

TEMA 5

Flujo de programa. Bucles.

1. Bucles en Visual Basic .NET

En este tema seguimos viendo las instrucciones básicas de VB.NET para el control de nuestro programa. Ahora le toca el turno a los bucles. Cada vez que queramos repetir una instrucción o un grupo de ellas un número determinado de veces, e incluso un número indeterminado de veces utilizaremos los bucles.

En Visual Basic .NET existen diferentes instrucciones para hacer bucles, cada tipo tiene sus peculiaridades que el conocerlas nos va a permitir elegir el que más se adecue para la resolución de nuestro problema.

1.1. Bucles For / Next.

Con este tipo de bucle podemos repetir un bloque de instrucciones un número determinado de veces. Su sintaxis es:

```
For <variable numérica> = <valor inicial> To <valor final> [Step <incremento>]  
    ' contenido del bucle, lo que se va a repetir  
Next
```

La *variable numérica* tomará valores que van desde el *valor inicial* hasta el *valor final*, si no se especifica el valor del *incremento*, éste será 1.

Pero si nuestra intención es que el valor del incremento sea diferente a 1, habrá que indicar un valor de incremento. Lo mismo tendremos que hacer si queremos que el valor inicial sea mayor que el final, con idea de que "cuenta" de mayor a menor, aunque en este caso el incremento en realidad será un "decremento" ya que el valor de incremento será negativo. Veámoslo con unos ejemplos:

```
Dim i As Integer  
For i = 1 To 10  
    ' contará de 1 hasta 10  
    ' la variable i tomará los valores 1, 2, 3, etc.  
Next  
'  
For i = 1 To 100 Step 2  
    ' contará desde 1 hasta 100 (realmente 99) de 2 en 2  
    ' la variable i tomará los valores 1, 3, 5, etc.  
Next
```

```
'  
For i = 10 To 1 Step -1  
    ' contará desde 10 hasta 1  
    ' la variable i tomará los valores 10, 9, 8, etc.  
Next  
'  
For i = 100 To 1 Step -10  
    ' contará desde 100 hasta 1, (realmente hasta 10)  
    ' la variable i tomará los valores 100, 90, 80, etc.  
Next  
'  
For i = 10 To 1  
    ' este bucle no se repetirá ninguna vez. Le falta el STEP  
Next  
'  
For i = 1 To 20 Step 50  
    ' esto sólo se repetirá una vez  
Next
```

Como ves, la sintaxis es muy sencilla. Pero en algunos casos hay que tener en cuenta que el valor final del bucle puede que no sea el indicado, todo dependerá del incremento que hayamos especificado. Por ejemplo, en el segundo bucle, le indicamos que cuente desde 1 hasta 100 de dos en dos, el valor final será 99.

En otros casos, puede incluso que no se repita ninguna vez. Este es el caso del penúltimo bucle, ya que le decimos que cuente de 10 a 1, pero al no indicar Step con un valor diferente, Visual Basic "supone" que será 1 y en cuanto le suma uno a 10, se da cuenta de que 11 es mayor que 1 y como le decimos que queremos contar desde 10 hasta 1, pues... sabe que no debe continuar.

1.2. Bucles For Each

Una instrucción **For Each...Next** repite bucles basados en los elementos de una expresión. Una instrucción **For Each** especifica una variable de control de bucle y una expresión enumeradora. El tipo del valor devuelto por la expresión en una instrucción **For Each** debe ser un tipo de esa misma colección.

El bucle **For Each...Next** es parecido al bucle **For...Next**, pero ejecuta el bloque de instrucciones una vez por cada elemento de una colección, en vez de un número de veces especificado. Se trata de una variante del bucle For-Next en la que se recorrerán los elementos de una colección (aunque veremos las colecciones más adelante). Digamos que es como moverse por los elementos de un array o matriz. Su sintaxis es:

```
For Each <variable> In <colección del tipo de la variable>  
    ' lo que se hará mientras se repita el bucle  
Next
```

Los elementos de la colección pueden ser de cualquier tipo de datos.

Este tipo de bucle lo veremos con más detalle en otros temas, pero aquí veremos un par de ejemplos. Por ejemplo, como un "string" es una cadena de caracteres, podemos usar este tipo de bucle para recorrer cada uno de ellos. Veamos:

```
Dim s As String
'
For Each s In "Hola Mundo"
    Console.WriteLine(s)
Next
Console.ReadLine()
```

Que viene a decir que "por cada carácter en 'Hola Mundo' escríbelo."

Podemos recorrer cualquier colección de objetos, por ejemplo la colección de controles de un formulario:

```
Sub cambia_color(ByVal formulario As System.Windows.Forms.Form)

    For Each miControl As System.Windows.Forms.Control In Controls
        miControl.BackColor = System.Drawing.Color.LightBlue 'es un valor de
color predefinido
    Next miControl
End Sub
```

Vemos que se ha hecho la declaración en una sola línea junto con el bucle:

```
For Each miControl As System.Windows.Forms.Control In Controls
```

En lugar de haber hecho:

```
Dim miControl As System.Windows.Forms.Control
For Each miControl In Controls
```

Da igual, el resultado es el mismo y veremos esta tendencia de declarar las variables en el momento de utilizarlas en muchas ocasiones. Otro ejemplo para buscar dentro de una colección, donde debemos fijarnos en las declaraciones de los objetos que se utilizan:

```
Dim encontrado As Boolean = False
Dim coleccion As New Collection

For Each cadena As String In coleccion
    If cadena = "Hola" Then
        encontrado = True
        Exit For
    End If
Next cadena
```

1.3. Bucles While / End While

Este bucle se repetirá mientras que se cumpla la expresión lógica que se indicará después del While. Su sintaxis es:

```
While <expresión>  
    ' lo que haya que hacer mientras se cumpla la expresión  
End While
```

Tiene una gran diferencia con los bucles For / Next. Así como en éstos debíamos decirle cuántas veces se debe repetir, en este caso no hace falta ya que se va a cumplir mientras sea válida una condición. Por ejemplo, tenemos un fichero con nuestros clientes, deseo recorrerlo entero y para esto voy a utilizar un bucle. Nos planteamos inicialmente el For / Next porque es más sencillo de implementar, pero... ¿cuántas veces debo realizar el bucle? No lo sé, hoy puede que tenga 200 clientes pero luego o mañana ese valor cambiará, así que debo utilizar otro tipo de bucle.

Puedo utilizar el *While* porque le puedo añadir una evaluación y ésta es *mientras existan registros*. Así que este es el adecuado:

```
While no_fin fichero  
    Escribe_datos  
End While
```

Este ejemplo no se va a ejecutar 200 ni 300 veces sino cuando se cumpla que ha llegado al final del fichero.

Con este tipo de bucles, se evalúa la expresión y si el resultado es un valor verdadero, se ejecutará el código que esté dentro del bucle, es decir, entre While y End While. La expresión será una expresión lógica que se evaluará para conseguir un valor verdadero o falso.

Veamos algunos ejemplos:

```
Dim i As Integer  
'  
While i < 10  
    Console.WriteLine(i)  
    i = i + 1  
End While  
Console.ReadLine()  
'  
'  
Dim n As Integer = 3  
i = 1  
While i = 10 * n  
    ' no se repetirá ninguna vez  
End While
```

En el primer caso, el bucle se repetirá mientras i sea menor que 10, fíjate que el valor de i se incrementa después de mostrarlo. Por tanto, se mostrarán los valores desde 0 hasta 9, ya que cuando i vale 10, no se cumple la condición y sale del bucle.

Nota: Un bucle For / Next se repite un número finito de veces indicado por el valor "To". Los bucles condicionales se ejecutan mientras se cumplen una expresión así que habrá que tener mucho cuidado para que se llegue a cumplir alguna vez. Si no se cumple esa terminación se quedará en un bucle infinito bloqueando la aplicación. En el ejemplo anterior incrementa la variable $i=i+1$ para seguir con las iteraciones o repeticiones del bucle.

En el segundo ejemplo no se repetirá ninguna vez, ya que la condición es que i sea igual a 10 multiplicado por n , cosa que no ocurre, ya que i vale 1 y n vale 3 y como sabemos 1 no es igual a 30.

1.4. Bucle Do / Loop

Es muy parecido al anterior pero más versátil. Si se utiliza sólo con esas dos instrucciones, este tipo de bucle no acabará nunca y repetirá todo lo que haya entre Do y Loop de forma infinita, así que le pondremos una expresión para que la evalúe y pueda terminar con el bucle.

```
'Está incompleto porque debemos poner una condición para que
finalice el bucle:
Do
    'instrucciones
Loop
```

Podemos utilizar dos instrucciones que nos permitirán evaluar expresiones lógicas: **While** y **Until**

La ventaja de usar While o Until con los bucles Do/Loop es que estas dos instrucciones podemos usarlas tanto junto a Do como junto a Loop. La diferencia está en que si los usamos con Do, la evaluación se hará antes de empezar el bucle, mientras que si se usan con Loop, la evaluación se hará después de que el bucle se repita al menos una vez.

Veamos cómo usar este tipo de bucle con las dos "variantes":

Caso 1:
Do While <expresión>
 'instrucciones
Loop

Caso 2:
Do
 'instrucciones
Loop While <expresión>

Caso 3:

```
Do Until <expresión>  
    'instrucciones  
Loop
```

Caso 4:

```
Do  
    'instrucciones  
Loop Until <expresión>
```

Si utilizamos Do con While no se diferencia en nada con lo que vimos anteriormente. La excepción está en que se utilice junto a Loop. En ese caso la evaluación de la expresión se hace después de que se repita como mínimo una vez.

Si utilizamos Until, a diferencia de While, la expresión se evalúa cuando no se cumple la condición, es como si negáramos la expresión con While (Not <expresión>) o mejor dicho y traducido uno hace "Mientras se cumpla esta condición haz esto: (Do While)" y el otro dice: "Haz esto hasta que se cumpla esta condición. (Do Until)"

Veamos un ejemplo con DO Until: Do Until X > 10 (repite hasta que X sea mayor que 10):

```
i = 0  
Do Until i > 9  
    Console.WriteLine(i)  
    i = i + 1  
Loop
```

Este bucle se repetirá para valores de i desde 0 hasta 9 (ambos inclusive). Y este también:

```
i = 0  
Do While Not (i > 9)  
    Console.WriteLine(i)  
    i = i + 1  
Loop
```

En el caso de que Until se use junto a Loop, la expresión se comprobará después de repetir al menos una vez el bucle.

1.5. Finalización anticipada de bucles

Normalmente los bucles terminarán cuando cumplan su condición de salida. Pero a veces nos interesa romper el bucle y terminarlo sin que lleguen a evaluarse más condiciones. O en el caso del For-Next sin llegar a cumplirse el valor de "To"

Para poder abandonar un bucle, (esto veremos que es ampliable a otros temas), hay que usar la instrucción **Exit** seguida del tipo de bucle que queremos abandonar:

- **Exit For**
- **Exit While**
- **Exit Do**

Esto es útil si necesitamos abandonar el bucle por medio de una condición, normalmente se utiliza con un If / Then.

1.6. Ejemplo

Vamos a ver dos algoritmos ampliamente utilizados para ver con más detalle esto de los bucles y su ruptura. El problema es el siguiente: tengo una matriz de 200 elementos y quiero buscar la cadena "Felipe" dentro de esa matriz.

Un algoritmo es un conjunto de instrucciones que resuelven un problema, así que a partir de ahora cuando digamos que vamos diseñar un algoritmo o lo pidamos en las prácticas significa que hay que diseñar una rutina o grupo de instrucciones para resolver el problema.

Sigamos con nuestro caso y veamos gráficamente la matriz o array:

Lista_empleados			
0	1	2	n
Ana	Luis	Angel	Felipe

Tenemos la ventaja de que sabemos que como máximo hay 200 elementos así que parece que el bucle a utilizar puede ser el For-Next. Pero si "Felipe" está en la posición 20, lo hemos encontrado pero el bucle seguirá hasta el final lo cual es una evidente pérdida de tiempo y un fallo de diseño (aunque funcione)

Podríamos solucionar la ruptura del bucle con el truco que hemos visto antes de "Exit For". Pero vayamos por partes, primero vamos a hacerlo mal y vemos el porqué de introducirle mejoras:

1ª Solución: bucle For-Next

Partimos de una matriz que se definió así:

```
dim lista_empleados (199) as string
```

Como existe el elemento 0, para que tenga 200 el tamaño debe ser en la declaración de 199. Luego otro procedimiento o rutina insertó los nombres en las 200 posiciones. Así que partimos de un problema habitual que es la búsqueda de un elemento en una matriz. Una resolución sencilla puede ser:

```
Sub buscar()  
    dim i as long  
    dim posicion as long  
    For i=0 to 199  
        if lista_empleados(i)="Felipe" then  
            posicion=i  
        end if  
    Next  
    Console.WriteLine ("Encontrado en la posición: " & posicion)  
End Sub
```

Está sin comentarios para ver el código más claro, ahora lo analizaremos. Primero creamos un procedimiento que va a buscar el elemento y le pondremos de nombre *buscar*. La sintaxis es: (más adelante veremos en profundidad esta sintaxis).

```
Sub buscar()  
    ...  
    ...  
End Sub
```

Ahora necesitamos recorrer el bucle entero para buscar el elemento así que declaro una variable "i" para el bucle For. Habitualmente se utilizan las variables i, j, k par recorrer bucles For. También voy a declarar otra variable que va a almacenar la posición de "Felipe" en la matriz que es el objetivo del algoritmo.

```
dim i as long  
dim posición as long
```

Ya podemos realizar un bucle desde el primer elemento hasta el último:

```
For i=0 to 199  
    ...  
Next
```

Con esto recorreré todos los elementos de la matriz desde el 0 al 199. Ahora debo comparar si el elemento en el que esté es el que estoy buscando:

```
if lista_empleados(i)="Felipe" then  
    ...  
end if
```

Es decir, si el elemento de la matriz posición "i" es igual a la palabra "Felipe" haz lo siguiente. Recuerda que tanto para acceder como para introducir un valor en una matriz basta con indicarle su posición o índice. Por ejemplo para acceder al elemento 13 será valor=matriz(12) y para meter un valor en la matriz será matriz(34)="Antonio". Sigamos, como ya he encontrado a "Felipe" asignaré en una variable el valor de i, es decir, la posición del bucle en la que estoy:


```
if lista_empleados(i)="Felipe" then
    posicion=i
end if
```

Luego tengo el resultado almacenado en la variable posición y el problema resuelto. Finalmente le escribo al usuario la posición donde se encontraba "Felipe"

```
Console.WriteLine ("Encontrado en la posición: " & posicion)
```

Aquí he utilizado la instrucción conocida para escribir en una aplicación de consola y he escrito dos cosas que aparecen concatenadas con el símbolo de concatenar "&". Por un lado, he concatenado la frase "Encontrado en la posición: " y por otro el valor de una variable 'posición'. Al escribir uno es constante y va entre comillas y el otro no porque es una variable. Si "Felipe" estuviese en la posición 30 escribiría:

```
Encontrado en la posición: 30
```

Y ya está terminado el algoritmo.

Ahora veamos las posibles mejoras. La primera y más importante es que si "Felipe" aparece en la posición 30 es absurdo que el bucle se realice 200 veces. Así que vamos con la primera mejora...

2º Solución: bucle For con ruptura

Sabemos ya que debemos finalizar el bucle For cuando hayamos logrado nuestro objetivo así que introduciremos la instrucción "Exit For" para decirle al programa que rompa el bucle para finalizarlo. ¿Dónde tendré que colocarlo? Pues lógicamente cuando encuentre lo que quiera, es decir, cuando se cumpla la comparación del elemento de la matriz con "Felipe":

```
Sub buscar()
    dim i as long
    dim posicion as long
    For i=0 to 199
        if lista_empleados (i)="Felipe" then
            posicion=i
            Exit For
        end if
    Next
    Console.WriteLine ("Encontrado en la posición: " & posicion)
End Sub
```

Y funcionará bien, en el momento que se cumpla esto se interrumpirá el bucle For y así aunque lo haya encontrado en la posición número 5 no continuará con las repeticiones.

Pero esto no es lo más elegante. Si un bucle "For" está diseñado para realizar una repetición un número determinado de veces, provocar una interrupción parece introducirle algo poco natural, como si fuera un "parche" para que funcione. Rompemos la filosofía del bucle For.

Así que vamos a pensar en otra solución más elegante: ¿Por qué no hacemos un bucle condicional que simplemente termine cuando encuentre a "Felipe"? Pues sí, esa es la mejor solución. Vamos con ella...

3ª Solución: bucle condicional

Utilizaremos un bucle "Do While" para recorrer nuestra matriz.

Para hacer un bucle para la matriz con un "Do-While" tengo un problema y es que así como en el bucle "For-Next" éste incrementa automáticamente el índice en un "Do-While" lo tendré que crear a mano así que sería algo parecido a esto, compáralo con el "For-Next" de la derecha.

<pre>Sub buscar() dim i as long i=0 Do While (i<199) i=i+1 Loop End Sub</pre>	<pre>Sub buscar() dim i as long For i=0 to 199 Next End Sub</pre>
--	--

La diferencia es que en el de la parte derecha (For) al incrementarse de forma automática el índice "i" no tenemos que preocuparnos de nada, en cambio en el otro tendremos que hacer dos modificaciones. Por un lado, le tenemos que decir el valor inicial: i=0 y luego dentro del "Do While" incrementaremos a mano el índice para que pase al siguiente.

En los bucles "Do While" la condición de salida se debe cumplir alguna, vez sino se quedaría en un bucle infinito. Si nosotros nos equivocamos (cosa normal en cualquier proceso de aprendizaje) y nos olvidamos de ese "i=i+1" resulta que nunca se podrá cumplir la condición de "i<199" porque "i" tendrá siempre el mismo valor "i=0" lo que provocará un bucle infinito y nos bloqueará la aplicación.

Luego la misma resolución sin romper el bucle sería:

```
Sub buscar()  
    dim i as long
```

```
        i=0
    Do While (i<199)
        if lista_empleados(i)="Felipe" then
            posicion=i
        end if
        i=i+1
    Loop
    Console.WriteLine ("Encontrado en la posición: " & posicion)
End Sub
```

Luego tenemos lo mismo que en el caso 1. Lo encontramos pero seguimos recorriendo el bucle así que vamos a mejorarlo.

4ª Solución: bucle condicional con ruptura

Así que una solución "a medias" es la inclusión de un "Exit Do" en el bucle cuando hayamos encontrado lo que queremos:

```
Sub buscar()
    dim i as long
    i=0
    Do While (i<199)
        if lista_empleados (i)="Felipe" then
            posicion=i
            Exit Do
        end if
        i=i+1
    Loop
    Console.WriteLine ("Encontrado en la posición: " & posicion)
End Sub
```

Funcionará, pero tampoco es la solución perfecta porque si el bucle "Do While" se cumple con determinadas condiciones ¿por qué no son estas condiciones las que interrumpen el bucle? Me explico: podemos hacer que la condición de salida de ese bucle sea cuando llegue hasta el último elemento o cuando lo encuentre, veamos el último caso.

5ª Solución: bucle condicional con "centinela"

La idea final es hacer que el bucle se repitan x veces mientras se cumpla que:

- No hemos llegado hasta el final de la matriz
- No hemos encontrado a "Felipe"

Cuando sea "True" alguna de esas dos comparaciones debemos finalizar el bucle. Para esto nos vamos a apoyar en una variable de tipo Boolean, que sólo puede contener dos valores: True ó False. La vamos a llamar "encontrado" y funcionará de esta forma: le pondremos como valor inicial "False". Cuando se vaya recorriendo el bucle haremos la comparación para buscar a

"Felipe". En cuanto lo encontremos le cambiaremos su valor a "True" para que provoque que se interrumpa el bucle al cumplirse esta comparación. Veamos este código:

```
encontrado=False
Do while Not encontrado 'o encontrado=false que es lo mismo
    if lista_empleados(i)="Felipe" then
        encontrado=True
    end if
Loop
```

Se ha omitido la parte del contador "i" para que se vea mejor. Así que simplemente lo que hace es que repite el bucle mientras "Not encontrado" o lo que es lo mismo: "encontrado=false". En el momento que se encuentre a "Felipe" le asignaremos el valor "True" a la variable "Encontrado". Esto provocará que ya no se cumpla la condición del "Do" porque "encontrado" vale "True" por lo que finalizará el bucle. Veamos todo junto:

```
Sub buscar()
    dim i as long
    dim posicion as long
    dim encontrado as boolean

    encontrado=False
    i=0
    Do while (i<199) and (Not encontrado)
        if lista_empleados(i)="Felipe" then
            encontrado=True
            posicion=i
        end if
        i=i+1
    Loop
    If encontrado=true then
        Console.WriteLine ("Encontrado en la posición: " & posicion)
    else
        Console.WriteLine ("No he encontrado a Felipe")
    end if
End sub
```

Fíjate bien en los pasos:

- Declaramos las variables a utilizar
- Iniciamos las variables
- Comenzamos el recorrido
- Termina el recorrido bien porque llegamos al final o porque hemos encontrado a "Felipe"
- Escribimos el resultado

La parte de condición del bucle queda como:

```
Do while (i<199) and (Not encontrado)
    ...
Loop
```

Es decir, cuando una de las dos condiciones no se cumpla terminará el bucle. Finalmente escribimos un resultado dependiendo de si lo ha encontrado o no, que en este caso será cuando la variable booleana "encontrado" tenga el valor "true".

Este algoritmo es muy especial porque pertenece a la lista de los estándar que se enseñan en todos los libros de programación e incluso tiene nombre: "**algoritmo del centinela**". El nombre es evidente, hay un centinela (variable *encontrado*) que indica cuándo se debe detener el bucle.

Sólo una mejora más para este ejemplo. En lugar de poner fijo el valor 199 de la matriz, ¿por qué no le ponemos una función que nos diga cuántos valores tiene la matriz?. Ya sabemos que con "Redim" se puede cambiar el tamaño de la matriz. Puede que a lo largo del programa se haya modificado y ahora tengamos 250 elementos. Entonces nuestro bucle no funcionará bien. Para solucionarlo, en lugar de ponerle 199 pondremos: Ubound(lista_empleados). "Ubound" nos decía el valor máximo de índice de la matriz, así que:

```
Sub buscar()
    dim i as long
    dim posicion as long
    dim encontrado as boolean

    encontrado=False
    i=0
    Do while (i<Ubound(lista_empleados)) and (Not encontrado)
        if lista_empleados(i)="Felipe" then
            encontrado=True
            posicion=i
        end if
        i=i+1
    Loop
    If encontrado=true then
        Console.WriteLine ("Encontrado en la posición: " & posicion)
    else
        Console.WriteLine ("No he encontrado a Felipe")
    end if
End sub
```

Es decir, mientras sea menor que el número de elementos (Ubound).

Una vez vistos todos los bucles, sigamos con más cosas importantes del código de VB .Net

2. Las enumeraciones (Enum)

Una enumeración es un tipo especial de variable numérica en la que los valores que puede tomar son constantes simbólicas. Esto es, que en lugar de utilizar un número, se usa una palabra (constante) que hace referencia a un número.

Las enumeraciones proporcionan una forma cómoda de trabajar con conjuntos de constantes relacionadas y de asociar valores de constantes con nombres. Por ejemplo, se puede declarar una enumeración para un conjunto de constantes de tipo entero asociadas con los días de la semana, y después utilizar los nombres de los días en el código en lugar de sus valores enteros.

Por ejemplo, si queremos tener una variable llamada *color* y queremos que contenga un valor numérico que haga referencia a un color en particular. Así en lugar de usar el valor 1, 2 ó 3, queremos usar la constante rojo, azul, verde, etc. Esto lo haríamos de esta forma:

```
Enum colores
    rojo = 1
    azul
    verde
End Enum
```

Las declaraciones de las enumeraciones hay que hacerlas fuera de cualquier procedimiento. Por ejemplo, dentro de una clase o un módulo, pero también pueden estar declaradas dentro de un espacio de nombres, todo dependerá del alcance (o ámbito de acceso) que queramos darle.

Los valores que pueden tener los miembros de una enumeración, pueden ser cualquiera del tipo entero o real. Por defecto, el tipo es Integer, pero las enumeraciones también pueden ser de tipo Byte, Long o Short. Para poder especificar un tipo diferente a Integer, lo indicaremos usando *As Tipo* después del nombre de la enumeración.

Por defecto, el primer valor que tendrá un elemento de una enumeración será cero y los siguientes elementos, salvo que se indique lo contrario, tendrán uno más que el anterior.

En el ejemplo mostrado, el elemento rojo, valdrá 1, azul valdrá 2 y verde tendrá un valor 3.

Para poder cambiar esos valores automáticos, podemos indicarlo usando una asignación como la usada para indicar que rojo vale 1: rojo = 1

En caso de que no indiquemos ningún valor, el primero será cero y los siguientes valdrán uno más que el anterior, por ejemplo, si la declaración anterior la hacemos de esta forma:

```
Enum colores
    rojo
    azul
    verde
End Enum
```

En este caso el rojo valdrá 0, azul será igual a 1 y verde tendrá el valor 2.

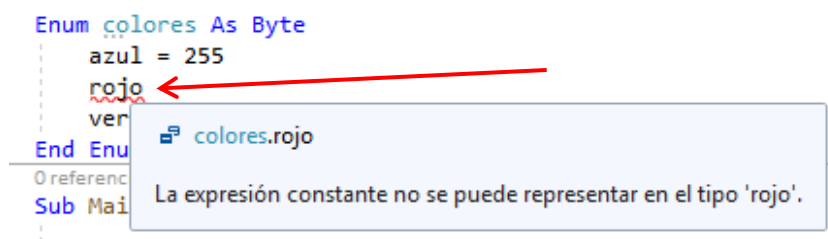
La asignación podemos hacerla en cualquier momento, en el siguiente caso, rojo valdrá cero, azul tendrá el valor 3 y verde uno más que azul, es decir, 4.

```
Enum colores
    rojo
    azul = 3
    verde
End Enum
```

Por supuesto, los valores que podemos asignar a los elementos (o miembros) de una enumeración serán valores que estén de acuerdo con el tipo de datos. Recordemos que si no indicamos nada, serán de tipo Integer, pero si especificamos el tipo de datos, por ejemplo, de tipo Byte, visto en el tema 4, sólo podrá contener valores enteros comprendidos entre 1 y 255. Sabiendo esto, no podríamos declarar la siguiente enumeración sin recibir un mensaje de error:

```
Enum colores As Byte
    azul = 255
    rojo
    verde
End Enum
```

¿Por qué? te preguntarás, ¿si el valor está dentro de los valores permitidos? Por la sencilla razón de que azul tiene un valor adecuado, pero tanto rojo como verde, tendrán un valor 256 y 257 respectivamente, los cuales están fuera del rango permitido por el tipo Byte.



Otra cosa que tendremos que tener en cuenta es que una variable declarada del tipo de una enumeración, en teoría no debería admitir ningún valor que no esté incluido en dicha enumeración. Aunque esta restricción es sólo recomendable no es algo obligatorio. Si tenemos activado Option Strict On, el IDE de Visual Studio .NET nos lo recordará, con lo que nos obligará a hacer una conversión de datos entre la variable (o el valor usado) y el tipo "correcto" al que corresponde la enumeración. Veamos un ejemplo:

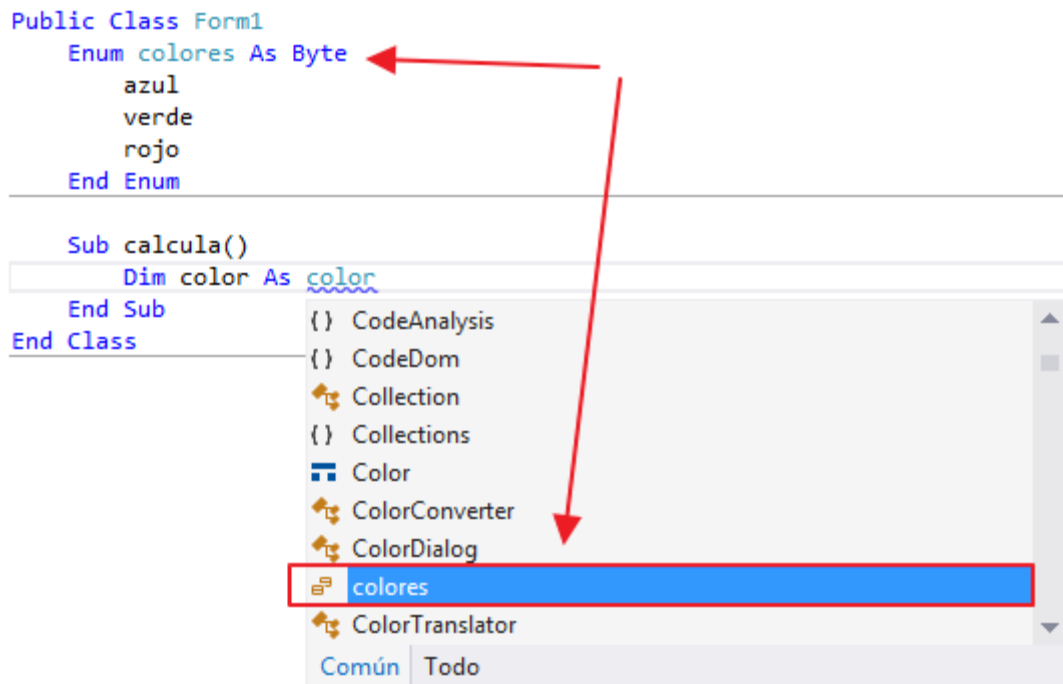
```
Dim unColor As colores
unColor = 1
```

Aunque el valor 1, sea un valor correcto, si tenemos Option Strict On, nos indicaría que no se permite la conversión implícita entre Integer y el tipo colores. Si no tuviéramos activada la comprobación estricta, no te mostraría ningún error. Pero si sabemos que para un buen uso de los tipos de datos, deberíamos hacer la conversión correspondiente, ¿por qué no hacerla?

```
unColor = CType(1, colores)
```

Es decir, usamos CType para hacer la conversión entre el número y el tipo correcto de datos.

De todas formas, el IDE de Visual Studio .NET nos mostrará los valores que puedes asignarle a la variable declarada del tipo colores. Además en la declaración veremos cómo aparece en la lista desplegable para que lo podamos utilizar en nuestro programa. Escribe el ejemplo de esta figura:



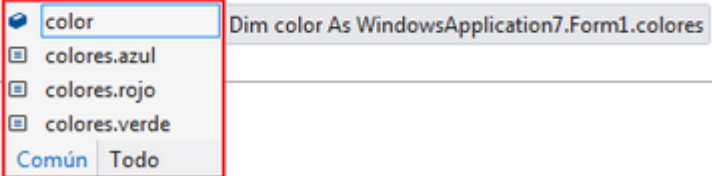
Es decir, primero creamos la enumeración con "Enum" y luego vamos a utilizarla. Escribimos un procedimiento cualquiera y al escribir "Dim color as ..." se desplegará una lista con los tipos posibles y veremos que aparece nuestro "colores", luego vamos bien. El segundo paso es utilizarlo:


```
Public Class Form1
    Enum colores As Byte
        azul
        verde
        rojo
    End Enum

    Sub calcula()
        Dim color As colores

        color=

    End Sub
End Class
```



Ya lo tenemos declarado ahora vamos a utilizarlo. Como "color" es una variable de tipo colores la enumeración anterior sólo me debe permitir seleccionar uno de los colores que he definido antes. Efectivamente al escribir "color=" aparece otro desplegable con las opciones posibles que son precisamente los colores definidos antes.

Ahora veremos algunas funciones o métodos de las enumeraciones que nos serán muy útiles. Observa esta línea de código

```
If System.Enum.IsDefined(GetType(colores), CType(12, Byte)) Then
```

Queremos saber si existe el color número 12. Supón que en el programa necesitamos un elemento de la enumeración, normalmente no tendremos problemas pero deberíamos controlar los errores para que no termine de forma anómala o simplemente no funcione bien nuestro programa.

Utilizaremos entonces las herramientas que nos proporciona VB.NET para manejar las enumeraciones. El ejemplo de antes devolverá un valor falso, ya que 12 no es miembro de la enumeración colores. En el primer caso usamos la enumeración colores y usamos el método **GetType** para saber de qué tipo de datos es esa enumeración.

La función **IsDefined** espera dos parámetros, el primero es el tipo de una enumeración, todos los objetos de .NET Framework tienen el método **GetType** que indica ese tipo de datos. En el segundo parámetro, hay que indicar un valor que será el que se compruebe si está o no definido en dicha enumeración, si ese valor es uno de los incluidos en la enumeración indicada por **GetType**, la función devolverá **True**, en caso de que no sea así, devolverá un valor **False**.

Además de **IsDefined**, la clase **Enum** tiene otros métodos que pueden ser útiles:

- **GetName**, indica el nombre con el que se ha declarado el miembro de la enumeración, por ejemplo, esta llamada devolvería "azul":

```
Dim nombreColor As String
nombreColor = System.Enum.GetName(unColor.GetType, 1)
Console.WriteLine(nombreColor)
```

En caso de que el valor indicado no pertenezca a la enumeración, devolverá una cadena vacía.

- **GetNames**, devuelve un array (recuerda las matrices) de tipo String con los nombres de todos los miembros de la enumeración.

```
Dim a() As String = System.Enum.GetNames(unColor.GetType)
Dim i As Integer
For i = 0 To Ubound(a)
    Console.WriteLine("el valor " & i & " es: " & a(i))
Next
```

Por un lado definimos una matriz a la que no le ponemos un tamaño fijo y le asignamos en la propia declaración un valor y ese valor es la lista de la enumeración definida. Así que hemos hecho tres cosas en una sólo línea. Luego hacemos un bucle desde el primer elemento hasta el último (o hasta a.length-1)

- **GetValues**, devuelve un array con los valores de los miembros de la enumeración. El tipo devuelto es del tipo Array, que en realidad no es un array (o matriz) de un tipo de datos específico, sino más bien es el tipo de datos en el que se basan los arrays o matrices.

```
Dim a As Array = System.Enum.GetValues(unColor.GetType)
For i = 0 To Ubound(a)
    Console.WriteLine("el valor " & i & " es: " & a.getvalue(i))
Next
```

Por último vamos a ver un método que, casi con toda seguridad veremos en más de una ocasión:

- **Parse**, devuelve un valor de tipo Object con el valor de la representación de la cadena indicada en el segundo parámetro. Esa cadena puede ser un valor numérico o una cadena que representa a un miembro de la enumeración.

```
System.Enum.Parse(unColor.GetType, "1")
System.Enum.Parse(unColor.GetType, "azul")
```

Hay más métodos, pero estos que son los más interesantes.

Ahora veamos más detalles del método *Parse*. Este método se utiliza para convertir una cadena en un valor numérico, el tipo de número devuelto dependerá del tipo desde el que hemos usado ese método, por ejemplo, si hacemos lo siguiente:

```
Dim s As String = "123"
```

```
Dim i As Integer = Integer.Parse(s)
```

El valor asignado a la variable numérica **i**, sería el valor 123 que es un número entero válido. Pero si hacemos esto otro:

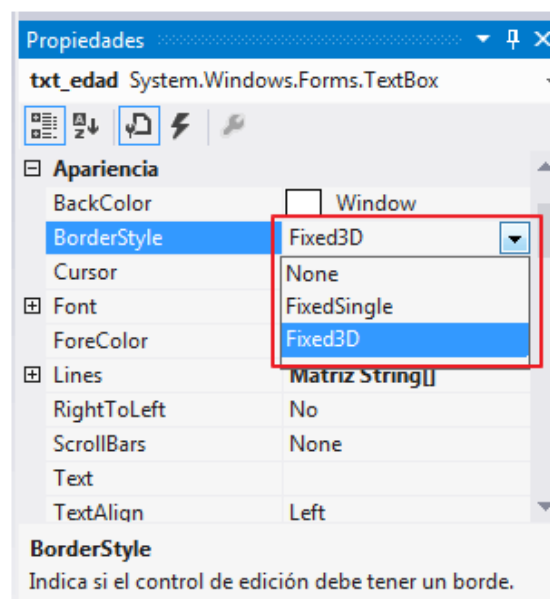
```
Dim b As Byte = Byte.Parse(s)
```

También se asignaría el valor 123 a la variable **b**, que es de tipo **Byte**, pero el número 123 ya no es un número entero, sino del tipo **byte**. De acuerdo, pero si el valor guardado en la variable **s** no estuviese dentro del "rango" de valores aceptados por el tipo **Byte**, esto produciría una excepción (o error).

```
s = "129.33"  
i = Integer.Parse(s)
```

En este caso, el error se produce porque 129.33 no es un número entero válido, por tanto, cuando usemos **Parse**, o cualquiera de las funciones de conversión, tendremos que tener cuidado de que el valor sea el correcto para el tipo al que queramos asignar el valor.

Como te habrás dado cuenta prácticamente todos los objetos de .NET tienen propiedades basadas en enumeraciones. Luego veremos más ejemplos, pero como recordatorio, la enumeración de estilos del contorno de un cuadro de texto:



Solo tiene tres valores pero nos sirve para entender que muchas de las propiedades tienen un valor que pertenece a una lista enumerada. Recuerda esto para más adelante!

3. Tratamiento de errores

Cuando en el código de nuestra aplicación se produce un error sintáctico, es decir, producido porque hayamos escrito mal alguna instrucción de Visual Basic .NET, será el propio entorno de

desarrollo el que nos avise de que hay algo que no es correcto. A este tipo de errores se les suele llamar **errores sintácticos** o más comúnmente **errores en tiempo de diseño**. Pero si lo que ocurre es que hemos asignado un valor erróneo a una variable o hemos realizado una división por cero o estamos intentando acceder a un archivo que no existe, entonces, se producirá un **error en tiempo de ejecución**, es decir, sólo sabremos que hay algo mal cuando el ejecutable esté funcionando.

Hay una ley de Murphy que dice que "cuando algo funciona a la primera es que está mal hecho". Esto nos dice que es normal tener errores en nuestros programas. La experiencia irá reduciendo los errores triviales pero cualquier programa tendrá su parte de errores que debemos controlar.

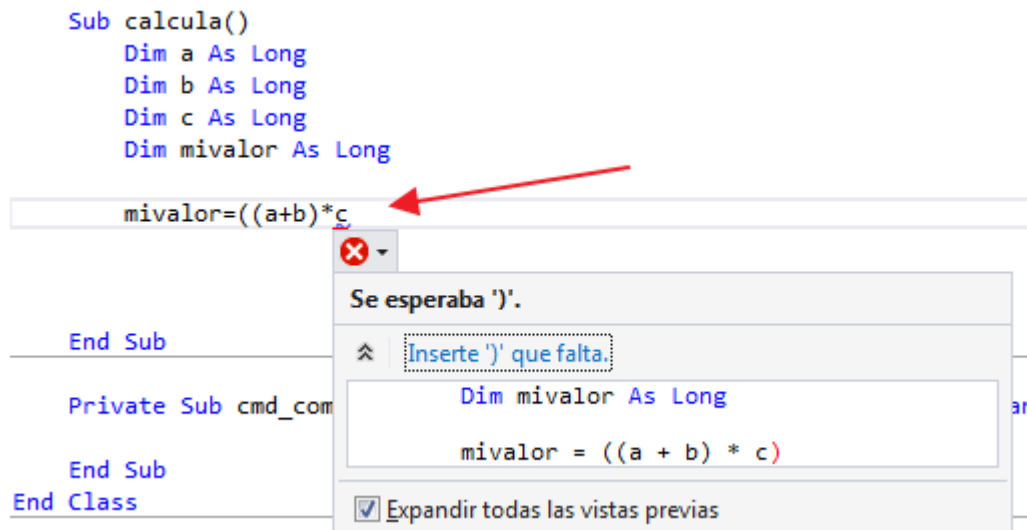
En nuestros programas realizaremos las interfaces y controles para evitar los errores en tiempo de ejecución que son los más afectan a la imagen de calidad del programa.

Durante el código pondremos los controles necesarios para controlar los errores pero debemos trabajar a fondo con el diseño del programa:

- Realizando interfaces de usuario completas. Si hay un campo de tipo fecha utilizaremos los controles que sólo permiten introducir una fecha. Si hay un campo numérico que está en un pequeño rango podemos poner una lista desplegable. Estas dos acciones nos aseguran que los datos introducidos son una fecha y un número por lo que ya no debemos controlarlos en el código.
- Validar en el código los resultados esperados. Es decir, cuando el usuario pulse "Aceptar" para empezar el proceso, primero validaremos si los datos son correctos: si había que escribir un texto en un cuadro lo comprobaremos con la función "Len". Si un valor numérico debe estar entre unos valores lo comprobaremos con If..then. Así sólo cuando todos los elementos de la interfaz estén correctos realizaremos el proceso.

Los errores que nos podemos encontrar son:

- Errores de escritura. Son errores sintácticos al escribir el código. Son los más fáciles de localizar porque el IDE nos avisará de dónde está el error: falta un paréntesis, asignación,... El IDE marcará la zona del error y mostrará un mensaje descriptivo:



- **Errores en tiempo de ejecución.** El ejemplo más claro es como hemos comentado antes la mala asignación de una variable. Por ejemplo, en un cuadro de texto un usuario debe escribir su edad y en lugar de escribir su edad escribe su nombre con lo que en lugar de asignar un valor numérico a la variable edad el programa intenta asignarle una cadena de caracteres. Esto producirá un error en tiempo de ejecución, por ejemplo:

```
Sub calcula()  
    Dim edad As Long  
  
    edad = "manuel"  
  
End Sub
```

- **Errores lógicos.** Los anteriores son difíciles de seguir porque puede que no lleguen a darse o al cabo del tiempo. En el caso de los errores lógicos son de difícil captura porque el fallo está en el planteamiento de la solución. Un algoritmo puede estar prácticamente bien escrito pero fallar con una determinada entrada de datos. Por ejemplo, no entra en un bucle porque una variable no toma nunca un determinado valor. En estos casos se hace obligatorio el uso de la depuración y utilización de puntos de ruptura para realizar un minucioso seguimiento a la ejecución.

3.1. Errores y Excepciones

Dentro de .NET encontramos dos áreas: los errores y las excepciones. Dentro del tratamiento de errores cada uno tiene su proceso:

- **Excepción.** Es un objeto generado por un error que contiene información sobre las características del error producido.
- **Error.** Evento que se produce durante el funcionamiento de un programa provocando una interrupción del programa y la generación de una excepción.

El control de errores se realiza de la siguiente forma:

Control estructurado de errores

En este tipo de tratamiento cada vez que se produce un error se genera un objeto de clase "Excepcion" conteniendo la información del error producido. VB.NET nos ofrece una estructura de control para capturar estos objetos: la estructura Try Catch Finally.

La forma de usar esta estructura sería así:

Try

' el código que puede producir error

Catch [tipo de error a capturar]

' código cuando se produzca un error

Finally

' código se produzca o no un error

End Try

En el bloque Try pondremos el código que puede que produzca un error. Es decir, una operación compleja que requiera que se haga un tratamiento de errores para evitar que se produzca un error en tiempo de ejecución.

Los bloques Catch y Finally no son obligatorios, pero al menos hay que usar uno de ellos, es decir, o usamos Catch o usamos Finally o, usamos los dos, pero como mínimo uno de ellos. Una vez aclarado que, además de Try, debemos usar o Catch o Finally, veamos para qué sirve cada uno de ellos:

En caso de utilizar **Catch**, el código se ejecutará si se produce un error. Es la parte que "capturará" el error. Después de Catch podemos indicar el tipo de error que queremos capturar, incluso podemos usar más de un bloque Catch, si es que nuestra intención es detectar diferentes tipos de errores.

En el caso de que sólo utilicemos **Finally**, tendremos que tener en cuenta que si se produce un error, el programa se detendrá de la misma forma que si no hubiésemos usado la detección de errores, por tanto, aunque usemos Finally (y no estemos obligados a usar Catch), es más que recomendable que siempre usemos una cláusula Catch, aunque en ese bloque no hagamos nada, pero aunque no "tratemos" correctamente el error, al menos no se detendrá porque se haya producido el error.

Si decidimos "prevenir" que se produzca un error, pero simplemente queremos que el programa continúe su ejecución, podemos usar un bloque Catch que esté vacío, con lo cual el error simplemente se ignorará. Si has usado o has leído sobre cómo funciona *On Error Resume Next*,

pensarás que esto es algo parecido. Si se produce un error, se ignora y se continúa el programa como si nada. Pero no te confundas que aunque lo parezca... no es igual.

Si tenemos el siguiente código, se producirá una excepción (o error), ya que al dividir *i* por *j*, se producirá un error de división por cero.

```
Dim i, j As Integer
Try
    i = 10
    j = 0
    i = i \ j
Catch
    ' nada que hacer si se produce un error
End Try
    ' continúa después del bloque de detección de errores
```

Pero cuando se produzca ese error, no se ejecutará ningún código de "tratamiento" de errores, ya que dentro del bloque Catch no hay ningún código.

Observemos el siguiente código:

```
Dim i, j As Integer
Try
    i = 10
    j = 0
    i = i \ j
    Console.WriteLine("el nuevo valor de i es: {0}", i)
Catch
    ' nada que hacer si se produce un error
End Try
    ' continúa después del bloque de detección de errores
Console.WriteLine("después del bloque de detección de errores")
```

Aquí tenemos prácticamente el mismo código que antes, con la diferencia de que tenemos dos "instrucciones" nuevas, una se ejecuta después de la línea que "sabemos" que produce el error y la otra después del bloque Try... Catch. Es una prueba, sabemos que la que produce el error es la línea "*i* = *i* \ *j*", pero el que sepamos qué línea es la que produce el error no es lo habitual, y si en este caso lo sabemos con total certeza, es sólo es para que comprendamos mejor todo esto.

Cuando se produce el error, Visual Basic .NET, o mejor dicho, el runtime del .NET Framework, deja de ejecutar las líneas de código que hay en el bloque Try, (las que hay después de que se produzca la excepción). Después continúa por el código del bloque Catch. Como en este caso no hay nada, busca un bloque Finally. Si existe ejecutará ese código, pero como no hay bloque Finally, continúa por lo que haya después de End Try.

Vamos a repetirlo una vez más. Mostraremos lo que ocurriría en un bloque Try... Catch si se produce o no un error:

Si se produce una excepción o error en un bloque Try, el código que siga a la línea que ha producido el error, deja de ejecutarse, para pasar a ejecutar el código que haya en el bloque Catch, se seguiría (si lo hubiera) por el bloque Finally y, por último, con lo que haya después de End Try.

Si no se produce ningún error, se continuaría con todo el código que haya en el bloque Try y después se seguiría, (si lo hubiera), con el bloque Finally y, por último, con lo que haya después de End Try.

Detalles de la clase Exception

Aunque luego la veremos con detalle, en este momento tenemos que saber más sobre esta clase.

Para facilitar el uso del control estructurado de excepciones, Visual Basic ofrece la posibilidad de separar el código estándar del código de control de excepciones. El código de control de excepciones obtiene acceso a una instancia de la clase **Exception**, que permite recuperar la información de cualquier excepción que se detecte.

Cada vez que se inicia una excepción, se establece el objeto **Err** global y se crea una nueva instancia de la clase **Exception**.

Las propiedades de la clase **Exception** ayudan a identificar la ubicación en el código, el tipo y la causa de las excepciones. Por ejemplo, la propiedad **StackTrace** muestra una lista de los métodos que fueron invocados antes de que se produjese la excepción, lo que ayuda a detectar el lugar del código en el que ocurrió el error. La propiedad **Message** devuelve un mensaje de texto que describe el error; se puede modificar este mensaje para que sea más fácil de entender. Si no especifica una cadena de texto para el mensaje de error, se utilizará el valor predeterminado. **HelpLink** obtiene o establece un vínculo a un archivo de ayuda asociado. **Source** obtiene o establece una cadena que contiene el nombre del objeto causante del error o el nombre del ensamblado en el que se originó la excepción.

Por tanto, la **clase Exception** representa los errores que se producen durante la ejecución de una aplicación. Cuando se produce un error, el sistema o la aplicación que se está ejecutando en ese momento informa del mismo iniciando una excepción que contiene información sobre el error. Una vez que se inicia una excepción, la aplicación o el controlador de excepciones predeterminado controlan dicha excepción.

Las propiedades de esta clase son:

Propiedad	Descripción
HelpLink	Obtiene o establece un vínculo al archivo de ayuda asociado a esta excepción.

InnerException	Obtiene la instancia de Exception que causó la excepción actual.
Message	Obtiene un mensaje que describe la excepción actual.
Source	Devuelve o establece el nombre de la aplicación o del objeto que generó el error
StackTrace	Obtiene una representación de cadena de los marcos de la pila de llamadas correspondiente al momento en que se inició la excepción actual.
TargetSize	Obtiene el método que inicia la excepción actual.

Otro ejemplo

Veamos un ejemplo en el que abrimos un archivo (o fichero) de texto y en el que vamos a escribir el contenido de una o varias variables. Si no se produce error, todo irá bien, pero si se produce un error justamente cuando vamos a abrir el archivo, (por ejemplo, porque el disco esté lleno o no tengamos acceso de escritura o cualquier otra cosa), el archivo no estará abierto y, por tanto, lo que guardemos en él, realmente no se guardará:

```

Abrir el archivo
    Guardar el contenido de la variable
    repetir la línea anterior mientras haya algo que guardar
Cerrar el archivo
Avisar al usuario de que todo ha ido bien, que ya se puede ir
a su casa a descansar porque todo se ha guardado.

```

Si usamos Try... Catch, todo el código lo escribiríamos dentro del bloque Try y si se produce un error, avisaríamos al usuario de que se ha olvidado de poner el USB, (por ejemplo), y le podríamos dar la oportunidad de volver a intentarlo.

Sabemos que el bloque Try/Catch sirve para detectar errores, incluso para detectar distintos tipos de errores, con idea de que el "runtime" de .NET Framework (CLR), pueda ejecutar el que convenga según el error que se produzca.

Esto es así, porque es posible que en un bloque Try se produzcan errores de diferente tipo y si tenemos la "previsión" de que se puede producir algún que otro error, puede que queramos tener la certeza de que estamos detectando distintas posibilidades. Por ejemplo, en el código anterior, es posible que el error se produzca porque el disco está lleno, porque no tenemos acceso de escritura, porque no haya usb en el puerto, etc. Y podría sernos interesante dar un aviso correcto al usuario de nuestra aplicación, según el tipo de error que se produzca.

Cuando queremos hacerlo de esta forma, lo más lógico es que usemos un Catch para cada uno de los errores que queremos interceptar, y lo haríamos de la siguiente forma:

```
Dim i, j As Integer
Dim s As String
'
Try
    Console.Write("Escribe un número (y pulsa Intro) ")
    s = Console.ReadLine
    i = CInt(s)
    Console.Write("Escribe otro número ")
    s = Console.ReadLine
    j = CInt(s)
    '
    Console.WriteLine("El resultado de dividir {0} por {1} es {2}",
i, j, i \ j)
    '
Catch ex As DivideByZeroException
    Console.WriteLine("ERROR: división por cero")
Catch ex As OverflowException
    Console.WriteLine("ERROR: de desbordamiento (número demasiado
grande)")
Catch ex As Exception
    Console.WriteLine("Se ha producido el error: {0}", ex.Message)
End Try
'
Console.ReadLine()
```

Aquí estamos detectando tres tipos de errores:

- El primero si se produce una división por cero.
- El segundo si se produce un desbordamiento, el número introducido es más grande de lo esperado.
- Y por último, un tratamiento "genérico" de errores, el cual interceptará cualquier error que no sea uno de los dos anteriores.

Si usamos esta forma de detectar varios errores, debes tener cuidado de poner el tipo genérico al final, (o el que no tenga ningún tipo de "error a capturar" después de Catch). El CLR siempre evalúa los tipos de errores a detectar empezando por el primer Catch. Si no coincide con el error producido, comprueba el siguiente y así hasta que llegue a uno que sea adecuado al error producido. Si da la casualidad de que el primer Catch es de tipo genérico, el resto no se comprobará, ya que ese tipo es adecuado al error que se produzca, por la sencilla razón de que **Exception** es el tipo de error más genérico que puede haber, por tanto, se adecúa a cualquier error.

Nota: Realmente el tipo Exception es la clase de la que se derivan (o en la que se basan) todas las clases que manejan algún tipo de excepción o error. Si nos fijamos, veremos que todos los tipos de excepciones que podemos usar con Catch, terminan con la palabra Exception. Esto, además de ser una "norma" o recomendación, nos sirve para saber que el objeto es válido para su

uso con Catch. Esto lo deberíamos tener en cuenta cuando avancemos en nuestro aprendizaje y sepamos crear nuestras propias excepciones.

Por otro lado, si sólo usamos tipos específicos de excepciones y se produce un error que no es adecuado a los tipos que queremos interceptar, se producirá una excepción "no interceptada" y el programa finalizará.

Para poder comprobarlo, puedes usar el siguiente código y si simplemente pulsas intro, sin escribir nada o escribes algo que no sea un número, se producirá un error que no está detectado por ninguno de los Catch:

```
Dim i, j As Integer
Dim s As String
'
Try
    Console.Write("Escribe un número (y pulsa Intro) ")
    s = Console.ReadLine
    i = CInt(s)
    Console.Write("Escribe otro número ")
    s = Console.ReadLine
    j = CInt(s)
    '
    Console.WriteLine("El resultado de dividir {0} por {1} es {2}", i, j, i \ j)
'
Catch ex As DivideByZeroException
    Console.WriteLine("ERROR: división por cero")
Catch ex As OverflowException
    Console.WriteLine("ERROR: de desbordamiento (número demasiado grande)")
End Try
```

Sabiendo esto, es recomendable que siempre uses un "capturador" genérico de errores, es decir, un bloque Catch con el tipo de excepción genérica: `Catch variable As Exception`.

Después de Catch puedes usar el nombre de variable que quieras, la recomendada es **ex**.

Todavía hay más cosas que contar sobre Try... Catch, como por ejemplo:

- Podemos "lanzar" excepciones, sin necesidad de que se produzca una explícitamente,
- Cómo se comporta Visual Basic .NET cuando ejecutamos un código que produce error y no hay un bloque específico de tratamiento para ese error, pero existe un bloque Try que aunque esté en otra parte del código sigue activo.
- Podemos crear nuestras propios tipos de excepciones, pero eso lo veremos cuando tratemos el tema de las clases un poco más a fondo.

Antes de continuar vamos a ver con más detalle la clase Exception

3.2. La clase Exception

Sabemos entonces que cuando se produce un error el entorno de ejecución genera una excepción con la información del error. Para facilitar la manipulación de estas excepciones creamos un "control de excepciones" que obtiene el valor de la clase Exception o alguna de sus derivadas de forma que a través de sus miembros podemos saber qué ha pasado. Entre las propiedades y métodos tenemos:

- Message. Contiene la descripción del error
- Source. Indica el objeto o aplicación que produjo el error.
- StackTrace. Ruta o traza del código en la que se produjo el error.
- ToString(). Devuelve una cadena con información detallada del error. Dada la información que incorpora en muchas ocasiones será el método que recomendemos para obtener los datos de la excepción.

Un ejemplo para obtener estos datos puede ser:

```
' ....
Try
' ....
' ....
Catch oExcep As Exception
' si se produce un error, se crea un objeto excepción
' que capturamos volcándolo a un identificador
' de tipo Exception

    Console.WriteLine("Se produjo un error. Información de la
excepción")
    Console.WriteLine("=====
=====")
    Console.WriteLine("Message: {0}", oExcep.Message)
    Console.WriteLine()
    Console.WriteLine("Source: {0}", oExcep.Source)
    Console.WriteLine()
    Console.WriteLine("StackTrace: {0}", oExcep.StackTrace)
    Console.WriteLine()
    Console.WriteLine(oExcep.ToString())

Finally
' ....
' ....
End Try
' ....
```

Producir excepciones

Para lanzar (o crear) excepciones tendremos que usar la instrucción **Throw** seguida de un objeto derivado del tipo Exception. Normalmente se hace de la siguiente forma:

```
Throw New Exception("Esto es un error personalizado")
```

Con este código estamos indicándole a Visual Basic .NET que queremos "lanzar" (Throw) una nueva excepción (New Exception) y que el mensaje que se mostrará, será el que le indiquemos dentro de los paréntesis.

Cuando nosotros lanzamos (o creamos) una excepción, el error lo interceptará un bloque Try que esté activo, si no hay ningún bloque Try activo, será el CLR (runtime de .NET Framework) el que se encargará de interceptar esa excepción, pero deteniendo la ejecución del programa.

Veamos lo que ocurre cuando hay un bloque Try "activo"...

Supongamos que tenemos un bloque Try desde el cual llamamos a algún procedimiento (Sub, Function, etc.), que puede que a su vez llame a otro procedimiento y resulta que en alguno de esos procedimientos se produce una excepción "no controlada", por ejemplo, si tenemos el siguiente código:

```
Sub Main()  
    Dim n, m As Integer  
    '  
    Try  
        n = 10  
        m = 15  
        Dim k As Integer = n + m  
        '  
        ' llamanos a un procedimiento  
        Prueba()  
        '  
        '... más código...  
        '  
    Catch ex As DivideByZeroException  
        Console.WriteLine("ERROR: división por cero")  
    Catch ex As OverflowException  
        Console.WriteLine("ERROR: de desbordamiento (número  
demasiado grande)")  
    Catch ex As InvalidCastException  
        Console.WriteLine("ERROR: lo escrito no es un número.")  
    Catch ex As Exception  
        Console.WriteLine("Se ha producido el error: {0} {1} {2}",  
_ ex.Message, vbCrLf, ex.ToString)  
    End Try  
    '  
    Console.WriteLine("Pulsa Intro para terminar")  
    Console.ReadLine()
```

```
End Sub

Sub Prueba()
    Dim i, j As Integer
    Dim s As String
    Console.Write("Escribe un número (y pulsa Intro) ")
    s = Console.ReadLine
    i = CInt(s)
    Console.Write("Escribe otro número ")
    s = Console.ReadLine
    j = CInt(s)
    Console.WriteLine("El resultado de dividir {0} por {1} es
{2}", i, j, i \ j)
End Sub
```

En el procedimiento **Main** tenemos cierto código, en el que hemos usado un bloque Try... Catch, dentro del bloque Try, además de otras cosas, llamamos al procedimiento **Prueba**, en el cual se piden dos números y se realizan unas operaciones con esos números. Pero en ese procedimiento no tenemos ningún bloque Try que pueda "interceptar" errores.

Tal como vimos anteriormente, si simplemente pulsamos Intro cuando nos pide alguno de esos números o cuando escribimos en el segundo un cero o si alguno de esos dos números que introducimos es más grande que lo que un tipo Integer puede soportar, se producirá un error (o excepción). Pero, resulta que dentro del procedimiento Prueba no tenemos nada que "intercepte" los posibles errores que se puedan producir. Si ese código fuese el único que tuviéramos en el programa y se produjera una excepción, sería el CLR o runtime del .NET Framework el que se encargaría de avisarnos de que algo va mal (deteniendo la ejecución del programa). Pero, cuando se llama al procedimiento Prueba desde Main, hay un bloque Try "activo" y el CLR se da cuenta de ese detalle y, en lugar de detener el programa y mostrar el error, lo que hace es "pasar" esa excepción a dicho bloque Try (porque está activo) y "confiar" que el error sea "tratado" por dicho bloque.

Veamos otro ejemplo para que quede más claro, es parecido al anterior, pero con un "pequeño" detalle que lo diferencia:

```
Sub Main()
    Dim n, m As Integer
    '
    Try
        n = 10
        m = 15
        Dim k As Integer = n + m
        '
        '... más código...
        '
    Catch ex As DivideByZeroException
        Console.WriteLine("ERROR: división por cero")
    End Try
End Sub
```

```
        Catch ex As OverflowException
            Console.WriteLine("ERROR: de desbordamiento (número
demasiado grande)")
        Catch ex As InvalidCastException
            Console.WriteLine("ERROR: lo escrito no es un número.")
        Catch ex As Exception
            Console.WriteLine("Se ha producido el error: {0} {1} {2}",
ex.Message, vbCrLf, ex.ToString)
        End Try
        ' llamamos a un procedimiento
        Prueba()
        '
        Console.WriteLine("Pulsa Intro para terminar")
        Console.ReadLine()
    End Sub

    Sub Prueba()
        Dim i, j As Integer
        Dim s As String
        Console.Write("Escribe un número (y pulsa Intro) ")
        s = Console.ReadLine
        i = CInt(s)
        Console.Write("Escribe otro número ")
        s = Console.ReadLine
        j = CInt(s)
        Console.WriteLine("El resultado de dividir {0} por {1} es
{2}", i, j, i \ j)
    End Sub
```

El código del procedimiento Prueba es exactamente igual que el anterior, pero en el procedimiento Main, la llamada a dicho procedimiento no está "dentro" del bloque Try. Por tanto, si ejecutamos el código cuando solicite alguno de esos dos números y escribamos algo "incorrecto", se producirá, como bien supones, una excepción, pero en esta ocasión no hay ningún bloque Try que intercepte ese error, por tanto, será el CLR el que se encargue de avisarnos de que algo va mal, pero el CLR nos lo dice de una forma "brusca" y poco educada, ya que detiene el programa.

Forzar la salida de un controlador de errores con Exit Try.

A través de esta sentencia de la estructura Try...End Try, obligamos al flujo del programa a salir de la estructura de control de errores, desde cualquier punto de la misma en que nos encontremos. Por ejemplo:

```
Try
    ' comienza el control de errores
```

```
        Console.WriteLine("Introducir un número")
        ' si no es un número Byte se produce error
        byMiNum = Console.ReadLine()
        ' salimos de controlador de errores
        ' sin finalizarlo
        Exit Try

        ' esta línea produce error siempre, ya
        ' que no existe el índice 5 en el array
        aValores(5) = "d"

    Catch oExcep As OverflowException
        ....

    Catch oExcep As Exception
        ....
        ....

End Try
    ....
```

4. Más sobre depuración

Cuando desarrollamos una aplicación el proceso de depuración consume una parte importante del proyecto. Aquí es donde el IDE nos prestará una gran ayuda aportándonos una importante colección de herramientas para ayudarnos en el proceso de localización y corrección de errores. Sabemos que nuestro proyecto siempre tendrá errores que intentaremos corregir, normalmente se consideran de tres tipos:

- **Errores de sintaxis.** Son los más fáciles de encontrar en nuestro entorno de desarrollo. Normalmente se producen con faltas de ortografía de palabras clave de VB o de variables.
- **Errores en tiempo de ejecución.** Se dan cuando, aun siendo correcto el código, se produce un error en la ejecución del programa. Por ejemplo, queremos hacer referencia a un objeto pero todavía no se ha creado. Hasta que no incluyamos algún sistema para interceptar estos errores la aplicación se interrumpirá. Con nuestro IDE serán relativamente fáciles de encontrar.
- **Errores lógicos.** Son los más difíciles de encontrar. Se producen cuando la aplicación tiene un comportamiento inesperado, por ejemplo, en errores lógicos en algoritmos. Aquí también el IDE nos ayudará a realizar una "traza" de estos algoritmos para encontrar el fallo.

Son varias las herramientas disponibles para la depuración:

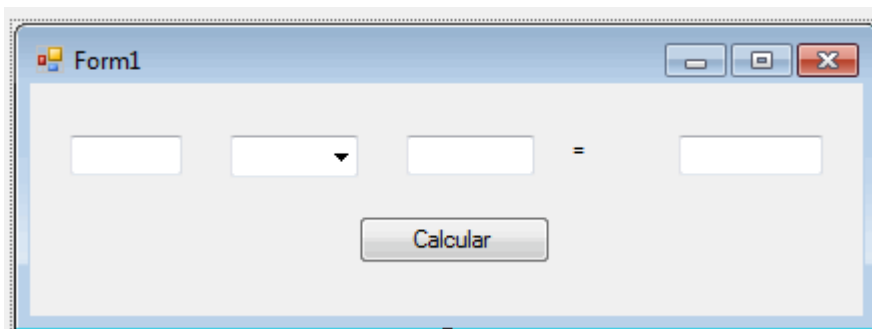
- Inspecciones
- Puntos de interrupción
- Ventana de excepciones
- Compilación condicional
- Trazas
- ...

Vamos realizar un ejemplo para ver todos estos métodos de depuración. Seguiremos estos pasos para preparar un sencillo programa:

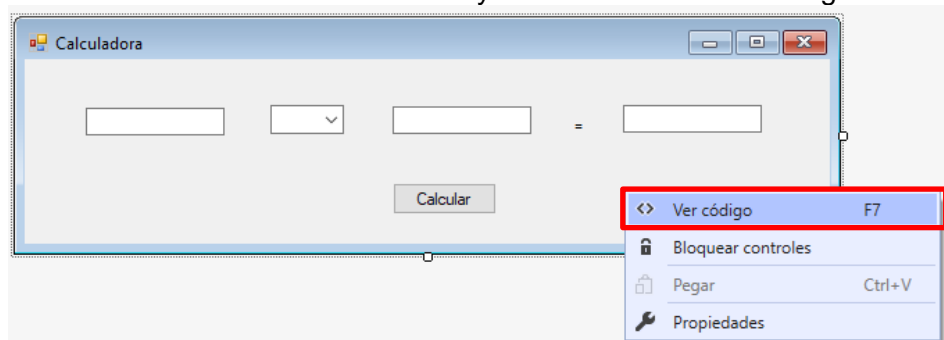
1. Inicia VB.NET
2. Crea un proyecto de Windows llamado *Calculadora*
3. Dibuja los siguientes controles con estos nombres:

Control	Nombre
textbox1	TxtIzquierda
combobox1	CboOperacion
textbox2	TxtDerecha
label1	Label1
textbox3	TxtResultado
button1	BtnCalcular

Y que quede de más o menos de esta forma:



4. Haz clic con el botón derecho en el formulario y seleccionamos "Ver código".



Después de "Public Class Form1" introduce estas dos variables:

```
Public Class Form1
    Private intnumeroizq As Integer
    Private intnumeroder As Integer
```

5. Introducimos el siguiente código en el método Load del formulario. Selecciona (*Form1 Eventos*) en el desplegable de la izquierda y *Load* en el de la derecha. Con esto conseguimos que se ejecuten estas instrucciones cuando se cargue el formulario:

```
Private Sub Form1_Load(sender As Object, e As EventArgs) Handles Me.Load
    'Añadimos las operaciones disponibles al cuadro combinado
    CboOperacion.Items.Add("+")
    CboOperacion.Items.Add("-")
    CboOperacion.Items.Add("*")
    CboOperacion.Items.Add(Chr(247))
End Sub
```

6. Parecido para el botón, lo seleccionamos a la izquierda y a la derecha buscamos el evento Click(). Este código se ejecutará al hacer clic en el botón:

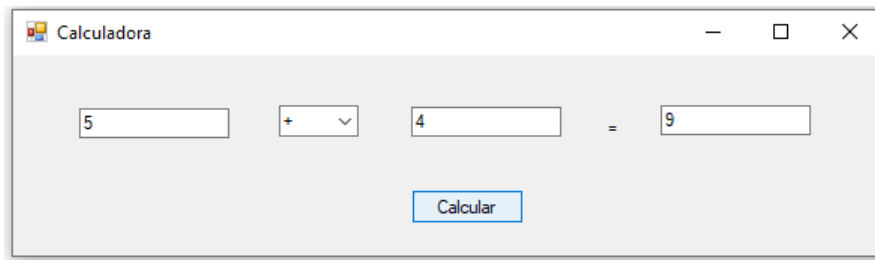
```
Private Sub BtnCalcular_Click(sender As Object, e As EventArgs) Handles BtnCalcular.Click
    intnumeroizq = CType(TxtIzquierda.Text, Integer)
    intnumeroder = CType(TxtDerecha.Text, Integer)
    Call Calcular()
End Sub
```

7. Creamos un procedimiento al final del programa:

```
Sub Calcular()
    Dim tmpresultado As Integer
    'Intentamos hacer la operación
    Try
        Select Case CboOperacion.SelectedItem
            Case "+"
                tmpresultado = intnumeroizq + intnumeroder
            Case "-"
                tmpresultado = intnumeroizq - intnumeroder
            Case "*"
                tmpresultado = intnumeroizq * intnumeroder
            Case Chr(247)
                tmpresultado = CType(intnumeroizq / intnumeroder, Integer)
        End Select
    Catch ex As Exception
        'Capturamos cualquier excepción, por ejemplo división por 0
        tmpresultado = 0
    End Try

    'Mostramos el resultado de la operación
    TxtResultado.Text = CType(tmpresultado, String)
End Sub
```

Y con esto tenemos terminado nuestro programa, que incluso si lo ejecutamos funciona:

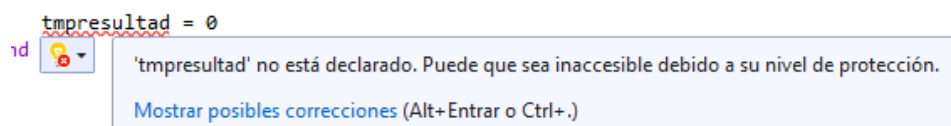


Para empezar con las ayudas, hemos visto como VB pone un color para los comentarios y otro color para las palabras clave del lenguaje. Además, si intentamos escribir una palabra clave: right/rigth veremos que se pone en color azul la correcta.

Cuando escribimos una variable de forma incorrecta nos avisa con un ligero subrayado en azul (como una falta de ortografía en Word). Por ejemplo, si me equivoco al escribir una variable nos mostrará esto:

```
tmpresultad = 0
```

Y si ponemos el ratón encima, nos mostrará qué es lo que pasa:



Toda esta ayuda de momento nos sirve para evitar errores en las variables y en la escritura de las palabras clave del lenguaje de VB.NET

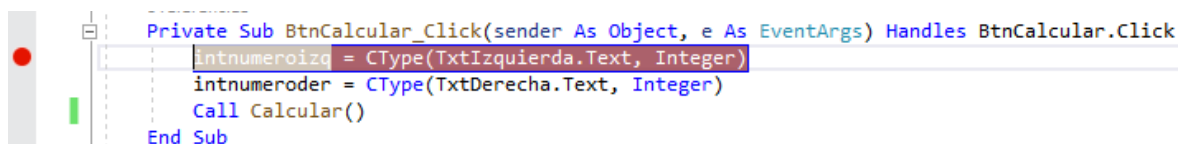
4.1. Menú Depuración

El menú Depuración del IDE nos proporciona un potente grupo de herramientas que serán de gran ayuda para realizar depuraciones en entornos de tiempo de ejecución, es decir, a la vez que se ejecuta el programa. Nos proporcionan una forma de ejecutar línea a línea el programa para controlar su ejecución. Las herramientas son:

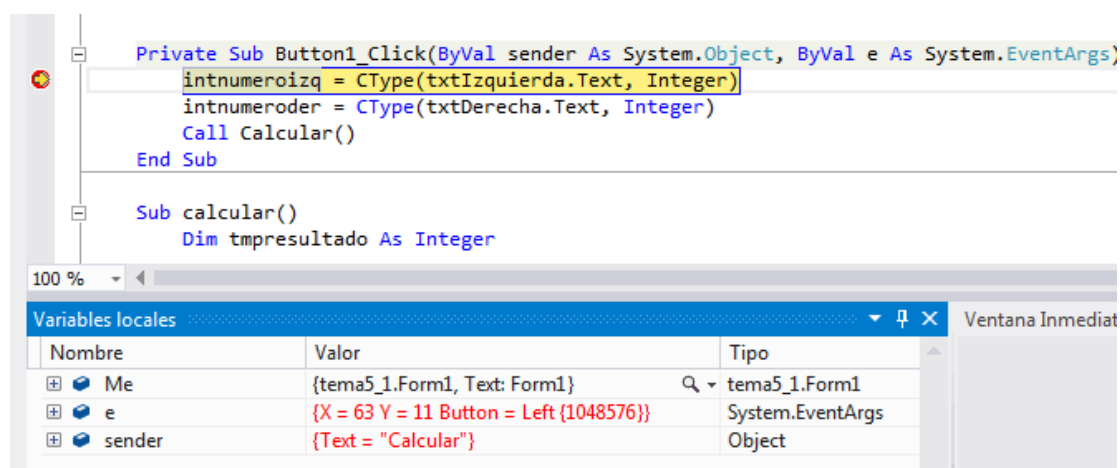
- Paso a paso por instrucciones, F11
- Paso a paso por procedimientos, F10
- Paso a paso para Salir, Mayúsculas + F11
- Ejecutar hasta el cursor, Control + F10

Vamos a ver cómo funciona siguiendo estos pasos:

1. Abrimos la ventana de código y nos situamos una línea debajo del método "BtnCalcular_Click"
2. Coloca un punto de interrupción en esa línea con F9. Un punto de interrupción como vimos en el tema anterior, es una señal para que se detenga el programa y pase a modo de depuración. Los puntos de interrupción se marcan en color granate y ponen un círculo rojo a la izquierda del editor. Pulsando sobre ese punto podremos desactivarlo.

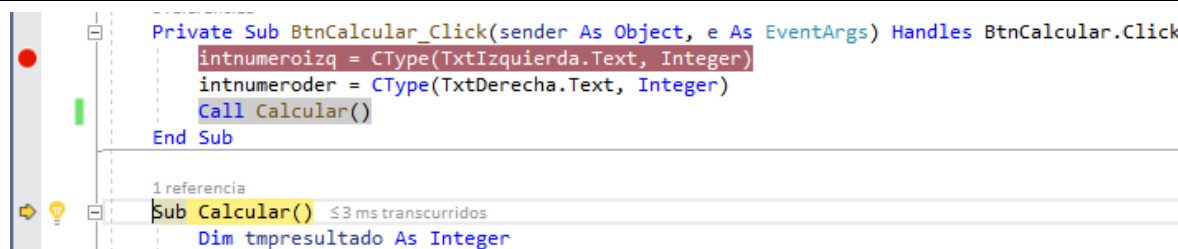


3. Ejecutamos el programa.
4. Introducimos dos valores, la operación de suma y pulsamos el botón de Calcular.
5. Veremos cómo se detiene la ejecución del programa y nos muestra una línea destacada en color amarillo:



La línea amarilla indica que esa es la instrucción que se va a ejecutar cuando se lo indiquemos, es decir, muestra la siguiente línea.

6. Vete al menú Depuración y pulsa "Paso a paso por instrucciones", verás que la ejecución ha pasado a la siguiente línea. Esto es otra gran ayuda porque podemos saber en qué línea se produce el error (si lo hubiera) al ir explorando la ejecución del procedimiento.
7. Continúa pulsando la opción de avanzar con la depuración (tecla F11), verás que cuando llegamos a la instrucción "Calcular" la ejecución salta al procedimiento con este nombre:



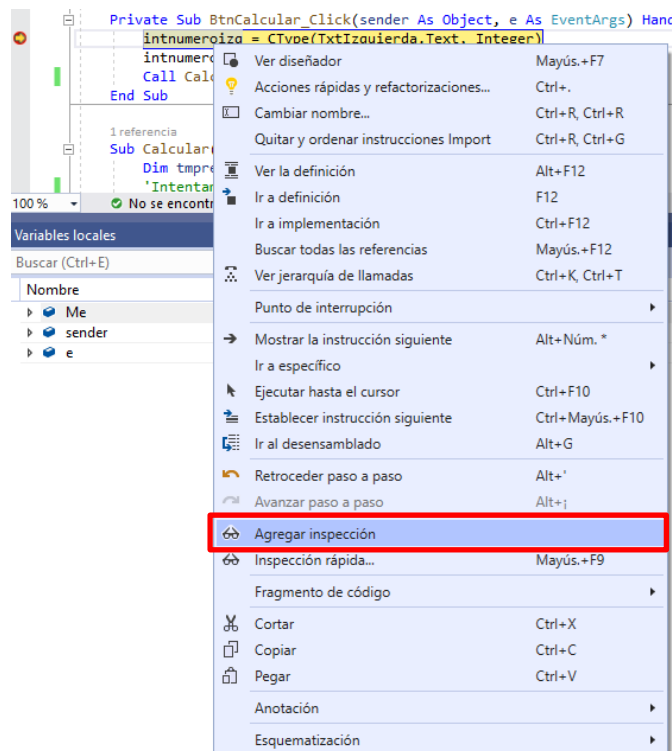
En este punto también nos encontramos con algo muy interesante, vemos cómo la depuración se mete en los procedimientos, cosa que podemos no desear ya que el error está en el principal, así que tenemos una forma de que pase por alto la depuración de los procedimientos y podamos continuar sin tener que meternos dentro. Esto lo realizamos con la segunda opción "paso a paso por procedimientos" (F10). Irá depurando línea a línea pero si hay llamadas a otros procedimientos los ejecuta enteros sin detenerse línea a línea en ellos.

8. Cuando estamos en depuración dentro de una rutina tenemos dos opciones. Podemos ir "paso a paso por instrucciones" para o podemos hacer un "paso a paso para salir" para volver al método que llamó a esa rutina.
9. Si pulsamos "paso a paso para salir" dentro del procedimiento de Calcular, veremos cómo la ejecución vuelve a donde se hizo la llamada: en el evento clic del botón. Se ejecuta todo el código del procedimiento Calcular antes de volver al evento clic del botón.

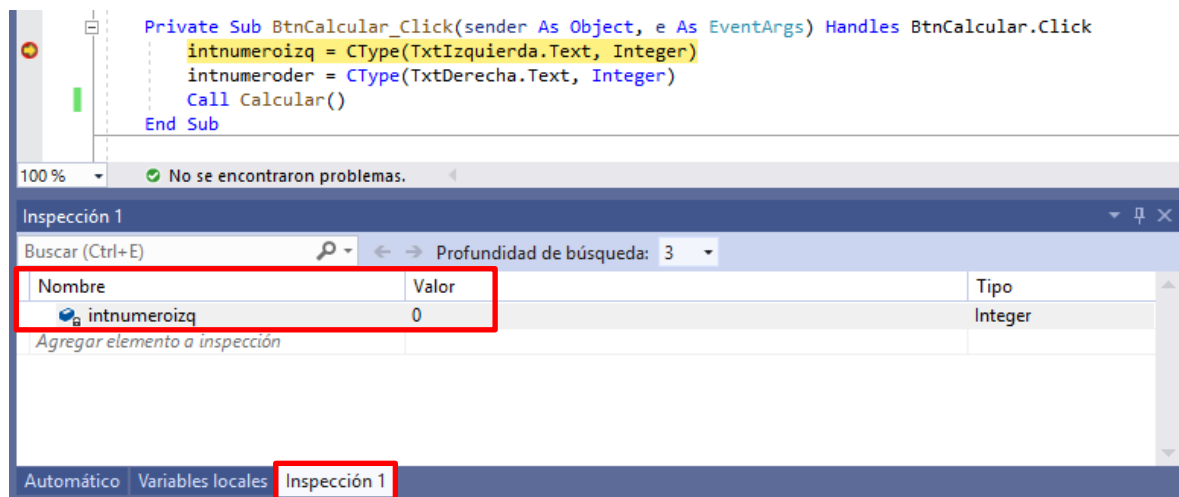
4.2. Inspecciones

Las inspecciones proporcionan un mecanismo donde podemos interactuar con los datos almacenados actualmente en nuestro programa en ejecución. Esto nos permite ver el valor de las variables y los valores de las propiedades de los objetos. Además de poder verlas, también podremos modificar sus valores. Esta opción es también tremendamente útil porque podemos ir viendo lo que sucede con una variable y comprobar los valores que va cogiendo a lo largo del programa. Vamos a practicar con nuestro ejemplo:

1. Para poder ver esta ventana debemos estar en modo de interrupción, es decir, dentro de la depuración. Asegúrate de que tenemos el punto de interrupción en la línea de antes del evento clic del botón.
2. Ejecuta el programa y pulsa en Calcular para pasar a depuración.
3. Sitúa el cursor sobre la variable "intnumeroizq" y selecciona "agregar inspección" con el botón derecho del ratón:



Verás que en la parte inferior del IDE aparece una ventana llamada Inspección.



En esta ventana podremos ir viendo el valor de la variable a lo largo de la ejecución del programa, o en su ámbito, ya que si es local lógicamente no la podremos ver cuando estemos en otro formulario. Si las variables son públicas las podremos ver a lo largo de todo el programa.

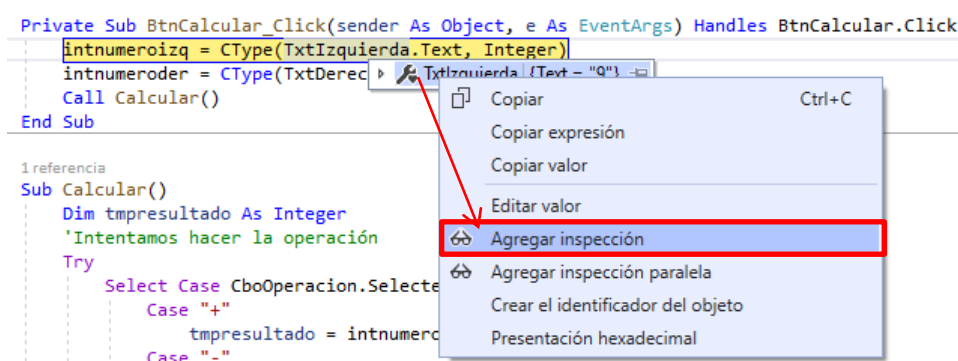
Como hemos comentado antes, además de ver su valor podemos cambiarlo. Simplemente sitúa el cursor en el valor de la variable y escribe un nuevo número, por ejemplo 15:

Nombre	Valor	Tipo
intnumeroizq	15	Integer

Agregar elemento a inspección

Nota: También podemos añadir variables escribiéndolas directamente en la ventana de Inspección. O también marcándola en el código y arrastrándola a esta ventana

También podemos poner objetos. Si por ejemplo añadimos uno de los cuadros de texto tendrá esta forma:



Veremos en la inspección:

Nombre	Valor	Tipo
intnumeroizq	15	Integer
intnumeroder	0	Integer
TxtIzquierda	{Text = "9"}	System.Windows.Forms.TextBox

Como ves, además de aparecer su valor, nos muestra en la tercera columna que pertenece a la clase "System.Windows.Forms.TextBox". Vemos a la izquierda un signo " ▸ ", eso significa que tiene elementos dentro que se pueden explorar, si fuese una matriz podríamos ver el contenido de la matriz, en este caso como es un cuadro de texto, exploraremos todas sus propiedades:

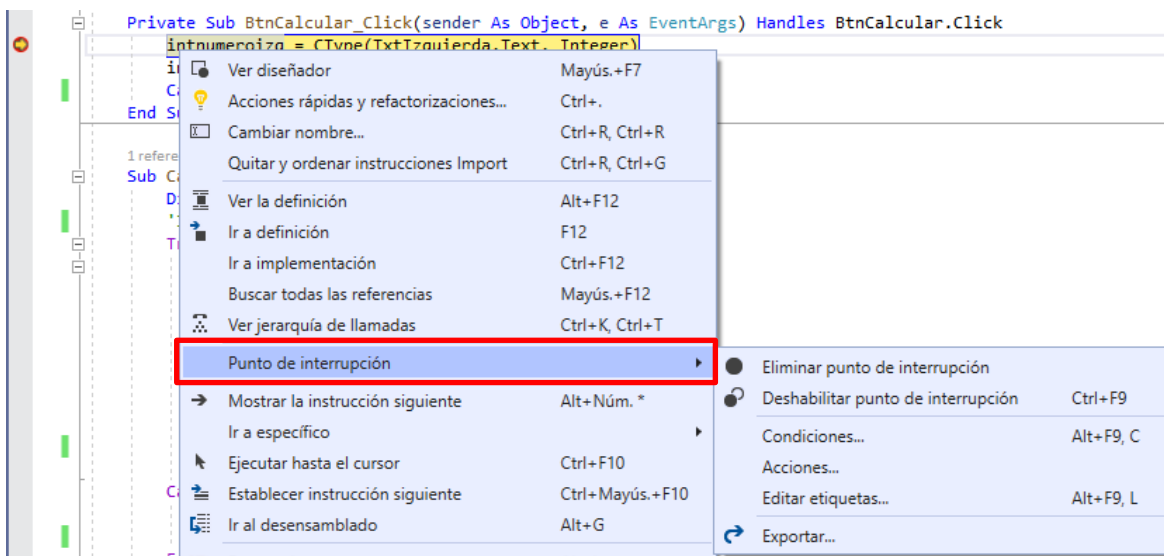
Nombre	Valor	Tipo
TxtIzquierda	{Text = "9"}	System.Windows.Forms.TextBox
AcceptsReturn	False	Boolean
AcceptsTab	False	Boolean
AccessibilityObject	{ControlAccessibleObject: Owner = System.Windows.Forms.TextBox...}	System.Windows.Forms.ControlAccessibleObject
AccessibleDefaultActionDescription	Nothing	String
AccessibleDescription	Nothing	String
AccessibleName	Nothing	String

4.3. Puntos de interrupción

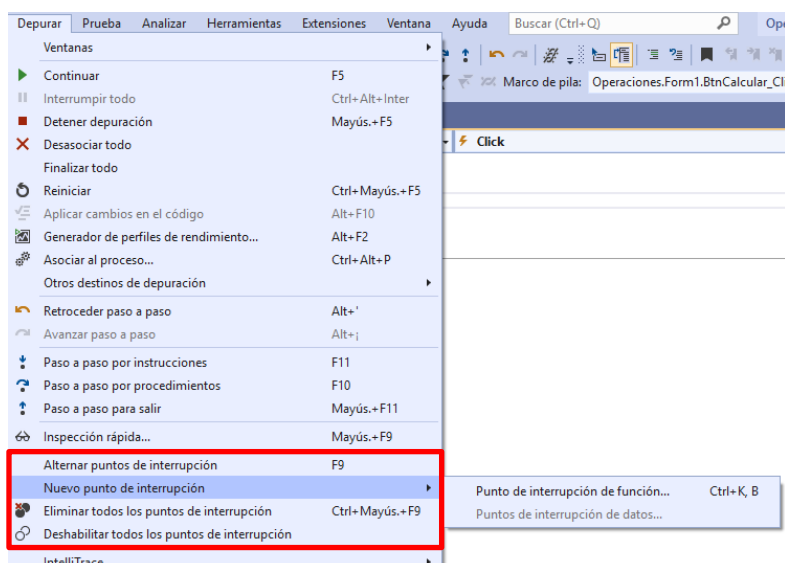
Como ya sabemos, un **punto de interrupción** nos permite detener la ejecución del programa en una línea específica. Esto nos ayudará a evitar depurar el código que ya sabemos que es correcto para pasar a las líneas conflictivas.

Podemos establecer los puntos de interrupción de varias formas:

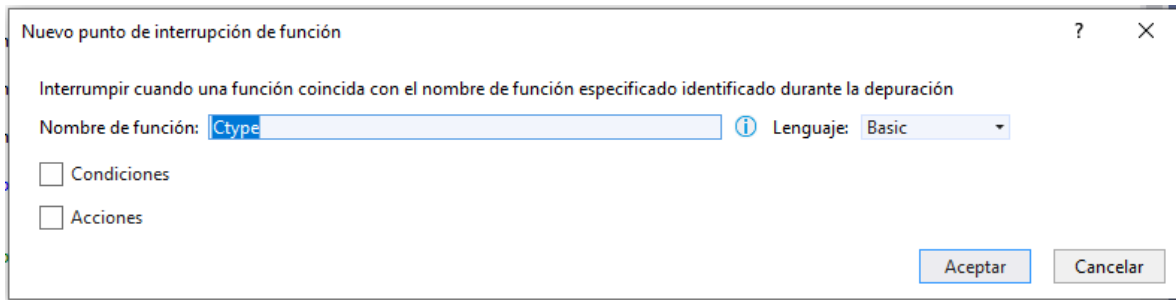
- Haciendo clic en el margen izquierdo de la línea donde queremos establecer el punto de interrupción
- Pulsar F9 en la línea donde queremos establecer el punto de interrupción. Si volvemos a pulsar F9 se desactiva.



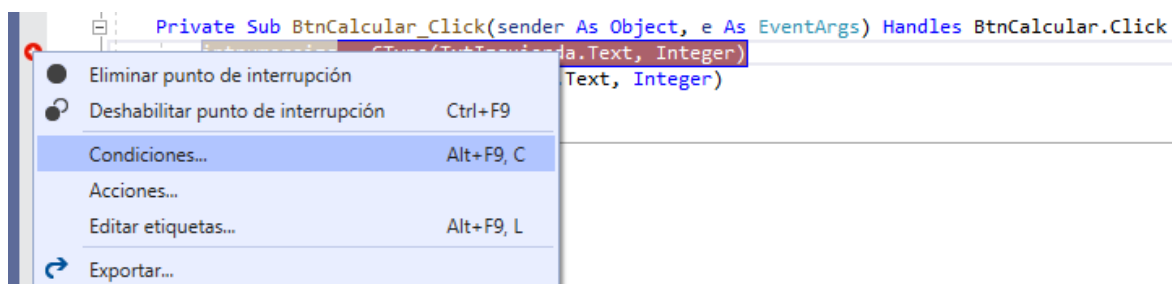
En el menú depurar tenemos las opciones de los puntos para eliminar los puntos, dejarlos temporalmente deshabilitados o alternar entre ellos:



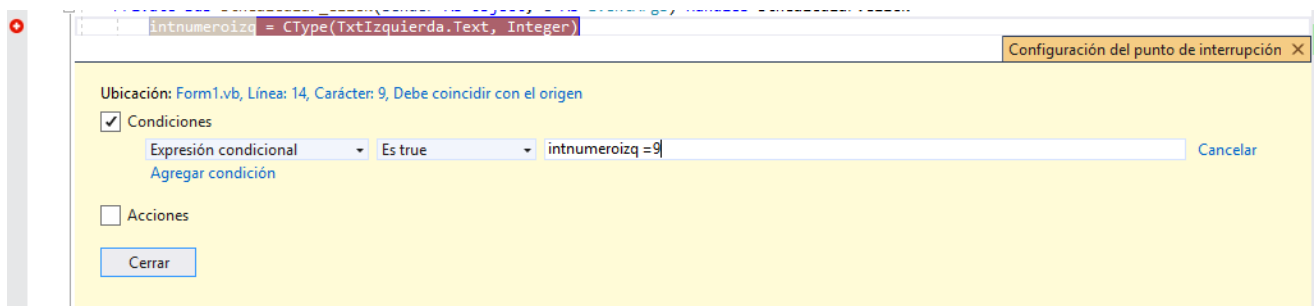
Vemos en la parte derecha la opción de crear un nuevo punto de interrupción cuando ejecute una función. Seleccionamos esta opción de "Punto de interrupción de función...":



En este caso se interrumpiría al ejecutarse esa función, por lo que sería condicional. También podemos poner condicional la parada del programa al modo de depuración:



Si pulsamos con el botón derecho en el punto de interrupción podemos ver las opciones para gestionar cuando queremos esta interrupción. Por ejemplo, veamos "Condiciones":

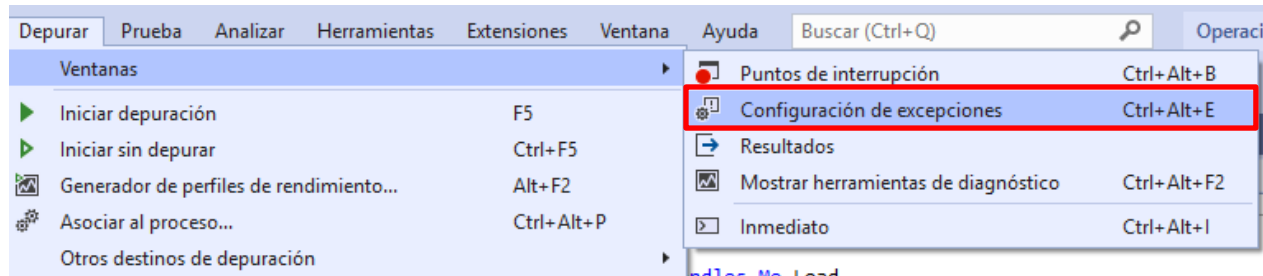


Parece que no tiene importancia pero se utiliza y mucho. Supongamos el caso de un proceso que funciona bien pero al tomar determinado valor no hace lo que esperábamos. Podemos poner un punto de interrupción e indicarle que la condición es que si nombre="jose" se detenga. En ese momento pasamos a modo de depuración y podemos ver el estado del código.

Otro ejemplo, al recorrer un bucle en la iteración número 50 el programa falla. Basta con poner un punto de interrupción condicional con $i=49$ y se detendrá justo antes de la iteración que produce el problema.

4.4. Ventana de Excepciones

Además del ya conocido Try...Catch...Finally para controlar la excepción, podemos configurar cómo debe manejarlas el compilador. Esto lo realizamos desde la ventana de "Configuración de excepciones":

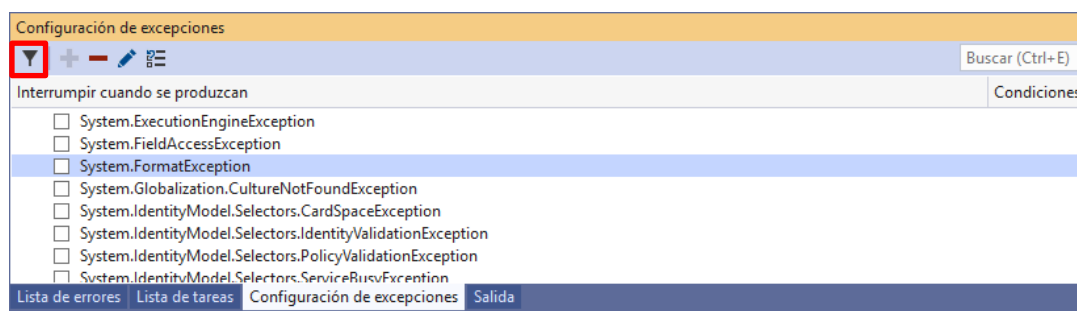


Aparece en la parte inferior y tiene este aspecto:



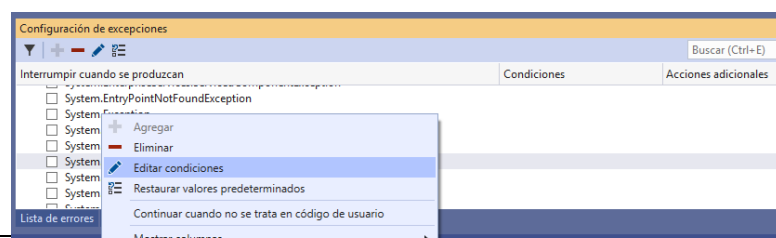
Vamos a explorar las excepciones de esta pantalla:

1. Expandimos la rama principal "Common Language Runtime Exceptions"
2. Desplazamos la lista hasta el elemento System.FormatException



Como vemos en el título pone "Interrumpir cuando se produzcan". Se seleccionan las excepciones deseadas y el programa se interrumpirá cuando se produzcan las mismas.

Además, para agregar excepciones condicionales: Se elige el botón *Editar condiciones* en la ventana Configuración de excepciones, o se hace clic con el botón derecho en la excepción y se elige *Editar condiciones*.



Nota: También podemos añadir nuestras propias clases de excepción personalizadas. La opción de "Restaurar la lista a la configuración predeterminada" puede ser muy útil cuando desarrollemos nuestras propias clases de excepciones.

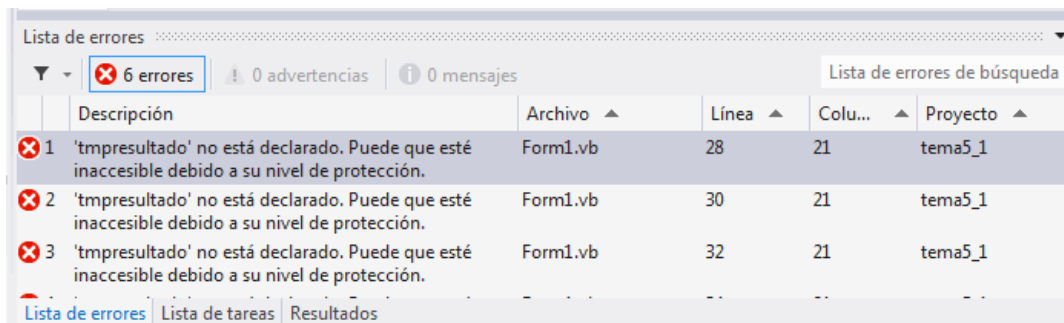
De esta forma le indicamos al compilador que determinada acción, como un error de formato, genere una excepción. No es habitual modificar este comportamiento pero puede ser interesante añadir nuestras propias excepciones.

4.5. Ventanas adicionales

Aunque ya nos ha tocado trabajar con ellas, hagamos un resumen final de algunas de las pantallas de ayuda que nos aporta el IDE.

Ventana de errores

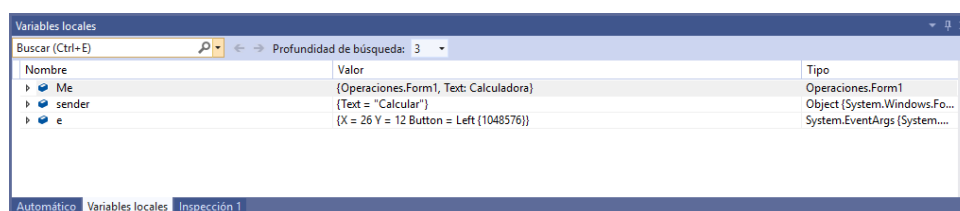
En la parte inferior de la pantalla hemos visto ya varias pantallas de ayuda para nuestro laborioso proceso de depuración. La lista de errores, alertas y mensajes ya la conocemos:



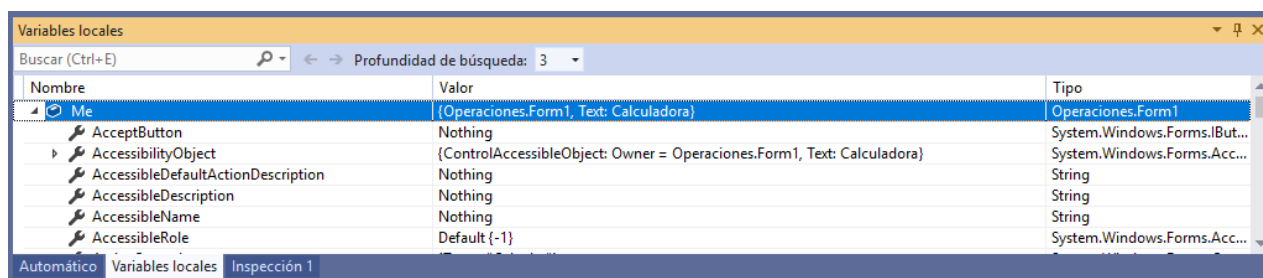
Nos muestra una lista de todos los errores de nuestra aplicación. Si hacemos clic en cada uno de ellos nos podremos mover a las páginas de los errores y tendremos el texto destacado para corregirlos.

Ventana de variables locales

Si estamos depurando la página y estamos detenidos en un punto de interrupción podemos ver debajo la ventana de variables locales:



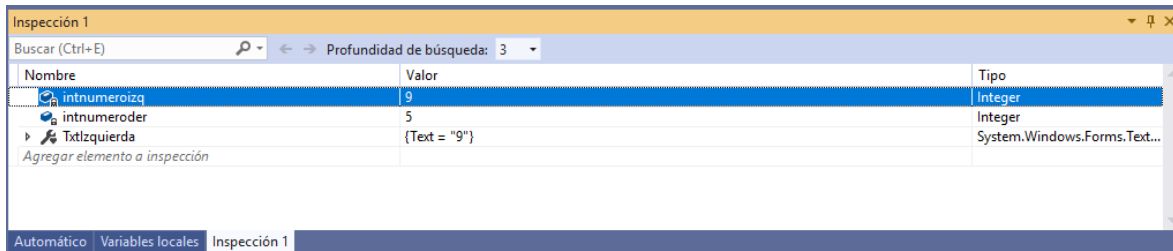
Bajo el nombre "Me" tenemos una referencia al propio formulario, es decir, todos los datos referentes al formulario y que nos servirá de gran ayuda para saber el estado de todas las variables y datos de los controles que componen el formulario:



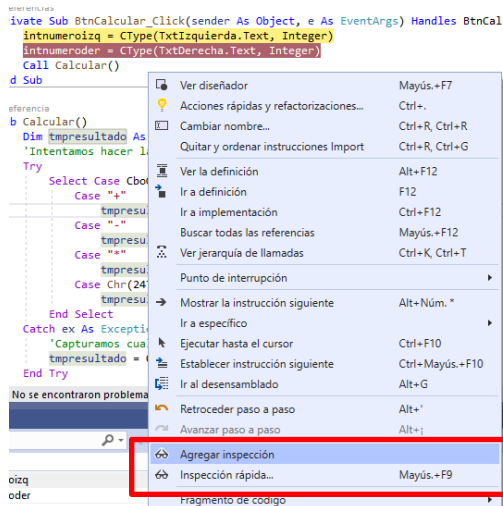
Se muestran las variables locales respecto al contexto o ámbito actual. Normalmente, se trata del procedimiento o de la función que se está ejecutando actualmente. El depurador rellena esta ventana automáticamente

Ventana Inspección

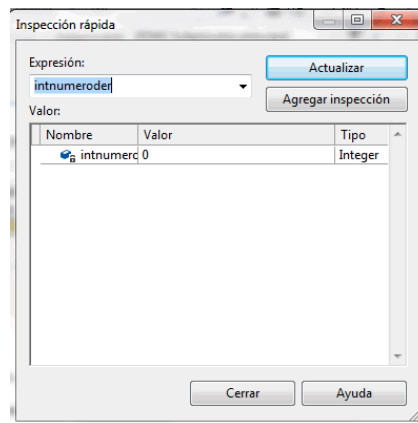
También hicimos anteriormente un ejemplo de añadir una variable a la ventana de inspección. Basta con añadirla con el botón derecho, así podemos ir viendo los valores a medida que se van ejecutando las líneas de código.



Las inspecciones las podemos añadir desde el menú contextual:



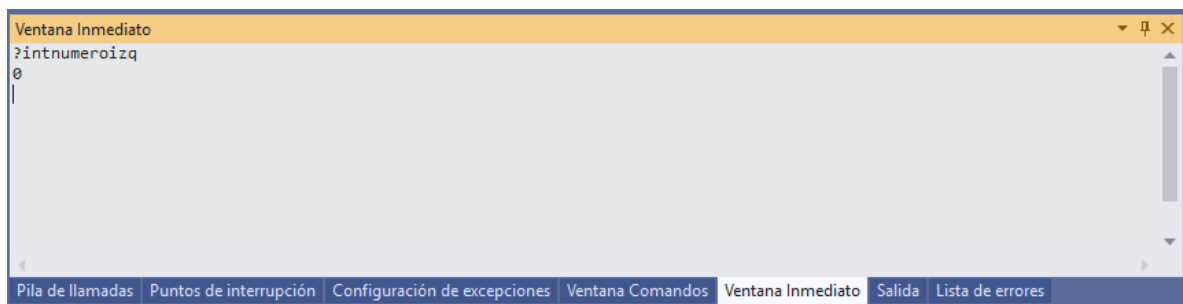
O desde el menú "Depurar":



Podemos añadir también variables, matrices, ... y al ir realizando la depuración realizar un seguimiento de los elementos de la matriz.

Ventana Inmediato

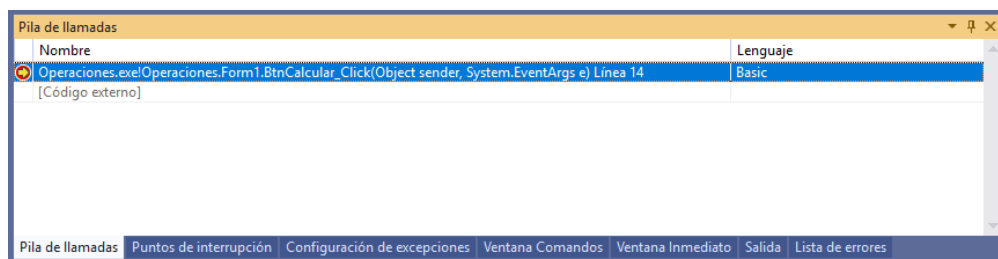
En los paneles de la parte derecha tenemos el de la ventana inmediato:



La ventana "inmediato" nos permite consultar cualquier valor actual escribiendo un "?" + variable. Por ejemplo ?i nos mostrará el valor de la variable i, si existe claro.

Ventana de pila de llamadas

La última ventana, la pila de llamadas:



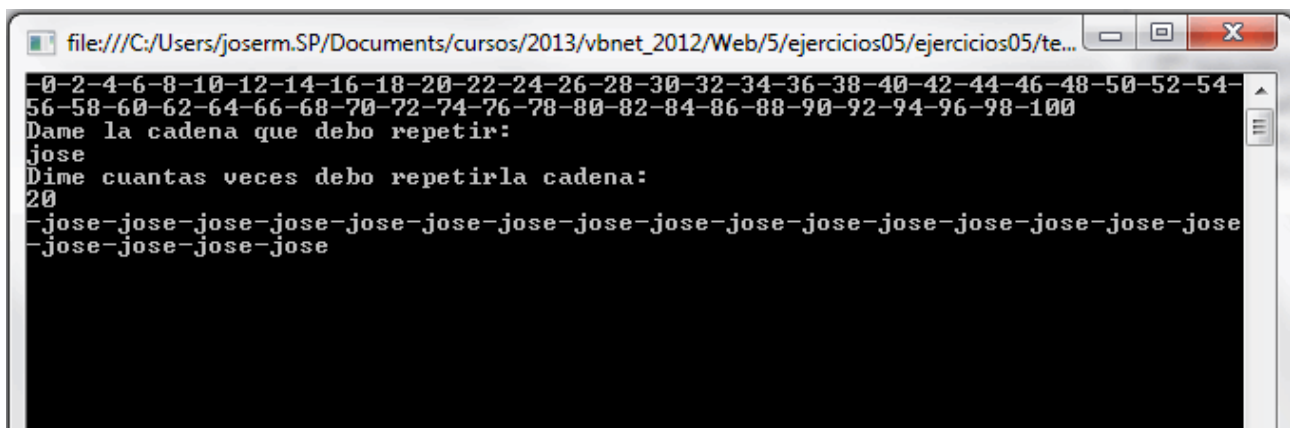
Nos muestra la pila de llamadas por las que va pasando nuestra aplicación. Es información más técnica pero cuando tenemos muchas llamadas entre procedimientos y páginas puede ser interesante para saber el proceso de llamadas que ha estado realizando.

Ejercicios

Ejercicio 1

Realiza un bucle en un programa de consola que escriba los valores pares del 0 al 100. Escríbelos seguidos separados por un guión.

Escribe otro bucle en el mismo programa para que escriba n veces en líneas distintas una cadena que pidas por teclado.



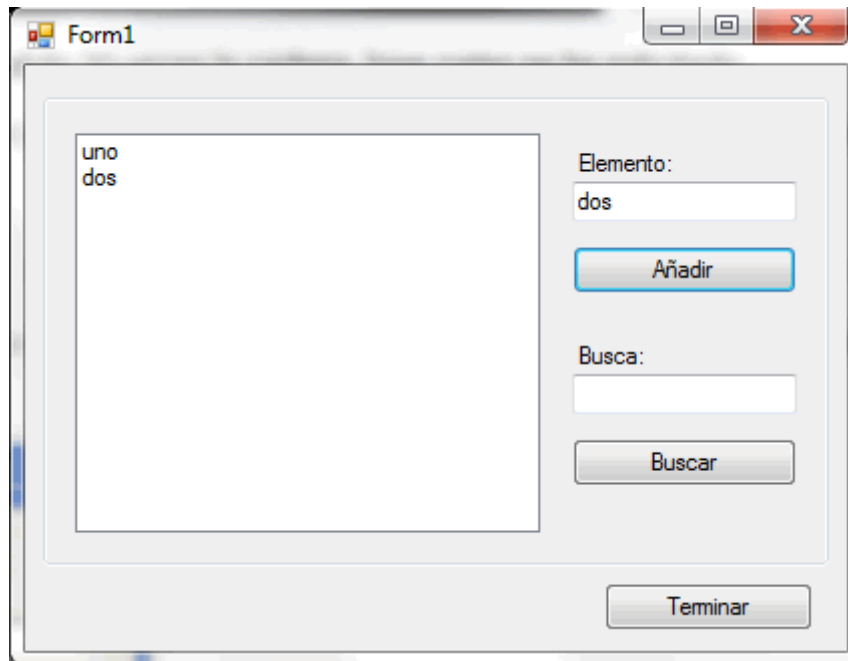
```
file:///C:/Users/josem.SP/Documents/cursos/2013/vbnet_2012/Web/5/ejercicios05/ejercicios05/te...
0-2-4-6-8-10-12-14-16-18-20-22-24-26-28-30-32-34-36-38-40-42-44-46-48-50-52-54-
56-58-60-62-64-66-68-70-72-74-76-78-80-82-84-86-88-90-92-94-96-98-100
Dame la cadena que debo repetir:
jose
Dime cuantas veces debo repetirla cadena:
20
-jose-jose-jose-jose-jose-jose-jose-jose-jose-jose-jose-jose-jose-jose-jose-jose
-jose-jose-jose-jose
```

En la pantalla se han escrito los valores y se ha repetido 30 veces la cadena Jose como se ha solicitado.

Nota: Para ver cuántas veces debe realizarse el bucle puede que necesites convertir la cadena leída de string a entero. Utiliza la función de conversión CType, por ejemplo.

Ejercicio 2

Lee una lista de elementos y los introduces en un cuadro de lista. Luego se introduce un elemento en una casilla y el programa debe decir si está o no en la lista. Comprueba que no se inserten cadenas vacías e implementa el algoritmo del centinela.



Nota: Para saber cuántos elementos hay en una lista utiliza la propiedad Count: `lista.items.count`

Ejercicio 3

Implementa uno de los ejemplos del tema para la interceptación de errores y realiza una traza viendo la bifurcación que se produce.

Ejercicio 4

Realiza la traza del ejercicio 2, añadiendo puntos de interrupción en el algoritmo del centinela añadiendo inspecciones de las variables que utilices.