C#

Nuevas características en C# 6

Miguel Muñoz Serafín
@msmdotnet

Octubre de 2015

TI Capacitación
http://ticapacitacion.com

Contenido

Introducción 2
Métodos de extensión Add en Inicializadores de Colecciones 3
Inicializadores de Índice (Index initializers) 4
Expresiones Nameof 6
Operador Null-Conditional 9
Interpolación de Cadenas (String interpolation) 13
using static 15
Uso del operador await en bloques catch y finally 18
Filtros de Excepción (Exception filters) 19
Inicializadores para propiedades Auto-Implementadas 23
Propiedades Auto-implementadas de sólo lectura 24
Miembros con una Expresión como cuerpo (Expression-bodied function members) 25

Introducción

C#, pronunciado en inglés como “C sharp”, es un lenguaje de programación diseñado para construir una gran variedad de aplicaciones que se ejecutan en el .NET Framework. C# es simple, poderoso, de tipos seguros y orientado a objetos. Las muchas mejoras que ha tenido C# permiten un rápido desarrollo de aplicaciones al mismo tiempo que mantiene la elegancia de los lenguajes de estilo C.

Visual C# es la implementación de Microsoft del lenguaje C#. La siguiente tabla resume las distintas características que han sido incorporadas a Visual C# así como las versiones de Visual Studio en las que se hicieron disponibles.

Versión	Versión de Visual Studio	Principales características
C# 1	Visual Studio .NET 2002	Primera versión.
C# 1.1	Visual Studio .NET 2003	Directiva #line y Comentarios de documentación XML.
C# 2	Visual Studio .NET 2005	Métodos anónimos, Tipos Genéricos, Tipos Anulables, Iteradores/yield, Clases estáticas, Covarianza y Contravarianza para delegados.
C# 3	Visual Studio .NET 2008	Inicializadores de Objetos y Colecciones, Expresiones Lambda, Métodos de Extensión, Tipos Anónimos, Propiedades Auto-implementadas, LINQ, inferencia de tipos (var).
C# 4	Visual Studio .NET 2010	Dynamic, Argumentos nombrados, Parámetros opcionales, covarianza y contravarianza para tipos Genéricos.
C# 5	Visual Studio .NET 2012	Async, Await, Atributos para obtener información de Invocadores: CallerMemberName, CallerFilePath, CallerLineNumber.
	Visual Studio .NET 2013	Corrección de algunos errores, mejoras en el rendimiento, versión preliminar de la plataforma .NET Compiler (Roslyn).
C# 6	Visual Studio .NET 2015	Métodos de extensión Add en Inicializadores de Colecciones, Inicializadores de Índice (Index initializers), Expresiones Nameof, Operador Null-Conditional, Interpolación de Cadenas (String interpolation), using static, await en bloques catch y finally, Exception filters, Inicializadores para propiedades Auto-Implementadas, Propiedades Auto-implementadas de sólo lectura, Miembros con una Expresión como cuerpo.

C# 6, es la versión actual de C#. Visual Studio 2015 es la versión que nos permite aprovechar las nuevas características del lenguaje.

Métodos de extensión Add en Inicializadores de Colecciones

Cuando se implementaron por primera vez los inicializadores de colecciones en C#, los métodos **Add** que se tenían que invocar, no podían ser métodos de extensión, tenían que ser métodos de instancia. En Visual C# 6, los métodos **Add** de extensión ya son vistos por los inicializadores de objetos.

Por ejemplo, el siguiente código en C# 5 no compila mientras que en C# 6 si compila.

```
class Program
{
    static void Main()
    {
        Queue Q = new Queue { 5, 10, 15 };
    }
}

static class ExtensionMethods
{
    public static void Add(this Queue collection, object item)
    {
        collection.Enqueue(item);
    }
}
```

Inicializadores de Índice (Index initializers)

En C# 6 se ha agregado una sintaxis nueva para para inicializar objetos permitiendo establecer valores a llaves a través de cualquier Indizador (Indexer) que el objeto implemente. Con esta nueva característica, ahora podemos inicializar elementos específicos de una colección que implemente Indizadores, tal como un objeto **Dictionary**.

El siguiente ejemplo muestra el uso de esta nueva característica a través de un objeto **Dictionary**.

En C# 5.

```
// Insertamos elementos llave-valor
var SpanishDays = new Dictionary<byte, string>
{
    {2, "Lunes"},
    {3, "Martes"},
    {4, "Miércoles"},
    {5, "Jueves"},
    {6, "Viernes"}
};
```

En C# 6.

```
// Insertamos los valores de las llaves 2, 3, 4, 5, 6
var SpanishDays = new Dictionary<byte, string>
{
    [2] = "Lunes",
    [3] = "Martes",
    [4] = "Miércoles",
    [5] = "Jueves",
    [6] = "Viernes"
};
```

El siguiente ejemplo muestra una clase **Inventory** que implementa un Indizador.

```
class Inventory
{
    Dictionary<string, int> Products =
        new Dictionary<string, int>();
    public int this[string index]
    {
        get { return Products[index]; }
        set { Products[index] = value; }
    }
}
```

En C# 5 podemos inicializar y agregar elementos a una instancia de **Inventory** de la siguiente manera. Nótese que no es posible Inicializar y agregar elementos en la misma sentencia.

```
// Creamos la Instancia
var UnitsInStock = new Inventory();
// Insertamos elementos
UnitsInStock["Azucar"] = 5;
UnitsInStock["Leche"] = 10;
UnitsInStock["Frijol"] = 100;
```

En C# 6 podemos inicializar y agregar elementos a una instancia de **Inventory** de la siguiente manera.

```
// Creamos e insertamos en la misma sentencia
var UnitsInStock = new Inventory
{
    ["Azucar"] = 5,
    ["Leche"] = 10,
    ["Frijol"] = 100
};
```

Expresiones Nameof

A través del operador **nameof**, C# 6 nos permite obtener una cadena con el nombre de una Variable, Clase, Método, Parámetro, Propiedad o incluso el nombre de un atributo. La cadena con el nombre obtenido no contiene el espacio de nombres correspondiente.

El siguiente ejemplo nos permite obtener el nombre de un parámetro como cadena.

```
static void GetParameterName(int productID)
{
    string Name = nameof(productID);
    Console.WriteLine(Name);
}
```

En este ejemplo, la Consola mostraría:

productID

Un ejemplo práctico de la utilidad de esta característica podemos verlo cuando desarrollamos utilizando el patrón MVVM y necesitamos notificar que el valor de una propiedad ha cambiado. Para realizar esta notificación es necesario proporcionar el nombre de la propiedad como cadena. Podemos especificar el nombre de la cadena con código duro pero eso nos traería resultados inesperados si se nos ocurre modificar el nombre de la propiedad y no modificamos la cadena en código duro.

Con el operador **nameof**, el código podría quedar de la siguiente manera.

```
public decimal UnitPrice
{
    set
    {
        // ...
        RaisePropertyChanged(nameof(UnitPrice));
    }
}

public void RaisePropertyChanged(string propertyname)
{
    Console.WriteLine(propertyname);
}
```

En este ejemplo, la Consola mostraría:

UnitPrice

Es importante mencionar que el operador **nameof** no obtiene el espacio de nombres, únicamente el identificador. Por lo tanto, las siguientes sentencias mostrarían en la Consola el mismo resultado:

Main

```
Console.WriteLine(nameof(_03.Program.Main));
Console.WriteLine(nameof(Main));
```

Proporcionar al operador **nameof** el espacio de nombres junto con el identificador, sólo le indica a **nameof** el lugar donde se encuentra el identificador. Esto es bastante útil cuando tenemos el mismo identificador en distintos espacios de nombres.

Otro ejemplo útil del operador **nameof** es en aplicaciones ASP.NET MVC donde tenemos que especificar los nombres como cadena de los Controladores y métodos de acción como en el siguiente ejemplo.

En C# 5 utilizamos cadenas de texto para especificar nombres de Controladores y Acciones.

```
public ActionResult Index()
{
    return RedirectToAction("Index", "Login");
}
```

En C# 6 podemos utilizar el operador **nameof** evitando resultados inesperados al cambiar los nombres de los Controladores o Acciones ya que el compilador nos avisaría al compilar la aplicación.

```
public ActionResult Index()
{
    return RedirectToAction(
        nameof(Controllers.AccountController.Login),
        nameof(Controllers.AccountController).Replace("Controller",""));
}
```

En aplicaciones ASP.NET, para que el operador **nameof** sea reconocido en las Vistas durante la parte de compilación en tiempo de ejecución, es necesario habilitar la compilación **Roslyn** en ASP.NET instalando el siguiente paquete NuGet en la aplicación Web ASP.NET:

CodeDOM Providers for .NET Compiler Platform ("Roslyn") NuGet package

Para instalar el paquete en el proyecto ASP.NET, podemos escribir lo siguiente desde **Package Manager Console** en Visual Studio.

install-package Microsoft.CodeDom.Providers.DotNetCompilerPlatform

Para mayor información sobre la forma de habilitar la compilación Roslyn en aplicaciones ASP.NET, pueden consultar el siguiente enlace:

http://blogs.msdn.com/b/webdev/archive/2014/05/12/enabling-the-net-compiler-platform-roslyn-in-asp-net-applications.aspx

Operador Null-Conditional

El operador **Null-Conditional** puede ser representado de dos formas:

- Mediante un símbolo de interrogación con un punto para evaluar si un objeto es null antes de acceder a sus miembros: **?.**
- Mediante un símbolo de interrogación y un corchete cuadrado de apertura para evaluar si un Tipo indexado es **null** antes de acceder a uno de sus elementos utilizando un índice: **?[**

Con el operador **Null-Conditional** podemos verificar si una expresión es **null** antes de intentar acceder a alguno de sus miembros (**?.**) o antes de acceder a uno de sus elementos indexados (**?[**). Este operador nos ayuda a escribir menos código para manejar la verificación de **null**, especialmente al acceder a los miembros de las estructuras jerárquicas de datos.

Si tuviéramos por ejemplo el siguiente código:

```
string Controller = null;
//...
//... Asignar un valor a Controller
//...
string ControllerName =
    Controller.Replace("Controller", "");
```

Al ejecutar la última instrucción, se dispararía la excepción **NullReferenceException** si **Controller** sigue siendo **null**.

En C# 5, para evitar la excepción podríamos escribir el siguiente código:

```
string ControllerName;
if(Controller==null)
{
    ControllerName = null;
}
else
{
    ControllerName =
        Controller.Replace("Controller", "");
}
```

En C# 6, podemos escribir el siguiente código que es el equivalente al código anterior:

```
string ControllerName =
    Controller?.Replace("Controller", ""); ;
```

El operador **Null-Conditional** también nos permite manejar valores **null** cuando accedemos a elementos indexados.

Por ejemplo, si tenemos la siguiente declaración:

```
byte[] Numbers = null;
//...
// Instanciar el arreglo
//...
```

El siguiente código nos permite almacenar en una variable el valor del primer elemento de un arreglo de bytes cuando el arreglo no es **null**. Si el arreglo es **null**, la variable que almacena el primer elemento será **null** igualmente.

En C# 5:

```
byte? FirstElement;
if(Numbers==null)
{
    FirstElement = null;
}
else
{
    FirstElement = Numbers[0];
}
```

En C# 6 utilizando el operador **Null-Conditional:**

```
byte? FirstElement = Numbers?[0];
```

El operador **Null-Conditional** tiene un comportamiento de Cortocircuito (short-circuiting), lo que significa que si una operación en la secuencia de acceso a miembros o a operaciones con índices devuelve **null**, entonces el resto de la cadena de ejecución se detiene.

Supongamos ahora que tenemos la siguiente declaración:

```
Customer[] Customers = null;
//...
//... Agregar Customers
//...
int? Count;
```

Queremos que si el arreglo **Customers** es **null** o el elemento **Customer[0]** es **null** o si **Customer[0].Orders** es **null**, entonces la variable **Count** reciba **null**, de lo contrario, la variable **Count** debe recibir el número de elementos de la colección **Customers[0].Orders**.

El código en C# 5 quedaría de la siguiente forma:

```
if(Customers==null ||
    Customers[0]==null ||
    Customers[0].Orders==null)
{
    Count = null;
}
else
{
    Count = Customers[0].Orders.Count();
}
```

El siguiente código C# 6 es equivalente al código anterior. Podemos notar la diferencia en el número de líneas de código necesarias para implementar la solución:

```
int? Count = Customers?[0]?.Orders?.Count();
```

En el ejemplo anterior, si tuviéramos en la expresión otras operaciones de menor precedencia, estas continuarían su ejecución como en el siguiente caso:

```
int Count2 = Customers?[0]?.Orders?.Count() ?? 0;
```

En este caso estamos utilizando el operador **null-coalescing** (**??**) dando como resultado que si la expresión **Customers?[0]?.Orders?.Count()** devuelve **null**, entonces la variable **Count2** recibe **0**, de lo contrario, la variable **Count2** recibe el valor **Customers[0].Orders.Count()**.

Otro uso bastante útil del operador **Null-Conditional** es la invocación de delegados en un hilo seguro. El siguiente ejemplo muestra el código C# 5 común utilizado para disparar el evento **PropertyChanged** en una clase ViewModel que implementa la Interface **INotifyPropertyChanged**.

```
void RaisePropertyChanged(string propertyName)
{
    var Subscribers = PropertyChanged;
    if (Subscribers != null)
    {
        Subscribers(this,
        new PropertyChangedEventArgs(propertyName));
    }
}
```

El código equivalente en C# 6 quedaría de la siguiente forma.

```
void RaisePropertyChanged(string propertyName)
{
    PropertyChanged?.Invoke(this,
    new PropertyChangedEventArgs(propertyName));
}
```

Esta nueva forma es de hilo seguro (Thread-safe) ya que el compilador genera el código para evaluar **PropertyChanged** una sola vez almacenando el resultado en una variable temporal.

Es necesario invocar explícitamente el método **Invoke** debido a que no existe una sintaxis **null-conditional** para invocar delegados.

Interpolación de Cadenas (String interpolation)

El concepto de Interpolar se define como la acción de poner algo entre otras cosas. También se define como la acción de intercalar palabras o frases dentro de un texto.

En C# 6 podemos utilizar expresiones de interpolación de cadenas para construir nuevas cadenas. Una expresión de cadena interpolada es similar a una plantilla de cadena que contiene expresiones.

Supongamos que tenemos el siguiente código:

```
string ProductName = "Azucar";
double UnitPrice = 12.45;
```

Queremos enviar a la Consola el siguiente texto:

El producto Azucar tiene un precio de 12.45

En C# 5 podemos formatear la salida con la técnica de argumentos de la siguiente manera:

```
Console.WriteLine(
    "El producto {0} tiene un precio de {1}",
    ProductName, UnitPrice);
```

En C# 6 podemos formatear la salida utilizando una cadena interpolada de la siguiente manera:

```
Console.WriteLine(
    $"El producto {ProductName} tiene un precio de {UnitPrice}");
```

Podemos ver que estamos incrustando las expresiones **ProductName** y **UnitPrice** dentro de la cadena**.**

Cada vez que se ejecuta el código con la cadena interpolada, C# crea una nueva cadena a partir de la cadena interpolada remplazando las expresiones por su representación ToString. Una cadena interpolada es muy fácil de entender en comparación con el formateo de cadenas que utilizan argumentos tales como String.Format.

Para incluir el carácter “{“ o “}” en una cadena interpolada, utilizamos llaves dobles como en el siguiente ejemplo.

```
Console.WriteLine(
    $"Nombre del producto: {{{ProductName}}}");
```

El resultado es el siguiente:

Nombre del producto: {Azucar}

Al igual que con **String.Format**, podemos especificar alineación y formatos como en el siguiente ejemplo.

```
Console.WriteLine(
    $"{ProductName,20} tiene un precio de {UnitPrice:C2} pesos");
Console.WriteLine(
    $"20 kilos equivalen a {UnitPrice*20:C0} pesos");
```

El código anterior produce la siguiente salida.

Azucar tiene un precio de $12.45 pesos

20 kilos equivalen a $249 pesos

Nótese que en la primera instrucción, el nombre del producto ocupa 20 posiciones.

Una cadena interpolada puede incluso incluir expresiones cadena como en el siguiente ejemplo.

```
int n = 40;
var s =
    $"El número {n} es {(n%2==0 ? "Par": "Impar")}";
```

Debemos notar que la expresión condicional está delimitada por paréntesis.

En este ejemplo, el valor final de la variable **s** sería:

El número 40 es Par

using static

La característica **using static** permite que todos los miembros estáticos accesibles de una clase, puedan ser importados para poder acceder a ellos sin la necesidad de especificar la clase en la que se encuentran implementados.

El siguiente ejemplo muestra el uso de la característica **using static** para acceder a los miembros estáticos de la clase **Console** sin necesidad de especificar el nombre de la clase **Console**.

```
using static System.Console;

namespace _06
{
    class Program
    {
        static void Main()
        {
            Write("¿Cuál es tu nombre?");
            var Name = ReadLine();
            WriteLine($"Hola {Name}!");
        }
    }
}
```

Esta característica es muy útil cuando tenemos un conjunto de funciones dentro de una clase que utilizamos frecuentemente.

La característica **using static**, permite también especificar directamente los valores nombrados de una enumeración como en el siguiente ejemplo.

```
using static Status;

namespace _06
{
    class Program
    {
        static void Main()
        {
            Status s = On;
            WriteLine($"Current status {s}");
        }
    }

}

enum Status
{
    On,
    Off
}
```

Métodos de Extensión

Los métodos de extensión son métodos estáticos diseñados para ser utilizados como métodos de instancia. El siguiente código muestra una clase con dos métodos estáticos, uno de ellos es un método de extensión de un tipo **string**.

```
public static class Helper
{
    // Método estático
    public static int GetRamdom()
    {
        return new System.Random().Next(1, 1000);
    }

    // Método de extensión de un tipo string
    public static int? ToInt(this string source)
    {
        int? Result = null;
        int ToConvert;
        if(int.TryParse(source, out ToConvert))
        {
            Result = ToConvert;
        }
        return Result;
    }
}
```

La característica **using static**, no importa a los métodos de extensión de la misma forma en que lo hace con los métodos estáticos que no son de extensión.

Tomando como base el código anterior de la clase **Helper**, la siguiente instrucción generaría el error de compilación "**The name 'ToInt' does not exist in the current context**".

```
using static Helper;

namespace _06
{
    class Program
    {
        static void Main()
        {
            var n2 = ToInt(n);
        }
    }
}
```

Sin embargo, las siguientes instrucciones son completamente válidas.

```
using static System.Console;
using static Helper;

namespace _06
{
    class Program
    {
        static void Main()
        {
            Write("Proporciona un número entero: ");
            string n = ReadLine();

            // Se hace disponible el método de extensión
            int? Number = n.ToInt();

            // El método de extensión puede utilizarse
            // especificando la Clase Helper
            int? Number3 = Helper.ToInt(n);

            // Los métodos estáticos se hacen disponibles
            // sin especificar la clase Helper
            var r = GetRamdom();
        }
    }
}
```

Uso del operador await en bloques catch y finally

En C# 5 y versiones anteriores no era posible realizar llamadas a métodos asíncronos dentro de los bloques **catch** y **finally**. Por ejemplo, el siguiente código generaba los errores de compilación "**Cannot await in the body of a catch clause**" y "**Cannot await in the body of a finally clause**".

```
static async Task<bool> DoSomething()
{
    var L = new Log();
    bool Result=true;
    try
    {
        await L.OpenAsync();
    }
    catch (Exception ex)
    {
        var Status =
            await L.WriteAsync(
            "Error encontrado: " + ex.Message);
        Result = false;
    }
    finally
    {
        var Status =
            await L.CloseAsync();
    }
    return Result;
}
```

Este mismo código ahora ya compila y se ejecuta sin problemas en C# 6.

Filtros de Excepción (Exception filters)

Los Filtros de Excepción son una capacidad del CLR disponible en Visual Basic y F# pero que no se encontraba disponible en C# hasta ahora con C# 6.

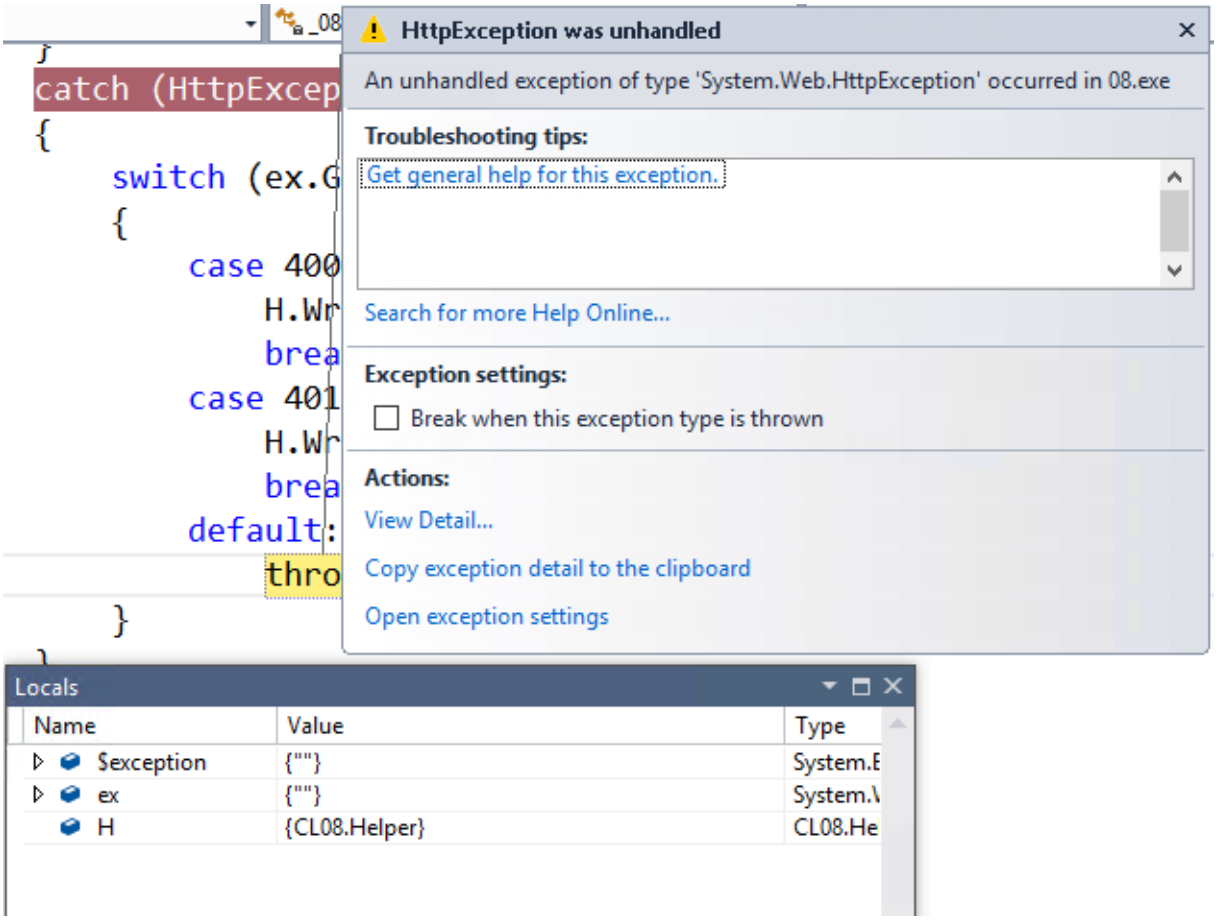
Supongamos que tenemos la necesidad de invocar dos métodos que podrían generar una excepción del tipo **HttpException.** En caso de que el código http de error sea 400 o 401 queremos registrar las excepciones a la bitácora de error de la aplicación e ignorar la excepción. En caso de que el código de error http sea distinto a 400 y 401, debemos dejar pasar la excepción al método que haya invocado al código que queremos crear.

El código propuesto es el siguiente.

```
static void DoSomething()
{
    var H = new Helper();
    try
    {
        string Message = "Hello, World";
        H.SendHttpMessage(Message, true);
        string Html = H.GetHttpResource("/Products", true);
    }
    catch (HttpException ex)
    {
        switch (ex.GetHttpCode())
        {
            case 400:
                H.WriteLog("Petición incorrecta");
                break;
            case 401:
                H.WriteLog("Petición no autorizada");
                break;
            default:
                throw;
        }
    }
}
```

En el código anterior, podemos notar que estamos invocando a los métodos **SendHttpMessage** y **GetHttpResource** de un objeto **Helper**. Al invocar a estos métodos podemos recibir una excepción del tipo **HttpException** que estamos atrapando. Si los códigos de error son el 400 o 401 escribimos una entrada en el log de la aplicación, en caso contrario volvemos a lanzar la excepción.

Cuando se genere una excepción con código diferente a 400 o 401, la ejecución se detendrá en la sentencia **throw** y si observamos la ventana **Locals**, veremos que las variables **Message** y **Html** se encuentran fuera de alcance por lo que no aparecerán. No veremos a simple vista cuál fue la línea que originó la excepción.



Con C# 6, podemos reescribir el código anterior utilizando un Filtro de Excepción de la siguiente manera:

```
static void DoSomething()
{
    var H = new Helper();
    try
    {
        string Message = "Hello, World";
        H.SendHttpMessage(Message, true);
        string Html = H.GetHttpResource("/Products", true);
    }
    catch (HttpException ex) when (WriteLog(ex.GetHttpCode(), H))
    {
    }
}
```

Cuando se genere una excepción del tipo **HttpException**, se evaluará la expresión **when()**. Si el resultado de la evaluación es **true** entonces se ejecutará el código del bloque **catch**, lo que significa que la excepción será atrapada. Si el resultado es **false**, el código del bloque catch **NO** será ejecutado,

lo que significa que la excepción no será atrapada y por consiguiente será pasada al código que haya invocado al método **DoSomething**.

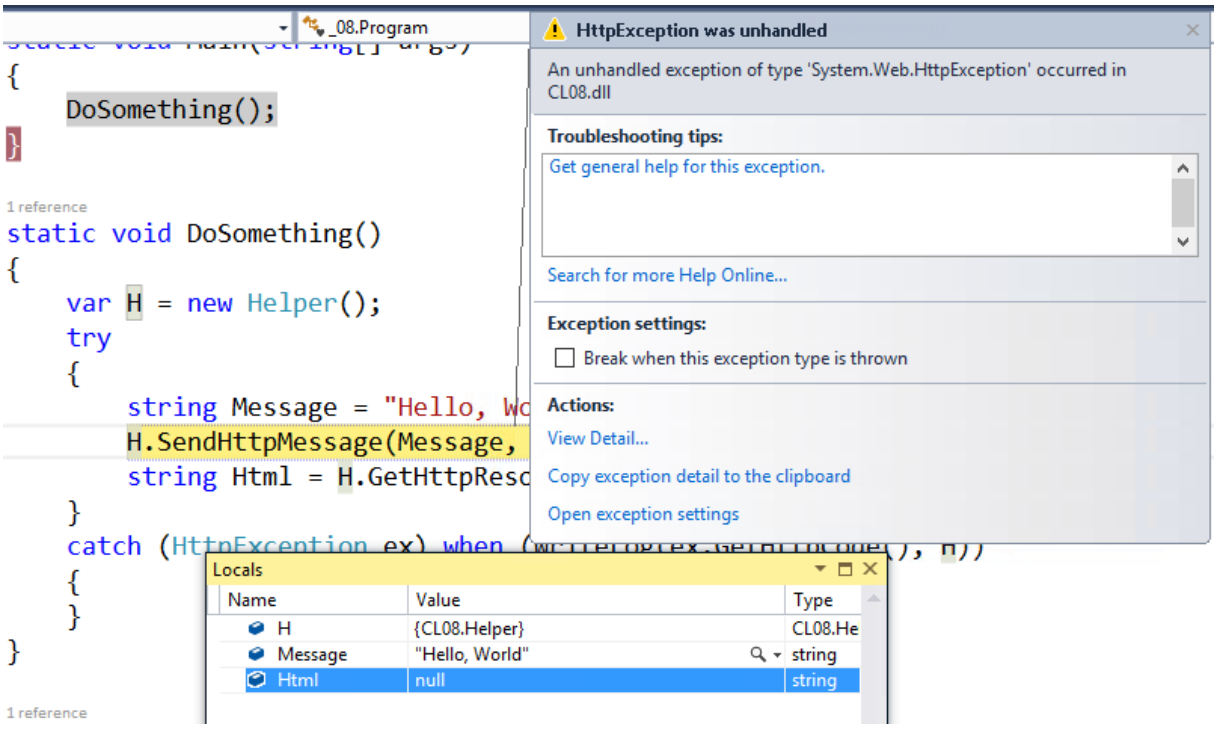
En caso de que la excepción no haya sido atrapada, se seguirán evaluando otros bloques **catch** en caso de que existan.

El código del método **WriteLog** quedaría de la siguiente manera.

```
static bool WriteLog(int httpCode, Helper H)
{
    bool ExecuteCatch = true;
    switch (httpCode)
    {
        case 400:
            H.WriteLog("Petición incorrecta");
            break;
        case 401:
            H.WriteLog("Petición no autorizada");
            break;
        default:
            ExecuteCatch = false;
            break;
    }
    return ExecuteCatch;
}
```

Podemos notar que si el código http no es 400 o 401, la variable **ExecuteCatch** obtiene el valor **false** lo que significa que no se ejecutará el bloque **catch**.

Cuando se genere una excepción con código diferente a 400 o 401, la ejecución se detendrá en la línea que causó la excepción y si observamos la ventana **Locals**, veremos que las variables **Message** y **Html** se encuentran ahí con sus valores actuales.



Los filtros de excepción son preferibles en lugar de atrapar y volver a lanzar la excepción ya que dejan el **Stack** intacto. Si posteriormente se dispara la excepción, podemos ver el lugar donde se originó la excepción en lugar de ver únicamente el lugar donde se volvió a disparar dicha excepción.

Es algo común y aceptado “abusar” del uso de filtros de excepción para invocar métodos que realicen tareas tales como guardar mensajes en el log de la aplicación sin manejar la excepción. En esos casos, el filtro de excepción (el método invocado) devolverá siempre **false**.

Inicializadores para propiedades Auto-Implementadas

C# 6 viene con una nueva característica que nos permite establecer el valor de una propiedad Auto-Implementada durante su declaración de la misma forma en que lo hacemos con los campos o variables.

Los siguientes son ejemplos de esta característica:

```
public int ProductID { get; set; } = 1;
public string ProductName { get; set; } = "Azucar";
public Double UnitPrice { get; set; } = 12;
```

Es posible también utilizar expresiones que hagan referencia a Constantes o miembros estáticos como en el siguiente ejemplo:

```
const int TAX = 16;

public Double UnitPrice2 { get; set; } = 12 * TAX;
public Double UnitPrice3 { get; set; } = 12 * GetTAX();
public string Location { get; set; } =
    ConfigurationManager.AppSettings["Location"];

static int GetTAX()
{
    // Algunos calculos
    return
    int.Parse(ConfigurationManager.AppSettings["TAX"]??"10");

}
```

El inicializador inicializa directamente el campo asociado a la propiedad (backing field). No lo hace a través del **accesor set** de la propiedad.

Los inicializadores son ejecutados en el orden en el que son escritos.

Los inicializadores de propiedades auto-implementadas no pueden referenciar a **this** ni a otros miembros de instancia ya que estos son ejecutados antes de que el objeto sea inicializado apropiadamente.

Propiedades Auto-implementadas de sólo lectura

Las propiedades Auto-implementadas ahora pueden ser declaradas sin un descriptor de acceso **set** como se muestra en el siguiente ejemplo:

```
public int ProductID { get; }
```

La variable (backing field) asociada a la propiedad es implícitamente declarada como **readonly**.

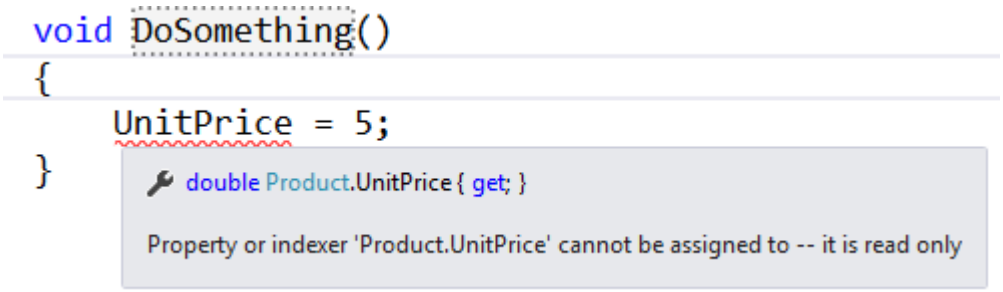
Podemos inicializar la propiedad directamente al declararla o bien en el constructor como se ve en el siguiente ejemplo:

```
public double UnitPrice { get; } = 10;
public double UnitPrice2 { get; }
public Product(int tax)
{
    UnitPrice2 = UnitPrice * tax;
}
```

Cualquiera que sea la forma en que inicialicemos la propiedad, el valor es asignado directamente a la variable (backing field) asociada a la propiedad.

Esta nueva característica hace que las propiedades auto-implementadas ahora puedan ser inmutables ya que al no tener un descriptor de acceso **set** y una variable backing field conocida, sólo podamos asignarle un valor durante su declaración o en el constructor.

El siguiente código mostraría un error al compilar.



Miembros con una Expresión como cuerpo (Expression-bodied function members)

Con C# 6 podemos implementar el cuerpo de Métodos, Propiedades, Indizadores o Sobrecarga de Operadores con la misma sintaxis que utilizamos al trabajar con expresiones lambda tal como se muestra en el siguiente ejemplo:

```
static void WriteMessage(string message) => Console.WriteLine(message);
```

El efecto es exactamente el mismo que tiene un método cuyo cuerpo es un bloque de código con una simple instrucción.

```
static void WriteMessage(string message)
{
    Console.WriteLine(message);
}
```

El siguiente ejemplo muestra un Método con una expresión como cuerpo:

```
public Complex Add(Complex b) =>
    new Complex(this.Real + b.Real, this.Imaginary + b.Imaginary);
```

El ejemplo anterior representa a un Método con sólo una instrucción **return** como se muestra a continuación:

```
public Complex Add(Complex b)
{
    return
        new Complex(this.Real + b.Real, this.Imaginary + b.Imaginary);
}
```

Propiedades e Indizadores de sólo lectura también pueden tener una expresión como cuerpo de su descriptor de acceso **get** tal y como se muestra en el siguiente ejemplo:

```
public string AsString => $"{Real} + {Imaginary}i";

public double this[byte index] =>
    index==0 ? Real:Imaginary;
```

Podemos notar que no especificamos la palabra clave **get**, esta es inferida por el uso de la sintaxis de la expresión como cuerpo.

En sobrecarga de operadores o sobre-escritura de miembros también podemos utilizar esta sintaxis tal y como se muestra en el siguiente ejemplo:

```
public static Complex operator +(Complex A, Complex B)=>A.Add(B);
public static implicit operator string(Complex A)=>A.AsString;
public override string ToString() => AsString;
```