

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM	
	MÓDULO	Desarrollo de Interfaces			CURSO: 2º
	PROTOCOLO:	Apuntes clases	AVAL:	DATA:	
	UNIDAD COMPETENCIA				

Tema 5 – Creación de componentes

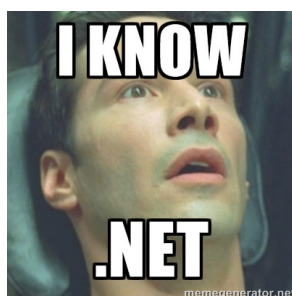
Hasta el momento hemos estado utilizando componentes gráficos incluidos en la librería Windows Forms, pero en ocasiones sucederá que los componentes existentes no nos solucionan lo que queremos hacer, ya sea porque son incompletos o porque no existe lo que queremos realizar.

Es por ello que veremos distintas formas de crear nuestros propios componentes y como reutilizarlos en futuras aplicaciones guardándolos en DLLs. Es un tema principalmente de diseño orientado a objetos pero teniendo como objetivo final la creación de un componente gráfico interactivo.

Cuando un programador en .Net en Windows Forms desea hacer sus componentes a medida tiene varias posibilidades. Las más típicas son:

- **Heredar uno existente** y modificarlo para que tenga un comportamiento y/o aspecto deseado distinto al original (Inherited controls). Es útil cuando queremos hacer un componente prácticamente igual a uno que ya existe pero con ligeras modificaciones o ampliaciones en su aspecto o comportamiento.
- Heredar de la clase **UserControl**. Esto permite crear un nuevo control agrupando controles ya existentes. También se le puede dar nuevas funcionalidades. La clase UserControl nos da una serie de elementos genéricos para la creación de cualquier componente como eventos predefinidos o diseño mediante IDE.
- Crear desde cero uno nuevo heredando la clase **Control** (Owner drawn controls). Este caso es cuando se desea la creación a bajo nivel. También existen algunos elementos heredados de Control, pero no tanto como en el caso de UserControl, sobre todo elementos que permiten usar el IDE de VisualStudio para el diseño. En heredados de Control no se usa el IDE de diseño, es todo programación.

En este capítulo hay algo que es importante tener en cuenta y es que no creamos programas, si no que estamos creando nuevas clases que no podremos usar hasta que se instancien como objetos en ciertas aplicaciones.



COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Desarrollo de Interfaces				CURSO: 2º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	UNIDAD	COMPETENCIA				

Creación de componentes UserControl

En primer lugar veremos como crear agrupaciones de componentes (agregación de objetos) como si fuera nuevos componentes heredando de la clase *UserControl*. Esto nos da una serie de elementos ya definidos a partir de los cuales resulta sencillo crear nuevos componentes.

Realmente para crear un nuevo componente llega con heredar de la clase Control como veremos en un apartado posterior. De hecho la clase UserControl también hereda de Control. Esta clase es una plantilla vacía que se puede usar para crear controles pero que ha heredado el código necesario para el control de algunos eventos, posicionamiento, uso en el IDE, etc... lo que facilita la escritura de nuevos componentes.

En este punto lo más complejo es hacer visible las propiedades y eventos de los componentes internos hacia el externo.

Aprenderemos a hacerlo mediante un ejemplo: Crearemos un **componente que contenga tanto un TextBox como una Label asociada al mismo de forma que sean una unidad indivisible** que podremos usar para múltiples aplicaciones.

Además daremos la posibilidad de que la etiqueta esté a la izquierda o a la derecha de la *TextBox* mediante una nueva propiedad.

También habrá una propiedad que describa la separación en píxeles entre ambos componentes.

Empezamos creando un nuevo proyecto tipo **Biblioteca de Clases (.Net Framework)** (Class Library (.Net Framework)).

En VS podemos ver varios tipos de Librería de Clases, debemos seleccionar la .Net Framework cuyo objetivo es una dll solo compatible con sistemas Windows y es la que usaremos pues está basada en la librería Windows Forms que ya conocemos. Ten cuidado y selecciona además la de lenguaje C# por si tienes otros lenguajes instalados como VB.

En las últimas versiones de VS también existe un tipo de proyecto que es Biblioteca de Controles de Windows Forms, que tambien puede ser usado.

Creamos por tanto el nuevo proyecto usando el nombre **NuevosComponentes** para el espacio de nombres. Aparece una clase que podemos eliminar (Class1).

Para crear un nuevo *UserControl* usamos el asistente del Visual Studio: Sobre el proyecto, botón derecho, agregar Control de Usuario (User Control). Crearemos un componente denominado **LabelTextBox**.

Aparece una ventana de diseño similar a la del formulario pero solo con una región rectangular sin cabecera. Es donde podremos poner componentes existentes para crear el nuevo componente.

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM	
	MÓDULO	Desarrollo de Interfaces			CURSO: 2º
	PROTOCOLO:	Apuntes clases	AVAL:	DATA:	
	UNIDAD	COMPETENCIA			

Si vas al apartado de código debes tener algo similar a esto:


```
namespace NuevosComponentes
{
    public partial class LabelTextBox : UserControl
    {
        public LabelTextBox()
        {
            InitializeComponent();
        }
    }
}
```

Efectivamente como comentamos se hereda de *UserControl* y ahí se puede escribir su código asociado.

Aparece también un archivo Designer que es dónde VS realiza los cambios. La estructura es similar a la del formulario. Lo que no hay que perder de vista es que no estamos haciendo un programa final, si no un componente que luego se usará en uno o más formularios de diversas aplicaciones.

En la pestaña de diseño podemos añadir una label y una textbox con los nombres **lbl** y **txt** respectivamente.

Regresamos a la pestaña de código y empezamos a definir los elementos que nos interesan.

	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Desarrollo de Interfaces				CURSO: 2º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	UNIDAD	COMPETENCIA				

Definición de nuevas propiedades y atributos del IDE

Posición

Lo primero es crear la **propiedad de posicionamiento** de la label respecto al textbox para lo cual necesitaremos un enumerado por claridad.

```
//Indica donde está la etiqueta respecto al TextBox
public enum ePosicion
{
    IZQUIERDA, DERECHA
}
```

La definición anterior puede meterse en un archivo aparte o en el mismo que LabelTextBox.

Ya dentro de la clase LabelTextBox añadimos el siguiente código:

```
private ePosicion posicion = ePosicion.IZQUIERDA;

[Category("Appearance")]
[Description("Indica si la Label se sitúa a la IZQUIERDA o DERECHA del
Textbox")]
public ePosicion Posicion
{
    set
    {
        if (Enum.IsDefined(typeof(ePosicion), value))
        {
            posicion = value;
            recolocar();
        }
        else
        {
            throw new InvalidEnumArgumentException();
        }
    }
    get
    {
        return posicion;
    }
}

void recolocar() { }
```

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Desarrollo de Interfaces				CURSO: 2º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	UNIDAD	COMPETENCIA				

La función *recolocar()* no la tenemos definida, la veremos más adelante, pero será una función que se encargue de colocar *lbl* y *txt* dentro de nuestro nuevo componente en la situación adecuada.

Los elementos colocados **entre corchetes** son los denominados **atributos**. La utilidad de estos es indicar al IDE o al compilador distintos comportamientos. En este caso los atributos usados sirven para establecer la categoría donde me aparece la propiedad (Category) y una descripción para la ventana de propiedades (Description).

En el caso de Category, si se indica una no existente aparecerá como nueva categoría en el IDE. Prueba a cambiarla a "Clase".

El resto del código debería de comprenderse bien: si la posición es válida en el enumerado se guarda, si no se lanza una excepción.

Una vez que tenemos definidos los elementos básicos **vamos a probar** el componente. Para ello no podemos simplemente ejecutar, pues no hay nada que ejecutar (**no existe Main**). Crearemos un **proyecto nuevo** en esta solución que será una **aplicación WindowsForms**.

Con el botón derecho sobre el proyecto lo establecemos como proyecto de inicio.

Si no hemos compilado el componente le damos a Generar Solución para que podamos utilizarlo en la aplicación de test.

En la ventana de herramientas, si hemos compilado nuestro nuevo componente, nos aparecerá en la parte superior. Lo añadimos como si se tratara de otro componente cualquiera al formulario.

Echa un vistazo en la ventana de propiedades y ahí deben aparecer las propiedades definidas previamente. Además cada propiedad aparece en la categoría y con la descripción indicada en los atributos.

El enumerado *ePosicion* puede accederse si lo has metido en la clase *LabelTextBox* de la forma:

```
LabelTextBox1.Posicion = NuevosComponentes.LabelTextBox.ePosicion.DERECHA;
```

También puedes importar mediante `using NuevosComponentes` por lo que te ahorrarías ponerlo en el código.

Otra opción más cómoda es que el enumerado esté fuera de la clase, entonces pondrías:

```
LabelTextBox1.Posicion = NuevosComponentes.ePosicion.DERECHA;
```

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Desarrollo de Interfaces				CURSO: 2º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	UNIDAD	COMPETENCIA				

O solo `ePosicion.DERECHA` si importas con using el espacio de nombres.

Cada vez que se cambia algo en el control hay que recompilar para que esté disponible el funcionamiento en tiempo de diseño.

Si al colocar un componente creado desde el IDE en un formulario da un error de valor no válido debe crearse en tiempo de ejecución para ver dónde salta el error.

Separación

El siguiente fragmento define la separación **en píxeles** que hay entre *lbl* y *txt*.

```
//Píxeles de separación entre label y textbox
private int separacion = 0;

[Category("Design")]
[Description("Píxeles de separación entre Label y Textbox")]
public int Separacion
{
    set
    {
        if (value >= 0)
        {
            separacion = value;
            recolocar();
        }
        else
        {
            throw new ArgumentOutOfRangeException();
        }
    }
    get
    {
        return separacion;
    }
}
```

Nuevamente si el valor no fuera válido se lanza una excepción. Si es correcto se recolocan los elementos.

Ahora podemos compilar de nuevo y en nuestro formulario de prueba ver que todo está correcto. Por supuesto aún no hace nada ya que no hemos definido la función que recoloca los componentes.

<div>COLEXIO</div> <div>VIVAS</div> <div>S.L.</div>	RAMA:	Informática	CICLO:	DAM				
	MÓDULO	Desarrollo de Interfaces					CURSO:	2º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:			
	UNIDAD		COMPETENCIA					

Acceso a miembros privados de interés

En principio tanto **txt** como **lbl** son privados, sin embargo nos interesa permitir acceso a alguno de los campos. Una solución sería hacerlos públicos de forma que se pueda acceder a estos "subcomponentes" libremente, pero esto puede causar funcionamientos extraños si se juega con determinadas propiedades.

La solución que planteamos es crear propiedades públicas que den acceso a las propiedades de estos elementos privados. Las más claras son las **propiedades Text** de *lbl* y *txt* que definimos a continuación:

```
[Category("Appearance")]
[Description("Texto asociado a la Label del control")]
public string TextLbl
{
    set
    {
        lbl.Text = value;
        recolocar();
    }
    get
    {
        return lbl.Text;
    }
}
```

```
[Category("Appearance")]
[Description("Texto asociado al TextBox del control")]
public string TextTxt
{
    set
    {
        txt.Text = value;
    }
    get
    {
        return txt.Text;
    }
}
```

Es importante resaltar que **no definimos ninguna variable privada** para usar con estas propiedades TextT y TextL **porque ya existen** y son *txt.Text* y *lbl.Text* respectivamente.

En el set de *lbl* se llama a *recolocar()* porque puede cambiar el tamaño del *lbl* y por tanto el de *Label/TextBox*.

Si fuera de interés tendrías que hacer esto con otras propiedades como las de Color, Size, Font, etc.

<div>COLEXIO</div> <div>VIVAS S.L.</div>	RAMA:	Informática	CICLO:	DAM				
	MÓDULO	Desarrollo de Interfaces					CURSO:	2º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:			
	UNIDAD		COMPETENCIA					

Organización de los componentes

A continuación hacemos la función **recolocar()** que coloca los componentes *txt* y *lbl* según el enumerado y la separación indicados. El enumerado indica dónde se encuentra la Label respecto al TextBox.

Uno de los puntos importantes es resaltar donde se llama a funciones como esta.

```
private void recolocar()
{
    switch (posicion)
    {
        case ePosicion.IZQUIERDA:
            //Establecemos posición del componente lbl
            lbl.Location = new Point(0, 0);

            //Establecemos posición componente txt
            txt.Location = new Point(lbl.Width + Separacion, 0);

            //Establecemos ancho del Textbox
            //(la label tiene ancho por autosize)
            txt.Width = this.Width - lbl.Width - Separacion;

            //Establecemos altura del componente
            this.Height = Math.Max(txt.Height, lbl.Height);
            break;

        case ePosicion.DERECHA:
            //Establecemos posición del componente txt
            txt.Location = new Point(0, 0);

            //Establecemos ancho del Textbox
            txt.Width = this.Width - lbl.Width - Separacion;

            //Establecemos posición del componente lbl
            lbl.Location = new Point(txt.Width + Separacion, 0);

            //Establecemos altura del componente (Puede sacarse del switch)
            this.Height = Math.Max(txt.Height, lbl.Height);
            break;
    }
}

// Esta función has de enlazarla con el evento SizeChanged.
// Sería necesario también tener en cuenta otros eventos como FontChanged
// que aquí nos saltamos.
private void LabelTextBox_SizeChanged(object sender, EventArgs e)
{
    recolocar();
}
```

Es importante el orden de las instrucciones ya que podría causar funcionamientos extraños.

<div>COLEXIO</div> <div>VIVAS</div> <div>S.L.</div>	RAMA:	Informática	CICLO:	DAM				
	MÓDULO	Desarrollo de Interfaces					CURSO:	2º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:			
	UNIDAD		COMPETENCIA					

Como los componentes van a cambiar de tamaño al cambiar el tamaño de *Label/TextBox*, debemos usar el evento asociado para recolocar los componentes. Por tanto, en principio, hemos creado *Label/TextBox_SizeChanged* que debe asociarse a su evento.

Sin embargo sería más correcto **sobreescibir** el evento **OnSizeChanged** que es el que se encarga de lanzar el evento correspondiente.

Sería algo así:

```
protected override void OnSizeChanged(EventArgs e)
{
    base.OnSizeChanged(e);
    recolocar();
}
```

Se llama a la base para que lance el evento *SizeChanged* y por tanto las funciones suscritas a él. Si no estas no funcionarían. Puedes probar a comentar la llamada a base, luego en el proyecto de prueba escribe una línea de depuración en una función asociada al evento *SizeChanged* y verás que no se ejecuta. Si lo descomentas vuelve a ejecutarse. Tienes más información de este efecto en:

<https://social.msdn.microsoft.com/Forums/en-US/90367818-5284-4d01-8931-af4576dcaa63/baseonpainte-needed-why?forum=Vsexpressvcs>

Cuando se están creando nuevos componentes **se recomienda, si existe, sobreescibir** el evento **OnXxx** que corresponda en lugar de escribir el método asociado.

Los motivos son claridad en cuanto a jerarquía de herencia y además que se realizan las acciones sin necesidad de llamar a un evento.


Aunque aquí no lo hagamos habría que tener en cuenta otros eventos de cambio de tamaño como *FontChanged*.

Finalmente en el **constructor** o en el **evento Load** del componente inicializamos todo lo que nos interese. En este caso puede quedar así:

```
public LabelTextBox()
{
    InitializeComponent();
    TextLbl = Name;
    TextTxt = "";
    recolocar();
}
```

Además lo que programemos en el constructor se ejecuta en el IDE cuando colocamos el componente.

Vuelve a compilar todo y ve a la aplicación de test. Prueba que todo funcione.

	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Desarrollo de Interfaces				CURSO: 2º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	UNIDAD	COMPETENCIA				

Más sobre atributos

Para seguir jugando con atributos podemos establecer los siguientes justo antes de la declaración de la clase (antes de class):

```
[
    DependencyProperty("TextLbl"),
    DefaultEvent("Load")
]
```

Recompila la clase y ve a la aplicación de test. Al acceder a las propiedades accede por defecto a la propiedad TextL.

También si haces doble click sobre el componente automáticamente crea la función asociada al evento Load.

Dispones de más información y atributos en:


[https://msdn.microsoft.com/es-es/library/tk67c2t8\(v=vs.120\).aspx](https://msdn.microsoft.com/es-es/library/tk67c2t8(v=vs.120).aspx)

[https://msdn.microsoft.com/es-es/library/ms171724\(v=vs.120\).aspx](https://msdn.microsoft.com/es-es/library/ms171724(v=vs.120).aspx)

<https://msdn.microsoft.com/en-us/library/a19191fh.aspx>

También si deseas ver otros atributos interesantes no solo para el diseño de componentes puedes verlo en:

<http://stackoverflow.com/questions/144833/most-useful-attributes>

	RAMA:	Informática	CICLO:	DAM				
	MÓDULO	Desarrollo de Interfaces					CURSO:	2º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:			
	UNIDAD		COMPETENCIA					

Más sobre Eventos

En temas anteriores vimos como manejar eventos que ya están programados, esencialmente vimos como añadir o eliminar funciones asociadas a eventos.

También hemos entendido el uso de `OnXxx` cuando estamos heredando de la clase `UserControl` o `Control` para gestionar eventos heredados.

En este apartado vamos a aumentar nuestro conocimiento de estos objetos. Veremos primero como llamar de un evento a otro (es decir, lanzar un evento) y como crear nuestros propios eventos cuando nos interese.

Lanzar eventos

Para lanzar un evento de algunos componente se usan los métodos **`OnXxx`** siendo `Xxx` el evento a lanzar. Es necesario pasarle un gestor de eventos (`EventHandler` o derivado) como parámetro.

Esto podemos usarlo para enlazar los eventos visibles de nuestro nuevo control con los ocultos en los controles internos `txt` y `lbl`.

Es decir, si el programador necesita utilizar el `KeyPress` del `TextBox` no puede hacerlo directamente, porque sólo tiene acceso al `KeyPress` del `LabelTextBox`. Sin embargo el usuario al pulsar la tecla provoca el lanzamiento del `KeyPress` del `TextBox`.

La solución es que internamente (como código de la clase `LabelTextBox`) se llame desde el evento generado por la `TextBox` al de la `LabelTextBox` pasándole el `KeyEventArgs`.

En la clase `LabelTextBox` crea el método asociado con el `KeyPress` del textbox `txt` y escribe el código tal que así:


```
private void txt_KeyPress(object sender, KeyPressEventArgs e)
{
    this.OnKeyPress(e);
}
```

De esta manera, el usuario al pulsar una tecla lanza el evento `KeyPress` del `TextBox`, y la función anterior provoca que se lance el evento `KeyPress` del `LabelTextBox`.

Ahora en el formulario de prueba crea un método asociado con el `KeyPress` del objeto `LabelTextBox` y escribe el código:

```
private void labelTextBox1_KeyPress(object sender, KeyPressEventArgs e)
{
    this.Text = "Letra: " + e.KeyChar;
}
```

Prueba a comentar la línea del `OnKeyPress` y verás que, lógicamente, deja de funcionar.

	RAMA:	Informática	CICLO:	DAM				
	MÓDULO	Desarrollo de Interfaces					CURSO:	2º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:			
	UNIDAD		COMPETENCIA					

Crear eventos

Vamos a profundizar un poco más en la realización de eventos. Sabemos que esta es una manera sencilla de comunicación entre objetos, veamos como funciona realmente.

Un objeto quiere avisar de algún hecho a otros objetos, para ello esos otros objetos deben estar "suscritos" a un evento del primer objeto.

La suscripción no es más que indicar qué método se ha de ejecutar cuando se lanza un evento. Lo hemos visto innumerables veces cuando asignamos funciones a eventos en componentes *WindowsForms*, eso es la suscripción.

Supongamos un método que se encarga de hacer una operación larga como calcular si un número es primo. Nos interesa que se lance como un hilo y que cuando acabe nos avise pero con un evento en lugar de usar sincronización vista. Al recibir el evento se mostrará el resultado en el programa principal (En el [Apéndice I](#) puedes ver una solución a este caso).

El evento tiene la siguiente forma cuando se define:

```
event TipoDelegado NombreEvento;
```

TipoDelegado es el delegado que define la signature (el aspecto) que debe tener el método asociado al evento, y *NombreEvento* es precisamente el nombre que queremos que tenga el evento. *TipoDelegado* puede ser algo así:

```
delegate void TipoDelegado(parametros);
```

Para suscribirse desde otro (u otros) objeto/s al evento, hay que realizar la operación ya conocida:

```
objetoOrigen.NombreEvento += new TipoDelegado(métodoAsociado);
```

donde *métodoAsociado* es un método que cumple la signature de *TipoDelegado* y que será ejecutado cuando el objeto origen lance el evento que se hace simplemente escribiendo el nombre del evento como si fuera un método y pasándole como parámetro el indicado por el delegado.

```
NombreEvento(parametros);
```

Nota: Un evento puede parecer muy similar a un delegado pero realmente, además de la diferencia conceptual (un delegado es una plantilla para una función y un evento es un "informador de sucesos") el evento crea una capa de abstracción más de seguridad ya que solo permite añadir y quitar funciones una a una por no aceptar el operador asignación =, solo añadir función += y eliminar función -=. Esto último se puede ver en la **2ª respuesta** de esta entrada:

<http://stackoverflow.com/questions/29155/what-are-the-differences-between-delegates-and-events>

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM	
	MÓDULO	Desarrollo de Interfaces			CURSO: 2º
	PROTOCOLO:	Apuntes clases	AVAL:	DATA:	
	UNIDAD	COMPETENCIA			

Nota 2: La realización de eventos en Java puede hacerse de diversas formas. Quizá la más habitual sea mediante la creación de listener. Tienes un ejemplo claro en el siguiente enlace:

<http://stackoverflow.com/questions/6270132/create-a-custom-event-in-java>

Nuevo evento en LabelTextBox

Aplicando lo anterior, se puede crear nuevos eventos desde cero para los componentes.

Por ejemplo puede ser interesante tener un evento que se lance si se detecta un cambio de posición para *lbl* respecto de *txt* de forma que se puedan cambiar la distribución de otros componentes en el formulario si fuera necesario.

Nos podemos ahorrar la creación del delegado y usar directamente *EventHandler*, por lo que el evento quedaría (importante hacerlo público):

```
[Category("La propiedad cambió")]
[Description("Se lanza cuando la propiedad Posición cambia")]
public event System.EventHandler CambiaPosicion;
```

Se puede observar que la categoría se puede especificar en castellano (de hecho se puede aplicar localización como un string cualquiera).

Y sólo habría que cambiar el set de **Posición** de la siguiente forma:


```
set //Posicion
{
    if (Enum.IsDefined(typeof(ePosicion), value))
    {
        posicion = value;
        recolocar();
        if (CambiaPosicion!=null)
        {
            CambiaPosicion(this, new EventArgs());
        }
    }
    else
    {
        throw new InvalidEnumArgumentException();
    }
}
```

Se comprueba que *CambiaPosición* tenga algún método asociado con el *if*. Esto se puede hacer más compacto desde la versión 6.0 de C# escribiéndolo así:

```
CambiaPosicion?.Invoke(this, EventArgs.Empty);
```

Esto vale para comprobar automáticamente si un objeto es null.

En ocasiones es necesario recompilar toda la solución para ver los eventos en la ventana de Propiedades y eventos.

	RAMA:	Informática	CICLO:	DAM				
	MÓDULO	Desarrollo de Interfaces					CURSO:	2º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:			
	UNIDAD		COMPETENCIA					

Ejercicio 1:

a) Prueba lo anterior poniendo en la aplicación de prueba un botón que fuerce el cambio de la posición y que aparezca el valor del enumerado en la barra de título del formulario pero cuando se produce el cambio de posición (Usando el vento creado).

b) Haz lo mismo con la propiedad Separacion creando un evento que se lance cuando esta propiedad cambie y haz una prueba de funcionamiento.

c) Haz que el evento KeyUp del LabelTextbox sea lanzado cuando suceda un evento KeyUp de txt. Haz prueba de funcionamiento.

d) Finalmente crea un evento en LabelTextbox denominado TxtChanged el cual sea lanzado cuando suceda el evento TextChanged del textbox interno.

e) Haz una nueva propiedad de LabelTextbox denominada PswChr que enlace con la propiedad PasswordChar del Textbox interno.

f) Cambia la funcionalidad de recolocación para que en lugar de que txt cambie de tamaño al cambiar la label, se mantenga siempre del mismo tamaño y cambie la anchura global del LabelTextBox.

(Nota: Si acabas este ejercicio puedes hacer el 3)

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Desarrollo de Interfaces				CURSO: 2º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	UNIDAD	COMPETENCIA				

Dibujando en 2D: GDI+

En este apartado aprenderemos a realizar dibujos directamente sobre la superficie de un componente.

Ya no solo a través de lo que nos permitan sus propiedades, si no que lo trataremos como un "lienzo" y pintaremos los pixels que nos interese.

Todo ello lo haremos a través de la librería denominada GDI+ (Graphical Device Interface Plus) la cual permite dibujar en tres tipos de superficie:

- La pantalla: Es decir, a la hora de dibujar cualquier cosa en un componente que esté en pantalla.
- Imágenes: Se puede dibujar sobre una imagen aunque no esté en pantalla.
- Impresora.

La forma en que el sistema dibuja es jerárquico por contenedor padre hijo. Si en un formulario hay un panel y dentro un botón, primero se dibuja el formulario, luego el panel y finalmente el botón.

Los espacios de nombres del GDI+ son:

System.Drawing : Clases genéricas de dibujo.

System.Drawing.Drawing2D : Dibujo de formas complejas.

System.Drawing.Imaging: dibujo y manipulación de imágenes.

System.Drawing.Printing: Trabajos sobre la impresora (imprimir, vistas previas, etc...)

System.Drawing.Text: Manipulación de texto a nivel de dibujo y codificación..

System.Drawing.Design: Clases para el manejo del diseñador gráfico del IDE.

Lienzo (Graphics), evento Paint y otros elementos básicos

Para pintar en un componente, realmente se hace sobre lo que sería su "lienzo" que se denomina objeto **Graphics** del componente.

En el evento Paint es donde se debe dibujar, ya que es el evento que se llama automáticamente cada vez que es necesario dibujar el control.

Prueba en un formulario el siguiente método asociado al Paint:

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    e.Graphics.DrawString("Prueba de escritura de texto",
                          this.Font, Brushes.BlueViolet, 10, 10);
}
```

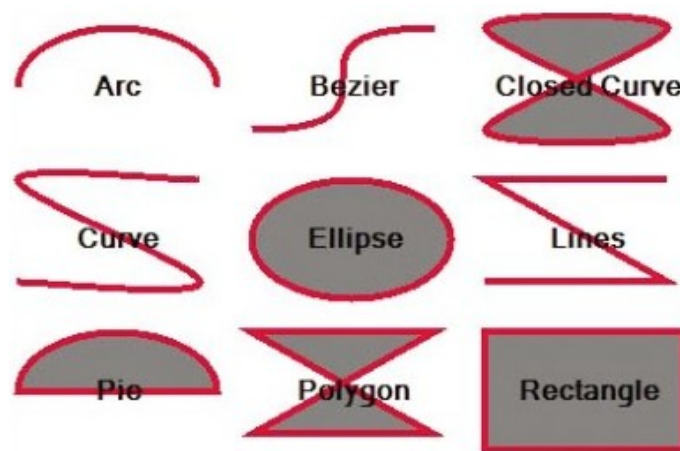
El método **DrawString**, en esta sobrecarga, dispone como parámetros el texto a dibujar, la fuente que ha de usar (coge en este caso la definida en el formulario), el color (más bien la brocha, lo veremos luego) y las coordenadas donde colocarlo.

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM	
	MÓDULO	Desarrollo de Interfaces			CURSO: 2º
	PROTOCOLO:	Apuntes clases	AVAL:	DATA:	
	UNIDAD	COMPETENCIA			

Si fuese necesario obtener el lienzo de un componente cualquier para dibujar en él, se puede usar el método:

```
componente.CreateGraphics(); //Devuelve un objeto tipo Graphics.
```

Cuando se accede al objeto Graphics puedes ver que tiene múltiples métodos para dibujar líneas, arcos, polígonos, figuras rellenas (fill), imágenes, iconos, etc.



En cualquier caso sobre todo cuando se están creando nuevos componentes se recomienda sobrescribir el evento **OnPaint** en lugar de escribir el método asociado al evento *Paint*. Este último caso se reserva para pintar sobre componentes en la aplicación final.

Cambia el ejemplo anterior por el siguiente:

```
protected override void OnPaint(PaintEventArgs e)
{
    base.OnPaint(e);
    e.Graphics.DrawString("Prueba de escritura de texto",
        this.Font, Brushes.BlueViolet, 10, 10);
}
```

Recuerda que se llama al *OnPaint* del base ya que este es el encargado de lanzar el evento *Paint* cuando el componente ha de repintarse, si no las funciones suscritas a él no serán ejecutadas.

La **invalidación** de un componente se produce cuando debe ser redibujado (Se tapa y vuelve a la vista, cambia su aspecto, etc...). Esto se dispara automáticamente.

Los métodos *Invalidate()*, *Update()* o el *Refresh()* del control lanza el evento *Paint* cuando sea necesario. Se puede invalidar una región pasándola como parámetro.

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM	
	MÓDULO	Desarrollo de Interfaces			CURSO: 2º
	PROTOCOLO:	Apuntes clases	AVAL:	DATA:	
	UNIDAD	COMPETENCIA			

Añade un botón al formulario y cambia el código de la siguiente forma:

```
bool bandera = true;
protected override void OnPaint(PaintEventArgs e)
{
    base.OnPaint(e);
    if (bandera)
    {
        e.Graphics.DrawString("Prueba de escritura de texto",
                               this.Font, Brushes.BlueViolet, 10, 10);
    }
}

private void button1_Click(object sender, EventArgs e)
{
    bandera = !bandera;
    this.Refresh();
}
```

Prueba a comentar el Refresh. Verás que no se actualiza el texto. Cuando en teoría esté la bandera a false, prueba a “esconder” un poco el formulario fuera de la pantalla, al volver a traerlo solo verás una parte del texto.

Respecto a los métodos de invalidación, los tres son muy similares y dependiendo del caso es mejor usar unas u otras. No vamos a entrar en las diferencias que son bastante sutiles. Puedes leer más en:

<http://blogs.msdn.com/b/subhagpo/archive/2005/02/22/378098.aspx>

Normalmente para el uso que damos nosotros lo que mejor funciona es el **Refresh()**, ya que se encarga de ejecutar los otros dos. Concretamente el *Invalidate* simplemente invalida una región a la espera que se lance un WM_PAINT (mensaje de windows para repintar). El Update lanza el WM_PAINT sobre las zonas invalidadas.

Es necesario usar los eventos al pintar porque si no se pierde el dibujo si no se repinta. Veámoslo con un ejemplo: Toma el caso anterior, comenta o elimina el código del OnPaint y escribe el siguiente código dentro del click del botón:

```
Graphics gr = this.CreateGraphics();
gr.DrawString("Escribo fuera del OnPaint", this.Font, Brushes.BlueViolet, 10, 10);
gr.DrawImage(new Bitmap(@"C:\Windows\Web\Wallpaper\Theme2\img7.jpg"), 10, 30);
```

Prueba a cambiar de tamaño el formulario o ponerle otros encima.

Es más, si le añades al final un *Refresh* parece que no pinta nada porque luego se repinta solo lo que hay en el *OnPaint*.

<div>COLEXIO</div> <div>VIVAS S.L.</div>	RAMA:	Informática	CICLO:	DAM				
	MÓDULO	Desarrollo de Interfaces					CURSO:	2º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:			
	UNIDAD		COMPETENCIA					

Adornando LabelTextBox

Vamos a subrayar la *lbl* de LabelTextBox. Para ello además vamos a cambiar la estructura pues al sobrescribir OnPaint ya no es necesario llamar a recolocar en las distintas situaciones si no que recolocamos solamente justo antes de pintar.

Empieza borrando por tanto todas las llamadas a *recolocar()*.

Luego sobrescribimos **OnPaint** (si empiezas escribiendo override y pulsas CTRL+Espacio te aparecen las posibilidades de sobreescritura).

Y es ahí donde llamamos a *recolocar()*.

```
protected override void OnPaint(PaintEventArgs e)
{
    base.OnPaint(e);
    recolocar();
}
```

Si ahora lo pruebas verás que en ciertos casos funciona bien (por ejemplo al cambiar el texto de la Label) pero otras no actualiza (como al cambiar la Separación o Posición). Esto se debe a que al cambiar algunas propiedades como el *TextLbl* la propia *Label* fuerza un refresh, sin embargo al cambiar la Separación no es así pues es una propiedad nueva que hemos añadido. La solución está en **añadir el Refresh solo en estas propiedades nuevas**.

Por ejemplo el set de **Separacion** quedaría:

```
set //Separación
{
    if (value >= 0)
    {
        separacion = value;
        Refresh();
    }
    else
    {
        throw new ArgumentOutOfRangeException();
    }
}
```

Habría que cambiar de forma similar el de **Posicion**.

Finalmente vamos a ampliar el OnPaint para que **subraye lbl**. Quedaría algo así:

```
protected override void OnPaint(PaintEventArgs e)
{
    base.OnPaint(e);
    recolocar();
    e.Graphics.DrawLine(new Pen(Color.Violet),
        lbl.Left, this.Height - 1,
        lbl.Left + lbl.Width, this.Height - 1);
} // Left equivale a Location.X
```

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Desarrollo de Interfaces				CURSO: 2º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	UNIDAD	COMPETENCIA				

La clase Control

Vamos a ver algún ejemplo de dibujo creando un componente de cero sin usar el diseñador. En este caso vamos a derivar de la clase *Control* ya que no usaremos ningún control predefinido.

Por supuesto este apartado se podría hacer entero usando la clase *UserControl*, simplemente lo vemos como ejemplo de realización de un componente partiendo de la clase base.

Para ello lo más sencillo es agregar un Control Personalizado (Custom Control) o si no puedes crear un nuevo *UserControl* como hicimos en los apartados anteriores y luego cambiar la herencia a *Control* y eliminar las líneas de error (que son propiedades de *UserControl* pero no de *Control*).

Lo denominaremos *EtiquetaAviso* y será una etiqueta que dispone de varios efectos de presentación.


Lo primero será escribir el campo *Text* en la propia Etiqueta. Para ello sobreescribimos el método *OnPaint* y nos encargamos de que aparezca el texto de la siguiente forma:

```
protected override void OnPaint(PaintEventArgs pe)
{
    base.OnPaint(pe);
    Graphics g = pe.Graphics;
    SolidBrush b=new SolidBrush(this.ForeColor);
    g.DrawString(this.Text, this.Font, b, 0, 0);
}
```

Aquí la novedad es *SolidBrush*. *Brushes* lo puedo usar pero es un enumerado predefinido. *SolidBrush* me da una brocha sólida (sin texturas ni gradientes) del color indicado en el constructor.

Otras brochas que puedes probar son *LinearGradientBrush*, *HatchBrush*, *TextureBrush*, *PathGradientBrush*,... Algunas están definidas en el espacio de nombres *Drawing2D*.



	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Desarrollo de Interfaces				CURSO: 2º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	UNIDAD	COMPETENCIA				

El problema del código anterior es que tenemos que cambiar de forma manual el tamaño. Si quieres automatizarlo se puede añadir al código anterior la siguiente línea:

```
this.Size = g.MeasureString(this.Text, this.Font).ToSize();
```

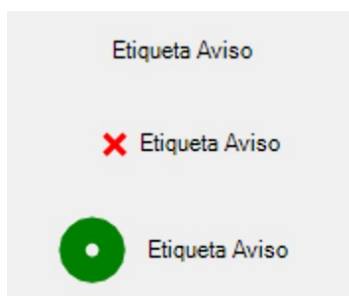
MeasureString devuelve un tipo *SizeF* (tamaño en float en lugar de int) que indica la región que ocupa la cadena de texto que tiene como parámetro. Lo convertimos a *Size* para obtener de forma automática el tamaño de nuestra etiqueta.

Un problema que puedes observar es que al cambiar la propiedad en la ventana de properties de VS no cambia directamente hasta que pulsamos en el formulario. Esto es un problema menor pues afecta solo al IDE, pero si quieres evitarlo se puede utilizar el **Refresh**:


- Si es una propiedad que hayamos definido nosotros simplemente se hace el refresh en el set.
- Si es una propiedad ya definida, como en el caso de *Text*, hay que provocar un Refresh cuando cambia dicha propiedad. Lo habitual es que todas las propiedades tengan un evento asociado a su cambio, y por tanto se puede sobrescribir el método *OnPropiedadChanged*. En este caso **OnTextChanged**:

```
protected override void OnTextChanged(EventArgs e)
{
    base.OnTextChanged(e);
    this.Refresh();
}
```

Ahora completaremos *EtiquetaAviso* con la posibilidad de que antes del texto aparezca una cruz o un círculo de "adorno". El ajuste de los elementos se puede mejorar pero sirve de forma ilustrativa.



Esta parte es ya repaso de distintas cosas que hemos realizado en apartados previos por lo que debería entenderse bien todo.

	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Desarrollo de Interfaces				CURSO: 2º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	UNIDAD	COMPETENCIA				

Lo primero que hará falta es definir el enumerado:

```
public enum eMarca
{
    Nada,
    Cruz,
    Circulo
}
```

Que nos indicará si queremos que aparezca un círculo una cruz o nada junto al texto tal y como se ve en la imagen previa.

Como siempre el enumerado se recomienda definirlo fuera de la clase (incluso en archivo aparte). Dependiendo de donde se defina puede cambiar la forma de acceso al mismo.

Luego definimos la propiedad (dentro de la clase):

```
private eMarca marca = eMarca.Nada;

[Category("Appearance")]
[Description("Indica el tipo de marca que aparece junto al texto")]
public eMarca Marca
{
    set
    {
        marca = value;
        this.Refresh();
    }

    get
    {
        return marca;
    }
}
```

donde guardaremos precisamente el enumerado anterior.

Lanzamos el Refresh en el set ya que se desea que se pinte alguna de las marcas (o nada) en cuanto esta propiedad cambie.

<div>COLEXIO</div> <div>VIVAS S.L.</div>	RAMA:	Informática	CICLO:	DAM					
	MÓDULO	Desarrollo de Interfaces						CURSO:	2º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:				
	UNIDAD		COMPETENCIA						

Finalmente **sobreescribimos el OnPaint** de la siguiente manera:

```
protected override void OnPaint(PaintEventArgs e)
{
    base.OnPaint(e);

    Graphics g = e.Graphics;
    int grosor = 0;      //Grosor de las líneas de dibujo
    int offsetX = 0;     //Desplazamiento a la derecha del texto
    int offsetY = 0;     //Desplazamiento hacia abajo del texto
    // Altura de fuente, usada como referencia en varias partes
    int h = this.Font.Height;

    //Esta propiedad provoca mejoras en la apariencia o en la eficiencia
    // a la hora de dibujar
    g.SmoothingMode = System.Drawing.Drawing2D.SmoothingMode.AntiAlias;

    //Dependiendo del valor de la propiedad marca dibujamos una
    //Cruz o un Círculo
    switch (Marca)
    {
        case eMarca.Círculo:
            grosor = 20;
            g.DrawEllipse(new Pen(Color.Green, grosor), grosor, grosor,
                h, h);
            offsetX = h + grosor;
            offsetY = grosor;
            break;

        case eMarca.Cruz:
            grosor = 3;
            Pen lapiz = new Pen(Color.Red, grosor);
            g.DrawLine(lapiz, grosor, grosor, h, h);
            g.DrawLine(lapiz, h, grosor, grosor, h);
            offsetX = h + grosor;
            offsetY = grosor / 2;
            //Es recomendable liberar recursos de dibujo pues se
            //pueden realizar muchos y cogen memoria
            lapiz.Dispose();
            break;
    }

    //Finalmente pintamos el Texto; desplazado si fuera necesario
    SolidBrush b = new SolidBrush(this.ForeColor);
    g.DrawString(this.Text, this.Font, b, offsetX + grosor, offsetY);
    Size tam = g.MeasureString(this.Text, this.Font).ToSize();
    this.Size = new Size(tam.Width+offsetX+grosor, tam.Height+offsetY*2);
    b.Dispose();
}
```

Las explicaciones se encuentran en el propio código, aunque muchas de las operaciones son simplemente de ajuste de tamaños y posiciones.

En el caso de las sobrecargas usadas para *DrawEllipse* y *DrawLine*, el primer

<div>COLEXIO</div> <div>VIVAS S.L.</div>	RAMA:	Informática	CICLO:	DAM				
	MÓDULO	Desarrollo de Interfaces					CURSO:	2º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:			
	UNIDAD		COMPETENCIA					

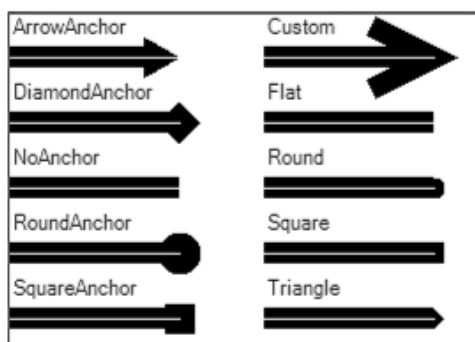
parámetro es el "lapiz" con el que queremos pintar. A continuación vienen dependiendo del caso coordenadas origen y tamaño (Círculo) o coordenadas origen y fin (Línea)

El OnPaint es un método al que se le llama muy a menudo por lo que es recomendable liberar los objetos (mediante Dispose) que se crean en él.

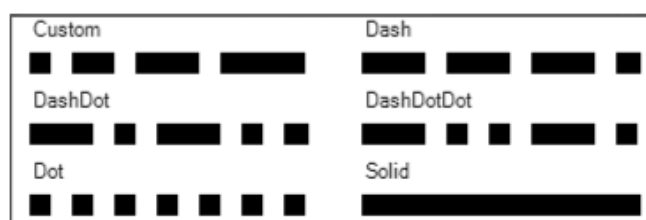
Se recomienda cambiar el tamaño de fuente a 16 para ver mejor los elementos.

El lápiz (clase Pen) tiene varias enumeraciones que permiten cambiar su estilo.

LineCap:



DashStyle:



COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Desarrollo de Interfaces				CURSO: 2º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	UNIDAD	COMPETENCIA				

Transformaciones

Son cambios de traslación, escalado y rotación sobre el sistema de coordenadas, de forma que lo que se dibuje a partir de ese momento y hasta que se haga un reset de la transformación se hará sobre el sistema de coordenadas transformado.

Veamos un ejemplo simple sobre un formulario:

```
protected override void OnPaint(PaintEventArgs e)
{
    base.OnPaint(e);
    Graphics g = e.Graphics;

    //Traslación
    g.TranslateTransform(100, 100);
    g.DrawLine(Pens.Red, 0, 0, 100, 0);
    g.ResetTransform();

    //Rotación de 30º en sentido horario
    g.RotateTransform(30);
    g.DrawLine(Pens.Blue, 0, 0, 100, 0);
    g.ResetTransform();

    //Traslación + rotación
    g.TranslateTransform(100, 100);
    g.RotateTransform(30);
    g.DrawLine(Pens.Green, 0, 0, 100, 0);
    g.ResetTransform();
}
```

Ten en cuenta que las transformaciones son sobre el sistema de coordenadas, no sobre la figura.

Mediante matrices de transformación se pueden hacer transformaciones más complejas usando el método MultiplyTransform y Transform. No los veremos pues son necesarios conocimiento de álgebra matricial para entenderlo bien. Más información en:

<https://docs.microsoft.com/es-es/dotnet/desktop/winforms/advanced/coordinate-systems-and-transformations?view=netframeworkdesktop-4.8>

<https://docs.microsoft.com/es-es/dotnet/desktop/winforms/advanced/matrix-representation-of-transformations?view=netframeworkdesktop-4.8>

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Desarrollo de Interfaces				CURSO: 2º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	UNIDAD	COMPETENCIA				

Otros elementos de trabajo

En este apartado resumiremos otras técnicas, clases y métodos a la hora de crear nuevos componentes.

La clase ControlPaint

Esta es una clase que tiene una serie de métodos estáticos para recrear el aspecto (no la funcionalidad) de algunos componentes estándar en sus distintos estados.

Puedes crear botones pulsados, sin pulsar, checkboxes marcados, botones de scroll, etc...

Es útil para realizar componentes similares a los que hay pero con comportamientos notablemente distintos.

```
protected override void OnPaint(PaintEventArgs e)
{
    base.OnPaint(e);

    Graphics g = e.Graphics;
    ControlPaint.DrawButton(g, 0, 40, 100, 60, ButtonState.Checked);
}
```

Puedes observar que al poner el punto tras ControlPaint salen un montón de posibilidades.

Modificando componentes existentes


Una forma típica y sencilla de crear un componente muy parecido a uno existente es heredarlo y modificarlo a nuestras necesidades.

Compartiendo componentes entre aplicaciones

Cuando hemos hecho una librería con componentes, el resultado (lo puedes ver en el directorio Debug) es una DLL.

Cuando quieras usar los componentes que vienen en dicha DLL simplemente en el nuevo proyecto, en la caja de herramientas encima del apartado de interés, por ejemplo Todo Windows Forms, le das con el botón derecho a Elegir elementos y en Examinar de Componentes de .Net añadís la DLL de interés.

Aparecen los componentes de la librería que se pueden usar en la nueva aplicación al final de la lista (se puede ordenar alfabéticamente con el botón derecho).

	RAMA:	Informática	CICLO:	DAM				
	MÓDULO	Desarrollo de Interfaces					CURSO:	2º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:			
	UNIDAD		COMPETENCIA					

En algunos de los siguientes ejercicios el enunciado indica exclusivamente la realización del componente. Debe incluirse un programa que **permita probar al usuario todas capacidades** del componente.

Además en todos los ejercicios donde se creen nuevas propiedades y eventos, estas deben llevar los atributos **Category y Description**. Y las clases que definen componentes deben tener **DefaultEvent y DefaultProperty**.

Ejercicio 2:

Toma el ejemplo **EtiquetaAviso** visto en teoría y realízalo de forma que se le añadan las siguientes características:

- Que pueda tener de forma opcional un fondo de gradiente entre dos colores. Establece los colores inicial y final del gradiente así como una booleana que indique si hay o no gradiente como nuevas propiedades.
- El enumerado eMarca tendrá una constante más denominada Imagen de forma que si se selecciona se colocará en la posición de la marca la imagen indicada en la propiedad imagenMarca. Si en dicha propiedad no hay nada (o la imagen no es válida) se comportará como Nada.
- Crea un evento denominado ClickEnMarca que será lanzado cuando el usuario pulsa el ratón pero solo en la zona donde está la marca (salvo que sea Nada).


Ejercicio 3:

Diseño del componente: El siguiente párrafo debe definirse todo en la clase de nuevo compoente (tipo UserControl)

Realiza un control que simule un **reproductor multimedia Simple**. Tendrá un botón único Play/Pausa que cada vez que se pulsa cambia su símbolo. Además lanzará el evento de pulsación del mismo. Tendrá también una etiqueta donde aparecerá tiempo en formato MM:SS (os digitos por cada valor). Tendrá propiedades separadas para MM (minutos) y SS (Segundos). Si SS es mayor que 59 se vuelve a poner al valor de resto entre 60. Si MM es mayor que 99 se pone a 0. Crea el evento DesbordaTiempo que se lanza cada vez SS supera 59.

Aplicación: A continuación están las especificaciones para la aplicación que usa el componente.

Para probarlo haz un programa que permita seleccionar un directorio y haga una presentación de las imágenes (1 por segundo) al darle al botón play/pausa para o continua la presentación. En la etiqueta de tiempo indica el tiempo transcurrido en minutos y segundos. Actualiza MM en el evento DesbordaTiempo. Si acaba las imágenes del directorio vuelve a la primera (el tiempo sigue avanzando, no se reinicia).

	RAMA:	Informática	CICLO:	DAM				
	MÓDULO	Desarrollo de Interfaces					CURSO:	2º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:			
	UNIDAD		COMPETENCIA					

(Opcional) Amplia e reproductor para que tenga además un modo completo. En dicho modo existen los botones Play, Pausa, Stop, Siguiente, Anterior, Avance y Retroceso. Tendrá también una barra que puede ir creciendo (para simular el avance de la reproducción) y una etiqueta donde se puede colocar un tiempo de reproducción y otra tiempo restante.

Dispondrá también de un listbox que funcionará como lista de reproducción.

Debes generar los eventos necesarios para los distintos botones.

Se lanzará un evento cuando se cambia de modo Simple a Completo y viceversa.

Para probarlo retoma el visor de imágenes anterior: Play hace una presentación, avance rápido o lento acelera o retrasa la presentación siguiente o anterior avanza o retrocede una imagen. En la lista de reproducción se ven los nombres de las imágenes. Al seleccionar una se salta directamente a ella. En la barra de progreso se muestra el porcentaje de imágenes mostradas. Cuando acaba de mostrar las imágenes vuelve a empezar.

Ejercicio 4:

Crear un componente **DibujoAhorcado** que disponga de una propiedad entera denominada errores. Dependiendo del número que contenga dibujará más o menos partes del ahorcado (el juego clásico). El componente debe disponer de un evento CambiaError que se lanza cada vez que cambia el número de errores y otro denominado Ahorcado que se lanza cuando se completa el dibujo.

Comprueba que funciona correctamente y úsalo para el cliente del programa del Ahorcado del tema de Networking.


Ejercicio 5:

Se creará un nuevo componente denominado **ValidateTextBox** que herede de UserControl y con estas características:

a) Dispondrá de un TextBox colocado en la posición 10,10 del nuevo componente. De esta forma le quedará un margen para la posibilidad de que quede rodeado por un recuadro de color.

El alto del componente siempre será del tamaño del textbox interno +20 pixels y el textbox será del ancho del componente -20 pixels.

b) Habrá propiedades en el nuevo componente para acceder al Text y a Multiline en el textbox interno. Se denominarán Texto y Multilinea respectivamente.

	RAMA:	Informática	CICLO:	DAM				
	MÓDULO	Desarrollo de Interfaces					CURSO:	2º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:			
	UNIDAD		COMPETENCIA					

c) Tendrá una propiedad pública (denominada tipo) que será un enumerado (denominado eTipo) con los valores:

- Numérico (Números enteros: sólo son válidos los dígitos y espacios en los extremos)
- Textual (No admitirá nada que no sea una serie de letras o espacios)

d) El componente estará rodeado por un recuadro inicialmente rojo (pintado mediante el GDI+) desde la posición x=5, y=5 hasta el ancho-5 y el alto-5 del componente.

e) El rectángulo pasará a verde si el contenido es correcto según el enumerado y rojo cuando no lo es.

f) Debe haber acceso al evento TextChange del textBox desde el nuevo componente. Aunque no se use dicho evento no debe saltar excepción.

Ejercicio 6:

a) Realiza un componente "gráfico de barras". Dispondrá de una colección de valores a partir de la cual dibuja un gráfico de barras con las siguientes características:

- Cada barra será de un color (se alternan) verde, azul y amarillo
- El tamaño de los ejes será manual o automático. Si es automático las barras se normalizarán a partir del valor máximo. Si es manual se establece un máximo en el ejeY y si la barra es mayor será del mismo tamaño que el eje pero se dibujará de color rojo.
- Tendrá dos textos que informan lo que es el eje X y el eje Y. El del ejeY debe estar escrito en vertical (usa transformaciones para girarlo y recolocar).


b) Amplía el componente anterior añadiendo un una propiedad con enumerado: BARRAS, LINEA de forma que al cambiar de una a otra cambie de gráfico de barras a gráfico de líneas. Usa en este caso una línea del color indicado en la propiedad Forecolor del componente.

c) (Se valida por separado) Realiza un programa que permita representar diversos gráficos de barras en pantalla a partir de un archivo.

En el archivo se guardará la información de varios gráficos. Así por cada gráfico debe almacenarse: Nombre del gráfico, texto del eje X, texto del eje Y, datos de las barras. Hazlo con un archivo Json o XML o ambos (busca información de cómo gestionarlo).

El programa tendrá un menú con opciones Abrir, Configurar un separador y Salir. Al darle a Abrir sale un cuadro de diálogo que permite seleccionar el archivo (haz los filtros que consideres, debes incluir todos los archivos).

Si se le da a aceptar lee el archivo y crea tantos gráficos de barras como sea necesario en varias filas y columnas según esté el programa configurado mostrando

	RAMA:	Informática	CICLO:	DAM				
	MÓDULO	Desarrollo de Interfaces					CURSO:	2º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:			
	UNIDAD		COMPETENCIA					

los datos. Por cada gráfico debes añadir una etiqueta con el título del gráfico.

Al pulsar Configuración sale un formulario secundario con diversas opciones de configuración (estas deben guardarse también en un archivo en %APPDATA%). Un combobox permite seleccionar el número de columnas por cada fila de gráficos. Permitirá entre 1 y 5. Dos radiobutton permitirán ver los gráficos como barras o línea. Un ListBox permitirá escoger entre al menos 5 colores distintos para la línea (si está en modo barra el listbox está inhabilitado, solo se habilita al seleccionar modo línea).

(Opcional) Ejercicio 7:


Realiza un componente reloj de agujas con horas minutos y segundos. Dispondrá de la posibilidad de sincronización manual (campo hora, minuto y segundo inicializadas por el programador) o por red (utilizará el servidor de tiempo creado en el tema de Networking).

Nota: Si quieres hacerlo con sincronización con un sistema real lee las especificaciones del protocolo NTP (funciona con UDP) y haz que el reloj se sincronice con un servidor NTP.

(Opcional) Ejercicio 8:

Se desea realizar un programa de dibujo. Para ello se debe hacer un componente que esté compuesto de un lienzo de dibujo y botones para dibujar puntos, líneas rectas, rectángulos y círculos al menos

El programa gestionará dicho componente: permitirá abrir una imagen o guardarla, cambiar los colores (usa un cuadro de diálogo de color) y una paleta rápida tipo la del Paint, y otras posibilidades que se te ocurran (Toma como modelo el MS Paint).

	RAMA:	Informática	CICLO:	DAM	
	MÓDULO	Desarrollo de Interfaces			CURSO: 2º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:
	UNIDAD	COMPETENCIA			

Fuentes:

Beginning Visual C# 2010. Karli Watson, Christian Nagel, Jacob Hammer Pedersen, Jon Reid, Morgan Skinner. 2010 Wiley Publishing, Inc., Indianapolis, Indiana

Visual C# 2010 Recipes: A Problem-Solution Approach © 2010 by Allen Jones and Adam Freeman. Apress.

Windows Forms Programming in C#. Chris Sells. Addison Wesley. 2003.

Wikipedia: <http://es.wikipedia.org> y <http://en.wikipedia.org>

Otros enlaces:

<http://stackoverflow.com/questions/29155/what-are-the-differences-between-delegates-and-events>

<https://blogs.msdn.microsoft.com/jfoscoding/2005/04/07/why-is-stathread-required/>

<http://stackoverflow.com/questions/144833/most-useful-attributes> Muy interesante

<https://www.codeproject.com/Articles/827091/Csharp-Attributes-in-minutes>

<https://www.dotnetperls.com/attribute>

https://www.akadia.com/services/dotnet_user_controls.html

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Desarrollo de Interfaces				CURSO: 2º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	UNIDAD	COMPETENCIA				

Apéndice I: Creación de eventos en Consola

Aunque los eventos es un concepto que asociamos inherentemente a entornos gráficos, estos también pueden ser usados como sistema de mensajería entre funciones (o hilos) en una aplicación de consola o en un servicio en background.

Veamos un ejemplo: Crea un nuevo proyecto de **consola** .

Tendremos dos clases, la clase Program con el Main y la clase Calculador con un método que calcula si cierto número es o no primo. Además está definido el delegado público.

El delegado y la clase Calculador:

```
//Delegado para gestionar el evento de aviso
public delegate void GestorDeCalculo(bool e);

public class Calculador
{
    private int numero;
    private bool primo=false;

    public event GestorDeCalculo CalculoFinalizado;

    public Calculador(int numero)
    {
        this.numero = numero;
    }

    public void esPrimo()
    {
        primo = true;
        for (int i = 2; i < numero; i++)
        {
            if (i % (numero / 1000) == 0)
                Console.WriteLine(".");
            if (numero % i == 0)
                primo = false;
        }
        CalculoFinalizado(primo); //Esto hay que comprobar que no sea nulo, ojo.
    }
}
```

En negrita se han resaltado las líneas implicadas en el evento.

Primero la creación del delegado:

```
public delegate void GestorDeCalculo(bool e);
```

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Desarrollo de Interfaces				CURSO: 2º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	UNIDAD	COMPETENCIA				

Luego la creación del evento que ha de cumplir el delegado anterior:

```
public event GestorDeCalculo CalculoFinalizado;
```

Cuando acaba la operación de cálculo, se lanza el evento con el parámetro definido:

```
CalculoFinalizado(primo);
```

La clase con el Main y con el método asociado al evento (Final) quedaría algo así:

```
class Program
{
    static void Final(bool e)
    {
        Console.WriteLine(e ? "Es primo" : "No es primo");
    }

    static void Main(string[] args)
    {
        Random g = new Random();
        Calculador c;
        int n = g.Next();
        Console.WriteLine("Veamos si {0} es primo. Pulse una tecla para
continuar", n);
        Console.ReadKey();

        c = new Calculador(n);
        c.CalculoFinalizado += new GestorDeCalculo(Final);

        Thread hilo = new Thread(c.esPrimo);
        hilo.IsBackground = true;
        hilo.Start();

        Console.WriteLine("Calculando. Ya me avisará cuando termine");
        Console.WriteLine("Aquí seguimos con otras tareas a la espera del
evento.");
        Console.WriteLine("Si pulsas una tecla muy pronto esto termina sin saber
el resultado.");
        Console.ReadKey();
    }
}
```

En este caso el método Final se ejecuta cuando Calculador lance el evento, que será al acabar el cálculo.

Por supuesto se pueden añadir varias funciones al evento de forma que se ejecutan todas cuando este se produzca. Tendría varias "suscripciones".

Está permitida la nomenclatura

```
c.CalculoFinalizado += Final;
```

pero no se recomienda por disminución de la claridad al no indicar el delegado que se

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM	
	MÓDULO	Desarrollo de Interfaces			CURSO: 2º
	PROTOCOLO:	Apuntes clases	AVAL:	DATA:	
	UNIDAD	COMPETENCIA			

asocia.

Para los eventos se suele usar un delegado muy estándar que acepta dos parámetros como ya hemos visto. Por un lado un objeto que contiene el objeto que lanza el evento y un parámetro con los datos necesarios sobre el evento que es un tipo *EventArgs* o uno derivado.

Para ajustarnos a este esquema haremos los siguientes cambios:

Creamos la clase *CalculoEventArgs* que deriva de *EventArgs* y modificamos el delegado:

```
public class CalculoEventArgs : EventArgs
{
    public CalculoEventArgs(bool resultado)
    {
        this.Resultado = resultado;
    }

    private bool resultado;
    public bool Resultado
    {
        set
        {
            resultado = value;
        }
        get
        {
            return resultado;
        }
    }
}

public delegate void GestorDeCalculo(object o, CalculoEventArgs e);
```

Cambiamos la llamada al evento en la clase Calculador:

```
CalculoFinalizado(this, new CalculoEventArgs(primo));
```

Cambiamos el método asociado al evento para cumplir la nueva signatura:

```
static void Final(object sender, CalculoEventArgs e)
{
    Console.WriteLine(e.Resultado ? "Es primo" : "No es primo");
}
```