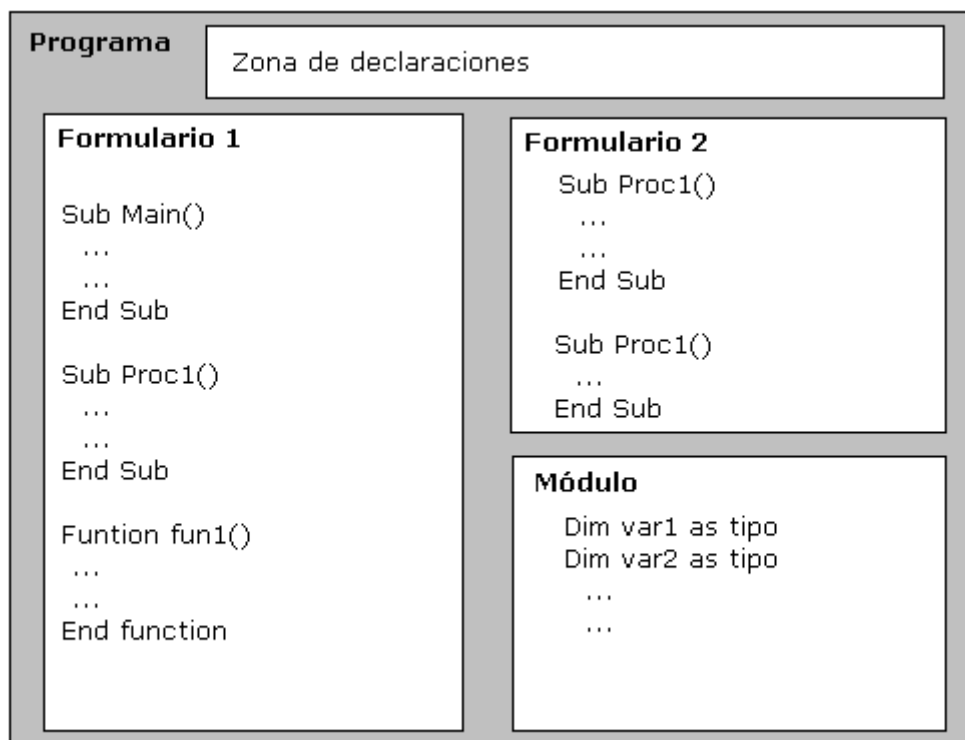


**TEMA 6****Procedimientos, funciones, módulos. Alcance****1. Módulos, Procedimientos y Funciones**

Ya estamos en condiciones de ver un programa completo con todas sus partes. Primero veremos un esquema de las partes en que se compone, algunas ya las hemos visto o tratado pero sin la suficiente profundidad.

El objetivo de VB.NET es generar un programa. Para esto disponemos de nuestro proyecto que tiene varios ficheros como ya comentamos. A estos ficheros que tienen extensión .VB les llamaremos de forma genérica contenedores físicos. En cambio, los contenedores lógicos son cada una de las partes en que se compone el fichero: módulos, clases o procedimientos. Recuerda el gráfico que vimos temas atrás:



Podemos ver que se divide en una zona de procedimientos y módulos y una Zona de declaraciones. Estas declaraciones están disponibles para todo el programa pero las que se realicen dentro de cada procedimiento sólo serán visibles desde dentro del procedimiento. Esto

era lo que llamábamos el alcance de las variables. Ahora lo que queremos ver con más profundidad es qué son y cómo se trabaja con módulos, procedimientos y funciones.

Un programa de Visual Basic se almacena en módulos de proyecto. Los proyectos se componen de archivos, que se compilan en aplicaciones. Al iniciar un proyecto y abrir el editor de código, verás que ya hay algo de código escrito en una determinada posición. Recuerda cuando analizamos nuestro primer proyecto en el tema 2 como veíamos que hay código oculto y código para iniciar la aplicación.

Este código que figura ya en el proyecto o que podemos incluir manualmente sigue esta secuencia:

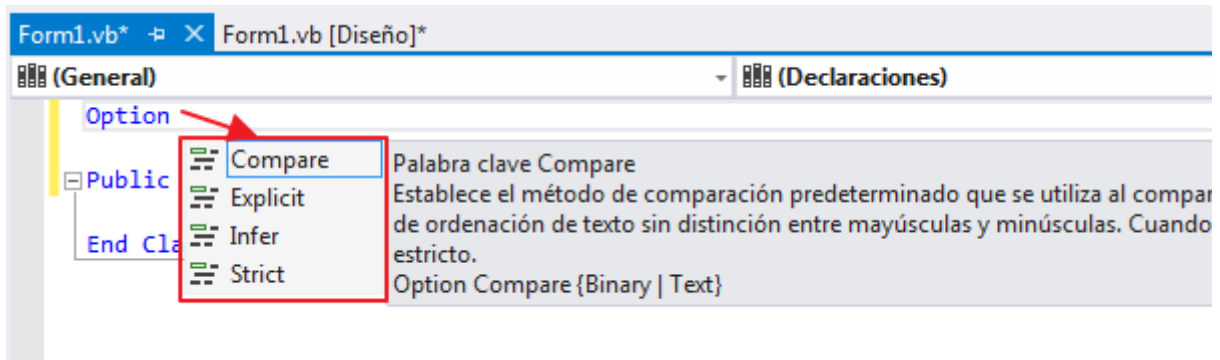
1. Instrucciones **Option**
2. Instrucciones **Imports**
3. Procedimiento **Main**
4. Instrucciones **Class**, **Module** y **Namespace**, si corresponde
5. Instrucciones de compilación condicional

Ahora veremos cada una de estas secciones para comprender de qué partes de código se compone un proyecto.

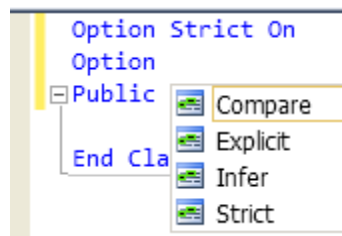
## 1.1. Instrucciones Option

Las instrucciones **Option** establecen reglas de base para el código y de esta forma ayudan a prevenir errores de sintaxis y de lógica. La instrucción **Option Explicit** asegura que todas las variables están declaradas y escritas correctamente, lo que reducirá el posible tiempo que deberá utilizarse posteriormente para depurar. La instrucción **Option Strict** ayuda a prevenir errores de lógica y pérdidas de datos que puedan producirse al trabajar entre variables de diferentes tipos. La instrucción **Option Compare** especifica la forma en que se comparan las cadenas entre sí, mediante su disposición de tipo **Binary** o **Text**. Hay una opción más: "**Infer**". Al establecerla en **On**, podemos declarar las variables sin especificar explícitamente un tipo de datos. El compilador deduce el tipo de datos de una variable a partir del tipo de su expresión de inicialización. Por ejemplo, si están desactivadas las instrucciones **Option Infer** y **Option Strict**, la variable en la declaración `Dim someVar = 2` se identifica únicamente como un objeto.

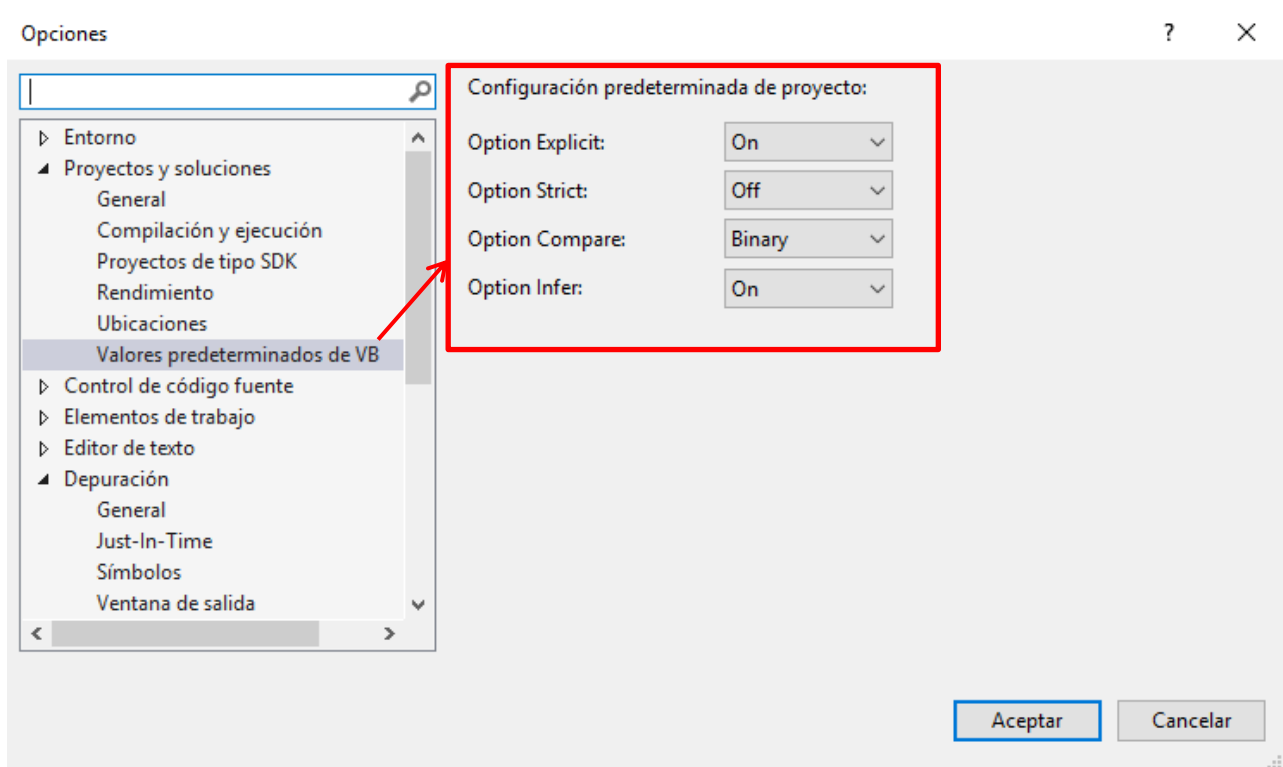
Estas instrucciones las podemos incluir en la parte superior del código:



Al escribir *Option* en la cabecera de un proyecto vacío, veremos que el IDE nos muestra las opciones posibles. Es una forma muy buena de asegurarnos de que escribimos en el sitio adecuado ya que si intentamos escribir estas instrucciones en otra parte del código no nos dejará.

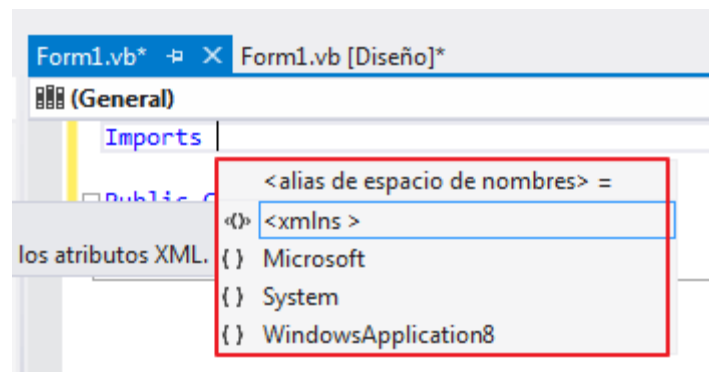


También vimos cómo estas cuatro opciones, que condicionan mucho el modo de funcionar del programa, se pueden poner desde las propiedades del proyecto. Si queremos que sean opciones permanentes para todos los proyectos lo podemos indicar aquí:

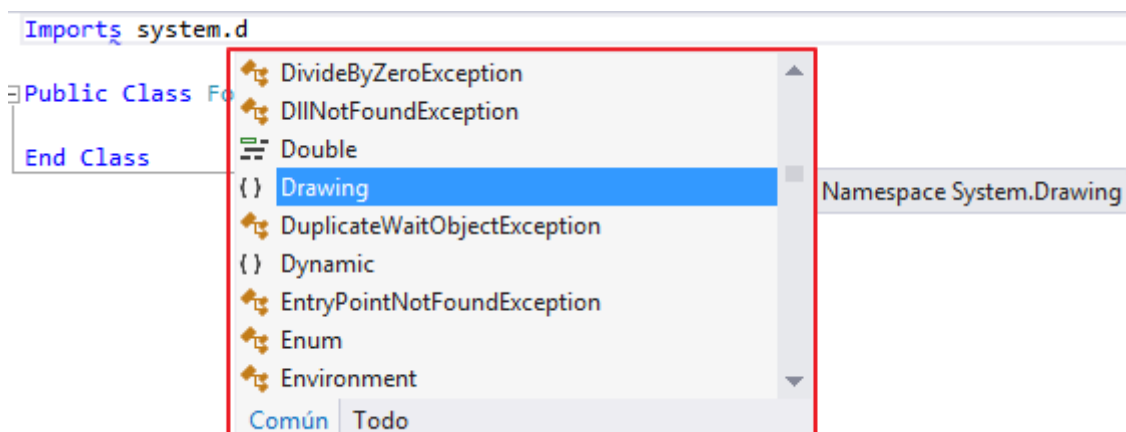


## 1.2. Instrucciones Imports

Las instrucciones **Imports** nos permiten poner nombre a clases y otros tipos definidos en el espacio de nombres importado sin tener que calificarlos. Ya sabemos qué son los "spacenames". Son los grupos de objetos que realizan tareas sobre el mismo entorno. Por ejemplo, para dibujar tenemos el spacename "System.Drawing" o para bases de datos "System.Data". Si quiero añadirlo escribiremos la instrucción Imports:



Y el editor me irá ayudando para escribirla correctamente, depende de System, así que escribimos "System.":



Con lo que he añadido a mi proyecto que importe el "spacename" del conjunto de rutinas gráficas:

```
Imports System.Drawing

Public Class Form1

End Class
```

Las instrucciones **Imports** importan los nombres de las entidades a un archivo de código fuente, permitiendo hacer referencias a los nombres sin calificación. La calificación significa indicarle el nombre completo

Recuerda que la biblioteca de clases de .NET Framework está constituida por espacios de nombres. Cada espacio de nombres contiene tipos que se pueden utilizar en el programa: clases, estructuras, enumeraciones, delegados e interfaces. Todos los espacios de nombres suministrados por Microsoft empiezan por System o por Microsoft.

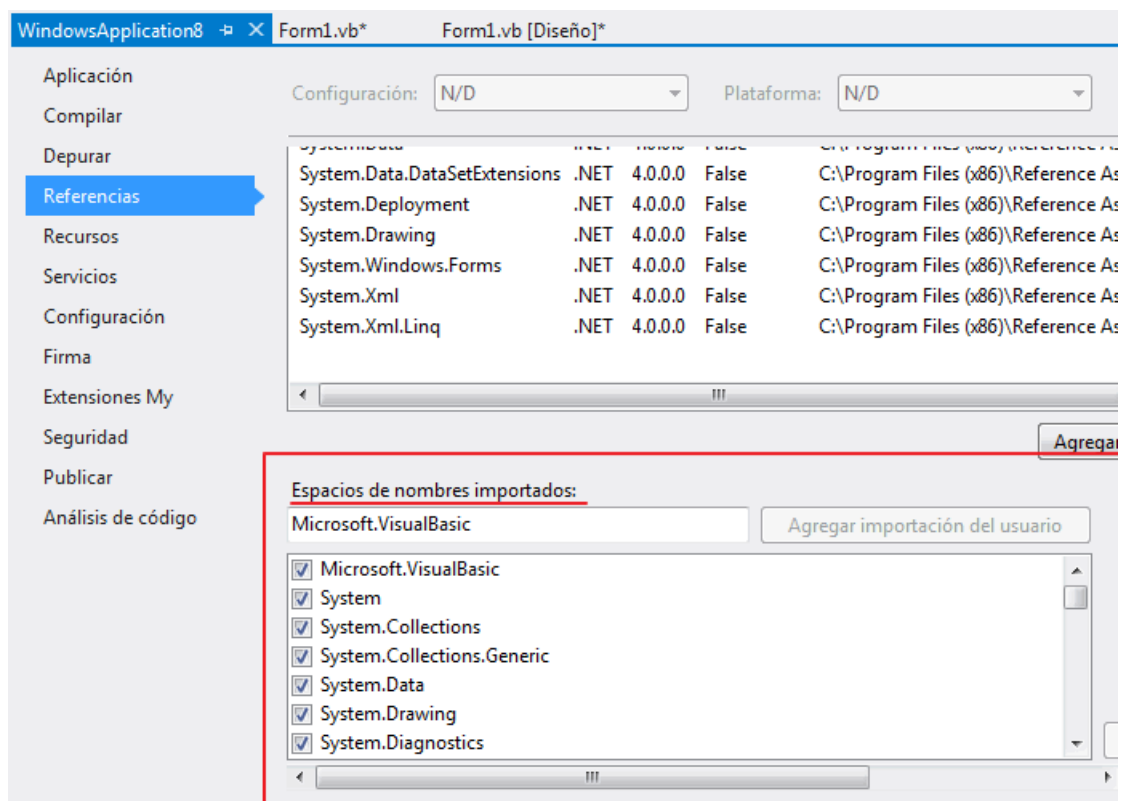
Si no añadimos los espacios de nombres tendremos que referirnos siempre al objeto con su **calificador completo**, por ejemplo, para hacer referencia a un color deberíamos poner todo esto:

```
System.Drawing.Color.AliceBlue
```

En caso de haberlo hecho nos bastaría con:

```
Color.AliceBlue
```

Si nos vamos a las propiedades del proyecto podemos ver lo siguiente en la sección de "Referencias"

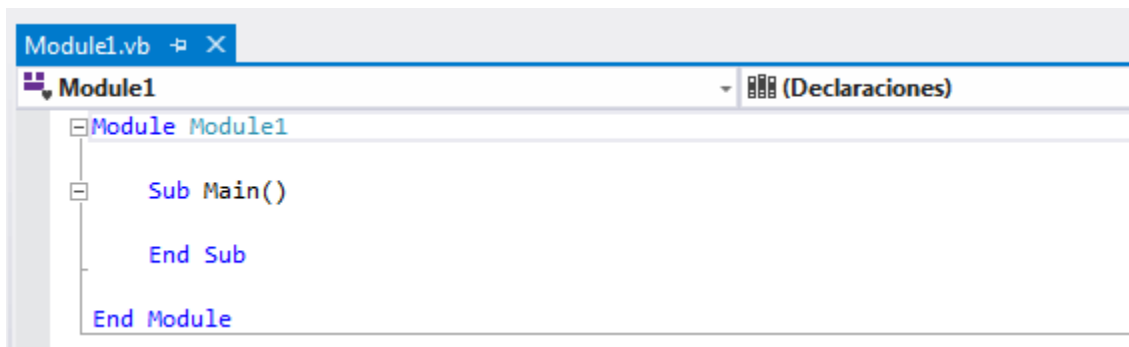


Estos son los componentes que ahora mismo reconoce nuestra aplicación. Se pueden añadir tantos como necesitemos. En la parte de abajo está la sección que nos importa ahora. ¿Qué espacios de nombres tiene añadidos mi proyecto? Esa es la lista de los espacios de nombres predeterminados. Por eso hay ya muchas instrucciones que nos reconoce directamente, ya que al estar importado por defecto a nuestro proyecto, no tenemos que escribir la instrucción con su espacio de nombres.

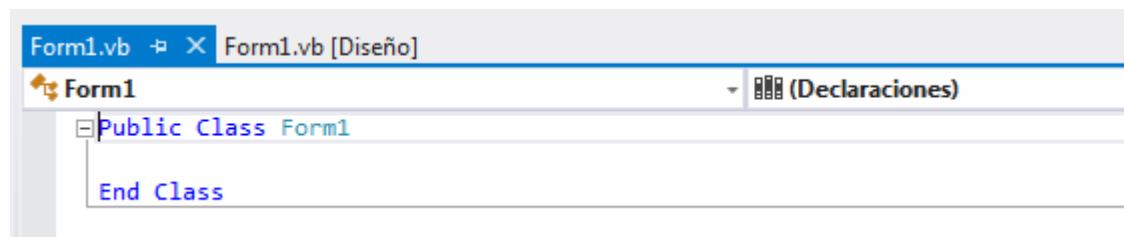
### 1.3. Procedimiento Main

El procedimiento **Main** es el "punto inicial" de una aplicación de tipo consola. Por tanto, **Main** señala el lugar que corresponde al código al cual debe obtenerse acceso en primer lugar. En **Main** se puede saber si se está ejecutando una copia de la aplicación en el sistema, establecer un conjunto de variables para la aplicación o abrir una base de datos que necesitemos en la aplicación.

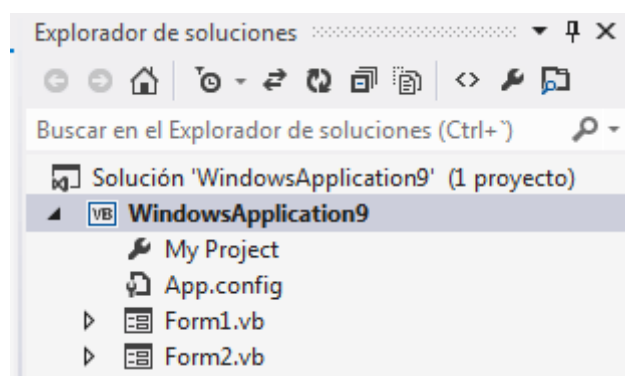
Recuerda que en una aplicación de consola es lo primero que aparece:



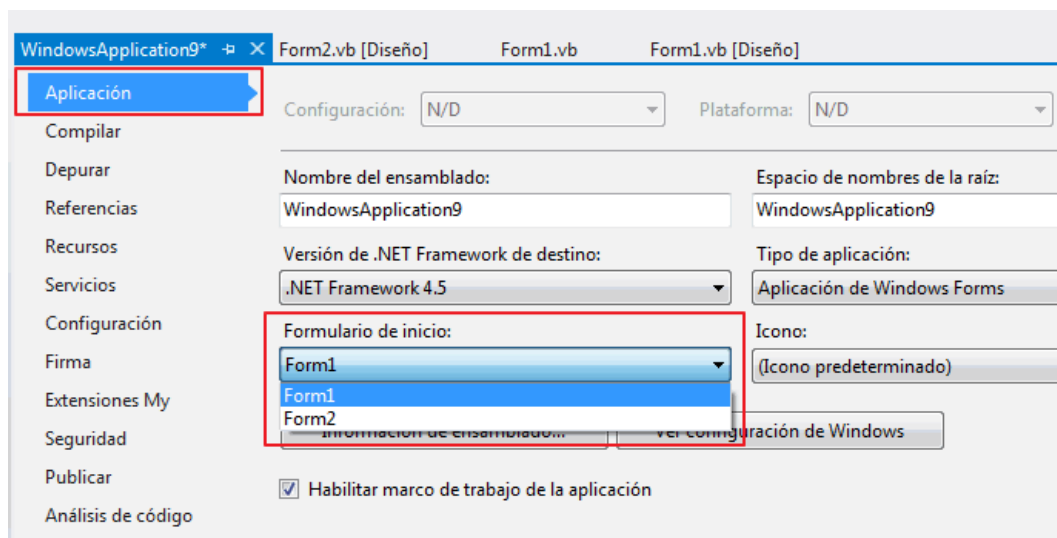
En cambio una aplicación Windows tiene como código base:



Donde el objeto inicial es en este caso el formulario Form1. Si tuviéramos otro formulario y queremos que ese segundo sea el formulario de inicio se lo tendríamos que indicar en las propiedades del proyecto. Por ejemplo, este proyecto con dos formularios:



Queremos que el formulario de inicio sea el segundo, ya que pediremos, por ejemplo, unos datos de conexión o un usuario y contraseña. Para que inicie en el que nosotros queramos lo indicamos en las propiedades del proyecto:



## 1.4. Instrucciones Class, Module y Namespace

Las clases y los módulos forman la mayoría del código del archivo de código fuente. Contienen la mayor parte del código que se escribe; principalmente instrucciones **Sub**, **Function**, **Method** y **Event**, junto con declaraciones de variable y otro código necesario para que funcione la aplicación.

Es el grueso del programa, donde tendré todos mis procedimientos y funciones. También toda la detección de los eventos de los componentes de los formularios: clic en los botones, carga de formularios, eventos de teclado,...

## 1.5. Instrucciones de compilación condicional

Las instrucciones de compilación condicional pueden aparecer en cualquier lugar dentro del módulo. Están configuradas para ejecutarse si se cumplen determinadas condiciones en tiempo de ejecución. También se pueden utilizar para depurar la aplicación, ya que el código condicional se ejecuta únicamente en modo de depuración.

## 1.6. Recapitulando

Los programas se componen de varios ficheros, algunos de ellos pueden ser los formularios y otros los módulos. Los formularios son los que tienen componentes de interfaz de usuario y el código para controlar estos componentes. Los módulos no contienen información gráfica, es decir, sólo son archivos de código: para definir procedimientos y funciones globales que se van a utilizar en todo el proyecto, definiciones de clases para utilizarse también en todo el proyecto,...

Inicialmente nuestros proyectos serán de un sólo formulario y no tendremos que preocuparnos de mucho más pero en cuanto crezcan un poco y pasen a tener varios formularios necesitaremos mecanismos para que ambos compartan rutinas o variables.

Ahora veremos los procedimientos que son parte fundamental de los programas.

## 2. Procedimiento (Sub)

Un procedimiento **Sub** consiste en una serie de instrucciones de Visual Basic delimitadas por las instrucciones **Sub** y **End**. Cada vez que se llama a un procedimiento, se ejecutan las instrucciones de éste. Comienza con la primera instrucción ejecutable (ya que pueden existir líneas de comentarios) tras la instrucción **Sub** hasta la primera instrucción **End Sub**, **Exit Sub** o **Return** que se encuentre.

Un procedimiento **Sub** ejecuta acciones, pero no devuelve ningún valor. Puede tomar argumentos, como constantes, variables o expresiones, que le pasa el código de llamada.

La sintaxis para declarar un procedimiento **Sub** es la siguiente:

```
[visibilidad] Sub nombreproc([listaargumentos])  
  
    ' Instrucciones para realizar el proceso.  
  
End Sub
```

La visibilidad o accesibilidad puede ser **Public**, **Protected**, **Friend**, **Protected Friend** o **Private**. Lo veremos más adelante pero recuerda que si teníamos una visibilidad para las variables también las tendremos para estos procedimientos. Son **Public** de forma predeterminada, lo que significa que se les puede llamar desde cualquier parte de una aplicación.

Los procedimientos **Sub** pueden definirse en módulos, clases y estructuras. De momento, sólo los utilizaremos en nuestros módulos

### 2.1. Declaración de argumentos

Los argumentos son los valores que necesita para realizar el proceso. Por ejemplo, si creamos un procedimiento para buscar un dato en un fichero, los argumentos serán el fichero y el dato a buscar. Pero no son obligatorios, en muchas ocasiones podemos llamar a un procedimiento sin indicarle ningún parámetro, por ejemplo para que ponga en negrita un dato en pantalla o abrir las conexiones con las bases de datos de nuestra aplicación.

Los argumentos de un procedimiento se declaran igual que las variables, especificando el nombre y el tipo de datos del argumento. También puede especificarse el mecanismo que se va a utilizar para pasar el argumento, así como si se trata de un argumento opcional.

La **sintaxis de los argumentos** en una lista de argumentos es la siguiente:

```
[Optional] [ByVal|ByRef] [ParamArray] nombreargumento As tipodatos
```

Si el argumento es opcional, la declaración de éste debe contener también un valor predeterminado, como se muestra a continuación:

```
Optional [ByVal|ByRef] nombreargumento As tipodatos = valorpredeterminado
```



## 2.2. Sintaxis de llamada

Los procedimientos **Sub** se invocan de forma explícita, con una instrucción de llamada independiente. No se les puede llamar utilizando su nombre en una expresión. La instrucción de llamada debe suministrar el valor de todos los argumentos que no sean opcionales, y debe incluir la lista de argumentos entre paréntesis.

La sintaxis para llamar a un procedimiento **Sub** es la siguiente:

`Nombresub [(Listaargumentos)]`

El procedimiento **Sub** que aparece a continuación notifica al usuario del equipo la tarea que está a punto de realizar la aplicación, y también muestra una marca de tiempo. En lugar de duplicar este fragmento de código al principio de cada tarea, la aplicación simplemente llama a "AvisaOperador" desde varios lugares. Cada llamada pasa una cadena al argumento Tarea que identifica la tarea que se va a iniciar.

```
Sub AvisaOperador(ByVal Tarea As String)
    Dim hora_inicio As Date      ' Variable local de tipo fecha.
    hora_inicio = TimeOfDay()    ' Obtiene la hora actual.
    ' Utilizamos el messagebox para escribir un texto.
    MessageBox.Show("Iniciando " & Tarea & " a las " & CStr(hora_inicio))
End Sub
```

Una llamada a "AvisaOperador" sería:

```
AvisaOperador("Actualizar fichero")
```

Delante de las llamadas se puede poner la palabra "Call" pero es opcional y no hace nada, más que otra cosa en por compatibilidad con otras versiones.

A lo largo del proyecto iremos creando muchos procedimientos para ir realizando pequeños procesos. Dividiremos tareas complicadas en pequeños Sub que nos facilitarán su depuración y mantenimiento.

Es decir, en lugar de escribir 100 líneas de código seguidas para resolver un programa, lo estructuraremos en pequeños fragmentos que irán realizando pequeños procesos para en conjunto, resolver el problema principal. ("**divide y vencerás**") es la primera frase que se enseña a los programadores)

Simplificará la tarea de programación ya que al dividir los programas en componentes lógicos más pequeños serán más fáciles de tratar. Los procedimientos resultan muy útiles para condensar las

tareas repetitivas o compartidas, como cálculos utilizados frecuentemente, manipulación de texto y controles y operaciones con bases de datos.

Hay dos ventajas principales cuando se programa con procedimientos:

- Los procedimientos nos permiten dividir los programas en unidades lógicas discretas, cada una de las cuales se puede depurar más fácilmente que un programa entero sin procedimientos.
- Los procedimientos que se utilizan en un programa pueden actuar como bloques de construcción de otros programas, normalmente con pocas o ninguna modificación

## 2.3. Procedimiento SUB de eventos

Cuando un objeto en VB.NET reconoce que se ha producido un evento, llama automáticamente al procedimiento de evento utilizando el nombre correspondiente al evento. Como el nombre establece una asociación entre el objeto y el código, se dice que los procedimientos de evento están adjuntos a formularios y controles.

¿Recuerdas cuando pusimos el código en el evento "clic" del botón? Esa parte del programa era un procedimiento Sub de evento. Te recuerdo:

```
Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click  
  
End Sub
```

Todos los objetos o controles de Windows atienden a una serie de eventos: clic, doble clic, pasar el ratón por encima, arrastrarlos... y para cada uno de esos eventos el IDE tiene preparado un procedimiento vacío por si queremos poner código a ese evento. En el ejemplo aparece:

```
Private Sub Button1_click (...)
```

Nos olvidamos de momento de esa palabra "Private" que veremos más adelante. Lo interesante viene después. Tenemos un procedimiento Sub que se va a activar en el evento clic de un botón llamado "Button1"

Un procedimiento de evento de un control combina el nombre real del control (especificado en la propiedad Name), un carácter de subrayado (\_) y el nombre del evento. Por ejemplo, si queremos que un botón de comando llamado cmdPlay llame a un procedimiento de evento cuando se haga clic en él, utilizaremos el procedimiento cmdPlay\_Click.

Un procedimiento de evento de un formulario combina la palabra "Form", un carácter de subrayado y el nombre del evento. Si deseamos que un formulario llame a un procedimiento de

evento cuando se hace clic en él, utilizaremos el procedimiento Form\_Click. (Como los controles, los formularios tienen nombres únicos.)

Todos los procedimientos de evento utilizan la misma sintaxis general.

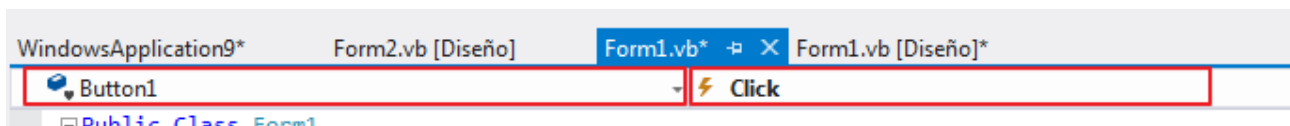
#### Sintaxis de un evento de control

```
Private Sub nombrecontrol_NombreEvento (argumentos)
    instrucciones
End Sub
```

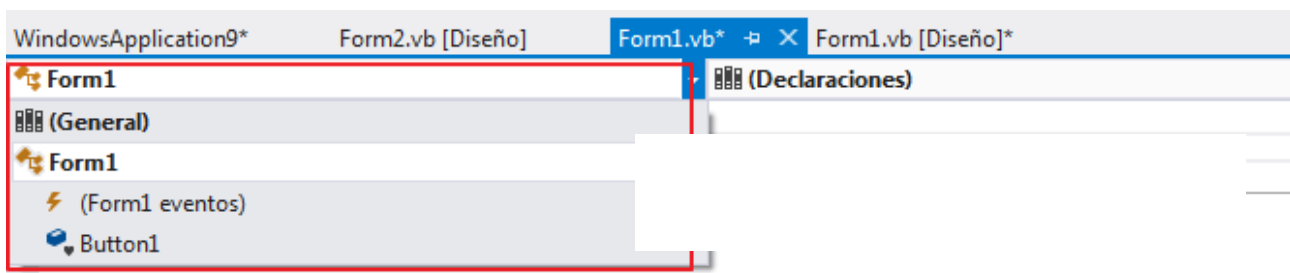
#### Sintaxis de un evento de formulario

```
Private Sub nombreForm_NombreEvento (argumentos)
    instrucciones
End Sub
```

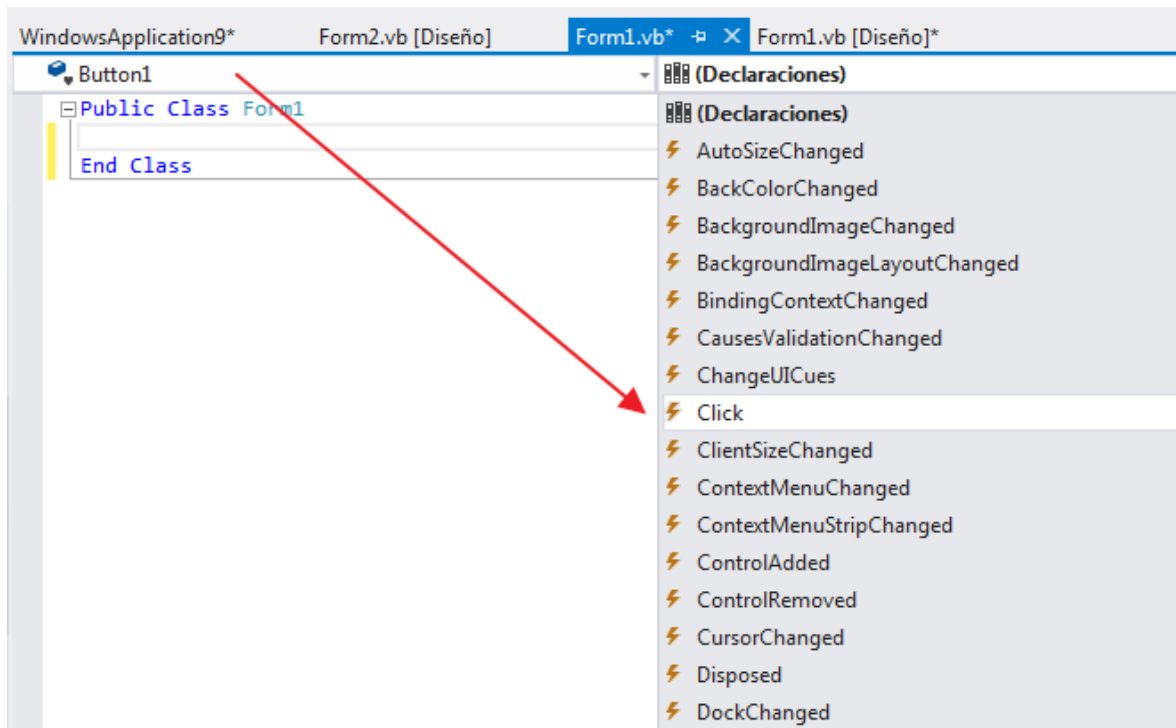
Para localizarlos en el IDE tenemos que fijarnos en dos partes. Por ejemplo, tenemos un formulario con un botón y quiero localizar el evento clic del botón. Primero nos situaremos en el editor de código. Podemos ver tres cuadros desplegables en la parte superior del editor. Nos interesan el del centro y el de la derecha:



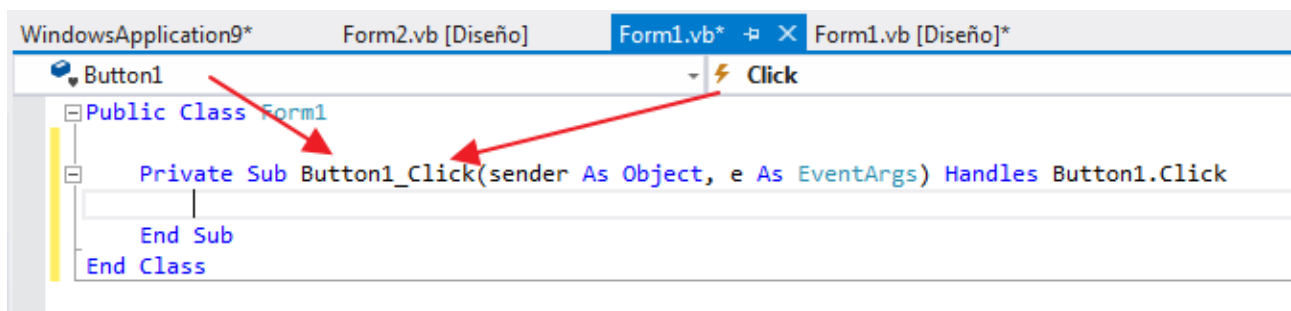
En la parte central tenemos todos los controles y elementos del formulario y a la derecha los eventos. Así que en la parte central buscamos el botón por su nombre, en nuestro caso con el nombre genérico que le puso el IDE al añadirlo al formulario:



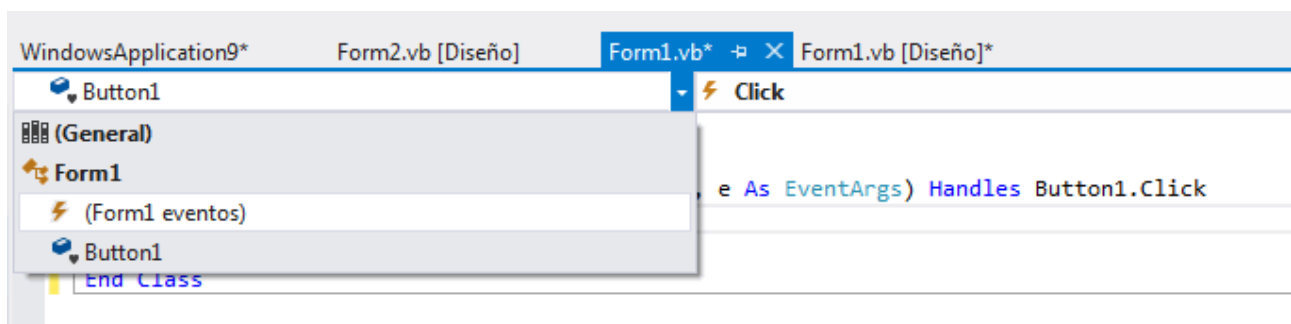
Ahora en la parte derecha tenemos todos los eventos que atenderá este botón, de ahí seleccionaremos el evento clic:



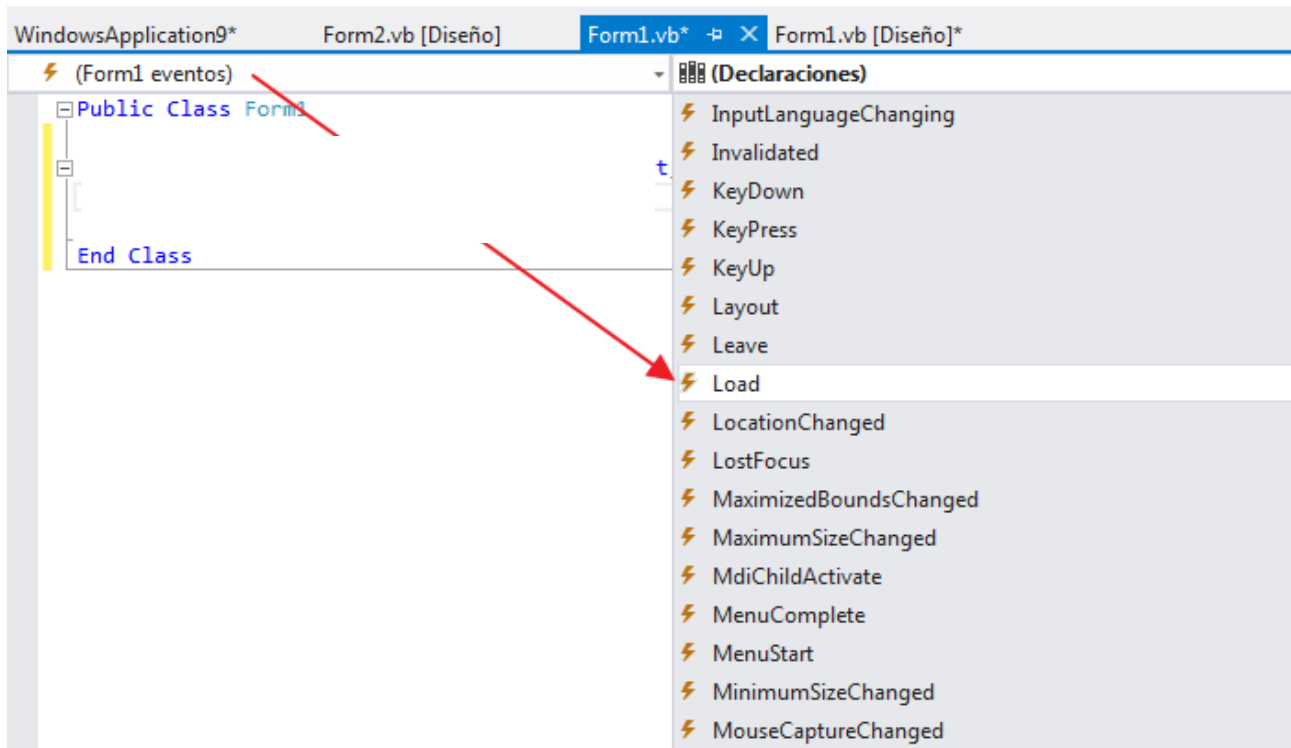
Con esta selección el IDE nos muestra el procedimiento que se ejecutará cuando se haga clic en el botón:



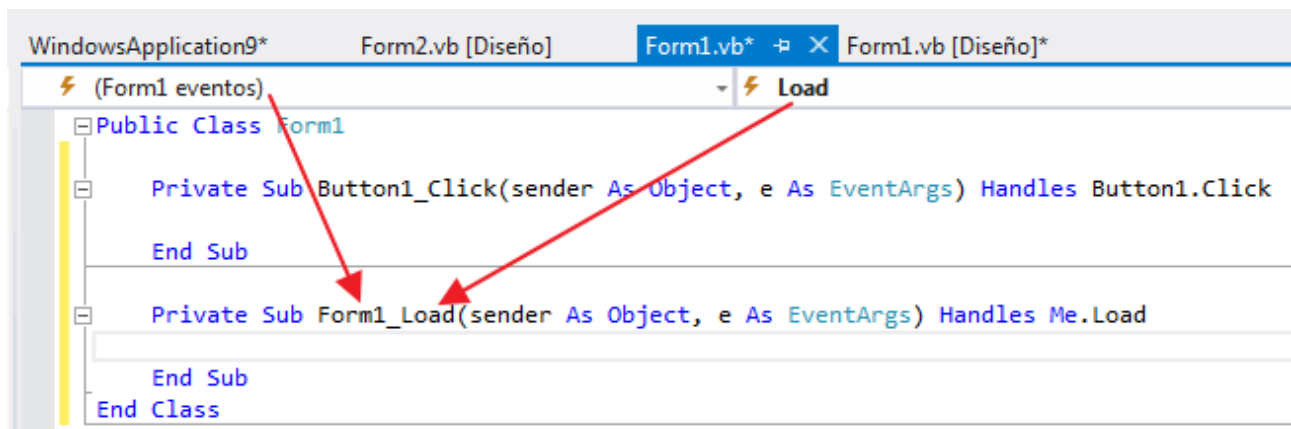
Veamos otro ejemplo. En este caso, quiero acceder al evento Load del formulario. Este evento se va a ejecutar cuando se ponga en marcha nuestro formulario. Si se trata del formulario principal es el primer código que se va a ejecutar en nuestra aplicación. Esto hace que este evento sea muy utilizado para realizar las operaciones previas a la presentación del formulario. Por ejemplo, si tenemos que cargar unos datos en el formulario para mostrarlos al usuario utilizaremos este evento Load. Para localizar este evento seleccionaremos en el centro:



Es decir, utilizaremos como elemento en el centro "Form1 eventos" que son los eventos que se pueden generar en el formulario. Una vez seleccionado, en la parte derecha desplegaremos todos los eventos de los formularios buscando el evento Load:



Al seleccionarlo, crea el procedimiento para insertar código en él:



No es necesario que haya código. Si nos equivocamos de evento lo creará pero podemos borrarlo sin problemas. Simplemente es una ayuda donde nos crea ya el procedimiento que atenderá el objeto seleccionado con el evento de la derecha. En este caso proporciona un procedimiento que se ejecutará en la carga del formulario y antes de mostrárselo al usuario.

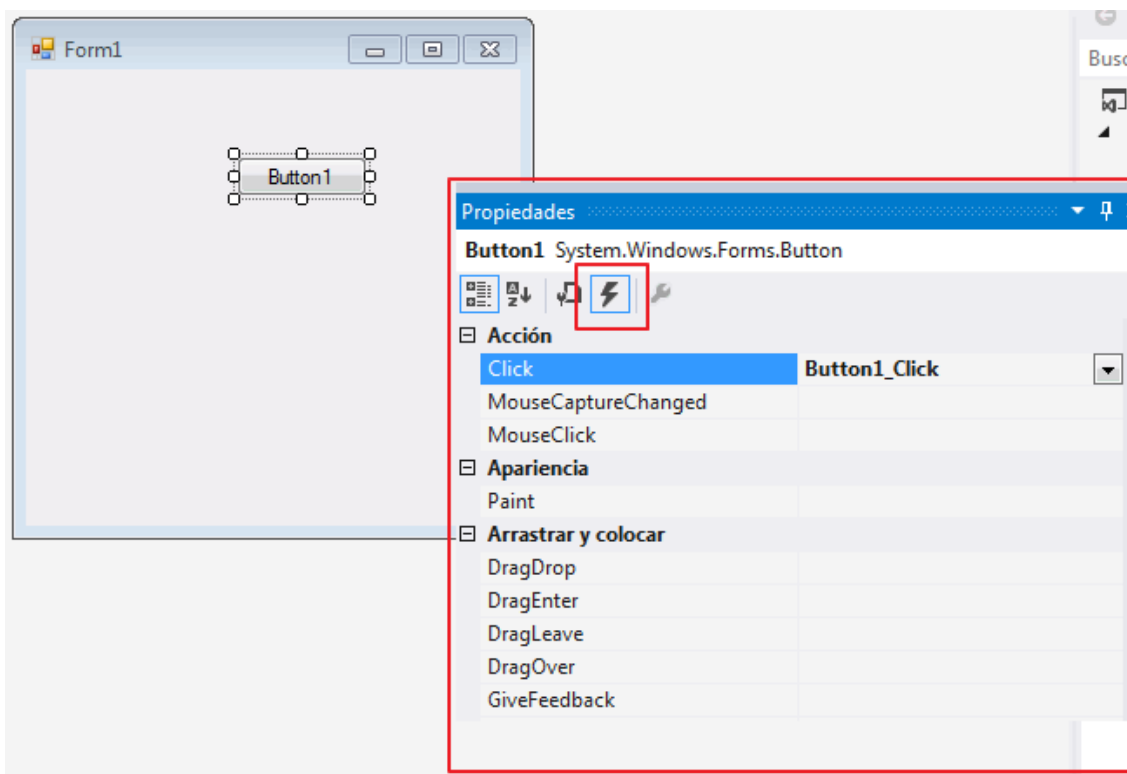
Aunque podemos escribir procedimientos de evento nuevos, es más sencillo usar los procedimientos de código que facilita Visual Basic .Net, que incluyen automáticamente los nombres correctos de procedimiento. También es conveniente establecer la propiedad Name de los controles antes de empezar a escribir los procedimientos de evento para los mismos. Si

cambiamos el nombre de un control tras vincularle un procedimiento, deberemos cambiar también el nombre del procedimiento para que coincida con el nuevo nombre del control. De lo contrario, Visual Basic no será capaz de hacer coincidir el control con el procedimiento. Cuando el nombre de un procedimiento no coincide con el nombre de un control, se convierte en un procedimiento general sin atender eventos.

### ¡IMPORTANTE!

Es muy importante lo que acabamos de comentar, así que lo repetimos para que lo tengamos en cuenta ya que puede provocarnos fallos en el programa cuando lo único que pasa es que no tiene el nombre adecuado. **Es conveniente establecer la propiedad Name de los controles antes de empezar a escribir los procedimientos de evento para los mismos.** Si cambiamos el nombre de un control tras vincularle un procedimiento, deberá cambiar también el nombre del procedimiento para que coincida con el nuevo nombre del control. De lo contrario, Visual Basic no será capaz de hacer coincidir el control con el procedimiento.

Para acceder a los eventos, tenemos una opción más y es desde el propio IDE en el modo de diseño de los formularios:



En la ventana de propiedades hacemos clic en el icono de los eventos. Ahí podemos ver todos los disponibles para este control. Si hacemos doble clic o escribimos el nombre que queramos nos creará el procedimiento del evento en el código.

### 3. Funciones (Function)

Las funciones son similares a los procedimientos pero devuelven siempre un valor. Por ejemplo, una función para calcular una raíz cuadrada: le proporcionamos unos datos de entrada y devuelve un resultado.

Una **función (Function)** consiste en una serie de instrucciones de Visual Basic delimitadas por las instrucciones **Function** y **End Function**. Cada vez que se llama a un procedimiento de este tipo, se ejecutan las instrucciones de éste, desde la primera instrucción ejecutable tras la instrucción **Function** hasta la primera instrucción **End Function**, **Exit Function** o **Return** que se encuentre.

Un procedimiento **Function** es similar a un procedimiento **Sub**, pero además devuelve un valor al programa que realiza la llamada al procedimiento. Como en los procedimientos, puede tomar argumentos, como constantes, variables o expresiones, que le pasa el código de llamada.

La sintaxis para declarar un procedimiento **Function** es la siguiente:

```
[visibilidad] Function nombrefuncion[(listaargumentos)] As tipodatos
    ' Instrucciones de la función...
End Function
```

La visibilidad puede ser Public, Protected, Friend, **Protected Friend** o Private, que veremos más tarde.

Los procedimientos **Function** pueden definirse en módulos, clases y estructuras. Son **Public** de forma predeterminada, lo que significa que se les puede llamar desde cualquier parte de una aplicación.

Los argumentos se declaran del mismo modo que en un procedimiento **Sub**.

#### 3.1. Valores devueltos

El valor que un procedimiento **Function** devuelve al programa que realiza la llamada se denomina *valor devuelto*. La función puede devolver dicho valor de dos maneras:

- La función asigna un valor a su propio nombre de función en una o más instrucciones del procedimiento. No se devuelve el control al programa que realiza la llamada hasta que se ejecuta una instrucción **Exit Function** o **End Function**, como en el siguiente ejemplo:

```
Function nombrefuncion[(Listaargumentos)] As tipodatos  
    ' ...  
    nombrefuncion = expresión  
    ' ...  
End Function
```

- La función utiliza la instrucción **Return** para especificar el valor devuelto, e inmediatamente devuelve el control al programa de llamada, como en el siguiente ejemplo:

```
Function nombrefunción[(Listaargumentos)] As tipodatos  
    ' ...  
    Return expresión  
    ' ...  
End Function
```

La ventaja de asignar el valor devuelto al nombre de la función es que el control permanece en la función hasta que el programa encuentra una instrucción **Exit Function** o **End Función**, lo que permite asignar un valor previo y, si es necesario, se puede ajustar después.

Todos los procedimientos **Function** tienen un tipo de datos, al igual que las variables. La cláusula **As** de la instrucción **Function** especifica el tipo de datos, y determina el tipo del valor devuelto. En las siguientes declaraciones de ejemplo se ilustra esto último:

```
Function ayer As Date  
    ' ...  
End Function
```

```
Function BuscaSqrt(ByValNumero As Single) As Single  
    ' ...  
End Function
```



### 3.2. Sintaxis de llamada

Para llamar a un procedimiento **Function**, hay que incluir el nombre y los argumentos de éste en la parte derecha de una asignación o en una expresión. Esto último se ilustra en los siguientes ejemplos de llamada:

```
valor = nombrefuncion[(listaargumentos)]  
If ((nombrefuncion[(listaargumentos)] / 3) <= expression) Then ...
```

El siguiente procedimiento **Function** calcula la hipotenusa de un triángulo rectángulo a partir de los valores de los catetos:

```
Function Hipotenusa (ByVal cateto1 As Single, ByVal cateto2 As  
Single) As Single  
    Return Math.Sqrt((cateto1 ^ 2) + (cateto2 ^ 2))  
End Function
```

Atento a estas dos formas de llamar a esta función Hipotenusa:

```
Dim PruebaHipotenusa, X, Y, Area As Single  
Pruebahipotenusa = Hipotenusa(20, 10.7)  
If Hipotenusa(X, Y)=30 then  
    ...  
End If
```

Como ves hay dos formas de llamar a las funciones. Una es asignando su valor a una variable como es en el primer caso, así ya tengo ese valor en la variable "Pruebahipotenusa" y puedo seguir con el programa. La otra forma es llamarla directamente como si fuera una expresión. Por ejemplo el caso del "If... then", donde se realizará la llamada para calcular el valor y luego hará la comparación para ver si su valor es 30.

En el siguiente tema veremos muchas de las funciones más utilizadas, no son todas ni mucho menos pero si las más útiles para nosotros.

Nos quedan dos partes de ver en los procedimientos y funciones: los argumentos y el alcance o visibilidad.

## 4. Argumentos en procedimientos y funciones

En la mayoría de los casos, un procedimiento necesita una serie de datos para realizar el proceso. Un procedimiento que ejecuta tareas repetidas o compartidas utiliza datos distintos en cada llamada. Estos datos se componen de variables, constantes y expresiones que se pasan al procedimiento cuando se le llama. Un valor que se pasa a un procedimiento se denomina **argumento**.

### 4.1. Declaración del tipo de datos de un argumento

El tipo de datos de un argumento se declara utilizando la cláusula **As** en su declaración. Por ejemplo, la siguiente función acepta una cadena (Dia) y un entero (Hora):

```
Function tarea(ByVal Dia As String, ByVal Hora As Integer) As String
    ' código...
End Function
```

Si el modificador de comprobación de tipos (Option Strict) está desactivado, la cláusula **As** será opcional, salvo que alguno de los argumentos la utilice, en cuyo caso el resto de los argumentos tendrá que utilizarla también. Si la comprobación de tipos está activada, la cláusula **As** será obligatoria para todos los argumentos del procedimiento.

### 4.2. Pasar un argumento con ByVal o ByRef

En Visual Basic.Net, podemos pasar un argumento a un procedimiento *por valor* o *por referencia*, especificando las palabras clave **ByVal** o **ByRef**, respectivamente. Pasar un argumento por valor implica que el procedimiento no puede modificar el contenido del elemento variable en el código de llamada subyacente al argumento. Pasar un argumento por referencia permite que el procedimiento modifique el contenido de la misma forma que puede hacerlo el propio código de llamada.

Digamos que en un caso se manda una copia del valor con lo cual por mucho que lo utilicemos en nuestro proceso no va a cambiar el valor de la original. En cambio, en el segundo caso es como si se manda un apuntador (o puntero) a la dirección de memoria de esa variable, con lo que si en este procedimiento la cambiamos, también lo hará el valor origen de la llamada.

La distinción que se hace entre pasar los argumentos por valor y por referencia no tiene nada que ver con la clasificación de los tipos de datos en tipos de valor y tipos de referencia. No obstante, ambas categorías interactúan entre sí.

### 4.3. Argumentos variables y no variables

El elemento de programación subyacente a un argumento puede ser un elemento variable, con capacidad para cambiar de valor, o un elemento no variable. La siguiente tabla muestra elementos variables y no variables. Es decir, que algunas cosas no van a variar aunque las pasemos por referencia.

Elementos variables	Elementos no variables
Variables declaradas, incluidas las variables de objetos	Constantes
Campos (de clases)	Literales
Elementos matriciales	Enumeraciones
Elementos estructurales	Expresiones

### 4.4. Argumentos opcionales

Un argumento de un procedimiento puede ser opcional si así se especifica, es decir, no es necesario suministrarlo al llamar al procedimiento. Los *argumentos opcionales* se indican mediante la palabra clave **Optional** en la definición del procedimiento. Se aplican las siguientes reglas:

- Todos los argumentos opcionales de la definición del procedimiento deben especificar un valor predeterminado.
- El valor predeterminado de un argumento opcional debe ser una expresión constante.
- Todos los argumentos que vayan a continuación de un argumento opcional en la definición del procedimiento también deben ser opcionales.

La siguiente sintaxis muestra una declaración de procedimiento con un argumento opcional:

```
Sub nombresub(ByVal arg1 As tipo1, Optional ByVal arg2 As tipo2 = default)
```

### Llamar a procedimientos con argumentos opcionales

Cuando se llama a un procedimiento con un argumento opcional, se puede elegir si se suministra o no el argumento. Si no se suministra, el procedimiento utiliza el valor predeterminado declarado para dicho argumento.

Si se omiten uno o más argumentos opcionales de la lista de argumentos, hay que utilizar comas sucesivas para delimitar sus posiciones. En el ejemplo de llamada siguiente se suministran los argumentos primero y cuarto, pero no el segundo ni el tercero:

```
' dejamos vacíos los dos argumentos opciones: arg2 y arg3.  
NombreProcedimiento(arg1, , , arg4)
```

### Determinar si un argumento opcional está presente

En tiempo de ejecución un procedimiento no puede detectar si un argumento determinado se ha omitido o si el código de llamada ha suministrado de forma explícita el valor predeterminado de dicho argumento. Si fuese necesario hacer esta distinción, se podría establecer como valor predeterminado un valor improbable. El procedimiento que se muestra a continuación incluye el argumento opcional "Oficina" y comprueba si su valor predeterminado, QJZ, ha sido omitido:

```
Sub Aviso(ByVal Empresa As String, Optional ByVal Oficina As _  
String = "Ofi1")  
    If Oficina = "Ofi1" Then  
        MsgBox("Atención: no se ha proporcionado Oficina")  
        Oficina = "Principal"  
    End If  
    ' Aviso a la oficina  
End Sub
```

### Pasar argumentos por posición o por nombre

Como ya hemos visto antes, los procedimientos pueden recibir una lista de argumentos para realizar un proceso. Por ejemplo, un procedimiento para calcular una suma.

```
Sub sumar(dato1 as long, dato2 as long)  
    dim resultado as long  
    resultado = dato1 + dato2  
    MsgBox(resultado)  
End Sub
```

Llamada al procedimiento:

```
sumar(5,6)
```

Vemos un procedimiento que necesita dos valores o argumentos que son dato1 y dato2. Cuando hacemos la llamada ponemos en la misma posición los valores que queremos enviar al procedimiento, en este caso el 5 se asignará a dato1 y el 6 al dato2. Esto es fácil de ver en una depuración que realizaremos luego. Finalmente "MsgBox" escribirá un cuadro en pantalla con el resultado de la variable "resultado" declarada como de tipo entero largo o "long". Además hará la suma con las variables dato1 y dato2 que tienen como valores 5 y 6.

Sabemos entonces que los argumentos los podemos pasar por posición, es decir, en el orden en que aparecen en la definición del procedimiento. La otra forma es por nombre sin tener en cuenta la posición, así que le diremos la variable en la llamada. Para pasar un argumento por nombre, hay que especificar el nombre declarado del argumento, seguido de un signo de dos puntos y un signo igual (:=) y seguido del valor del argumento. Los argumentos que se pasan por nombre pueden suministrarse en cualquier orden.

En el ejemplo anterior la llamada sería:

```
sumar(dato1:=5, dato2:=6)
```

Veamos otro ejemplo: un procedimiento **Sub** con tres argumentos:

```
Sub informacionEstudiante(ByVal Nombre As String, Optional ByVal Edad As Integer = 0, Optional ByVal FechaNacimiento As Date = #1/1/2000#)
    ...
End Sub
```

Cuando se llama a este procedimiento, los argumentos pueden suministrarse por posición, por nombre o mediante una combinación de ambos.

### Pasar argumentos por posición

Se puede llamar al procedimiento InformacionEstudiante pasando sus argumentos por posición y delimitados por comas, como se muestra en el ejemplo siguiente (que es la forma más estándar):

```
InformacionEstudiante("Antonio", 19, #9/21/1981#)
```

Si se omite un argumento opcional de una lista de argumentos por posición, se deberá mantener su posición mediante una coma. El ejemplo siguiente llama a InformacionEstudiante sin el argumento Edad:

```
InformacionEstudiante("Ana", , #9/21/1981#)
```

### **Pasar argumentos por nombre**

Otra opción es llamar al procedimiento `InformacionEstudiante` pasando sus argumentos por nombre, también delimitados por comas. Por ejemplo:

```
Informacionestudiante(Edad:=23, Fechanacimiento:=#9/21/1981#, Nombre:="Pepe")
```

### **Pasar argumentos por posición y por nombre**

Se pueden suministrar los argumentos por posición y por nombre a la vez en la misma llamada a un procedimiento:

```
prueba(Cantidad:=Importe, Fecha:=diahoy, Ciudad:=Localidad)
```

Cuando el procedimiento tiene esta definición:

```
Sub Prueba (ByVal Ciudad as String, ByVal Cantidad As Integer,  
    ByVal Fecha as Date)  
    ...  
End Sub
```

Si se suministran argumentos por nombre y por posición, los argumentos por posición deben preceder al resto. Si se ha suministrado un argumento por nombre, el resto de los argumentos deberán especificarse también por nombre.

### **Suministrar argumentos opcionales por nombre**

Pasar argumentos por nombre tiene especial utilidad si se llama a un procedimiento que tiene varios argumentos opcionales. Si se suministran argumentos por nombre, no es necesario utilizar comas consecutivas para denotar los argumentos por posición ausentes. Pasar argumentos por nombre facilita también la tarea de realizar un seguimiento de los argumentos que se pasan y de los que se omiten.

### **Restricciones al suministro de argumentos por nombre**

No se pueden pasar argumentos por nombre a fin de evitar tener que especificar argumentos requeridos. Sólo pueden omitirse los argumentos opcionales.

No se puede pasar una matriz de parámetros por nombre. Esto es porque, cuando se llama al procedimiento, se suministra a la matriz de parámetros un número indeterminado de argumentos separados por comas, y el compilador no puede asociar más de un argumento a cada nombre individual.

## 4.5. Matrices de parámetros

Normalmente no es posible llamar a un procedimiento con un número de argumentos superior al especificado en la declaración del procedimiento. Si se necesita un número indeterminado de argumentos, puede declararse una *matriz de parámetros*, que permite que un procedimiento acepte una matriz de valores para un argumento. No es necesario conocer el número de elementos de la matriz de parámetros en el momento de definir el procedimiento. El tamaño de la matriz se determina individualmente en cada llamada al procedimiento.

Para esto utilizaremos la palabra clave **ParamArray** que define una matriz de parámetros, teniendo en cuenta que hay que aplicar estas reglas:

- Un procedimiento sólo puede tener una matriz de parámetros. Debe ser el último argumento de la definición del procedimiento.
- La matriz de parámetros debe pasarse por valor. Es un hábito de programación recomendado incluir de manera explícita la palabra clave **ByVal** en la definición del procedimiento.
- El código del procedimiento debe considerar a la matriz de parámetros una matriz unidimensional; el tipo de datos de los elementos de la matriz ha de ser el mismo que el tipo de datos de **ParamArray**.
- La matriz de parámetros es opcional de forma automática. Su valor predeterminado es una matriz unidimensional vacía del tipo de elemento de la matriz de parámetros.
- Todos los argumentos que preceden a la matriz de parámetros deben ser obligatorios. La matriz de parámetros debe ser el único argumento opcional.

Cuando uno de los argumentos del procedimiento al que se llame sea una matriz de parámetros, ésta podrá tomar cualquiera de estos valores:

- Ninguno, es decir, puede omitirse el argumento **ParamArray**. En este caso, se pasará una matriz vacía al procedimiento. También puede pasarse la palabra clave **Nothing**, obteniéndose el mismo efecto.
- Una lista con un número de argumentos indeterminado, separados por comas. El tipo de los datos de cada argumento debe poder convertirse implícitamente al tipo de elemento **ParamArray**.
- Una matriz con el mismo tipo de elemento que la matriz de parámetros.

En el siguiente ejemplo se muestra cómo se puede definir un procedimiento con una matriz de parámetros:

```
Sub PuntuacionesAlumnos(ByVal Nombre As String, ByVal ParamArray
Puntuaciones() As String)
Dim i As Integer
MsgBox("Puntuaciones de " & Nombre & ":")
' Utilizamos otra vez Ubound para saber cuántos elementos hay en
' la matriz y poder hacer el bucle:
For i = 0 To UBound(Puntuaciones)
    MsgBox("Puntuación " & i & ": " & Puntuaciones(i))
Next i
End Sub
```

En los siguientes ejemplos se muestran llamadas a PuntuacionesAlumnos:

```
PuntuacionesAlumnos("Ana", "10", "26", "32", "15", "22", "24", "16")
PuntuacionesAlumnos("María", "Alto", "Bajo", "Media", "Maximo")
Dim PuntuacionesJuan() As String = {"35", "Ausente", "21", "30"}
PuntuacionesAlumnos("Juan", PuntuacionesJuan)
```

Las dos primeras están claras, el primer argumento es "Nombre", como hay más argumentos el procedimiento sabe que son una matriz de elementos del mismo tipo así que los va metiendo en una matriz "Puntuaciones"

El último caso es algo más complejo, primero se ha declarado una matriz de tipo String donde ya le indicamos los elementos que tiene. Luego hacemos la llamada con Juan como "nombre" y la matriz anterior como argumento que introducirá en "Puntuaciones".



## 5. Sobrecarga de procedimientos

Hemos visto la utilidad de poder utilizar distinto número de parámetros para las llamadas a funciones y procedimientos con **Optional**. Pero, hay que tener en cuenta que la instrucción **Optional** existe "por compatibilidad" con las versiones anteriores de Visual Basic.

Visual Basic .NET permite la sobrecarga de procedimientos, y gracias a esto podemos hacer lo mismo de una forma más intuitiva.

Veamos lo que nos dice la documentación de Visual Studio .NET sobre la sobrecarga de propiedades y métodos (o lo que es lo mismo, sobre la sobrecarga de procedimientos):

*La sobrecarga consiste en crear más de un procedimiento, constructor de instancias o propiedad en una clase con el mismo nombre y distintos tipos de argumento.*

*La sobrecarga es especialmente útil cuando un modelo de objeto exige el uso de nombres idénticos para procedimientos que operan en diferentes tipos de datos.*

La sobrecarga es uno de los pilares de la programación orientada a objetos y existe desde el principio de los tiempos de la misma. Si necesitamos un procedimiento que utilice distinto número de parámetros o parámetros de distintos tipos, podemos utilizar la sobrecarga de procedimientos.

Sabiendo esto, podríamos hacer lo mismo que con el procedimiento **Prueba** mostrado anteriormente con estas declaraciones:

```
Prueba()  
Prueba(ByVal uno As Integer)  
Prueba(ByVal uno As Integer, ByVal dos As Integer)
```

Entonces, si escribo tres procedimientos con diferente número de parámetros ¿debo repetir el código tres veces, uno para cada uno de los procedimientos?

La respuesta es: sí o no... depende. Es decir, si quieres, puedes escribir tres veces o tres códigos distintos, uno para cada procedimiento. Pero si no quieres no.

Pero, ¿cómo puedo hacer que los tres procedimientos sean operativos sin tener que repetir prácticamente lo mismo en los tres procedimientos?

De acuerdo, nos imaginamos que queremos que funcione como en el primer ejemplo, el que usaba los parámetros opcionales. Es decir, si no se indicaba uno de los parámetros tomará un valor por defecto. Podríamos hacerlo de la siguiente forma:

```
Sub Prueba2()  
    ' Si no se indica ninguno de los dos parámetros,  
    ' usar los valores "por defecto"  
    Prueba2(3, 5)  
End Sub  
  
Sub Prueba2(ByVal uno As Integer)  
    ' Si no se indica el segundo parámetro,  
    ' usar el valor "por defecto"  
    Prueba2(uno, 5)  
End Sub  
  
Sub Prueba2(ByVal uno As Integer, ByVal dos As Integer)  
    MsgBox("uno = " & uno & ", dos = " & dos)  
    MsgBox("...")  
End Sub
```

Es decir, si no se indica ningún parámetro, se usará la primera declaración.

Si se usa la segunda declaración, se usará como primer valor el que se ha indicado y como segundo el predeterminado. Y por último, si indicamos los dos parámetros se llamará a la tercera declaración.

**Nota:** Cuando se usan procedimientos sobrecargados, es el propio compilador de Visual Basic .NET el que decide cual es el procedimiento que mejor se adecúa a los parámetros que se han indicado al llamar a ese procedimiento.

Otra de las ventajas de la sobrecarga de procedimientos, es que además de poder indicar un número diferente de parámetros, podemos indicar parámetros de distintos tipos. Esto es útil si queremos tener procedimientos que reciban parámetros de tipo **Integer** o que reciba parámetros de tipo **Double**.

Incluso podemos hacer que una función devuelva valores de tipos diferentes, aunque en este caso el número o tipo de los parámetros debe ser diferente, ya que **no se pueden sobrecargar procedimientos si sólo se diferencian en el tipo de datos devuelto**.

Tampoco se pueden sobrecargar Propiedades con métodos (Sub o Function), es decir, sólo podemos sobrecargar propiedades con otras propiedades o procedimientos (Sub o Function) con otros procedimientos (Sub o Function).

Como hemos señalado antes, la sobrecarga de métodos es una de las particularidades de la programación orientada a objetos, en inglés, "OverLoading". Es un concepto muy sencillo y potente y, de ahí, su importancia. Esta técnica permite realizar varias versiones distintas de un mismo método. Por ejemplo, puedes hacer una consulta a una base de datos que devuelve una matriz de objetos de productos que representan registros o filas de la base de datos. Puesto que queremos crear tres funciones distintas para obtener distintas consultas (para obtener todos los productos, categorías de los productos y Productos en stock) podíamos crear solo una que trabaje con las tres operaciones. Cada método tendrá el mismo nombre pero distinta declaración, dependiendo del número de parámetros necesarios. Además pondremos la palabra clave "overload" para que se sepa que esos métodos están "conectados" y se utilizarán según el número o tipo de parámetros.

Veamos otro ejemplo con dos versiones sobrecargadas del método ObtenerPrecio\_del\_producto()

```
Private Overloads Function ObtenerPrecio_del_producto (id as Integer) as  
Decimal  
    'Código...  
End function
```

```
Private Overloads Function ObtenerPrecio_del_producto (nombre as String) as  
Decimal  
    'Código...  
End function
```

**Nota:** Fíjate que se están declarando las funciones como de ámbito local: Private Overload Function: función privada sobrecargada.

Ahora dependiendo de si hacemos la llamada a esta función con un valor de tipo string (nombre) o integer (id) se ejecutará una función u otra. Así que no hay que crear distintas funciones con distintos nombres, solo con llamarla con el tipo de datos que queremos trabajar ejecutará una función u otra.

```
Dim precio as Decimal
```

```
'Obtiene el precio de un producto por su número de identificación ID  
Precio=ObtenerPrecio_del_Producto (1001)
```

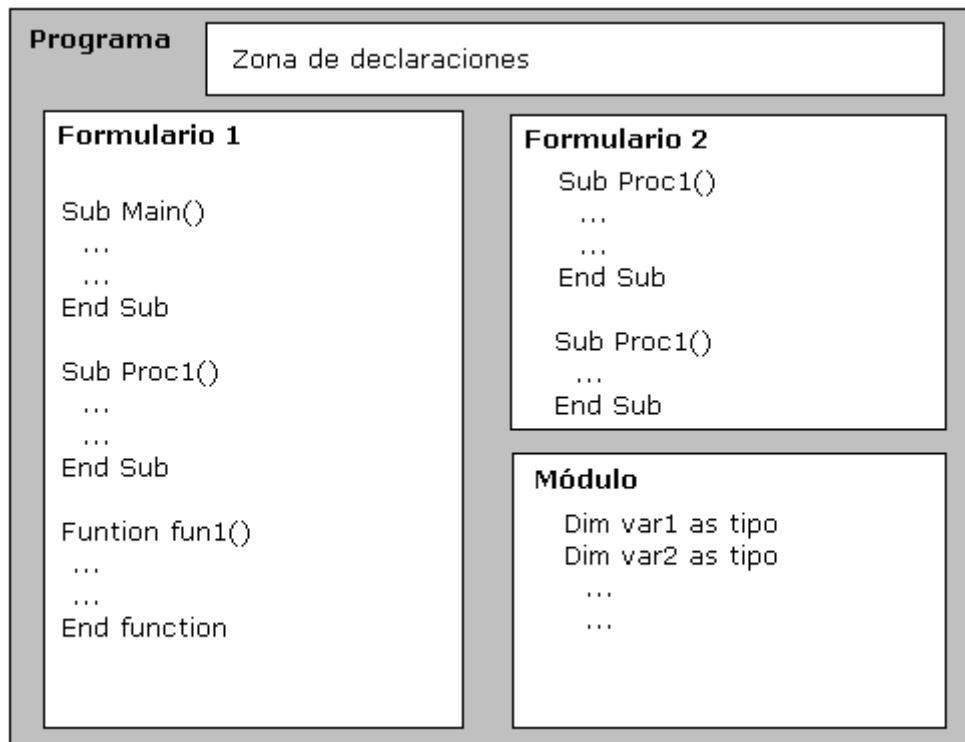
```
'Obtiene el precio de un producto por su nombre  
Precio=ObtenerPrecio_del_Producto ("Reproductor DVD")
```

Obviamente no podremos sobrecargar funciones que tengan el mismo número y tipo de parámetros porque el CLR (common language runtime) no sabría cual tendría que ejecutar.

Los métodos sobrecargados se utilizan mucho, prácticamente en todas las clases de .NET existen métodos así. Los veremos cuando realicemos nuestros programas ya que la ayuda del entorno de desarrollo nos indicará cuando utilicemos una función que ésta está sobrecargada y así la podremos llamar con los parámetros que necesitemos en ese momento.

## 6. Organización del proyecto

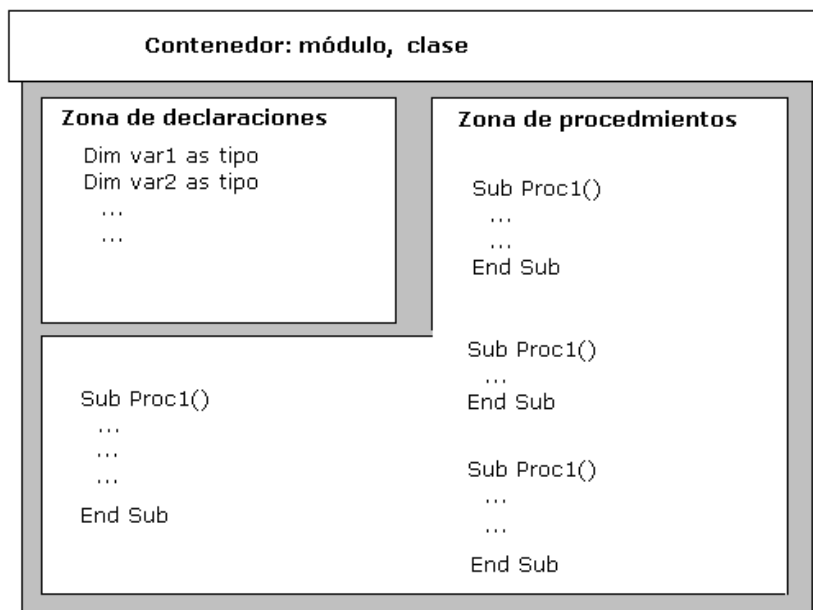
Ya hemos visto las partes de que se compone un proyecto: módulos, clases, formularios, procedimientos, funciones,... Vamos a organizar todos estos elementos para encajarlos todos dentro de un proyecto:



En esta parte seguiremos hablando de los módulos, recuerda que los formularios (los veremos más tarde) son otros ficheros del proyecto pero que son los que contienen la interfaz de usuario. Nuestros proyectos se van a componer habitualmente de varios formularios y algún que otro módulo. El código lo colocaremos siempre "detrás" de los eventos de la interfaz, como ya hemos visto en los ejemplos y otros temas. Pero una parte importante del proyecto son los módulos, así que profundizaremos un poco más en ellos. Un módulo es una parte del programa que contiene código sin estar asociado directamente a los formularios.

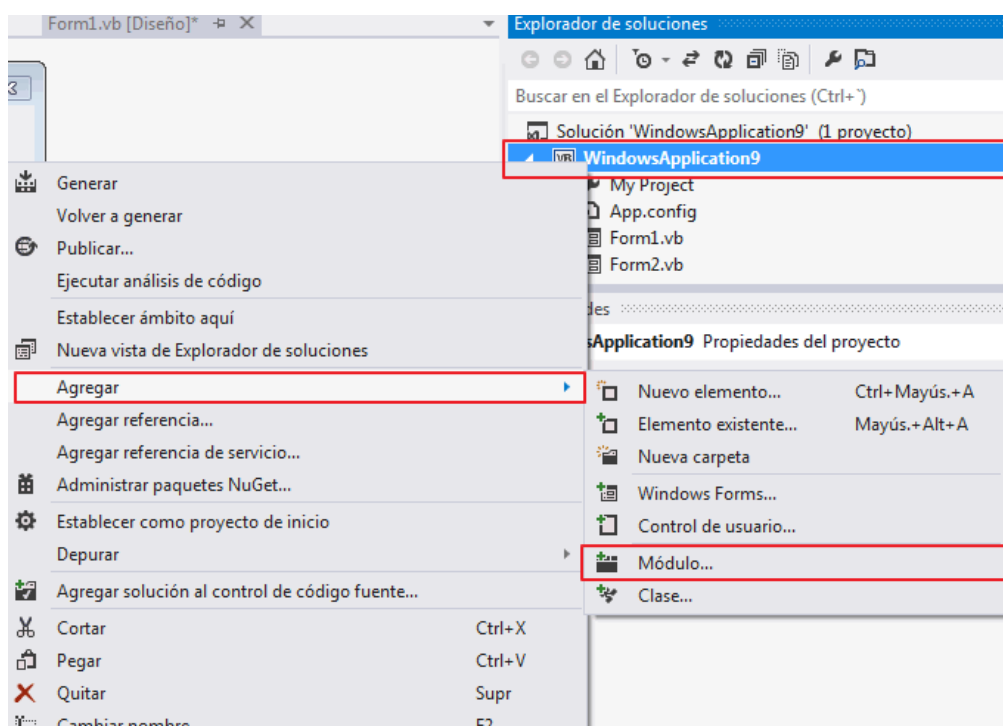
Tienen visibilidad global, es decir, puedo llamar al código que aquí escriba desde cualquier formulario. Por lo tanto, es el lugar adecuado para escribir y declarar las funciones y procedimientos que necesite. Las Clases son un tipo de módulo que me servirán para crear un "objeto" para mi programa.

Sigamos con los módulos. Observa este gráfico:

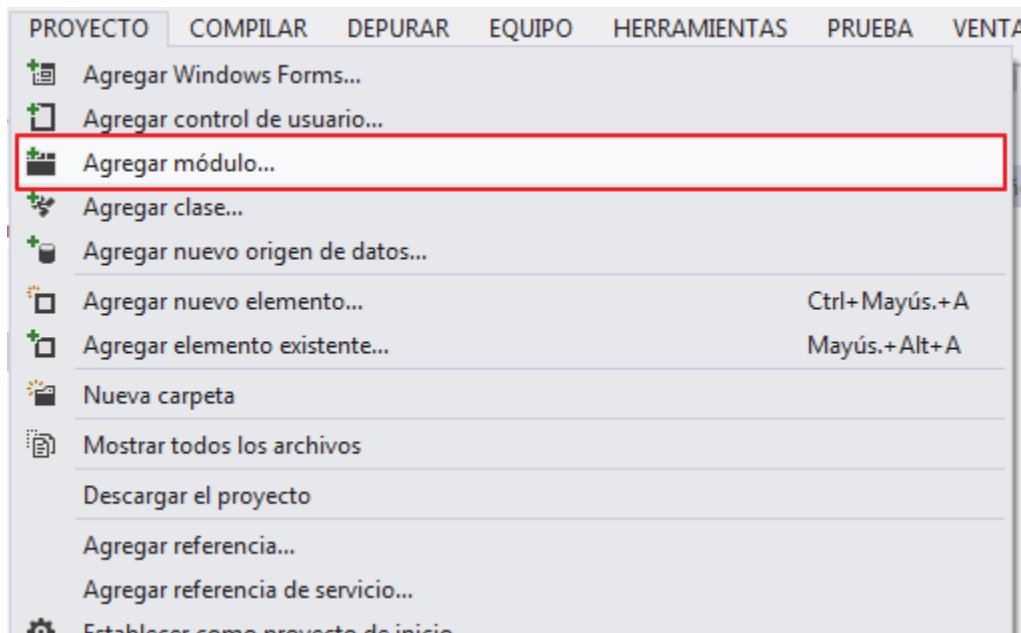


Ahora veamos la parte de VB.NET con los módulos. Cuando en VB.NET añadimos un nuevo módulo para trabajar con él se crea un nuevo fichero con extensión .VB que contiene el módulo o clase. Podemos crear cuantos módulos necesitemos, de hecho en proyectos grandes es una forma de separar el código de diferentes fases del programa. Por ejemplo, si nuestro programa realiza operaciones de facturación y contabilidad podemos separar cada colección de procedimientos y funciones que nos van a realizar las operaciones en dos módulos distintos. Esto favorece la organización del proyecto y su reutilización para otros proyectos.

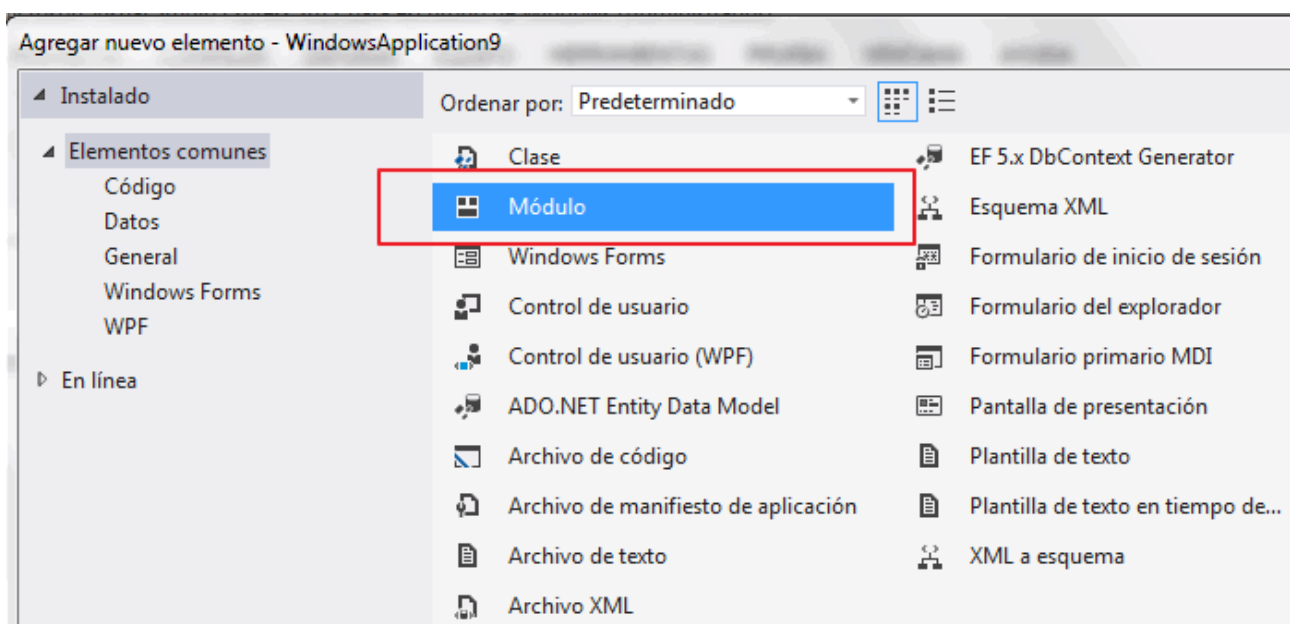
Para añadir un módulo a nuestro proyecto seleccionaremos la opción "Agregar módulo" del menú contextual del administrador de proyectos:



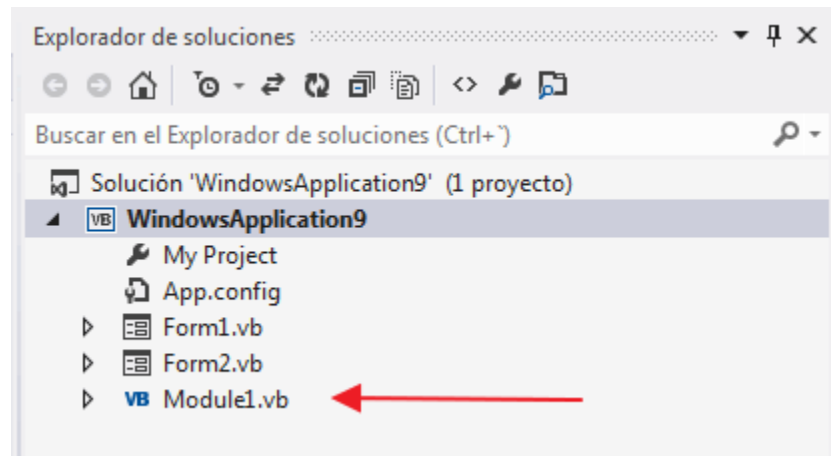
También podemos acceder a esta opción desde el menú:



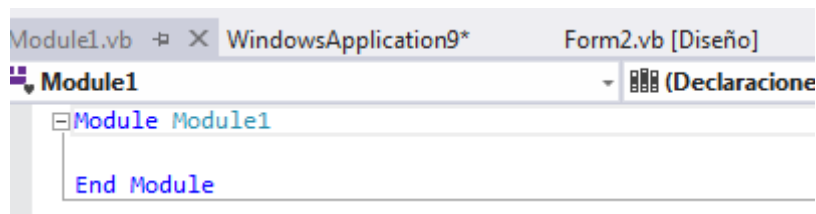
Si en lugar de "Agregar módulo" seleccionamos "Agregar nuevo elemento", opción que tenemos en las dos opciones que hemos visto, nos muestra una ventana donde podemos añadir a nuestro proyecto diferentes elementos, siendo uno de ellos el "módulo":



Como siempre el IDE nos pondrá un nombre genérico y podremos empezar a trabajar con él. Si vemos el explorador de soluciones veremos que aparece ahora en la lista de elementos del proyecto:

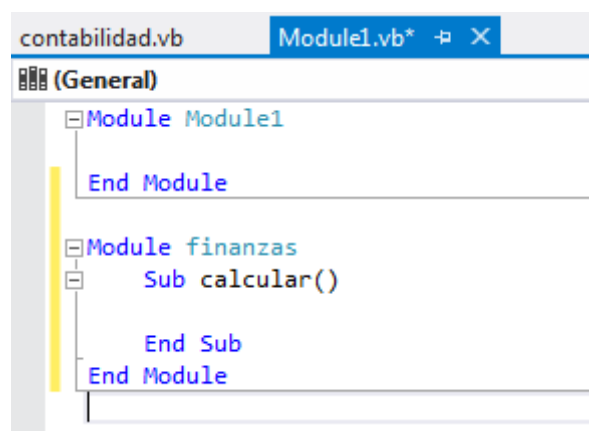


Si hacemos doble clic veremos su código listo para comenzar a trabajar con él:



## 6.1. Crear un módulo en un fichero

Dentro de un "contenedor", es decir, dentro de un fichero podemos crear más módulos, simplemente utilizando las instrucciones que delimitan un módulo (que son muy parecidas a los Sub y Function...):

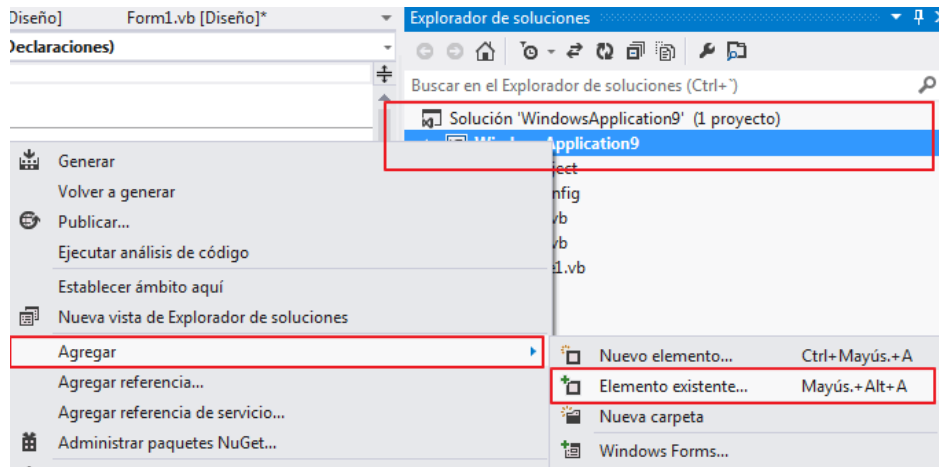


**Nota:** No se pueden crear módulos anidados, es decir un Module... dentro de otro. Debe ser como el ejemplo anterior separados por su "End Module" correspondiente.

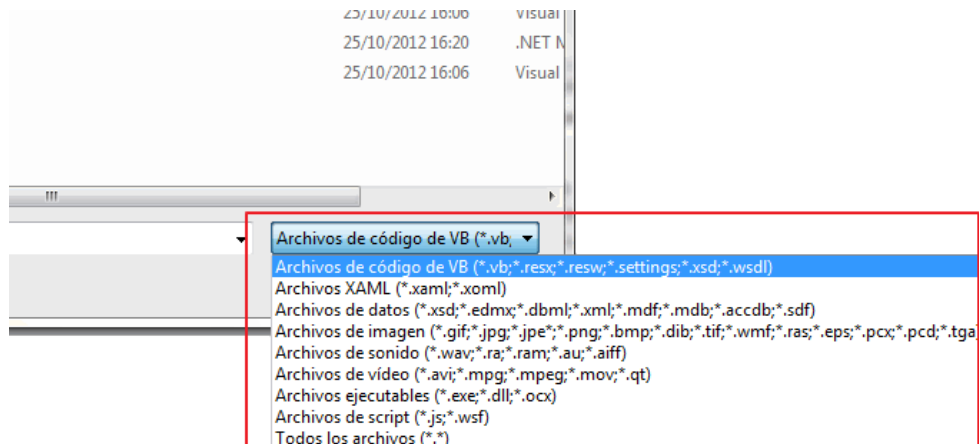
## 6.2. Añadir un módulo existente

Imaginad que habéis creado un módulo con una colección de procedimientos y funciones, o simplemente con una clase y queréis utilizar este código en otro proyecto.

Para añadir un fichero a nuestro proyecto utilizaremos la opción "Agregar" → "Elemento existente":



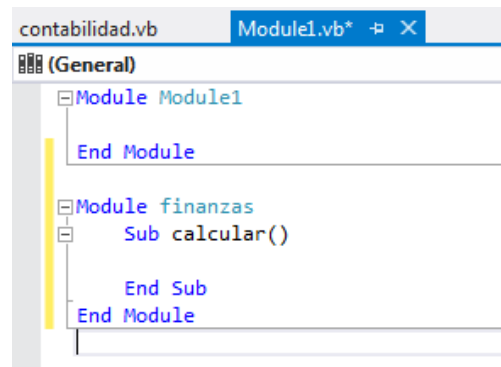
Fíjate en las extensiones que permite cargar:



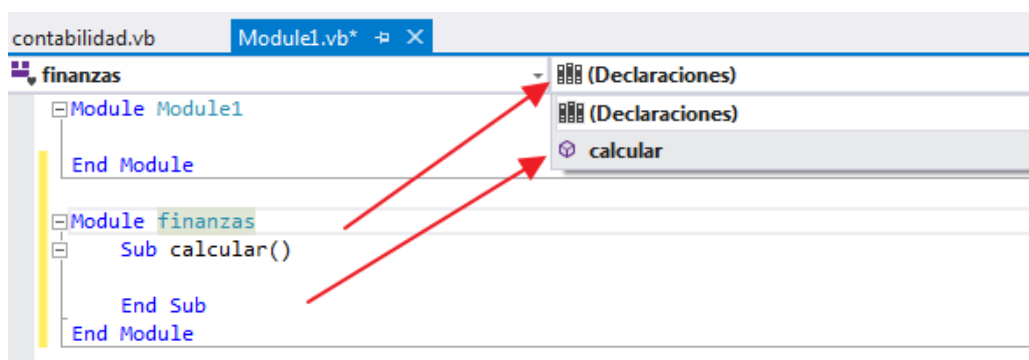
El módulo debe ser un archivo de código de VB.

Como hemos comentado antes, dentro de un módulo podemos crear otros. Por ejemplo, veamos esta pantalla:



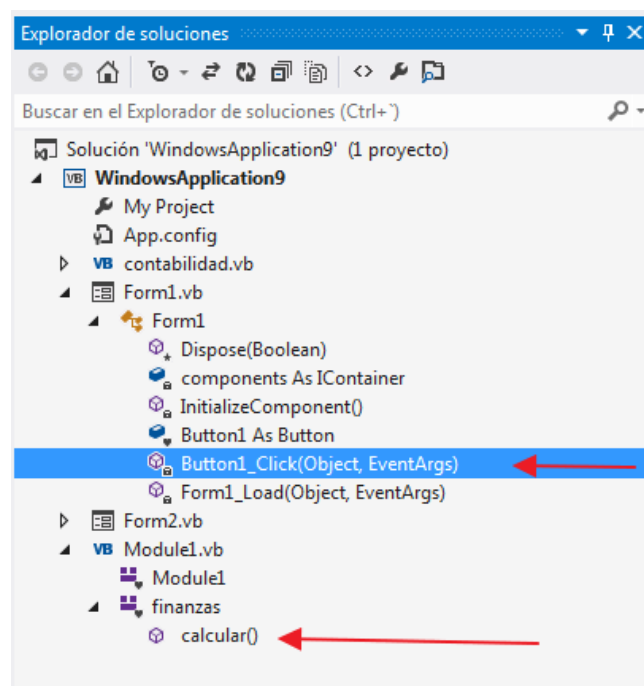


Hay un módulo nuevo que se llama "finanzas" con sus propias funciones, por ejemplo la de "Calcular".



Para movernos más rápidamente dentro del módulo podemos seleccionar en la parte derecha la zona a la que nos queremos mover. Si se compone de multitud de procedimientos, los podemos ver aquí.

Esta misma estructura la podemos ver en el explorador de soluciones. Veamos esto mismo pero ampliando también al formulario que tiene algún evento de botón:



Dentro del formulario podemos ver la definición de "Form1" con los eventos de creación (InitializeComponents) y finalización (Dispose). Además, podemos ver los eventos clic del botón y el "load" del formulario. En cuando el módulo "module1.vb", podemos ver que tiene dos módulos dentro y uno de ellos, llamado finanzas, tiene un método llamado "calcular".

### 6.3. Módulos y Formularios

Para dejar un poco más claro la parte del proyecto que corresponde al formulario y no a los módulos vamos a comentar algunas características sobre los formularios.

- Son también ficheros de nuestro proyecto que aparecerán en el explorador de soluciones como ficheros con extensión .VB.
- La diferencia es que el código en lugar de tener un "Module" tienen un "Public Class Form1", es decir, la definición de un formulario.
- Tiene una ventana para colocar controles: el formulario
- En el código en la parte central el desplegable nos mostrará cada uno de los controles o clases incluidas en el formulario y en el de la derecha los eventos disponibles para cada clase.
- Podemos escribir todas las funciones y procedimientos que necesitemos dentro del formulario pero su ámbito será local, es decir, sólo se pueden utilizar dentro de ese formulario, para ser visibles estos procedimientos por todos los formularios los declararé en un módulo.

## 7. Ámbito de variables y procedimientos

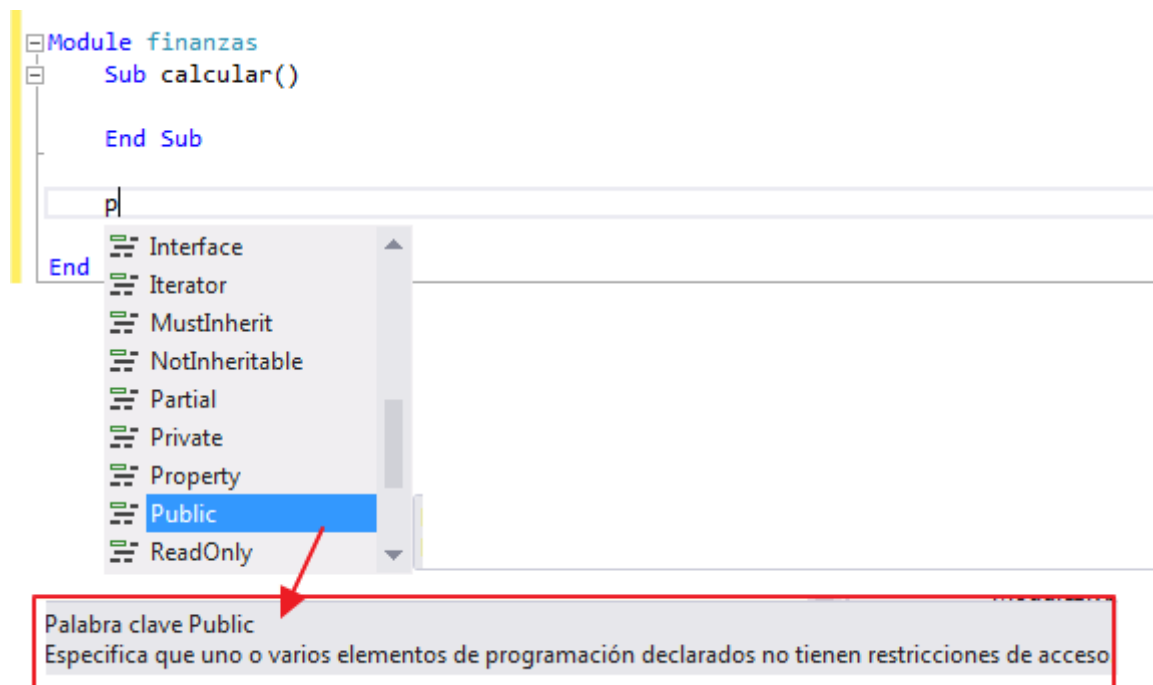
Anteriormente habíamos visto ya una introducción del ámbito o visibilidad de las variables:

- Si se declaran dentro de un procedimiento sólo están disponibles para ese procedimiento (o función)
- Si se declaran fuera de él (normalmente en la parte superior del formulario) están disponibles para todos los procedimientos y funciones de ese formulario.

Ahora vamos a profundizar un poco más y ver todas las posibilidades de este ámbito.

## 7.1. Ámbito de procedimientos

Hemos visto en declaraciones que los procedimientos pueden ser de uno o varios de estos tipos, veamos qué significa cada uno de ellos. En la edición del código nos mostrará estos valores al comenzar a escribir:



Veamos los valores de ámbitos más importantes:

### Friend

En numerosas ocasiones queremos que elementos de programación como clases y estructuras se utilicen en todo el ensamblado y no sólo en el componente en que se declaran. Sin embargo, posiblemente no deseemos que código ajeno al ensamblado tenga acceso a estos elementos, por ejemplo, si se trata de una aplicación propia. Si queremos limitar el acceso a un elemento de este modo, podemos declararlo con Friend.

El acceso de tipo Friend suele ser el nivel preferido de los elementos de programación de una aplicación. Ten en cuenta que el nivel de acceso de una interfaz, módulo, clase o estructura es Friend de manera predeterminada si no se declara lo contrario.

### Protected

A veces un elemento de programación declarado en una clase contiene datos confidenciales o código restringido, por lo que conviene limitar el acceso a dicho elemento. Sin embargo, si la clase se puede heredar y se espera una jerarquía de clases derivadas, quizás sea necesario que estas

clases derivadas tengan acceso a los datos y al código. En este caso, desearemos que se pueda tener acceso al elemento desde la clase base y desde todas las clases derivadas. Para limitar el acceso a un elemento de este modo, lo declararemos con Protected.

## Private

Si un elemento de programación representa la funcionalidad de propiedad o contiene datos confidenciales, por lo general desearemos limitar el acceso a este elemento con toda la rigurosidad que sea posible. Para alcanzar la limitación máxima, permitiremos únicamente que el módulo, la clase o la estructura que define el elemento tenga acceso a este elemento. Para limitar el acceso a un elemento de este modo lo declararemos con Private.

Sólo puede utilizar Private en el nivel de módulo. Esto significa que el contexto de la declaración de un elemento Private debe ser un módulo, una clase o una estructura, y no un archivo de código fuente, un espacio de nombres, una interfaz o un procedimiento.

## Public

La palabra clave Public en la instrucción de declaración especifica que se puede tener acceso a los elementos desde el código en cualquier parte del mismo proyecto, desde otros proyectos que hagan referencia al proyecto y desde un ensamblado generado a partir del proyecto. Podemos utilizar Public solamente en el nivel de módulo, interfaz o espacio de nombres. Es decir, podemos declarar un elemento público en el nivel de archivo de código fuente o espacio de nombres o dentro de una interfaz, módulo, clase o estructura, pero no dentro de un procedimiento.

Por ejemplo, un formulario es público para así poder invocarlo desde otras partes del programa. Lógico ya que los formularios son la interfaz con el usuario:

```
Public Class Form1
    Private Sub Button1_Click(sender As Object, e As EventArgs)
    End Sub
    Private Sub Form1_Load(sender As Object, e As EventArgs)
    End Sub
End Class
```

Sin embargo, los eventos del formulario, como el clic de un botón o la carga "Load", sólo son accesibles desde ese propio formulario. Lógico también porque se trata de operaciones dentro de ese formulario.

## Protected Friend

Las palabras clave "Protected Friend" juntas en la instrucción de declaración especifican que se puede tener acceso a los elementos desde las clases derivadas, desde dentro del mismo ensamblado o ambos. Sólo podremos utilizar "Protected Friend" en el nivel de clase y sólo al declarar un miembro de una clase. Es decir, podemos declarar un elemento de tipo "Protected Friend" en una clase, pero no en el nivel de archivo de código fuente o espacio de nombres, o dentro de una interfaz, módulo, estructura o procedimiento

Las palabras clave que especifican el nivel de acceso se llaman modificadores de acceso. Veamos un resumen final.

Modificador de acceso	Nivel de acceso concedido	Elementos que puede declarar con este nivel de acceso	Contexto de declaración dentro de cual puede utilizar este modificador
Public	Sin restricciones:  Cualquier código que puede ver un elemento público puede tener acceso a él.	Interfaces Módulos Clases Estructuras Miembros de estructura Procedimientos Propiedades Variables miembros Constantes Enumeraciones Eventos Declaraciones externas Delegados	Archivo de código fuente  Espacio de nombres  Interfaz  Módulo  Clase  Estructura
Protected	De derivación:  El código de la clase que declara un elemento protegido, o una clase derivada de él, puede tener acceso al elemento.	Interfaces Clases Estructuras Procedimientos Propiedades Variables miembros Constantes Enumeraciones Eventos Declaraciones externas Delegados	Clase
Friend	Ensamblado:  El código del ensamblado que declara un elemento de tipo Friend puede tener acceso a él.	Interfaces Módulos Clases Estructuras Miembros de estructura Procedimientos Propiedades Variables miembros Constantes	Archivo de código fuente  Espacio de nombres  Interfaz  Módulo  Clase

		Enumeraciones Eventos Declaraciones externas Delegados	Estructura
Protected Friend	Unión de Protected y Friend:  El código de la misma clase o el mismo ensamblado que el elemento de tipo Protected Friend o aquel que está dentro de cualquier clase derivada de la clase del elemento, puede tener acceso a él.	Interfaces Clases Estructuras Procedimientos Propiedades Variables miembros Constantes Enumeraciones Eventos Declaraciones externas Delegados	Clase
Private	Contexto de declaración:  El código del tipo que declara un elemento privado, incluido el código de los tipos contenidos, puede tener acceso al elemento.	Interfaces Clases Estructuras Miembros de estructura Procedimientos Propiedades Variables miembros Constantes Enumeraciones Eventos Declaraciones externas Delegados	Módulo  Clase  Estructura

## 7.2. Ámbito de las variables

Igual que en los formularios, el ámbito o visibilidad de las variables definirá su capacidad de acceso que tenemos a esta variable. Su sintaxis será también:

**Ámbito** Dim variable as tipoVariable

La palabra Dim la podremos omitir en algunas partes de nuestro código, ahora veremos los tipos de ámbito y ejemplos.

### Nivel de procedimiento

Una variable declarada dentro de un procedimiento sólo es visible para él.

## Nivel de módulo

Una variable declarada en la zona de declaraciones del módulo es visible dentro de ese módulo. Acceden a estas variables todos los procedimientos:

```
Module Module1
    Dim valor1 As Long
    Dim cadena1 As String
    Private valor2 As Long

    Public Sub procedimiento1()
        ...
    End Sub
End Module
```

Debemos tener cuidado ya que deben estar declaradas en el módulo, es decir, dentro de la definición del módulo "Module ...End Module". Y aquí... ¿qué significado tiene poner Dim o Private? Las dos son correctas pero si van a ser sólo para este módulo es más correcto poner "Private", así hacemos una pequeña distinción: Dim para las de ámbito de procedimiento y Private para las de ámbito de módulo.

## Nivel de proyecto

Quedan por ver las variables visibles por todo el proyecto entero: las globales. Ya podrás adivinar que la palabra "Public" tiene algo que ver, igual que en los procedimientos. Pues sí, una variable declarada como Public en la zona de declaraciones de un módulo es visible por todo el proyecto completo. Luego si en un módulo declaro una variable como Public su valor será accesible por los demás módulos y formularios. Esto nos será muy útil más adelante.

## 7.3. Duración de las variables

Este es el tiempo en el que las variables están activas y dependen de su ámbito:

- Procedimiento: Se crean al entrar al procedimiento y se liberan o finalizan al terminar el procedimiento.
- Módulo. Si son privadas para ese módulo se inician con él y finalizan al descargarse el módulo.
- Proyecto. Se crean al inicio del proyecto y terminan con él, son pues globales.

## 7.4. Variables Static

La palabra clave **Static** indica que una o varias variables declaradas son estáticas. Las variables estáticas permanecen y conservan los valores más recientes después de que el procedimiento en el que se declaran ha finalizado.

La palabra clave **Static** se utiliza en este contexto:

```
Static variable as long
```

¿Pero qué diferencia hay con las de nivel de procedimiento? Cuando se ejecuta un procedimiento VB.NET inicia las variables para trabajar con ellas. Si declaramos una variable de tipo Static no se inician sino que conserva el valor que tenía en la última llamada. Esto significa que si ya hemos llamado a este procedimiento en otras ocasiones al volver a él conservaremos el valor de las variables Static.

Las utilizaremos poco porque parece más sencillo que si voy a utilizar un valor y quiero que esté disponible más veces lo mejor sería declarar una variable Public y ya está o mandarle un argumento (parámetro) a ese procedimiento con el valor adecuado.

## 8. Las partes o elementos de un proyecto de Visual Basic .NET

Hemos visto una serie de elementos en los que podemos "repartir" el código de un proyecto. Repasemos un poco para consolidar todas estas partes del proyecto. Empecemos con los ensamblados.

### Los ensamblados (assembly)

Para simplificar, un ensamblado es el ejecutable o la librería que podemos crear con VB .NET. En un ensamblado podemos tener clases, módulos y otros elementos tales como los espacios de nombres.

### Los espacios de nombres (namespace)

Los espacios de nombres se usan para agrupar clases y otros tipos de datos que estén relacionados entre sí.

Para acceder a los tipos incluidos en un espacio de nombres hay que indicar el namespace seguido de un punto y el nombre de ese tipo, por ejemplo, una clase. Por ejemplo, para acceder a la clase Console que está en el espacio de nombres System, habría que hacerlo así:

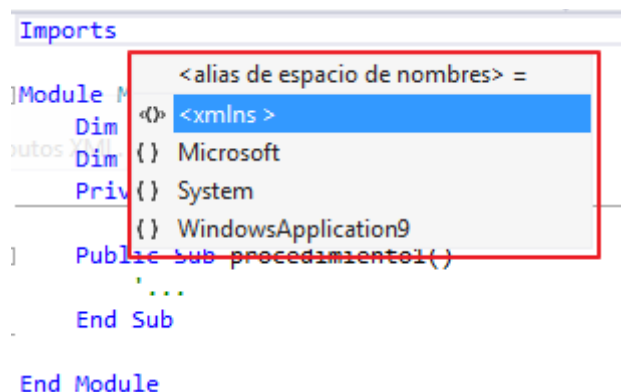
```
System.Console.
```

Para poder definir nuestros propios espacios de nombres, tendremos que usar la instrucción **Namespace** seguida del nombre que queramos darle, y para indicar cuando termina ese espacio de nombres, lo indicaremos con **End Namespace**. Dentro de un espacio de nombres podemos declarar otros espacios de nombres.



**Nota:** Cuando creamos un proyecto de Visual Basic, por defecto se crea un espacio de nombres llamado de la misma forma que el proyecto. Aunque si el nombre del proyecto incluye espacios u otros caracteres "no estándar", estos serán sustituidos por guiones bajos. Todas las declaraciones que hagamos en dicho proyecto se "supondrán" incluidas en ese espacio de nombres, por tanto, para poder acceder a ellas desde fuera de ese proyecto, habrá que usar ese espacio de nombres

Para comprobar esto veamos qué pasa al intentar realizar un Imports en la cabecera del proyecto para añadir un espacio de nombres:



Que confirma lo que hemos dicho porque el tercer "spacename" que aparece en la lista desplegable es el propio proyecto.

## Los módulos y las clases

En Visual Basic .NET podemos crear clases de dos formas distintas, usando la instrucción **Module** o usando la instrucción **Class**. En los dos casos indicaremos el nombre que tendrá ese elemento después de esas instrucciones.

Tanto los módulos como las clases, deben estar declarados dentro de un espacio de nombres. Dentro de una clase podemos definir otras clases, pero no podemos definir módulos dentro de otros módulos.

La diferencia entre un módulo y una clase, es que un módulo define todos sus miembros como compartidos (**Shared**), esto lo veremos con más detalle en otra ocasión, pero ahora lo veremos de forma simple:

Como sabemos, cuando queremos crear un objeto basado en una clase, debemos usar **New** para crear una nueva instancia en la memoria. Cada nuevo objeto creado con New será independiente de los otros que estén basados en esa clase. Por otro lado, para usar los elementos contenidos en un módulo, no necesitamos crear una nueva instancia, los usamos directamente. Esto es así porque esos elementos están compartidos, es decir, siempre existen en la memoria y, por tanto, no es necesario crear un nuevo objeto. Al estar siempre disponible, sólo existe una copia en la memoria.

## Las enumeraciones

Como ya vimos anteriormente las enumeraciones son constantes que están relacionadas entre sí. Las enumeraciones podemos declararlas a nivel de espacios de nombres o a nivel de clases (y/o módulos).

## Las estructuras (Structure)

Las estructuras o tipos definidos por el usuario, son un tipo especial de datos que también hemos visto y que se comportan casi como las clases, permitiendo tener métodos, propiedades, etc. La diferencia principal entre las clases y las estructuras es que éstas últimas son tipos por valor, mientras que las clases son tipos por referencia.

Las estructuras, al igual que las clases, las podemos declarar a nivel de espacios de nombres y también dentro de otras estructuras e incluso dentro de clases y módulos.

Como hemos podido comprobar, tenemos un amplio abanico de posibilidades entre las que podemos escoger a la hora de crear nuestros proyectos, pero básicamente tenemos tres partes bien distintas:

- Los ensamblados
- los espacios de nombres
- el resto de elementos de un proyecto.

En realidad, tanto los espacios de nombres como las clases, estructuras y enumeraciones estarán dentro de los ensamblados. Pero las clases, estructuras y enumeraciones suelen estar incluidas dentro de los espacios de nombres, aunque, no tiene porqué existir un espacio de nombres para que podamos declarar las clases, estructuras y enumeraciones. Lo habitual es que siempre exista un espacio de nombres, el cual es definido de forma explícita al crear un nuevo proyecto.

En el caso de que no definamos un espacio de nombres, y eliminemos el que Visual Basic crea para nosotros, no habrá ningún espacio de nombres y, por tanto, las clases o miembros que definamos en el proyecto estarán accesibles a nivel de ensamblado, sin necesidad de usar ningún tipo de "referencia" extra.

## 9. Las partes o elementos de una clase

Ya hemos hablado en varias ocasiones de las clases y hemos comentado que .NET Framework se compone de clases. Una clase no es ni más ni menos que código. Aunque dicho de esta forma, cualquier programa sería una clase.

Cuando definimos una clase, realmente estamos definiendo dos cosas diferentes: los datos que dicha clase puede manipular o contener y la forma de acceder a esos datos.

Por ejemplo, si tenemos una clase de tipo Cliente, por un lado tendremos los datos de dicho cliente y por otro la forma de acceder o modificar esos datos. En el primer caso, los datos del Cliente, como por ejemplo el nombre, domicilio etc., estarán representados por una serie de campos o propiedades, mientras que la forma de modificar o acceder a esa información del Cliente se hará por medio de métodos. Esas propiedades o características y las acciones a realizar son las que definen a una clase.

Veámoslo de otra forma: una clase es el conjunto de especificaciones o normas que definen cómo se va a crear un objeto, es decir, las instrucciones para crear ese objeto. Una clase es la representación abstracta de algo y el objeto es la representación concreta de lo que una clase define.

Tal y como hemos tenido la oportunidad de ver varias veces, podemos declarar variables y usar procedimientos. Cuando esas variables y procedimientos forman parte de una clase, módulo o estructura tienen un comportamiento "especial", ya que dejan de ser variables y procedimientos para convertirse en "miembros" de la clase, módulo o estructura (e incluso de la enumeración) que lo declare.

Veamos los distintos miembros de las clases y, por extensión, de los módulos y las estructuras:

En las clases podemos tener: campos, propiedades, métodos y eventos:

- **Los métodos** son procedimientos de tipo Sub o Function que realizan una acción.
- **Los campos** son variables usadas a nivel de la clase. Son variables normales que son accesibles desde cualquier parte dentro de la clase e incluso fuera de ella.
- **Las propiedades** podemos decir que son procedimientos especiales. Al igual que los campos, representan una característica de las clases, pero a diferencia de los campos nos permiten hacer validaciones o acciones extras que un campo nunca podrá hacer.
- **Los eventos** son mensajes que utilizará la clase para informar de un hecho que ha ocurrido. Se utilizan para comunicar al que utilice la clase de que se ha producido algo digno de notificar.

La diferencia entre los campos y propiedades lo veremos en otra ocasión posterior, así como el tema de los eventos.

## 9.1. Los procedimientos: métodos de las clases.

Ya hemos utilizado procedimientos en varias ocasiones. De hecho, en toda aplicación de consola existe un procedimiento de tipo Sub llamado Main que es el que se utiliza como punto de entrada del ejecutable.

Los métodos de una clase pueden ser de dos tipos: Sub o Function. Los procedimientos Sub son como las instrucciones o palabras clave de Visual Basic: realizan una tarea. Los procedimientos Function además de realizar una tarea, devuelven un valor, el cual suele ser el resultado de la tarea que realizan. Debido a que las funciones devuelven un valor, esos valores se pueden usar para asignarlos a una variable además de poder usarlos en cualquier expresión.

Para que veamos claras otra vez las diferencias entre estos dos tipos de procedimientos, en el siguiente ejemplo vamos a usar un procedimiento de tipo Sub y otro de tipo Function:

```
Module Module1
    Sub Main()
        MostrarS()
        Dim s As String = MostrarF()
        Console.WriteLine(s)
        Console.ReadLine()
    End Sub
    Sub MostrarS()
        Console.WriteLine("Este es el procedimiento MostrarS")
    End Sub
    Function MostrarF() As String
        Return "Esta es la function MostrarF"
    End Function
End Module
```

La salida producida por este código será la siguiente:

Este es el procedimiento MostrarS

Esta es la función MostrarF

En este módulo tenemos tres procedimientos, dos de tipo Sub y uno de tipo Function, el Sub Main ya lo conocemos de temas anteriores. Es un procedimiento de tipo Sub y, como ya hemos comprobado, ejecuta el código que esté entre la definición del procedimiento, que comienza con la declaración del procedimiento y que siempre se hace de la misma forma: usando Sub seguido del nombre del procedimiento y terminando con End Sub.

Por otro lado, los procedimientos de tipo Function empiezan con la instrucción Function seguido del nombre de la función y el tipo de dato que devolverá la función, ya que, debido a que las funciones siempre devuelven un valor, lo lógico es que podamos indicar el tipo que devolverá. El final de la función viene indicado por End Function.

Pero, las funciones devuelven un valor, el valor que una función devuelve se indica con la instrucción **Return** seguido del valor a devolver. En este ejemplo, el valor devuelto por la función **MostrarF** es el texto que está entrecomillado.

En el procedimiento Main utilizamos el procedimiento Sub usando simplemente el nombre del mismo: **MostrarS**. Ese procedimiento se usa en una línea independiente. Cuando la ejecución del código llegue a esa línea, se procesará el contenido del mismo donde simplemente mostramos un mensaje en la consola.

Por otro lado, el resultado devuelto por la función **MostrarF** se asigna a la variable **s**. Cuando Visual Basic se encuentra con este tipo de asignación, procesa el código de la función y asigna el valor devuelto, por tanto la variable **s** contendrá la cadena "**Esta es la función MostrarF**". Como podemos comprobar por la salida producida al ejecutar este proyecto, eso será lo que se muestre en la consola.

Cuando los procedimientos de tipo Sub o las funciones (Function) pertenecen a una clase se dicen que son métodos de esa clase. Los métodos siempre ejecutan una acción, y en el caso de las funciones, esa acción suele reportar algún valor. Éste se podrá usar para asignarlo a una variable o para usarlo en una expresión, es decir, el valor devuelto por una función se puede usar en cualquier contexto en el que se podría usar una variable o una constante. Por otro lado, los procedimientos de tipo Sub sólo ejecutan la acción sin devolver ningún valor.

Cuando los procedimientos se convierten en métodos, (porque están declarados en una clase), estos suelen representar lo que la clase (o módulo o estructura) es capaz de hacer. Es decir, siempre representarán una acción de dicha clase.

### Parámetros o argumentos de los procedimientos

Podemos pasar parámetros a los procedimientos para proporcionarles datos para los cálculos o procesos que deban realizar.

### Parámetros por valor y parámetros por referencia

También sucede lo mismo en las clases, podemos pasar las variables por valor o por referencia.

Normalmente, cuando se pasa un parámetro a un procedimiento, éste se suele pasar o indicar lo que se llama por valor, es decir, el parámetro será una copia del valor indicado. Cualquier cambio que se realice dentro del procedimiento a la variable nombre no afectará al parámetro. Veamos otra vez un ejemplo:

```
Sub Saludar2(ByVal nombre As String)
    nombre = "Hola " & nombre
```

```
        Console.WriteLine(nombre)
    End Sub
```

Al que llamamos con este otro código:

```
Sub PruebaSaludar2()
    Dim elNombre As String
    elNombre = "Jose"
    Saludar2(elNombre)
    '
    ' ¿qué valor mostraría esta línea?
    Console.WriteLine(elNombre)
End Sub
```

¿Qué valor tendrá la variable **elNombre** después de llamar al procedimiento **Saludar2**?

La respuesta es: el que tenía antes de llamar al procedimiento. ¿Por qué? Si el valor se ha modificado... Porque se ha pasado por valor (**ByVal**) y por tanto, lo que se ha pasado al procedimiento es una copia del contenido de la variable **elNombre**, con lo cual, cualquier cambio que se realice en la variable **nombre** sólo afectará a la copia, no al original.

Pero si queremos que el procedimiento pueda modificar el valor recibido como parámetro, tendremos que indicarle a Visual Basic .NET que lo pase por referencia, para ello habrá que usar la instrucción **ByRef** en lugar de **ByVal**. Veámoslo con un ejemplo:

```
Sub Saludar3(ByRef nombre As String)
    nombre = "Hola " & nombre
    Console.WriteLine(nombre)
End Sub
```

```
Sub PruebaSaludar3()
    Dim elNombre As String
    elNombre = "Jose"
    Saludar3(elNombre)
    '
    ' ¿qué valor mostraría esta línea?
    Console.WriteLine(elNombre)
End Sub
```

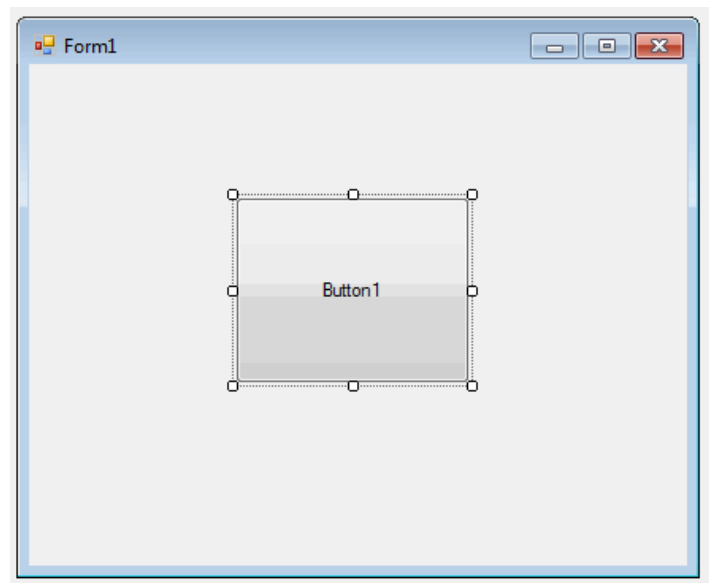
En esta ocasión la variable **elNombre** contendrá "Hola Jose". La explicación es que al pasar la variable por referencia (**ByRef**), VB asigna a la variable **nombre** del procedimiento la misma dirección de memoria que tiene la variable **elNombre**, de forma que cualquier cambio realizado en **nombre** afectará a **elNombre**.

**Nota:** En Visual Basic .NET, de forma predeterminada, los parámetros serán ByVal (por valor). Es decir, si declaras un parámetro sin indicar si es ByVal o ByRef, VB.NET lo interpretará como si fuera ByVal

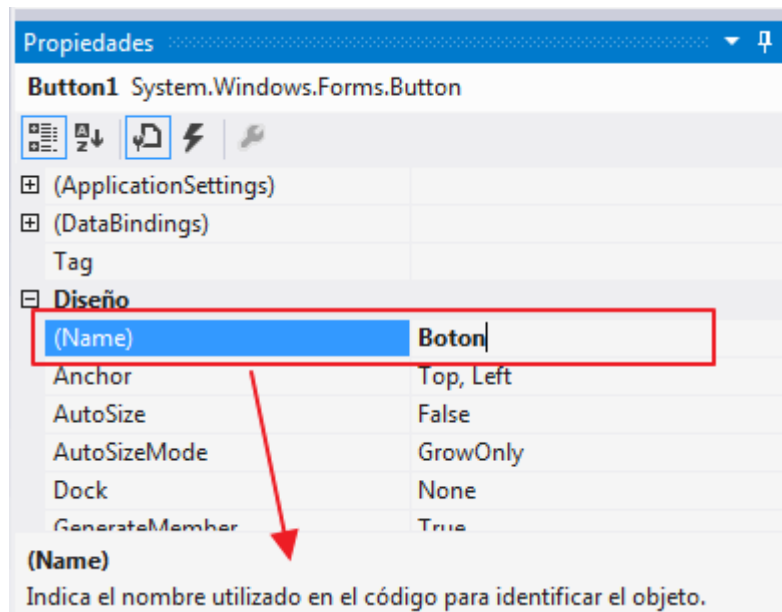
Toda esta sección nos servirá como anticipo de la programación orientada a objetos. Con esto hemos visto algo de lo que son las clases y de qué se componen. En el siguiente tema veremos varios grupos de funciones imprescindibles para desarrollar aplicaciones y veremos varios ejemplos de código aplicándolas.

## 10. Más cosas sobre los eventos

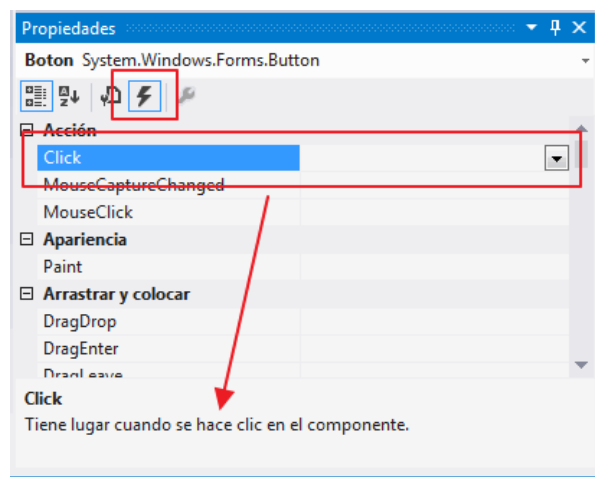
Ya hemos trabajado varias veces con los eventos pero vamos a hacer una pequeña aplicación que nos muestra más cosas sobre cómo trabajar con los eventos. Creamos una aplicación y le ponemos un botón en medio que se llame "boton":



El nombre se lo indicamos en las propiedades, teniendo seleccionado el botón en el editor:



Ya sabemos tratar el evento clic del botón. Para acceder al controlador de este evento podemos hacerlo desde el código o desde la ventana de propiedades, y seleccionando los eventos:



Hacemos doble clic en la línea que he marcado y nos iremos directamente al controlador del evento clic del botón. Escribimos:

```
Public Class Form1
    Private Sub Boton_Click(sender As Object, e As EventArgs) Handles Boton.Click
        MsgBox("Has pulsado el botón con el texto: " & Boton.Text)
    End Sub
End Class
```

Ahora vamos a detectar cuando el usuario pasa el ratón por encima del botón y cuando deja de pasarlo. Estos eventos están prácticamente en todos los controles de Windows. El primero de ellos es el de "Mouse\_Enter" y el otro el de "Mouse\_Leave" así que los seleccionamos:

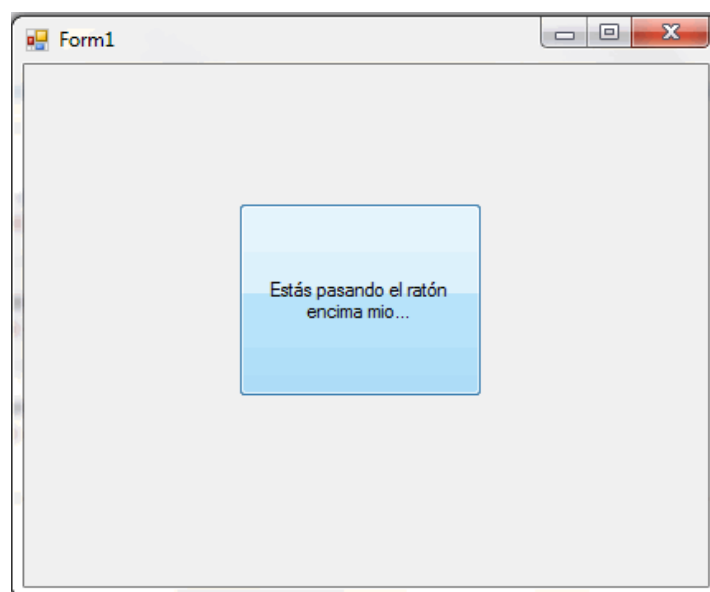
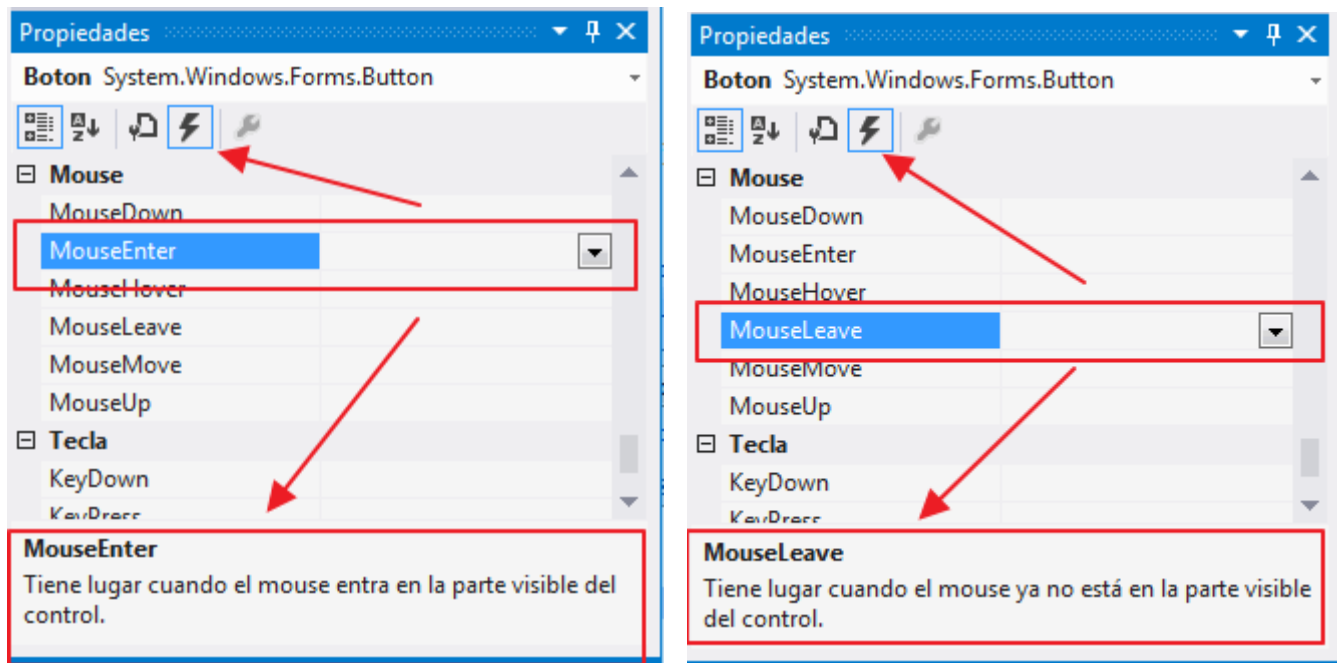
Y escribimos en ellos lo siguiente:



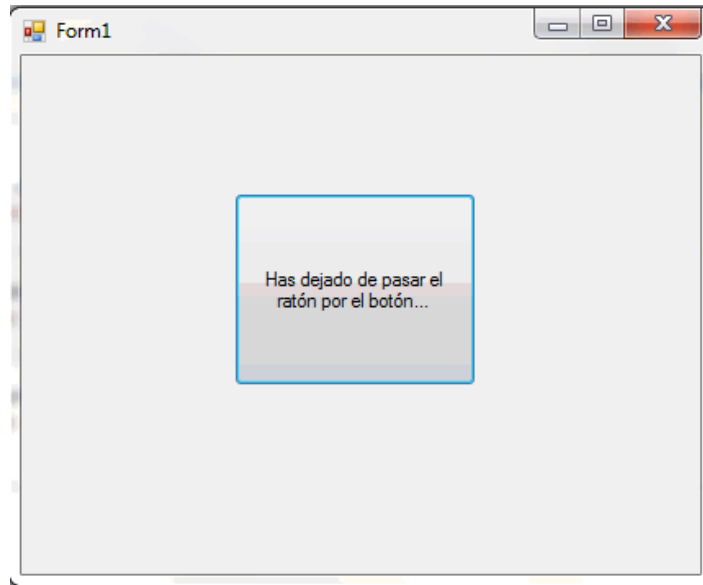
```
Private Sub Boton_MouseEnter(sender As Object, e As EventArgs) Handles Boton.MouseEnter
    Boton.Text = "Estás pasando el ratón encima mio..."
End Sub
```

```
Private Sub Boton_MouseLeave(sender As Object, e As EventArgs) Handles Boton.MouseLeave
    Boton.Text = "Has dejado de pasar el ratón por el botón..."
End Sub
```

Pulsamos en ejecutar y pasamos el ratón por encima:



Y dejamos de pasar por él...

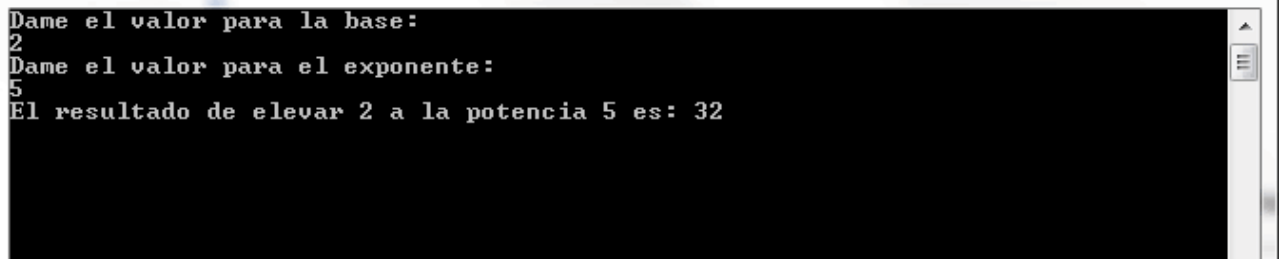


Es un ejemplo muy sencillo pero muy significativo de la filosofía de los eventos. Ya has podido comprobar que hay multitud de eventos a los que pueden atender los controles. Habitualmente utilizaremos un par de ellos pero tendremos a nuestra disposición todas las posibilidades.

## Ejercicios

### Ejercicio 1

Lee dos datos en una aplicación de consola, crea una función para calcular una potencia y escribe el resultado.

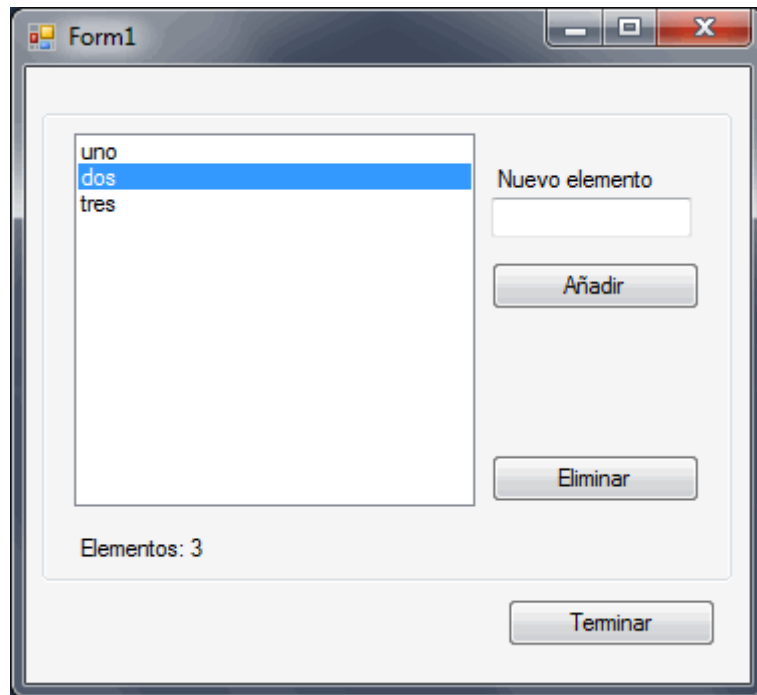


```
Dame el valor para la base:
2
Dame el valor para el exponente:
5
El resultado de elevar 2 a la potencia 5 es: 32
```

Nota: La potencia la puedes hacer multiplicando n veces la base. Es decir,  $2^3$  será multiplicar 3 veces 2:  $2 \times 2 \times 2 = 8$

## Ejercicio 2

Crea una aplicación Windows con un cuadro de texto, un cuadro de lista, un botón y un label que indique cuántos elementos hay en la lista.



Escribe procedimientos independientes para insertar y eliminar elementos del cuadro de lista. Comprueba que sólo se inserten elementos si hay algo en el cuadro de texto y que se eliminen elementos sólo si hay alguno seleccionado.

Utiliza una variable global declarada en un archivo de módulo que añadirás al proyecto para almacenar el número de elementos de la lista. Este módulo también tendrá el procedimiento para escribir este número de elementos.

Nota: Para referirse a un elemento de otro formulario lo indicaremos con el nombre completo: en lugar de Me. que hace referencia al formulario actual, utilizar `form1.control.text="Hola"`