

Tema 1: Desarrollo de Interfaces

¿Qué aprenderás?

- Entender el concepto de entorno de desarrollo.
- Características de las diferentes versiones de Visual Studio .NET
- Realizar la instalación de Visual Studio .NET
- Realizar la primera aplicación de forma guiada.
- Conocer y utilizar los controles disponibles en Visual Studio .NET
- Controlar los dispositivos de entrada de datos: Teclado, ratón y otros.

¿Sabías que...?

- Visual Studio .NET es un entorno de desarrollo de Microsoft y disponemos de una versión gratuita totalmente funcional llamada Communit.



1.1. Introducción a Visual Studio .NET

1.1.1. ¿Qué es Visual Studio .NET?

Visual Studio .NET es un paquete de desarrollo de aplicaciones (Software Development Kit o SDK en inglés), propiedad de Microsoft. Originariamente cuando Windows se puso en marcha, Microsoft disponía de dos líneas diferenciadas de productos para desarrollar aplicaciones, basadas en dos lenguajes de programación diferentes: Visual C++ y Visual Basic . A mediados de los años 90 y con la aparición del lenguaje Java, Microsoft desarrolló una nueva línea, el Visual J++ basado en este lenguaje. Esto hizo que Microsoft se planteara la posibilidad de crear un producto que englobase todos estos lenguajes de programación en un único entorno. Estos programas se llaman Entornos de Desarrollo Integrado (Integrated Development Environment o IDE en inglés). Este producto apareció en 1998 y se le llamó Visual Studio 6, en él se incluye Visual Basic, Visual C++, Visual J++ y algunas otras pequeñas tecnologías del momento. Al cabo de unos años, cuatro exactamente, se produce la gran revolución de Visual Studio, en el 2002, aparece la primera versión de Visual Studio .NET.

La gran novedad de .NET, no es el hecho de agrupar en un único entorno todos los lenguajes, sino el hecho de que se puedan mezclar todos los lenguajes entre sí en un mismo proyecto. Todo esto es posible, gracias a la existencia de un lenguaje intermedio de intercambio (parecido a la función que haría el inglés entre un sueco y un chino para poder entenderse). Este lenguaje intermedio se llama MSIL (Microsoft Intermediate Language). Además, Microsoft desarrolló el motor de aplicaciones llamado .NET framework que permite que una misma aplicación se pueda ejecutar en máquinas con versiones muy diferentes a las de Windows, incluso con otros sistemas operativos, ya que este motor hace de intermediario entre el sistema operativo del cliente y la aplicación. Lógicamente, el cliente debe responsabilizarse de tener el .NET Framework actualizado. Hoy en día, esto se consigue con las actualizaciones del sistema operativo que descargamos de la red.

Otro cambio que se produce con la aparición de la generación .NET, es la desaparición de J++, que queda sustituido por J# y también, la creación de un nuevo lenguaje específico para trabajar con Visual Studio .NET (formado a partir de C++ i J++) y llamado C#.



A partir de ese momento fueron apareciendo versiones de Visual Studio .NET, más o menos cada dos o tres años (Visual Studio .NET 2003, 2005, 2008, 2010, 2013, 2015 y actualmente 2017) en estas versiones no se han introducido grandes cambios, pero sí han ido mejorando el interfaz de trabajo y mejoras en el lenguaje ASP, que es el soporte para los dispositivos móviles y para el XML. En esencia, todo lo que es válido para el Visual Studio .NET más antiguo, es aplicable a la versión del 2017.

1.1.2. ¿Qué versión de Visual Studio necesitamos?

Para poder realizar este curso, sería ideal que tuviéramos un Visual Studio.NET lo más actualizado posible, pero una versión a partir del 2010 es perfectamente válida. Si no tenemos instalada ninguna versión en este momento, es hora de decidir qué versión queremos instalar. Para esto, hemos de tener en cuenta el rendimiento de nuestro ordenador, ya que las versiones más modernas de Visual Studio son muy exigentes con los requisitos que ofrece la máquina.

A pesar de esto, si consultamos la web de Microsoft, solo se necesita un mínimo de 2GB de RAM para instalar la versión 2017 (2,5 GB si es una maquina virtual) y con menos de 4GB, la velocidad de trabajo es muy baja. También hemos de tener en cuenta, que las últimas versiones exigen versiones de Windows 7 SP1 o superior. De este modo, podemos establecer un cuadro de requisitos para obtener el mejor rendimiento de trabajo:

Memoria RAM disponible	Versión Visual Studio .NET recomendada
2GB o menos	2013 /2015
2GB a 4GB	2015 / 2017
>4GB	2017



Requisitos de sistema extraídos de la web oficial:

Supported Operating Systems	Visual Studio 2017 will install and run on the following operating systems: <ul style="list-style-type: none">• Windows 10 version 1507 or higher: Home, Professional, Education, and Enterprise (LTSB is not supported)• Windows Server 2016: Standard and Datacenter• Windows 8.1 (with Update 2919355): Core, Professional, and Enterprise• Windows Server 2012 R2 (with Update 2919355): Essentials, Standard, Datacenter• Windows 7 SP1 (with latest Windows Updates): Home Premium, Professional, Enterprise, Ultimate
Hardware	<ul style="list-style-type: none">• 1.8 GHz or faster processor. Dual-core or better recommended• 2 GB of RAM; 4 GB of RAM recommended (2.5 GB minimum if running on a virtual machine)• Hard disk space: 1GB to 40GB, depending on features installed• Video card that supports a minimum display resolution of 720p (1280 by 720); Visual Studio will work best at a resolution of WXGA (1366 by 768) or higher

Si no disponemos de una versión oficial completa visual Studio, podemos descargarnos de la web de Microsoft lo que se llaman versiones Community (antes se llamaban express). Estas versiones son legales y gratuitas, pero tienen algunos pequeños inconvenientes. El primero es que pide cuenta live-mail (aunque se puede usar sin ella). El segundo es que algunos complementos (aunque son pocos) no están disponible.

Durante este curso, la documentación (sobre todo capturas de imágenes) será entregada utilizando la versión 2017 (versión en Español), porque es la versión más simple de obtener actualmente y los requisitos de maquinaria, para obtener un buen rendimiento de trabajo, son fácilmente asumibles; pero en el caso de utilizar alguna otra versión, no habrá ningún problema para seguir el curso, ya que todos los ejercicios, ejemplos y aplicaciones que realizaremos a lo largo del curso, son compatibles con todas las versiones de Visual Studio. El lenguaje de programación que utilizaremos a lo largo de todo el curso es el C#, ya que de todos los soportados por Visual Studio .NET, es el más frecuente en la actualidad.



1.2. Guía de instalación:

Lo primero que necesitamos tener es una versión de Visual Studio.

1.2.1. Adquisición del software

Para obtener una versión Community hay que ir a la web de Microsoft:

<https://www.visualstudio.com/es/downloads/>



Una vez hemos accedido a la web, desplegamos el menú de descarga y elegimos la versión **Community** {1}.

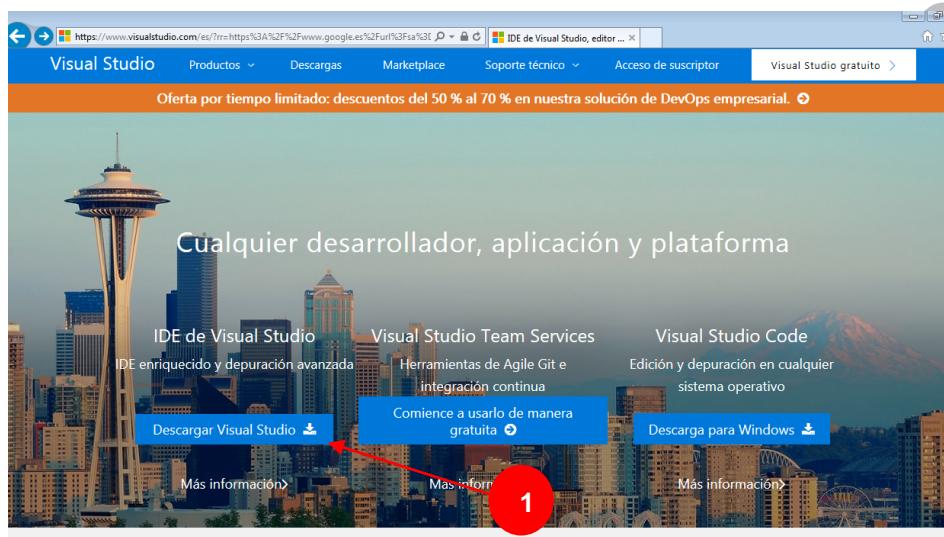


Figura: Página de Visual Studio

A continuación, elegimos la versión Community del desplegable y se descargará el ejecutable en nuestro PC.



Ahora ya podemos proceder a la instalación del entorno de programación.



1.2.2. Instalación y primer uso del Visual Studio

La instalación del Visual Studio no es nada compleja. Aunque hay que decir que puede llevar mucho tiempo ya que el instalador no contiene los ficheros sino que solo es un asistente que nos permite elegir los componentes que deseamos instalar. Luego se descargan de la red a medida que se instalan por lo que la instalación se puede alargar incluso algunas horas en función de la cantidad de complementos elegidos.

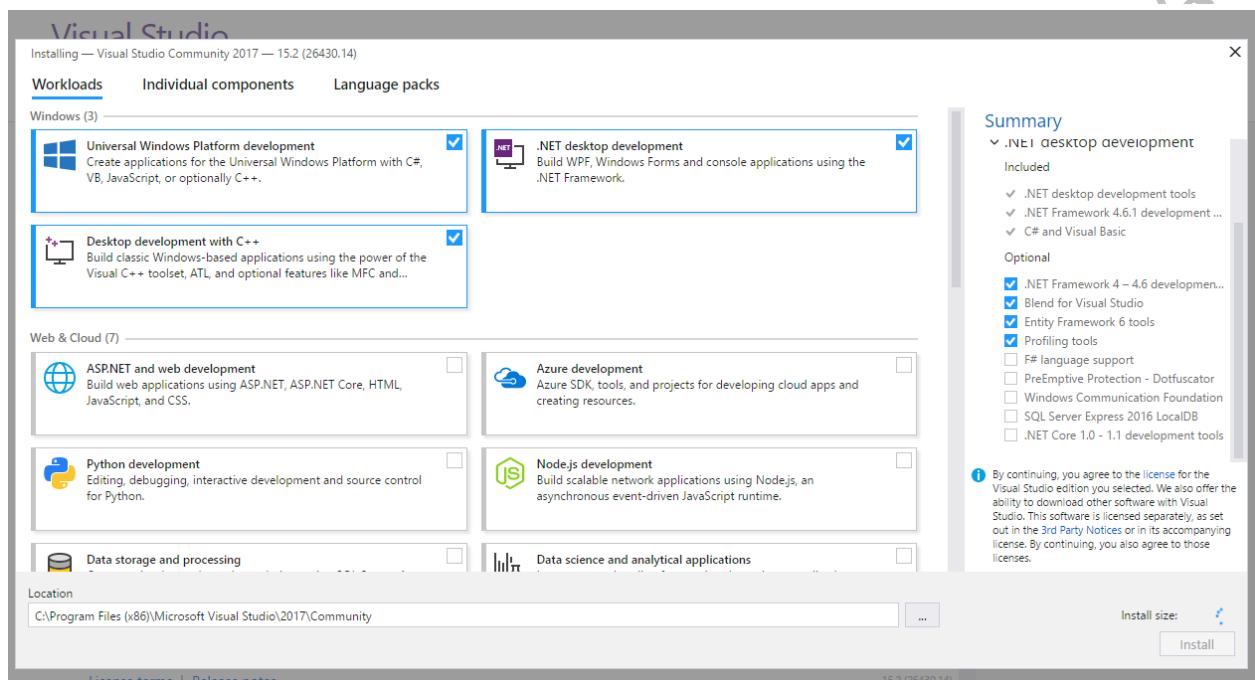


Figura: Instalación de Visual Studio 2017

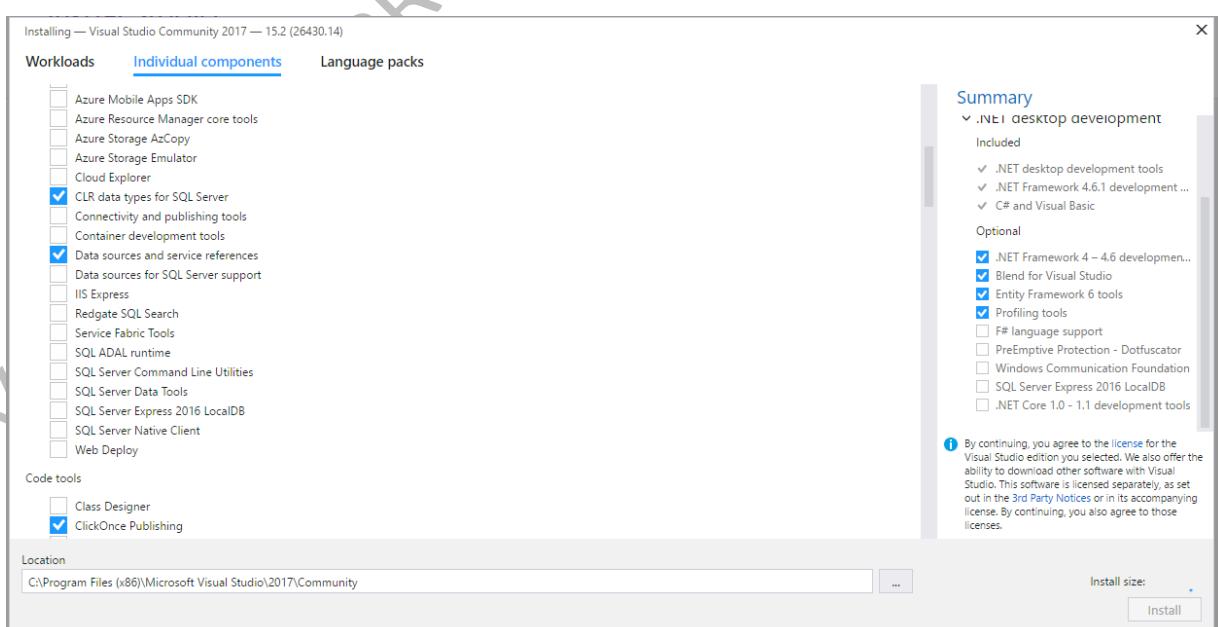


Figura: Instalación de Visual Studio 2017 (selección de componentes individuales)



Installing — Visual Studio Community 2017 — 15.2 (26430.14)

Workloads Individual components Language packs

Choose your language pack

- Chinese (Simplified)
- Chinese (Traditional)
- Czech
- English
- French
- German
- Italian
- Japanese
- Korean
- Polish
- Portuguese (Brazil)
- Russian
- Spanish
- Turkish

Figura: Instalación de Visual Studio 2017 (Selección de Idiomas)

Visual Studio

Figura: Instalación de Visual Studio 2017 (progreso de la instalación)

Products

Installed

Visual Studio Community 2017

Acquiring Microsoft.VisualStudio.Cpp.Redist.14
28%
Applying Microsoft.VisualStudio.DotNetNative.ILC
7%

Cancel

Available

Una vez finalizado, iniciamos Visual Studio y la primera vez se iniciará un pequeño asistente. Lo primero que nos va a pedir es que nos conectemos con una cuenta Microsoft. Aunque este paso se puede omitir con el enlace de la parte inferior:

Not now, maybe later {1}

Visual Studio

Welcome!

Connect to all your developer services.

Sign in to start using your Azure credits, publish code to a private Git repository, sync your settings, and unlock the IDE.

[Learn more](#)

[Sign in](#)
Don't have an account? [Sign up](#)

[Not now, maybe later.](#)

1

Figura: Primer uso de Visual Studio



Lo siguiente que nos pide el asistente es que elijamos un estilo para el entorno y un lenguaje de programación preferido (aunque siempre podemos usar el que queramos):

Figura: Primer uso de Visual Studio

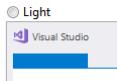
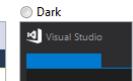
Visual Studio

Start with a familiar environment

Development Settings: General

Apply customizations from the previous version to the environment selected above.

Choose your color theme



You can always change these settings later.

[Start Visual Studio](#)

En nuestro caso en el desplegable **Development Settings** elegiremos **Visual C#**

Start with a familiar environment

Development Settings: Visual C#

Apply customizations from the previous version to the environment selected above.

Figura: Primer uso de Visual Studio

Ahora ya podemos iniciar el entorno por primera vez la pantalla que veremos tiene el aspecto siguiente:

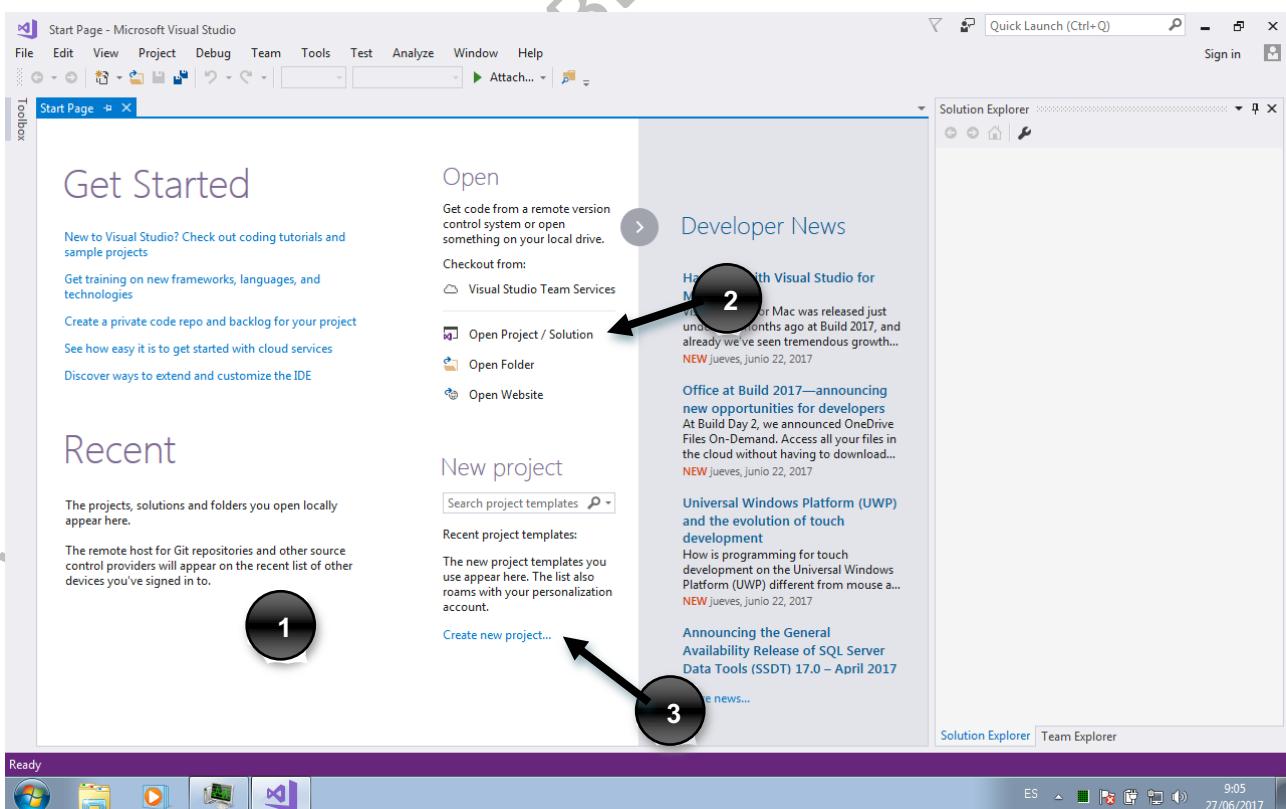
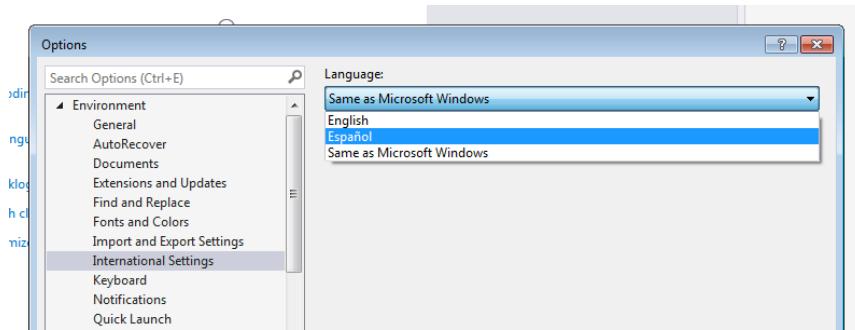


Figura: Pantalla principal de Visual Studio



Si Visual Studio aparece en inglés en vez en español Selecciónar la opción **Options** del menú **Tools**. Luego en la lista de la parte izquierda buscar la opción **International Settings**. En el desplegable que aparece en la parte derecha elegir **Español**. Luego hay que reiniciar Visual Studio para que los cambios tengan efecto.



1.3. Primera aplicación .NET

Ahora vamos a conocer los pasos necesarios que hay que seguir para poder crear nuestra primera aplicación en Visual Studio .NET. Lo haremos de una forma totalmente guiada y no es necesario preocuparse ahora de si se entiende o no el código programado, lo que en este apartado interesa es adquirir la mecánica de creación de una aplicación y un dominio del entorno Visual Studio y así poder seguir el resto de la guía.

iRecuerda!

A lo largo del documento, las opciones de menú se mostrarán con el formato siguiente: **Menú → Opción → Sub-opción1 → Sub-opción2...** Los atajos del teclado se muestran con el formato siguiente: **<atajo>**

1.3.1. Creación de un nuevo proyecto

En la ventana de la figura 5, podemos apreciar diversas zonas que iremos explicando más adelante a lo largo del curso. De momento, nos interesa la zona central de la ventana donde podemos apreciar una pequeña lista de los últimos proyectos utilizados {1} y dos enlaces: uno para abrir proyectos guardados en el disco {2} y el otro para crear nuevos proyectos {3}.

Ahora vamos a crear un nuevo proyecto. Para hacerlo podemos utilizar el enlace comentado anteriormente o directamente hacerlo desde el menú de la aplicación (**Menú Archivo → NuevoProyecto**) o directamente con el atajo del teclado **<Ctrl+Mayus+N>**.

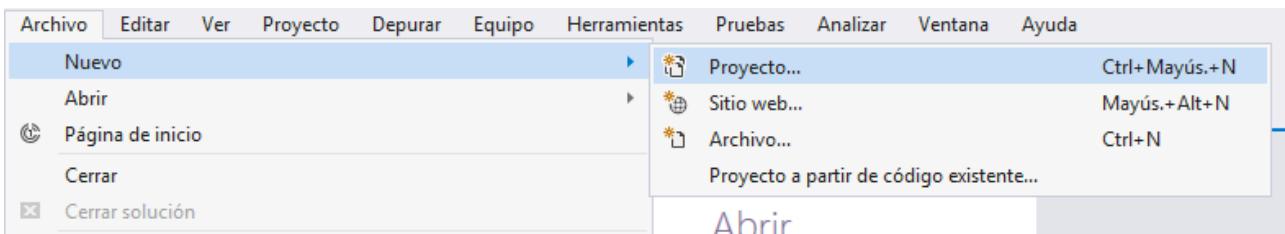


Figura: Menú “Archivo”

Cuando lo hagamos, aparecerá una ventana que nos permitirá escoger el tipo y las características del proyecto que queremos crear.

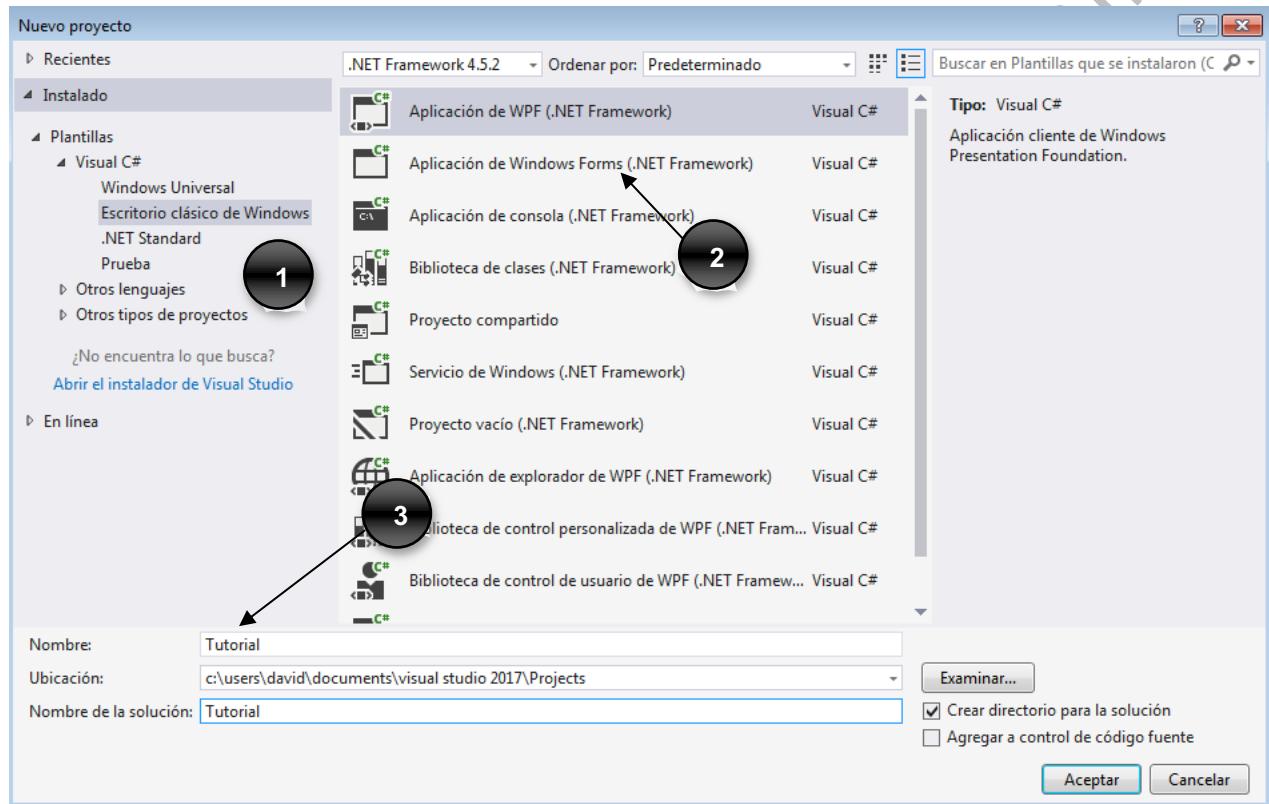


Figura: Ventana de nuevo proyecto

En esta ventana, lo primero que hemos de hacer es escoger el lenguaje de programación y el tipo de proyecto mediante el menú de la izquierda {1}. En nuestro caso, escogeremos como lenguaje de programación el C# y dentro de esta sección como tipo de proyecto Escritorio clásico de Windows. A continuación y en función de la opción escogida, en la parte derecha de la ventana {2} aparecen algunas opciones de proyecto; en nuestro caso, escogemos la primera de las opciones disponibles: **“Aplicación de Windows Forms”**. Para finalizar, escogemos un nombre para el proyecto que vamos a crear {3}, en nuestro caso le vamos a poner el nombre de **“Tutorial”**.



Cuando aceptamos y creamos el proyecto, el entorno tendrá un aspecto como el que se muestra en la imagen siguiente:

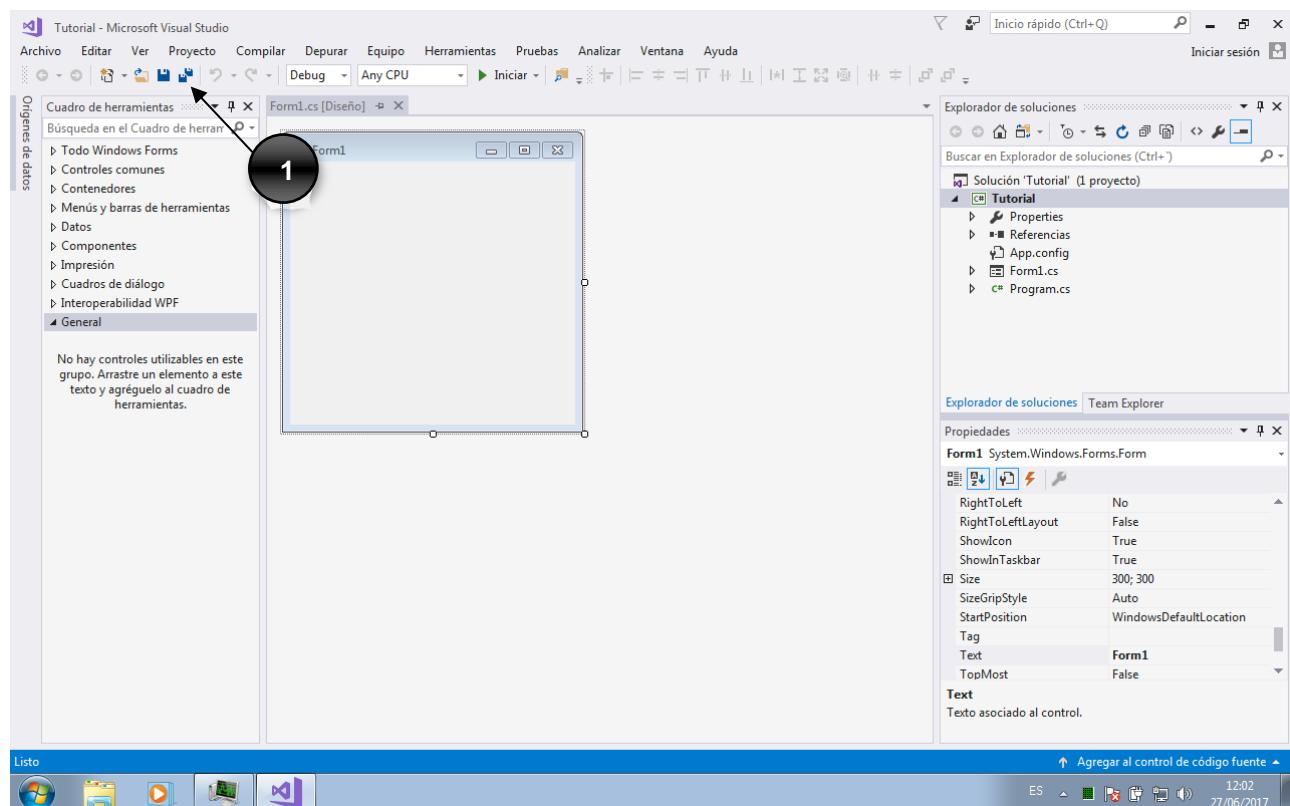


Figura: Aspecto de un Nuevo Proyecto

Aquí podemos observar muchas zonas diferentes que iremos explicando a lo largo del curso a medida que las vayamos utilizando. Ahora lo más importante es guardar el proyecto, ya que, de la misma forma que ocurre con un documento de texto cuando lo creamos, si cerramos el entorno, perderemos cualquier tarea que hayamos hecho. Para guardar un proyecto, lo podemos hacer de diversas formas con el menú (**Menú Archivo → Guardar Todo**) con el atajo **<Ctrl+Mayús+S>** o bien con el ícono que muestra un montón de discquetes {1} de la barra de herramientas.

iRecuerda!

Es importante recordar que debemos utilizar la opción “Guardar Todo” y no las otras opciones de guardado, ya que solo esta última guarda la totalidad de los archivos del proyecto.

La ubicación predeterminada de los proyectos de Visual Studio se encuentra en la carpeta “Mis Documentos/Visual Studio 2017/Projects” pero podemos guardar y cargar proyectos desde cualquier ubicación. Una vez guardado, podemos comprobarlo cerrando el proyecto (**Menú Archivo**



→**CerrarProyecto**) y volveremos a la pantalla inicial de Visual Studio. Ahora podemos abrir de nuevo nuestro proyecto utilizando el acceso directo de la zona de “**ProyectosRecientes**” con el menú (**Menú Archivo →AbrirProyecto**) o bien <Ctrl+O>. El fichero que hay que seleccionar es el que tiene la extensión “.sln”. En la parte derecha de la ventana podemos ver un cuadro que nos muestra los ficheros que forman parte de nuestro proyecto (Visual Studio los llama Soluciones).

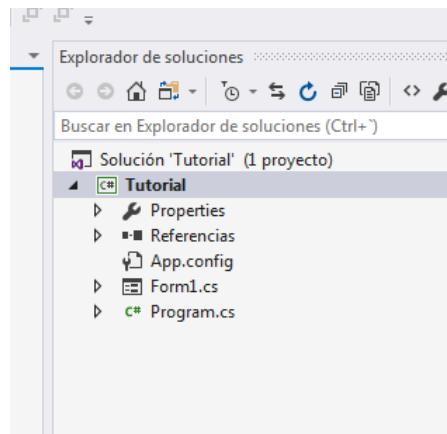


Figura: Explorador de ficheros del proyecto

En esta ventana se muestran los archivos que forman el proyecto. Los archivos con extensión “.cs” son archivos con código. En nuestro caso solo hay un archivo de código. Si activamos el menú de contexto (utilizando el botón derecho del ratón) sobre el archivo Form1.cs veremos las opciones que podemos utilizar sobre el fichero:

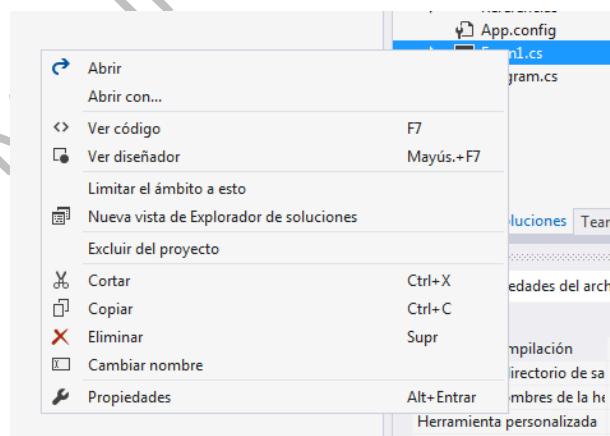


Figura: Menú contextual

Si escogemos la opción “**Ver diseñador**” o bien, sencillamente hacemos doble clic sobre el nombre del fichero, veremos como aparece la ventana donde se muestra el diseño de nuestro formulario (que en estos momentos está vacío). **IMPORTANTE:** En el entorno Visual Studio, a las ventanas, de ahora en adelante las llamaremos formularios (el objeto se llama Form).



Observad las solapas que hay {1} y que nos permiten alternar entre las diferentes ventanas, mejorando la eficiencia del uso del espacio de pantalla disponible.

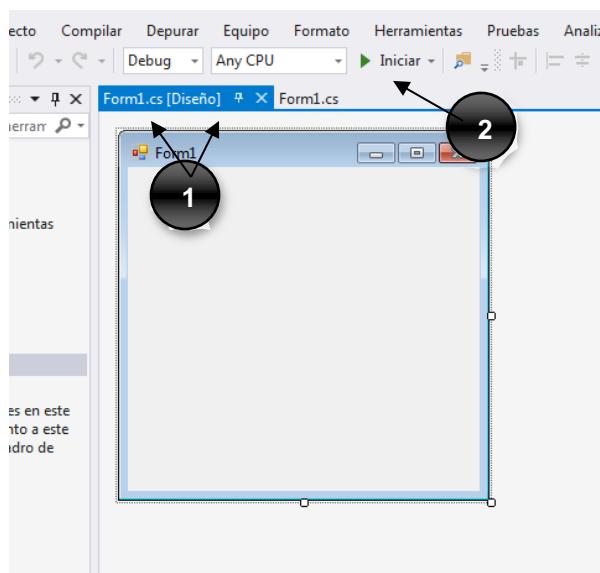
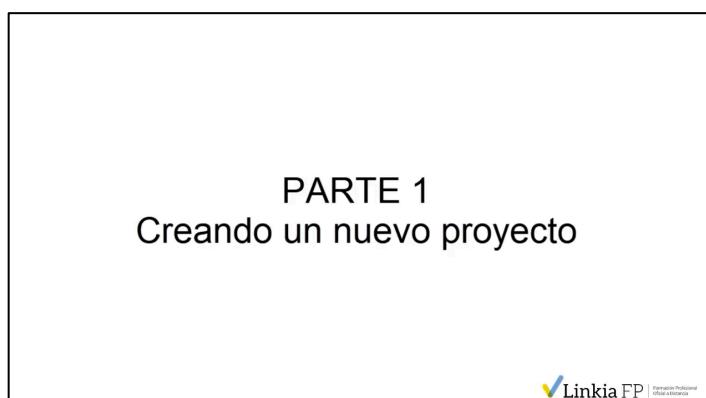


Figura: Iniciar ejecución

Ahora podemos ejecutar nuestra aplicación. Podemos hacerlo desde el menú (**Menú Depurar → Iniciar Depuración**) con la tecla F5 o bien, con la pequeña flecha verde que hay en la barra de herramientas {2}. Cuando ejecutamos la aplicación, vemos como aparece una ventana llamada Form1 completamente funcional (la podemos maximizar, minimizar, redimensionar,...).

Para detener la ejecución, simplemente cerramos la ventana con el botón X que está en la parte superior derecha y volvemos al entorno de programación.



Video: Primera aplicación .NET (parte 1)



1.3.2. La programación orientada a objetos y .NET

El Visual Studio .NET es un entorno de programación orientado a objetos que de forma más breve llamamos POO (Oriented Object Programming o también OOP en inglés). Estos entornos de programación están basados en la creación y usos de objetos, es decir, que un programa, en realidad, es un grupo de objetos que se interrelacionan los unos con los otros para llevar a cabo una tarea. Si reflexionamos, veremos que muchos de los objetos cotidianos que utilizamos, como un coche, una televisión, un armario...siguen esta filosofía. Resulta evidente que la mayoría de nosotros no podríamos hacer un armario de la nada, pero casi todo el mundo sería capaz de encajar las piezas del kit de un armario comprado en unos grandes almacenes. Esta es la idea, los desarrolladores de Microsoft nos proporcionan una colección de piezas (en el lenguaje técnico las llamamos objetos) que muchos de nosotros, probablemente, no seríamos capaces de crear, para que las combinemos de forma inteligente para construir nuestra aplicación.

De esta manera, el formulario blanco que se ha generado al crear nuestro proyecto Tutorial es nuestro primer objeto. Cada una de las ventanas que utilizamos en nuestras aplicaciones es un objeto. De hecho, todo lo que utilizamos en nuestros proyectos son objetos. Pero, ¿todas las ventanas son iguales? Obviamente no, algunas son más grandes, otras más pequeñas o de diferente color, etc. por tanto, ¿qué va a determinar el aspecto de nuestra ventana? La respuesta es: Las Propiedades.

Todos los objetos tienen una serie de propiedades que regulan su aspecto y funcionalidades. En el caso, por ejemplo, de un coche serían: el color, la potencia, el número de puertas, etc. cada una de las propiedades de nuestro objeto tiene un valor. Siguiendo con el caso del coche, la propiedad del color podría tener como valores válidos el rojo, el negro, el blanco, etc. Por tanto, todos los objetos del mismo tipo tienen las mismas propiedades, pero el valor de cada uno de ellos depende de cada caso. Para trabajar con las propiedades de nuestro formulario, utilizamos el inspector de propiedades que es una ventana que encontraremos en la parte inferior derecha del entorno Visual Studio:

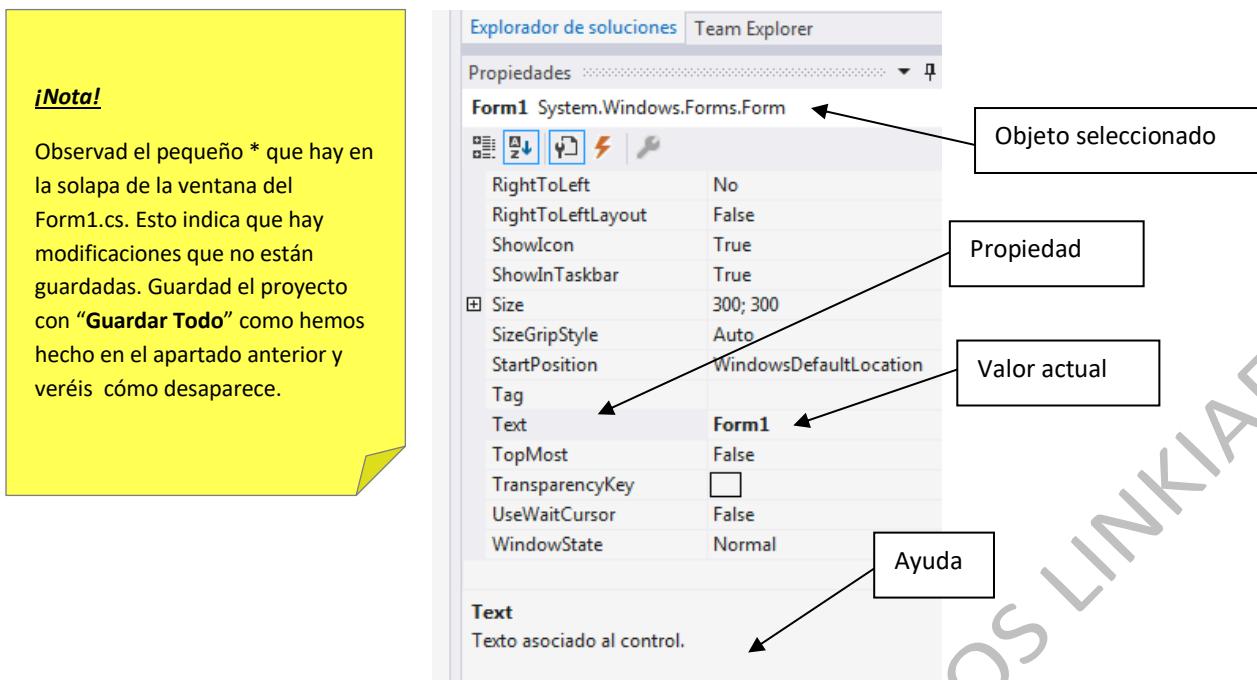


Figura: Inspector de Propiedades

En esta ventana podemos observar todas las propiedades del objeto seleccionado, así como sus valores. Observamos la propiedad llamada "Text" que actualmente tiene el valor "Form 1". El cuadro de ayuda, nos informa que esta propiedad regula el texto asociado al objeto Form1. Ahora lo que vamos a hacer es modificarle el valor y lo cambiaremos por "TUTORIAL". El resultado es que el título de la ventana ha cambiado por la de TUTORIAL. De esta manera, nos queda claro que la propiedad Text del objeto Form1 controla el texto de la barra del título de la ventana.

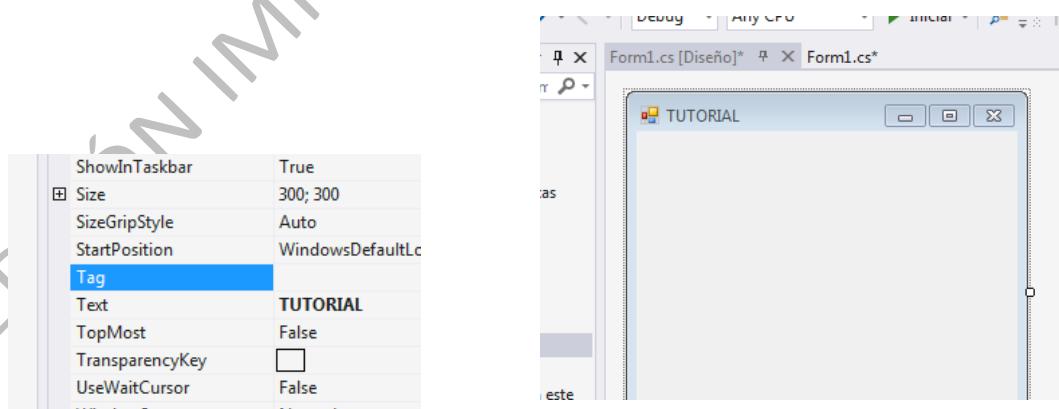


Figura: Modificación de Propiedades



1.3.5. Añadir nuevos elementos a la aplicación

Ahora vamos a poner nuestro primer control a la aplicación. Para poder hacerlo, nos vamos a fijar en la ficha que dice “**Cuadro de Herramientas**” que hay en la parte izquierda del entorno de programación. El cuadro de herramientas está organizado por fichas, de momento solo trabajaremos con la que dice “**Controles comunes**”. En esta ficha localizamos un control que se llama “**Label**”, que quiere decir etiqueta.

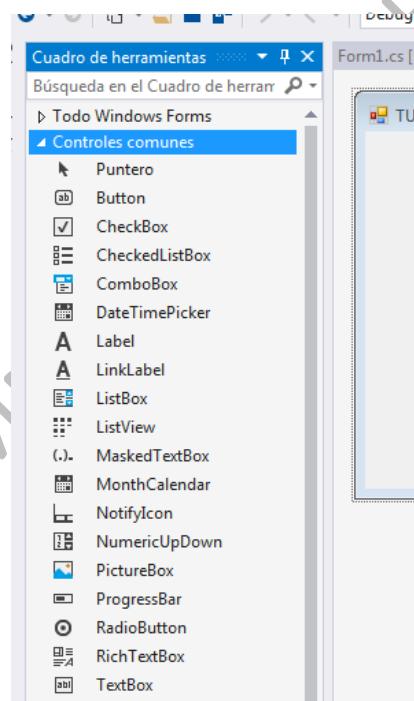


Figura: Cuadro de herramientas.

Para añadir un control del tipo **Label** a nuestro proyecto, solo hemos de hacer clic en el elemento **“Label”** del cuadro de herramientas y entonces, cuando ponemos el puntero del ratón sobre la superficie del formulario, observaremos como este se transforma en una cruz con una versión en miniatura del mismo ícono con el que aparece el control **“Label”** en el cuadro de herramientas. Con el cursor en esta forma, volvemos a hacer clic en el punto del formulario donde queremos colocar nuestra etiqueta.

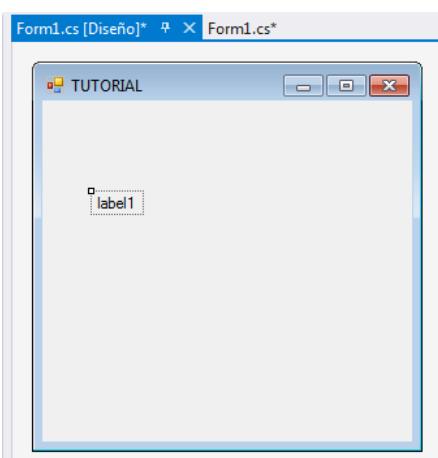


Figura: Nueva Etiqueta



Ahora lo que hay que hacer, es dar los valores que queramos a las propiedades del control que acabamos de poner. Observad que si seleccionamos el control haciendo clic encima del control “**Label**” que acabamos de poner (hemos de tener cuidado de no hacer doble clic sobre el control, ya que si lo hacemos se nos abrirá el editor de código, y esto lo aprenderemos más adelante) el inspector de propiedades nos muestra las propiedades de un objeto llamado “**Label1**”. Este nombre **Label1** es un identificador **ÚNICO** que nos permitirá referirnos a esta etiqueta en concreto en cualquier momento que lo necesitemos (por ejemplo, a la hora de asignar valores a sus propiedades). Podemos considerar a “**Label1**” el nombre propio de esta etiqueta.

¡¡Ahora pondremos el valor **HOLA A TODOS!!** a la propiedad “**Texto**” del **Label1**. Y el aspecto debería ser este:

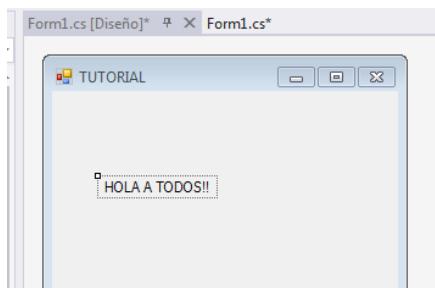


Figura: Cambio del texto

A continuación, buscaremos la propiedad “**Font**”. Observad el pequeño signo de suma que hay delante de la propiedad, esto indica que, de hecho, esta propiedad está formada por un conjunto de pequeñas propiedades agrupadas bajo un solo nombre. Hacemos clic sobre el signo más para que se desplieguen todas las sub propiedades de fuente y localizaremos una que se llama “**Bold**”. Esta propiedad indica si la fuente de letra de la etiqueta tiene estilo *negrilla*. Los valores posibles son verdadero (“**True**”) o falso (“**False**”). Actualmente es falso. Para modificar el valor, podemos hacerlo mediante el despegable que posee la propiedad o sencillamente haciendo doble clic encima de la palabra “**False**”. A continuación, establece el tamaño de la letra “**Size**” con el valor 16. Ahora deberíamos tener algo parecido a esto:

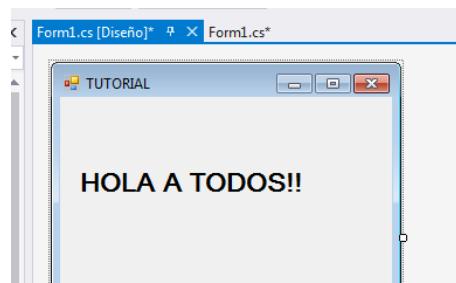


Figura: Cambio del texto



Si queremos que la etiqueta quede centrada en la ventana, lo podemos hacer desplazándola con el ratón, hasta que quede más o menos centrada. También podemos hacerlo utilizando las opciones disponibles del menú “Formato”. En este menú encontraremos todas las opciones que nos permiten modificar el tamaño y la posición de los controles colocados en el formulario. En nuestro caso, con la etiqueta seleccionada elegiremos **Menú Formato → Centrar en el formulario →horizontalmente**. Con esto conseguiremos que la etiqueta quede exactamente en el centro de la ventana. Si en algún momento el tamaño de la ventana o de la etiqueta cambiase, habríamos de repetir esta operación para mantenerla en el centro.

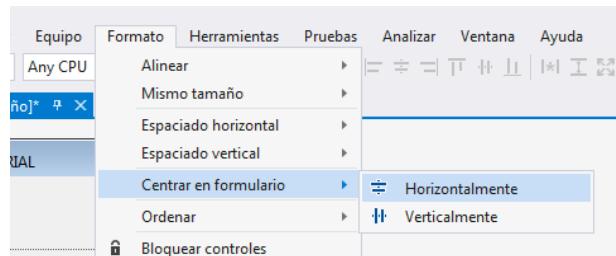


Figura: Menú formato

Ahora si ejecutamos de nuevo nuestro programa tendremos una aplicación como la que vemos en la siguiente imagen:

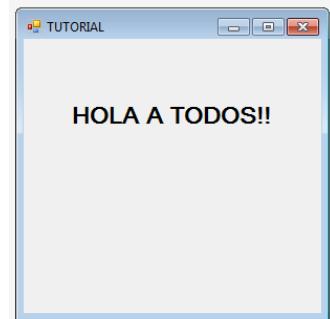
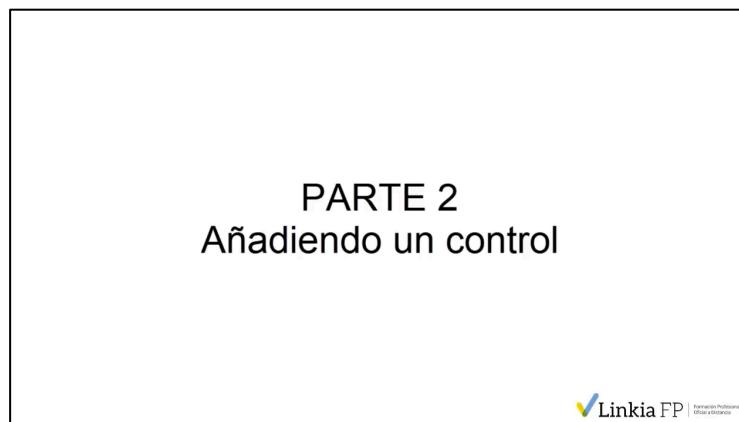


Figura: Nuevo control



Video: Primera aplicación .NET (parte 2)



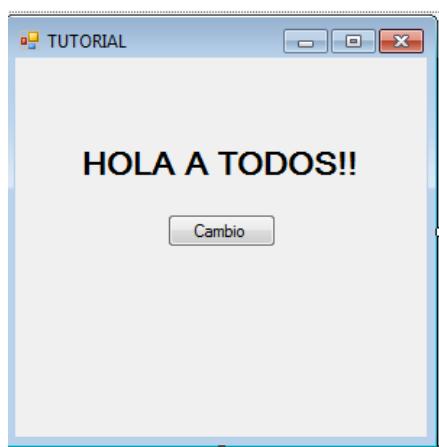
1.3.6. Modificación de propiedades durante la ejecución

Hasta este momento, hemos hecho las modificaciones de las propiedades de los objetos (título de la ventana y texto de la etiqueta), mediante la ventana de propiedades que tenemos a la derecha del entorno de programación, y lo más importante, con la **APLICACIÓN DETENIDA**. Es evidente, que esto no es nada complicado, pero no permite que el usuario interactúe con la aplicación para que esta se comporte de forma diferente. Este sistema de trabajo que hemos seguido en el apartado anterior, se le llama modificación de propiedades en **tiempo de diseño**.

Ahora vamos a ver el otro método para modificar propiedades, y a este se le llama modificación de propiedades en **tiempo de ejecución**. Como ya podemos suponer, lo que se trata es de modificar el valor de una propiedad cuando la aplicación está en funcionamiento. Sin dudas, necesitaremos que se produzca alguna cosa que desencadene este cambio de propiedad. En programación orientada a objetos, a estos factores desencadenantes se les llama Eventos (Events en inglés). Hay muchas clases diferentes de eventos, la mayoría son activados por el usuario, como por ejemplo, clics con el ratón, activación de botones, etc., pero hay otros que se activan de manera automática en determinadas situaciones, como por ejemplo, cuando cerramos la aplicación.

Ahora lo que vamos a hacer es permitir al usuario que pueda modificar el texto de la etiqueta cuando lo deseé. Para poder hacer esto, pondremos un botón que el usuario apretará cuando quiera hacer este cambio. Para añadir este botón, seguiremos el procedimiento que hemos visto en el apartado anterior para poner un control del tipo **Button** a nuestro formulario. Le asignaremos el valor “**Cambio**” a su propiedad **texto** y lo colocaremos centrado debajo de la etiqueta. El resultado debería ser como el que vemos en la siguiente imagen:

Figura: Nuevo botón





Ahora debemos programar que cuando se produzca el evento “**el usuario hace clic sobre el botón cambio**”, el texto que muestra la etiqueta cambie por “¡Adiós a todos!”. Para poder hacer esto debemos generar un evento click para nuestro botón. Para ello con el botón seleccionado nos dirigimos al cuadro de propiedades y pulsamos sobre el botón que tiene un símbolo de un relámpago {1}. Con ello abrimos la lista de eventos del control.

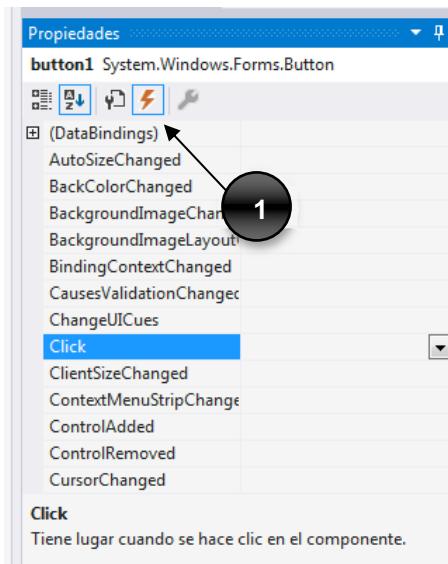


Figura: Lista de eventos

Una vez se abre la lista de eventos localizamos el que queremos activar (en nuestro caso **click**) y podemos hacer doble clic en el con lo que se genera un evento con un nombre genérico formado por el nombre del control y tipo de evento (**button1_click** en ese caso) o si deseamos ponerla un nombre personalizado lo escribimos en el cuadro de texto.

En nuestro caso escribiremos saludar en el cuadro de texto y luego pulsamos retorno. Al hacerlo se abre automáticamente la ventana de código y se sitúa en la función generada. El aspecto de la ventana debería ser el siguiente:



The screenshot shows the Microsoft Visual Studio IDE. The title bar indicates the project is named 'Tutorial' and the file is 'Form1.cs [Diseño]*'. The code editor displays the following C# code:

```
1  using System;
2  using System.Collections.Generic;
3  using System.ComponentModel;
4  using System.Data;
5  using System.Drawing;
6  using System.Linq;
7  using System.Text;
8  using System.Threading.Tasks;
9  using System.Windows.Forms;
10
11 namespace Tutorial
12 {
13     public partial class Form1 : Form
14     {
15         public Form1()
16         {
17             InitializeComponent();
18         }
19
20         private void saludar(object sender, EventArgs e)
21         {
22             // Code here
23         }
24     }
25 }
26
```

Figura: Editor de código

Resulta evidente que cuando en un formulario ponemos, por ejemplo, diez botones o más (cantidad habitual), el hecho que se llamen Button1, Button2, Button3,... resulta muy poco práctico, ya que cuando tenemos que hacer referencia a un objeto, no sabremos cuál de todos los botones queremos. Por este motivo, debemos acostumbrarnos desde un principio a modificar el nombre que el programa asigna de forma predeterminada a los controles, por un nombre que dé más información sobre de qué control se trata.

Para hacer esto debemos volver a la vista de diseño e ir al cuadro de propiedades y modificar la propiedad “**(Name)**” de los dos controles que hemos puesto para los siguientes valores:

- Etiqueta → et_mensaje
- Botón → bt_cambio



Volvemos a la ventana de código. De momento no analizaremos estas líneas en profundidad, ya lo haremos más adelante. Ahora lo que debemos entender, es que las líneas de código que ponemos entre estas dos, se ejecutarán cada vez que el usuario haga clic en el botón cambio. Ahora hay que escribir las instrucciones que modificarán el valor de la propiedad “Texto” de la etiqueta “et_mensaje” cuando este evento se produzca. En Visual Basic .NET la estructura básica de una instrucción que modifica el valor de una propiedad sigue el modelo siguiente:

objeto.propiedad=nuevo valor

Entonces como el objeto a modificar es “et_mensaje”, la propiedad a modificar es “Texto” y el nuevo valor es “!Adiós a todos！”, la instrucción sería:

et_mensaje.Texto="¡Adiós a todos!"

Ahora vamos a escribir esta instrucción. Lo primero que hemos de tener en cuenta es de poner la instrucción dentro de la zona delimitada por las dos líneas de código, creadas antes cuando hemos activado el evento clic.

Como es la primera instrucción que escribimos en Visual Studio, veremos con detalle, las posibilidades que nos ofrece el editor de código de Visual Studio. Si vamos escribiendo la instrucción letra a letra, veremos que justo cuando escribimos la primera “et” ya aparece un desplegable con opciones.

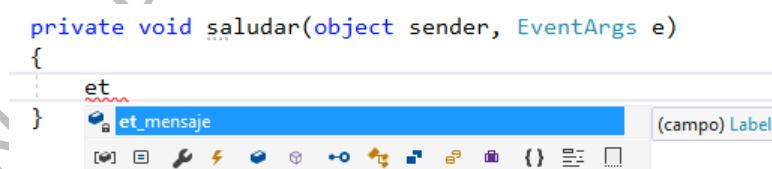


Figura: Auto completado de código (1)

Luego elegimos el control de la lista emergente y pulsamos un punto para que aparezca la lista de propiedades y eventos de dicho control.

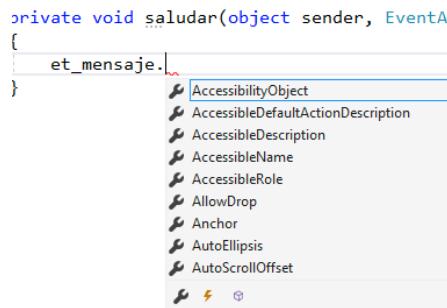


Figura: Auto completado de código (2)

De la lista emergente elegimos Text. Podemos acceder rápidamente a el si escribimos las primeras tetras (una T, une e,...). También podemos usar los iconos de la parte inferior para filtrar la lista (propiedades, eventos,...).

Una vez hecho esto completamos la instrucción.

```
private void saludar(object sender, EventArgs e)
{
    et_mensaje.Text = "Adios a Todos!!";
}
```

Figura: Línea de código completa

Observamos que el texto escrito entre comas (valores literales) es de color rojo, las palabras especiales del lenguaje C# son de color azul. Objetos y propiedades salen en color negro.

Ahora solo debemos ejecutar el programa otra vez y pulsar el botón **Cambio**, para observar como el texto de la etiqueta cambia.

Ya hemos visto la mecánica básica para generar y ejecutar un proyecto, así como se pueden añadir objetos y editar sus propiedades, tanto en tiempo de diseño como en tiempo de ejecución. La sintaxis de C# es idéntica a Java.

Video: Primera aplicación .NET (parte 3)

PARTE 3 Creando interacción



1.4. Controles básicos

1.4.1. Cajas de Texto (TextBox)

Las cajas de texto son uno de los controles más habituales en las aplicaciones, gracias a la gran polivalencia que ofrecen, porque el usuario puede escribir cualquier cosa: fechas, enteros, decimales, horas, texto,... Esta polivalencia, a su vez, lo convierte en un control que a veces puede generar excepciones y errores y, por tanto, tenemos que tener cuidado con lo que programamos.

Su propiedad más importante es **Text** que es una variable del tipo **String** y contiene el valor que hay escrito en la caja de texto. El hecho de que esta propiedad sea una cadena de texto hace que, cuando lo que tengamos escrito en la caja represente algún otro tipo de dato (como por ejemplo, un número decimal), tendremos que realizar la conversión del valor de la propiedad **Text** al tipo de dato que se espera mediante las funciones de **Parse**. Lógicamente, si la conversión no es posible, se genera una excepción que tendremos que capturar y reconducir de lo contrario nuestro programa “petará”.

Ejemplo:

Queremos poner una caja de texto para que el usuario introduzca un importe en euros. Por lo tanto, cuando tengamos que utilizar el contenido de la caja de texto para realizar alguna operación, tendremos que convertir la cadena de texto en un número decimal.

```
Double.Parse(textBox1.Text)
```

Resulta útil guardar el valor convertido en una variable para no tener que realizar la conversión otra vez, ya que esta función consume tiempo de proceso y no es necesario hacerlo cada vez.

Para evitar las excepciones provocadas por cadenas de texto que no se pueden convertir en un número decimal utilizaremos la sentencia **Try/Cath**.



```
double importe;  
try  
{  
    importe = Double.Parse(textBox1.Text);  
}  
catch  
{  
    textBox1.Text = "Importe incorrecto!!";  
}
```

Con este sistema podemos tratar, prácticamente, cualquier caso que nos podamos encontrar. Otra consecuencia que genera que la propiedad **Text** sea un **String** es que a menudo, tendremos que procesar la cadena de texto y esto nos lleva a que tendremos que dominar las propiedades y los métodos de la clase **String**.

Tener que poner los **Parse** en un bloque **try/cath** es algo tan frecuente que en C# existe un método rápido para solucionar esos casos. Se llama **TryParse** y devuelve **true** si el **parse** se ha realizado con éxito o **false** en caso contrario. Pero **NO** genera ninguna excepción ni “peta” el programa. La variable dónde vamos a almacenar el resultado del **Parse** debe ir precedida de la palabra **out**.

```
double importe;  
  
if (!Double.TryParse(textBox1.Text, out importe))  
{  
    TextBox1.Text = "Importe incorrecto!!";  
}
```

Ejemplo:

Queremos mejorar el caso anterior para permitir que el usuario ponga el símbolo del € al final del importe y no se genere una excepción en la conversión a número decimal. Lo podemos hacer de diferentes maneras; una de ellas es asumir que el símbolo del euro está al final de la cadena y utilizar todos los caracteres menos el último para hacer la conversión.

```
importe = Doble.Parse(textBox1.Text.Substring(0, textBox1.Text.Length - 1))
```



La función **Substring** coge una porción de la cadena completa, en nuestro caso la subcadena empieza en el carácter 0 de la cadena principal y tiene una cantidad de caracteres igual a la longitud total de la cadena completa menos uno (que es el símbolo del €).

La otra opción es utilizar un método llamado **Trim** que elimina un determinado carácter de la cadena y elimina el símbolo del €.

```
importe = Double.Parse(textBox1.Text.Trim('€'))
```

Como vemos es necesario aprender las posibilidades que nos ofrecen las clases básicas como son **Array**, **String**, etc. para solucionar distintas dificultades en el momento de utilizar los controles del formulario.

1.4.2. Cajas Numéricas (NumericUpDown)

Estas son una variante de las Cajas de Texto, que se utilizan cuando la información de la caja de texto DEBE SER OBLIGATORIAMENTE un número. La principal ventaja es que ya no utilizamos la propiedad **Text** sino que se utiliza la propiedad **Value**, que ya es del tipo **decimal**, y por tanto, no hemos de hacer ningún tipo de conversión ni control de excepciones. Se deben tener en cuenta dos propiedades más que son **Maximum** i **Minimum** que controlan los valores máximo y mínimo que puede contener la caja, de esta manera ahorramos instrucciones que controlen este hecho. Con la propiedad **DecimalPlaces** podemos restringir el numero de decimales aceptado.

Ejemplo:

Queremos poner una caja para que el usuario indique su edad. En este caso resulta mucho más adecuado utilizar un **NumericUpDown** más que un **TextBox**. Pondremos el valor 18 a la propiedad **Minimum** (el usuario debe ser mayor de edad) y el valor 110 a la propiedad **Maximum**. La propiedad **DecimalPlaces** la dejaremos en 0.

```
int edad;  
  
edad = (int)numericUpDown1.Value;
```

Es importante hacer la conversión de tipo de decimal a int puesto que en caso contrario el compilador da un error.



1.4.3. Cajas para fechas (DateTimePicker)

Estas son un tipo de **TextBox**, especiales para introducir fechas. Cuando el programa está en ejecución podemos escoger la fecha mediante un calendario que se despliega cuando hacemos clic en una flecha pequeña que hay en la parte derecha del control.

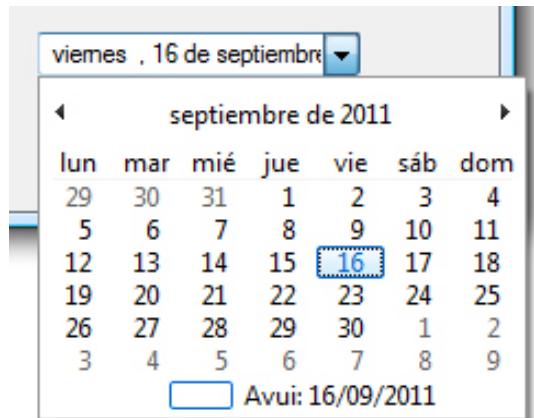


Figura36: DateTimePicker

De esta manera reducimos a cero la posibilidad de tener un dato con el formato incorrecto en la caja de Texto. Una propiedad interesante para este control es la llamada **Format** que básicamente lo que controla es el formato con que se presenta la fecha a la caja de texto. En la imagen anterior, el valor de la propiedad **Format** es **Long**, y el resultado es la visualización de la fecha en formato completo; en cambio, si utilizamos el valor **Short**, el formato de la fecha es el clásico **DD/MM/AAAA**. Entonces, de la misma forma que en el caso anterior, la fecha introducida por el usuario la tenemos en la propiedad **Value** que es del tipo **DateTime**, pero lo que hay que aprender son las propiedades que se necesitan para manipular el contenido:

- **Day:** Día del mes
- **DayOfWeek:** Es un tipo enumerado que contiene el día de la semana.
- **DayOfYear:** Día del año del 1 al 366.
- **Month:** Número del mes.
- **Year:** Número del año.
- **Today:** Fecha correspondiente al día de hoy.



Ejemplo:

Queremos hacer una aplicación en la que el usuario pueda indicar un día laboral para pedir fiesta:

```
DateTime dia;  
dia = datePicker1.Value;  
if ((dia.DayOfWeek >= DayOfWeek.Monday) && (dia.DayOfWeek <= DayOfWeek.Friday))  
    textBox1.Text = "Fecha Acceptada!";  
else  
    textBox1.Text = "Debes introducir un dia laborable";
```

Notad que, para facilitar la manipulación de la fecha asignada por el usuario, la guardamos de forma temporal en una variable de tipo **DateTime**.

1.4.4. Casillas de Verificación (CheckBox)

Estas son un tipo de control que sirve para indicar si algún elemento está activado o no. Por esta razón la propiedad fundamental es **Checked** que es una variable del tipo **bool** que puede tener los estados **True** (casilla activada) o **False** (casilla desactivada).

Ejemplo:

Diseñamos un formulario donde ponemos un **CheckBox**, un **Label** y un **Button** como se muestra en la siguiente imagen:

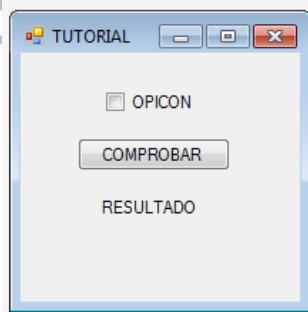


Figura: CheckBox

Entonces generamos un evento **Click** para el botón y ponemos el código siguiente:

```
if (checkBox1.Checked)  
    label1.Text = "CASILLA ACTIVADA";  
else  
    label1.Text = "CASILLA DESACTIVADA";
```



Comprobamos, como en efecto, si marcamos la casilla y pulsamos el botón, aparece el mensaje de casilla activada, y en cambio, si desmarcamos la casilla, el mensaje es casilla desactivada.

Ahora ponemos un segundo **CheckBox** en la aplicación. Podemos comprobar que cada uno de ellos es independiente del otro; es decir, que podemos marcar uno, el otro, los dos o ninguno. Si queremos que la etiqueta muestre un mensaje indicando cuáles de las dos casillas están activas, el código será:

```
if (checkBox1.Checked)  
    label1.Text = "CASILLA 1 ACTIVADA";  
  
else  
    label1.Text = "CASILLA 1 DESACTIVADA";  
  
if (checkBox2.Checked)  
    label1.Text = "CASILLA 2 ACTIVADA";  
  
else  
    label1.Text = "CASILLA 2 DESACTIVADA";
```

Resulta evidente que este sistema presenta un inconveniente a medida que el número de **CheckBox** aumenta. Solo hay que hacer un pequeño esfuerzo para imaginar la complejidad del código si tenemos diez casillas o más. Para resolver este tipo de problemas, la solución es agrupar los controles en un Array. De esta manera podemos utilizar bucles para procesar toda la lista de casillas y el programa es independiente del número de casillas utilizadas.

```
label1.Text = "";  
  
CheckBox[] cajas = new CheckBox[] {checkBox1, checkBox2, checkBox3,  
checkBox4, checkBox5, checkBox6, checkBox7, checkBox8, checkBox9, checkBox10};  
for (int i = 0; i < 10; i++)  
{  
    if (cajas[i].Checked)  
        label1.Text += "CASILLA " + i+1 + " ACTIVADA\n";  
    else  
        label1.Text += "CASILLA " + i+1 + " DESACTIVADA\n";  
}
```

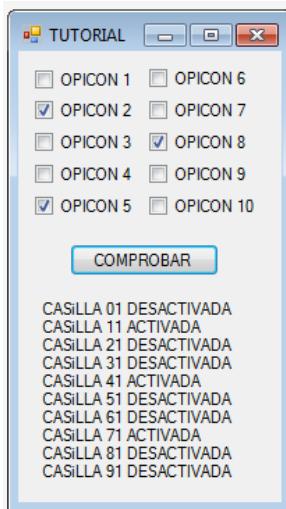


Figura: CheckBox (2)

En algunas ocasiones lo que nos interesa es reaccionar inmediatamente a la acción de marcar o desmarcar la casilla en lugar de esperar más adelante para comprobar el estado de la propiedad **Checked**. En estos casos lo que hemos de hacer es generar un evento del tipo **CheckedChanged** para la casilla de verificación.

```
private void checkBox1_CheckedChanged(object sender, EventArgs e)
```

Figura: Evento CheckedChanged

Este evento se produce cada vez que, por la razón que sea, el estado de la caja (valor de la propiedad **Checked**) cambia.

Ejemplo:

Diseñamos un formulario donde ponemos un CheckBox y un Label; entonces generamos un evento **CheckedChanged** para el **CheckBox** y escribimos el siguiente código:

```
if (checkBox1.Checked)
    label1.Text = "CASILLA ACTIVADA";
else
    label1.Text = "CASILLA DESACTIVADA";
```



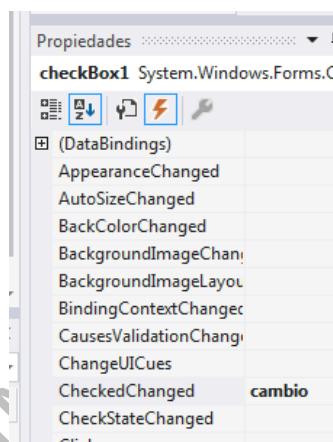
Notad que ahora la etiqueta cambia en el momento exacto en que la casilla modifica su estado.

¿Qué pasaría en un caso similar al anterior donde tenemos más de un **CheckBox** a controlar? En este caso, no nos sirve la técnica de poner las casillas de verificación en un Array porque lo que necesitamos es generar un evento **CheckedChanged** para cada una de ellas. Para solucionar esto, utilizaremos los argumentos del evento.

Todos los eventos tienen dos argumentos entre paréntesis:

- **sender**: Es el objeto que ha generado el evento.
- **e**: Son todos aquellos datos adicionales del evento y que nos pueden ser de utilidad.

Lo primero que vamos a hacer es escribir el evento de cambio para el primero de los checkbox:



En nuestro caso, vamos a utilizar el argumento **sender** para averiguar qué casilla ha cambiado. El problema está en que el argumento **sender** es del tipo **object**, es un tipo muy genérico y no dispone de propiedades y métodos de utilidad (como podemos apreciar en la siguiente imagen):

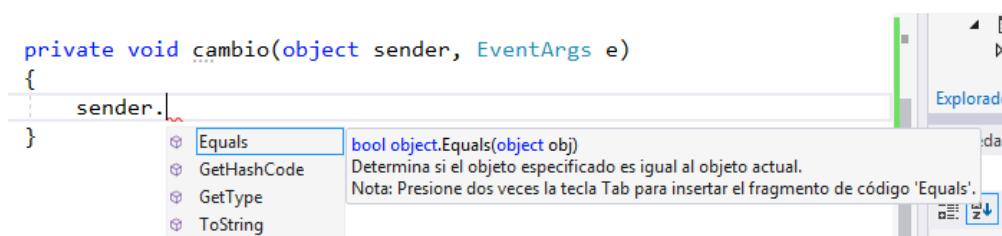


Figura: Propiedades del Objeto



A nosotros nos interesa poder referirnos a la propiedad **Checked** del **sender** para utilizarla en la sentencia condicional. Para poder conseguirlo, hemos de transformar el tipo del objeto **Sender** de **object** (tipo genérico) a **CheckBox** (tipo más concreto). Pero es evidente que esta conversión solo se podrá hacer si realmente el objeto que ha generado el evento era un **CheckBox**, si no es así, el programa generaría una excepción y se pararía. Esto no es ningún problema, porque nosotros sabemos que, en este caso, **sender SIEMPRE** será un **CheckBox** y por tanto, no hay ningún peligro en realizar la conversión de tipos sin necesidad de recurrir a un try/catch.

```
CheckBox caja = (CheckBox)sender;
```

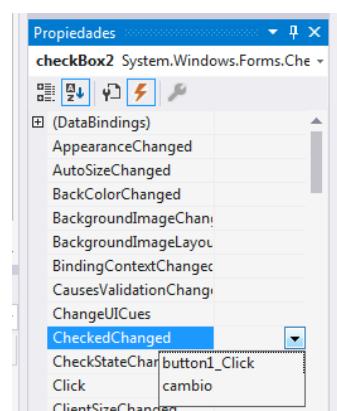
El código completo para el evento quedará de la siguiente manera:

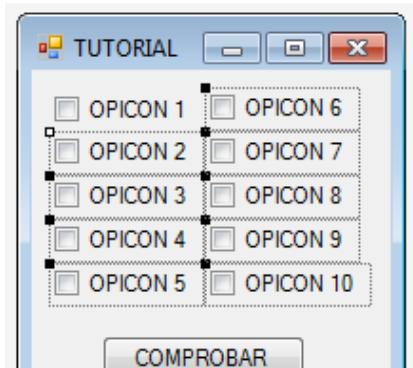
```
CheckBox caja = (CheckBox)sender;  
  
if (caja.Checked)  
  
    label1.Text = "CASILLA " + caja.Text + " ACTIVADA";  
  
else  
  
    label1.Text = "CASILLA " + caja.Text + " DESACTIVADA";
```

Fijémonos que ahora el código es absolutamente independiente del nombre que tenga el control **CheckBox** que genera el evento. Si ponemos dos casillas de verificación en lugar de una, lo único que hemos de hacer es generar dos eventos **CheckedChanged** y el código es idéntico para los dos.

Resulta evidente que solo hemos resuelto la mitad del problema, porque si tuviéramos diez casillas, tendríamos que repetir diez veces el código. Para solucionar este pequeño obstáculo lo único que hay que hacer es en cada casilla que debe utilizar el mismo evento en la ventana de eventos en vez de escribir un nombre de evento en el cuadro de texto utilizamos el desplegable para elegir un evento ya existente. En nuestro caso el evento cambio programado antes.

Repetir el proceso para todas las casillas que deban usar el evento cambio. Lo podemos hacer con facilidad si seleccionamos todas las casillas (pulsando la tecla mays) y activando el evento cambio. Todas quedan alteradas.





Esta técnica resulta de gran utilidad cuando hay que modificar la misma propiedad a muchos controles.

1.4.5. Botones de Selección (RadioButton)

Los botones de selección funcionan de una forma prácticamente idéntica a cómo funcionan las casillas de verificación, con la diferencia que solo podemos activar uno a la vez; es decir, si activamos un segundo botón, el primero se deselecciona de forma automática. Esto permite que el usuario pueda escoger una Y SOLO UNA, opción de un grupo. La propiedad que utilizamos en **Checked** es igual que en el caso anterior.

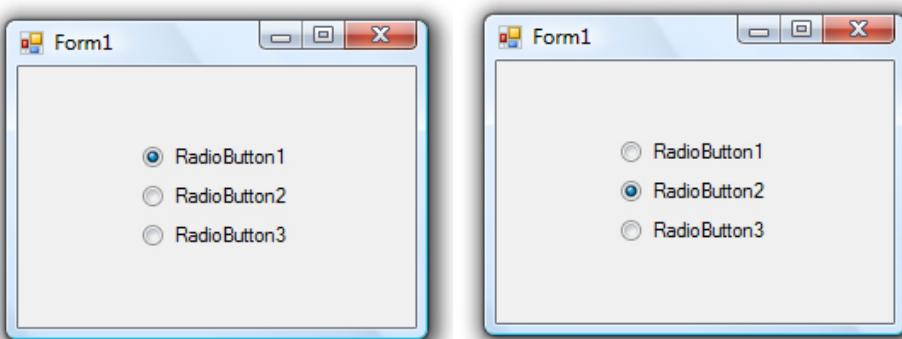


Figura: RadioButton

Si quisieramos crear dos agrupaciones de botones para escoger dos características diferentes como se muestra en el gráfico:

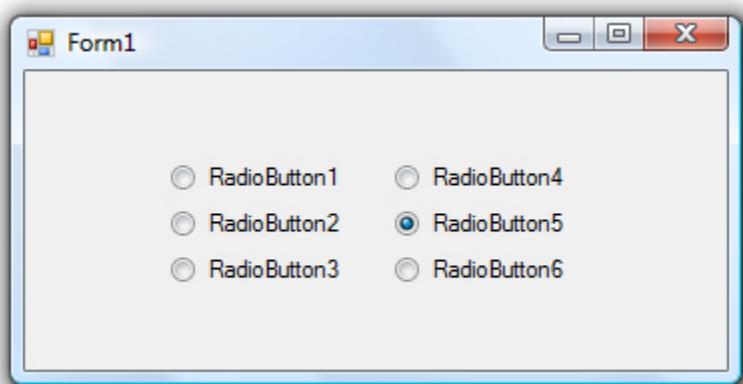


Figura: Grupos de RadioButton

Tendríamos un problema, y es que de los seis botones, solo podemos activar uno. Para solucionarlo, hemos de agrupar los botones en dos grupos. Para conseguirlo, utilizamos alguno de los controles del grupo de contenedores.

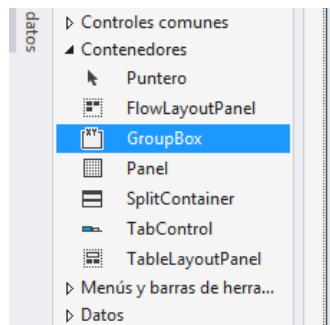


Figura: Controles contenedores

Estos controles permiten formar grupos autónomos de controles y de esta manera, podemos activar un botón de selección por grupo. En este caso, el control más idóneo sería el **GroupBox** ya que dibuja una marca alrededor del grupo y nos permite ponerle un título.

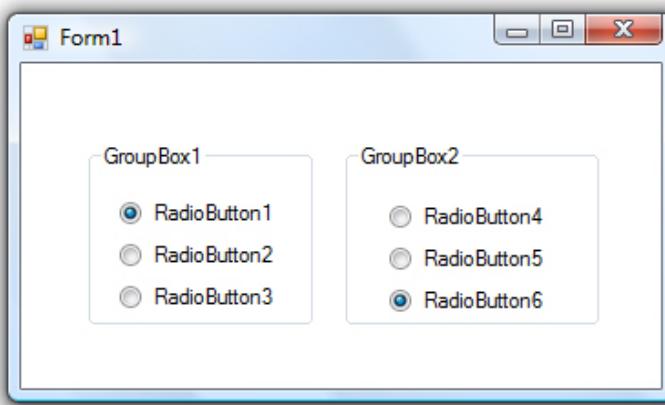


Figura: RadioButtons agrupados



1.4.6. Caja de Imagen (PictureBox)

Este control es un contenedor rectangular donde podemos cargar una imagen (con los formatos típicos bmp,gif,png,jpg,..) del disco.

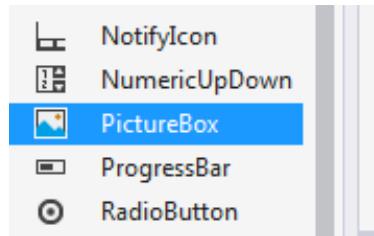


Figura: Control PictureBox

Para asignar una imagen al **PictureBox** en tiempo de diseño, utilizaremos la ventana de propiedades y localizamos la propiedad **Image**.

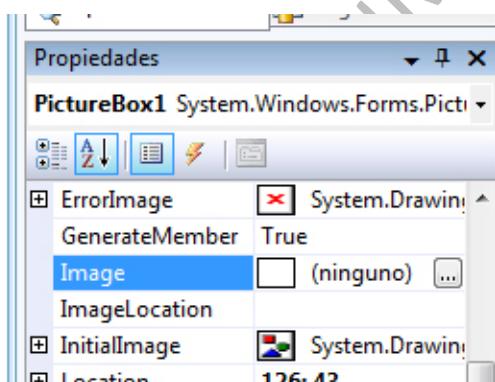


Figura: Propiedad Image

Cuando editamos esta propiedad, nos aparece una ventana que nos permitirá seleccionar la imagen deseada del disco.

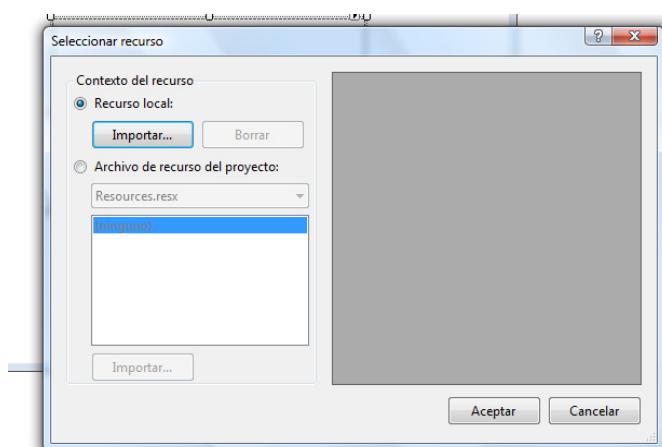


Figura: Selección de Imagen



Una vez cargada la imagen, podremos ver como esta se muestra dentro del **PictureBox**. La imagen cargada no se redimensiona y por tanto, mantiene el tamaño que tenía originariamente; esto puede hacer que no se vea correctamente porque el tamaño del **PictureBox** sea demasiado pequeño. Si queremos modificar la forma con la que la imagen se muestra dentro del **PictureBox**, lo hacemos con la propiedad **SizeMode** que puede tener los siguientes valores:

- **Normal**: la imagen se muestra en su tamaño real a partir de la esquina superior izquierda del **PictureBox**, si parte de la imagen no cabe al **PictureBox**, entonces no se muestra.
- **StretchImage**: La imagen ocupa toda la superficie del **PictureBox**. Esta se aumenta o se reduce según convenga y como ha de ocupar toda la superficie del **PictureBox** se deformará si es necesario.
- **AutoSize**: La caja de imagen cambia de tamaño (aumentando o disminuyendo) para adaptarse al tamaño de la imagen.
- **CenterImage**: Es como la vista Normal pero **PictureBox** se sitúa centrado en la imagen y no en la esquina superior de la izquierda.
- **Zoom**: La imagen se redimensiona (aumentando o reduciéndose de tamaño) para ocupar la superficie del **PictureBox** de la manera más eficiente posible pero sin ninguna deformación.

Si queremos cargar o modificar una imagen en tiempo de ejecución, hemos de utilizar la clase **Bitmap**, que contiene, entre otros, un método para cargar imágenes desde un fichero. Tener en cuenta que en C# al escribir una ruta entre comillas hay que doblar las contra barras.

```
pictureBox1.Image = Bitmap.FromFile("C:\\\\Users\\\\Public\\\\Pictures\\\\Sample Pictures\\\\Desert.jpg");
```

1.4.7. Lista Desplegable (ComboBox)

Los últimos tres controles que vamos a ver sirven para que el usuario pueda escoger una o más opciones de una lista. El más simple de los controles que hay en la lista desplegable es una lista, **ComboBox**, este control consiste en una lista oculta que podemos desplegar para seleccionar una de las opciones que queda visible, una vez plegada de nuevo la lista.

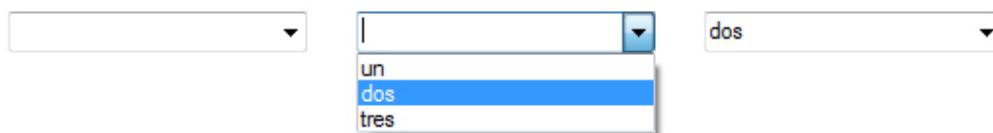


Figura: Selección con un ComboBox

Lo primero que hemos de resaltar es que la lista de elementos no es un objeto simple sino una colección. Primero vamos a ver cómo gestionar esta colección en tiempo de diseño. Para poder hacerlo, vamos a utilizar el inspector de propiedades y buscamos la propiedad que se llama **Items** donde observamos que es un elemento del tipo **Colección**. Entonces, hacemos clic en el pequeño botón con unos puntos suspensivos y aparece una ventana donde podemos escribir la lista de elementos que mostrará el desplegable.

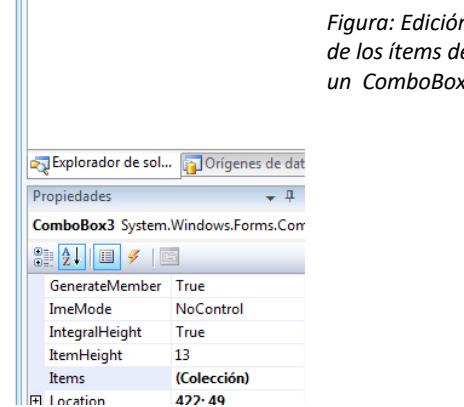
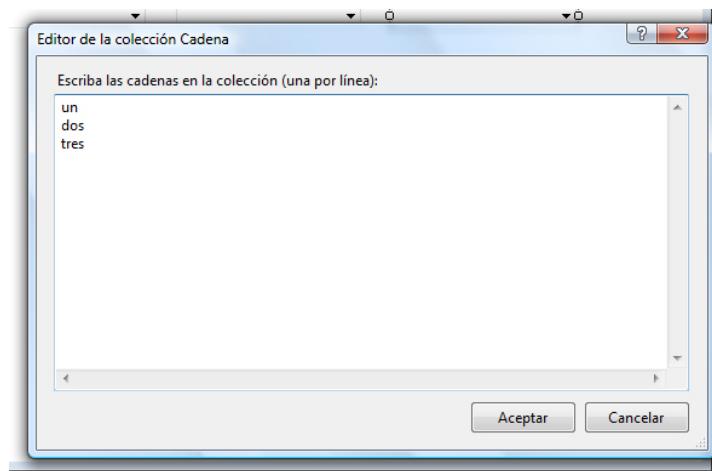


Figura: Edición de los ítems de un ComboBox

A continuación, hemos de aprender a gestionar el elemento seleccionado en tiempo de ejecución. Este queda guardado en la propiedad **SelectedItem** que es una cadena de texto que contiene el elemento escogido por el usuario. También podemos utilizar la propiedad **SelectedIndex** que es un entero y lo que contiene es la posición (tomando el 0 como primer elemento) del elemento escogido dentro de la lista de elementos.

Ejemplo:

Pondremos un **ComboBox** con una lista de elementos, cuando el usuario escoja uno de los elementos y pulse un botón, mostraremos en un **TextBox** el elemento escogido.



```
private void Button1_Click(object sender, EventArgs e)
{
    textBox1.Text = comboBox1.SelectedItem.ToString();
}
```

Podemos ver que el código es muy simple, solo hemos de coger la propiedad **SelectedItem** y asignarla al cuadro de texto (recordar que no se puede poner un numero directamente en el cuadro de texto hay que usar `ToString()`). Si lo que nos interesa es intervenir en el momento en que se hace la selección del elemento (sin tener que esperar que se haga clic en el botón), podemos utilizar el evento **SelectedIndexChanged** de la lista desplegable que se produce cada vez que el elemento seleccionado es modificado. De esta manera, tenemos el mismo código pero situado en un evento diferente:

```
private void ComboBox1_SelectedIndexChanged(object sender, EventArgs e)
{
    textBox1.Text = comboBox1.SelectedItem.ToString();
}
```

Finalmente, nos queda gestionar la colección de **Items** de la lista desplegable en tiempo de ejecución.

Ejemplo:

Pondremos un **ComboBox** en una lista de elementos, y un **TextBox** con un botón, para que el usuario pueda añadir nuevos elementos a la lista (que no estén en blanco ni repetidos).

```
private void Button1_Click(object sender, EventArgs e)
{
    if (textBox1.Text == "") return;
    if (comboBox1.Items.IndexOf(textBox1.Text) != -1) return;
    comboBox1.Items.Add(textBox1.Text);
}
```

Observamos que hemos utilizado el método **IndexOf** para buscar la posición de un elemento dentro de la lista con el texto que el usuario quiere añadir. La función **IndexOf** devuelve el índice del



elemento especificado o bien el valor -1 si este no se encuentra en la lista (que es justamente lo que nos interesa en este caso). Por tanto, cualquier valor diferente a -1 significaría que el elemento ya se encuentra en la lista y entonces estaría repetido. Para acabar, si todas las comprobaciones han sido correctas, añadimos el nuevo elemento a la lista, simplemente invocando el método **Add**.

Observamos que para manipular la colección de los elementos de la lista utilizamos la notación:

combobox.Items.<Método>

NUNCA lo hemos de hacer directamente sobre el objeto **Combobox**:

Combobo1.Add (text)

Si queremos mejorar la funcionalidad de nuestro programa, podemos añadirle un botón que permita eliminar de la lista el primer elemento.

```
private void Button2_Click(object sender, EventArgs e)
{
    comboBox1.Items.RemoveAt(0);
}
```

Comprobamos que no resulta muy difícil gestionar los elementos de la lista en tiempo de ejecución.

1.4.8. Lista (ListBox)

Este control funciona de manera muy parecida al control que hemos visto antes. El aspecto es un poco diferente en este caso porque en este caso, los elementos están a la vista (no están escondidos en un desplegable). Pero hay una diferencia más importante, y es que este control nos permite la selección de múltiples elementos de la lista y no solo de uno. Para permitir esta selección múltiple, debemos modificar el valor de la propiedad **SelectionMode** al valor **MultiSimple** o **MultiExtended** (cualquiera de estos dos permite la selección de más de un elemento, pero con la opción **MultiExtended** podemos hacer una selección masiva, mientras que con la **MultiSimple** hay que escoger los elementos uno a uno).

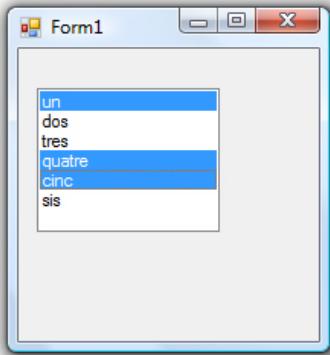


Figura: ListBox con selección múltiple

La gestión de la colección de los elementos de la lista es exactamente idéntica a la de un **ComboBox** pero ahora cambia la gestión del elemento seleccionado, ya que en este caso, pueden ser más de uno. En estos momentos, el conjunto de los elementos seleccionados se comporta como una nueva colección llamada **SelectedItems** y con las mismas características que la de **Items**.

Ejemplo:

Vamos a programar un botón que borre de una lista todos los elementos seleccionados por el usuario.

```
int num = listBox1.SelectedItems.Count;  
for (int i=0;i<num;i++)  
{  
    listBox1.Items.Remove(listBox1.SelectedItems[0]);  
}
```

Fijarse que estamos haciendo un bucle que recorre toda la colección de los elementos escogidos. Para cada uno de ellos lo borramos de la colección principal. Es importante que os fijéis que el argumento de la función **Remove** es SIEMPRE el primer elemento de la colección de elementos seleccionados, porque al eliminar un elemento de la lista, también lo estamos eliminando de la lista de seleccionados, en consecuencia, siempre hemos de borrar el primer elemento de la lista. Esa también es la razón por la que debemos crear una variable temporal **num** para almacenar los elementos a borrar ya que al borrar cada uno el tamaño de la colección **SelectedItems** se reduce por lo que no podemos usarla como control del comando **for**.



1.5. Dispositivos de entrada

1.5.1. Eventos del Teclado

Es de gran importancia en los proyectos de Visual Studio .NET poder controlar y gestionar los diferentes dispositivos de entrada que interactúan con nuestra aplicación. Los más comunes son el teclado y el ratón. Aunque en la mayoría de los casos, vamos a dejar que el usuario manipule el teclado y el ratón como quiera y solo reaccionaremos a botones en algunos casos, hay que intervenir de forma inmediata a la pulsación de teclas o botones del ratón.

Respecto al teclado disponemos de tres eventos:

- **KeyPress:** Se produce cuando se pulsa y se suelta una tecla que tenga un código ASCII asociado, es decir, todos los caracteres alfanuméricos más las teclas RETURN, BACKSPACE y ESCAPE). El evento nos proporciona (como parámetro) el carácter pulsado.

Ejemplo:

Creamos un nuevo proyecto y añadimos una caja de texto (TextBox1) y una etiqueta (Label1). A continuación desde la ventana de código seleccionamos el evento **KeyPress** para el objeto TextBox1.

El programa nos genera, en este momento, una cabecera de evento como la siguiente:

```
private void textBox1_KeyPress(object sender, KeyPressEventArgs e)  
{  
}
```

Anteriormente hemos trabajado con el **Sender** de la cabecera de los eventos; ahora vamos a trabajar con el objeto **e** que contiene toda la información adicional que necesitamos para gestionar adecuadamente el evento. En nuestro caso, contiene la tecla pulsada. Añadimos la línea siguiente al código del evento:

```
label1.Text = e.KeyChar
```



Esta línea lo que hace es mostrar la tecla pulsada al **Label**. Notad que el evento reacciona de forma inmediata a cualquier pulsación de tecla (menos a las que no corresponden a un código ASCII, como por ejemplo, las teclas de función, el control, el shift...).

- **KeyDown:** Es una versión más sofisticada del evento anterior. Las diferencias principales son que este reacciona a la pulsación de cualquier tecla (no solo las ASCII) y que se produce cuando apretamos la tecla y antes de soltarla. Esta vez el argumento no es una variable del tipo **Char** como antes, ya que las teclas no ASCII no tienen un carácter asociado, sino que nos devuelve la codificación UNICODE, utilizada por Windows, correspondiente a la tecla pulsada.

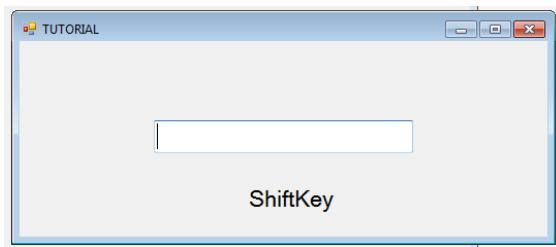
Ejemplo:

A partir del programa anterior: Comentamos el evento **KeyPress**. Generamos un evento **KeyDown** para el **TextBox1** y ponemos la siguiente línea:

```
Label1.Text = e.KeyCode.ToString();
```

Ahora lo que se muestra en el **Label** es el nombre de la correspondiente a la tecla pulsada en una colección enumerada.

Figura51: KeyCode



Esto evita saberse de memoria los códigos correspondientes a cada una de las teclas. Supongamos que queremos que la caja de texto se borre por completo cuando pulsamos F8; para poder hacerlo escribimos el siguiente código:

```
if (e.KeyCode == Keys.F8)
{
    textBox1.Text = "";
}
```

Notad que cuando escribimos el signo = a la instrucción If, el ordenador nos presenta una lista de las teclas para que escojamos la que queramos y de esta manera no es necesario memorizar los códigos.

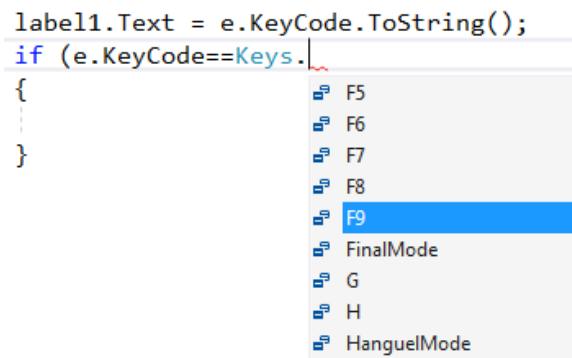


Figura: Evento KeyPress

Con este evento también podemos controlar las combinaciones de teclas (pulsaciones de teclas con las teclas de Control, alt o shift apretados). Podemos saber cuál de estas teclas está apretada en el momento del evento con las variables bool:

```
e.Control
```

```
e.Shift
```

```
e.Alt
```

Si modificamos el programa anterior para que la combinación de teclas que borra el textbox sea Ctrl+F8 queda de esta manera:

```
if ((e.KeyCode == Keys.F8) && e.Control)
{
    textBox1.Text = "";
}
```

- **KeyUp:** Funciona de forma idéntica al anterior, pero se produce cuando soltamos la tecla y no cuando la pulsamos.

1.5.2. Eventos del Ratón

Funcionan de una manera similar a los eventos del teclado, pero la información proporcionada por los argumentos, es más extensa. Este es un dispositivo más complejo de controlar que el teclado. Como en el anterior caso, tenemos un método sencillo para detectar simples pulsaciones del botón izquierdo del ratón (normal) y después, disponemos de una versión más compleja que nos proporciona toda la información, como por ejemplo, la posición del puntero, botones pulsados, giros de la ruedecita superior, etc.



Disponemos de los siguientes eventos:

- **MouseClick:** Se produce cuando hacemos clic con el botón izquierdo del ratón.
- **MouseDoubleClick:** Se produce cuando hacemos doble clic en el botón izquierdo del ratón.
- **MouseDown:** Se produce cuando apretamos cualquiera de los botones del ratón.
- **MouseUp:** Se produce cuando soltamos cualquiera de los botones del ratón.
- **MouseWheel:** Se produce cuando giramos la rueda superior del ratón.
- **MouseEnter:** Se produce cuando entra el puntero del ratón en el área ocupada por el control.
- **MouseLeave:** Se produce cuando sacamos el puntero del ratón del área ocupada por el control.
- **MouseHover:** Se produce cuando mantenemos el puntero del ratón sobre el área ocupada por el control durante unos instantes.
- **MouseMove:** Se produce cuando movemos el puntero del ratón sobre el área ocupada por el control.

Entre los argumentos proporcionados por los eventos pueden resultar útiles los siguientes:

e.Clicks (Integer): cantidad de veces que hemos pulsado el botón.

e.Location (Point): Posición del puntero del ratón en relación a la posición del control.

e.Button (MouseButtons): Botones pulsados del ratón.

e.Delta (Integer): Giro aplicado a la rueda superior del ratón.

Ejemplo:

Queremos desarrollar un mecanismo para desplazar una imagen (**PictureBox**) con el ratón. Para hacer esto, combinaremos los eventos **MouseDown**, **MouseUp** y **MouseMove**.

```
bool arrastre = false;  
Point offset;  
  
public Form1()  
{  
    InitializeComponent();  
}
```



```
private void pictureBox1_MouseDown(object sender, MouseEventArgs e)
{
    arrastre = true;
    offset = e.Location;
}

private void pictureBox1_MouseMove(object sender, MouseEventArgs e)
{
    if (arrastre)
    {
        int nuevaX = e.X + pictureBox1.Location.X - offset.X;
        int nuevaY = e.Y + pictureBox1.Location.Y - offset.Y;
        pictureBox1.Location = new Point(nuevaX, nuevaY);
    }
}

private void pictureBox1_MouseUp(object sender, MouseEventArgs e)
{
    arrastre = false;
}
```

Para controlar el proceso usamos una variable boolean **arrastre** que nos indica si debemos desplazar la imagen o no en el momento de mover el ratón, y la variable **offset** para recordar la distancia entre el puntero del ratón y la esquina superior izquierda, al empezar el proceso de arrastre y así poder mantener la imagen correctamente en cada ejecución del evento **MouseMove**. Notar que no podemos sumar directamente los objetos Point. Hay que sumar las coordenadas X e Y por separado y reconstruir el objeto.



Conceptos clave

- **Historia:** En este primer tema hemos visto una pequeña revisión histórica sobre el entorno de programación Visual Studio .NET, así como una explicación sobre las diferentes versiones existentes del entorno, con sus pros y sus contras.
- **Instalación:** También hemos visto cómo obtener e instalar nuestra versión del Visual Studio .NET.
- **Glosario:** De la misma forma hemos hecho una introducción de los conceptos de Objeto, Propiedad y Evento, que más adelante serán desarrollados con más detalle.
- **Primera Aplicación:** Hemos llevado a cabo nuestro primer proyecto en Visual Studio .NET en el cual hemos aprendido a añadir nuevos controles, modificar las propiedades en tiempo de diseño, mediante el inspector de propiedades, y también hemos aprendido a generar eventos que nos permitían modificar las propiedades de los objetos añadidos en tiempo de ejecución, mediante instrucciones de C#.
- **Elección de Controles:** Elegir el control mas adecuado para cada dato, recurriendo al cuadro de texto solo cuando no haya otro control mejor.
- **Propiedades:** Modificar la propiedades del control para que se ajuste de la mejor forma posible a las necesidades de la interfaz.
- **Control de Errores:** Evitar y controlar la inserción de datos incorrectos que pueden resultar en la finalización incontrolada del programa.
- **Elección de Controles:** Elegir el control mas adecuado para cada dato, recurriendo al cuadro de texto solo cuando no haya otro control mejor.



Test de autoevaluación

1. Como se llama el lenguaje intermedio que permite enlazar distintos lenguajes de programación en un único proyecto:
 - a) XML
 - b) ASP
 - c) MSIL
 - d) PHP

2. La primera vez que ejecutamos Visual Studio .NET:
 - a) Debemos elegir el idioma del entorno.
 - b) Debemos elegir nuestro lenguaje de programación preferido.
 - c) Es obligatorio conectarse con una cuenta Microsoft.
 - d) Tarda mas de lo normal.

3. Como se llama Visual Studio .NET la clase que representa las ventanas que desarrollamos:
 - a) Panel.
 - b) Frame.
 - c) Activity.
 - d) Form.

4. Cual de los siguientes controles es el mas adecuado para pedir al usuario cual es su navegador de internet preferido:
 - a) TextBox.
 - b) ComboBox.
 - c) CheckBox.
 - d) ListBox.



5. Cuál de los siguientes no es un valor válido para la propiedad `SizeMode` de un control

PictureBox:

a) Normal.

b) CenterImage.

c) MaxSize.

d) AutoSize.

6. Cuál de los siguientes no es un evento de ratón:

a) MouseLeft.

b) MouseLeave.

c) MouseHover.

d) MouseUp.

Ponlo en práctica

Actividad 1

Queremos diseñar la interfaz de un programa capaz de monitorizar el desarrollo de un partido de fútbol, parecido al que podemos encontrar en las webs de los principales periódicos deportivos. Para ello debemos permitir al usuario elegir los equipos (local y visitante) que van a disputar el partido. Una vez elegidos pondremos un botón que dará comienzo al partido. A partir de ese momento lo único que se puede hacer es modificar el marcador, es decir los goles de cada uno de los equipos.

Dispondremos de un botón que reiniciará todo el proceso para monitorizar otro partido.



Solucionarios

Test de autoevaluación

1. Como se llama el lenguaje intermedio que permite enlazar distintos lenguajes de programación en un único proyecto:
 - a) XML
 - b) ASP
 - c) **MSIL**
 - d) PHP

2. La primera vez que ejecutamos Visual Studio .NET:
 - a) Debemos elegir el idioma del entorno.
 - b) **Debemos elegir nuestro lenguaje de programación preferido.**
 - c) Es obligatorio conectarse con una cuenta Microsoft.
 - d) Tarda más de lo normal.

3. Como se llama Visual Studio .NET la clase que representa las ventanas que desarrollamos:
 - a) Panel.
 - b) Frame.
 - c) Activity.
 - d) **Form.**

4. Cual de los siguientes controles es el mas adecuado para pedir al usuario cual es su navegador de internet preferido:
 - a) TextBox.
 - b) **ComboBox.**
 - c) CheckBox.
 - d) ListBox.



5. Cual de los siguiente no es un valor válido para la propiedadSizeMode de un control PictureBox:

- a) Normal.
- b) CenterImage.
- c) **MaxSize.**
- d) AutoSize.

6. Cual de los siguientes no es un evento de ratón:

- a) **MouseLeft.**
- b) MouseLeave.
- c) MouseHover.
- d) MouseUp.

Ponlo en práctica

Actividad 1

Queremos diseñar la interfaz de un programa capaz de monitorizar el desarrollo de un partido de futbol, parecido al que podemos encontrar en las webs de los principales periódicos deportivos. Para ello debemos permitir al usuario elegir los equipos (local y visitante) que van a disputar el partido. Una vez elegidos pondremos un botón que dará comienzo al partido. A partir de ese momento lo único que se puede hacer es modificar el marcador, es decir los goles de cada uno de los equipos.

Dispondremos de un botón que reiniciará todo el proceso para monitorizar otro partido.

Solución:

Dispones de una posible solución en el archivo de solucionario (ver TEMA 1)