede enumerar, foreach, o acceder a los elementos a partir de las propiedades First, Last, Next y Previous
- Ejemplo:

```
LinkedList<double> lista = new LinkedList<double>();  
  
lista.AddLast(3.141); lista.AddLast(3.1415); lista.AddLast(3.14159);  
  
foreach(double x in lista) Console.WriteLine(x);  
  
lista.Clear();
```



Dictionary<K, T>

- Tabla de dispersión (hash) con tratamiento de colisiones por encadenamiento
- Métodos más comunes: Add(clave, valor), ContainsKey, Remove(clave), TryGetValue ...
- Se puede usar [] con la clave como índice (se lanza excepción si la llave no está en la tabla)
- Ejemplo:

```
Dictionary<string,string> abrirCon = new Dictionary<string,string>();  
  
abrirCon.Add("txt", "notepad.exe"); abrirCon.Add("jpg", "gimp.exe");  
abrirCon.Add(".doc", "word.exe");  
  
Console.WriteLine("Los txt se abren con " + abrirCon["txt"]);
```



Programació amb C# .NET

Tema 5 (b): Ficheros

Jordi Linares i Pellicer



Introducción

- Utilizamos el espacio de nombres System.IO.
- La clase principal es FileStream, aunque utilizaremos StreamReader/StreamWriter para ficheros de texto y BinaryReader/BinaryWriter para ficheros binarios.

Trabajando con texto

- Utilizamos las clase StreamReader para lectura y StreamWriter para escritura
- Codificación por defecto UTF-8
- Métodos StreamReader:
 - Read: lee un carácter
 - ReadLine: lee hasta fin de línea y lo almacena en un string
 - ReadToEnd: de la posición actual hasta el final
 - ReadBlock: lee un bloque de caracteres

Trabajando con texto

- Métodos StreamWriter: Write y WriteLine
- Ejemplo escritura:

```
using System;
using System.IO;

class Test
{
    public static void Main()
    {
        // Create an instance of StreamWriter to write text to a file.
        // The using statement also closes the StreamWriter.
        using (StreamWriter sw = new StreamWriter("TestFile.txt"))
        {
            // Add some text to the file.
            sw.Write("This is the ");
            sw.WriteLine("header for the file.");
            sw.WriteLine("-----");
            // Arbitrary objects can also be written to the file.
            sw.Write("The date is: ");
            sw.WriteLine(DateTime.Now);
        }
    }
}
```

Trabajando con texto

- Ejemplo escritura:

```
using System;
using System.IO;

class Test
{
    public static void Main()
    {
        try
        {
            // Create an instance of StreamReader to read from a file.
            // The using statement also closes the StreamReader.
            using (StreamReader sr = new StreamReader("TestFile.txt"))
            {
                String line;
                // Read and display lines from the file until the end of
                // the file is reached.
                while ((line = sr.ReadLine()) != null)
                {
                    Console.WriteLine(line);
                }
            }
        }
        catch (Exception e)
        {
            // Let the user know what went wrong.
            Console.WriteLine("The file could not be read:");
            Console.WriteLine(e.Message);
        }
    }
}
```

Trabajando con archivos binarios

- Trabajamos con BinaryReader y BinaryWriter
- Para guardar en el archivo usamos Write
- Para leer podemos usar: ReadByte, ReadBoolean, ReadInt16, ReadInt32, ReadDouble, ReadString etc.



Programació amb C# .NET

Tema 5 (c): Serializació XML

Jordi Linares i Pellicer



Introducción

- XML (eXtensible Markup Language) es un estándar diseñado para el almacenamiento y transporte de información.
- El proceso de serialización de un objeto consiste en almacenar la totalidad de la información de un objeto en memoria secundaria para su posterior recuperación.
- Aunque en .NET se puede serializar en binario, la serialización en XML es especialmente interesante dada la fácil lectura de los ficheros generados.

Serialización XML en .NET

- En .NET el mecanismo de serialización XML más cómodo es mediante el uso de reflexión, mediante el cual prácticamente obtenemos esta serialización de forma automática.
- Básicamente debemos procurar propiedades públicas de todo elemento a serializar de la clase (siempre mejor que atributos/campos públicos). El constructor por defecto también debe estar disponible. La clase tiene que ser pública.
- Mediante atributos específicos, mediante [] antes de cada campo o clase, podemos controlar de forma más exacta esta serialización.

Ejemplo:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Xml;
using System.Xml.Serialization;

namespace ConsoleApplication2
{
    public class Persona
    {
        public string Nombre { get; set; }
        public string Apellido1 { get; set; }
        public string Apellido2 { get; set; }
        public int Edad { get; set; }

        public Persona(string nombre, string apellido1, string apellido2, int edad)
        {
            Nombre = nombre;
            Apellido1 = apellido1;
            Apellido2 = apellido2;
            Edad = edad;
        }

        // Es necesario para serializar XML, aunque
        // no le demos código.
        public Persona()
        {
        }
    }
}
```

Ejemplo:

```
// Serializando ...|
Persona p = new Persona("Jordi", "Linares", "Pellicer", 24);

string filename = @"c:\persona.xml";

Stream stream = null;
try
{
    stream = File.Create(filename);
    XmlSerializer serializer = new XmlSerializer(typeof(Persona));
    serializer.Serialize(stream, p);
}
finally
{
    if (stream != null)
        stream.Close();
}
```

Ejemplo:

```
// Deserializando|
FileStream stream = null;
Persona p = null;

try
{
    stream = File.OpenRead(@"c:\persona.xml");
    XmlSerializer serializer = new XmlSerializer(typeof(Persona));
    p = (Persona)serializer.Deserialize(stream);
}
finally
{
    if (stream != null)
        stream.Close();
}

Console.WriteLine(p.Nombre + " " + p.Apellido1 + " " + p.Apellido2 + " " + p.Edad);
```

Fichero "personal.xml"

```
<?xml version="1.0"?>
<Persona xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Nombre>Jordi</Nombre>
  <Apellido1>Linares</Apellido1>
  <Apellido2>Pellicer</Apellido2>
  <Edad>24</Edad>
</Persona>
```

Otro ejemplo:

```
List<Persona> personas = new List<Persona>();

personas.Add(new Persona("Jordi", "Linares", "Pellicer", 24));
personas.Add(new Persona("Empar", "Llorens", "Garcia", 32));
personas.Add(new Persona("Toni", "Colomina", "Rodriguez", 48));

string filename = @"c:\personas.xml";

Stream stream = null;
try
{
    stream = File.Create(filename);
    XmlSerializer serializer = new XmlSerializer(typeof(List<Persona>));
    serializer.Serialize(stream, personas);
}
finally
{
    if (stream != null)
        stream.Close();
}
```

Fichero "personas.xml":

```
<?xml version="1.0"?>
<ArrayOfPersona xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Persona>
    <Nombre>Jordi</Nombre>
    <Apellido1>Linares</Apellido1>
    <Apellido2>Pellicer</Apellido2>
    <Edad>24</Edad>
  </Persona>
  <Persona>
    <Nombre>Empar</Nombre>
    <Apellido1>Llorens</Apellido1>
    <Apellido2>Garcia</Apellido2>
    <Edad>32</Edad>
  </Persona>
  <Persona>
    <Nombre>Toni</Nombre>
    <Apellido1>Colomina</Apellido1>
    <Apellido2>Rodriguez</Apellido2>
    <Edad>48</Edad>
  </Persona>
</ArrayOfPersona>
```

Algunos atributos útiles

- Mediante atributos específicos, mediante [] antes de cada campo o clase, podemos controlar de forma más exacta esta serialización.
- Algunos atributos útiles:
 - XmlRoot(), para dar un nombre distinto al objeto serializado.
 - XmlAttribute(), para definir un campo como atributo y no elemento XML
 - XmlIgnore(), para que se ignore un elemento en la serialización.
 - etc.

Ejemplo:

```
public class Persona
{
    [XmlAttribute()]
    public string DNI { get; set; }

    public string Nombre { get; set; }
    public string Apellido1 { get; set; }
    public string Apellido2 { get; set; }
    public int Edad { get; set; }

    public Persona(string dni, string nombre, string apellido1, string apellido2, int edad)
    {
        DNI = dni;
        Nombre = nombre;
        Apellido1 = apellido1;
        Apellido2 = apellido2;
        Edad = edad;
    }

    // Es necesario para serializar XML, aunque
    // no le demos código.
    public Persona()
    {
    }
}
```

Ejemplo:

```
<?xml version="1.0"?>
<Persona xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  DNI="21133219Z">
  <Nombre>Jordi</Nombre>
  <Apellido1>Linares</Apellido1>
  <Apellido2>Pellicer</Apellido2>
  <Edad>24</Edad>
</Persona>
```



Programació amb C# .NET

Tema 5(d): Windows Forms

Jordi Linares i Pellicer




Índice

- Introducción
- Creación de una Aplicación Windows
- Windows Forms
- Gestión de Eventos
- La clase Control



Introducción

- Actualmente, las aplicaciones diseñadas para interactuar con usuarios cuentan con una interfaz basada en **ventanas**.
- Las **interfaces gráficas de usuario** (GUI) son un estándar pues simplifican el uso de programas proporcionando al usuario un conjunto de componentes gráficos muy intuitivos y fáciles de usar.
- Visual Studio cuenta con un diseñador de formularios y herramientas para crear interfaces de usuario de una forma gráfica, sencilla y eficiente con generación automática de código.

- 
- Windows Forms fue el primer conjunto de clases que se diseñó en .NET para la creación de aplicaciones gráficas.
 - La alternativa más actual consiste en Windows Presentation Foundation (WPF).
 - Windows Forms continua siendo muy utilizada y es imprescindible en estos momentos para desarrollar aplicaciones multiplataforma, ya que el proyecto Mono, hoy por hoy, únicamente soporta Windows Forms.

Creación de una Aplicación Windows

- Una aplicación Windows Forms ha de cumplir que
 - Tiene al menos un formulario (ventana principal de la aplicación).
 - Ha de mostrar dicho formulario en el arranque.
 - Ha de terminar adecuadamente.
- El punto de entrada es el método estático **Main**

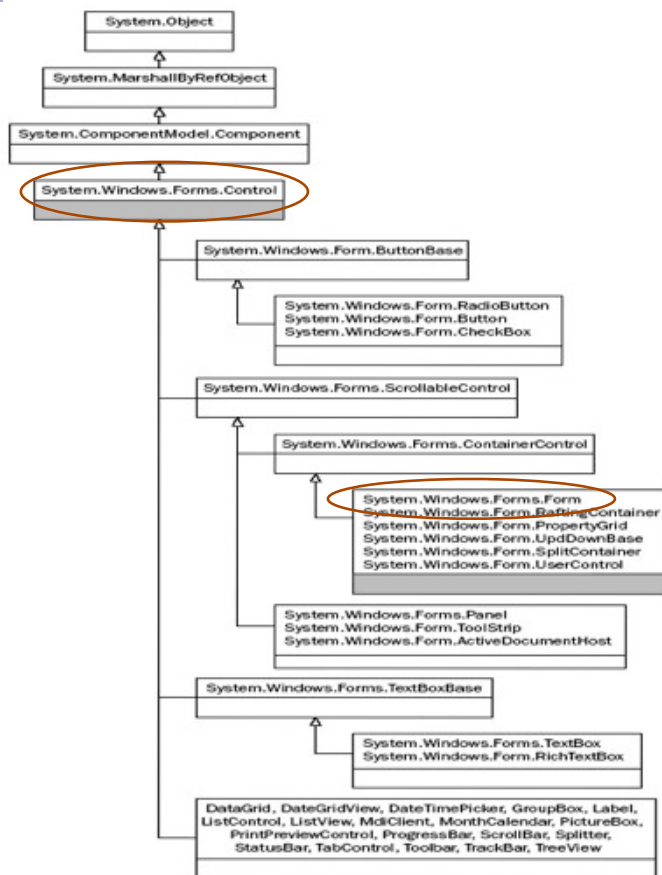
Ejemplo:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Windows.Forms;

namespace windowsFormsApplication1
{
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new Form1());
        }
    }
}
```

Windows Forms

- Permiten crear la interfaz gráfica de nuestros programas.
- Los formularios se pueden diseñar desde Forms Dessigner.
- El espacio de nombres System.Windows.Forms contiene clases para crear aplicaciones Windows que aprovechan todas las ventajas de las características de la interfaz de usuario disponibles en Windows.
- Todos los formularios derivan de System.Windows.Forms.Form
- Pueden contener otros controles (deriva de ContainerControl) y de desplazar en su interior los controles no caben en su área (deriva de ScrollableControl).
- Además, heredan todas las características y funcionalidad de la clase Control .
- Pueden mostrarse en modo **Modal** y en modo **No modal**.
- Un formulario puede ser:
 - Un Cuadro de Diálogo,
 - Una Ventana (SDI),
 - Una Ventana Multidocumento (MDI).



■ Conceptos:

□ Componente

- Clase que implementa la interfaz IComponent.
- Carece de partes visuales, sólo tiene código.

□ Control

- Componente con interfaz gráfica (botón, cuadro de texto, ...)
- Son visibles.

□ Evento

- Se genera por el movimiento del ratón, el teclado, ...
- Los Manejadores de Eventos ejecutan la acción correspondiente al evento producido (implementados por el programador).

- En Windows siempre hay una sola ventana activa y se dice que tiene el foco (**focus**).
- Un formulario es un **contenedor** de controles y componentes.
- VS permite la **programación visual**:
 - Al arrastrar un control o componente dentro del formulario, VS genera automáticamente todo el código para crear el objeto y establece sus propiedades básicas.
 - Si cambian las propiedades, etc. de un componente o control se modifica automáticamente el código asociado.
 - Si algún componente se elimina, se elimina también el código subyacente.

- El método `Application.EnableVisualStyles()` habilita los estilos visuales del sistema operativo para la aplicación actual.
- Es importante el orden de los eventos de **instanciación** y **destrucción** de un formulario:
 1. **Constructor**, formulario en proceso de creación.
 2. **Load**, creado pero no visible.
 3. **Activated**, formulario visible y activo.
 4. **Closing**, en proceso de cierre. Posibilidad de cancelar.
 5. **Closed**, formulario ya cerrado.
 6. **Deactivate**, formulario inactivo.
- Cuando se llama al método `Application.Exit()` no se producen los eventos `Closing` y `Closed` pero sí se llama al método `Dispose`.

Métodos más comunes	
Close	Cierra el formulario y libera sus recursos.
Hide	Oculto el formulario, pero sin cerrarlo.
Show	Hace visible un formulario oculto.

Propiedades más comunes	
AcceptButton	Botón por defecto al pulsar la tecla Intro
CancelButton	Botón por defecto al pulsar la tecla Esc
AutoScroll	Barras de scroll automáticas
Font	Estilo de fuente para el formulario y sus controles
Text	Texto de la barra de título del formulario
StartPosition	Posición inicial del formulario
ShowInTaskbar	Mostrar en la barra de tareas de Windows
TopMost	Formulario en primer plano
ControlBox	Mostrar el cuadro de control en la barra de título del formulario
MaximizeBox MinimizeBox	Mostrar el botón de maximizar y minimizar
FormBorderStyle	Estilo del borde de la ventana del formulario: Fixed3D, FixedDialog, FixedSingle, FixedToolWindow, None, Sizable, SizableToolWindow.

Eventos

- Las aplicaciones Windows son **dirigidas por eventos**.
- Los **Manejadores de Eventos** son los métodos que se encargan de procesar los eventos y ejecutar la tarea adecuada al evento.
- **Ejemplo:**
 - Aplicación con sólo un formulario y un botón que muestra un mensaje al pulsarlo.



- El código manejador del evento clic del botón sería:

```
private void button1_Click(object sender, EventArgs e)
{
    MessageBox.Show("Hola, benvingut");
}
```

- **object** → referencia al objeto que genera el evento (botón)
- **EventArgs** → referencia al objeto de los argumentos del evento. Contiene información extra del evento ocurrido.
- Visual Studio genera el código que crea e inicializa nuestra interfaz gráfica realizada con el diseñador.
- Este código se encuentra en el fichero Form1.Designer.cs
 - El código generado por el diseñador no debe modificarse directamente pues puede producir que la aplicación no funcione correctamente.

■ Delegados y eventos.

- Un control no sabe, a priori, qué métodos responderán a sus eventos.
- Los **delegados** son los objetos que mantienen una referencia al método con la misma signatura que especifica el tipo del delegado.
- Todos los controles tienen ya **predefinidos** delegados para cada evento que generan.
- Por ejemplo, el delegado del evento click del botón es del tipo System.EventHandler.
- La declaración del delegado EventHandler, mirandola en la ayuda de Visual Studio, es:

```
public delegate void EventHandler( object sender, EventArgs e );
```

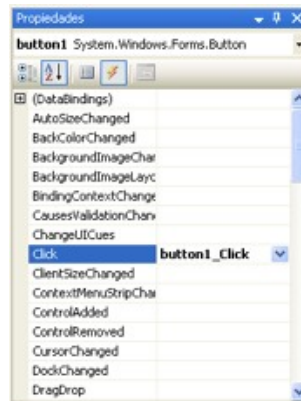
que coincide con la declaración del evento click del botón:

```
private void button1_Click(object sender, EventArgs e)
```

- El control llama al delegado cuando se produce el evento.
- La tarea del delegado es invocar al método adecuado.
- Visual Studio genera el código que asigna el manejador `button1_Click` al delegado:

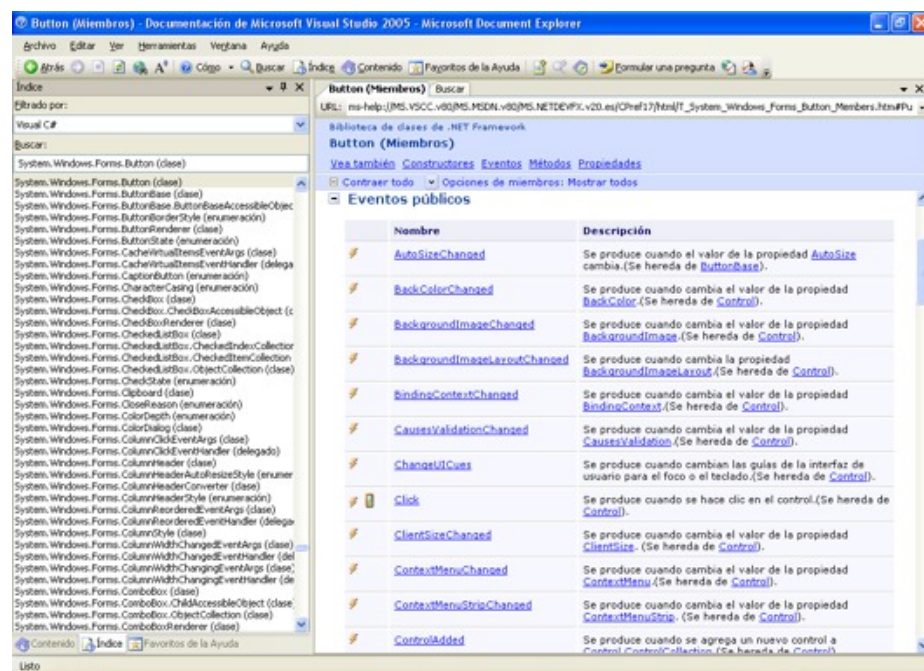
```
this.button1.Click += new System.EventHandler(this.button1_Click);
```

- Desde la ventana Propiedades se pueden crear manejadores para otros eventos del control:



- En la documentación de Visual Studio se puede consultar los eventos que genera cada control.


- **Ejemplo:** En el índice de la Ayuda de VS buscamos la clase `System.Windows.Forms.Button`:





La Clase Control

- La clase `System.Windows.Forms.Control` contiene toda la funcionalidad e infraestructura para todo lo que se muestra en controles y formularios.
- La clase `Control` deriva de `System.ComponentModel.Component`
- **Propiedades** comunes:
 - **BackColor**: color de fondo del control
 - **BackgroundImage**: imagen de fondo del control
 - **Enabled**: Si true activado. El usuario puede interactuar con él.
 - **Focused**: Indica si el control tiene el focus.
 - **Font**: Tipo de fuente para mostrar el texto del control.
 - **ForeColor**: Color del primer plano. Color del texto del control.
 - **TabIndex**: Orden del control en la lista de tabulación.
 - **TabStop**: true si puede tener el focus con la tecla TAB
 - **Text**: Texto asociado al control.
 - **Visible**: Indica si el control es visible al usuario.

- 
- **Métodos** comunes:
 - **Focus**: El control toma el focus.
 - **Hide**: Oculta el control y pone la propiedad `Visible` a false.
 - **Show**: Muestra el control y pone la propiedad `Visible` a true.
 - **Propiedades de Distribución**:
 - **Anchor**: El control siempre permanece a una distancia fija del borde de su contenedor aunque este se redimensione.
 - **Dock**: Permite adosar el control a un lado de su contenedor expandiéndolo o ocupándolo por completo.
 - **Margin**: Margen externo del control.
 - **Padding**: Margen interno del control.
 - **Location**: Posición (coordenadas) del borde superior izquierdo del control respecto a su contenedor.
 - **Size**: Tamaño del control en píxels como un objeto `Size` (`Width`, `Height`).
 - **MinimumSize**, **MaximumSize**: Indica el tamaño mínimo y máximo del control.



■ **Eventos** comunes:

- **KeyDown, KeyPress, KeyUp:** Se producen en este orden al pulsar una tecla y liberarla. KeyDown y KeyUp interceptan cualquier pulsación de tecla no sólo las alfanuméricas como KeyPress.
- **MouseDown, MouseUp, MouseMove:** Se generan al pulsar y soltar un botón del ratón y al mover el puntero sobre el control.
- **Click, DbClick:** Indican que se ha hecho clic o doble clic sobre el control. La tecla Intro, en ocasiones, también produce un evento clic sobre un botón.
- **Validating, Validated, Enter, Leave, GotFocus y LostFocus:** Son eventos que se producen cuando un control toma el focus o lo pierde en este orden: **GotFocus**, **Leave**, **Validating**, **Validated**, y **LostFocus**.

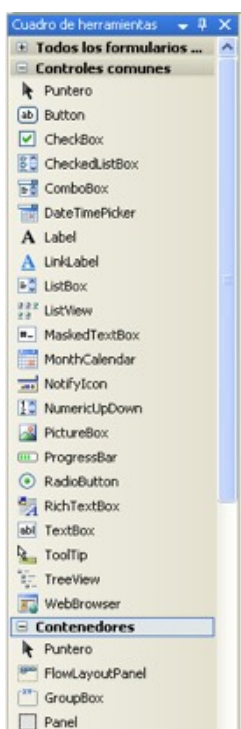
Validating y **Validated** se producen en la validación del control si su propiedad CausesValidation=true. Permiten, si es necesario, cancelar los eventos sucesivos.

GotFocus i **LostFocus** son de bajo nivel. En su lugar deben usarse **Enter** i **Leave** para todos los controles.

Programació amb C# .NET

Tema 5(e) Controles y Componentes habituales en WinForms

Índice

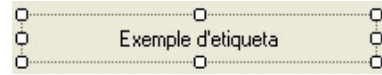


- Etiquetas y Cuadros de texto
- Botones
- Marcos y paneles
- Opciones
- Listas
- Imágenes
- Ayuda emergente
- Control numérico
- Fecha y hora
- Eventos del ratón

Etiquetas y cuadros de texto

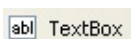
■ **Label**

- ☐ Muestra un texto descriptivo.
- ☐ El usuario no puede modificar el texto.
- ☐ El texto se puede cambiar desde el programa.
- ☐ Propiedades:
 - **Font** → Tipo de fuente para el texto de la etiqueta.
 - **Text** → El texto que se muestra.
 - **AutoSize** → Si = true el tamaño de la etiqueta se adapta al texto.
 - **TextAlign** → Si AutoSize=False, alineación del texto dentro de la etiqueta (en horizontal y en vertical).



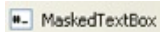
■ **TextBox, MaskedTextBox y RichTextBox**

- ☐ Muestran texto que el usuario puede modificar.
- ☐ Derivan de TextBoxBase y heredan las propiedades:
 - **MultiLine** → Si = true indica que el control muestra el texto en varias líneas.
 - **Lines** → Array de strings que contiene las líneas del texto.
 - **Text** → Devuelve en un solo string todo el texto del control.
 - **TextLenght** → Devuelve la longitud del texto.
 - **MaxLenght** → Establece la longitud máxima del texto.
 - **SelectedText, SelectionLength, and SelectionStart** → Hacen referencia al texto que se selecciona en el cuadro de texto.
- ☐ El evento **TextChanged** se produce al modificar el texto.
- ☐ **TextBox** añade las propiedades:



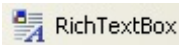
- **AcceptsReturn** → Si = true la tecla Intro se interpreta como una nueva línea en el texto del control.
- **CharacterCasing** → Puede ser Lower, Normal o Upper.
- **PasswordChar** → Carácter que se muestra al escribir.

- **MaskedTextBox** permite limitar y dar formato al texto que introduce el usuario en este control:



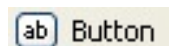
- **Mask** → Contiene la máscara que se va a utilizar.
- **InputText** → Devuelve el texto introducido sin formato.
- **OutputText** → Devuelve el texto con formato.

- **RichTextBox** permite editar texto con formato específico usando el formato RTF.



- La propiedades **SelectionFont**, **SelectionColor**, **SelectionBullet**, **SelectionIndent**, **SelectionRightIndent**, **SelectionHangingIndent** afectan al texto seleccionado, y si no hay, el formato se aplica al nuevo texto introducido.
- **Text** → Contiene el texto sin formato.
- **Rtf** → contiene el texto en formato RTF.
- **LoadFile** → Carga texto desde un fichero.
- **SaveFile** → Guarda el texto del control en un fichero.

Botones: Button

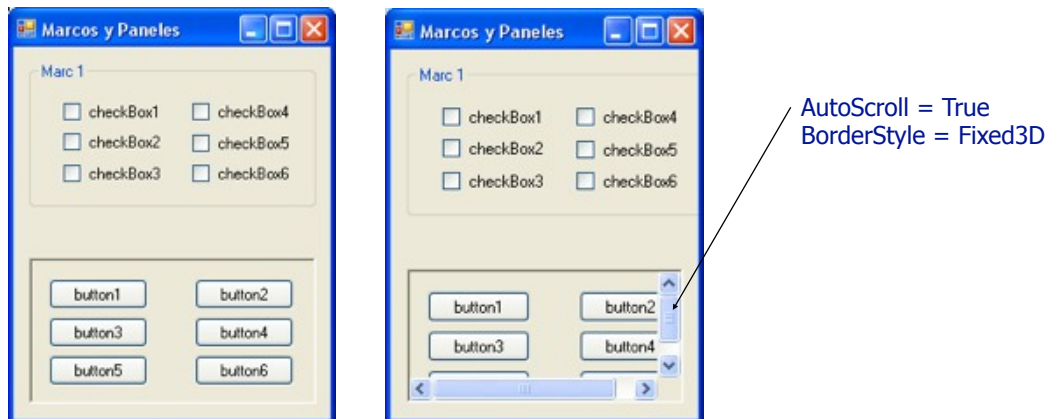


- Un botón es un control que el usuario pulsa para ejecutar una acción.
- El control Button deriva de ButtonBase.
- Se escribirá el código para responder al evento **click** del botón que se produce al pulsarlo.
- Propiedades:
 - **Text** → Indica el texto que muestra el botón.
 - **FlatStyle** → Modifica la apariencia del botón.
- Un botón también puede mostrar una imagen:



Marcos y paneles: GroupBox, Panel

- Permiten agrupar controles en su interior que pueden anclarse y adosarse.
- Pueden contener a su vez otros marcos y paneles.
- La propiedad **Controls** da acceso a los controles que contienen.



Opciones: CheckBox y RadioButton

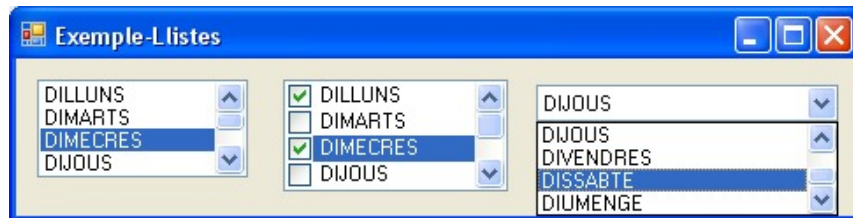
- Derivan de la clase **ButtonBase**.
- Pueden estar activados o desactivados (**true/false**)
- Los **RadioButton** se utilizan en grupo de opciones donde sólo se puede elegir una de ellas.



- La propiedad **Checked** indica si està activado o no.
- El evento **CheckedChanged** se produce cuando cambia el estado de la propiedad **Checked**.

Listas: ListBox, CheckedListBox, ComboBox

- Todas derivan de la clase **ListControl**.
- Contienen una lista de elementos que pueden añadirse, seleccionarse y eliminarse.
- **ListBox** y **CheckedListBox** permiten ver y seleccionar varios elementos de forma simultánea.
- **ComboBox** contiene además, un cuadro de texto y sólo permite seleccionar un elemento de la lista.
- Los elementos de la lista pueden ser objetos luego cualquier tipo de .NET puede ser añadido a la lista.



■ Propiedades

- **Items** → Colección de elementos de la lista.
- **SelectionMode** → None, One, MultiSimple, MultiExtended
- **SelectedItem** → Referencia al elemento seleccionado.
- **SelectedItems** → Colección de elementos seleccionados.
- **SelectedIndex** → Índice del elemento seleccionado o -1 si no hay ninguno.
- **SelectedIndices** → Colección de índices de los elementos seleccionados.
- **Sorted** → Indica si están ordenados. Por defecto = false.

■ Métodos

- **ClearSelected()** → Quita la selección actual.
- **GetSelected(ind)** → Devuelve true si el elemento de índice ind está seleccionado.

- **Evento** común:
 - **SelectedIndexChanged** → Se genera cada vez que SelectedIndex cambia.

- **Manejar la lista**

.Items.Add(element)
 .Items.RemoveAt(posIndex)
 .Items.Clear()
 .Items.Count



Control de imagen: PictureBox

- Muestra una imagen en su interior.
- Acepta múltiples formatos: BMP, GIF, TIFF, JPEG, ...

- **Propiedades:**

- **Image** → Imagen a mostrar en el control. Usar el método FromFile de la clase Image para cargarla desde el código.
- **SizeMode** → Normal (defecto), StretchImage, AutoSize y CenterImage.

- **Evento:**

- **Click** → Se produce al hacer clic sobre el control.

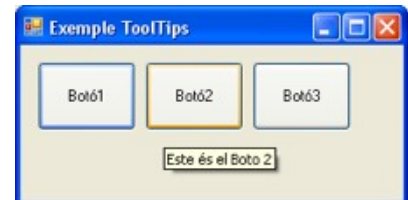


Ayuda emergente: ToolTip

- Texto de ayuda que aparece al detener el ratón sobre un elemento de la ventana.
- Añadido a un formulario aparece una nueva propiedad disponible: **ToolTip on** <nomToolTip>

- **Propiedades:**

- ☐ **AutoPopDelay** → Tiempo (ms) que se muestra el texto.
- ☐ **InitialDelay** → Tiempo de espera hasta mostrarlo.
- ☐ **ReshowDelay** → Tiempo para mostrar dos ToolTips de dos controles diferentes.
- ☐ **ToolTipIcon** → Icono a mostrar junto al texto.



Control numérico: NumericUpDown

- Permite elegir entre un rango de valores.



- **Propiedades:**

- ☐ **Increment** → valor de incremento.
- ☐ **Minimum** → valor mínimo del rango.
- ☐ **Maximum** → valor máximo del rango.
- ☐ **UpDownAlign** → posición de los botones: izq/der.
- ☐ **Value** → valor actual.

- **Evento**

- ☐ **ValueChanged** → se produce cuando cambia el valor del control.

Fecha y hora: DateTimePicker

- Permite elegir la fecha/hora en diferentes formatos.
- El calendario aparece al pulsar sobre la flecha.

- **Propiedades:**

- ☐ **Format** → Long/Short/Time/Custom
- ☐ **CustomFormat** → Cadena de formato cuando Format=Custom.
- ☐ **MinDate/MaxDate** → Menor y mayor fecha que se puede seleccionar.
- ☐ **Value** → Objeto `DateTime` con la fecha seleccionada.
- ☐ **Text**: Cadena de texto con formato de la fecha seleccionada.

- **Evento:**

- ☐ **ValueChanged**: Se produce cuando cambia value.



Eventos del ratón

- Se producen al interaccionar el usuario con el ratón.
- La información del evento se pasa a un objeto del tipo `MouseEventArgs`.
- **MouseEnter** / **MouseLeave** → el ratón entra/sale de los límites del control.
- **MouseDown** → se ha pulsado un botón del ratón.
- **MouseMove** → el ratón se mueve sobre el control.
- **MouseUp** → se libera un botón del ratón.
- **Propiedades** de `MouseEventArgs`:
 - ☐ **Button** → botón pulsado: Left, Right, Middle o none
 - ☐ **Clicks** → número veces que se ha hecho clic.
 - ☐ **X e Y** → coordenada x e y dentro del control.



Programació amb C# .NET

Tema 5 (f) Diseño avanzado en WinForms

Jordi Linares i Pellicer



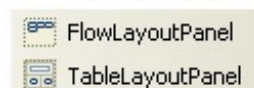
Índice

- Introducción
- Distribución automática de controles
- Otros contenedores: SplitContainer y TabControl
- Menús
- Barras de herramientas y de estado
- Cuadros de diálogo
- Aplicaciones MDI

Introducción

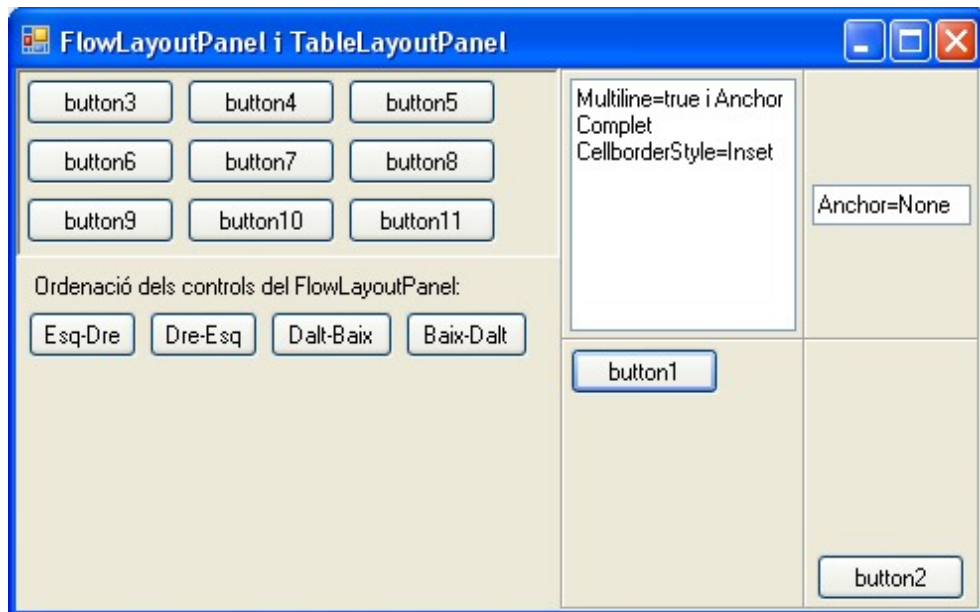
- Las interfaces de usuario se diseñan para que sean cómodas al usuario.
- La aplicación debe dar, en conjunto, una imagen de **homogeneidad**.
- La interfaz ha de ajustarse a las **preferencias** del usuario final.
- Contenedores, menús, barras de herramientas, cuadros de diálogo, etc. permiten diseñar interfaces más elaboradas y con un aspecto profesional.

Distribución Automática de Controles

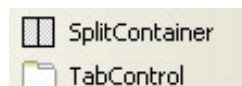


- **FlowLayoutPanel** y **TableLayoutPanel** son dos controles contenedores que permiten distribuir los controles en su interior de forma automática:
 - **FlowLayoutPanel** → Permite distribuir los controles en horizontal o vertical. Propiedades: FlowDirection y WrapContents.
 - **TableLayoutPanel** → Permite distribuir los controles en forma de tabla.
 - Las filas y columnas se pueden editar.
 - Cualquier control puede añadirse a las celdas de este panel.

Ejemplo:



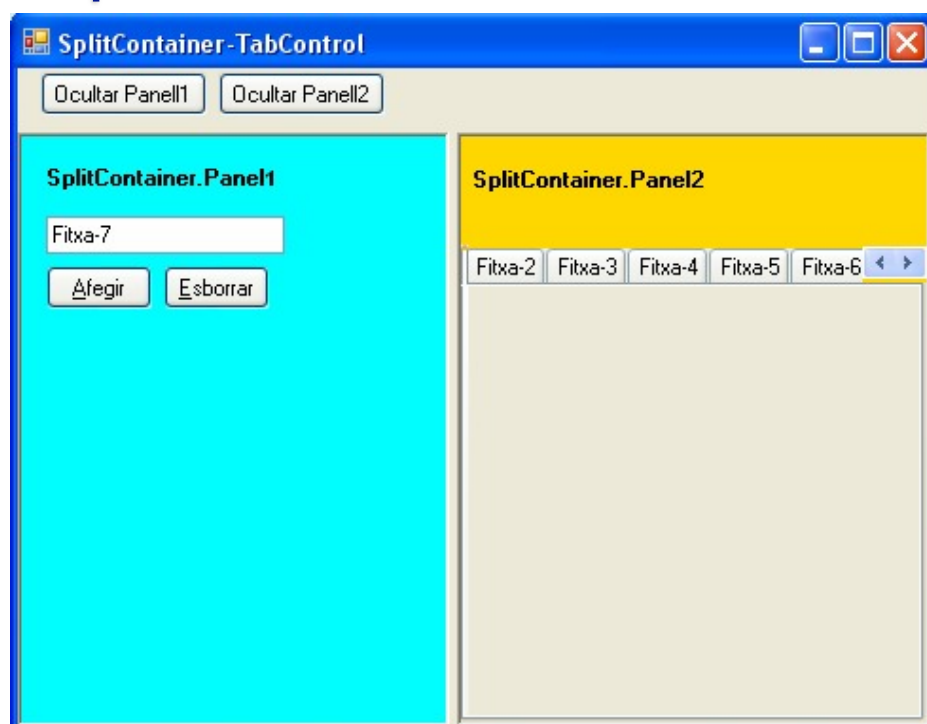
SplitContainer y TabControl



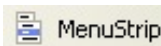
- El control **SplitContainer** es un panel ajustable que contiene a su vez dos paneles separados por una barra divisoria móvil ([Splitter](#)).
- El usuario puede desplazar la barra provocando que cambien de tamaño los paneles.
- Un ejemplo lo tenemos en el Explorador de Ficheros.
- **Propiedades:**
 - **Orientation**: Paneles en Horizontal/Vertical.
 - **IsSplitterFixed**: Indica si el divisor se puede mover.
 - **Panel1MinSize**, **Panel2MinSize**: Tamaño mínimo del panel.
 - **Panel1Collapsed**, **Panel2Collapsed**: Ocultar panel.

- **TabControl** permite agrupar los controles en una serie de fichas (**TabPage**).
- **Propiedades:**
 - **TabAppearance**: Aspecto de la ficha: FlatButtons, Buttons o Normal.
 - **Multiline**: Indica si se presenta más de una fila de fichas.
 - **SelectedTab**: Ficha seleccionada
 - **SelectedIndex**: Índice de la ficha activa.
 - **TabCount**: Número de fichas.
 - **TabPages**: Colección de TabPage.
- **Evento:**
 - **SelectedIndexChanged**: se produce cuando cambia el valor de SelectedIndex, es decir, cuando se selecciona otra ficha.

Ejemplo:

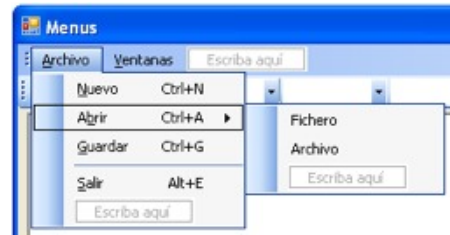


Menús



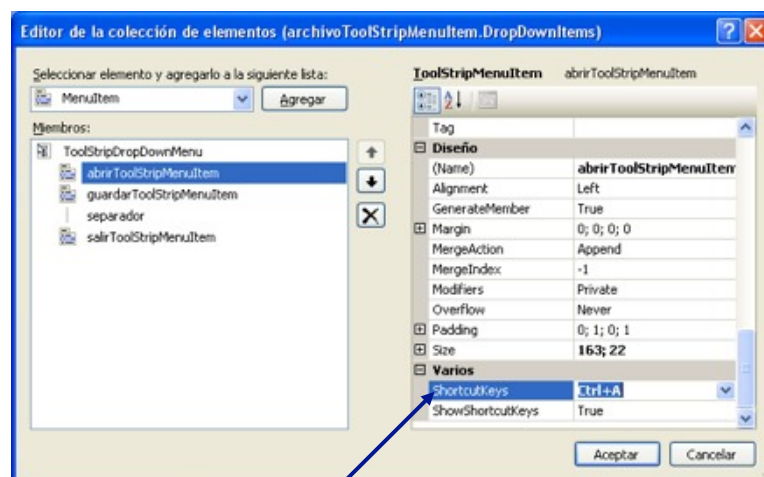
- El control **MenuStrip** permite introducir una barra de menú en nuestra aplicación. Deriva de la clase ToolStrip.
- Se construye añadiendo objetos ToolStripMenu al control MenuStrip en modo diseño o mediante código.
- Permite insertar elementos de menú estándar.
- **Propiedades de ToolStripMenuItem:**

- **Check:** Indica que el elemento está marcado.
- **CheckOnClick :** Si=true muestra una marca al hacer clic.
- **Enabled:** Si =false indica que está desactivado.
- **DropDownItems:** Muestra la lista de elementos asociada.



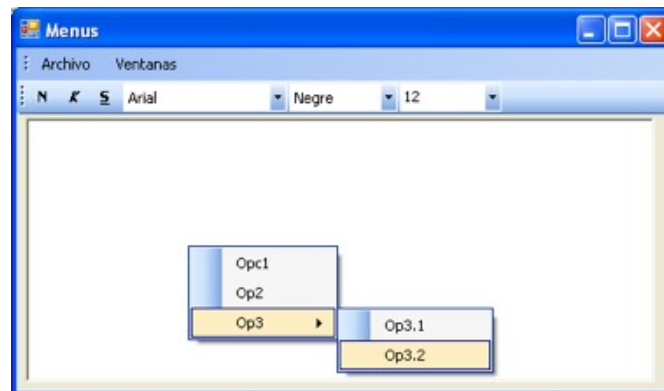
Menús

- **Eventos:**
 - **Click** se produce al hacer clic sobre un elemento.
 - **CheckedChanged:** se produce al hacer clic si el elemento tiene asignada la propiedad CheckOnClick=true.
- Edición y propiedades de los elementos del menú:



Teclas de acceso rápido

- El control **ContextMenuStrip** permite mostrar un menú contextual al pulsar botón derecho del ratón sobre un control.  ContextMenuStrip
- Contiene elementos del tipo ToolStripMenuItem.
- Se crea de la misma forma que un MenuStrip.
- Para mostrar un menú de contexto para un control determinado se asigna el menú contextual creado a la propiedad ContextMenuStrip del control.

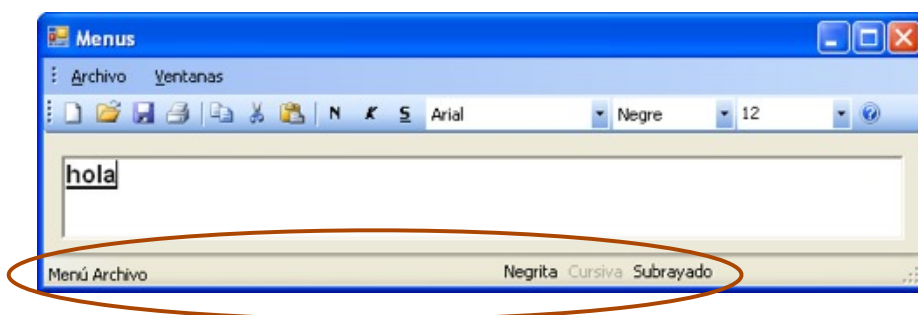


Barras de Herramientas y de Estado

- Las barras de herramientas **ToolStrip** proporcionan un acceso directo a las acciones más utilizadas de una aplicación.
- Suelen contener botones pero también cuadros de texto, listas, etc.
- Permite también insertar botones estándar: nuevo, abrir, guardar, ... etc.
- **Propiedades:**
 - ☐ GripStyle, LayoutStyle, Items , Stretch, ...

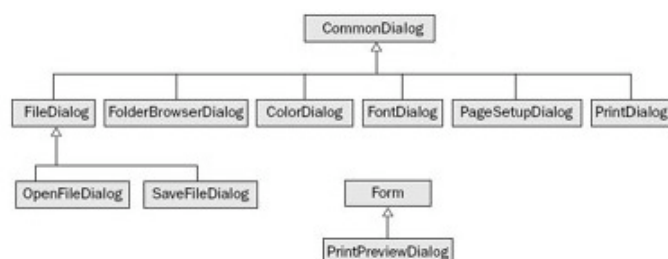
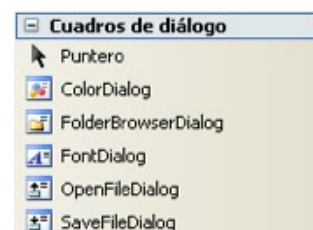


- El control **StatusStrip** permite mostrar una barra con paneles que informan del estado actual de la aplicación.
- Deriva también de ToolStrip.
- El elemento ToolStripStatusLabel es además específico de la barra de estado. **Propiedades:**
 - **AutoSize:** Indica si se redimensiona automáticamente al texto que contenga. Es preferible ponerlo a false.
 - **DoubleClickEnable:** Indica si el DobleClic sobre la misma provoca un evento.



Cuadros de diálogo

- Son componentes que se añaden a nuestro formulario.
- Permiten realizar tareas cotidianas como: abrir y guardar ficheros, cambiar la fuente, elegir un color, etc.
- Todos tienen un método común: ShowDialog() que abre el cuadro de diálogo con las opciones establecidas.
- Jerarquía:



- **Ejemplo** de uso.

```
if (colorDialog1.ShowDialog() == DialogResult.OK)
    this.BackColor=colorDialog1.Color;
```

Aplicaciones MDI

- Son aplicaciones de Interfaz de Documento Múltiple.
- Un ejemplo lo tenemos en el Adobe Reader.
- Una ventana actúa como madre de las ventanas hija que se alojan en la misma.
- Para tener una aplicación MDI la propiedad **IsMDIContainer = true** en el formulario que actuará como ventana principal.
- A las ventanas hijas se asigna la propiedad **MDIParent** la referencia a la ventana principal.
- Pueden abrirse tantas ventanas hijas como se desee desde el formulario principal:

```
VentanaHija UnaHija = new VentanaHija();  
UnaHija.MdiParent = this;  
UnaHija.Show();
```

- **Propiedades** en la ventana madre:
 - **ActiveMDIChild** → devuelve la ventana hija activa.
 - **MdiChildren** → devuelve un array con las ventanas hijas.
- **Método** en la ventana madre:
 - **LayoutMdi(MdiLayout)** → organiza las ventanas.
- **Evento** en la ventana madre:
 - **MdiChildActivate** → ocurre cada vez que se activa o una ventana hija.