Colecciones, control de errores y funciones en ASP

Indice

Nº 8 - Colecciones, control de errores y funciones en ASP

1. Colecciones

Temas atrás vimos un poco por encima las colecciones, puesto que se trata de una parte muy importante de nuestra programación tenemos que profundizar un poco mas para manejarlas con soltura. Prácticamente en todos los temas que tratemos de programación nos encontramos con colecciones, es un tema sencillo así que vamos a empezar...

Una colección no es mas que una conjunto de elementos agrupados sin un orden en particular. En programación a un conjunto de elementos le llamamos "set". Cada elemento de la colección debe ser único. Como una colección sólo tiene una dimensión {objeto1,objeto2,objeto3,objeto4} los elementos deben ser únicos ya que sino no sabríamos como hacer referencia a ellos porque serían idénticos.

Cuando una colección los elementos u objetos están almacenados en una secuencia particular se llama lista (obvio ¿no?). En esta lista podemos especificar las posiciones en las que queremos añadir o eliminar elementos y obviamente variará de tamaño. Una lista de autores puede ser...

Autor	Índice
Juan	0
María	1
Antonio	2
Luis	3

Si queremos añadir otro elemento "Luis" se añadirá por defecto al final de la lista pero si queremos que se añada en la segunda posición le diremos que el índice será 1:

Autor	Índice
Juan	0
Luis	1
María	2
Antonio	3
Luis	4

Lógicamente los índices se actualizan automáticamente, esto es igual que cuando vimos las matrices. Un mapa de la colección almacena los objetos en pares índice/valor: uno de los pares es el índice único y el otro el valor. Si tenemos una lista de elementos:

ISBN	Título
102939384	"ASP.NET"
102936754	"VB.NET"

102992134	"Intranets"
102933456	"Cocina Riojana"

Cada libro tiene un código ISBN único así que lo mas conveniente es que cuando hagamos referencia a un libro lo hagamos por este código, así nos aseguramos de que sea único. Otros tipos de listas son:

- Pilas. Los elementos sólo se pueden añadir y eliminar por un sentido. Por ejemplo si tenemos una pila de libros sólo podemos acceder al primero y si añadimos los ponemos encima.
- Colas. En las colas los elementos se introducen por un lado y se extraen por el lado contrario, similar a una cola de personas en una ventanilla, el primero en llegar es el primero en salir.

1.1 Matrices como colecciones

Muchas de las cosas que estamos viendo nos son familiares: índices, elementos, ... como las matrices que vimos capítulos atrás, de hecho una colección es idéntica a una matriz aunque tiene un concepto mas amplio por los tipos de datos que puede contener.

Si pensamos en una matriz de una sola dimensión (vector) con unos datos el mapa es idéntico a los casos anteriores, son pares de índice-valor pero una vez creada la matriz es de tamaño fijo, no podemos variar su tamaño (por lo menos a priori) y no podemos añadir mas elementos. Si eliminamos un elemento en la matriz se queda el hueco vació, con las listas en cambio como hemos visto arriba los elementos u objetos se recolocan pudiendo variar de tamaño dinámicamente.

La clase "System.Array" maneja todo lo referente a las matrices que nos permiten:

- CreateInstance. Inicia una nueva instancia de la clase "System.Array", es decir, crea una matriz
- SetValue. Establece el valor de un elemento en el índice especificado
- GetValue. Devuelve el valor de un elemento en el índice especificado

Para verlo mas claro, vamos con un ejemplo:

```
Dim matriz_animales(4) as string

dim stranimal as string

matriz_animales(0)="León"

matriz_animales(1)="Gato"

matriz_animales(2)="Perro"

matriz_animales(3)="Elefante"

matriz_animales(4)="Gato"

stranimal=matriz_animales(2)
```

Comenzamos "instanciando" o creando la matriz de 5 elementos, recuerda que el primer valor es el 0. Internamente ha creado la matriz que en notación completa sería similar a:

```
dim matriz_animales as Array=Array.CreateInstance(GetTye(string),4)
```

Luego está llamando al método "CreateInstance" para crear una nueva matriz. Los parámetros son el tipo de datos y el número de elementos, el tipo de datos lo podemos obtener de la función "GetType".

Cuando establecemos valores lo que estamos haciendo en realidad es:

```
matriz_animales.SetValue ("León",0)
```

Lo mismo para leer, utilizaremos el método correspondiente con el índice:

```
stranimal=matriz_animales.GetValue (2)
```

Por lo tanto podemos escribir lo anterior de esta forma, así cambiamos de trabajar con matrices algo mucho mas completo utilizando los objetos:

```
dim matriz_animales as Array=Array.CreateInstance(GetTye(string),4)

dim stranimal as string

matriz_animales.SetValue ("León",0)

matriz_animales.SetValue ("Gato",1)

matriz_animales.SetValue ("Perro",2)

matriz_animales.SetValue ("Elefante",3)

matriz_animales.SetValue ("Gato",4)

stranimal=matriz_animales.GetValue (2)
```

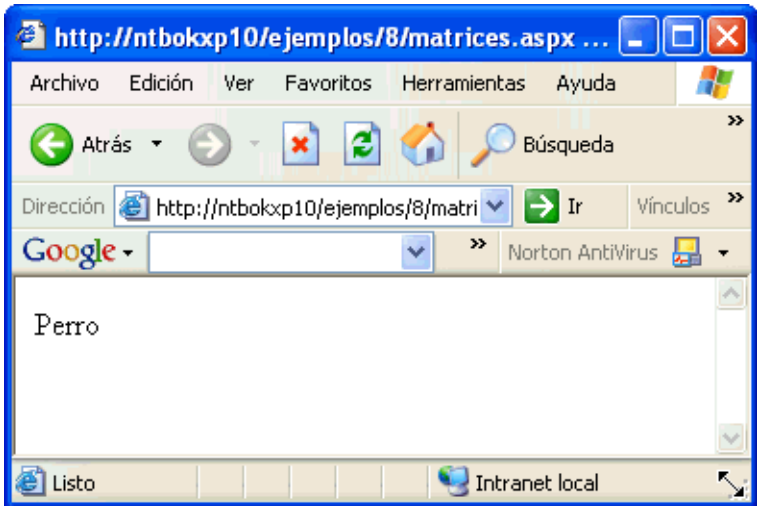
Quizás el código no es mas claro pero como a partir de ahora tenemos que trabajar con colecciones y hemos visto como una matriz se puede asimilar a una colección, de esta forma hemos comprendido mucho mejor como funciona el objeto "array" al ver su equivalente con una simple declaración de matrices. Vamos a copiar este ejemplo en una página para comprobar que funciona...

1. Crea una página web, le pones de nombre "matrices.aspx" y pones un control de tipo label en la página:

Y en el evento Load de la página le pones:

```
Protected Sub Page_Load(ByVal sender As Object, ByVal e As System.EventArgs) Handles Me.Load

    Dim matriz_animales(4) As String

    matriz_animales(0) = "León"
    matriz_animales(1) = "Gato"
    matriz_animales(2) = "Perro"
    matriz_animales(3) = "Elefante"
    matriz_animales(4) = "Gato"
    animal.Text = matriz_animales.GetValue(2)
End Sub
```

2. Ejecuta la página:



Que nos muestra el tercer elemento de la matriz.

1.2 Métodos de las matrices

Buscar en una matriz

Podemos (por fin) buscar elementos dentro de las matrices, la clase "Array" proporciona unos cuantos métodos muy interesantes, uno de ellos es "IndexOf" que devuelve la primera posición donde encuentre el valor especificado. Por ejemplo para buscar la primera vez que aparezca el elemento "Gato":

```
Array.Indexof (matriz_animales,"Gato")
```

Y nos devolverá el valor entero "1", podemos decirle también añadiendo un parámetro que nos busque a partir de una posición (¿te acuerdas de la sobrecarga?) Por ejemplo:

```
Array.Indexof (matriz_animales,"Gato",2)
```

La búsqueda comenzará por el segundo elemento y devolverá la siguiente posición de "Gato", que sería el 4 en nuestro ejemplo. También podemos buscar por el final, es decir busca la última aparición de "Gato" con "LastIndex"

```
Array.LastIndexof (matriz_animales,"Gato")
```

Que devolverá lógicamente 4.

Trabajar con varios elementos en una matriz

Tenemos varios métodos que permiten trabajar con mas de un elemento a la vez, en concreto podemos ordenar en orden ascendente y descendente una matriz. Para esto utilizaremos los métodos "reverse" y "Sort" de esta forma:

```
Array.Reverse (matriz_animales)
```

y

```
Array.Sort (matriz_animales)
```

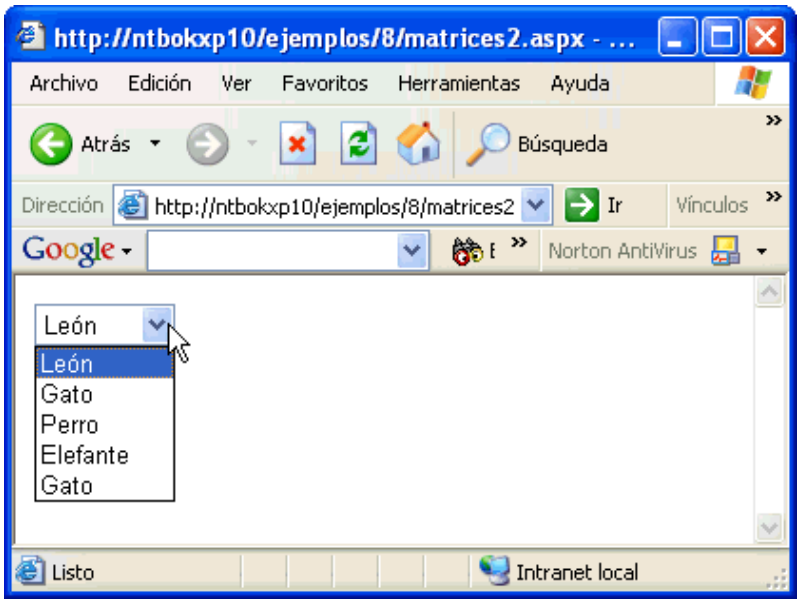
Ejemplo

Vamos a rellenar un cuadro de lista desplegable con nuestra matriz de ejemplo. Crea una página "matrices2.aspx" que tenga un cuadro desplegable:

Y en el evento load de la página le pones:

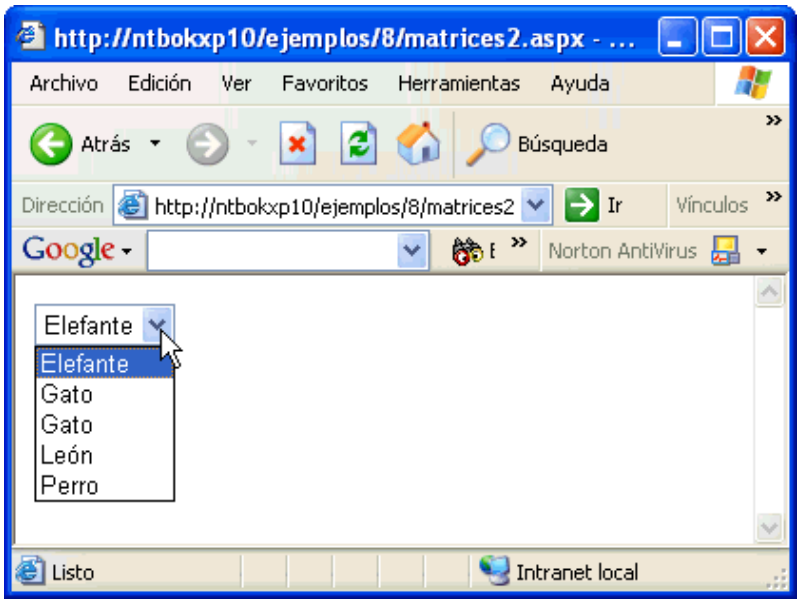
```
Protected Sub Page_Load(ByVal sender As Object, ByVal e As System.EventArgs) Handles Me.Load
    Dim matriz_animales(4) As String
    Dim stranimal As String
    matriz_animales(0) = "León"
    matriz_animales(1) = "Gato"
    matriz_animales(2) = "Perro"
    matriz_animales(3) = "Elefante"
    matriz_animales(4) = "Gato"
    For Each stranimal In matriz_animales
        lista.items.add(stranimal)
    Next

End Sub
```

El resultado será:



Para manejar elementos en un cuadro de lista desplegable añadimos los elementos a su colección. Recuerda que la colección de los elementos de un cuadro de lista desplegable está en "items", así que llamaremos a su método "add" para añadir un elemento a su colección.

Ahora escribe la instrucción "Array.Sort (matriz_animales)" justo antes del bucle "For Each .. Next" y actualiza la página:



La lista aparece ahora ordenada.

Enlace a datos

El enlace a datos es una de las cosas que mas utilizaremos cuando trabajemos con bases de datos, es el proceso de enlazar una matriz o una colección con un origen de datos. El origen de datos puede ser una base de datos pero también puede ser otra matriz. Por ejemplo queremos enlazar (binding) un cuadro de lista desplegable con la matriz de los animales:

```
matriz_animales(0)="León"

matriz_animales(1)="Gato"

matriz_animales(2)="Perro"

matriz_animales(3)="Elefante"

matriz_animales(4)="Gato"

lista.DataSource=matriz_animales

lista.DataBind()
```

Pero esto lo veremos mas adelante...

Ventajas e inconvenientes de trabajar con matrices

Las matrices son elementos muy utilizados para agrupar elementos y funcionan bien pero también hay que tener en cuenta sus limitaciones...

Ventajas

- Facilidad de uso. Prácticamente hemos visto todo lo que se puede hacer con matrices y es realmente sencillo.
- Rápidas para modificar elementos. Podemos localizar de una forma rápida un elemento y modificarlo...
- Desplazamiento rápido por los elementos. A partir de un índice el desplazamiento por los elementos es rápido porque se almacenan en una zona contigua de memoria
- Podemos especificar el tipo de elementos. Al crear la matriz podemos indicarle el tipo de datos que va a almacenar.

Inconvenientes

- Tamaño fijo. Cuando declaramos una matriz le indicamos el tamaño fijo, ni mas ni menos elementos. Si redimensionamos con Redim el proceso es lento con matrices grandes y tenemos que realizar este proceso siempre que varíe su tamaño
- Es complicado insertar elementos. Si queremos insertar elementos es un proceso complicado, primero hay que redimensionar la matriz y luego desplazar los elementos una posición. Las matrices no están diseñadas para realizar inserciones en medio de ellas.

Veamos ahora otros objetos que nos solucionarán algunas de estas cosas

1.3 ArrayList

Implementa una matriz cuyo tamaño aumenta dinámicamente según se requiera. Perfecto, es como una matriz pero se reconfigura cuando varía de tamaño, las ventajas son evidentes:

- Soporta redimensionamiento automático. Cuando creamos un ArrayList no necesitamos decirle los límites, no hace falta que sepamos cuantos elementos va a tener la lista
- Flexibilidad al insertar elementos. Podemos crear un ArratList vacío e ir introduciendo los valores según nos haga falta.
- Flexibilidad para borrar elementos.
- Facilidad de uso. Se trabaja de una forma muy intuitiva y es una sintaxis muy similar a la de muchos controles de .NET

En cambio como límite podemos destacar el rendimiento, son mas lentas que las matrices, pero no es significativo.

Para crear un ArrayList utilizaremos la sintaxis:

```
Dim mimatriz as new ArrayList()
```

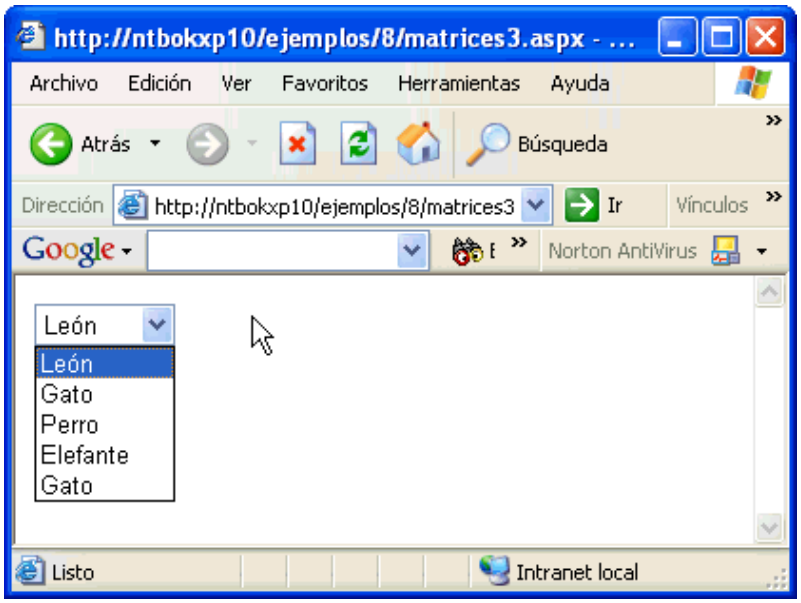
Es decir, crea un objeto "mimatriz" a partir de la clase "ArrayList". Para añadir elementos utilizaremos el método "Add":

```
mimatriz.add ("León")

mimatriz.add ("Gato")

mimatriz.add ("Perro")

mimatriz.add ("Elefante")

mimatriz.add ("Gato")
```

La sintaxis es algo diferente de las matrices pero hazte idea que su uso es idéntico, veamos un ejemplo con nuestros animales. Crea una páginas "matrices3.aspx" que tenga un cuadro desplegable como el de antes pero con este código en el evento Load de la página:

```
Protected Sub Page_Load(ByVal sender As Object, ByVal e As System.EventArgs) Handles Me.Load
    Dim mimatriz As New ArrayList()

    mimatriz.Add("León")
    mimatriz.Add("Gato")
    mimatriz.Add("Perro")
    mimatriz.Add("Elefante")
    mimatriz.Add("Gato")

    lista.DataSource = mimatriz
    lista.DataBind()
End Sub
```

Ejecútalo en el navegador:

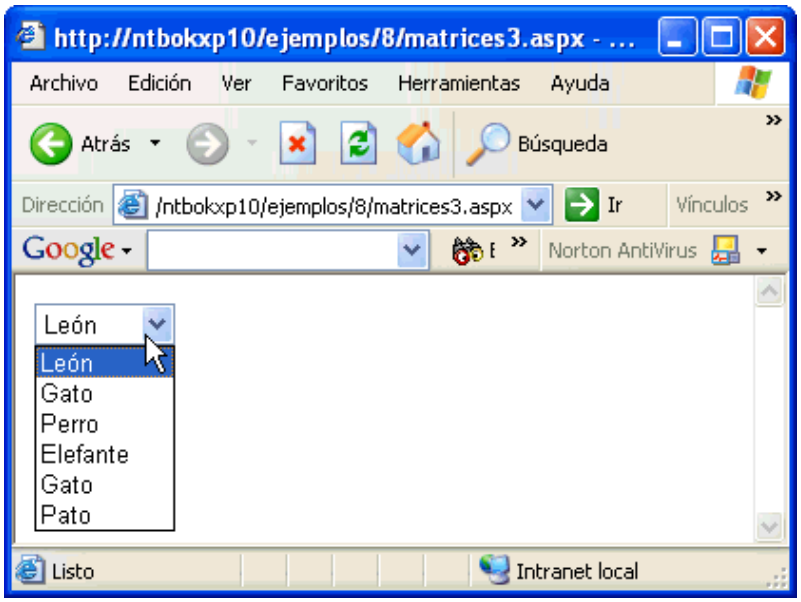


Ahora añade un animal justo debajo de los otros:

```
mimatriz.add ("León")

mimatriz.add ("Gato")

mimatriz.add ("Perro")
```

```
mimatriz.add ("Elefante")

mimatriz.add ("Gato")

mimatriz.add ("Pato")
```

Y actualiza la página:



Sin redimensionar la matriz ni ninguna operación especial la matriz ha cambiado de tamaño y ahora tiene el nuevo elemento. Observa que hemos "enlazado" con un método que veremos mas adelante el cuadro de lista desplegable con una matriz:

```
lista.datasource=mimatriz
lista.databind()
```

Insertar elementos en las matrices

Para insertar elementos al final de la lista utilizaremos el método ya conocido:

```
mimatriz.add ("Pato")
```

Si queremos insertarlo en otro posición, es decir, en medio de la lista utilizaremos el método "Insert"

```
mimatriz.Insert (2,"Gato")
```

Que lo insertaré en la posición 2. Además tenemos estos métodos:

- Sort. Como en las matrices ordena.
- Reverse. La ordena al revés

- IndexOf. Busca un elemento
- Count. Devuelve el número de elementos de la matriz

Para eliminar podemos hacerlo por su elemento

```
mimatriz.remove ("Pato")
```

O por su índice:

```
mimatriz.removeAt (2)
```

2. Depuración y control de errores

Una de las partes principales en cualquier tipo de desarrollo es el control de los errores que se puedan dar en el programa y cómo depurar nuestra aplicación para encontrar fallos de funcionamiento. Según Murphy, y además es implacable, "si algo puede salir mal , saldrá mal". Esto quiere decir que si hemos dejado de comprobar algún tipo de operación seguramente sea la primera que se ejecute. El ejemplo típico es pedirle un dato numérico al usuario y que éste escriba su nombre. Como nos esperábamos un valor numérico nuestro programa provocó una "excepción" o un error.

Tenemos dos formas de preparar nuestras aplicaciones para el mundo real:

- Depuración. Nuestro programa funciona pero en algunas ocasiones hace cosas extrañas, en este caso la mejor forma es ir buscando el error depurando el código. Un ejemplo típico es cuando una variable de tipo "integer" recoge un valor mayor que el que admite, el programa no falla y recoge un dato erróneo.
- Interceptación de errores. Esto es preparar nuestro código para posibles fallos. Por ejemplo si vamos a grabar en un disquete externo en lugar de decirle que lo grabe directamente le diremos que "lo intente" para que en el caso de que no haya disquete metido lo podamos controlar.

A pesar de que llegaremos a ser expertos en esto del ASP.NET es necesario pasar por las fases obligatorias de todo programador: desconcierto, tiempo perdido, malos resultados... todo esto se soluciona con paciencia y buenas costumbres. Para poder controlar lo que nos pasa en el programa debemos:

- Entender nuestro código. Un código sencillo y que sea lea bien es un buen código. Aquí es absolutamente falso eso de al ver un código decir "esto es muy difícil no entiendo nada". En ese caso con toda seguridad está mal escrito, un problema por muy complejo que sea tienen una solución legible y estructurada. Además si no entendemos muy bien nuestro código... ¿cómo vamos a encontrar el error?. Lo mejor de todo, instrucciones sencillas y sobre todo muchos comentarios, te ayudarán muchísimo cuando tengas que depurar un programa o simplemente tengas que ampliarlo al cabo de unos meses
- Identificar la fuente de problemas. Con el tiempo sabremos que partes del código son las que provocan los errores y son las que mas debemos controlar. Por ejemplo en un programa de bases de datos, la parte crítica es la parte de la conexión a ésta, luego ahí es dónde debemos tener mas cuidado.
- Prevenir. Escribir mas código que controle los casos de los usuarios, si escribimos las rutinas para evitar que el usuario meta su nombre en una casilla numérica mejor que esperar a que se provoque el error.

2.1 Errores de sintaxis

Son los que mas nos van a acompañar durante este aprendizaje. En este curso ya ves que he preferido poner capturas de pantalla de los ejemplos para "obligarte" a escribirlos, es la mejor forma de coger práctica. Los casos mas comunes, que ya se te habrán dado son:

- Error de sintaxis del código. Por ejemplo escribir <asp:textbx> en lugar de <asp:textbox>. En este caso el mensaje del navegador es claro
- Sintaxis del código incompleta. Por ejemplo puedes haber olvidado el cerrar una etiqueta:

```
<asp:TextBox id="nombre" runat="server">
```

  en lugar de

```
<asp:TextBox id="nombre" runat="server" />
```

- Combinar palabras claves de otros lenguajes. A menudo después de haber trabajado con varios lenguajes podemos tener la lógica confusión incluso en las instrucciones mas sencillas. Por ejemplo "elseif" de VB.NET es "else if" en Javascript
- No cerrar un construcción, por ejemplo dejarnos el "next" en un bucle for. O mira este ejemplo:

```
If condicion1 then

    'Haz esto

elseif condicion2 then

    'Haz esto

if condicion2a then

    'Haz esto

else

    'Haz esto

end if
```

  Da error y a simple vista parece completo pero no, le falta un "end if" al final, esto lo solucionamos si escribimos bien el código con las tabulaciones:

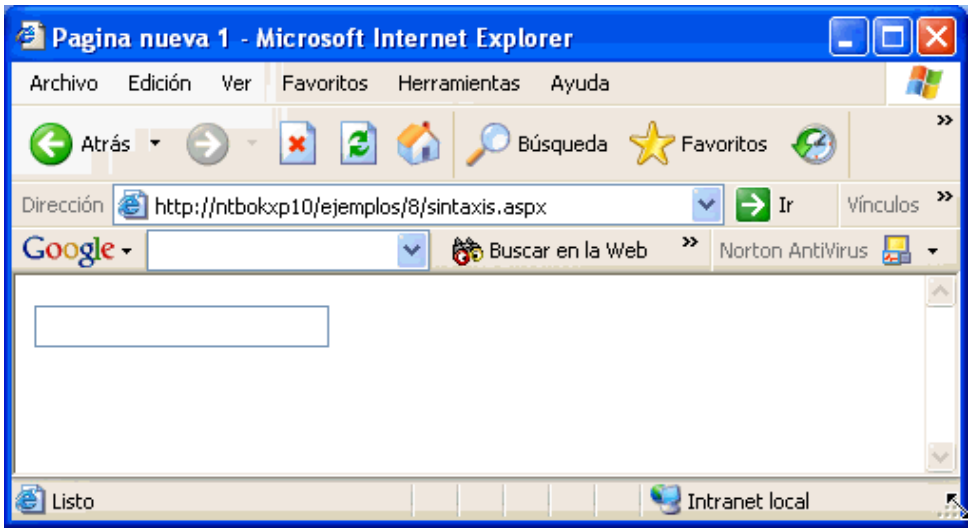
```
If condicion1 then

    'Haz esto

elseif condicion2 then

    'Haz esto

    if condicion2a then

        'Haz esto

    else
```

```
            'Haz esto

        end if

    end if
```

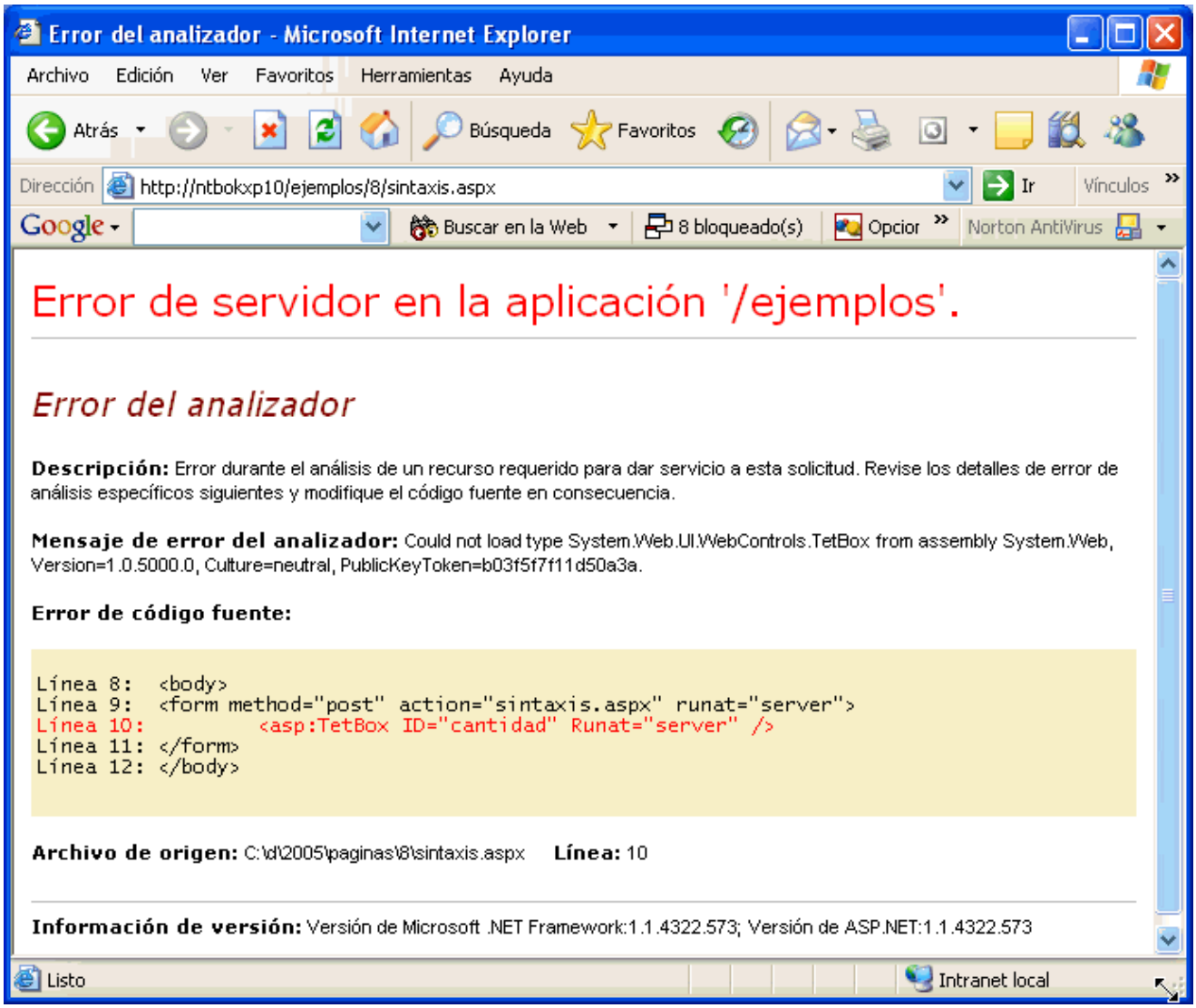
Vamos a provocar un error para ver la información de pantalla. Crea una nueva página que se llame "sintaxis.aspx" con este código:

```
<html>

<head>
<meta http-equiv="Content-Type" content="text/html; charset=windows-1252">
<title>Pagina nueva 1</title>
</head>

<body>
<form method="post" action="sintaxis.aspx" runat="server">
    <asp:TetBox ID="cantidad" Runat="server" />
</form>
</body>

</html>
```

El resultado debería ser este:



Pero recibimos este:



Esta pantalla ya nos es bastante familiar... pero es muy buena ya que nos indica perfectamente el error. Como ves ha destacado la línea 10 en mi caso y dice que no puede "Could not load type System.Web.UI.WebControls.TetBox" Es decir que no encuentra en el espacio de nombres el control "TetBox". Obviamente porque está mal escrito, corregimos por "TextBox y solucionado.

2.2 Errores lógicos

Los errores lógicos son mas difíciles de encontrar ya que intervienen factores como variables y operadores. El ejemplo mas típico es hacer una calculadora y que se presenta una división por 0. Como esto no se puede realizar, sino hemos interceptado el error, se produce una parada del programa. Así que las razones que producen estos fallos son:

- División por 0. No se puede predecir este fallo porque pueden intervenir muchos factores pero si podemos cambiar algo nuestro código y en las operaciones de este tipo decirle de alguna forma que "intente" hacer la operación, esto lo veremos mas adelante...
- Tipo desconocido o asignación incorrecta. Se produce cuando asignamos tipos de datos incompatibles, por ejemplo querer asignarle la palabra "pepe" a una variable de tipo entero

- Salida incorrecta. Cuando un procedimiento o función devuelven un valor no esperado por nuestro programa
- Utilizar un objeto que no existe. Hacemos referencia a un objeto que todavía no se ha creado o se ha destruido.
- Proceso de datos no válido. De produce cuando un programa acepta datos no válidos

2.3 Errores de sistema

Son los generados por el propio ASP.NET, producidos por fallos en el código, un fallo en ASP.NET y incluso fallos en el sistema operativo.

2.4 Buenas costumbres

Una de las mejores formas de prevenir errores es tener limpieza y buenas costumbres al escribir el código, me refiero estos caso.

Tabular el código

Es muy importante agrupar las partes de código con las tabulaciones para saber las partes que pertenecen a otras. En el ejemplo anterior si leemos el código:

```
If condicion1 then

'Haz esto

elseif condicion2 then

'Haz esto

if condicion2a then

'Haz esto

else

'Haz esto

end if

end if
```

En lugar de

```
If condicion1 then

    'Haz esto

elseif condicion2 then

    'Haz esto

    if condicion2a then
```

```
            'Haz esto
        else
            'Haz esto
        end if
    end if
```

seguramente nos evitará mucho tiempo de investigación en localizar el fallo.

Utilizar nombres descriptivos

Siempre será mejor utilizar como variables "txtnombre" y "txtapellidos" que "textbox1" y "textbox2"

Comentar el código

Todo lo que escribamos como comentarios se convertirá en ayuda al leer el código así que comenta cuanto mas mejor.

Estructurar el código

Utiliza funciones y procedimientos para dividir una acción compleja en varias mas pequeñas... "divide y vencerás" es el lema del buen programador. Los módulos pequeños son mucho mas fáciles de depurar y corregir que un "tocho" de código. Siempre que veas un ejemplo por ahí y no lo entiendas seguramente es porque está mal hecho, el buen código se lee muy fácil. Do todas las soluciones posibles a un problema la mas sencilla es la mejor.

Convierte las variables a su valor correcto

Asegúrate de transformar los datos al recogerlos del formulario para trabajar mejor con ellos. Las funciones de VB.NET para conversiones de tipos son:

Nombre de la función	Tipo de valor devuelto	Intervalo de valores del argumento expresión
CBool	Boolean	Cualquier expresión numérica o de cadena (String) válida.
CByte	Byte	0 a 255; las fracciones se redondean.
CChar	Char	Cualquier expresión String válida, valores comprendidos entre 0 y 65535.
CDate	Date	Cualquier representación válida de fecha y hora.
CDbl	Double	-1,79769313486231E+308 a -4,94065645841247E-324 para valores negativos; 4,94065645841247E-324 a 1,79769313486231E+308 para valores positivos.

CDec	Decimal	+/-79.228.162.514.264.337.593.543.950.335 para números a partir de cero, es decir, números sin decimales. Para números con 28 decimales, el rango es +/-7.9228162514264337593543950335. El menor número distinto de cero es 0,0000000000000000000000000001.
CInt	Integer	-2.147.483.648 a 2.147.483.647; las fracciones se redondean.
CLng	Long	-9.223.372.036.854.775.808 a 9.223.372.036.854.775.807; las fracciones se redondean.
CObj	Object	Cualquier expresión válida.
CShort	Short	-32.768 a 32.767; las fracciones se redondean.
CSng	Single	De -3,402823E+38 a - 1,401298E-45 para valores negativos; de 1,401298E-45 a 3,402823E+38 para valores positivos.
CStr	String	Los valores devueltos para CStr dependen del argumento expresión. Mira la siguiente tabla

En la tabla siguiente se describen los valores devueltos por la función **CStr** en varios tipos de *expression*

Si el tipo de *expression* es	**CStr devuelve**
Boolean	Cadena que contiene "True" o "False".
Date	Cadena que contiene un valor **Date** (fecha y hora) en el formato de fecha corta del sistema.
Numeric	Cadena que representa el número.

2.5 Controlar errores de ejecución

Vamos a intentar aislar los errores que se puedan dar en nuestros programas. Primero vamos a interceptar datos no válidos realizando un ejemplo.

En este programa queremos leer un valor numérico que esté entre 0 y 10. Podemos hacer un control del programa de dos formas ya que hay que cuidar que el usuario:

- Escriba un dato
- Que lo que haya escrito sea un número
- Que el número que haya escrito sea mayor que 0

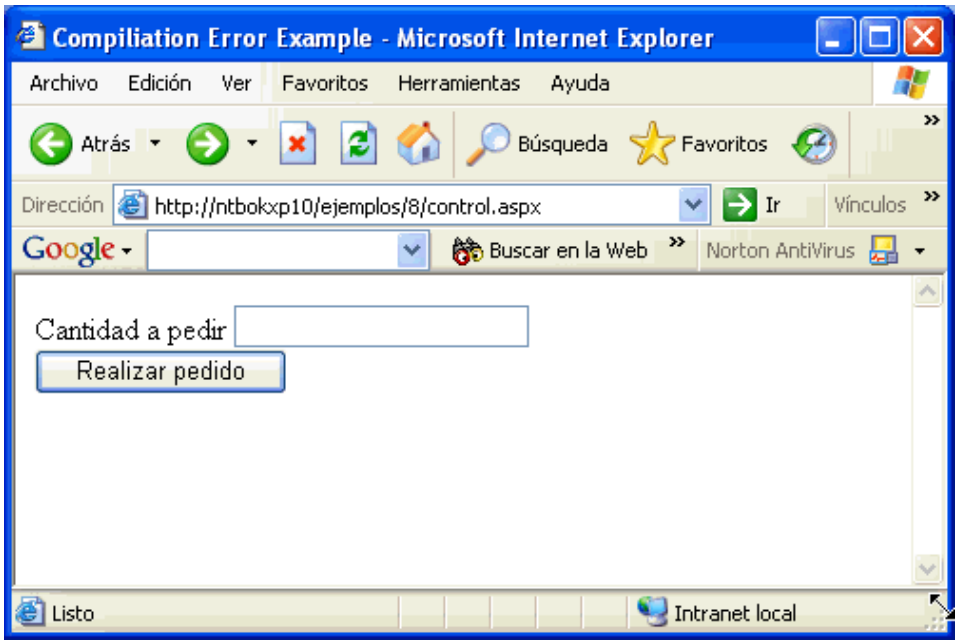
Todo esto tenemos que comprobar... vamos a hacerlo de dos formas, una es manual y la otra con validaciones. Vamos con la primera

Crea una nueva página llamada "control.aspx" que tenga un literal, un cuadro de texto (que se llame txtcantidad), un botón (btnpedir) y control label (lblconfirmacion) así:

Y le pones en el evento clic del botón este código:

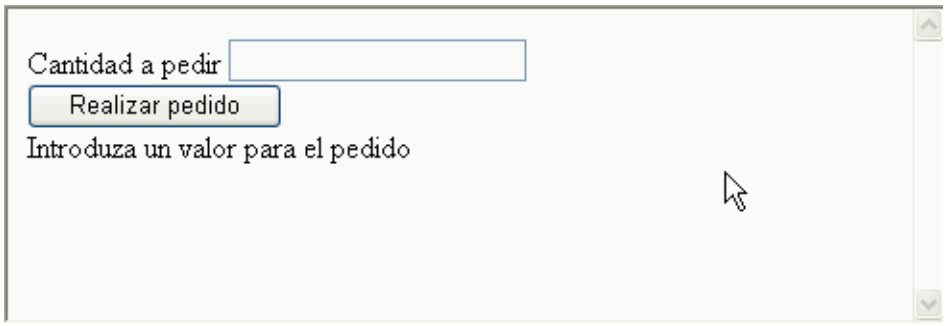
```
Protected Sub btnpedir_Click(ByVal sender As Object, ByVal e As System.EventArgs) Handles
    If txtcantidad.Text = "" Then
        lblconfirmacion.Text = "Introduza un valor para el pedido"
    ElseIf Not isNumeric(txtcantidad.text) Then
        lblconfirmacion.Text = "El valor no es numérico"
    ElseIf CInt(txtcantidad.Text) <= 0 Then
        lblconfirmacion.Text = "La cantidad debe ser mayor de 0."
    ElseIf CInt(txtcantidad.Text) > 0 Then
        lblconfirmacion.Text = "Pedido realizado correctamente."
    Else
        lblconfirmacion.Text = "Introduzaca un valor correcto."
    End If
End Sub
```

Ejecuta la página:

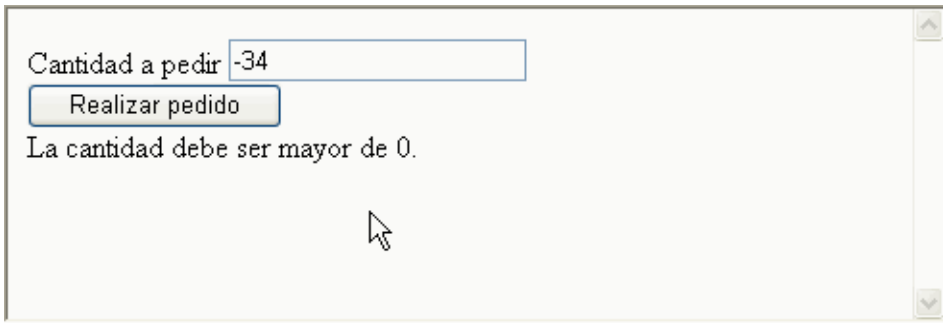


Ahora vamos a probarla, introduce un valor no numérico:

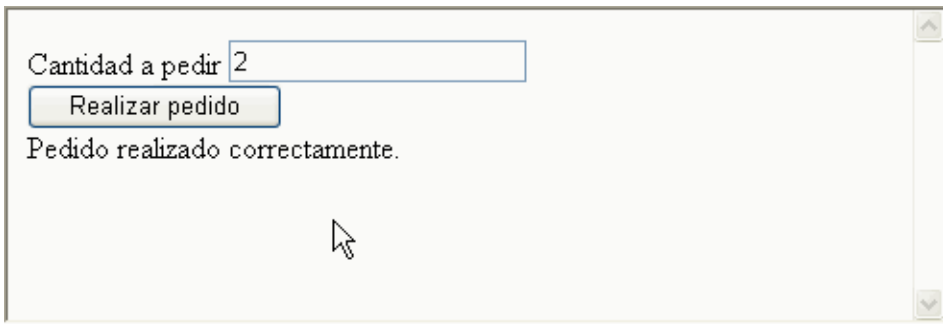
Ahora deja el cuadro en blanco:



Ahora con un valor negativo:



Y por fin un valor correcto:



Bien, nos han funcionado todos los casos. Esto es un poco laborioso pero no hay posibilidad de que falle nuestro programa. Para poder ejecutar esto hemos en un entorno de pruebas y así poder depurarlo hemos activado la depuración en el fichero de configuración config.web:

```
<system.web>
    <!--
        Set compilation debug="true" to insert debugging
        symbols into the compiled page. Because this
        affects performance, set this value to true only
        during development.

        Visual Basic options:
        Set strict="true" to disallow all data type conversions
        where data loss can occur.
        Set explicit="true" to force declaration of all variables.
    -->
    <compilation debug="true" strict="false" explicit="true">
```

Es muy importante que recuerdes esta directiva ya que nos va a proporcionar información extra muy útil para solucionar los problemas. Incluso en ocasiones cuando se nos produzca un error el mensaje nos dirá que "si activas el debug=true" se proporcionará información detallada del error. Esto es muy práctico para nuestras depuraciones así que de momento la activaremos.

Sigamos con el ejemplo anterior donde vemos que en el código no tenemos ninguna sorpresa, un cuadro de texto y un botón que ejecuta un procedimiento en el evento "click":

```
<asp:TextBox id="txtcantidad" runat="server" />
<br />
<asp:Button id="btnpedir" Text="Realizar pedido"
       onclick="realizarpedido"
       runat="server"/>
```

El trabajo está precisamente en ese procedimiento, donde tendremos que evaluar una por una todas las posibilidades:

Que no haya escrito nada

```
if txtcantidad.Text = "" then
  lblconfirmacion.Text = "Introduza un valor para el pedido"
```

Que lo que haya escrito no sea numérico

```
else if not isNumeric (txtcantidad.text) then
  lblconfirmacion.Text = "El valor no es numérico"
```

Que lo que haya escrito sea inferior a 0

```
else if CInt(txtcantidad.Text) <= 0 then
  lblconfirmacion.Text = "La cantidad debe ser mayor de 0."
```

Bienvenido al laborioso mundo de la depuración. Es muy fácil escribir programas rápidos, en Visual Basic pones cuatro controles en pantalla y ya está todo funcionando. Pero lo difícil es hacer que funcione correctamente en todos los "escenarios", es decir, el control de errores necesario para que funcione bien el programa. Esta parte lleva normalmente el doble de tiempo que la realización de los propios algoritmos del programa pero ahí está la diferencia entre un buen programa y otro mal hecho.

El segundo caso para controlar los posibles errores de nuestro programa es utilizar los controles de validación que tiene ASP.NET.

3. Controles de validación

Mediante los controles de validación se puede agregar validación de entrada a páginas de formularios Web Forms. Los controles de validación proporcionan un mecanismo fácil de usar para todos los tipos comunes de validación estándar (por ejemplo, probar fechas válidas o valores comprendidos en un intervalo), además de otras formas para proporcionar validación escrita personalizada. Además, los controles de validación permiten personalizar completamente cómo se muestra la información de errores al usuario.

Los controles son:

Propiedad	Descripción
CompareValidator	Compara el valor especificado por el usuario en un control de entrada con el valor especificado en otro control de entrada o con un valor constante
CustomValidator	Realiza una validación definida por el usuario en un control de entrada
RangeValidator	Comprueba si el valor de un control de entrada está comprendido en un intervalo especificado de valores
RegularExpressionValidator	Comprueba si el valor de un control de entrada asociado coincide con el modelo especificado por una expresión. Por ejemplo, una dirección IP, un número de teléfono, email, ...
RequiredFieldValidator	Convierte el control de entrada asociado en un campo obligatorio

En un formulario se pueden asociar más de un control de validación a un control. Por ejemplo, podría especificar que un control se requiera y que contenga un intervalo de valores específico. Es decir, la idea es asociar un control de validación con un control de nuestra página, por ejemplo con nuestro cuadro de texto. Esto hace que se le añada el control de errores del control de validación.

Es muy fácil de utilizar, lo verás al utilizarlo dos veces y proporciona un ahorro considerable de tiempo al no tener que escribir el código de validación y además es mas potente al controlar mas opciones. Así que por mi parte es **OBLIGATORIO** que los utilices para prevenir problemas en las lecturas de datos por pantalla. (no se mi he explicado bien, no es opcional: es obligatorio)

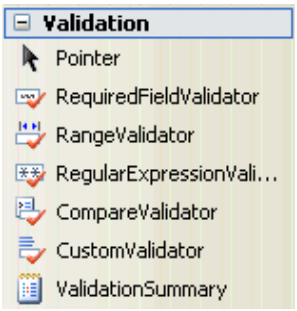
Estos controles no son visibles en las páginas, simplemente aportan esa funcionalidad oculta.

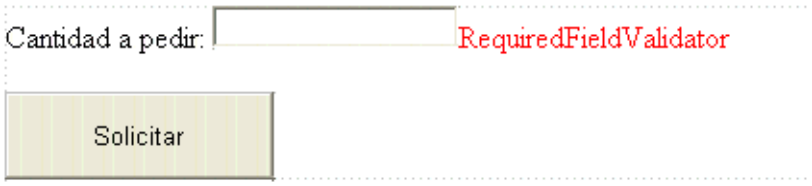
Normalmente, los controles de validación no son visibles en el formulario representado. No obstante, si el control detecta un error, produce el texto de mensaje de error que especifique. El mensaje de error se puede mostrar de formas diferentes, como se indica en esta tabla:

Opción de presentación	Descripción
En contexto	Cada control de validación puede mostrar individualmente un mensaje de error en contexto (normalmente al lado del control donde se ha producido el error).
Resumen	Los errores de validación pueden recopilarse y mostrarse en un lugar, por ejemplo, en la parte superior de la página. Esta estrategia se utiliza a menudo junto con la presentación de un glifo al lado de los campos de entrada con errores. Si el usuario

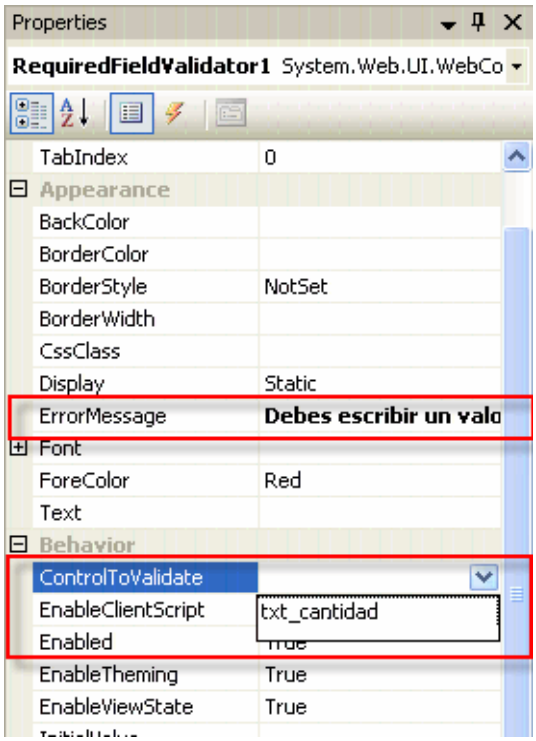
	trabaja con Internet Explorer 4.0 o posterior, el resumen puede mostrarse en un cuadro de mensaje.
En contexto y en un resumen	El mensaje de error puede ser diferente en el resumen y en contexto. Podemos utilizar esta opción para mostrar un mensaje de error más corto en contexto y más detallado en el resumen, o para mostrar un glifo de error al lado del campo de entrada y un mensaje de error en el resumen.
Personalizar	Podemos crear una presentación personalizada de un error capturando la información de error y diseñando su propio resultado.

Veamos el ejemplo anterior utilizando un control de validación. Creamos una página en la que nos debe solicitar un valor, pondremos un cuadro de texto, un botón y control de validación en el que nos obligue a poner un valor, es decir, de tipo "RequiredFieldValidator". Los controles los tienes en la barra de controles de la izquierda:

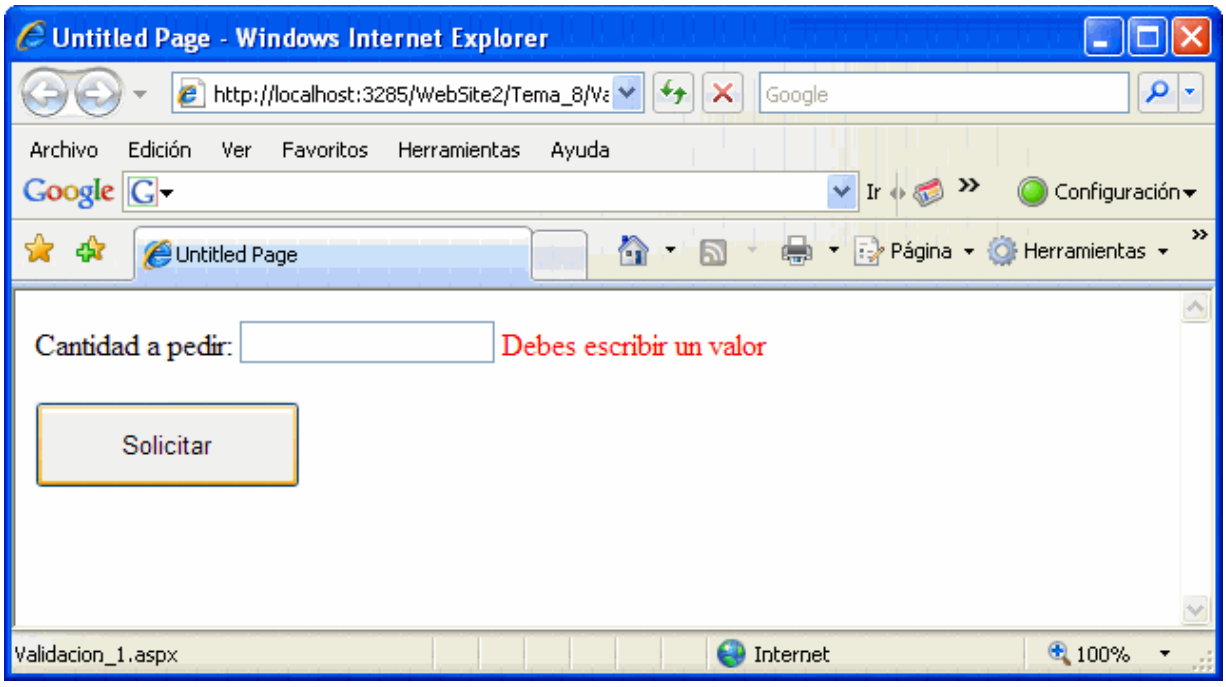




Hemos puesto un control de cuadro de texto y uno de validación, ahora debemos personalizar éste para que sepa con que control está asociado, es decir, con el que realizará las validaciones. Así que pondremos estas dos propiedades, una para el mensaje que nos va a mostrar "ErrorMEssage" y otro al control que va a controlar, que es un desplegable con la lista de controles de la página.



Recuerda poner la propiedad "ControlToValidate". Ahora ejecuta la página, no escribas nada en el cuadro de texto y pulsa el botón, te mostrará un mensaje como este:

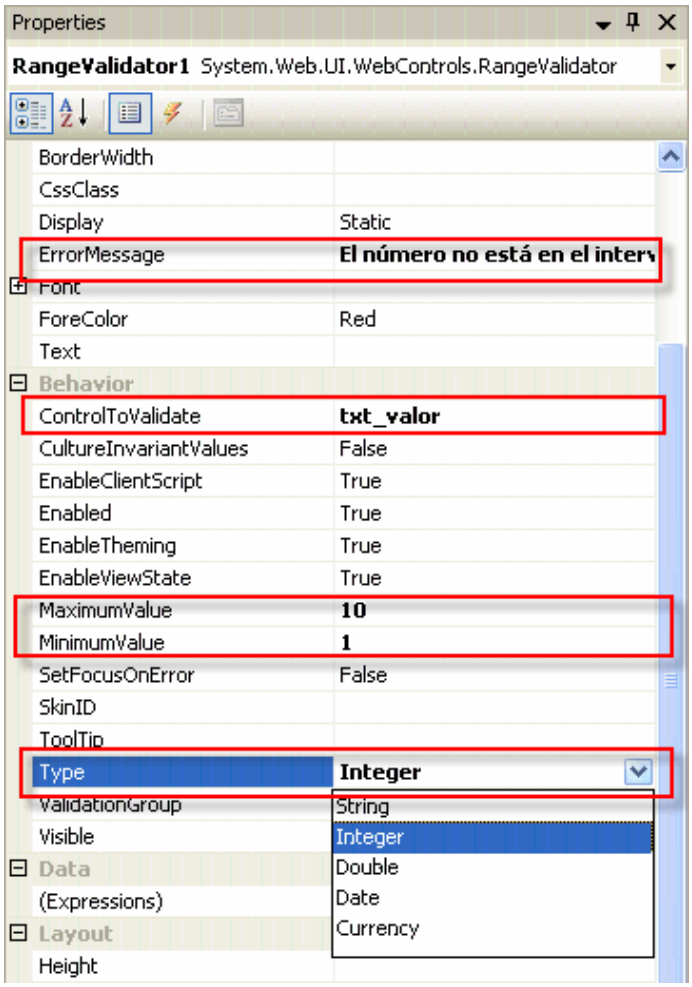


El funcionamiento es muy sencillo y nos va a ahorrar muchas líneas de código de comprobación. Si esto lo combinamos con otro control que nos permite leer un valor entre un rango, (del 1 al 10 por ejemplo) hemos sustituido todo el código del ejemplo del principio con un par de controles.

Ejemplo

Veamos un ejemplo, ahora queremos controlar un intervalo entre el 1 y 20. Así que será de tipo "rango", vamos con la página:

En este caso ponemos un control de validación de tipo "rangevalidator" para que nos chequee que el valor esté en el intervalo:

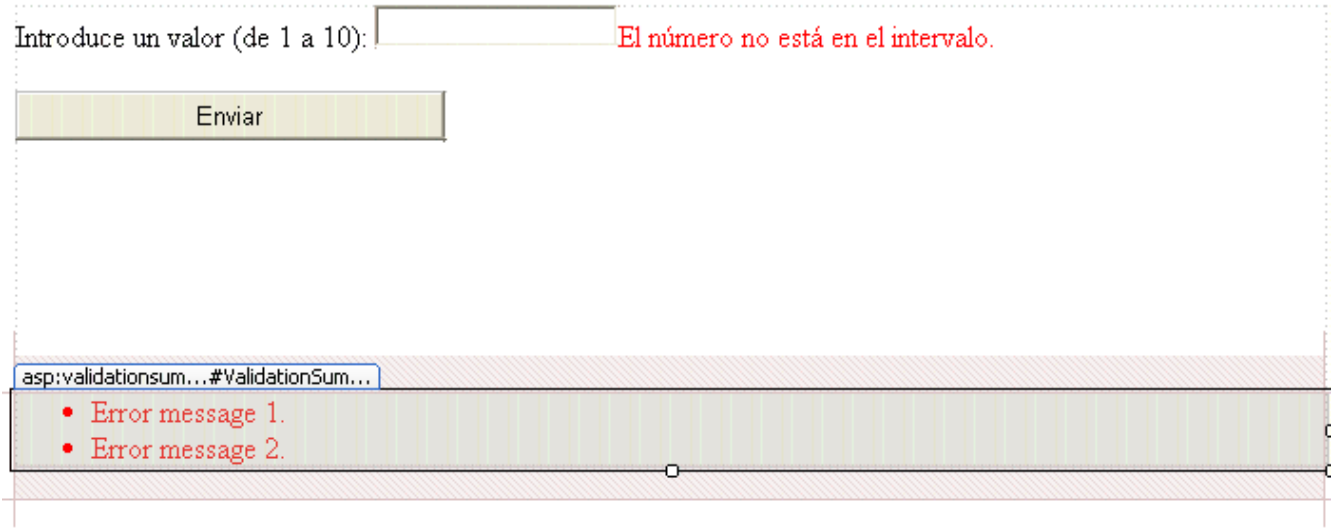


Como ves, las propiedades en este caso han sido la del mensaje de error, el control a validar, los valores máximo y mínimo del intervalo y el tipo de datos. Si la ejecutamos:

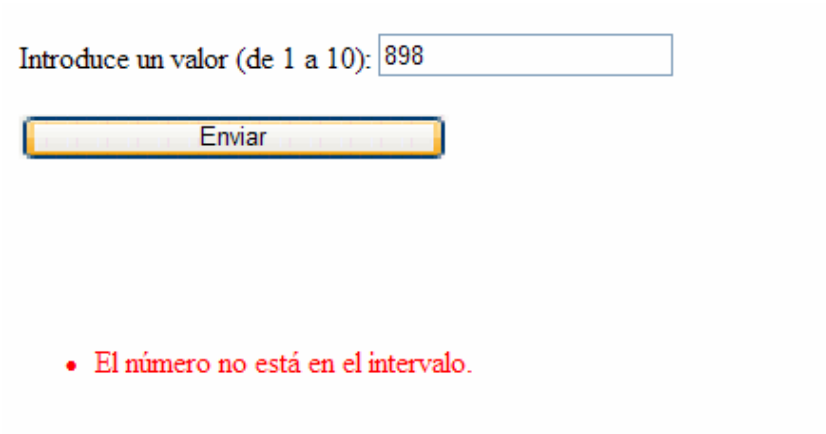
Nos funcionará comprobando que se haya escrito algo, que sea numérico y que esté en el intervalo.

Mas detalles de los controles de validación

Puede que no queramos mostrar los errores a continuación del control y centralizar todos los mensajes del formulario en un mismo sitio. Me explico, nuestro formulario puede tener muchos controles y cada uno con sus controles de validación y puede que no nos guste que cada control muestre su mensaje de error, además, si es un formulario muy denso puede que no tengamos o no queramos dejar espacio para estos mensajes. Vamos con el ejemplo anterior y le ponemos debajo un control de tipo "ValidationSummary":



Como ves, en la parte inferior vamos a escribir todos los mensajes. Solo tenemos que modificar una propiedad en el control de validación del intervalo para que no nos muestre el mensaje y así lo deje al control de resumen de validación que acabamos de añadir. La propiedad es "Display" que debe estar a "none". El resultado será que nos va a mostrar los mensajes de error que se produzcan en todos los controles en la parte inferior:

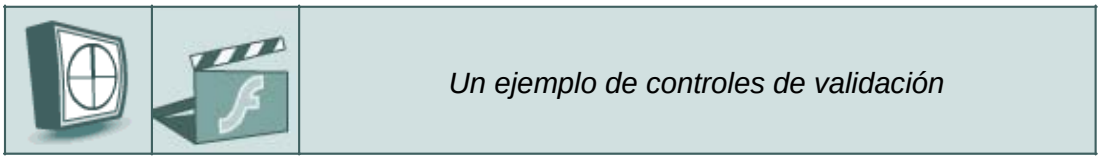


Si el formulario tuviese varios controles, nos mostraría los posibles errores de todos.

Tenemos también un control de validación que nos va a comparar entre dos valores (CompareValidator). Por ejemplo cuando introducimos una contraseña en un formulario y nos pide que la escribamos de nuevo para comprobar si se ha escrito bien. En ese caso los controles serían de este tipo:

```
<asp:RequiredFiledVaildator id="contrasena" runat="sever"
```

```
        ErrorMessage="Debes escribir una contraseña" CotrolTOValidate="txt_password" / >

<asp:CompareValidator id="repetir_contrasena" runat="sever"

        ErrorMessage="Las contraseñas no coinciden"

                ControlToCompare="contrasena" CotrolToValidate="txt_repetir" / >
```



Un ejemplo de controles de validación

4. Encontrar errores

¡Qué fácil es decirlo!. Pues si, al principio se nos darán tantos errores que no será complicado hasta aislarlos para ir corrigiéndolos, pero esto sólo es problema de práctica, aun así veamos las situaciones mas normales.

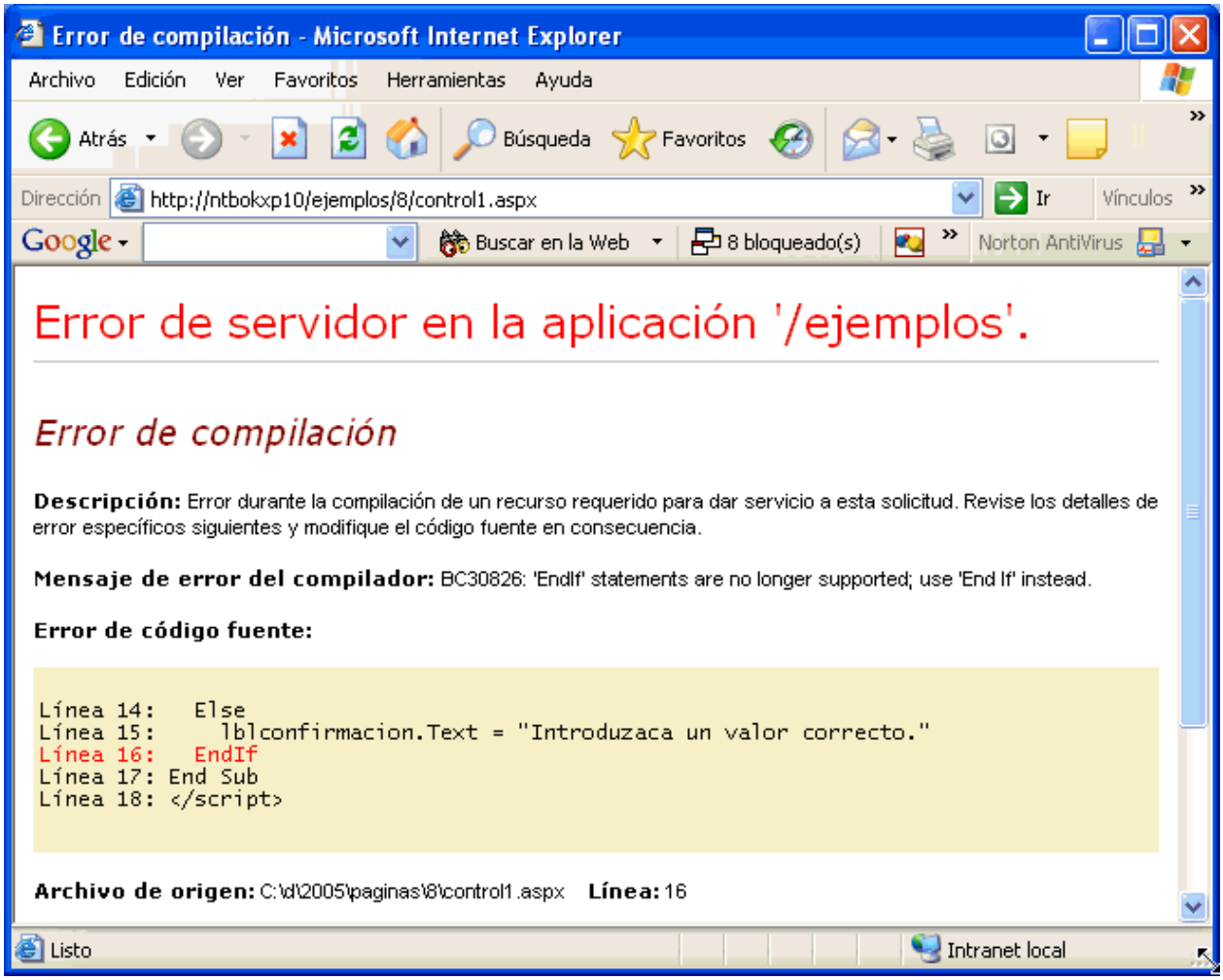
Vamos a provocar varios errores para ver los mensajes.

Error de compilación

Se producen al generar la página y son debidos a errores de sintaxis normalmente. Modifica la última línea del procedimiento del programa anterior y pon "endif" junto en lugar de separado "end if":

```
Else
  lblconfirmacion.Text = "Introduzaca un valor correcto."
EndIf
```

Ejecuta la página:



Bien, se ha producido un error pero bien por ASP.NET nos indica en rojo la línea del error, y además nos dice en el "mensaje de error":

Mensaje de error del compilador: BC30826: 'EndIf' statements are no longer supported; use 'End If' instead.

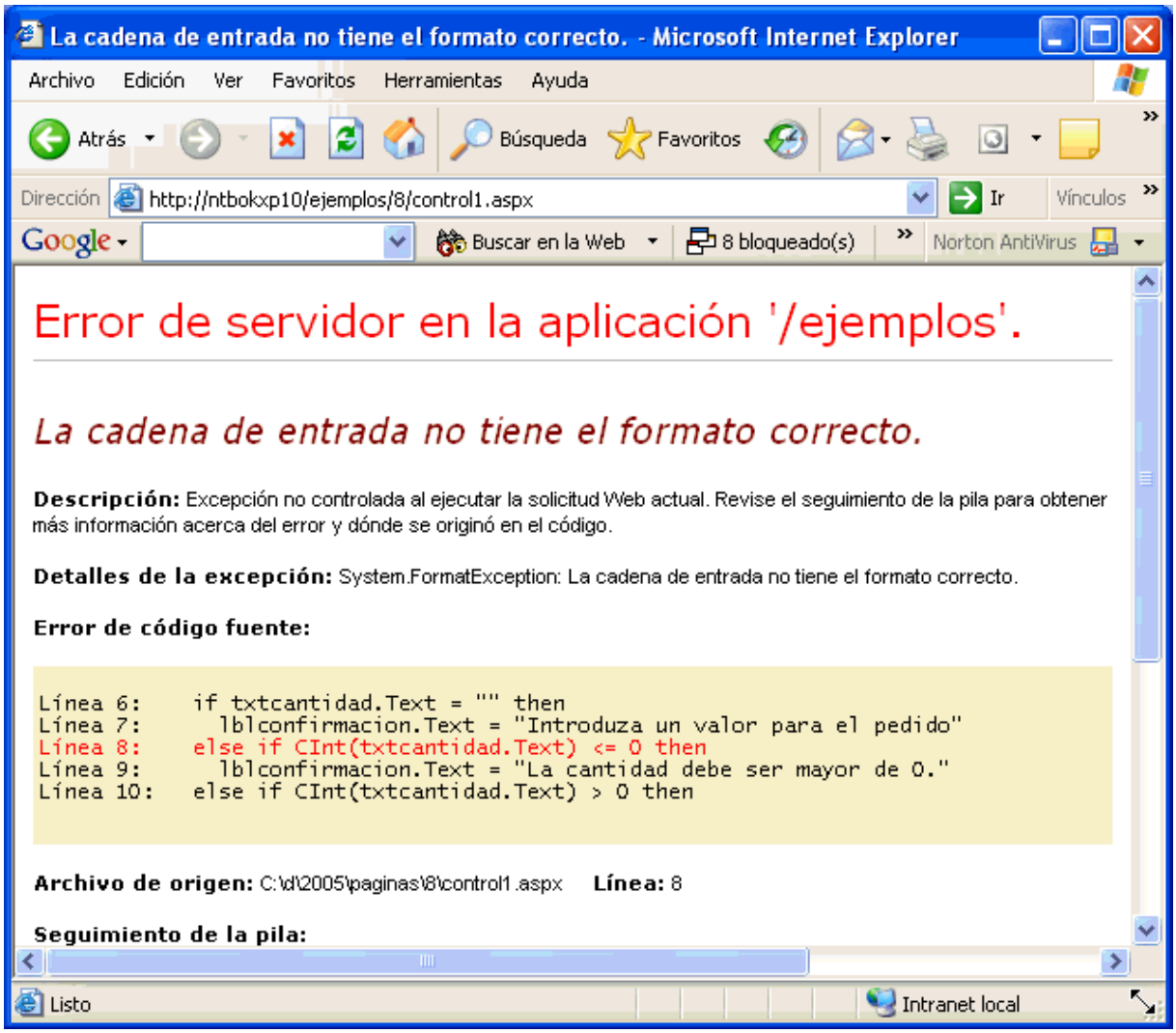
Que utilicemos "end if", mas ayuda no se puede tener así da gusto...

Error en tiempo de ejecución

Se producen cuando se espera un valor y se recibe otro, modifica el ejemplo con:

```
Sub realizarpedido(sender As Object, e As EventArgs)
  if txtcantidad.Text = "" then
    lblconfirmacion.Text = "Introduza un valor para el pedido"
  else if CInt(txtcantidad.Text) <= 0 then
    lblconfirmacion.Text = "La cantidad debe ser mayor de 0."
  else if CInt(txtcantidad.Text) > 0 then
      lblconfirmacion.Text = "Pedido realizado correctamente."
  Else
    lblconfirmacion.Text = "Introduzaca un valor correcto."
  End If
End Sub
```

Le hemos quitado el control de si es numérico lo que se ha escrito, así que si escribimos un texto en el cuadro al ejecutar la página obtendremos este bonito mensaje:



En este caso la función "CInt" intenta convertir a numérico el valor que le hemos pasado, como no puede se genera un error en tiempo de ejecución, es decir durante el transcurso de la ejecución de las páginas. La mejor solución es ponerle unos controles de validación de datos.

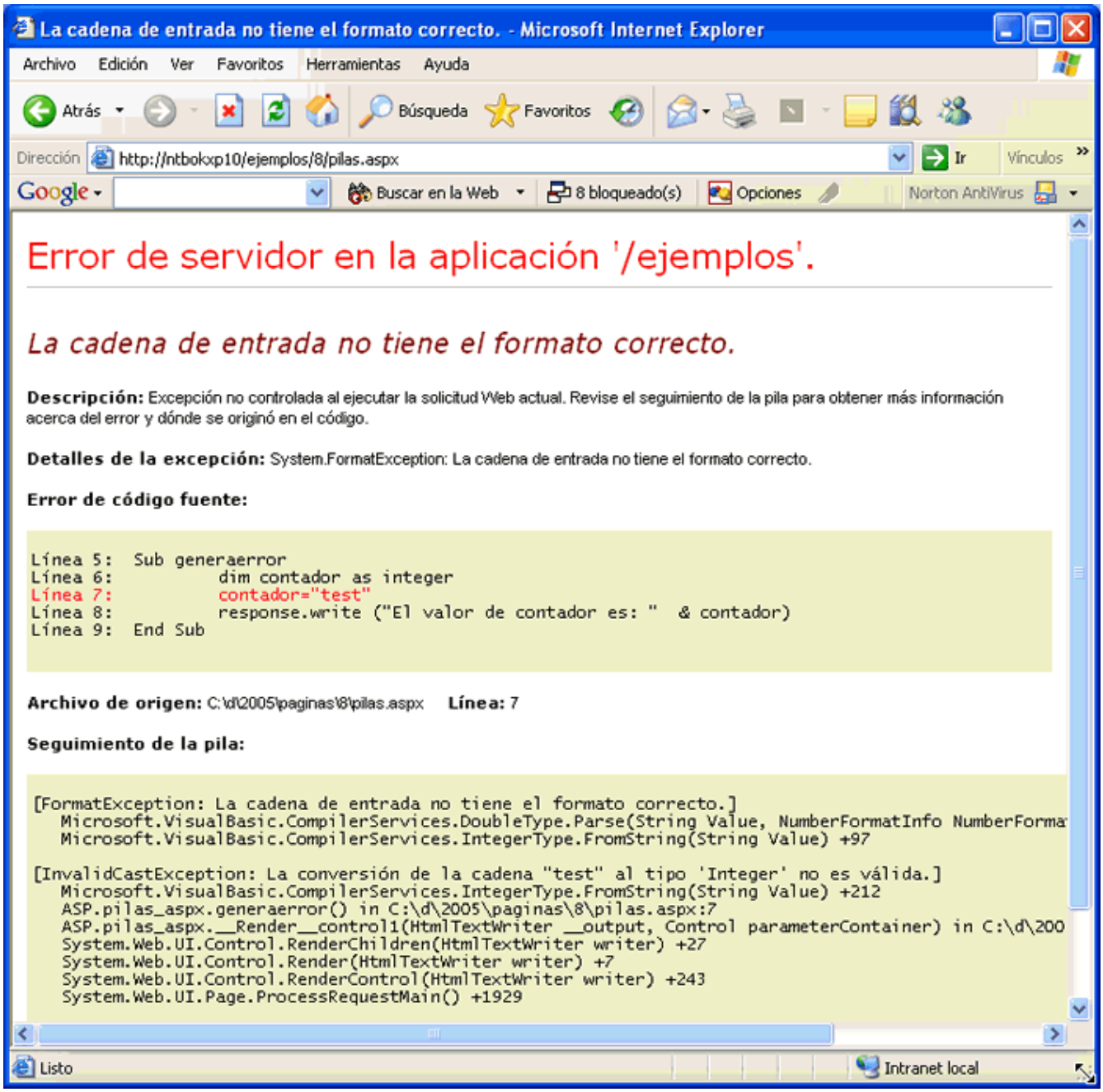
Muy importante Estos mensajes de error tan detallados no se muestran a todos los usuarios (menos mal). Sólo se muestran cuando se está ejecutando localmente, es decir en el mismo equipo que está instalado el servidor Web. Así que todo esto es una ayuda para el desarrollador.

Error de llamada

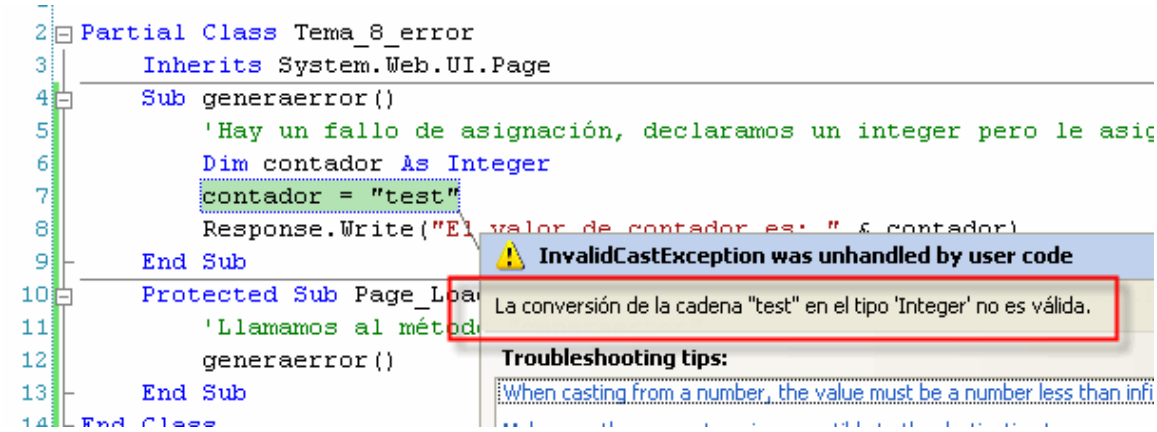
Sigamos con otro ejemplo, en este caso crea una página llamada "error.aspx" con este código en el evento Load de la página y con el procedimiento llamado "generaerror":

```
Sub generaerror()
    'Hay un fallo de asignación, declaramos un integer pero le asignamos un string
    Dim contador As Integer
    contador = "test"
    Response.Write("El valor de contador es: " & contador)
End Sub
Protected Sub Page_Load(ByVal sender As Object, ByVal e As System.EventArgs) Handles Me.Lo
    'Llamamos al método "generaerror"
    generaerror()
End Sub
```

Ejecuta, y nos puede dar dos mensajes distintos. Si está ejecutando la página desde un servidor tendrás



Y si la ejecutas dentro del IDE:



En los dos casos es lo mismo, se ha producido un error de llamada porque se produce cuando llamamos a un procedimiento, en este caso el fallo es obvio, le hemos asignado una variable de tipo entera una cadena de caracteres. Si te fijas en la pantalla, ésta tiene varias partes: Descripción, Detalles de la excepción, Error de código fuente, ... Una de ellas es "Seguimiento de la pila" o "StackTrace" si estás en inglés que contiene la pila de llamadas donde dice que el valor que se está asignando a una variable entera no es válido.

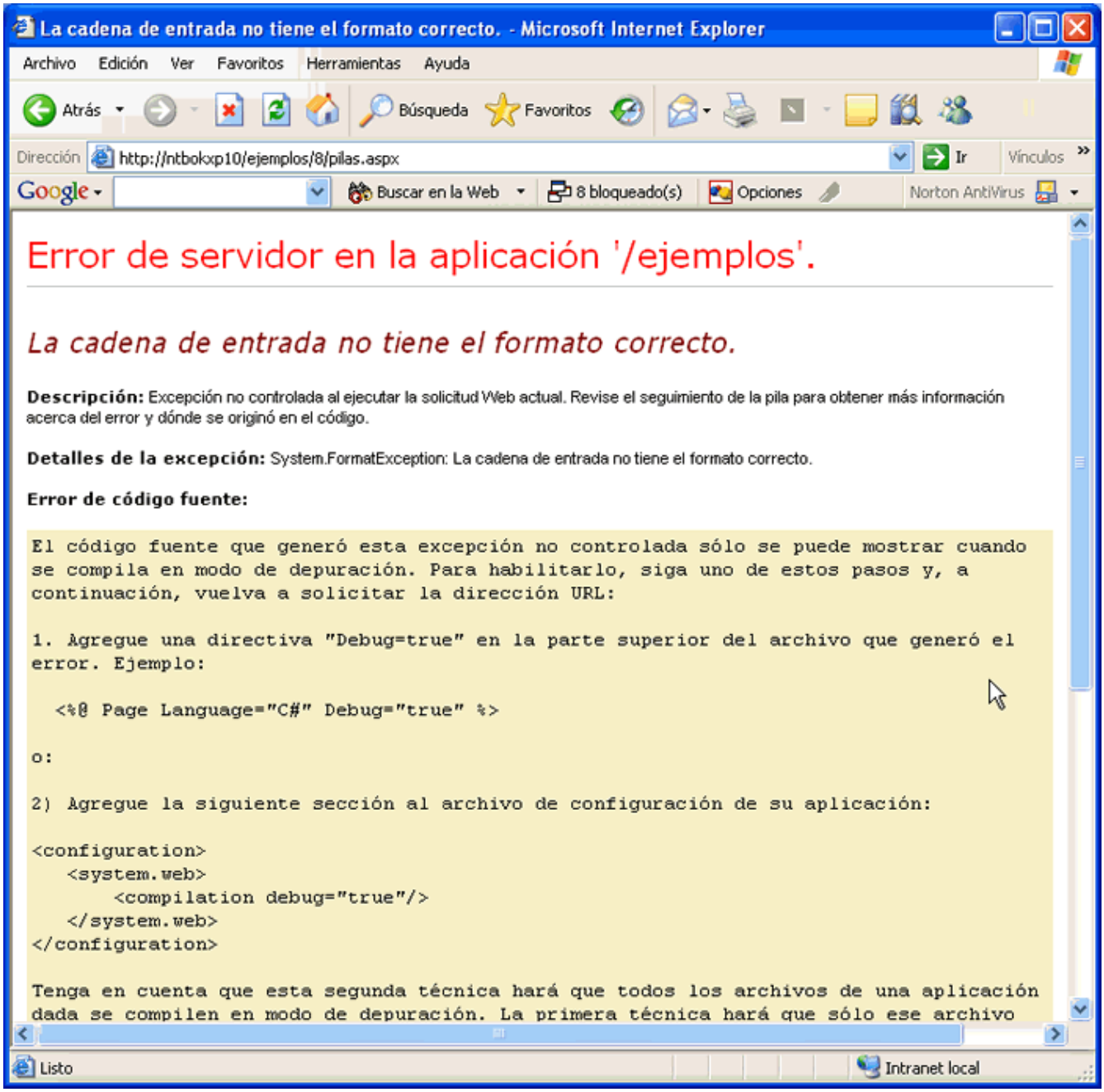
Si, ya se es un poco lioso leer semejante ladrillo, pero es fundamental para encontrar hasta el último detalle del error y su origen. Te aseguro que te evitará muchas horas de investigación intentado localizar el error cuando te lo está mostrando en este texto... Si te fijas en la pila de llamadas podemos ver los procedimientos que producen la excepción o error

Nota Como habrás visto en este tema los errores en .NET se llaman excepción así que si se produce una excepción quiere decir que se produce un error. Sigamos...

Si finalmente cambiamos nuestro código por "123" funcionará porque aunque estamos poniendo "123" entre comillas y lo va a tratar como una cadena de caracteres, ASP.NET si consigue traducirlo a valor numérico.

4.1 Modo depuración

Si activamos la depuración el origen del error se muestra como parte del mensaje de error. Por ejemplo elimina la directiva de "debug=true" del ejemplo anterior. Al ejecutarse la página en lugar de mostrarnos la anterior pantalla nos muestra:



Donde nos indica que se ha producido una excepción (error) y que los detalles sólo se pueden mostrar si "debug=true" así que obedecemos a su sugerencia y activamos la depuración. Por lo tanto para activar la depuración incluiremos esta directiva en nuestras páginas:

4.2 Interceptación de errores

Ya tenemos varias formas de controlar o mejorar nuestro código, ahora veamos la mas potente y que te recomiendo leas muy detenidamente. La interceptación de errores se puede realizar de varias formas veamos...

Interceptación de errores no estructurada

Este método era el antiguo sistema de ASP y de las versiones anteriores de Visual Basic. Se basaban en unas instrucciones muy sencillas:

- On error goto linea: se produce un error vete a la línea "linea"
- On error resume next: si se produce un error pásalo por alto y sigue en la siguiente línea
- On error goto 0. Desactiva la interceptación de errores.

Era un sistema muy limitado y poco eficaz así que lo pasamos por alto para centrarnos en el sistema que ofrece .NET que es mucho mas adecuado para nuestros desarrollos.

Interceptación de errores estructurado

Simplemente significa que vamos a interceptar los errores mediante una estructura. Agrupamos las líneas de código juntas y crearemos distintos controladores para manejar los distintos errores en esos grupos. Tranquilo es mucho mas sencillo de lo que parece. Esta es la secuencia de eventos que se darán en nuestro control estructurado de errores:

1. Ejecutamos una línea de código de un grupo de líneas
2. Puede no generar un error o puede producir uno o varios
3. Si se produce un error pasamos a ejecutar un grupo de líneas de código según el error generado. Si no hay error no se ejecutan estas líneas.
4. Podemos definir un controlador de errores genérico para no tener que controlar cada uno de los errores que se puedan producir.

Dentro de .NET encontramos dos aspectos de los errores: 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.

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 será algo 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

que sirve cada uno de ellos:

Si usamos Catch, esa parte 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 usemos Finally, tendremos que tener en cuenta de 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.

Aclaremos este punto, ya que puede parecer extraño. 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

' se 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. Seguro que pensarás (o al menos así deberías hacerlo), que eso es claro y evidente. Si usáramos On Error Resume Next, el código podría ser algo como esto (instrucciones de Vb6):

```
Dim i, j As Integer

On Error Resume Next

i = 10

j = 0

i = i \ j

' se continúa después del bloque de detección de errores
```

Bueno, si nos basamos en estos dos ejemplos, ambos hacen lo mismo: Si se produce un error se continúa justo por donde está el comentario ese que dice ' se continúa... pero... cuando se usan los bloques Try... Catch... el tratamiento de errores es diferente a cuando se usa el "antiguo" On Error Resume Next, para que nos entendamos:

```
Dim i, j As Integer

Try
```

```
        i = 10
        j = 0
        i = i \ j
        texto.text="el nuevo valor de i es: " & i
    Catch
        ' nada que hacer si se produce un error
    End Try
    ' se continúa 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, (sí, la línea `i = i \ j` es la que produce el error, 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...), y la otra después del bloque Try... Catch.

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), y continúa por el código del bloque Catch, que en este caso no hay nada, así que, busca un bloque Finally, y si lo hubiera, ejecutaría ese código, pero como no hay bloque Finally, continúa por lo que haya después de End Try.

Vamos a repetirlo una vez mas... mostramos 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.

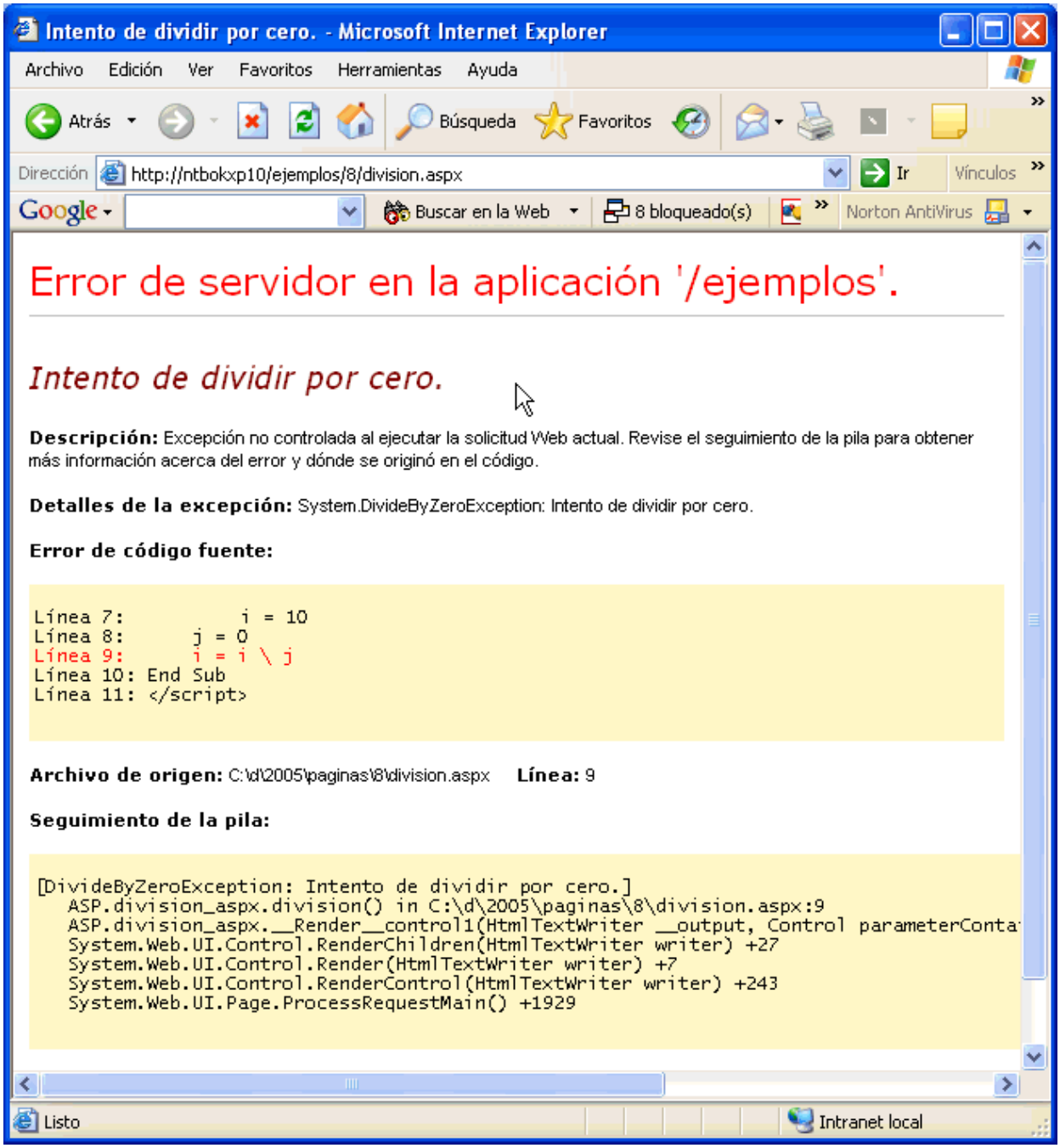
Si no se produce ningún error, se continuaría con todo el código que haya en el bloque Try y después se seguiría, (si lo hubiera), con el bloque Finally y por último con lo que haya después de End Try.

Antes de continuar vamos a hacer este sencillo ejemplo para ver que pasa en nuestro programa si no interceptamos las posibles fuentes de errores.

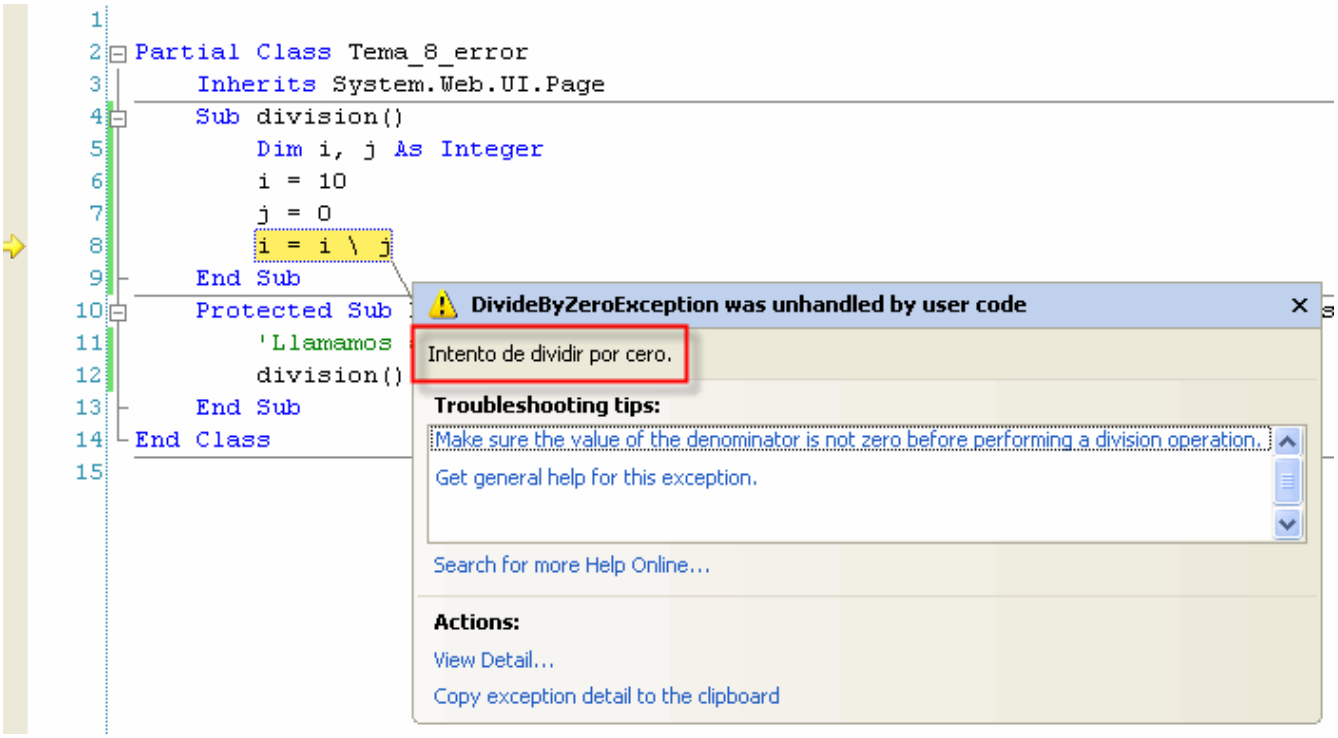
Crea una página con este código. Lo crearemos en la página de extensión .vb y tendrá un procedimiento llamado "division" y una llamada desde el evento Load de la página:

```
Sub division()
    Dim i, j As Integer
    i = 10
    j = 0
    i = i \ j
End Sub
Protected Sub Page_Load(ByVal sender As Object, ByVal e As System.EventArgs) Handles Me.Load
    'Llamamos al método "division"
    division()
End Sub
```

Ahora ejecuta la página:



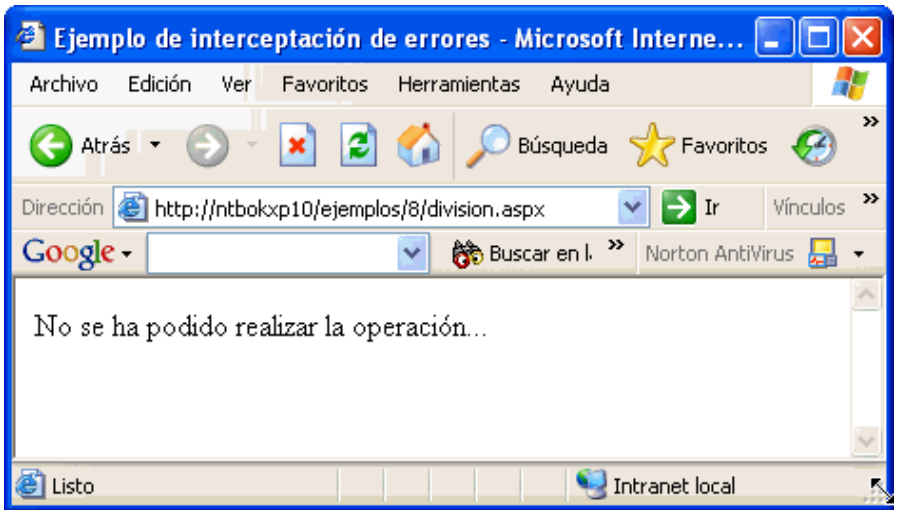
Se ha producido lo esperado: un error en tiempo de ejecución al producirse una división por cero. Dependiendo de la configuración que tengas puede que el error te lo indique el mismo entorno de desarrollo:

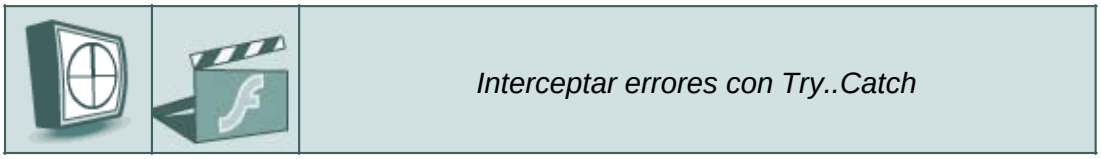


En el primer caso el error se dio en ejecución de la página como si la tuviéramos ya en el servidor definitivo y en este segundo caso se ha dado por ejecutarla con el "depurador" de nuestro entorno de desarrollo. El resultado es el mismo la página nos ha fallado. Ahora haz estas pequeñas modificaciones:

```
Sub division()
    Dim i, j As Integer
    i = 10
    j = 0
    Try
        i = i \ j
        response.write("el valor de i es " & i)
    Catch
        response.write("No se ha podido realizar la operación...")
    End Try

End Sub
Protected Sub Page_Load(ByVal sender As Object, ByVal e As System.EventArgs) Handles Me.Load
    'Llamamos al método "division"
    division()
End Sub
```

Si ejecutas ahora la página ya no se producirá el error porque se ha interceptado una posible fuente de errores. De ahí que antes al principio de este tema comentaba que "intentaremos" (try) ejecutar operaciones susceptibles de producir errores:





Detalles de la clase Exception

Para facilitar el uso del control estructurado de excepciones, .NET 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; 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.

Cuando interceptamos una excepción tendremos un objeto que representa el tipo de error. Este objeto está basado en una de las muchas clases de System.Exception que incluye muchas clases como "DivideByZeroException", "ArithmeticException", "IOException", "SecuryException", ... el tener tantas clases nos permite como ya has visto antes personalizar el error porque sabemos exactamente el fallo que se ha producido. Si no queremos utilizar estas clases para no tener que escribir demasiado código para tratar todas las opciones posibles no pasa nada, ponemos los datos genéricos del error y listos.

Intercepción de errores anidada

Cuando se produce una excepción .NET intenta encontrar una instrucción Catch que coincida con el método actual. Si el código no está en el bloque local de control de excepciones o la excepción no coincide con las interceptadas, .NET mueve un nivel la pila de llamadas buscando otros controladores de eventos. Fíjate en este ejemplo:

```
Sub Page_Load

Try

    Divide_numeros (5,0)

Catch err as DivideByzeroException

    'Informe...

End Try

End Sub

Private Function Divide_numeros (Byval numero as Decimal, Byval Divisor as Decimal) as Decimal

    Return (numero/Divisor)

end function
```

Aunque la excepción no se produce estrictamente dentro del "try... catch" queda interceptada porque la llamada se hace dentro de ella. Esto está bastante bien porque podemos poner a ejecutar en un "try" una función o procedimiento complejo y así asegurarnos de que si falla no nos afectará al programa y le devolverá una página de error al usuario.

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 On Error Resume Next, se producirá un error al abrir el archivo, pero como le hemos dicho que queremos continuar, seguirá e intentará guardar información en dicho archivo, pero como no puede guardar esa información, se volverá a producir un error, pero como continúa aunque haya errores, ni nos enteramos, pero se estarán produciendo errores de continuo y "no pasará nada" porque así lo hemos querido.

Bueno, esa es la intención inicial de haberle puesto un de Resume Next. Pero si lo que teníamos que guardar en el fichero ese eran 10000 líneas de texto, seguramente habremos desperdiciado un tiempo "precioso", además de que puede que el usuario ni se haya percatado de que ha estado esperando un buen rato y al final no se ha guardado la información.

Si en lugar de usar On Error Resume Next (e incluso On Error Goto nnn), hemos optado por usar Try... Catch, todo código lo escribiríamos dentro del bloque Try y si se produce un error, avisaríamos al usuario de que se ha olvidado de poner el disquete, (por ejemplo), y le podríamos dar la oportunidad de volver a intentarlo...

Como te he comentado antes, el bloque 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 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 disco en la disquetera, 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

    i=10

    j=0

    i=i/j    '

Catch ex As DivideByZeroException

    texto.text="ERROR: división por cero"

Catch ex As OverflowException

    texto.text="ERROR: de desbordamiento (número demasiado grande)"
```

```
Catch ex As Exception

    texto.text="Se ha producido el error: " & ex.Message

End Try

'
```

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, te comentaré que debes tener cuidado de poner el tipo genérico al final, (o el que no tenga ningún tipo de "error a capturar" después de Catch), ya que el CLR siempre evalúa los tipos de errores a detectar empezando por el primer Catch y si no se amolda al error producido, comprueba el siguiente, así hasta que llegue a uno que sea adecuado al error producido, y 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 adecua 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 te fijas, verás 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 ese 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:

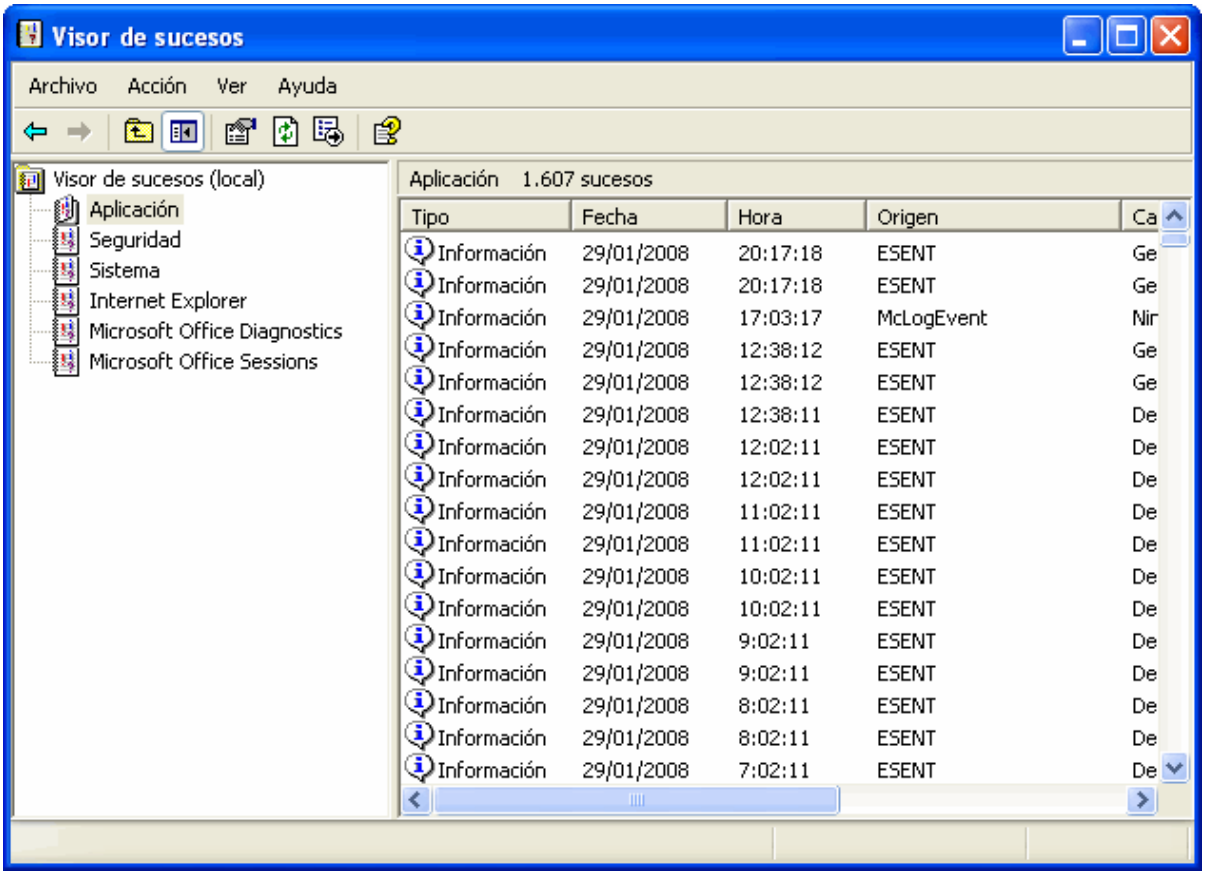
```
Dim i, j As Integer

Dim s As String

'

Try

    i=10

    j=0

    i=i/j    '

Catch ex As DivideByZeroException

    Response.Write("ERROR: división por cero")

Catch ex As OverflowException

    Response.Write("ERROR: de desbordamiento (número demasiado grande)")

End Try
```

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

4.3 Mas información de errores

Con lo que hemos visto tenemos suficiente para nuestro curso de iniciación a ASP.NET pero tenemos mas opciones para el control de errores que te voy a enumerar para que puedas avanzar con mas temas:

- Definir excepciones personalizadas. Puedes definir tus propios objeto de excepciones para representar un tipo de error personalizado. Para realizar esto necesitas crear una instancia de la excepción apropiada y después utilizar la instrucción "Thow"
- Registro de excepciones. Esto es una curiosa novedad que nos va a permitir almacenar los eventos producidos en Windows de la misma forma que los eventos del sistema:



Para escribir en el registro debemos hacer como este ejemplo:

```
sub boton_ejecutar ()

Try

    Dim a, b , resultado as Decimal

    A=10

    B=0

Resultado=A/B

Catch err as Exception
```

```
Dim Log as New EventLog()

Log.Source="Pagina_Division"

Log.WriteEntry (err.Message,EventLogEntryType.Error)

End Try

En Sub
```

Páginas de error

Las páginas de error nos muestran mucha información sobre el error, pero no es información relevante para el usuario así que habitualmente no les mandaremos un error detallado sino un mensaje genérico. Los errores detallados como los que has visto nos han salido gracias a que estamos ejecutando las páginas en nuestro entorno de desarrollo que es nuestro propio equipo con el servidor Web que nos proporciona ASP.NET

Si ejecutamos las páginas desde un servidor "normal" de IIS no obtendremos detalles del error. Como desarrolladores puede que alguna rara vez cometamos algún fallo en nuestro código y claro, como no nos muestra los detalles no podemos tomar las medidas oportunas. Para que el servidor web mande los detalles a otros ordenadores que no sea el que ejecuta la página debemos activar la siguiente clave en el fichero de configuración:

```
<authentication mode="Windows"/>
<!--
    The <customErrors> section enables configuration
    of what to do if/when an unhandled error occurs
    during the execution of a request. Specifically,
    it enables developers to configure html error pages
    to be displayed in place of a error stack trace.

<customErrors mode="RemoteOnly" defaultRedirect="GenericErrorPage.htm">
    <error statusCode="403" redirect="NoAccess.htm" />
    <error statusCode="404" redirect="FileNotFound.htm" />
</customErrors>

-->
<customErrors mode="RemoteOnly" />
```

Las posibilidades de los modos de error son:

- "RemoteOnly". Se muestran errores genéricos a los usuarios. Las páginas con el error detallado sólo se muestran en el equipo que ejecuta el servidor Web. Es el valor predeterminado.
- "Off". Se muestran las páginas con los errores a todos los usuarios. Debe estar poco tiempo activado, lo justo para localizar y corregir el error. Mas que nada porque puede mostrar código sensible al usuario.
- "On". Se desactiva para todos la presentación de detalles de los errores.

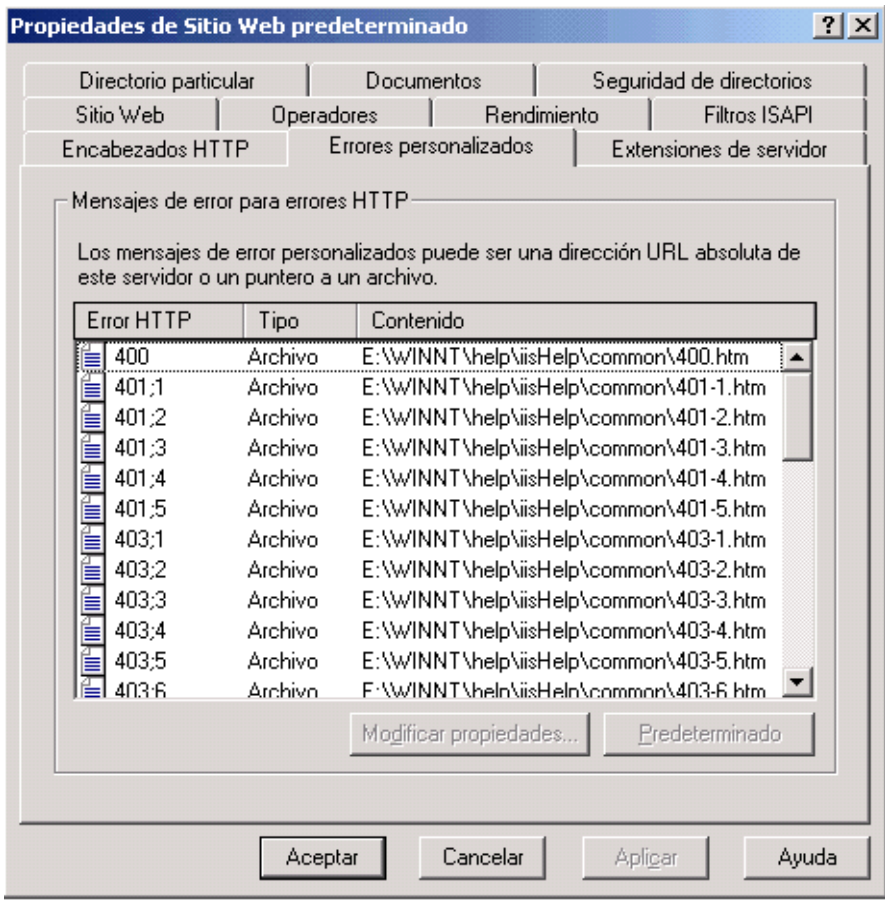
Como has visto en el código del fichero de configuración, podemos redirigir al usuario a una página genérica con:

```
<customErrors mode="RemoteOnly" defaultRedirect="Pagina_error_generica.htm">
```

La propiedad defaultRedirect indica la página que mostrará al usuario cuando se produzca cualquier tipo de error. Ahora las posibilidades serán:

- Si se produce un error ASP.NET, éste redirigirá al usuario a la página "Pagina_error_generica.htm"
- Si se produce un error no interceptada y el modo es "On" redirigirá al usuario a la página "Pagina_error_generica.htm". Los usuarios remotos nunca verá la página de error genérica ASP.NET
- Si se produce un error no interceptada y el modo es "Off" se mostrará una página de error ASP.NET
- Si se produce un error no interceptada y el modo es "RemoteOnly", el comportamiento dependerá del origen del error. Si es en el equipo local se mostrará la página de error de ASP.NET de lo contrario redirigirá al usuario a la página "Pagina_error_generica.htm"

Además podemos crear páginas personalizadas. Si recuerdas en el capítulo 2 podíamos hacer algo parecido pero a nivel de configuración del Internet Information Server:



Ahora podemos decirle en la configuración que si se da algún error especial redirija a una página en particular, al igual que se puede hacer de forma global en el servidor Web:

```
<customErrors mode="RemoteOnly">
    <error statusCode="403" redirect="Sin_acceso.htm" />
    <error statusCode="404" redirect="Fichero_no_encontrado.htm" />
</customErrors>
```

4.5 Traza de páginas

Finalmente la última opción para realizar seguimientos en las páginas para obtener información de su ejecución y así encontrar errores es la traza. Muchas veces metemos código extra para ayudarnos a depurar la página mientras estamos en desarrollo. Pero cuando está ya en marcha no tenemos posibilidad de ver esa información aunque si podemos activar la traza para que nos proporcione información exhaustiva de lo que está pasando.

La forma mas sencilla de activa la traza es poniendo una directiva en la página que queremos analizar:

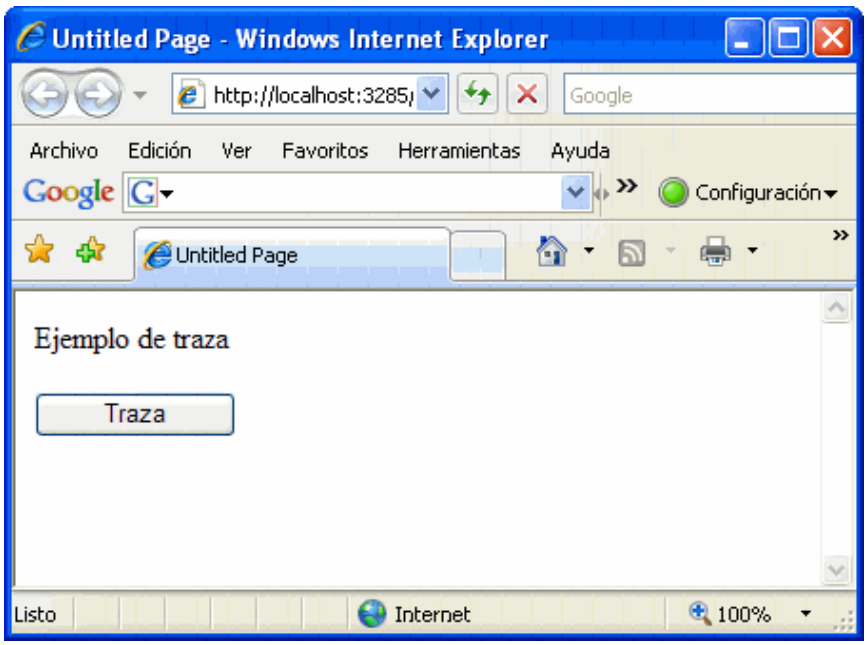
```
<%@ Page Trace="True" ... %>
```

También podemos habilitarlo con el objeto de AP.NET "Trace":

```
Sub Page_Load ()

Trace.IsEnabled=True

End Sub
```

Información de la traza

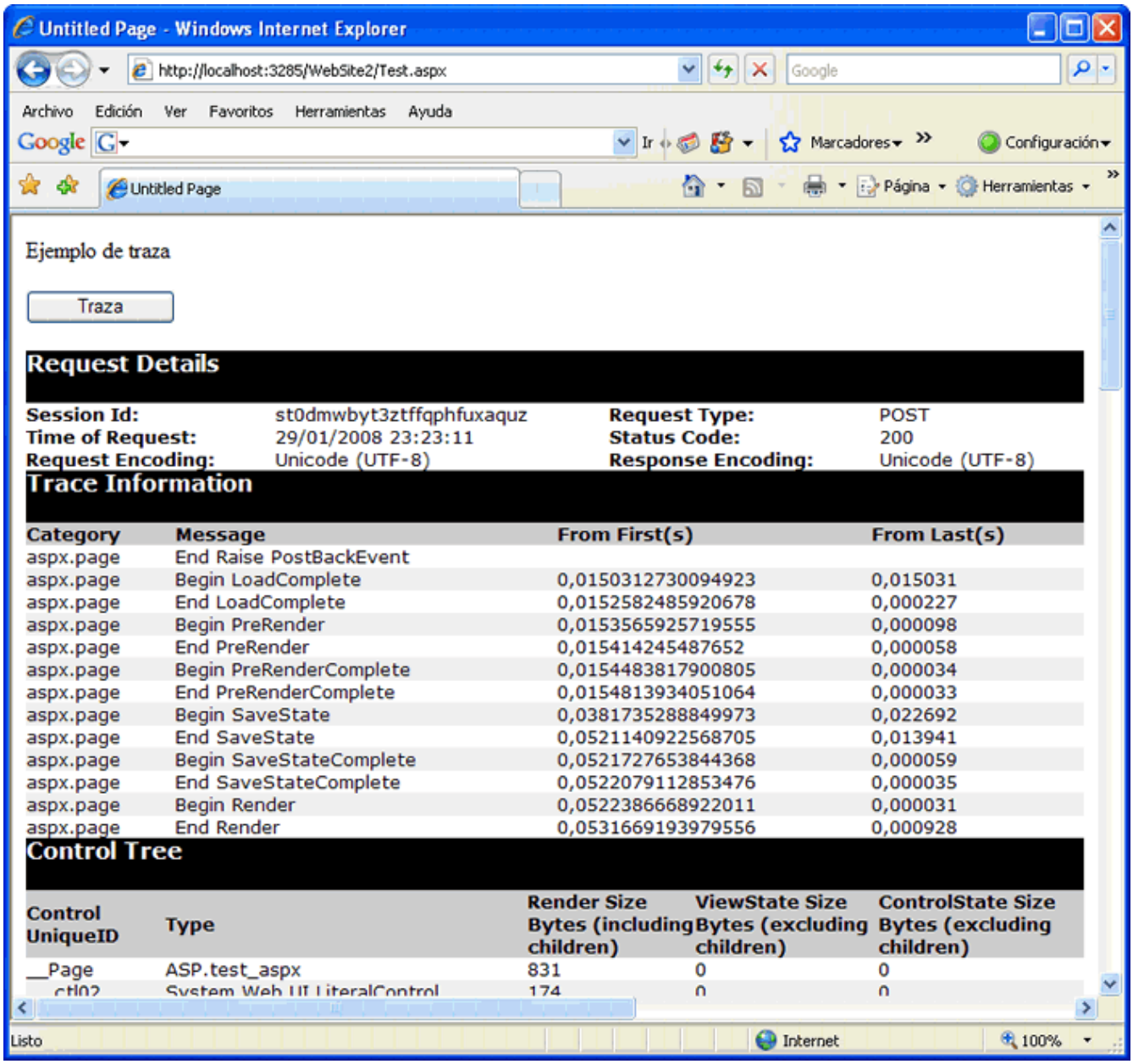
Esta página simplemente muestra un texto pero al pulsar el botón en su evento clic activamos la traza:



La activaremos con la propiedad que hemos visto antes:

```
Protected Sub Button1_Click(ByVal sender As Obj
    Trace.IsEnabled = True
End Sub
```

Con este resultado:



Si exploras esta extensa página verás varias categorías con mucha información, quizás mas de la que puedas interpretar pero mejor que sobre información que nos pueda ayudar. Veamos que secciones tenemos:

- En la primera sección tenemos los detalles de la solicitud con valores como el identificador de la sesión que ha hemos visto en otra sección de este curso.
- En la sección de "Trace Information" tenemos información de los distintos estados por los que ha pasado la página antes de enviarse al cliente.
- En al árbol de control (Control Tree) nos muestra todos los controles que tiene nuestra página: botones, literales, ... con información del tamaño y del tamaño de su estado si lo tiene
- Las secciones de estado de la sesión y de la aplicación nos proporcionarían las variables de estos dos tipo que tendríamos en nuestra página. Información muy interesante porque podríamos ver si está recogiendo estas variables. Lo mismo si ha proceso "cookies"
- La colección "Headers nos muestra la información del cliente enviada al servidor, que ya hemos visto en otros capítulos. Por ejemplo vemos el idioma del cliente.

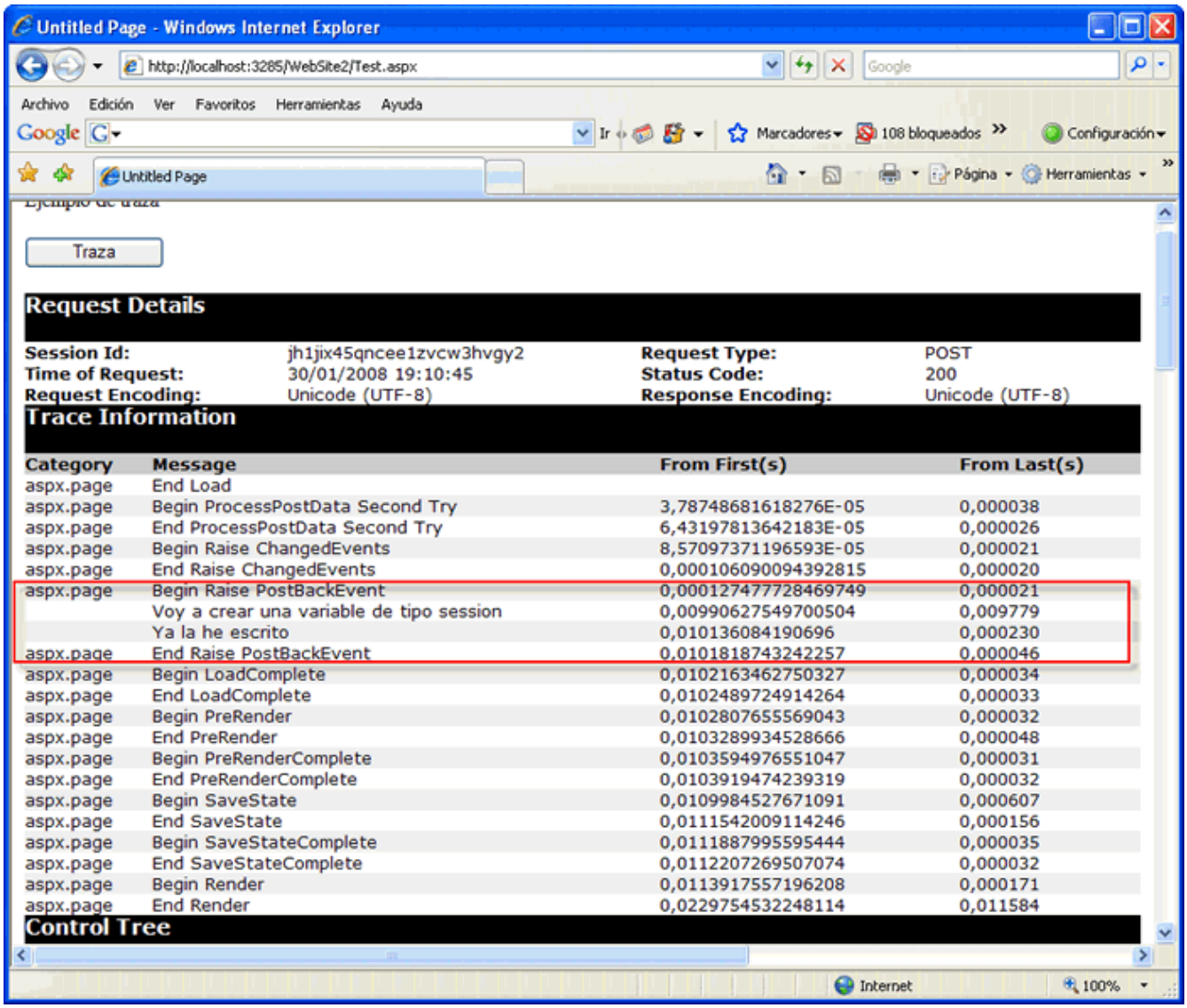
- "form collection" contiene toda la información de la colección de objetos que conforman el formulario. Muy útil, que en este caso solo tiene los estados de los controles que hemos puesto.
- "Server variables". Nos muestra todas las variables de servidor con sus valores. También lo vimos capítulos atrás y es una forma estupenda de comprobar si nuestro servidor está funcionando como debe.

Mas información en las trazas

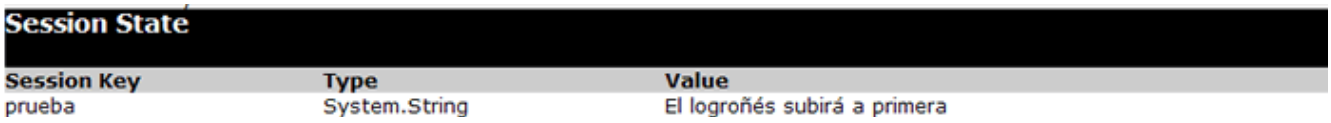
Podemos enviar información a la traza para ver cómo se está ejecutando utilizando el método "write" del objeto "trace", por ejemplo, vamos a poner un botón a nuestra página de prueba con este código:

```
Partial Class Test
    Inherits System.Web.UI.Page

    Protected Sub Page_Load(ByVal sender As Object, ByVal e As System.EventArgs) Handles Me.Load
        Trace.IsEnabled = True
    End Sub
    Protected Sub Button1_Click(ByVal sender As Object, ByVal e As System.EventArgs) Handles Button1.Click
        Trace.Write("Voy a crear una variable de tipo session")
        Session("prueba") = "El logroñés subirá a primera"
        Trace.Write("Ya la he escrito")
    End Sub
End Class
```

Ojo, la traza la he activado en el "Page_load". Ahora ejecuta la página y obtendrás la traza de ella. Pulsa ahora el botón y observa la traza:



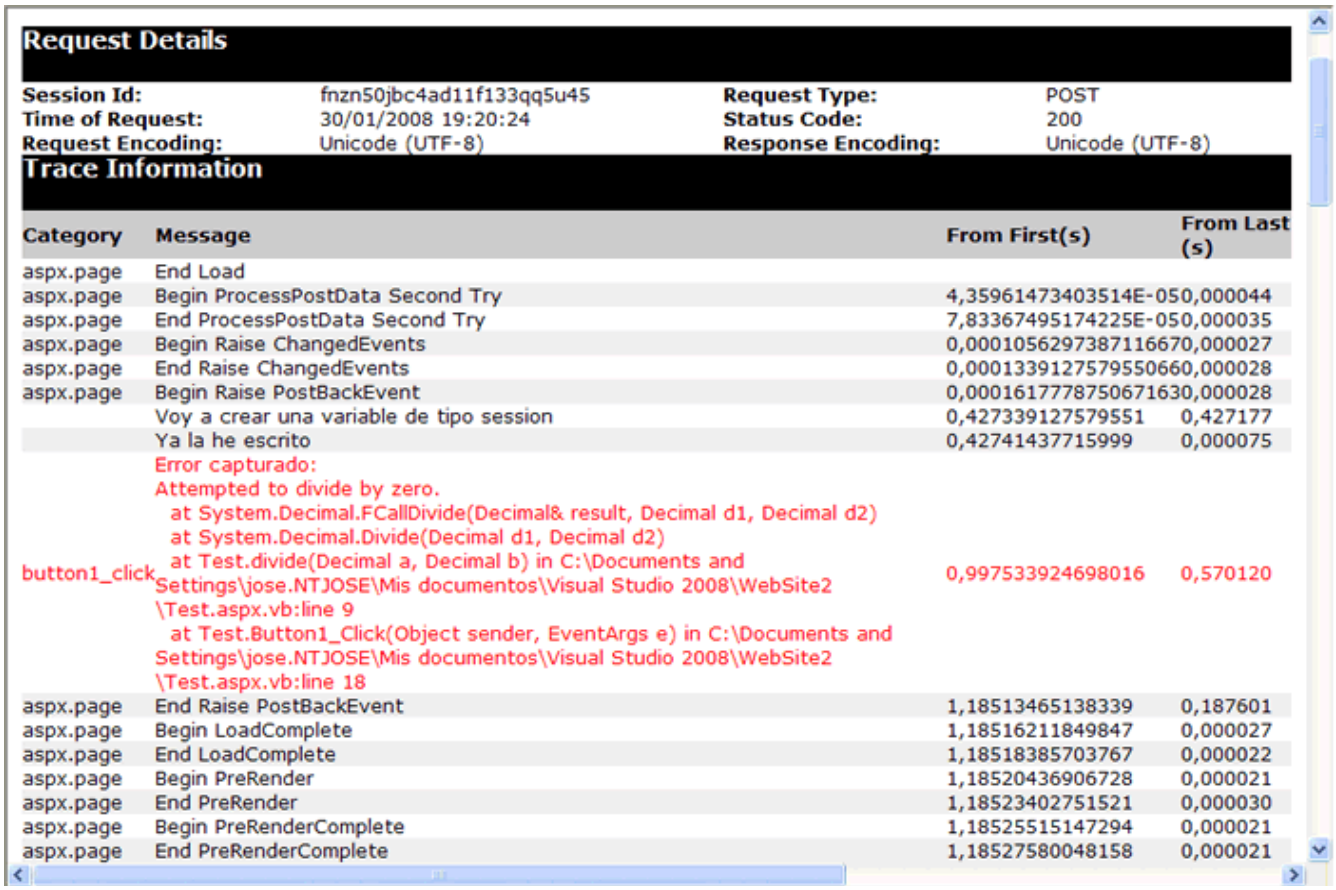
Nos muestra el código que se ha ejecutado al dispararse el evento que es la escritura de estos literales de prueba y además mas abajo veremos la variable de tipo "session" que hemos creado:



También podemos hacer que nos monitorice la ejecución de código como este:

```
Protected Sub Page_Load(ByVal sender As Object, ByVal e As System.EventArgs) Handles Me.Load
    Trace.IsEnabled = True
End Sub
Protected Function divide(ByVal a As Decimal, ByVal b As Decimal) As Decimal
    Return a / b
End Function

Protected Sub Button1_Click(ByVal sender As Object, ByVal e As System.EventArgs) Handles Button1.Click
    Trace.Write("Voy a crear una variable de tipo session")
    Session("prueba") = "El logroñés subirá a primera"
    Trace.Write("Ya la he escrito")
    'Llamamos a una función interceptando los errores
    Try
        Divide(5, 0)
    Catch ex As Exception
        Trace.Warn("button1_click", "Error capturado:", ex)
    End Try
End Sub
```

Donde le decimos que nos monitorice el error producido dentro del "try":



Request Details

Session Id:	fnzn50jbc4ad11f133qq5u45	**Request Type:**	POST
Time of Request:	30/01/2008 19:20:24	**Status Code:**	200
Request Encoding:	Unicode (UTF-8)	**Response Encoding:**	Unicode (UTF-8)

Trace Information

Category	Message	From First(s)	From Last (s)
aspx.page	End Load		
aspx.page	Begin ProcessPostData Second Try	4,359614734035 14E-05	0,000044
aspx.page	End ProcessPostData Second Try	7,833674951742 25E-05	0,000035
aspx.page	Begin Raise ChangedEvents	0,000105629738711 6670	0,000027
aspx.page	End Raise ChangedEvents	0,000133912757955 0660	0,000028
aspx.page	Begin Raise PostBackEvent	0,000161777875067 1630	0,000028
	Voy a crear una variable de tipo session	0,427339127579551	0,427177
	Ya la he escrito	0,42741437715999	0,000075
button1_click	Error capturado: Attempted to divide by zero. at System.Decimal.FCallDivide(Decimal& result, Decimal d1, Decimal d2) at System.Decimal.Divide(Decimal d1, Decimal d2) at Test.divide(Decimal a, Decimal b) in C:\Documents and Settings\jose.NTJOSE\Mis documentos\Visual Studio 2008\WebSite2 \Test.aspx.vb:line 9 at Test.Button1_Click(Object sender, EventArgs e) in C:\Documents and Settings\jose.NTJOSE\Mis documentos\Visual Studio 2008\WebSite2 \Test.aspx.vb:line 18	0,997533924698016	0,570120
aspx.page	End Raise PostBackEvent	1,18513465138339	0,187601
aspx.page	Begin LoadComplete	1,18516211849847	0,000027
aspx.page	End LoadComplete	1,18518385703767	0,000022
aspx.page	Begin PreRender	1,18520436906728	0,000021
aspx.page	End PreRender	1,18523402751521	0,000030
aspx.page	Begin PreRenderComplete	1,18525515147294	0,000021
aspx.page	End PreRenderComplete	1,18527580048158	0,000021

¿no está mal? Como ves es una herramienta muy poderosa para poder realizar un seguimiento perfecto a nuestra página. Todo este código de traza no hace falta quitarlo porque simplemente desactivaremos la traza y la activaremos cuando necesitemos rastrear algún error en nuestra página.

5. Funciones de VB.NET

Como ves este mundo del ASP.NET es un poco denso ya que necesitamos aprender VB.NET para tener una buena base. Ahora nos toca revisar las funciones del lenguaje mas importantes. Sirva este contenido como una referencia de las funciones que utilizaremos en el 99% de nuestros programas...

Así que no te "aprendas" esta parte hasta el final del tema, repásala, y tómala como referencia indispensable para encontrar las funciones mas utilizadas en Visual Basic.NET. Con esto termina nuestro aprendizaje de Visual Basic.Net porque pasaremos en el tema siguiente a las bases de datos. Pero insisto, lee bien este resumen de funciones porque pon un lado serán tu referencia del lenguaje a partir de ahora y por otro te servirán para pasar el test de final de lección :-)

Las funciones del lenguaje nos van a proporcionar todas las herramientas necesarias para trabajar con los objetos y los formularios. Una vez que sepamos como acceder a los formularios y objetos necesitaremos utilizar funciones para preparar, procesar o producir un resultado. Tenemos decenas de funciones que serán muy importantes pero entre todas hay varios grupos que utilizaremos muy a menudo. Para eso está este capítulo para describir con ejemplos las mas importantes.

Como hemos comentado en otras ocasiones a lo largo del curso, no hay que aprendérselas de memoria pero si saber que existen y cómo funcionan. Luego cuando se nos plantee un problema podremos acudir a esta referencia para ayudarnos e resolverlo.

5.1 Cómo utilizarlas en el código

Al usar una función de una fórmula, escribiremos el nombre de la función y le proporcionaremos los argumentos requeridos. En toda esta labor nos ayudará permanentemente la tecnología "intelisense" de la escritura del código, así que será una de las mejores formas de ayuda para saber la sintaxis, sobrecarga que pueda tener y el tipo de datos de los argumentos o parámetros.

Las clases de funciones son: matemáticas, resumen, finanzas, cadena, fecha/hora, rango de fechas, matriz, conversión de tipos, accesos directos de programación, tiempo de evaluación, estado de impresión, propiedades de documentos y funciones adicionales. Existen también algunas funciones específicas a las fórmulas de formato condicional.

Comenzaremos con las mas utilizadas... en todos los casos veremos un ejemplo de como funciona así que tranquilo.

5.2 Funciones de conversión de tipos

Una expresión de conversión convierte una expresión en un tipo determinado. Existen palabras clave de conversión específicas que convierten expresiones en tipos primitivos; también existen dos palabras clave generales de conversión, **CType** y **DirectCast**, que convierten una expresión en cualquier tipo.

Es decir, por un lado tendremos una colección de funciones una para cada tipo de datos y otra general (CType) que convierte a cualquier tipo. Esta función la hemos visto en algún ejemplo y como permite convertir cualquier formato es mas fácil de acordarse que las otras...

DirectCast es un caso especial, porque las conversiones de **Object** a cualquier otro tipo se realizan por conversión directa en orden jerárquico (todos los comportamientos especiales de conversión de **Object** se omiten). Cuando se convierte una expresión de tipo **Object** cuyo tipo en tiempo de ejecución es un tipo de valor primitivo, **DirectCast** inicia una excepción **System.InvalidTypeException** si el tipo especificado no es el mismo que el tipo en tiempo de ejecución de la expresión. No obstante, si el tipo especificado y el tipo de tiempo de ejecución de la expresión son los mismos, el rendimiento en tiempo de ejecución de **DirectCast** es mejor que el de **CType**.

Si no existe conversión del tipo de la expresión en el tipo especificado, se produce un error. En caso contrario, la expresión se clasifica como un valor y el resultado es el valor producido por la conversión.

Estas funciones se compilan en línea, es decir, el código de conversión forma parte del código que evalúa la expresión. La ejecución es más rápida, ya que no existen llamadas a un procedimiento que realiza la conversión. Cada función convierte una expresión a un tipo de datos específico.

- `CBool (expresión)`
- `CByte (expresión)`
- `CChar (expresión)`
- `CDate (expresión)`
- `CDbl (expresión)`
- `CDec (expresión)`
- `CInt (expresión)`
- `CLng (expresión)`
- `CObj (expresión)`
- `CShort (expresión)`
- `CSng (expresión)`
- `CStr (expresión)`

La expresión es un dato obligatorio y debe ser una expresión numérica o una cadena de caracteres (string).

Tipos devueltos

El nombre de la función determina el tipo devuelto, como muestra la siguiente tabla:

Nombre de la función	Tipo de valor devuelto	Intervalo de valores del argumento expression
CBool	Boolean	Cualquier expresión numérica o de cadena (String) válida.
CByte	Byte	0 a 255; las fracciones se redondean.
CChar	Char	Cualquier expresión String válida, valores comprendidos entre 0 y 65535.
CDate	Date	Cualquier representación válida de fecha y hora.
CDbl	Double	-1,79769313486231E+308 a -4,94065645841247E-324 para valores negativos; 4,94065645841247E-324 a 1,79769313486231E+308 para valores positivos.
CDec	Decimal	

		+/-79.228.162.514.264.337.593.543.950.335 para números a partir de cero, es decir, números sin decimales. Para números con 28 decimales, el rango es +/-7.9228162514264337593543950335. El menor número distinto de cero es 0,0000000000000000000000000001.
CInt	Integer	-2.147.483.648 a 2.147.483.647; las fracciones se redondean.
CLng	Long	-9.223.372.036.854.775.808 a 9.223.372.036.854.775.807; las fracciones se redondean.
CObj	Object	Cualquier expresión válida.
CShort	Short	-32.768 a 32.767; las fracciones se redondean.
CSng	Single	De -3,402823E+38 a - 1,401298E-45 para valores negativos; de 1,401298E-45 a 3,402823E+38 para valores positivos.
CStr	String	Los valores devueltos para CStr dependen del argumento expression. Mira la siguiente tabla

En la tabla siguiente se describen los valores devueltos por la función **CStr** en varios tipos de *expression*

Si el tipo de *expression* es	**CStr devuelve**
Boolean	Cadena que contiene "True" o "False".
Date	Cadena que contiene un valor **Date** (fecha y hora) en el formato de fecha corta del sistema.
Numeric	Cadena que representa el número.

Comentarios

Si el argumento expresión que se pasa a la función está fuera del intervalo de valores del tipo de datos al que se va a convertir, se produce un error, así que tendremos que tener cuidado, pero sólo es un poco de práctica...

En general, podemos utilizar las funciones de conversión de tipos de datos para expresar el resultado de alguna operación como un tipo de datos concreto en vez de como el tipo de datos predeterminado. Por ejemplo, utilizaremos CDec para forzar la ejecución de operaciones con aritmética decimal en los casos en los que se haría con precisión simple, doble precisión o aritmética entera.

Cuando la parte fraccionaria es exactamente 0,5, CInt y CLng siempre redondean al número par más cercano. Por ejemplo, 0,5 se redondea a 0 y 1,5 a 2. CInt y CLng se diferencian de las funciones Fix e Int en que éstas últimas truncan la parte fraccionaria de un número en lugar de redondearla. Además, Fix e Int siempre devuelven un valor del mismo tipo que reciben.

CDate reconoce literales de fecha y literales de hora además de números comprendidos dentro del intervalo de fechas aceptables. CDate reconoce formatos de fecha que se ajusten a la configuración regional del sistema. Debe suministrar el día, mes y año en el orden correcto para la configuración regional, de no hacerlo así, es posible que la fecha no se interprete de forma correcta. No se puede reconocer un formato de fecha largo si contiene la cadena del día de la semana, por ejemplo "Miércoles".

El tipo de datos Date siempre contiene información de fecha y hora. Para la conversión de tipos, Visual Basic .NET considera 1/1/1 (1 de enero del año 1) un valor neutral de fecha y 00:00:00 (medianoche) un valor

neutral de hora. Si se convierte un valor Date a una cadena, Cstr no incluye valores neutrales en la cadena resultante. Por ejemplo, si se convierte #January 1, 0001 9:30:00# en una cadena, el resultado sería "9:30:00 a.m.", ya que la información de fecha se omite. No obstante, la información de fecha sigue estando presente en el valor Date original y se puede recuperar mediante funciones como DatePart.

La función CType acepta un segundo argumento, typename, y convierte expression a typename, donde typename puede ser cualquier tipo de datos, estructura, clase o interfaz. Luego veremos mas cosas de la función CType

Ejemplo de la función CBool

Este ejemplo utiliza la función CBool para convertir expresiones en valores Boolean. Si una expresión se evalúa a un valor distinto de cero, CBool devuelve True; en caso contrario, devuelve False.

```
Dim A, B, C As Integer

Dim Check As Boolean


A = 5

B = 5

Check = CBool(A = B)   ' Check se establece a True.

' ...

C = 0

Check = CBool(C)   ' Check se establece a False.
```

Ejemplo de la función CByte

Este ejemplo utiliza la función CByte para convertir expresiones en valores Byte.

```
Dim MyDouble As Double

Dim MyByte As Byte

MyDouble = 125.5678

MyByte = CByte(MyDouble)   ' MyByte se establece a 126.
```

Ejemplo de la función CChar

En este ejemplo, la función CChar convierte el primer carácter de una expresión String en un tipo Char.

```
Dim MyString As String

Dim MyChar As Char

MyString = "BCD"   ' CChar convierte sólo el primer carácter de la cadena.
```

```
MyChar = CChar(MyString)   ' MyChar se establece a "B".
```

El argumento de entrada para CChar debe ser el tipo de datos String. No puede utilizar CChar para convertir un número en un carácter, porque CChar no acepta un tipo de datos numéricos. Este ejemplo obtiene un número que representa un punto de código (código de carácter) y lo convierte en el carácter correspondiente. Utiliza InputBox para obtener la cadena de dígitos, CInt para convertir la cadena en el tipo Integer y ChrW para convertir el número en el tipo Char.

```
Dim MyDigits As String          ' Cadena de dígitos de entrada para convertir.

Dim CodePoint As Integer        ' Números a representar como caracteres.

Dim MyChar As Char

MyDigits = 1234,5432

CodePoint = CInt(MyDigits)      ' Convierte la cadena entera en Integer.

MyChar = ChrW(CodePoint)        ' MyChar se establece a tipo Char.
```

Ejemplo de la función CDate

Este ejemplo utiliza la función CDate para convertir cadenas en valores Date. En general, no se recomienda especificar las fechas y horas como cadenas en el código, tal y como se puede ver en este ejemplo. Utiliza literales de fecha y hora, por ejemplo #Feb 12, 1969# y #4:45:233 p.m.#.

```
Dim MyDateString, MyTimeString As String

Dim MyDate, MyTime As Date

MyDateString = "February 12, 1969"

MyTimeString = "4:35:47 PM"

' ...

MyDate = CDate(MyDateString)   ' Convierte la cadena en tipo Date.

MyTime = CDate(MyTimeString)   ' Convierte la cadena en tipo Date.
```

Ejemplo de la función CDbl

Este ejemplo utiliza la función CDbl para convertir expresiones en valores Double.

```
Dim MyDec As Decimal

Dim MyDouble As Double

MyDec = 234.456784D                    'El caracter D hace que Mydec sea decimal.

MyDouble = CDbl(MyDec * 8.2D * 0.01D)   ' Convierte el resultado a Double.
```

Ejemplo de la función CDec

Este ejemplo utiliza la función CDec para convertir un valor numérico en Decimal.

```
Dim MyDouble As Double

Dim MyDecimal As Decimal

MyDouble = 10000000.0587

MyDecimal = CDec(MyDouble)   ' Convierte a decimal.
```

Ejemplo de la función CInt

Este ejemplo utiliza la función CInt para convertir un valor en Integer.

```
Dim MyDouble As Double

Dim MyInt As Integer

MyDouble = 2345.5678

MyInt = CInt(MyDouble)   ' MyInt se establece a 2346.
```

Ejemplo de la función CLng

Este ejemplo utiliza la función CLng para convertir valores en Long.

```
Dim MyDbl1, MyDbl2 As Double

Dim MyLong1, MyLong2 As Long

MyDbl1 = 25427.45

MyDbl2 = 25427.55

MyLong1 = CLng(MyDbl1)   ' MyLong1 contiene 25427.

MyLong2 = CLng(MyDbl2)   ' MyLong2 contiene 25428.
```

Ejemplo de la función CObj

Este ejemplo utiliza la función CObj para convertir un valor numérico en Object. La variable Object en sí contiene sólo un puntero de cuatro bytes, que señala al valor Double que tiene asignado.

```
Dim MyDouble As Double

Dim MyObject As Object

MyDouble = 2.7182818284

MyObject = CObj(MyDouble)   ' El valor Double está apuntado por MyObject.
```

Ejemplo de la función CShort

Este ejemplo utiliza la función CShort para convertir un valor numérico en Short.

```
Dim MyByte as Byte

Dim MyShort as Short

MyByte = 100

MyShort = CShort(MyByte)   ' Convierte a Short.
```

Ejemplo de la función CSng

Este ejemplo utiliza la función CSng para convertir valores en Single.

```
Dim MyDouble1, MyDouble2 As Double

Dim MySingle1, MySingle2 As Single

MyDouble1 = 75.3421105

MyDouble2 = 75.3421567

MySingle1 = CSng(MyDouble1)   ' MySingle1 se establece a 75.34211.

MySingle2 = CSng(MyDouble2)   ' MySingle2 se establece a 75.34216.
```

Ejemplo de la función CStr

Este ejemplo utiliza la función CStr para convertir un valor numérico en String.

```
Dim MyDouble As Double

Dim MyString As String

MyDouble = 437.324

MyString = CStr(MyDouble)   ' MyString se establece a "437.324".
```

Este ejemplo utiliza la función CStr para convertir valores Date en valores String.

```
Dim MyDate As Date

Dim MyString As String

' ...

MyDate = #Febrero 12, 1969 00:00:00#   ' formato inválido.

' La fecha debe estar en el formato #m/d/yyyy#

' ...

MyDate = #2/12/69 00:00:00#   ' Medianoche.
```

```
' El valor neutral de hora 00:00:00 se elimina de la conversión

MyString = CStr(MyDate)   ' MyString se establece a "2/12/1969".

' ...

MyDate = #2/12/69 00:00:01#   ' Un segundo después de medianoche

' El componente de la hora pasa a ser parte del valor convertido

MyString = CStr(MyDate)   ' MyString es ahora "2/12/1969 12:00:01 AM".
```

Cstr siempre procesa un valor Date en el formato corto estándar de la configuración regional actual, por ejemplo, "15/6/2003 4:35:47 p.m.".

Función CType

Esta función es importante así que le dedicaremos algunas líneas mas...Devuelve el resultado de convertir explícitamente una expresión a un tipo de datos, objeto, estructura, clase o interfaz.

```
CType(expression, typename)
```

Partes

expression
: Cualquier expresión válida. Si el valor de *expression* está fuera del intervalo permitido por *typename*, se produce un error.

typename
: Cualquier expresión válida dentro de una cláusula **As** de una instrucción **Dim**, es decir, el nombre de cualquier tipo de datos, objeto, estructura, clase o interfaz.

Comentarios

Ctype se compila en línea, es decir, el código de conversión forma parte del código que evalúa la expresión. La ejecución es más rápida, ya que no existen llamadas a un procedimiento que realiza la conversión.

Ejemplo

En este ejemplo se utiliza la función **CType** para convertir una expresión al tipo de datos especificado.

```
Dim Minumero As Long

Dim Minuevonumero As Single

Minumero = 1000

Minuevonumero= CType(Minumero,Single)   ' Se establece Minuevonumero a 1000.0
```

Mas sobre conversiones

Antes comentamos que había dos formas de cambiar de tipo de datos o como se dice en programación hacer un "casting". Visual Basic.Net proporcionados formas para hacer casting. CType, como hemos visto, convierte un tipo de datos en otro.

La otra función es que convierte un tipo en otro pero con mejor rendimiento que Ctype... entonces ¿porqué no utilizamos esta? Muy sencillo porque DirectCat no realiza comprobación si el tipo de datos coincide con el tipo de datos que se espera convertir.

CType incluye todas las funciones de conversión de Visual Basic: CBool, CByte, CChar, CDate, CDec, CDbl, CInt, CLng, CObj, CShort, CSng, and CStr.

La principal diferencia entre los dos es que DirectCast solo funciona si el tipo de datos especificado y el tipo proporcionado en la expresión en tiempo de ejecución es el mismo. Esta diferencia sólo se presenta cuando estamos convirtiendo desde un tipo objeto a un tipo valor. Por ejemplo la siguiente operación con DirectCast fallará porque el tipo del objeto O no es un entero:

```
Dim O As Object = 1.5

Dim I As Integer = DirectCast(O, Integer)       ' Se produce un error en tiempo de ejecuci

Dim I As Integer = CType(O, Integer)            ' No produce un error
```

En cambio en este caso si funcionará:

```
Dim O As Object = 1

Dim I As Integer = DirectCast(O, Integer) Funciona

Dim I As Integer = CType(O, Integer)    ' Funciona pero es mas lento que DirectCast
```

Cuando estamos convirtiendo de un objeto a un valor, DirectCast tiene mejor rendimiento que Ctype. El compilador de Visual Basic .NET genera 4 líneas de código para DirectCast. Sin embargo utilizando CType el compilador genera una llamada a un método de conversión que necesita unas 100 líneas de código interno, esto es porque internamente llama a otros métodos. En aplicaciones donde el rendimiento es crítico la diferencia puede ser sustancial si se realiza un proceso masivo de cálculo.

5.3 Funciones de comprobación de tipos

Esta colección de funciones las vamos a utilizar en muchas ocasiones en nuestra interfaz. Por ejemplo, tenemos en nuestra interfaz una petición de datos entre los que se encuentra la fecha de nacimiento de la persona. Como es un valor obligatorio comprobamos si ha escrito algo el usuario, es decir:

```
dim mifecha as date

if dato_fecha.text="" then

    Response.write ("Debes escribir un valor en esta casilla")

else

    mifecha=cdate(dato_fecha.text)

    'seguimos proceso...
```

```
        ' ...
    end if
```

En principio bien porque ha escrito un valor así que se lo asignaré a mi variable de trabajo "mifecha" y para esto la he convertido primero de string a fecha con "cdate" Pero... ¿y si el usuario ha escrito un valor que no es de tipo fecha? nuestro programa fallará y mostrará un feo mensaje de error. Así que tenemos que controlar que además de escribir valores, el usuario los escriba correctamente. Para esto utilizaremos las funciones de comprobación de tipos. Sigamos con nuestro ejemplo... la función **Isdate** nos va a devolver True si puede convertir a fecha el dato que le demos, es decir si es de tipo fecha, y false si no es de tipo fecha:

```
dim mifecha as date

    if dato_fecha.text="" then

        Response.Write ("Debes escribir un valor en esta casilla")

    else

        if isdate (dato_fecha.text) then

            mifecha=cdate(dato_fecha.text)

            'seguimos proceso...

            ' ...

        else

           Response.Write ("El valor introducido no es correcto")

        end if

    end if
```

Así que primero comprobamos que el usuario a escrito un valor y luego si ese valor es de tipo fecha: "if isdate (dato_fecha.text) then"

Nota: En las instrucciones "If the..." se evalúa si se cumple o no una expresión y su resultado sólo puede ser True o False. Además las funciones de conversión de tipo devuelven igualmente true o false dependiendo de si pueden convertir o no la expresión al tipo de datos indicado. Así que es igual escribir esta expresión:

If isdate (dato_fecha.text) <----> if isdate(dato_fecha.text)=true

If not isdate (dato_fecha.text) <----> if isdate(dato_fecha.text)=false

Nosotros utilizaremos la primera notación pero puede que al principio te quede mas clara la segunda

Las funciones para comprobar los tipos de datos son:

IsNumeric (Función)

Devuelve un valor de tipo **Boolean** que indica si una expresión se puede evaluar como un número.

Sintaxis

IsNumeric(*expresión***)**

El argumento *expresión* requerido, es un tipo de datos Variant que contiene una expresión numérica o una expresión de tipo cadena.

Comentarios

La función **IsNumeric** devuelve **True** si la *expresión* completa se reconoce como un número; en otro caso, devuelve **False**.

Ejemplo

En este ejemplo se utiliza la función IsNumeric para determinar si el contenido de una variable puede evaluarse como un número.

```
Dim MyVar As Object

Dim MyCheck As Boolean

' ...

MyVar = "53" ' Asignamos valor.

MyCheck = IsNumeric(MyVar) ' Devuelve True.

' ...

MyVar = "459.95" ' Asignamos valor.

MyCheck = IsNumeric(MyVar) ' Devuelve True.

' ...

MyVar = "45 Help" ' Asignamos valor.

MyCheck = IsNumeric(MyVar) ' Devuelve False.
```

IsArray (Función)

Devuelve un valor de tipo **Boolean** que indica si una variable es una matriz.

Sintaxis

IsArray(*nombrevariable***)**

El argumento requerido *nombrevariable*, es un identificador que especifica una variable.

Comentarios

La función **IsArray** devuelve **True** si la variable es una matriz; en caso contrario, devuelve **False**. La función **IsArray** es especialmente útil en el caso de objetos que contienen matrices.

En este ejemplo se utiliza la función IsArray para comprobar si varias variables hacen referencia a una matriz.

```
Dim MyArray(4), YourArray(3) As Integer ' Declara variables de matriz.

Dim MyString As String

Dim MyCheck As Boolean

MyCheck = IsArray(MyArray) ' Devuelve True.

MyCheck = IsArray(YourArray) ' Devuelve True.

MyCheck = IsArray(MyString) ' Devuelve False.
```

IsDate (Función)

Devuelve un valor de tipo **Boolean** que indica si una expresión se puede convertir en una fecha.

Sintaxis

IsDate(*expresión***)**

El argumento *expresión* requerido, es un tipo de datos Variant que contiene una expresión de fecha o una expresión de cadena reconocible como una fecha o una hora.

Comentarios

La función **IsDate** devuelve **True** si la expresión es una fecha o se puede reconocer como una fecha válida; en caso contrario, devuelve **False**. En Microsoft Windows, el intervalo de fechas válidas va desde el 1 de enero de 100 D. de C. hasta el 31 de diciembre de 9999 D.de C.; los intervalos varían de un sistema operativo a otro.

Ejemplo

En este ejemplo se utiliza la función IsDate para determinar si varias variables se pueden convertir a fechas.

```
Dim MyDate, YourDate As Date

Dim NoDate As String

Dim MyCheck As Boolean

MyDate = "February 12, 1969"

YourDate = #2/12/1969#
```

```
NoDate = "Hello"

MyCheck = IsDate(MyDate) ' Devuelve True.

MyCheck = IsDate(YourDate) ' Devuelve True.

MyCheck = IsDate(NoDate) ' Devuelve False.
```

IsNothing (Función)

Devuelve un valor de tipo **Boolean** que indica si una variable ha sido inicializada. Devuelve un valor de tipo Boolean que indica si una expresión no tiene ningún objeto asignado.

```
Public Function IsNothing(ByVal Expression As Object) As Boolean
```

Comentarios

IsNothing devuelve True si la expresión representa una variable de tipo Object que no tiene actualmente ningún objeto asignado; en caso contrario, devuelve False.

Ejemplo

En este ejemplo se utiliza la función IsNothing para determinar si una variable de objeto está asociada a alguna instancia de objeto.

```
Dim MyVar As Object ' No hay instancia asignada todavía.

Dim MyCheck As Boolean

' ...

MyCheck = IsNothing(MyVar) ' Devuelve True.

' ...

MyVar = "ABCDEF" ' Asignamos una instancia de string a la variable.

MyCheck = IsNothing(MyVar) ' Devuelve False.

' ...

MyVar = Nothing ' Quita la asociación de la variable desde cualquier instancia

MyCheck = IsNothing(MyVar) ' Devuelve True.
```

IsError (Función)

Devuelve un valor de tipo **Boolean** que indica si una expresión tiene un valor de error.

Sintaxis

IsError(*expresión***)**

El argumento *expresión* requerido, puede ser cualquier expresión válida.

Comentarios

Los valores de error se crean al convertir números reales a valores de error utilizando la función **CVErr**. La función **IsError** se utiliza para determinar si una expresión numérica representa un error. La función **IsError** devuelve **True** si el argumento *expresión* indica un error; en caso contrario, devuelve **False**.

Ejemplo

En este ejemplo se utiliza la función IsError para comprobar si una expresión representa una excepción del sistema.

```
Dim ReturnVal As Object

Dim BadArg As String ' Nombre del argumento fuera de rango

Dim MyCheck As Boolean

' ...

ReturnVal = New System.ArgumentOutOfRangeException(BadArg)

' ...

MyCheck = IsError(ReturnVal) ' Devuelve True
```

IsDBNull (Función)

Devuelve un valor de tipo **Boolean** que indica si una expresión contiene datos no válidos (Null).

Sintaxis

IsNull(*expresión***)**

El argumento *expresión* requerido, es un tipo de datos Variant que contiene una expresión numérica o una expresión de cadena.

Comentarios

IsDBNull devuelve **True** si el tipo de datos de *Expression* se evalúa como de tipo **DBNull**; si no, **IsDBNull** devuelve **False**.

El valor **System.DBNull** indica que la expresión de tipo **Object** representa datos que no se encuentran o no existen. **DBNull** no es lo mismo que **Nothing**, que indica que una variable no se ha inicializado todavía. Tampoco es lo mismo que una cadena de longitud cero (""), a la que a veces se hace referencia como cadena de valor null

IsDBNull devuelve **True** si el tipo de datos de *Expression* se evalúa como de tipo **DBNull**; si no, **IsDBNull** devuelve **False**.

El valor **System.DBNull** indica que la expresión de tipo **Object** representa datos que no se encuentran o no existen. **DBNull** no es lo mismo que **Nothing**, que indica que una variable no se ha inicializado todavía. Tampoco es lo mismo que una cadena de longitud cero (""), a la que a veces se hace referencia como cadena de valor null

Ejemplo

En este ejemplo se utiliza la función IsDBNull para determinar si una variable se evalúa como DBNull.

```
Dim MyVar As Object

Dim MyCheck As Boolean

MyCheck = IsDBNull(MyVar) ' Devuelve False.

MyVar = ""

MyCheck = IsDBNull(MyVar) ' Devuelve False.

MyVar = System.DBNull.Value

MyCheck = IsDBNull(MyVar) ' Devuelve True.

Response.Write(MyCheck)
```

TypeName (Función)

Devuelve una cadena (**String)** que proporciona información acerca de una variable.

Sintaxis

TypeName(*nombrevariable***)**

El argumento *nombrevariable* requerido, es un tipo de datos Variant que contiene cualquier variable excepto una variable de un tipo definido por el usuario.

Comentarios

En la siguiente tabla se muestran los valores **String** devueltos por **TypeName** para distintos contenidos de *VarName*:

Contenido *VarName*	Cadena devuelta
Tipo de valor **True** o **False** de 16 bits	"Boolean"
Valor binario de 8 bits	"Byte"
Carácter de 16 bits	"Char"
Valor de fecha y hora de 64 bits	"Date"
Tipo de referencia que indica que los datos faltan o no existen	"DBNull"
Valor numérico de punto fijo de 128 bits	"Decimal"
Valor numérico de punto flotante de 64 bits	"Double"
Valor entero de 32 bits	"Integer"
Referencia que apunta a un objeto no especializado	"Object"
Referencia que apunta a un objeto especializado creado a partir de la clase <*objectclass*>	"<*objectclass*>"
Valor entero de 64 bits	"Long"
Tipo de referencia sin ningún objeto actualmente asignado	"Nothing"
Valor entero de 16 bits	"Short"
Valor numérico de punto flotante de 32 bits	"Single"
Referencia que apunta a una cadena de caracteres de 16 bits	"String"

Si VarName es una matriz, la cadena devuelta puede ser cualquiera de las cadenas de la tabla anterior seguida de paréntesis vacíos. Por ejemplo, si VarName apunta a una matriz de enteros, TypeName devuelve "Integer()".

Cuando TypeName devuelve el nombre de un tipo de referencia, tal como una clase, sólo devuelve el nombre simple, no el nombre completo. Por ejemplo, si VarName apunta a un objeto de la clase System.Drawing.Printing.PaperSource, TypeName devuelve "PaperSource".

Ejemplo

En este ejemplo se utiliza la función TypeName para devolver información acerca del tipo de datos de varias variables.

```
Dim MyType As String

Dim StrVar As String = "MyString"

Dim DecVar As Decimal

Dim IntVar, ArrayVar(5) As Integer

MyType = TypeName(StrVar) ' Devuelve "String".
```

```
MyType = TypeName(IntVar) ' Devuelve "Integer".

MyType = TypeName(ArrayVar) ' Devuelve "Integer()".
```

VarType (Función)

Devuelve un entero (**Integer)** que indica el subtipo de una variable.

Sintaxis

VarType(*nombrevariable***)**

El argumento *nombrevariable* requerido, es un tipo Variant que contiene cualquier variable excepto una variable de un tipo definido por el usuario.

El valor entero devuelto por **VarType** es un miembro de la enumeración **VariantType**. En la siguiente tabla se muestran los valores devueltos por **VarType** para casos especiales de *VarName*.

Tipo de datos representado por *VarName*	Valor devuelto por *VarType*
Nothing	**VariantType.Object**
DBNull	**VariantType.Null**
Enumeración	Tipo de datos subyacente (**Byte**, **Short**, **Integer** o **Long**)
Matriz	OR bit a bit de tipo de elemento de matriz y **VariantType.Array**
Matriz de matrices	OR bit a bit de **VariantType.Object** y **VariantType.Array**
Estructura (**System.ValueType**)	**VariantType.UserDefinedType**
System.Exception	**VariantType.Error**
Desconocido	**VariantType.Object**

Ejemplo

En este ejemplo se utiliza la función VarType para devolver información acerca de la clasificación del tipo de datos de varias variables.

```
Dim MyString As String = "MyString"

Dim MyObject As Object

Dim MyNumber, MyArray(5) As Integer

Dim MyVarType As VariantType    ' Enumeración de Integer

MyVarType = VarType(MyVarType)  ' Devuelve VariantType.Integer.

MyVarType = VarType(MyString)   ' Devuelve VariantType.String.
```

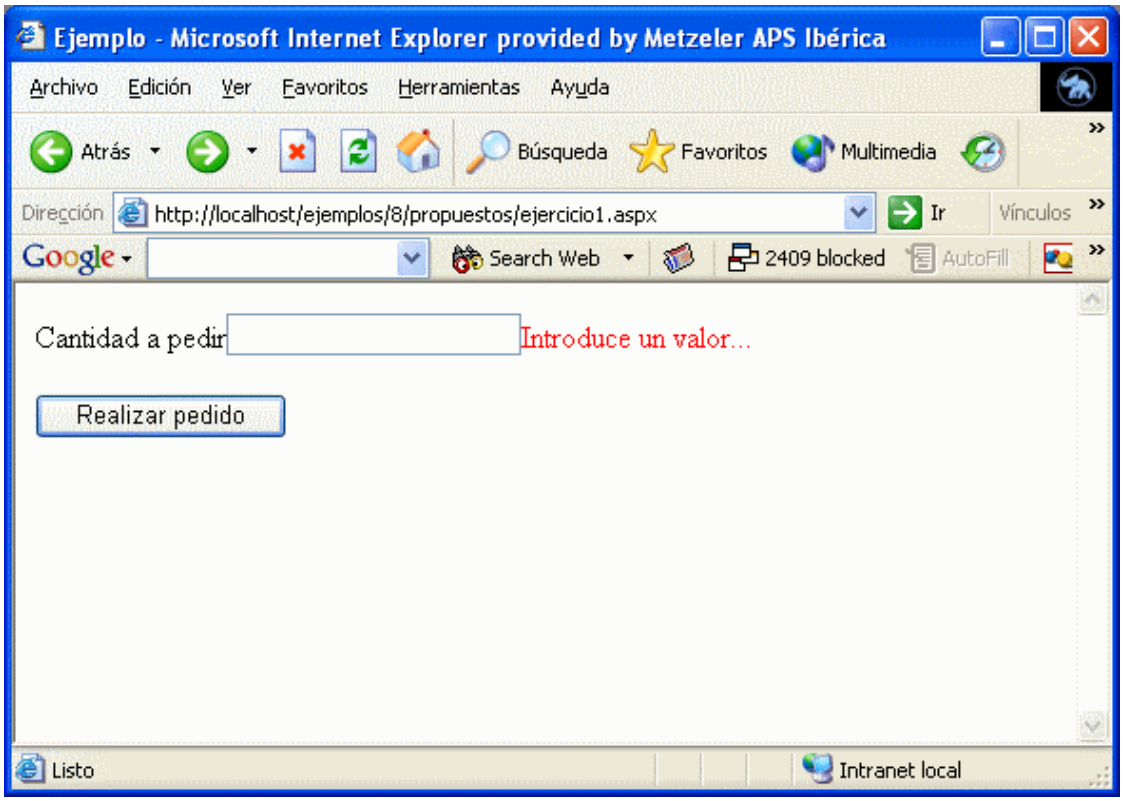
```
MyVarType = VarType(MyObject)   ' Devuelve VariantType.Object.
MyVarType = VarType(MyArray)    ' Devuelve la combinación de VariantType.Array y VariantTy
```

Pulsa aquí para descargar los ejemplos de este tema

Ejercicios

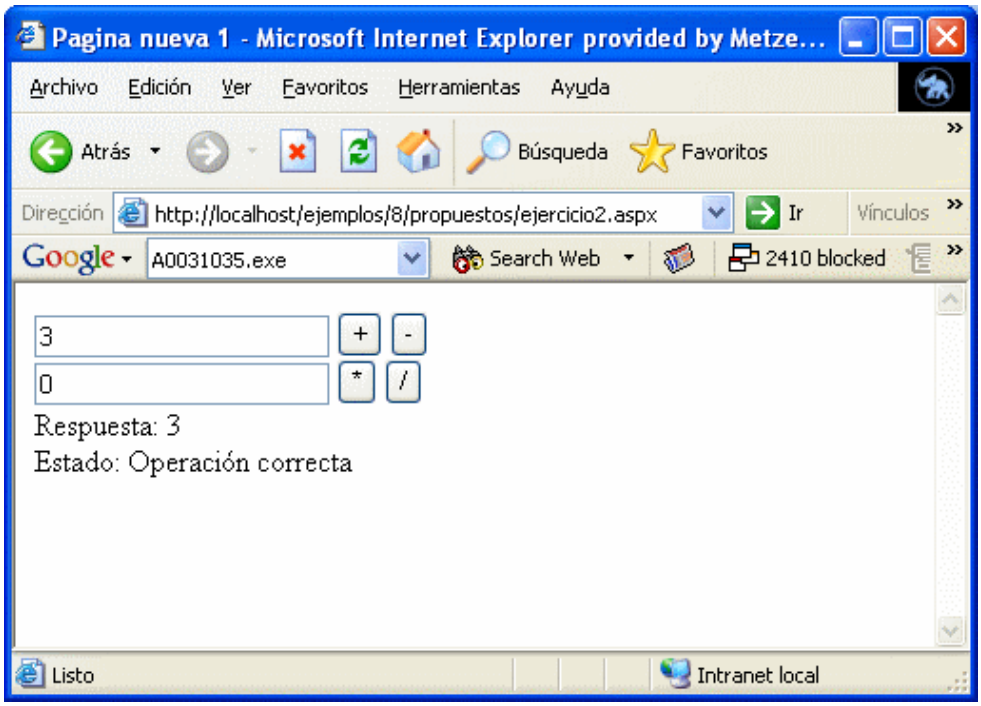
Ejercicio 1

Valida en una página que se:
- que se introduzca un valor
- que sea numérico
- que esté en el intervalo 1-1000

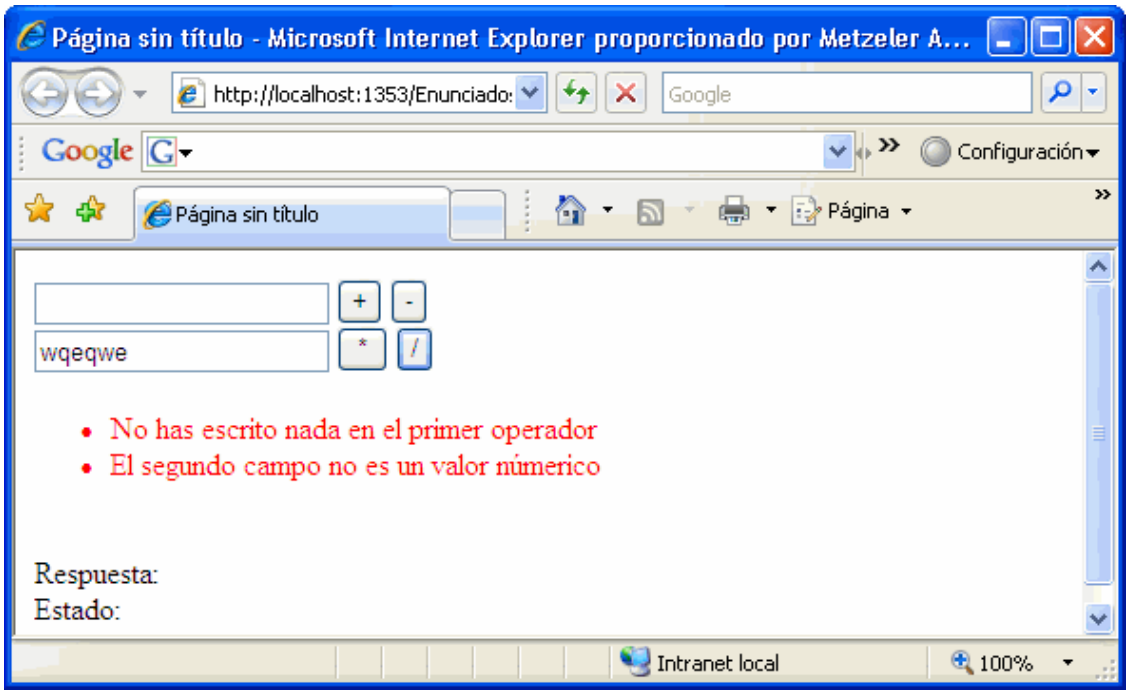


Ejercicio 2

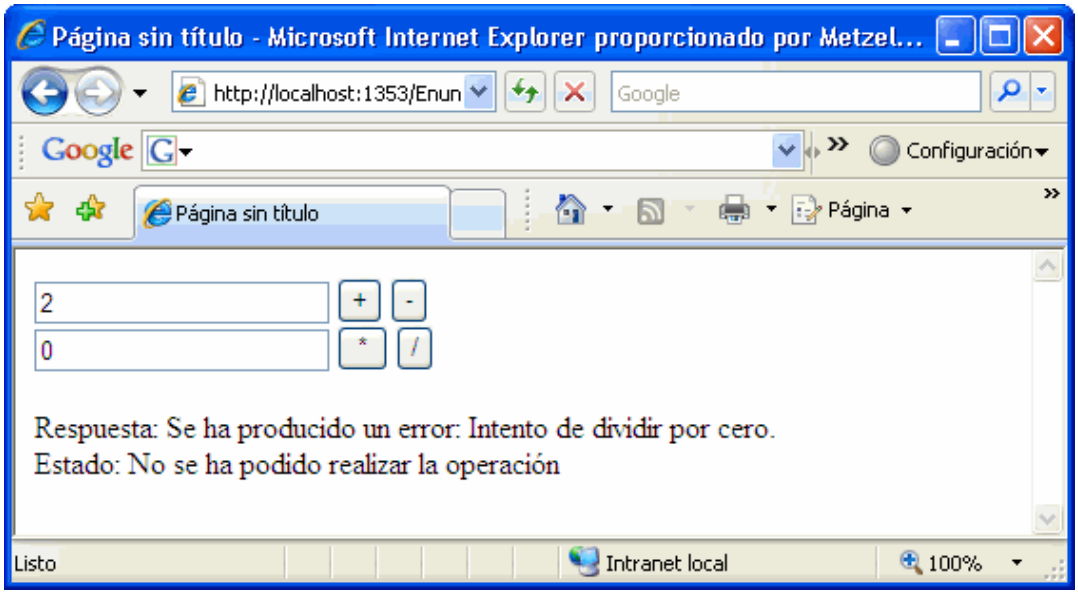
Haz una calculadora con controles de validación para admitir solo valores numéricos y que intercepte los posibles errores, como la división por 0. El resultado de la evaluación de si son correctos lo pondremos en un control de validación de resumen.



Por ejemplo:



o si es una division por 0:


Página sin título - Microsoft Internet Explorer proporcionado por Metzel...
http://localhost:1353/Enun
Google
Configuración
Página sin título
Página
2
+
-
0
*
/
Respuesta: Se ha producido un error: Intento de dividir por cero.
Estado: No se ha podido realizar la operación
Listo
Intranet local
100%