EBook Gratis

# APRENDIZAJE

# C# Language

Free unaffiliated eBook created from
**Stack Overflow contributors.**

#c#

# Tabla de contenido

# Acerca de

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: csharp-language

It is an unofficial and free C# Language ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official C# Language.



Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

# Capítulo 1: Empezando con C # Language

## Observaciones

C # es un lenguaje de programación multi-paradigma y descendiente de C de Microsoft. C # es un lenguaje administrado que se compila en CIL , bytecode intermedio que se puede ejecutar en Windows, Mac OS X y Linux.

Las versiones 1.0, 2.0 y 5.0 fueron estandarizadas por ECMA (como ECMA-334 ), y los esfuerzos de estandarización para el C # moderno están en marcha.

## Versiones

| Versión | Fecha de lanzamiento |
|---|---|
| 1.0 | 2002-01-01 |
| 1.2 | 2003-04-01 |
| 2.0 | 2005-09-01 |
| 3.0 | 2007-08-01 |
| 4.0 | 2010-04-01 |
| 5.0 | 2013-06-01 |
| 6.0 | 2015-07-01 |
| 7.0 | 2017-03-07 |

## Examples

### Creando una nueva aplicación de consola (Visual Studio)

1. Abrir Visual Studio
2. En la barra de herramientas, vaya a **Archivo** → **Nuevo proyecto**
3. Seleccione el tipo de proyecto de **aplicación de consola**
4. Abra el archivo `Program.cs` en el Explorador de soluciones
5. Agregue el siguiente código a `Main()` :

```
public class Program
{
    public static void Main()
    {
        // Prints a message to the console.
```

```
        System.Console.WriteLine("Hello, World!");

        /* Wait for the user to press a key. This is a common
           way to prevent the console window from terminating
           and disappearing before the programmer can see the contents
           of the window, when the application is run via Start from within VS. */
        System.Console.ReadKey();
    }
}
```

6. En la barra de herramientas, haga clic en **Depurar** -> **Iniciar depuración** o presione **F5** o **ctrl + F5** (en ejecución sin depurador) para ejecutar el programa.

Demo en vivo en ideone

---

# Explicación

- `class Program` es una declaración de clase. El `Program` clase contiene las definiciones de datos y métodos que utiliza su programa. Las clases generalmente contienen múltiples métodos. Los métodos definen el comportamiento de la clase. Sin embargo, la clase de `Program` tiene un solo método: `Main` .

- `static void Main()` define el método `Main` , que es el punto de entrada para todos los programas de C #. El método `Main` indica lo que hace la clase cuando se ejecuta. Solo se permite un método `Main` por clase.

- `System.Console.WriteLine("Hello, world!");` El método imprime un dato dado (en este ejemplo, `Hello, world!` ) como una salida en la ventana de la consola.

- `System.Console.ReadKey()` asegura que el programa no se cerrará inmediatamente después de mostrar el mensaje. Lo hace esperando que el usuario presione una tecla del teclado. Cualquier tecla presionada por el usuario terminará el programa. El programa termina cuando ha terminado la última línea de código en el método `main()` .

---

# Usando la línea de comando

Para compilar a través de la línea de comandos, use `MSBuild` o `csc.exe` *(el compilador de C #)* , ambos parte del paquete Microsoft Build Tools .

Para compilar este ejemplo, ejecute el siguiente comando en el mismo directorio donde se encuentra `HelloWorld.cs` :

```
%WINDIR%\Microsoft.NET\Framework64\v4.0.30319\csc.exe HelloWorld.cs
```

También puede ser posible que tenga dos métodos principales dentro de una aplicación. En este caso, hay que indicar al compilador qué método principal para ejecutar escribiendo el siguiente

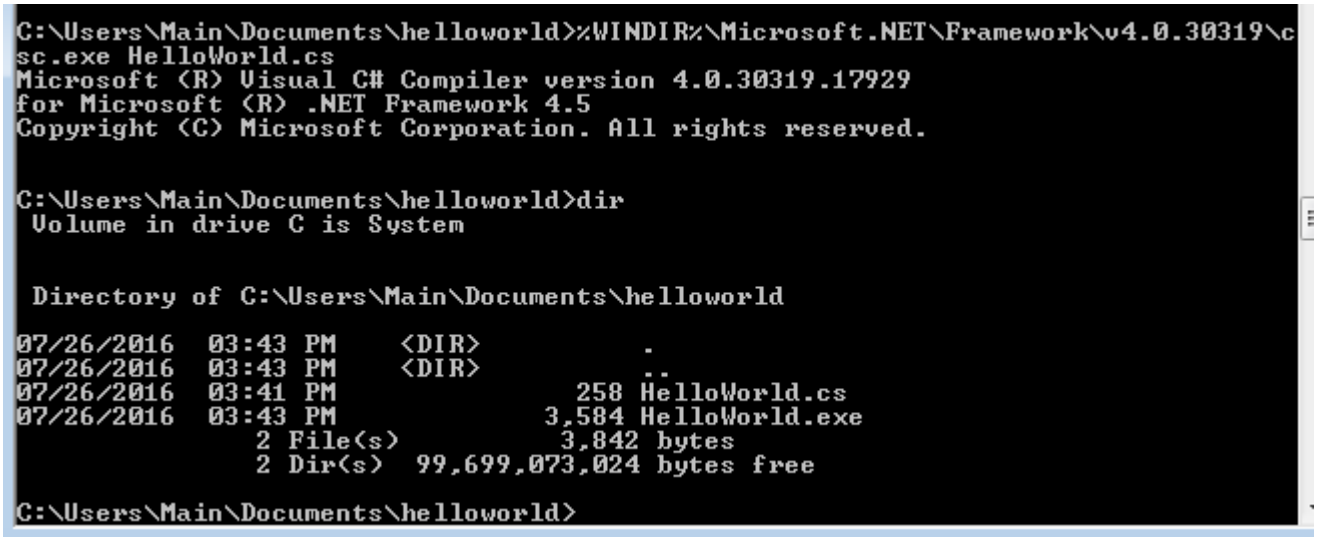
comando en la **consola.** (Supongamos que la clase `ClassA` también tiene un método principal en el mismo `HelloWorld.cs` archivo en HelloWorld espacio de nombres)

```
%WINDIR%\Microsoft.NET\Framework64\v4.0.30319\csc.exe HelloWorld.cs /main:HelloWorld.ClassA
```

donde HelloWorld es espacio de nombres

***Nota** : Esta es la ruta donde se encuentra **.NET framework v4.0** en general. Cambia la ruta según tu versión .NET. Además, el directorio podría ser **framework en** lugar de **framework64** si está utilizando .NET Framework de 32 bits. Desde el símbolo del sistema de Windows, puede enumerar todas las rutas de csc.exe Framework ejecutando los siguientes comandos (el primero para Frameworks de 32 bits):*

```
dir %WINDIR%\Microsoft.NET\Framework\csc.exe /s/b
dir %WINDIR%\Microsoft.NET\Framework64\csc.exe /s/b
```



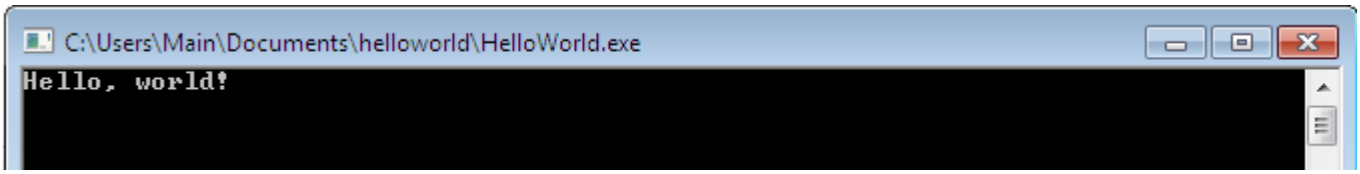

Ahora debería haber un archivo ejecutable llamado `HelloWorld.exe` en el mismo directorio. Para ejecutar el programa desde el símbolo del sistema, simplemente escriba el nombre del ejecutable y presione `Enter de la` siguiente manera:

```
HelloWorld.exe
```

Esto producirá:

> ¡Hola Mundo!


También puede hacer doble clic en el archivo ejecutable e iniciar una nueva ventana de consola con el mensaje " **¡Hola, mundo!** "

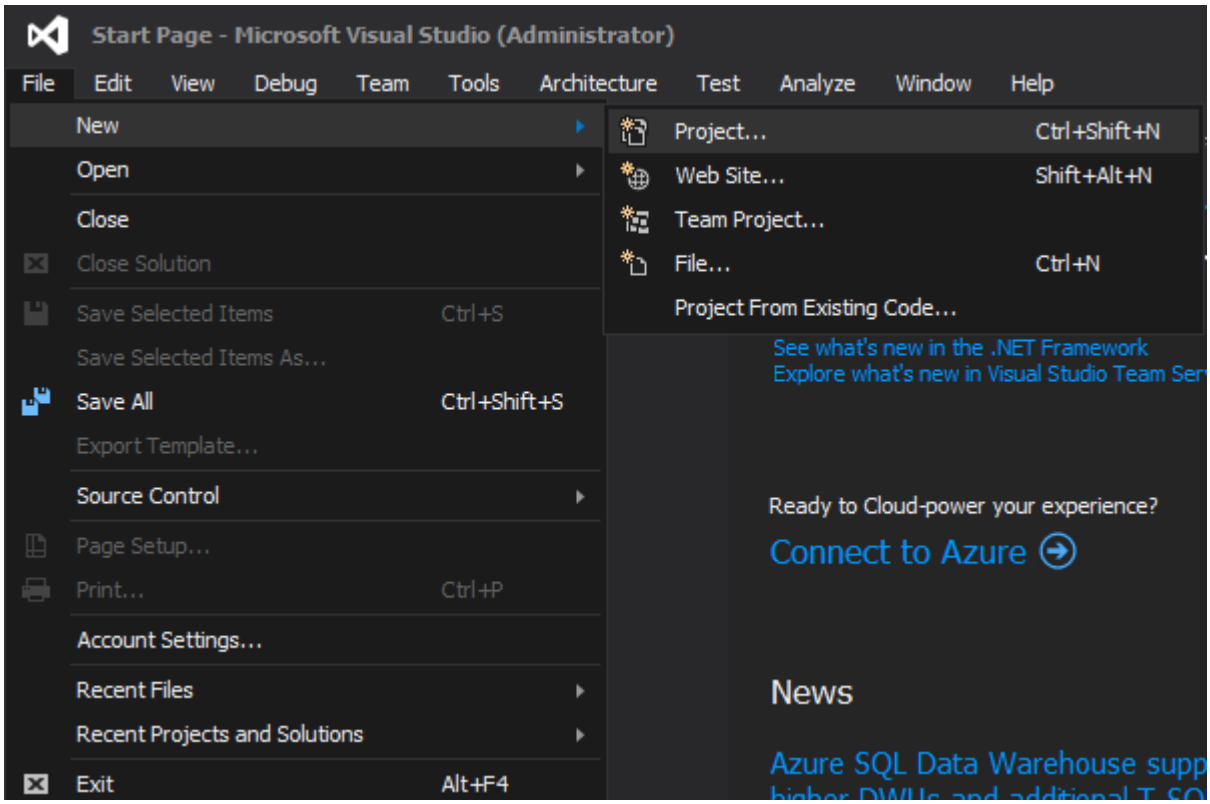



## Creando un nuevo proyecto en Visual Studio (aplicación de consola) y ejecutándolo en modo de depuración

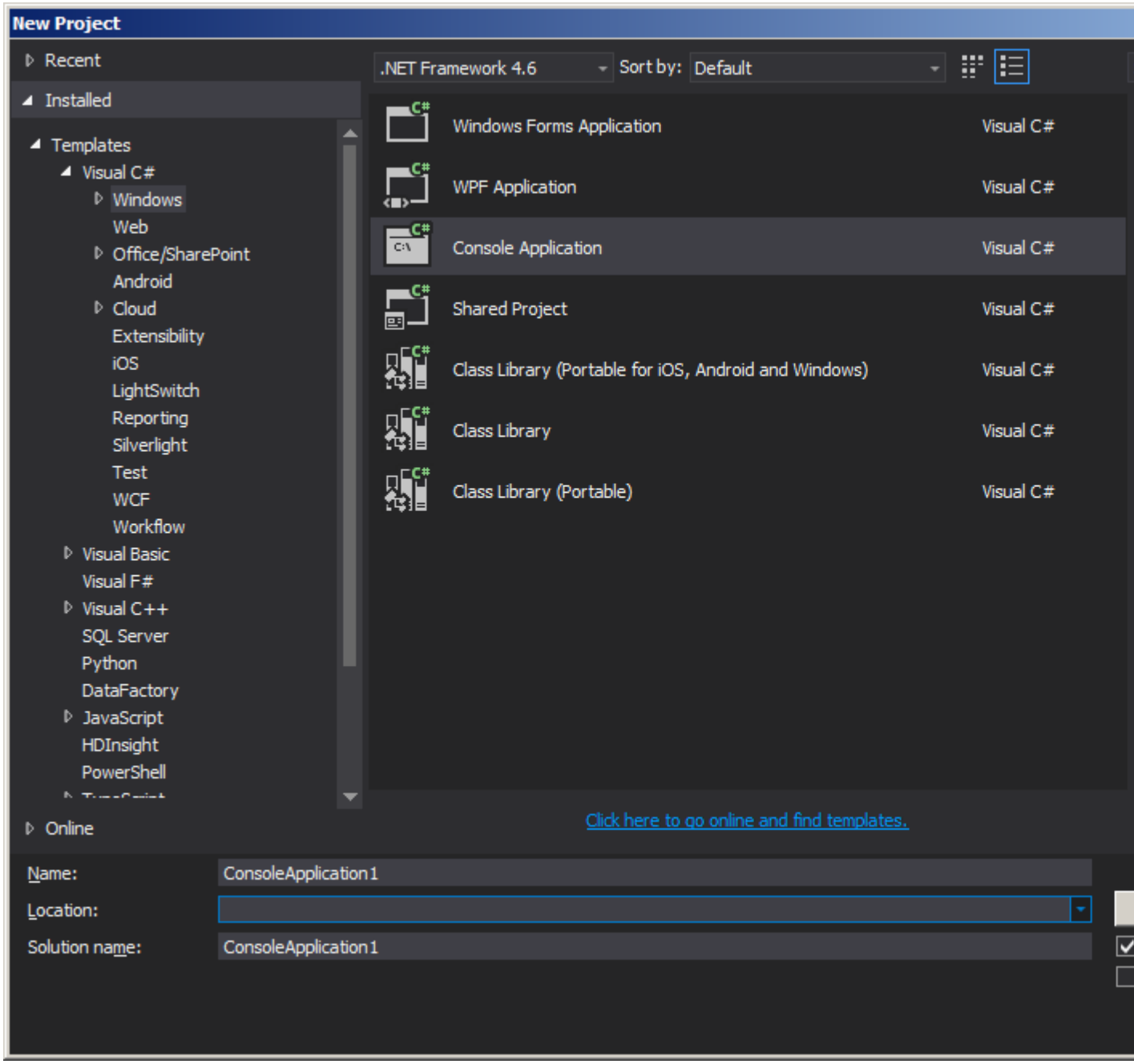
1. **Descargue e instale Visual Studio** . Visual Studio se puede descargar desde VisualStudio.com . Se sugiere la edición de la Comunidad, primero porque es gratuita, y segundo porque incluye todas las características generales y se puede ampliar aún más.

2. **Abra Visual Studio.**

3. **Bienvenido.** Ir a **Archivo → Nuevo → Proyecto** .

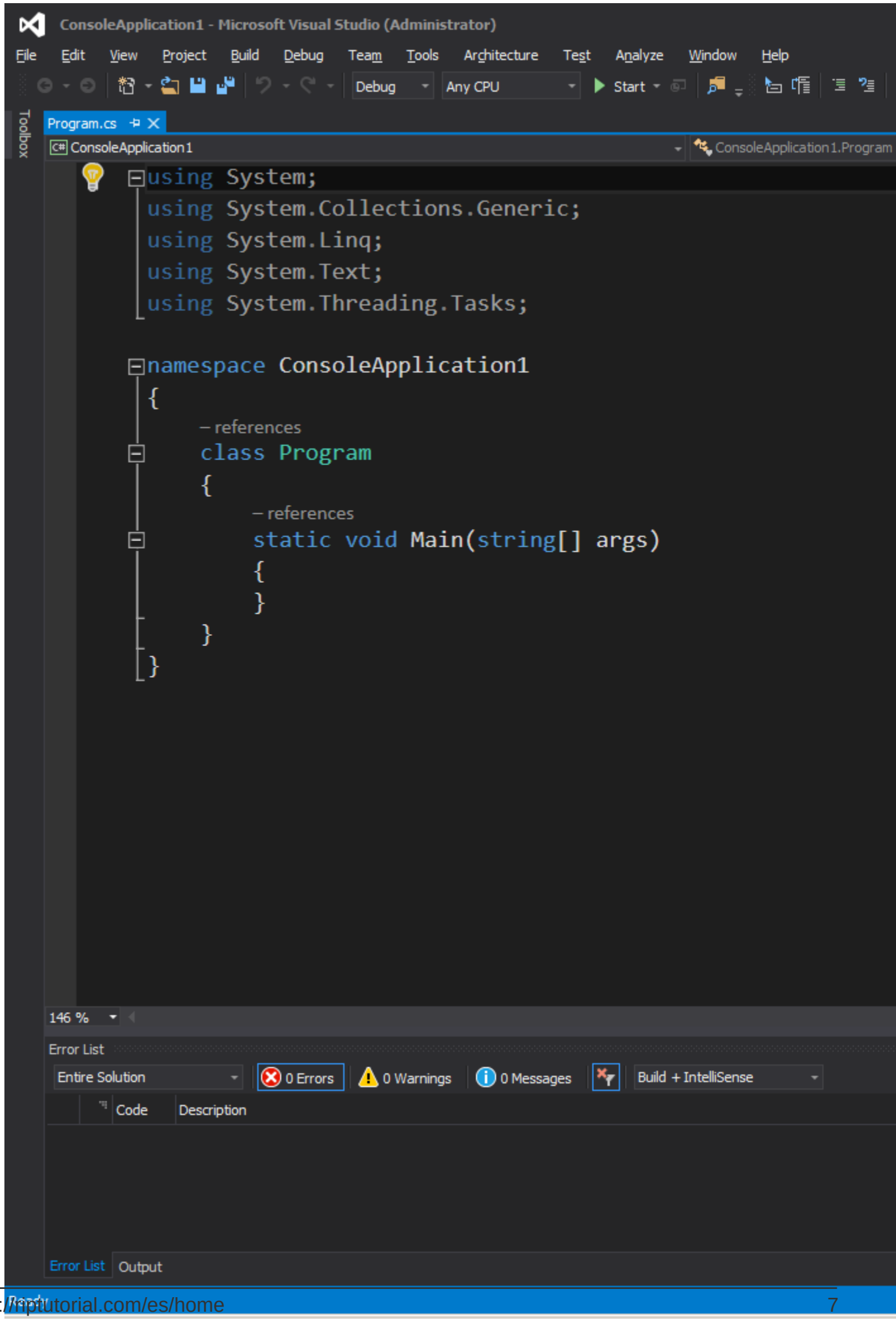



4. Haga clic en **Plantillas** → **Visual C #** → **Aplicación de consola**




5. **Después de seleccionar Aplicación de consola,** ingrese un nombre para su proyecto y una ubicación para guardar y presione `Aceptar` . No te preocupes por el nombre de la solución.

6. **Proyecto creado** . El proyecto recién creado se verá similar a:


ConsoleApplication1 - Microsoft Visual Studio (Administrator)
File Edit View Project Build Debug Team Tools Architecture Test Analyze Window Help
Debug
Any CPU
Start
Toolbox
Program.cs
ConsoleApplication1
ConsoleApplication1.Program
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
namespace ConsoleApplication1
{
0 references
class Program
{
0 references
static void Main(string[] args)
{
}
}
}
146 %
Error List
Entire Solution
0 Errors
0 Warnings
0 Messages
Build + IntelliSense
Code
Description
Error List
Output
Ready

*(Siempre use nombres descriptivos para los proyectos para que puedan distinguirse fácilmente de otros proyectos. Se recomienda no usar espacios en el proyecto o nombre de clase).*

7. **Escribir código.** Ahora puede actualizar su `Program.cs` para presentar "¡Hola mundo!" al usuario.

   para presentar "¡Hola mundo!" al usuario.

```
using System;

namespace ConsoleApplication1
{
    public class Program
    {
        public static void Main(string[] args)
        {
        }
    }
}
```

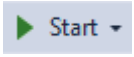
   Agregue las siguientes dos líneas al objeto principal `public static void Main(string[] args)` en `Program.cs` : (asegúrese de que esté dentro de las llaves)

```
Console.WriteLine("Hello world!");
Console.Read();
```

   **¿Por qué** `Console.Read()` **?** La primera línea imprime el texto "¡Hola mundo!" a la consola, y la segunda línea espera que se ingrese un solo carácter; En efecto, esto hace que el programa ponga en pausa la ejecución para que pueda ver la salida mientras se realiza la depuración. Sin `Console.Read();` , cuando comiences a depurar la aplicación, simplemente se imprimirá "¡Hola mundo!" A la consola y luego cerrar inmediatamente. Su ventana de código ahora debería verse como la siguiente:

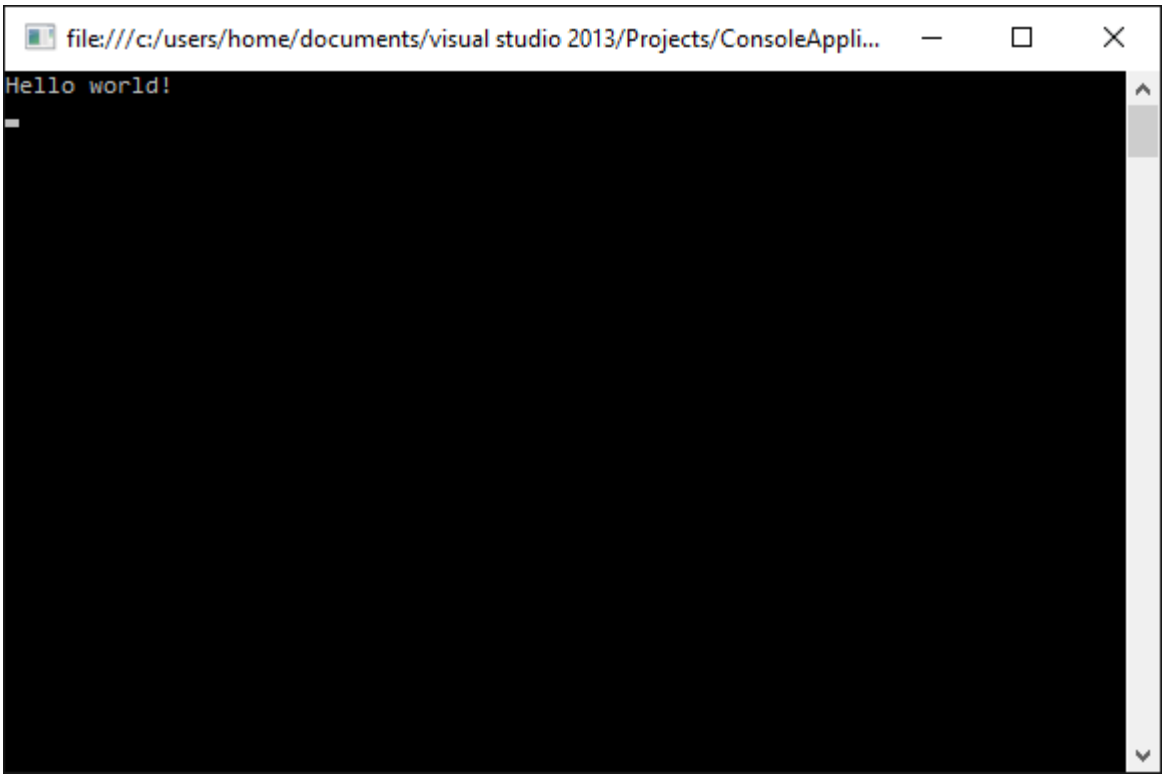
```
using System;

namespace ConsoleApplication1
{
    public class Program
    {
        public static void Main(string[] args)
        {
            Console.WriteLine("Hello world!");
            Console.Read();
        }
    }
}
```

8. **Depura tu programa.** Presione el botón de inicio en la barra de herramientas cerca de la

   parte superior de la ventana 

o presione `F5` en su teclado para ejecutar su aplicación. Si el botón no está presente, puede ejecutar el programa desde el menú superior: **Depurar → Iniciar depuración** . El programa compilará y luego abrirá una

ventana de consola. Debería verse similar a la siguiente captura de pantalla:




9. **Detener el programa.** Para cerrar el programa, simplemente presione cualquier tecla en su teclado. La `Console.Read()` que agregamos fue para este mismo propósito. Otra forma de cerrar el programa es ir al menú donde estaba el botón `Inicio` y hacer clic en el botón `Detener` .

## Creando un nuevo programa usando Mono

Primero instale Mono siguiendo las instrucciones de instalación para la plataforma de su elección como se describe en la sección de instalación .

Mono está disponible para Mac OS X, Windows y Linux.

Una vez realizada la instalación, cree un archivo de texto, `HelloWorld.cs` nombre `HelloWorld.cs` y copie el siguiente contenido en él:

```
public class Program
{
    public static void Main()
    {
        System.Console.WriteLine("Hello, world!");
        System.Console.WriteLine("Press any key to exit..");
        System.Console.Read();
    }
}
```

Si está utilizando Windows, ejecute el símbolo del sistema Mono que se incluye en la instalación de Mono y asegúrese de que se configuran las variables de entorno necesarias. Si está en Mac o Linux, abra un nuevo terminal.

Para compilar el archivo recién creado, ejecute el siguiente comando en el directorio que contiene `HelloWorld.cs` :

```
mcs -out:HelloWorld.exe HelloWorld.cs
```

El `HelloWorld.exe` resultante se puede ejecutar con:

```
mono HelloWorld.exe
```

que producirá la salida:

```
Hello, world!
Press any key to exit..
```

## Creando un nuevo programa usando .NET Core

Primero instale **.NET Core SDK** siguiendo las instrucciones de instalación para la plataforma que elija:

- Windows
- OSX
- Linux
- Estibador

Una vez completada la instalación, abra un símbolo del sistema o una ventana de terminal.

1. Cree un nuevo directorio con `mkdir hello_world` y cambie al directorio recién creado con `cd hello_world` .
2. Crear una nueva aplicación de consola con la `dotnet new console` .
   Esto producirá dos archivos:
   - **hello_world.csproj**

     ```
     <Project Sdk="Microsoft.NET.Sdk">

       <PropertyGroup>
         <OutputType>Exe</OutputType>
         <TargetFramework>netcoreapp1.1</TargetFramework>
       </PropertyGroup>

     </Project>
     ```

   - **Programa.cs**

     ```
     using System;

     namespace hello_world
     {
         class Program
     ```

```
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

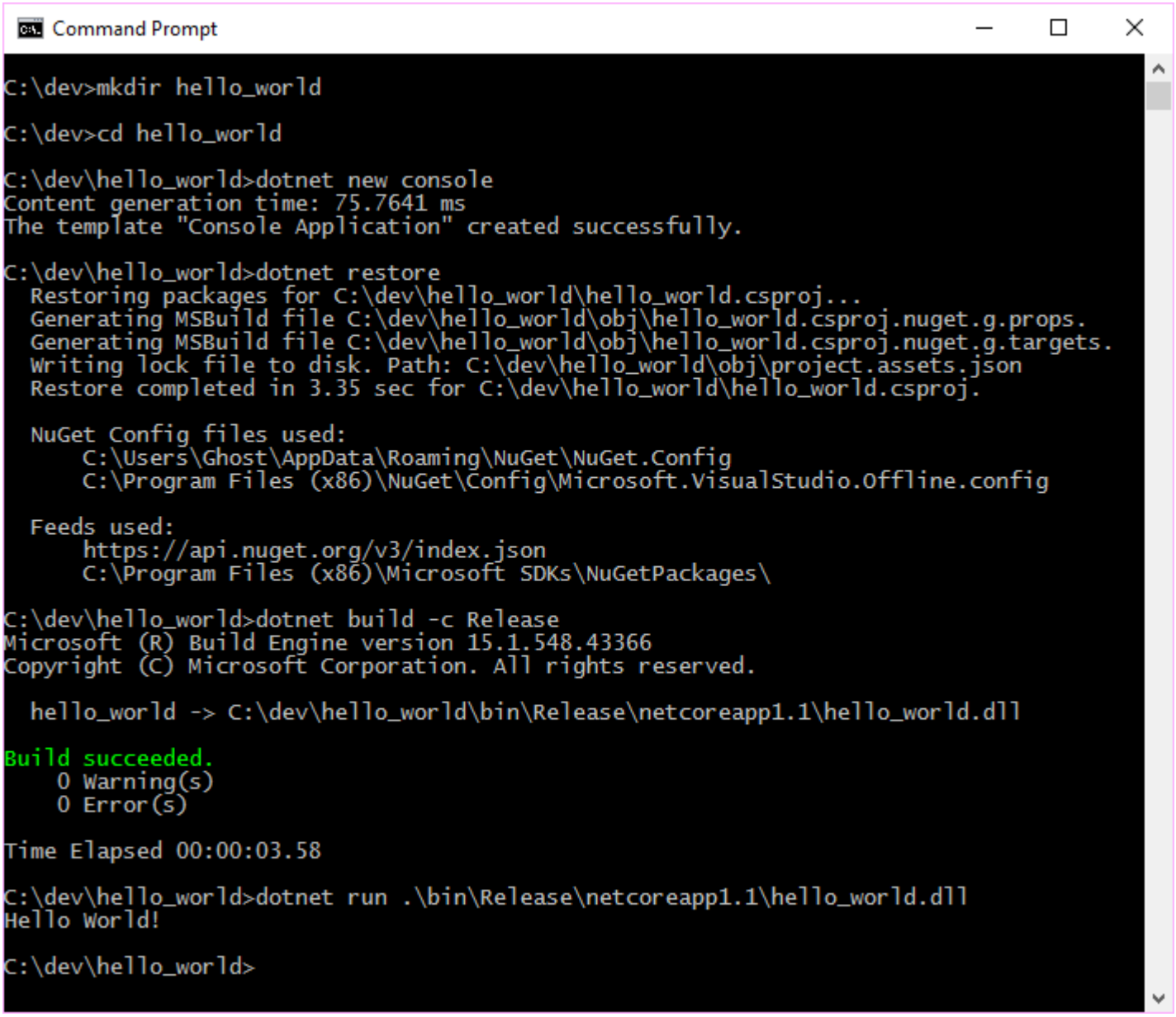
3. Restaurar los paquetes necesarios con la `dotnet restore` .

4. *Opcional* Construya la aplicación con `dotnet build` para Debug o `dotnet build -c Release` for Release. `dotnet run` también ejecutará el compilador y lanzará errores de compilación, si se encuentra alguno.

5. Ejecute la aplicación con `dotnet run` para `dotnet run` Debug o para `dotnet run .\bin\Release\netcoreapp1.1\hello_world.dll` para Release.

---

## Salida de solicitud de comando

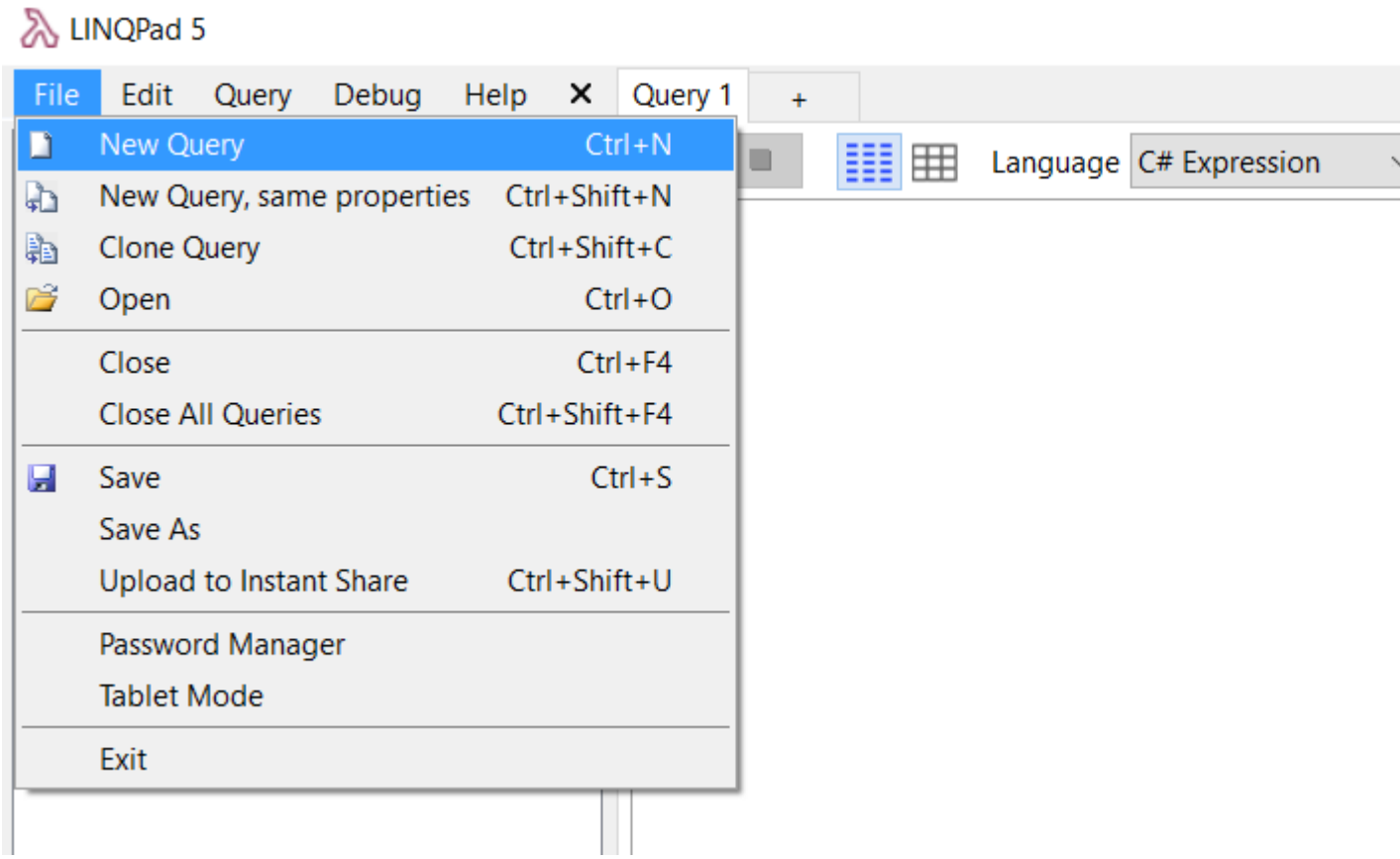



## Creando una nueva consulta usando LinqPad

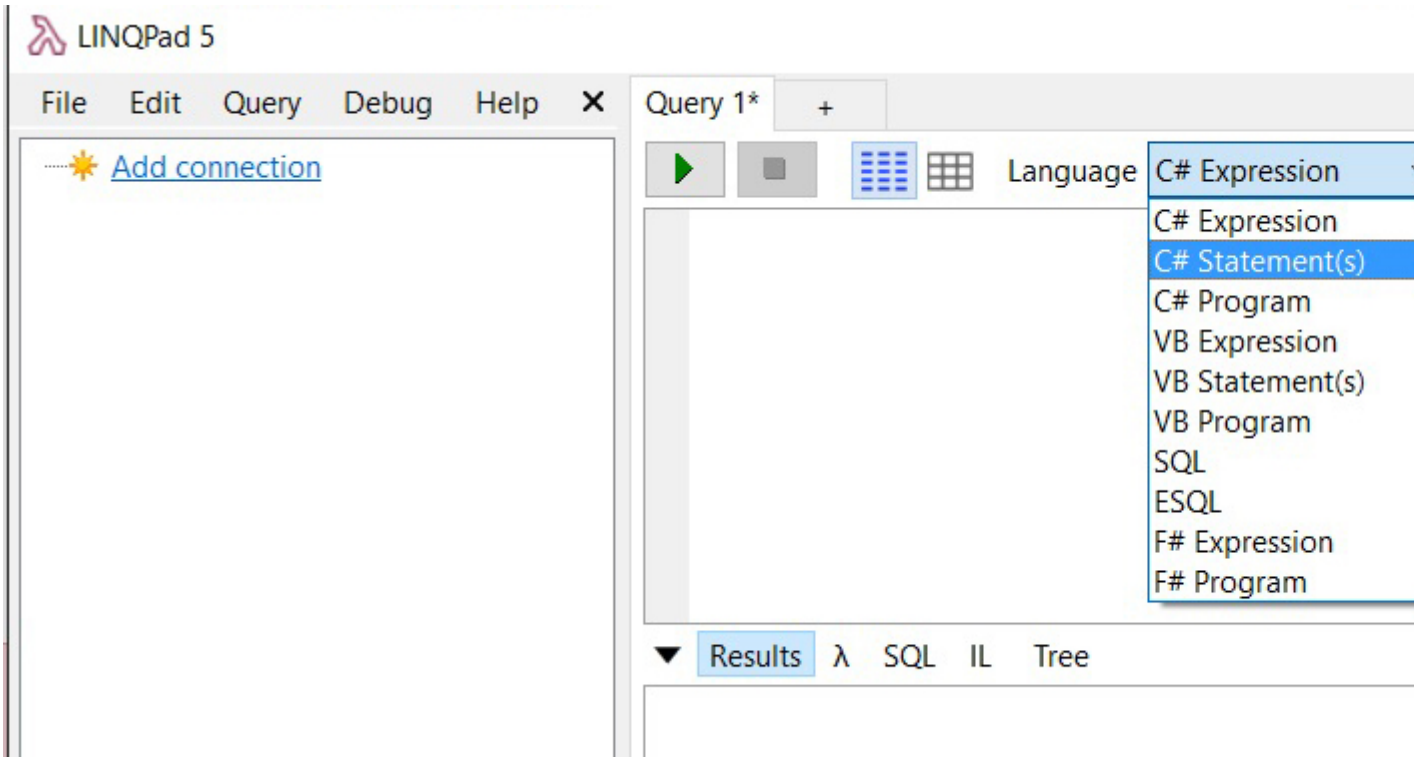
LinqPad es una gran herramienta que te permite aprender y probar las características de los lenguajes .Net (C #, F # y VB.Net).

1. Instala LinqPad

2. Crear una nueva consulta ( `Ctrl` + `N` )

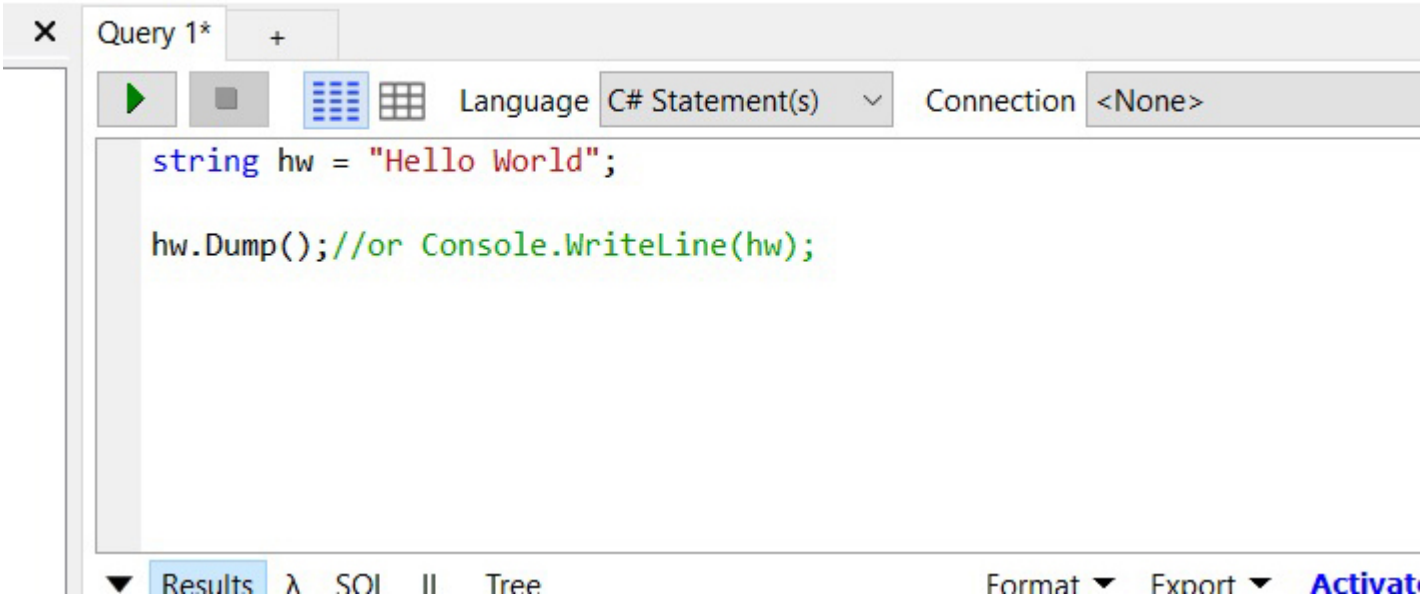



3. Debajo del idioma, seleccione "declaraciones C #"

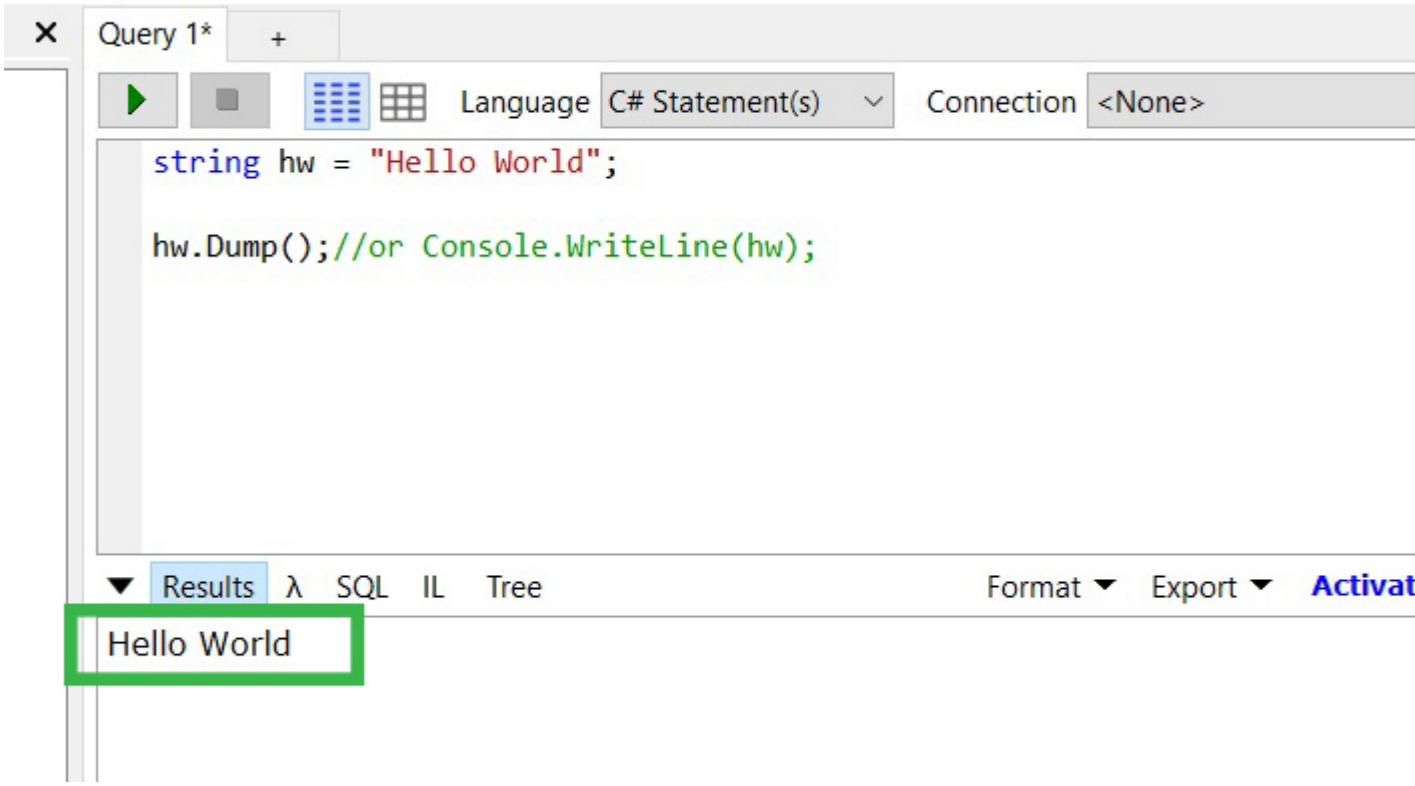



4. Escribe el siguiente código y pulsa ejecutar ( `F5` )

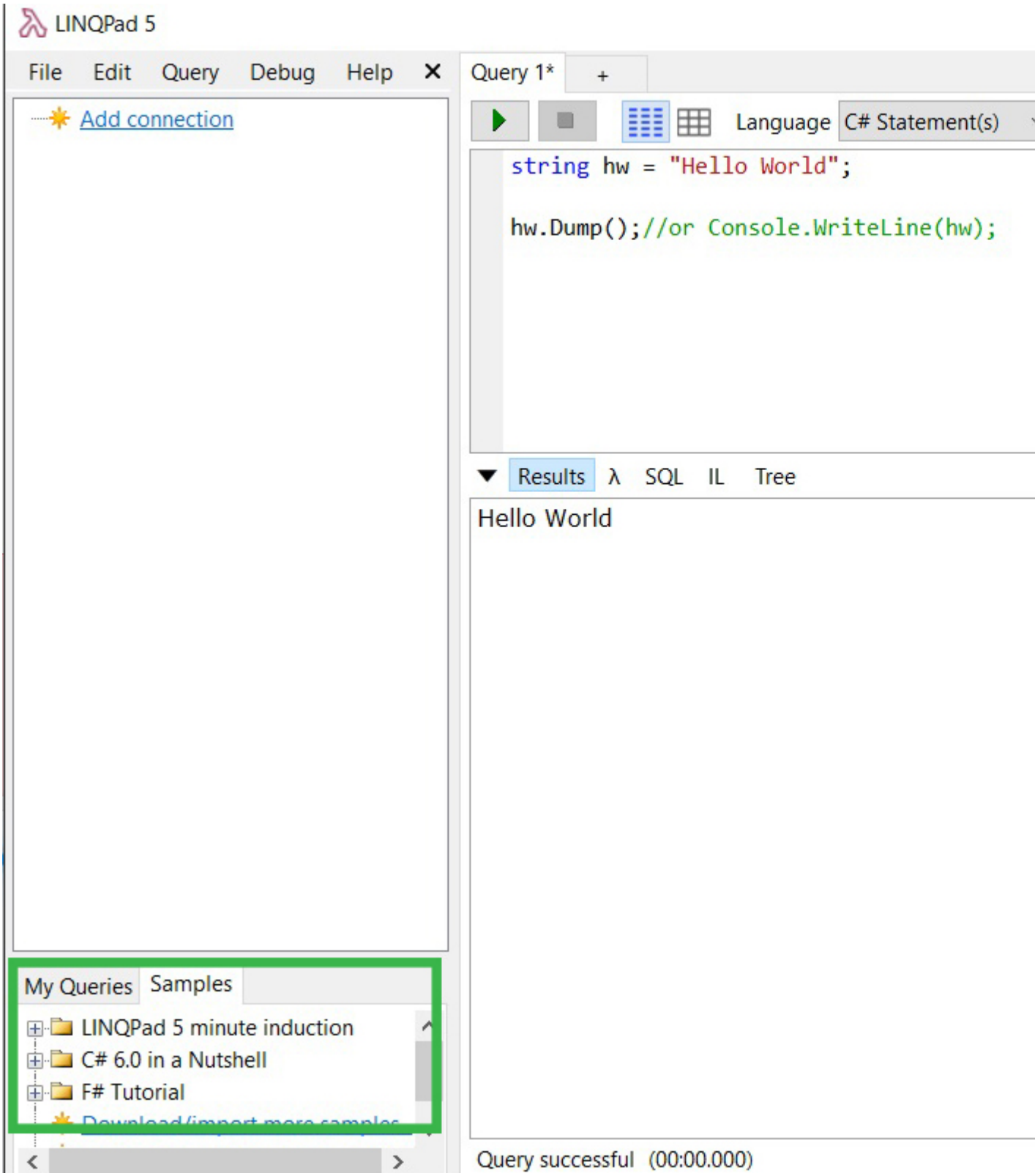
```
string hw = "Hello World";

hw.Dump(); //or Console.WriteLine(hw);
```




5. Debería ver "Hello World" impreso en la pantalla de resultados.

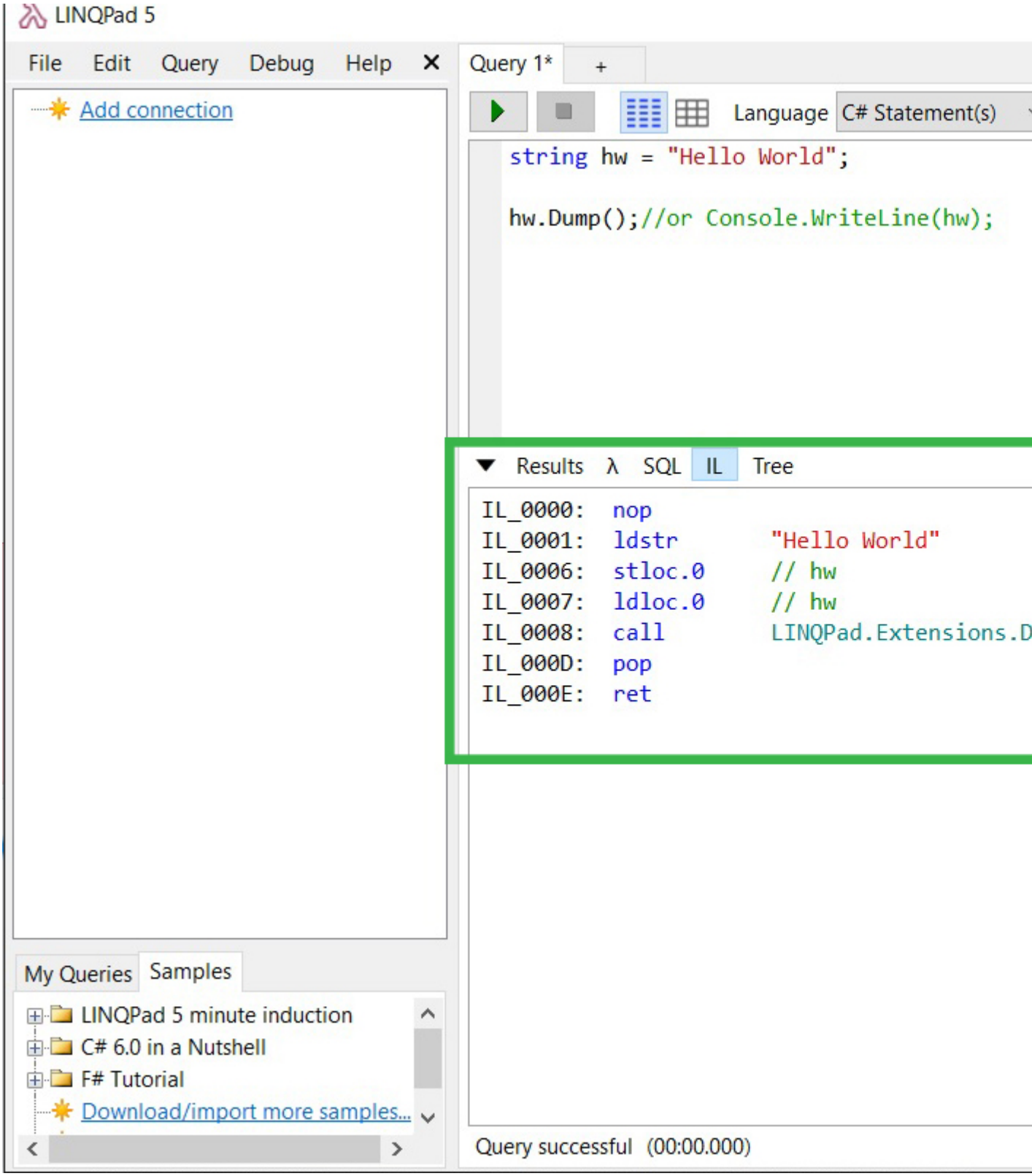



6. Ahora que ha creado su primer programa .Net, vaya y eche un vistazo a las muestras incluidas en LinqPad a través del navegador de "Muestras". Hay muchos ejemplos excelentes que le mostrarán muchas características diferentes de los lenguajes .Net.




**Notas:**

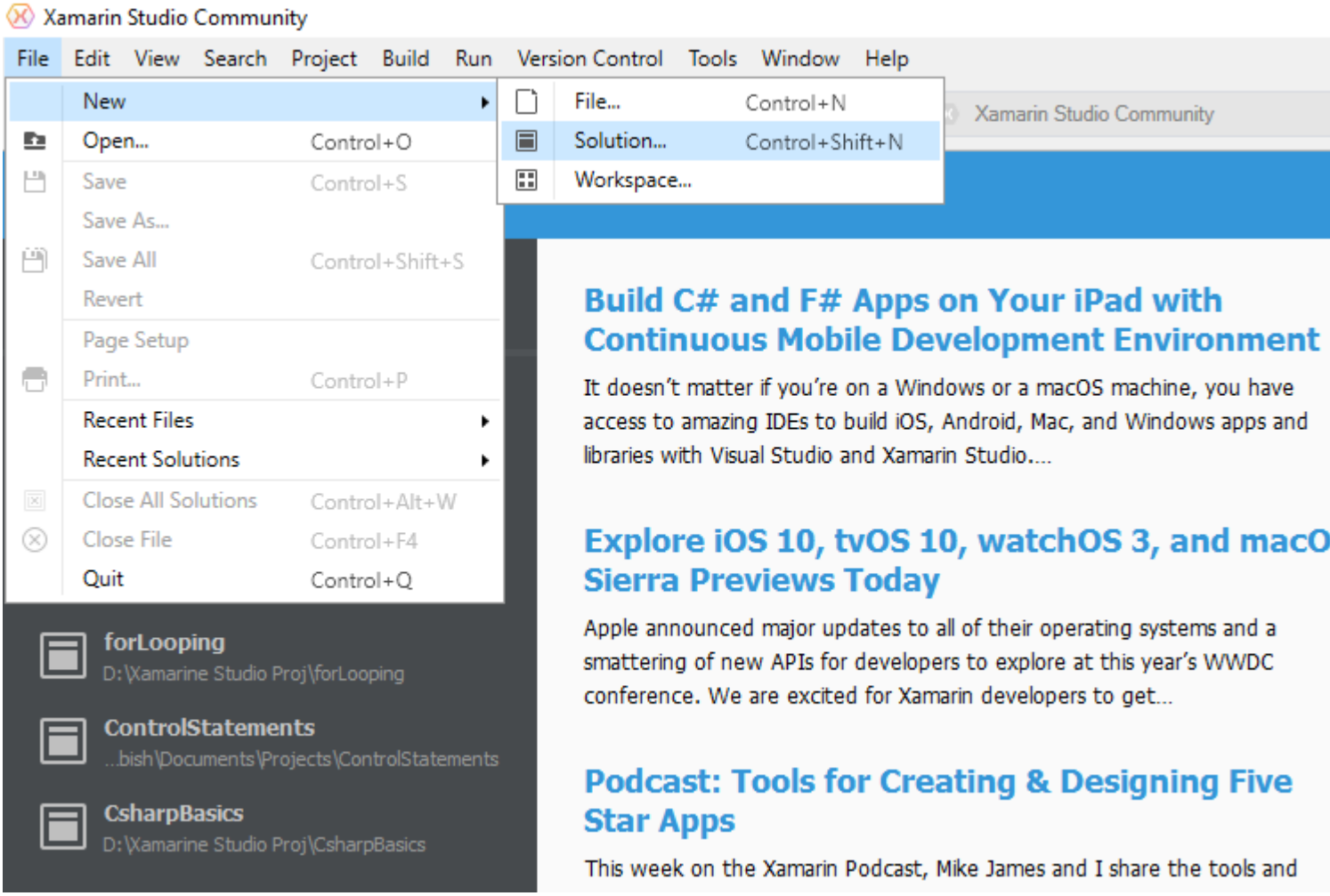
1. Si hace clic en "IL", puede inspeccionar el código IL que genera su código .net. Esta es una gran herramienta de aprendizaje.



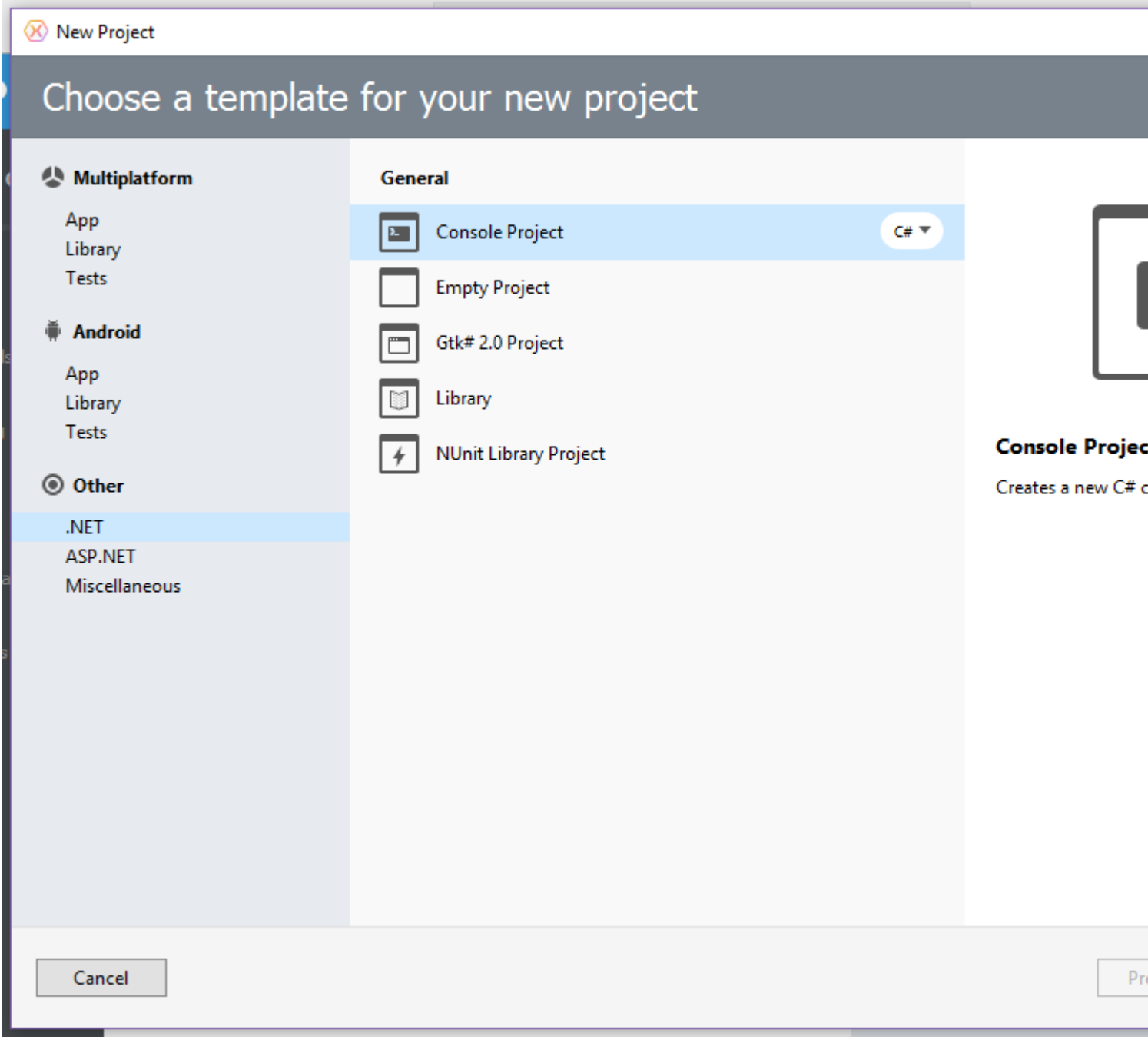

2. Al usar `LINQ to SQL` o `Linq to Entities` , puede inspeccionar el SQL que se está generando, que es otra excelente manera de aprender acerca de LINQ.

## Creando un nuevo proyecto usando Xamarin Studio

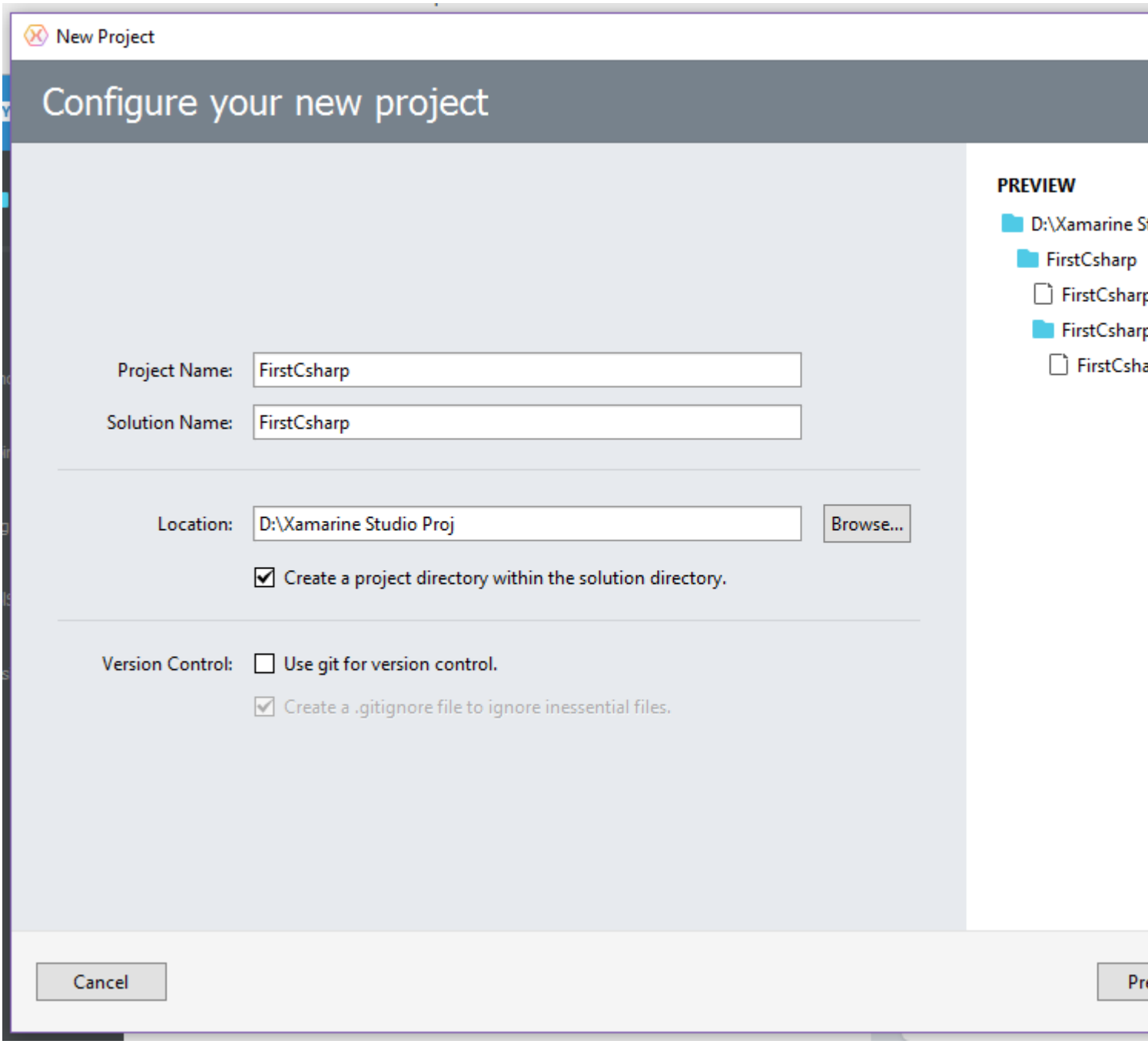
1. Descarga e instala la comunidad de Xamarin Studio .
2. Abrir el estudio de Xamarin.
3. Haga clic en **Archivo** → **Nuevo** → **Solución** .



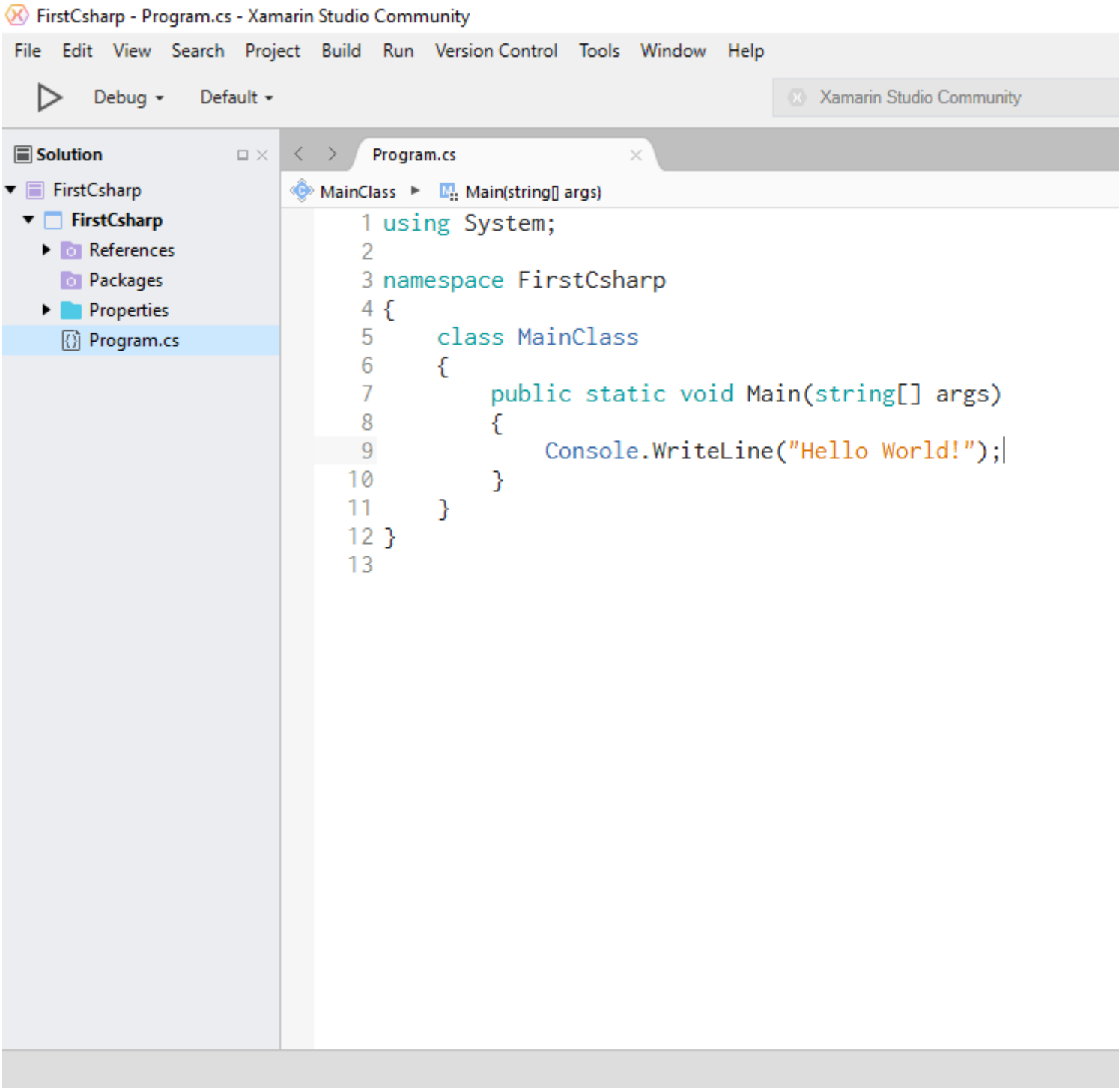

4. Haga clic en **.NET** → **Proyecto de consola** y elija **C #** .
5. Haga clic en `Siguiente` para continuar.




6. Ingrese el **nombre** del **proyecto** y `busque ...` una **ubicación** para guardar y luego haga clic en `Crear` .




7. El proyecto recién creado se verá similar a:

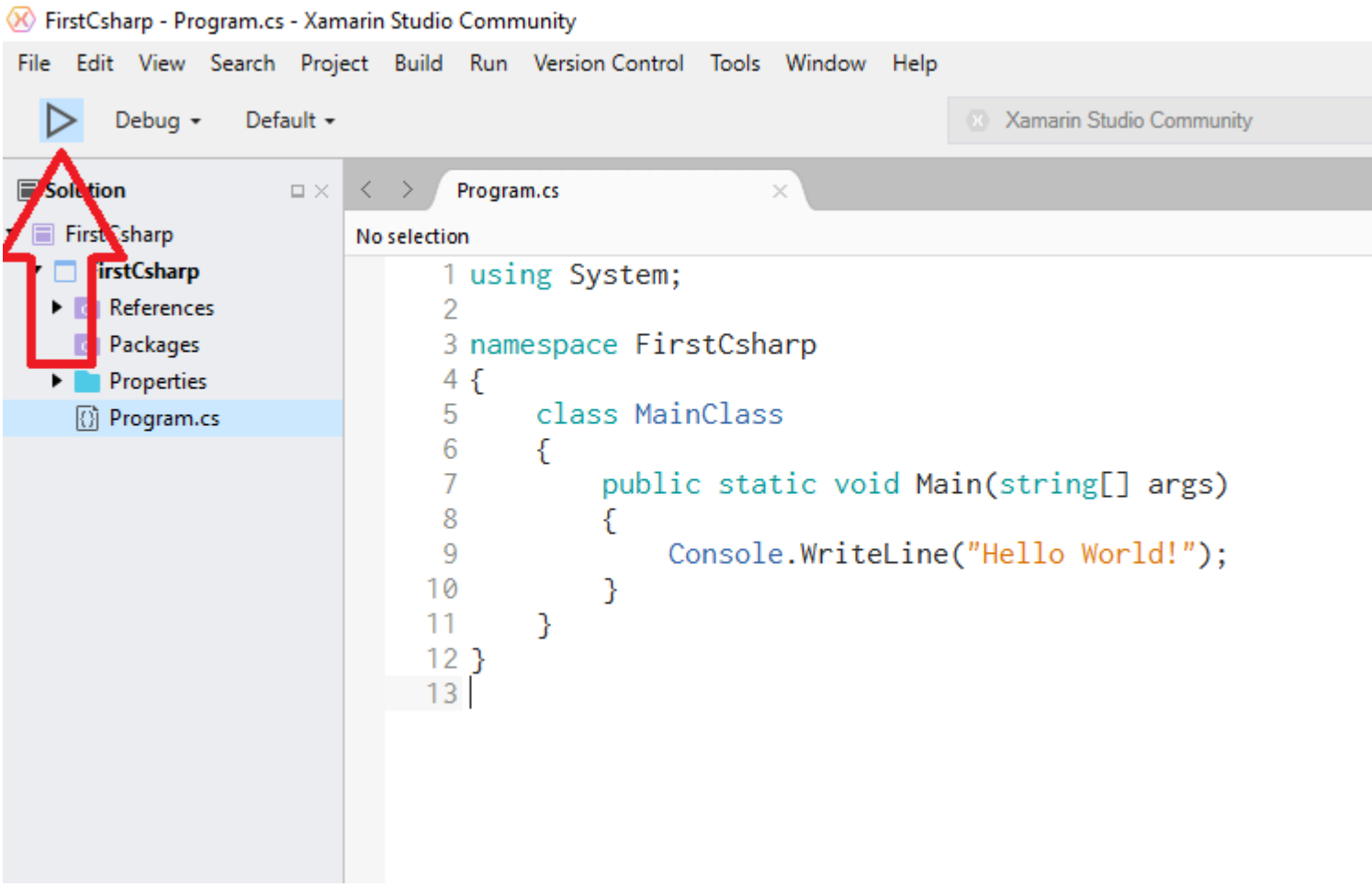



8. Este es el código en el editor de texto:

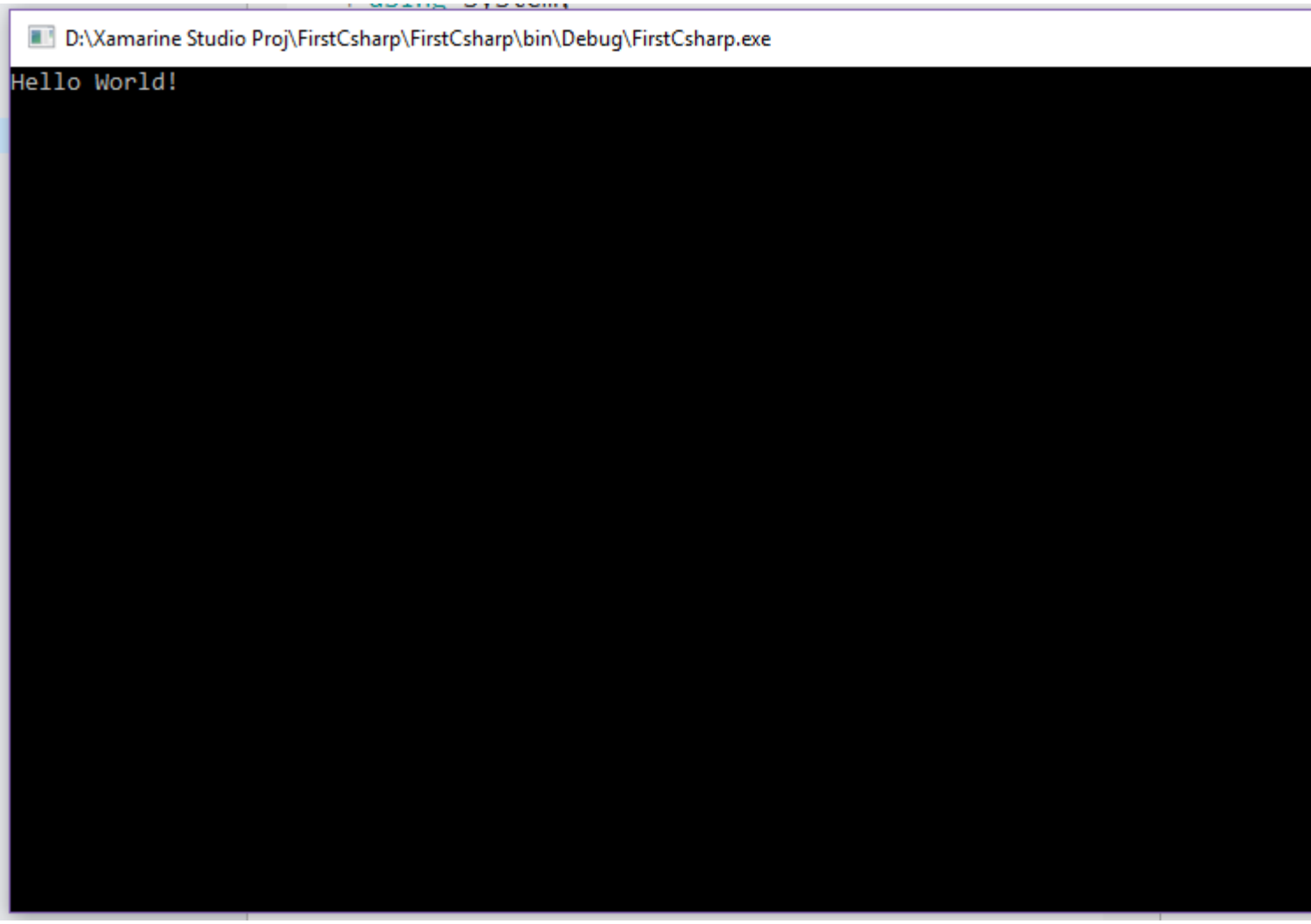
```
using System;

namespace FirstCsharp
{
    public class MainClass
    {
        public static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
            Console.ReadLine();
        }
    }
```

```
}
```

9. Para ejecutar el código, presione `F5` o haga clic en el **botón Reproducir** como se muestra a continuación:




10. Lo siguiente es la salida:




Lea Empezando con C # Language en línea:
https://riptutorial.com/es/csharp/topic/15/empezando-con-c-sharp-language

# Capítulo 2: Acceso a la carpeta compartida de la red con nombre de usuario y contraseña.

## Introducción

Accediendo al archivo compartido de red usando PInvoke.

## Examples

### Código para acceder a la red de archivos compartidos.

```
public class NetworkConnection : IDisposable
    {
        string _networkName;

        public NetworkConnection(string networkName,
            NetworkCredential credentials)
        {
            _networkName = networkName;

            var netResource = new NetResource()
            {
                Scope = ResourceScope.GlobalNetwork,
                ResourceType = ResourceType.Disk,
                DisplayType = ResourceDisplaytype.Share,
                RemoteName = networkName
            };

            var userName = string.IsNullOrEmpty(credentials.Domain)
                ? credentials.UserName
                : string.Format(@"{0}\{1}", credentials.Domain, credentials.UserName);

            var result = WNetAddConnection2(
                netResource,
                credentials.Password,
                userName,
                0);

            if (result != 0)
            {
                throw new Win32Exception(result);
            }
        }

    ~NetworkConnection()
    {
        Dispose(false);
    }

    public void Dispose()
```

```
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    protected virtual void Dispose(bool disposing)
    {
        WNetCancelConnection2(_networkName, 0, true);
    }

    [DllImport("mpr.dll")]
    private static extern int WNetAddConnection2(NetResource netResource,
        string password, string username, int flags);

    [DllImport("mpr.dll")]
    private static extern int WNetCancelConnection2(string name, int flags,
        bool force);
}

[StructLayout(LayoutKind.Sequential)]
public class NetResource
{
    public ResourceScope Scope;
    public ResourceType ResourceType;
    public ResourceDisplaytype DisplayType;
    public int Usage;
    public string LocalName;
    public string RemoteName;
    public string Comment;
    public string Provider;
}

public enum ResourceScope : int
{
    Connected = 1,
    GlobalNetwork,
    Remembered,
    Recent,
    Context
};

public enum ResourceType : int
{
    Any = 0,
    Disk = 1,
    Print = 2,
    Reserved = 8,
}

public enum ResourceDisplaytype : int
{
    Generic = 0x0,
    Domain = 0x01,
    Server = 0x02,
    Share = 0x03,
    File = 0x04,
    Group = 0x05,
    Network = 0x06,
    Root = 0x07,
    Shareadmin = 0x08,
    Directory = 0x09,
```

```
    Tree = 0x0a,
    Ndscontainer = 0x0b
}
```

Lea Acceso a la carpeta compartida de la red con nombre de usuario y contraseña. en línea: https://riptutorial.com/es/csharp/topic/9627/acceso-a-la-carpeta-compartida-de-la-red-con-nombre-de-usuario-y-contrasena-

# Capítulo 3: Acceso a las bases de datos

## Examples

### Conexiones ADO.NET

Las conexiones ADO.NET son una de las formas más simples de conectarse a una base de datos desde una aplicación C #. Se basan en el uso de un proveedor y una cadena de conexión que apunta a su base de datos para realizar consultas.

## Clases comunes de proveedores de datos

Muchas de las siguientes son clases que se usan comúnmente para consultar bases de datos y sus espacios de nombres relacionados:

- `SqlConnection` , `SqlCommand` , `SqlDataReader` de `System.Data.SqlClient`
- `OleDbConnection` , `OleDbCommand` , `OleDbDataReader` de `System.Data.OleDb`
- `MySqlConnection` , `MySqlCommand` , `MySqlDbDataReader` de `MySql.Data`

Todos estos se usan comúnmente para acceder a los datos a través de C # y se encontrarán comúnmente en las aplicaciones de creación de datos centradas. Se puede esperar que muchas otras clases que no se mencionan y que implementan las mismas `FooConnection` , `FooCommand` , `FooDataReader` se comporten de la misma manera.

## Patrón de acceso común para conexiones ADO.NET

Un patrón común que se puede usar al acceder a sus datos a través de una conexión ADO.NET puede tener el siguiente aspecto:

```
// This scopes the connection (your specific class may vary)
using(var connection = new SqlConnection("{your-connection-string}")
{
    // Build your query
    var query = "SELECT * FROM YourTable WHERE Property = @property");
    // Scope your command to execute
    using(var command = new SqlCommand(query, connection))
    {
         // Open your connection
         connection.Open();

         // Add your parameters here if necessary

         // Execute your query as a reader (again scoped with a using statement)
         using(var reader = command.ExecuteReader())
         {
               // Iterate through your results here
         }
    }
}
```

O si solo estuvieras realizando una actualización simple y no necesitaras un lector, se aplicaría el mismo concepto básico:

```
using(var connection = new SqlConnection("{your-connection-string}"))
{
     var query = "UPDATE YourTable SET Property = Value WHERE Foo = @foo";
     using(var command = new SqlCommand(query,connection))
     {
          connection.Open();

          // Add parameters here

          // Perform your update
          command.ExecuteNonQuery();
     }
}
```

Incluso puede programar contra un conjunto de interfaces comunes y no tener que preocuparse por las clases específicas del proveedor. Las interfaces principales proporcionadas por ADO.NET son:

- IDbConnection - para gestionar conexiones de base de datos
- IDbCommand - para ejecutar comandos SQL
- IDbTransaction - para gestionar transacciones
- IDataReader - para leer los datos devueltos por un comando
- IDataAdapter - para canalizar datos hacia y desde conjuntos de datos

```
var connectionString = "{your-connection-string}";
var providerName = "{System.Data.SqlClient}"; //for Oracle use
"Oracle.ManagedDataAccess.Client"
//most likely you will get the above two from ConnectionStringSettings object

var factory = DbProviderFactories.GetFactory(providerName);

using(var connection = new factory.CreateConnection()) {
    connection.ConnectionString = connectionString;
    connection.Open();

    using(var command = new connection.CreateCommand()) {
        command.CommandText = "{sql-query}";    //this needs to be tailored for each database
system

        using(var reader = command.ExecuteReader()) {
            while(reader.Read()) {
                ...
            }
        }
    }
}
```

## Entity Framework Connections

Entity Framework expone las clases de abstracción que se utilizan para interactuar con las bases de datos subyacentes en forma de clases como `DbContext` . Estos contextos generalmente consisten en `DbSet<T>` que exponen las colecciones disponibles que se pueden consultar:

```
public class ExampleContext: DbContext
{
    public virtual DbSet<Widgets> Widgets { get; set; }
}
```

El propio `DbContext` manejará las conexiones con las bases de datos y generalmente leerá los datos de la Cadena de conexión apropiados de una configuración para determinar cómo establecer las conexiones:

```
public class ExampleContext: DbContext
{
    // The parameter being passed in to the base constructor indicates the name of the
    // connection string
    public ExampleContext() : base("ExampleContextEntities")
    {
    }

    public virtual DbSet<Widgets> Widgets { get; set; }
}
```

# Ejecutando consultas de Entity Framework

En realidad, la ejecución de una consulta de Entity Framework puede ser bastante sencilla y simplemente requiere que cree una instancia del contexto y luego use las propiedades disponibles para extraer o acceder a sus datos.

```
using(var context = new ExampleContext())
{
      // Retrieve all of the Widgets in your database
      var data = context.Widgets.ToList();
}
```

Entity Framework también proporciona un extenso sistema de seguimiento de cambios que se puede usar para manejar la actualización de entradas dentro de su base de datos simplemente llamando al método `SaveChanges()` para enviar cambios a la base de datos:

```
using(var context = new ExampleContext())
{
      // Grab the widget you wish to update
      var widget = context.Widgets.Find(w => w.Id == id);
      // If it exists, update it
      if(widget != null)
      {
            // Update your widget and save your changes
            widget.Updated = DateTime.UtcNow;
            context.SaveChanges();
      }
}
```

## Cuerdas de conexión

Una Cadena de conexión es una cadena que especifica información sobre un origen de datos en

particular y cómo conectarse a ella mediante el almacenamiento de credenciales, ubicaciones y otra información.

```
Server=myServerAddress;Database=myDataBase;User Id=myUsername;Password=myPassword;
```

## Almacenar su cadena de conexión

Normalmente, una cadena de conexión se almacenará dentro de un archivo de configuración (como `app.config` o `web.config` dentro de las aplicaciones ASP.NET). El siguiente es un ejemplo del aspecto que podría tener una conexión local dentro de uno de estos archivos:

```
<connectionStrings>
   <add name="WidgetsContext" providerName="System.Data.SqlClient"
connectionString="Server=.\SQLEXPRESS;Database=Widgets;Integrated Security=True;"/>
</connectionStrings>

<connectionStrings>
   <add name="WidgetsContext" providerName="System.Data.SqlClient"
connectionString="Server=.\SQLEXPRESS;Database=Widgets;Integrated Security=SSPI;"/>
</connectionStrings>
```

Esto permitirá que su aplicación acceda a la cadena de conexión programáticamente a través de `WidgetsContext` . Aunque tanto la `Integrated Security=SSPI` como la `Integrated Security=True` realizan la misma función; Se prefiere la `Integrated Security=SSPI` ya que funciona con el proveedor SQLClient y OleDB, donde la `Integrated Security=true` lanza una excepción cuando se usa con el proveedor OleDb.

## Diferentes conexiones para diferentes proveedores

Cada proveedor de datos (SQL Server, MySQL, Azure, etc.) presenta su propia sintaxis de cadenas de conexión y expone diferentes propiedades disponibles. ConnectionStrings.com es un recurso increíblemente útil si no está seguro de cómo debe lucir el suyo.

Lea Acceso a las bases de datos en línea: https://riptutorial.com/es/csharp/topic/4811/acceso-a-las-bases-de-datos

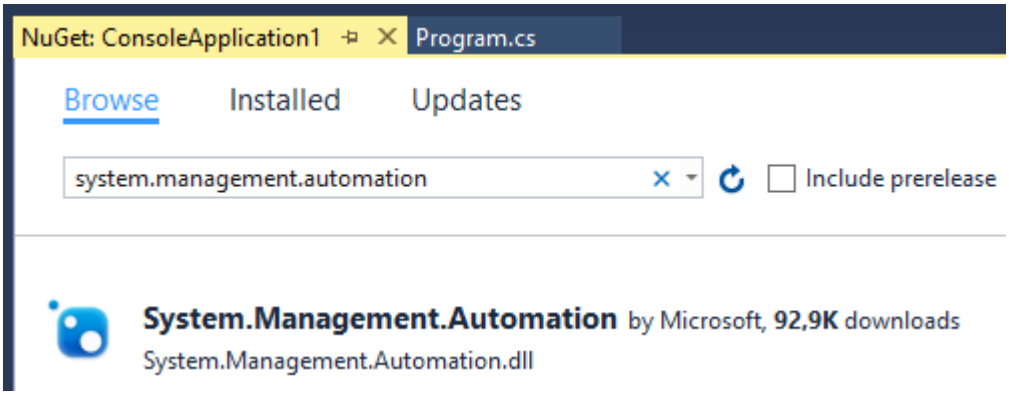
# Capítulo 4: Administración del sistema.Automation

## Observaciones

> El espacio de nombres *System.Management.Automation* es el espacio de nombres raíz de Windows PowerShell.

System.Management.Automation es una biblioteca de extensión de Microsoft y se puede agregar a los proyectos de Visual Studio mediante el administrador de paquetes NuGet o la consola del administrador de paquetes.

---




---

```
PM> Install-Package System.Management.Automation
```

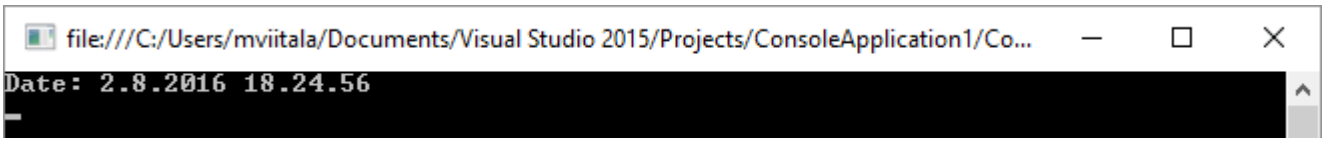
## Examples

### Invocar tubería simple sincrónica.

Obtenga la fecha y hora actual.

```
public class Program
{
    static void Main()
    {
        // create empty pipeline
        PowerShell ps = PowerShell.Create();

        // add command
        ps.AddCommand("Get-Date");

        // run command(s)
        Console.WriteLine("Date: {0}", ps.Invoke().First());

        Console.ReadLine();
    }
```

```
}
```




Lea Administración del sistema.Automation en línea:
https://riptutorial.com/es/csharp/topic/4988/administracion-del-sistema-automation

# Capítulo 5: Alias de tipos incorporados

## Examples

### Tabla de tipos incorporados

La siguiente tabla muestra las palabras clave para los tipos `C#` incorporados, que son alias de tipos predefinidos en los espacios de nombres del Sistema.

| Tipo C # | Tipo de Framework .NET |
|---|---|
| bool | System.Boolean |
| byte | System.Byte |
| sbyte | System.SByte |
| carbonizarse | System.Char |
| decimal | System.Decimal |
| doble | Sistema.Double |
| flotador | Sistema.Single |
| En t | System.Int32 |
| uint | System.UInt32 |
| largo | System.Int64 |
| ulong | System.UInt64 |
| objeto | Sistema.Objeto |
| corto | System.Int16 |
| ushort | System.UInt16 |
| cuerda | System.String |

Las palabras clave de tipo `C#` y sus alias son intercambiables. Por ejemplo, puede declarar una variable entera utilizando una de las siguientes declaraciones:

```
int number = 123;
System.Int32 number = 123;
```

Lea Alias de tipos incorporados en línea: https://riptutorial.com/es/csharp/topic/1862/alias---de-tipos-incorporados

# Capítulo 6: Almacenamiento en caché

## Examples

### Memoria caché

```
//Get instance of cache
using System.Runtime.Caching;

var cache = MemoryCache.Default;

//Check if cache contains an item with
cache.Contains("CacheKey");

//get item from cache
var item = cache.Get("CacheKey");

//get item from cache or add item if not existing
object list = MemoryCache.Default.AddOrGetExisting("CacheKey", "object to be stored",
DateTime.Now.AddHours(12));

//note if item not existing the item is added by this method
//but the method returns null
```

Lea Almacenamiento en caché en línea:
https://riptutorial.com/es/csharp/topic/4383/almacenamiento-en-cache

# Capítulo 7: Anotación de datos

## Examples

### DisplayNameAttribute (atributo de visualización)

`DisplayName` establece el nombre de visualización de una propiedad, evento o método de anulación público que tiene cero (0) argumentos.

```
public class Employee
{
    [DisplayName(@"Employee first name")]
    public string FirstName { get; set; }
}
```

#### Ejemplo de uso simple en la aplicación XAML

```
<Window x:Class="WpfApplication.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:wpfApplication="clr-namespace:WpfApplication"
        Height="100" Width="360" Title="Display name example">

    <Window.Resources>
        <wpfApplication:DisplayNameConverter x:Key="DisplayNameConverter"/>
    </Window.Resources>

    <StackPanel Margin="5">
        <!-- Label (DisplayName attribute) -->
        <Label Content="{Binding Employee, Converter={StaticResource DisplayNameConverter},
ConverterParameter=FirstName}" />
        <!-- TextBox (FirstName property value) -->
        <TextBox Text="{Binding Employee.FirstName, Mode=TwoWay,
UpdateSourceTrigger=PropertyChanged}" />
    </StackPanel>

</Window>
```

---

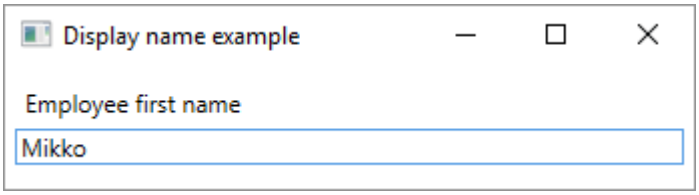
```
namespace WpfApplication
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        private Employee _employee = new Employee();

        public MainWindow()
        {
            InitializeComponent();
            DataContext = this;
        }
```

```
        public Employee Employee
        {
            get { return _employee; }
            set { _employee = value; }
        }
    }
}
```

---

```
namespace WpfApplication
{
    public class DisplayNameConverter : IValueConverter
    {
        public object Convert(object value, Type targetType, object parameter, CultureInfo
culture)
        {
            // Get display name for given instance type and property name
            var attribute = value.GetType()
                .GetProperty(parameter.ToString())
                .GetCustomAttributes(false)
                .OfType<DisplayNameAttribute>()
                .FirstOrDefault();

            return attribute != null ? attribute.DisplayName : string.Empty;
        }

        public object ConvertBack(object value, Type targetType, object parameter, CultureInfo
culture)
        {
            throw new NotImplementedException();
        }
    }
}
```

---




## EditableAttribute (atributo de modelado de datos)

`EditableAttribute` establece si los usuarios deberían poder cambiar el valor de la propiedad de clase.

```
public class Employee
{
    [Editable(false)]
    public string FirstName { get; set; }
}
```

### Ejemplo de uso simple en la aplicación XAML

```
<Window x:Class="WpfApplication.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
```

```
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:wpfApplication="clr-namespace:WpfApplication"
        Height="70" Width="360" Title="Display name example">

    <Window.Resources>
        <wpfApplication:EditableConverter x:Key="EditableConverter"/>
    </Window.Resources>

    <StackPanel Margin="5">
        <!-- TextBox Text (FirstName property value) -->
        <!-- TextBox IsEnabled (Editable attribute) -->
        <TextBox Text="{Binding Employee.FirstName, Mode=TwoWay,
UpdateSourceTrigger=PropertyChanged}"
                 IsEnabled="{Binding Employee, Converter={StaticResource EditableConverter},
ConverterParameter=FirstName}"/>
    </StackPanel>

</Window>
```

---

```
namespace WpfApplication
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        private Employee _employee = new Employee() { FirstName = "This is not editable"};

        public MainWindow()
        {
            InitializeComponent();
            DataContext = this;
        }

        public Employee Employee
        {
            get { return _employee; }
            set { _employee = value; }
        }
    }
}
```

---

```
namespace WpfApplication
{
    public class EditableConverter : IValueConverter
    {
        public object Convert(object value, Type targetType, object parameter, CultureInfo
culture)
        {
            // return editable attribute's value for given instance property,
            // defaults to true if not found
            var attribute = value.GetType()
                .GetProperty(parameter.ToString())
                .GetCustomAttributes(false)
                .OfType<EditableAttribute>()
                .FirstOrDefault();

            return attribute != null ? attribute.AllowEdit : true;
```

```
        }

        public object ConvertBack(object value, Type targetType, object parameter, CultureInfo
culture)
        {
            throw new NotImplementedException();
        }
    }
}
```

---


### Atributos de Validación

Los atributos de validación se utilizan para imponer varias reglas de validación de forma declarativa en clases o miembros de clase. Todos los atributos de validación se derivan de la clase base ValidationAttribute .

---

## Ejemplo: RequiredAttribute

Cuando se valida a través del método `ValidationAttribute.Validate` , este atributo devolverá un error si la propiedad `Name` es nula o solo contiene espacios en blanco.

```
public class ContactModel
{
    [Required(ErrorMessage = "Please provide a name.")]
    public string Name { get; set; }
}
```

---

## Ejemplo: StringLengthAttribute

El `StringLengthAttribute` valida si una cadena es menor que la longitud máxima de una cadena. Opcionalmente puede especificar una longitud mínima. Ambos valores son inclusivos.

```
public class ContactModel
{
    [StringLength(20, MinimumLength = 5, ErrorMessage = "A name must be between five and
twenty characters.")]
    public string Name { get; set; }
}
```

---

## Ejemplo: RangeAttribute

El `RangeAttribute` da el valor máximo y mínimo para un campo numérico.

```
public class Model
{
    [Range(0.01, 100.00,ErrorMessage = "Price must be between 0.01 and 100.00")]
    public decimal Price { get; set; }
}
```

---

# Ejemplo: CustomValidationAttribute

La clase `CustomValidationAttribute` permite invocar un método `static` personalizado para su validación. El método personalizado debe ser `static ValidationResult [MethodName] (object input)` .

```
public class Model
{
    [CustomValidation(typeof(MyCustomValidation), "IsNotAnApple")]
    public string FavoriteFruit { get; set; }
}
```

Declaración de método:

```
public static class MyCustomValidation
{
    public static ValidationResult IsNotAnApple(object input)
    {
        var result = ValidationResult.Success;

        if (input?.ToString()?.ToUpperInvariant() == "APPLE")
        {
            result = new ValidationResult("Apples are not allowed.");
        }

        return result;
    }
}
```

## Creación de un atributo de validación personalizado

Los atributos de validación personalizados se pueden crear derivando de la clase base `ValidationAttribute` , y luego anulando `virtual` métodos `virtual` según sea necesario.

```
[AttributeUsage(AttributeTargets.Property, AllowMultiple = false, Inherited = false)]
public class NotABananaAttribute : ValidationAttribute
{
    public override bool IsValid(object value)
    {
        var inputValue = value as string;
        var isValid = true;

        if (!string.IsNullOrEmpty(inputValue))
        {
            isValid = inputValue.ToUpperInvariant() != "BANANA";
        }
```

```
        return isValid;
    }
}
```

Este atributo puede ser usado así:

```
public class Model
{
    [NotABanana(ErrorMessage = "Bananas are not allowed.")]
    public string FavoriteFruit { get; set; }
}
```

### Fundamentos de la anotación de datos

Las anotaciones de datos son una forma de agregar más información contextual a las clases o miembros de una clase. Hay tres categorías principales de anotaciones:

- Atributos de validación: agregar criterios de validación a los datos
- Atributos de visualización: especifique cómo deben mostrarse los datos al usuario
- Atributos de modelado: agregue información sobre el uso y la relación con otras clases

## Uso

Aquí hay un ejemplo donde se usan dos `DisplayAttribute` `ValidationAttribute` y un `DisplayAttribute` :

```
class Kid
{
    [Range(0, 18)] // The age cannot be over 18 and cannot be negative
    public int Age { get; set; }
    [StringLength(MaximumLength = 50, MinimumLength = 3)] // The name cannot be under 3 chars
or more than 50 chars
    public string Name { get; set; }
    [DataType(DataType.Date)] // The birthday will be displayed as a date only (without the
time)
    public DateTime Birthday { get; set; }
}
```

Las anotaciones de datos se utilizan principalmente en marcos como ASP.NET. Por ejemplo, en `ASP.NET MVC` , cuando un modelo es recibido por un método de controlador, se puede usar `ModelState.IsValid()` para determinar si el modelo recibido respeta todos sus `ValidationAttribute` . `DisplayAttribute` también se usa en `ASP.NET MVC` para determinar cómo mostrar los valores en una página web.

### Ejecutar manualmente los atributos de validación

La mayoría de las veces, los atributos de validación se usan dentro de marcos (como ASP.NET). Esos frameworks se encargan de ejecutar los atributos de validación. ¿Pero qué pasa si quieres ejecutar los atributos de validación manualmente? Solo use la clase `Validator` (no se necesita reflexión).

## Contexto de Validación

Cualquier validación necesita un contexto para proporcionar información sobre lo que se está validando. Esto puede incluir información diversa como el objeto a validar, algunas propiedades, el nombre a mostrar en el mensaje de error, etc.

```
ValidationContext vc = new ValidationContext(objectToValidate); // The simplest form of
validation context. It contains only a reference to the object being validated.
```

Una vez que se crea el contexto, hay varias formas de hacer la validación.

## Validar un objeto y todas sus propiedades

```
ICollection<ValidationResult> results = new List<ValidationResult>(); // Will contain the
results of the validation
bool isValid = Validator.TryValidateObject(objectToValidate, vc, results, true); // Validates
the object and its properties using the previously created context.
// The variable isValid will be true if everything is valid
// The results variable contains the results of the validation
```

## Validar una propiedad de un objeto

```
ICollection<ValidationResult> results = new List<ValidationResult>(); // Will contain the
results of the validation
bool isValid = Validator.TryValidatePropery(objectToValidate.PropertyToValidate, vc, results,
true); // Validates the property using the previously created context.
// The variable isValid will be true if everything is valid
// The results variable contains the results of the validation
```

## Y más

Para obtener más información sobre la validación manual, consulte:

- ValidationContext Class Documentation
- Validador de la clase de documentación

Lea Anotación de datos en línea: https://riptutorial.com/es/csharp/topic/4942/anotacion-de-datos

# Capítulo 8: Arboles de expresion

## Introducción

Los árboles de expresión son expresiones organizadas en una estructura de datos en forma de árbol. Cada nodo en el árbol es una representación de una expresión, una expresión que es código. Una representación en memoria de una expresión Lambda sería un árbol de expresión, que contiene los elementos reales (es decir, el código) de la consulta, pero no su resultado. Los árboles de expresión hacen que la estructura de una expresión lambda sea transparente y explícita.

## Sintaxis

- Expresión <TDelegate> name = lambdaExpression;

## Parámetros

| Parámetro | Detalles |
|---|---|
| TDelegate | El tipo de delegado que se utilizará para la expresión. |
| expresión lambda | La expresión lambda (ej. `num => num < 5` ) |

## Observaciones

# Introducción a los árboles de expresión

## De donde venimos

Los árboles de expresiones tratan de consumir "código fuente" en tiempo de ejecución. Considere un método que calcula el impuesto a las ventas adeudado en una orden de venta `decimal CalculateTotalTaxDue(SalesOrder order)` . El uso de ese método en un programa .NET es fácil: solo se le llama `decimal taxDue = CalculateTotalTaxDue(order);` . ¿Qué sucede si desea aplicarlo a todos los resultados de una consulta remota (SQL, XML, un servidor remoto, etc.)? ¡Esas fuentes de consulta remotas no pueden llamar al método! Tradicionalmente, tendrías que invertir el flujo en todos estos casos. Realice la consulta completa, guárdela en la memoria, luego repita los resultados y calcule los impuestos para cada resultado.

## Cómo evitar problemas de memoria y latencia en la inversión de flujo.

Los árboles de expresión son estructuras de datos en un formato de árbol, donde cada nodo contiene una expresión. Se utilizan para traducir las instrucciones compiladas (como los métodos utilizados para filtrar datos) en expresiones que podrían usarse fuera del entorno del programa, como dentro de una consulta de base de datos.

El problema aquí es que una consulta remota *no puede acceder a nuestro método* . Podríamos evitar este problema si, en cambio, enviamos las *instrucciones* del método a la consulta remota. En nuestro ejemplo de `CalculateTotalTaxDue` , eso significa que enviamos esta información:

1. Crear una variable para almacenar el impuesto total.
2. Recorrer todas las líneas del pedido.
3. Para cada línea, compruebe si el producto está sujeto a impuestos.
4. Si es así, multiplique el total de la línea por la tasa de impuesto aplicable y agregue esa cantidad al total
5. De lo contrario no hacer nada

Con esas instrucciones, la consulta remota puede realizar el trabajo a medida que crea los datos.

Hay dos desafíos para implementar esto. ¿Cómo transforma un método .NET compilado en una lista de instrucciones, y cómo formatea las instrucciones de manera que puedan ser consumidas por el sistema remoto?

Sin árboles de expresiones, solo se podría resolver el primer problema con MSIL. (MSIL es el código tipo ensamblador creado por el compilador .NET). Analizar MSIL es *posible* , pero no es fácil. Incluso cuando lo analiza correctamente, puede ser difícil determinar cuál fue la intención del programador original con una rutina en particular.

## Árboles de expresión salvan el día

Los árboles de expresión abordan estos problemas exactos. Representan instrucciones de programa, una estructura de datos de árbol donde cada nodo representa *una instrucción* y tiene referencias a toda la información que necesita para ejecutar esa instrucción. Por ejemplo, una `MethodCallExpression` tiene una referencia a 1) `MethodInfo` a la que llamará, 2) una lista de `Expression` que pasará a ese método, 3) para los métodos de instancia, la `Expression` a la que llamará el método. Puede "recorrer el árbol" y aplicar las instrucciones en su consulta remota.

## Creando arboles de expresion

La forma más fácil de crear un árbol de expresiones es con una expresión lambda. Estas expresiones se ven casi iguales a los métodos normales de C #. Es importante darse cuenta de que esto es *magia compilador* . Cuando creas una expresión lambda por primera vez, el compilador verifica a qué lo asignas. Si es un tipo de `Delegate` (incluyendo `Action` o `Func` ), el compilador convierte la expresión lambda en un delegado. Si se trata de una `LambdaExpression` (o una `Expression<Action<T>>` o una `Expression<Func<T>>` que son de tipo `LambdaExpression` ), el compilador la transforma en una `LambdaExpression` . Aquí es donde entra en `LambdaExpression` la magia. Entre bambalinas, el compilador *utiliza la API del árbol de expresiones* para transformar la expresión lambda en una expresión `LambdaExpression` .

Las expresiones Lambda no pueden crear todo tipo de árbol de expresión. En esos casos, puede utilizar la API de expresiones manualmente para crear el árbol que necesita. En el ejemplo Entendiendo las expresiones API , creamos la expresión `CalculateTotalSalesTax` utilizando la API.

NOTA: Los nombres se ponen un poco confusos aquí. Una *expresión lambda* (dos palabras, minúsculas) se refiere al bloque de código con un indicador `=>` . Representa un método anónimo en C # y se convierte en un `Delegate` o `Expression` . Una *`LambdaExpression`* (una palabra, PascalCase) se refiere al tipo de nodo dentro de la API de Expresión que representa un método que puede ejecutar.

# Árboles de expresión y LINQ

Uno de los usos más comunes de los árboles de expresión es con LINQ y consultas de base de datos. LINQ empareja un árbol de expresiones con un proveedor de consultas para aplicar sus instrucciones a la consulta remota de destino. Por ejemplo, el proveedor de consultas de LINQ to Entity Framework transforma un árbol de expresiones en SQL que se ejecuta directamente en la base de datos.

Poniendo todas las piezas juntas, puedes ver el verdadero poder detrás de LINQ.

1. Escriba una consulta usando una expresión lambda: `products.Where(x => x.Cost > 5)`
2. El compilador transforma esa expresión en un árbol de expresiones con las instrucciones "compruebe si la propiedad de costo del parámetro es mayor que cinco".
3. El proveedor de consultas analiza el árbol de expresiones y genera una consulta SQL válida `SELECT * FROM products WHERE Cost > 5`
4. El ORM proyecta todos los resultados en POCO y obtiene una lista de objetos de vuelta

# Notas

- Los árboles de expresión son inmutables. Si desea cambiar un árbol de expresiones, necesita crear uno nuevo, copie el existente en el nuevo (para recorrer un árbol de expresiones puede usar `ExpressionVisitor` ) y realice los cambios deseados.

## Examples

### Creando árboles de expresiones usando la API

```
using System.Linq.Expressions;

// Manually build the expression tree for
// the lambda expression num => num < 5.
ParameterExpression numParam = Expression.Parameter(typeof(int), "num");
ConstantExpression five = Expression.Constant(5, typeof(int));
BinaryExpression numLessThanFive = Expression.LessThan(numParam, five);
Expression<Func<int, bool>> lambda1 =
    Expression.Lambda<Func<int, bool>>(
        numLessThanFive,
```

```
        new ParameterExpression[] { numParam });
```

## Compilando arboles de expresion

```
// Define an expression tree, taking an integer, returning a bool.
Expression<Func<int, bool>> expr = num => num < 5;

// Call the Compile method on the expression tree to return a delegate that can be called.
Func<int, bool> result = expr.Compile();

// Invoke the delegate and write the result to the console.
Console.WriteLine(result(4)); // Prints true

// Prints True.

// You can also combine the compile step with the call/invoke step as below:
Console.WriteLine(expr.Compile()(4));
```

## Análisis de árboles de expresión

```
using System.Linq.Expressions;

// Create an expression tree.
Expression<Func<int, bool>> exprTree = num => num < 5;

// Decompose the expression tree.
ParameterExpression param = (ParameterExpression)exprTree.Parameters[0];
BinaryExpression operation = (BinaryExpression)exprTree.Body;
ParameterExpression left = (ParameterExpression)operation.Left;
ConstantExpression right = (ConstantExpression)operation.Right;

Console.WriteLine("Decomposed expression: {0} => {1} {2} {3}",
                  param.Name, left.Name, operation.NodeType, right.Value);

// Decomposed expression: num => num LessThan 5
```

## Crear árboles de expresión con una expresión lambda

A continuación se muestra el árbol de expresiones más básico creado por lambda.

```
Expression<Func<int, bool>> lambda = num => num == 42;
```

Para crear árboles de expresión 'a mano', se debe usar `Expression` clase de `Expression` .

La expresión anterior sería equivalente a:

```
ParameterExpression parameter = Expression.Parameter(typeof(int), "num"); // num argument
ConstantExpression constant = Expression.Constant(42, typeof(int)); // 42 constant
BinaryExpression equality = Expression.Equals(parameter, constant); // equality of two
expressions (num == 42)
Expression<Func<int, bool>> lambda = Expression.Lambda<Func<int, bool>>(equality, parameter);
```

## Entendiendo la API de expresiones

Vamos a utilizar la API del árbol de expresiones para crear un árbol `CalculateSalesTax` . En un lenguaje sencillo, aquí hay un resumen de los pasos necesarios para crear el árbol.

1. Compruebe si el producto está sujeto a impuestos.
2. Si es así, multiplique el total de la línea por la tasa de impuesto aplicable y devuelva esa cantidad
3. De lo contrario devuelve 0

```
//For reference, we're using the API to build this lambda expression
    orderLine => orderLine.IsTaxable ? orderLine.Total * orderLine.Order.TaxRate : 0;

//The orderLine parameter we pass in to the method.  We specify it's type (OrderLine) and the
name of the parameter.
    ParameterExpression orderLine = Expression.Parameter(typeof(OrderLine), "orderLine");

//Check if the parameter is taxable;  First we need to access the is taxable property, then
check if it's true
    PropertyInfo isTaxableAccessor = typeof(OrderLine).GetProperty("IsTaxable");
    MemberExpression getIsTaxable = Expression.MakeMemberAccess(orderLine, isTaxableAccessor);
    UnaryExpression isLineTaxable = Expression.IsTrue(getIsTaxable);

//Before creating the if, we need to create the braches
    //If the line is taxable, we'll return the total times the tax rate; get the total and tax
rate, then multiply
    //Get the total
    PropertyInfo totalAccessor = typeof(OrderLine).GetProperty("Total");
    MemberExpression getTotal = Expression.MakeMemberAccess(orderLine, totalAccessor);

    //Get the order
    PropertyInfo orderAccessor = typeof(OrderLine).GetProperty("Order");
    MemberExpression getOrder = Expression.MakeMemberAccess(orderLine, orderAccessor);

    //Get the tax rate - notice that we pass the getOrder expression directly to the member
access
    PropertyInfo taxRateAccessor = typeof(Order).GetProperty("TaxRate");
    MemberExpression getTaxRate = Expression.MakeMemberAccess(getOrder, taxRateAccessor);

    //Multiply the two - notice we pass the two operand expressions directly to multiply
    BinaryExpression multiplyTotalByRate = Expression.Multiply(getTotal, getTaxRate);

//If the line is not taxable, we'll return a constant value - 0.0 (decimal)
    ConstantExpression zero = Expression.Constant(0M);

//Create the actual if check and branches
    ConditionalExpression ifTaxableTernary = Expression.Condition(isLineTaxable,
multiplyTotalByRate, zero);

//Wrap the whole thing up in a "method" - a LambdaExpression
    Expression<Func<OrderLine, decimal>> method = Expression.Lambda<Func<OrderLine,
decimal>>(ifTaxableTernary, orderLine);
```

## Árbol de Expresión Básico

Los árboles de expresión representan el código en una estructura de datos similar a un árbol,

donde cada nodo es una expresión

Expression Trees permite la modificación dinámica del código ejecutable, la ejecución de consultas LINQ en varias bases de datos y la creación de consultas dinámicas. Puedes compilar y ejecutar código representado por árboles de expresión.

También se usan en el tiempo de ejecución del lenguaje dinámico (DLR) para proporcionar interoperabilidad entre los lenguajes dinámicos y .NET Framework y para permitir que los escritores de compiladores emitan árboles de expresión en lugar del lenguaje intermedio de Microsoft (MSIL).

Los árboles de expresión se pueden crear a través de

1. Expresión lambda anónima,
2. Manualmente utilizando el espacio de nombres System.Linq.Expressions.

**Árboles de expresión de Expresiones Lambda**

Cuando se asigna una expresión lambda a la variable de tipo de expresión, el compilador emite código para construir un árbol de expresión que representa la expresión lambda.

Los siguientes ejemplos de código muestran cómo hacer que el compilador de C # cree un árbol de expresión que represente la expresión lambda num => num <5.

```
Expression<Func<int, bool>> lambda = num => num < 5;
```

**Árboles de expresión mediante el uso de la API**

Los árboles de expresión también se crearon utilizando la clase de **expresión** . Esta clase contiene métodos de fábrica estáticos que crean nodos de árbol de expresión de tipos específicos.

A continuación se muestran algunos tipos de nodos de árbol.

1. ParameterExpression
2. MethodCallExpression

El siguiente ejemplo de código muestra cómo crear un árbol de expresión que representa la expresión lambda num => num <5 mediante el uso de la API.

```
ParameterExpression numParam = Expression.Parameter(typeof(int), "num");
ConstantExpression five = Expression.Constant(5, typeof(int));
BinaryExpression numLessThanFive = Expression.LessThan(numParam, five);
Expression<Func<int, bool>> lambda1 = Expression.Lambda<Func<int, bool>>(numLessThanFive,new
ParameterExpression[] { numParam });
```

## Examinar la estructura de una expresión usando el visitante

Defina una nueva clase de visitante anulando algunos de los métodos de ExpressionVisitor :

```
class PrintingVisitor : ExpressionVisitor {
    protected override Expression VisitConstant(ConstantExpression node) {
        Console.WriteLine("Constant: {0}", node);
        return base.VisitConstant(node);
    }
    protected override Expression VisitParameter(ParameterExpression node) {
        Console.WriteLine("Parameter: {0}", node);
        return base.VisitParameter(node);
    }
    protected override Expression VisitBinary(BinaryExpression node) {
        Console.WriteLine("Binary with operator {0}", node.NodeType);
        return base.VisitBinary(node);
    }
}
```

Llame a `Visit` para usar este visitante en una expresión existente:

```
Expression<Func<int,bool>> isBig = a => a > 1000000;
var visitor = new PrintingVisitor();
visitor.Visit(isBig);
```

Lea Arboles de expresion en línea: https://riptutorial.com/es/csharp/topic/75/arboles-de-expresion

# Capítulo 9: Archivo y Stream I / O

## Introducción

Gestiona archivos.

## Sintaxis

- `new System.IO.StreamWriter(string path)`
- `new System.IO.StreamWriter(string path, bool append)`
- `System.IO.StreamWriter.WriteLine(string text)`
- `System.IO.StreamWriter.WriteAsync(string text)`
- `System.IO.Stream.Close()`
- `System.IO.File.ReadAllText(string path)`
- `System.IO.File.ReadAllLines(string path)`
- `System.IO.File.ReadLines(string path)`
- `System.IO.File.WriteAllText(string path, string text)`
- `System.IO.File.WriteAllLines(string path, IEnumerable<string> contents)`
- `System.IO.File.Copy(string source, string dest)`
- `System.IO.File.Create(string path)`
- `System.IO.File.Delete(string path)`
- `System.IO.File.Move(string source, string dest)`
- `System.IO.Directory.GetFiles(string path)`

## Parámetros

| Parámetro | Detalles |
|---|---|
| camino | La ubicación del archivo. |
| adjuntar | Si el archivo existe, verdadero agregará datos al final del archivo (adjuntar), falso sobrescribirá el archivo. |
| texto | Texto a escribir o almacenar. |
| contenido | Una colección de cuerdas para escribir. |
| fuente | La ubicación del archivo que desea utilizar. |
| destino | La ubicación a la que desea que vaya un archivo. |

## Observaciones

- Asegúrate siempre de cerrar objetos `Stream` . Esto se puede hacer con un bloque `using` como se muestra arriba o llamando manualmente a `myStream.Close()` .
- Asegúrese de que el usuario actual tenga los permisos necesarios en la ruta que intenta

crear el archivo.
- Las cadenas verbales se deben usar cuando se declara una cadena de ruta que incluye barras diagonales inversas, como por ejemplo: `@"C:\MyFolder\MyFile.txt"`

# Examples

## Leyendo de un archivo usando la clase System.IO.File

Puede usar la función **System.IO.File.ReadAllText** para leer todo el contenido de un archivo en una cadena.

```
string text = System.IO.File.ReadAllText(@"C:\MyFolder\MyTextFile.txt");
```

También puede leer un archivo como una matriz de líneas utilizando la función **System.IO.File.ReadAllLines** :

```
string[] lines = System.IO.File.ReadAllLines(@"C:\MyFolder\MyTextFile.txt");
```

## Escribir líneas en un archivo usando la clase System.IO.StreamWriter

La clase **System.IO.StreamWriter** :

> Implementa un TextWriter para escribir caracteres en un flujo en una codificación particular.

Usando el método `WriteLine` , puede escribir contenido línea por línea en un archivo.

Observe el uso de la palabra clave `using` que garantiza que el objeto StreamWriter se elimine tan pronto como salga de su alcance y, por lo tanto, el archivo se cierre.

```
string[] lines = { "My first string", "My second string", "and even a third string" };
using (System.IO.StreamWriter sw = new System.IO.StreamWriter(@"C:\MyFolder\OutputText.txt"))
{
    foreach (string line in lines)
    {
        sw.WriteLine(line);
    }
}
```

Tenga en cuenta que StreamWriter puede recibir un segundo parámetro `bool` en su constructor, lo que permite `Append` un archivo en lugar de sobrescribirlo:

```
bool appendExistingFile = true;
using (System.IO.StreamWriter sw = new System.IO.StreamWriter(@"C:\MyFolder\OutputText.txt",
appendExistingFile ))
{
    sw.WriteLine("This line will be appended to the existing file");
}
```

## Escribir en un archivo usando la clase System.IO.File

Puede usar la función **System.IO.File.WriteAllText** para escribir una cadena en un archivo.

```
string text = "String that will be stored in the file";
System.IO.File.WriteAllText(@"C:\MyFolder\OutputFile.txt", text);
```

También puede usar la función **System.IO.File.WriteAllLines** que recibe una `IEnumerable<String>` como segundo parámetro (a diferencia de una única cadena en el ejemplo anterior). Esto le permite escribir contenido desde una matriz de líneas.

```
string[] lines = { "My first string", "My second string", "and even a third string" };
System.IO.File.WriteAllLines(@"C:\MyFolder\OutputFile.txt", lines);
```

## Lealmente leyendo un archivo línea por línea a través de un IEnumerable

Cuando trabaje con archivos grandes, puede usar el método `System.IO.File.ReadLines` para leer todas las líneas de un archivo en una `IEnumerable<string>` . Esto es similar a `System.IO.File.ReadAllLines` , excepto que no carga todo el archivo en la memoria de una vez, lo que lo hace más eficiente cuando se trabaja con archivos grandes.

```
IEnumerable<string> AllLines = File.ReadLines("file_name.txt", Encoding.Default);
```

*El segundo parámetro de File.ReadLines es opcional. Puede usarlo cuando sea necesario para especificar la codificación.*

Es importante tener en cuenta que llamar a `ToArray` , `ToList` u otra función similar obligará a que todas las líneas se carguen a la vez, lo que significa que el beneficio de usar `ReadLines` se anula. Es mejor enumerar sobre `IEnumerable` usando un bucle `foreach` o LINQ si se usa este método.

## Crea un archivo

### Archivo clase estática

Usando el método `Create` de la clase estática `File` , podemos crear archivos. Method crea el archivo en la ruta dada, al mismo tiempo que abre el archivo y nos da el `FileStream` del archivo. Asegúrese de cerrar el archivo una vez que haya terminado con él.

ex1:

```
var fileStream1 = File.Create("samplePath");
/// you can write to the fileStream1
fileStream1.Close();
```

ex2:

```
using(var fileStream1 = File.Create("samplePath"))
{
```

```
    /// you can write to the fileStream1
}
```

ex3:

```
File.Create("samplePath").Close();
```

### Clase FileStream

Hay muchas sobrecargas de este constructor de clases que en realidad están bien documentadas aquí . El siguiente ejemplo es para el que cubre las funcionalidades más utilizadas de esta clase.

```
var fileStream2 = new FileStream("samplePath", FileMode.OpenOrCreate, FileAccess.ReadWrite,
FileShare.None);
```

Puede consultar las enumeraciones para FileMode , FileAccess y FileShare desde esos enlaces. Lo que básicamente significan son los siguientes:

*Modo de archivo:* Respuestas "¿Se debe crear el archivo? ¿Abrir? ¿Crear si no existe? ¿Abrir?" un poco preguntas

*FileAccess:* Respuestas "¿Debo poder leer el archivo, escribir en el archivo o en ambos?" un poco preguntas

*FileShare:* Respuestas "¿Deberían otros usuarios poder leer, escribir, etc. en el archivo mientras lo uso simultáneamente?" un poco preguntas

## Copiar archivo

### Archivo clase estática

`File` clase estática puede ser fácilmente utilizado para este propósito.

```
File.Copy(@"sourcePath\abc.txt", @"destinationPath\abc.txt");
File.Copy(@"sourcePath\abc.txt", @"destinationPath\xyz.txt");
```

**Nota:** mediante este método, el archivo se copia, lo que significa que se leerá desde el origen y luego se escribirá en la ruta de destino. Este es un proceso que consume recursos, llevaría un tiempo relativo al tamaño del archivo y puede hacer que su programa se congele si no utiliza subprocesos.

## Mover archivo

### Archivo clase estática

Archivo de clase estática puede ser fácilmente utilizado para este propósito.

```
File.Move(@"sourcePath\abc.txt", @"destinationPath\xyz.txt");
```

**Nota1:** solo cambia el índice del archivo (si el archivo se mueve en el mismo volumen). Esta operación no toma tiempo relativo al tamaño del archivo.

**Nota2:** no se puede anular un archivo existente en la ruta de destino.

### Borrar archivo

```
string path = @"c:\path\to\file.txt";
File.Delete(path);
```

Si bien `Delete` no lanza una excepción si el archivo no existe, generará una excepción, por ejemplo, si la ruta especificada no es válida o la persona que llama no tiene los permisos necesarios. Siempre debe ajustar las llamadas para `Delete` dentro del bloque try-catch y manejar todas las excepciones esperadas. En caso de posibles condiciones de carrera, ajuste la lógica dentro de la instrucción de bloqueo .

### Archivos y directorios

#### Obtener todos los archivos en el Directorio

```
var FileSearchRes = Directory.GetFiles(@Path, "*.*", SearchOption.AllDirectories);
```

Devuelve una matriz de `FileInfo` , que representa todos los archivos en el directorio especificado.

#### Obtener archivos con extensión específica

```
var FileSearchRes = Directory.GetFiles(@Path, "*.pdf", SearchOption.AllDirectories);
```

Devuelve una matriz de `FileInfo` , que representa todos los archivos en el directorio especificado con la extensión especificada.

### Async escribe texto en un archivo usando StreamWriter

```
// filename is a string with the full path
// true is to append
using (System.IO.StreamWriter file = new System.IO.StreamWriter(filename, true))
{
   // Can write either a string or char array
   await file.WriteAsync(text);
}
```

Lea Archivo y Stream I / O en línea: https://riptutorial.com/es/csharp/topic/4266/archivo-y-stream-i--o

# Capítulo 10: Argumentos con nombre

## Examples

### Argumentos con nombre pueden hacer que su código sea más claro

Considera esta clase simple:

```
class SmsUtil
{
    public bool SendMessage(string from, string to, string message, int retryCount, object
attachment)
    {
         // Some code
    }
}
```

Antes de C # 3.0 era:

```
var result = SmsUtil.SendMessage("Mehran", "Maryam", "Hello there!", 12, null);
```

Puede hacer que esta llamada de método sea aún más clara con los **argumentos** con **nombre** :

```
var result = SmsUtil.SendMessage(
    from: "Mehran",
    to:  "Maryam",
    message "Hello there!",
    retryCount: 12,
    attachment: null);
```

### Argumentos con nombre y parámetros opcionales

Puede combinar argumentos con nombre con parámetros opcionales.

Veamos este método:

```
public sealed class SmsUtil
{
    public static bool SendMessage(string from, string to, string message, int retryCount = 5,
object attachment = null)
    {
         // Some code
    }
}
```

Cuando quiera llamar a este método *sin* establecer el argumento `retryCount` :

```
var result = SmsUtil.SendMessage(
                        from       : "Cihan",
                        to         : "Yakar",
```

```
                        message    : "Hello there!",
                        attachment : new object());
```

## El orden del argumento no es necesario

Puede colocar los argumentos con nombre en el orden que desee.

Método de muestra:

```
public static string Sample(string left, string right)
{
     return string.Join("-",left,right);
}
```

Muestra de llamada:

```
Console.WriteLine (Sample(left:"A",right:"B"));
Console.WriteLine (Sample(right:"A",left:"B"));
```

Resultados:

```
A-B
B-A
```

## Argumentos con nombre evita errores en parámetros opcionales

Utilice siempre argumentos con nombre para parámetros opcionales, para evitar posibles errores cuando se modifica el método.

```
class Employee
{
    public string Name { get; private set; }

    public string Title { get; set; }

    public Employee(string name = "<No Name>", string title = "<No Title>")
    {
        this.Name = name;
        this.Title = title;
    }
}

var jack = new Employee("Jack", "Associate");   //bad practice in this line
```

El código anterior compila y funciona bien, hasta que el constructor se cambia algún día como:

```
//Evil Code: add optional parameters between existing optional parameters
public Employee(string name = "<No Name>", string department = "intern", string title = "<No
Title>")
{
    this.Name = name;
    this.Department = department;
```

```
    this.Title = title;
}

//the below code still compiles, but now "Associate" is an argument of "department"
var jack = new Employee("Jack", "Associate");
```

Las mejores prácticas para evitar errores cuando "alguien más en el equipo" cometió errores:

```
var jack = new Employee(name: "Jack", title: "Associate");
```

Lea Argumentos con nombre en línea: https://riptutorial.com/es/csharp/topic/2076/argumentos-con-nombre

# Capítulo 11: Argumentos nombrados y opcionales

## Observaciones

### Argumentos con nombre

*Ref .:* Los argumentos con nombre de *MSDN le* permiten especificar un argumento para un parámetro en particular asociando el argumento con el nombre del parámetro en lugar de con la posición del parámetro en la lista de parámetros.

Según lo dicho por MSDN, un argumento con nombre,

- Le permite pasar el argumento a la función asociando el nombre del parámetro.
- No hay necesidad de recordar la posición de los parámetros que no conocemos siempre.
- No es necesario mirar el orden de los parámetros en la lista de parámetros de la función llamada.
- Podemos especificar el parámetro para cada argumento por su nombre.

### Argumentos opcionales

*Ref: MSDN* La definición de un método, constructor, indexador o delegado puede especificar que sus parámetros son necesarios o que son opcionales. Cualquier llamada debe proporcionar argumentos para todos los parámetros requeridos, pero puede omitir argumentos para parámetros opcionales.

Como dijo MSDN, un argumento opcional,

- Podemos omitir el argumento en la llamada si ese argumento es un argumento opcional
- Cada argumento opcional tiene su propio valor predeterminado
- Tomará valor predeterminado si no suministramos el valor
- Un valor predeterminado de un argumento opcional debe ser un
    - Expresión constante.
    - Debe ser un tipo de valor como enum o struct.
    - Debe ser una expresión del formulario por defecto (valueType)
- Debe establecerse al final de la lista de parámetros

## Examples

### Argumentos con nombre

Considere la siguiente es nuestra llamada de función.

```
FindArea(120, 56);
```

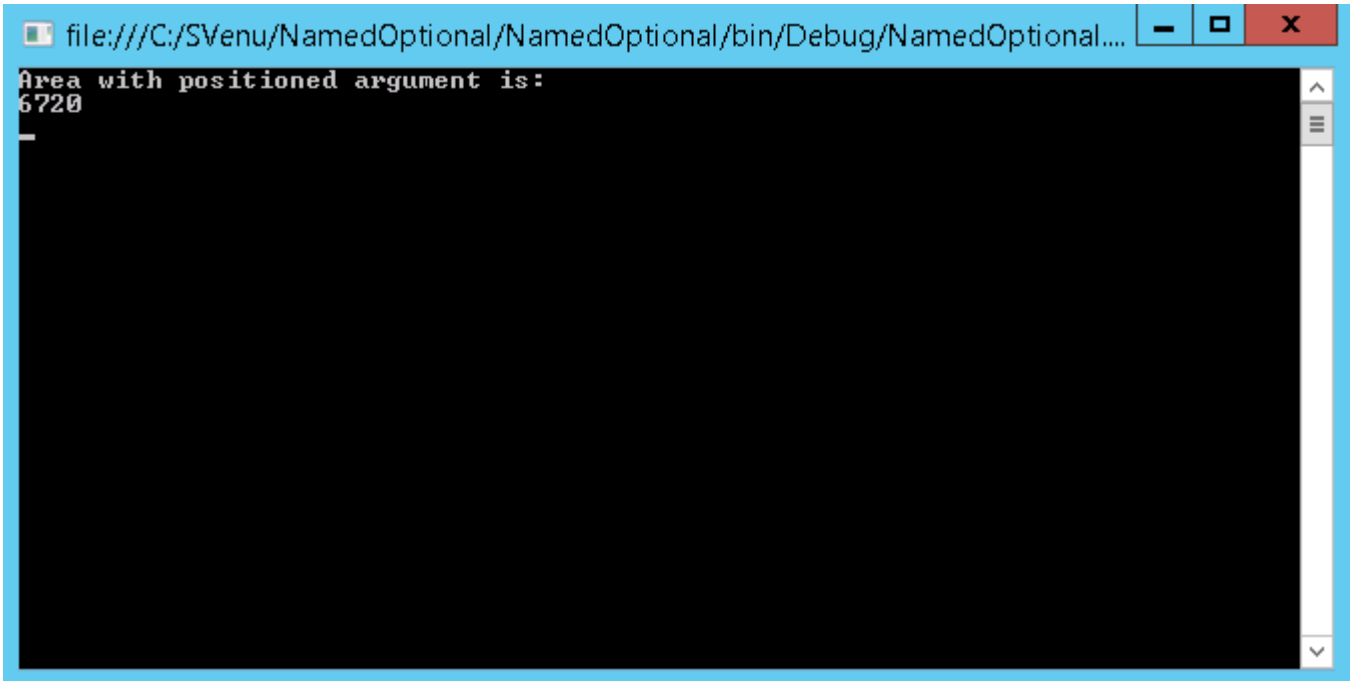
En este nuestro primer argumento es la longitud (es decir, 120) y el segundo argumento es el ancho (es decir, 56). Y estamos calculando el área por esa función. Y siguiente es la definición de la función.

```
private static double FindArea(int length, int width)
       {
           try
           {
               return (length* width);
           }
           catch (Exception)
           {
               throw new NotImplementedException();
           }
       }
```

Así que en la primera llamada a la función, acabamos de pasar los argumentos por su posición. ¿Derecha?

```
double area;
Console.WriteLine("Area with positioned argument is: ");
area = FindArea(120, 56);
Console.WriteLine(area);
Console.Read();
```

Si ejecuta esto, obtendrá una salida de la siguiente manera.




Ahora aquí vienen las características de un argumento con nombre. Por favor, consulte la función de llamada anterior.

```
Console.WriteLine("Area with Named argument is: ");
area = FindArea(length: 120, width: 56);
Console.WriteLine(area);
Console.Read();
```

Aquí estamos dando los argumentos nombrados en la llamada al método.

```
area = FindArea(length: 120, width: 56);
```

Ahora, si ejecuta este programa, obtendrá el mismo resultado. Podemos dar los nombres al revés en la llamada al método si estamos usando los argumentos nombrados.

```
Console.WriteLine("Area with Named argument vice versa is: ");
area = FindArea(width: 120, length: 56);
Console.WriteLine(area);
Console.Read();
```

Uno de los usos importantes de un argumento con nombre es que, cuando lo usa en su programa, mejora la legibilidad de su código. Simplemente dice cuál es tu argumento o qué es.

También puedes dar los argumentos posicionales. Eso significa, una combinación de ambos argumentos posicionales y argumentos con nombre.

```
Console.WriteLine("Area with Named argument Positional Argument : ");
            area = FindArea(120, width: 56);
            Console.WriteLine(area);
            Console.Read();
```

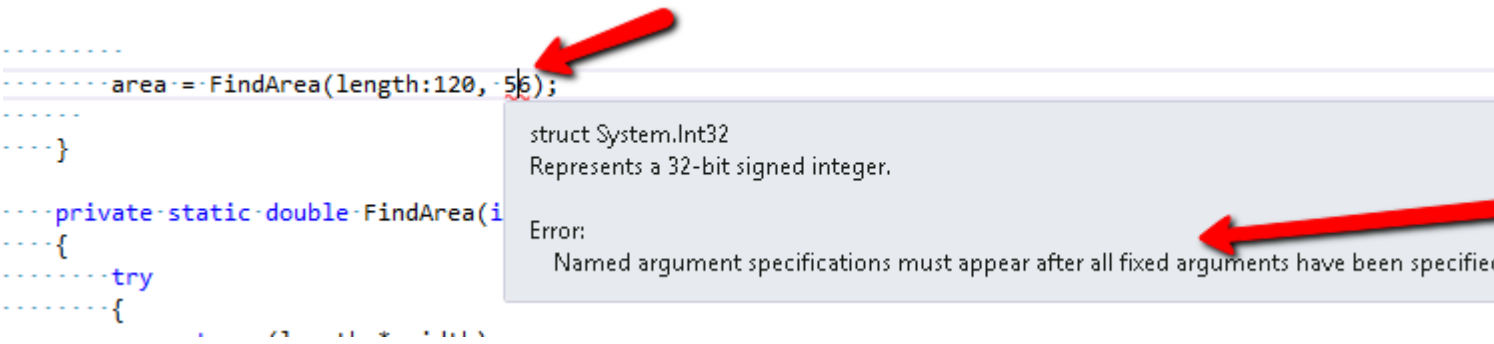
En el ejemplo anterior pasamos 120 como la longitud y 56 como un argumento con nombre para el ancho del parámetro.

También hay algunas limitaciones. Vamos a discutir la limitación de un argumento con nombre ahora.

**Limitación de usar un argumento con nombre**

La especificación de argumento con nombre debe aparecer después de que se hayan especificado todos los argumentos fijos.

Si usa un argumento con nombre antes de un argumento fijo, obtendrá un error de tiempo de compilación de la siguiente manera.



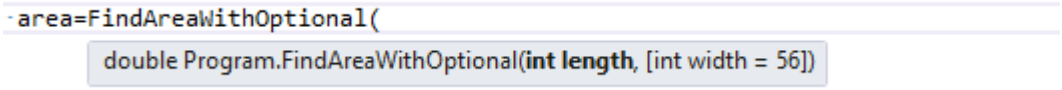

La especificación del argumento con nombre debe aparecer después de que se hayan especificado todos los argumentos fijos

## Argumentos opcionales

Consideremos que lo anterior es nuestra definición de función con argumentos opcionales.

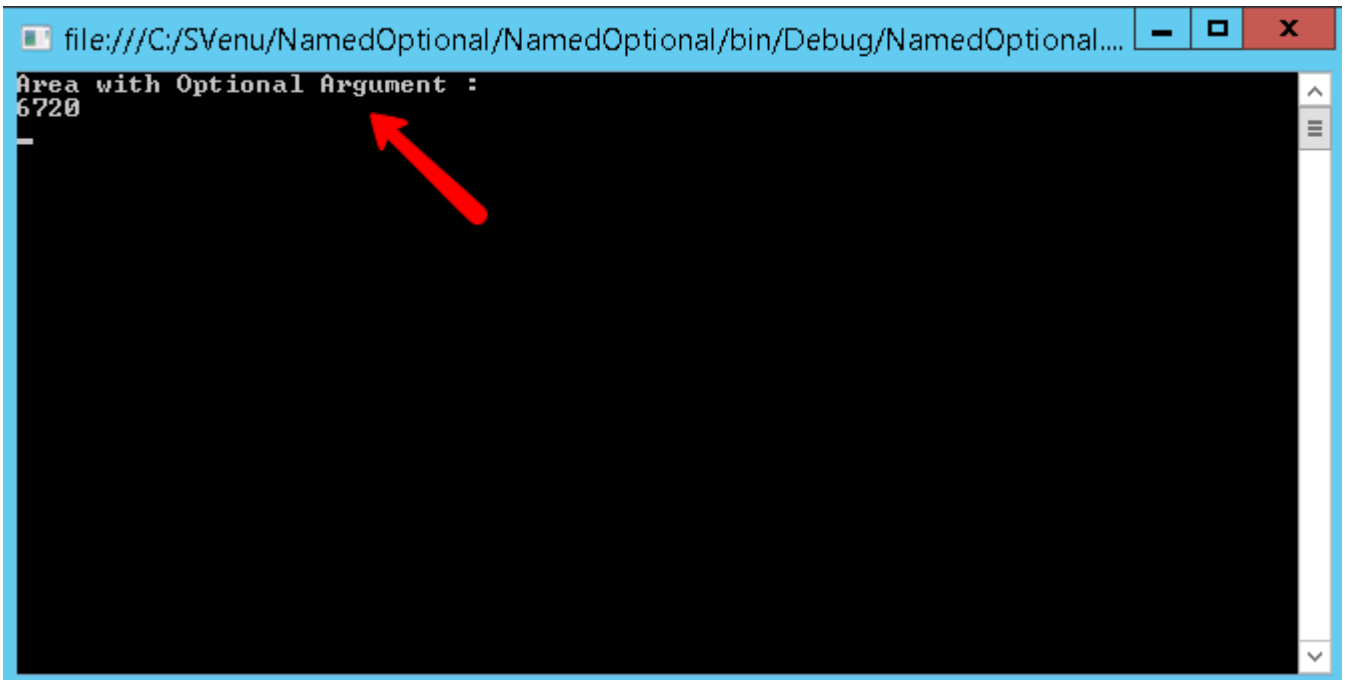
```
private static double FindAreaWithOptional(int length, int width=56)
       {
           try
           {
               return (length * width);
           }
           catch (Exception)
           {
               throw new NotImplementedException();
           }
       }
```

Aquí hemos establecido el valor para el ancho como opcional y hemos dado el valor como 56. Si observa, el IntelliSense mismo le muestra el argumento opcional como se muestra en la imagen de abajo.




```
Console.WriteLine("Area with Optional Argument : ");
area = FindAreaWithOptional(120);
Console.WriteLine(area);
Console.Read();
```

Tenga en cuenta que no obtuvimos ningún error durante la compilación y le daremos una salida de la siguiente manera.




**Usando el atributo opcional.**

Otra forma de implementar el argumento opcional es mediante el uso de la palabra clave `[Optional]` . Si no pasa el valor del argumento opcional, el valor predeterminado de ese tipo de datos se asigna a ese argumento. La palabra clave `Optional` está presente en el espacio de nombres "Runtime.InteropServices".

```
using System.Runtime.InteropServices;
private static double FindAreaWithOptional(int length, [Optional]int width)
   {
       try
       {
           return (length * width);
       }
       catch (Exception)
       {
           throw new NotImplementedException();
       }
   }

area = FindAreaWithOptional(120);  //area=0
```

Y cuando llamamos a la función, obtenemos 0 porque el segundo argumento no se pasa y el valor predeterminado de int es 0, por lo que el producto es 0.

Lea Argumentos nombrados y opcionales en línea:
https://riptutorial.com/es/csharp/topic/5220/argumentos-nombrados-y-opcionales

# Capítulo 12: Arrays

## Sintaxis

- **Declarar una matriz:**

  <tipo> [] <nombre>;

- **Declarando matriz bidimensional:**

  <tipo> [,] <nombre> = nuevo <tipo> [<valor>, <valor>];

- **Declarar una matriz irregular:**

  <tipo> [] <nombre> = nuevo <tipo> [<valor>];

- **Declarar un subarreglo para una matriz irregular:**

  <nombre> [<valor>] = nuevo <tipo> [<valor>];

- **Inicializando una matriz sin valores:**

  <nombre> = nuevo <tipo> [<longitud>];

- **Inicializando una matriz con valores:**

  <nombre> = nuevo <tipo> [] {<valor>, <valor>, <valor>, ...};

- **Inicializando una matriz bidimensional con valores:**

  <nombre> = nuevo <tipo> [,] {{<valor>, <valor>}, {<valor>, <valor>}, ...};

- **Accediendo a un elemento en el índice i:**

  <nombre> [i]

- **Obtención de la longitud de la matriz:**

  <nombre> .Longitud

## Observaciones

En C #, una matriz es un tipo de referencia, lo que significa que es *anulable* .

Una matriz tiene una longitud fija, lo que significa que no puede `.Add()` a ella o `.Remove()` de ella. Para utilizar estos, necesitaría una matriz dinámica: `List` o `ArrayList` .

## Examples

## Covarianza Array

```
string[] strings = new[] {"foo", "bar"};
object[] objects = strings; // implicit conversion from string[] to object[]
```

Esta conversión no es de tipo seguro. El siguiente código generará una excepción de tiempo de ejecución:

```
string[] strings = new[] {"Foo"};
object[] objects = strings;

objects[0] = new object(); // runtime exception, object is not string
string str = strings[0];   // would have been bad if above assignment had succeeded
```

## Obtención y configuración de valores de matriz

```
int[] arr = new int[] { 0, 10, 20, 30};

// Get
Console.WriteLine(arr[2]); // 20

// Set
arr[2] = 100;

// Get the updated value
Console.WriteLine(arr[2]); // 100
```

## Declarando una matriz

Una matriz se puede declarar y rellenar con el valor predeterminado utilizando la sintaxis de inicialización entre corchetes ( `[]` ). Por ejemplo, creando una matriz de 10 enteros:

```
int[] arr = new int[10];
```

Los índices en C # están basados en cero. Los índices de la matriz anterior serán 0-9. Por ejemplo:

```
int[] arr = new int[3] {7,9,4};
Console.WriteLine(arr[0]); //outputs 7
Console.WriteLine(arr[1]); //outputs 9
```

Lo que significa que el sistema comienza a contar el índice de elementos desde 0. Además, los accesos a los elementos de las matrices se realizan en **un tiempo constante** . Eso significa que acceder al primer elemento de la matriz tiene el mismo costo (en el tiempo) de acceder al segundo elemento, al tercer elemento y así sucesivamente.

También puede declarar una referencia simple a una matriz sin crear una instancia de una matriz.

```
int[] arr = null;   // OK, declares a null reference to an array.
int first = arr[0]; // Throws System.NullReferenceException because there is no actual array.
```

Una matriz también se puede crear e inicializar con valores personalizados utilizando la sintaxis de inicialización de la colección:

```
int[] arr = new int[] { 24, 2, 13, 47, 45 };
```

La `new int[]` porción `new int[]` se puede omitir cuando se declara una variable de matriz. Esta no es una *expresión* independiente, por lo que usarla como parte de una llamada diferente no funciona (para eso, usa la versión con la `new` ):

```
int[] arr = { 24, 2, 13, 47, 45 };  // OK
int[] arr1;
arr1 = { 24, 2, 13, 47, 45 };       // Won't compile
```

**Arrays implícitamente escritos**

Alternativamente, en combinación con la palabra clave `var` , el tipo específico se puede omitir para que se infiera el tipo de la matriz:

```
// same as int[]
var arr = new [] { 1, 2, 3 };
// same as string[]
var arr = new [] { "one", "two", "three" };
// same as double[]
var arr = new [] { 1.0, 2.0, 3.0 };
```

## Iterar sobre una matriz

```
int[] arr = new int[] {1, 6, 3, 3, 9};

for (int i = 0; i < arr.Length; i++)
{
    Console.WriteLine(arr[i]);
}
```

utilizando foreach:

```
foreach (int element in arr)
{
    Console.WriteLine(element);
}
```

Uso de acceso no seguro con punteros https://msdn.microsoft.com/en-ca/library/y31yhkeb.aspx

```
unsafe
{
    int length = arr.Length;
    fixed (int* p = arr)
    {
        int* pInt = p;
        while (length-- > 0)
        {
            Console.WriteLine(*pInt);
```

```
            pInt++;// move pointer to next element
        }
    }
}
```

Salida:

1
6
3
3
9

## Matrices multidimensionales

Las matrices pueden tener más de una dimensión. El siguiente ejemplo crea una matriz bidimensional de diez filas y diez columnas:

```
int[,] arr = new int[10, 10];
```

Una matriz de tres dimensiones:

```
int[,,] arr = new int[10, 10, 10];
```

También puede inicializar la matriz tras la declaración:

```
int[,] arr = new int[4, 2] { {1, 1}, {2, 2}, {3, 3}, {4, 4} };

// Access a member of the multi-dimensional array:
Console.Out.WriteLine(arr[3, 1]);  // 4
```

## Matrices dentadas

Las matrices irregulares son matrices que, en lugar de tipos primitivos, contienen matrices (u otras colecciones). Es como una matriz de matrices: cada elemento de matriz contiene otra matriz.

Son similares a las matrices multidimensionales, pero tienen una ligera diferencia: como las matrices multidimensionales se limitan a un número fijo de filas y columnas, con matrices irregulares, cada fila puede tener un número diferente de columnas.

**Declarar una matriz irregular**

Por ejemplo, declarando una matriz irregular con 8 columnas:

```
int[][] a = new int[8][];
```

El segundo `[]` se inicializa sin un número. Para inicializar las matrices secundarias, tendría que

hacerlo por separado:

```
for (int i = 0; i < a.length; i++)
{
    a[i] = new int[10];
}
```

**Obtención / Configuración de valores**

Ahora, obtener uno de los subarrays es fácil. Imprimamos todos los números de la tercera columna de `a` :

```
for (int i = 0; i < a[2].length; i++)
{
    Console.WriteLine(a[2][i]);
}
```

Obteniendo un valor específico:

```
a[<row_number>][<column_number>]
```

Establecer un valor específico:

```
a[<row_number>][<column_number>] = <value>
```

**Recuerde** : siempre se recomienda utilizar matrices dentadas (matrices de matrices) en lugar de matrices multidimensionales (matrices). Es más rápido y seguro de usar.

---

**Nota sobre el orden de los corchetes.**

Considere una matriz tridimensional de matrices de cinco dimensiones de matrices unidimensionales de `int` . Esto está escrito en C # como:

```
int[,,][,,,,][] arr = new int[8, 10, 12][,,,,][];
```

En el sistema de tipos de CLR, el Convenio para la ordenación de los soportes se invierte, por lo que con lo anterior `arr` ejemplo tenemos:

```
    arr.GetType().ToString() == "System.Int32[][,,,,][,,]"
```

y de la misma manera:

```
    typeof(int[,,][,,,,][]).ToString() == "System.Int32[][,,,,][,,]"
```

## Comprobando si una matriz contiene otra matriz

```
public static class ArrayHelpers
```

```
{
    public static bool Contains<T>(this T[] array, T[] candidate)
    {
        if (IsEmptyLocate(array, candidate))
            return false;

        if (candidate.Length > array.Length)
            return false;

        for (int a = 0; a <= array.Length - candidate.Length; a++)
        {
            if (array[a].Equals(candidate[0]))
            {
                int i = 0;
                for (; i < candidate.Length; i++)
                {
                    if (false == array[a + i].Equals(candidate[i]))
                        break;
                }
                if (i == candidate.Length)
                    return true;
            }
        }
        return false;
    }

    static bool IsEmptyLocate<T>(T[] array, T[] candidate)
    {
        return array == null
            || candidate == null
            || array.Length == 0
            || candidate.Length == 0
            || candidate.Length > array.Length;
    }
}
```

/// Muestra

```
byte[] EndOfStream = Encoding.ASCII.GetBytes("---3141592---");
byte[] FakeReceivedFromStream = Encoding.ASCII.GetBytes("Hello, world!!!---3141592---");
if (FakeReceivedFromStream.Contains(EndOfStream))
{
    Console.WriteLine("Message received");
}
```

## Inicializando una matriz llena con un valor no predeterminado repetido

Como sabemos, podemos declarar una matriz con valores predeterminados:

```
int[] arr = new int[10];
```

Esto creará una matriz de 10 enteros con cada elemento de la matriz que tiene el valor `0` (el valor predeterminado de tipo `int` ).

Para crear una matriz inicializada con un valor no predeterminado, podemos usar `Enumerable.Repeat` desde el espacio de nombres `System.Linq` :

1. Para crear una matriz `bool` de tamaño 10 rellena con **"true"**

```
bool[] booleanArray = Enumerable.Repeat(true, 10).ToArray();
```

2. Para crear una matriz `int` de tamaño 5 rellena con **"100"**

```
int[] intArray = Enumerable.Repeat(100, 5).ToArray();
```

3. Para crear una matriz de `string` de tamaño 5 rellena con **"C #"**

```
string[] strArray = Enumerable.Repeat("C#", 5).ToArray();
```

## Copiando matrices

Copiando una matriz parcial con el `Array.Copy()` estático `Array.Copy()` , comenzando en el índice 0 tanto en el origen como en el destino:

```
var sourceArray = new int[] { 11, 12, 3, 5, 2, 9, 28, 17 };
var destinationArray= new int[3];
Array.Copy(sourceArray, destinationArray, 3);

// destinationArray will have 11,12 and 3
```

Copiando toda la matriz con el método de instancia `CopyTo()` , comenzando en el índice 0 del origen y el índice especificado en el destino:

```
var sourceArray = new int[] { 11, 12, 7 };
var destinationArray = new int[6];
sourceArray.CopyTo(destinationArray, 2);

// destinationArray will have 0, 0, 11, 12, 7 and 0
```

`Clone` se utiliza para crear una copia de un objeto de matriz.

```
var sourceArray = new int[] { 11, 12, 7 };
var destinationArray = (int)sourceArray.Clone();

//destinationArray will be created and will have 11,12,17.
```

Tanto `CopyTo` como `Clone` realizan una copia superficial, lo que significa que el contenido contiene referencias al mismo objeto que los elementos de la matriz original.

## Creando una matriz de números secuenciales

LINQ proporciona un método que facilita la creación de una colección con números secuenciales. Por ejemplo, puede declarar una matriz que contiene los números enteros entre 1 y 100.

El método `Enumerable.Range` nos permite crear una secuencia de números enteros desde una posición de inicio especificada y una serie de elementos.

El método toma dos argumentos: el valor de inicio y el número de elementos a generar.

```
Enumerable.Range(int start, int count)
```

*Tenga en `count` que la `count` no puede ser negativa.*

# Uso:

```
int[] sequence = Enumerable.Range(1, 100).ToArray();
```

Esto generará una matriz que contiene los números del 1 al 100 ( `[1, 2, 3, ..., 98, 99, 100]` ).

Debido a que el método `Range` devuelve un `IEnumerable<int>` , podemos usar otros métodos LINQ en él:

```
int[] squares = Enumerable.Range(2, 10).Select(x => x * x).ToArray();
```

Esto generará una matriz que contiene 10 cuadrados enteros comenzando en `4` : `[4, 9, 16, ..., 100, 121]` .

## Comparando matrices para la igualdad

LINQ proporciona una función incorporada para verificar la igualdad de dos `IEnumerable` s, y esa función se puede usar en arreglos.

La función `SequenceEqual` devolverá `true` si las matrices tienen la misma longitud y los valores en los índices correspondientes son iguales, y `false` contrario.

```
int[] arr1 = { 3, 5, 7 };
int[] arr2 = { 3, 5, 7 };
bool result = arr1.SequenceEqual(arr2);
Console.WriteLine("Arrays equal? {0}", result);
```

Esto imprimirá:

```
Arrays equal? True
```

## Arreglos como instancias IEnumerable <>

Todas las matrices implementan la interfaz `IList` no genérica (y, por lo tanto, las interfaces de base `ICollection` y `IEnumerable` no genéricas).

Más importante aún, las matrices unidimensionales implementan las interfaces genéricas `IList<>` e `IReadOnlyList<>` (y sus interfaces base) para el tipo de datos que contienen. Esto significa que pueden tratarse como tipos enumerables genéricos y pasarse a una variedad de métodos sin necesidad de convertirlos primero a una forma no de matriz.

```
int[] arr1 = { 3, 5, 7 };
IEnumerable<int> enumerableIntegers = arr1; //Allowed because arrays implement IEnumerable<T>
List<int> listOfIntegers = new List<int>();
listOfIntegers.AddRange(arr1); //You can pass in a reference to an array to populate a List.
```

Después de ejecutar este código, la lista `listOfIntegers` contendrá una `List<int>` contiene los valores 3, 5 y 7.

La `IEnumerable<>` significa que las matrices se pueden consultar con LINQ, por ejemplo, `arr1.Select(i => 10 * i)` .

Lea Arrays en línea: https://riptutorial.com/es/csharp/topic/1429/arrays

# Capítulo 13: Asíncrono-espera

## Introducción

En C #, un método declarado `async` no se bloqueará dentro de un proceso síncrono, en caso de que esté utilizando operaciones basadas en E / S (por ejemplo, acceso web, trabajo con archivos, ...). El resultado de dichos métodos marcados asíncronos puede esperarse mediante el uso de la palabra clave `await` .

## Observaciones

Un método `async` puede devolver `void` , `Task` o `Task<T>` .

La `Task` tipo de retorno esperará a que finalice el método y el resultado se `void` . `Task<T>` devolverá un valor del tipo `T` después de que se complete el método.

`async` métodos `async` deben devolver `Task` o `Task<T>` , en lugar de `void` , en casi todas las circunstancias. `async void` métodos de `async void` no se pueden `await` , lo que conduce a una variedad de problemas. El único escenario en el que un `async` debe devolver un `void` es en el caso de un controlador de eventos.

`async` / `await` funciona al transformar su método `async` en una máquina de estado. Lo hace creando una estructura detrás de escena que almacena el estado actual y cualquier contexto (como las variables locales), y expone un método `MoveNext()` para avanzar los estados (y ejecutar cualquier código asociado) cada vez que se completa un proceso esperado.

## Examples

### Simples llamadas consecutivas

```
public async Task<JobResult> GetDataFromWebAsync()
{
  var nextJob = await _database.GetNextJobAsync();
  var response = await _httpClient.GetAsync(nextJob.Uri);
  var pageContents = await response.Content.ReadAsStringAsync();
  return await _database.SaveJobResultAsync(pageContents);
}
```

Lo principal a tener en cuenta es que, si bien todos los `await` método -ed se llama de forma asincrónica - y por el tiempo de la llamada del control es cedido de nuevo al sistema - el flujo en el interior del método es lineal y no requiere ningún tratamiento especial debido a las asincronía. Si alguno de los métodos llamados falla, la excepción se procesará "como se esperaba", lo que en este caso significa que la ejecución del método se anulará y la excepción subirá la pila.

### Probar / Atrapar / Finalmente

6.0

A partir de C # 6.0, la palabra clave `await` ahora se puede usar dentro de un bloque `catch` y `finally` .

```
try {
   var client = new AsyncClient();
   await client.DoSomething();
} catch (MyException ex) {
   await client.LogExceptionAsync();
   throw;
} finally {
   await client.CloseAsync();
}
```

5.0 6.0

Antes de C # 6.0, tendría que hacer algo como lo siguiente. Tenga en cuenta que 6.0 también limpió las comprobaciones nulas con el operador de propagación nula .

```
AsynClient client;
MyException caughtException;
try {
     client = new AsyncClient();
     await client.DoSomething();
} catch (MyException ex) {
     caughtException = ex;
}

if (client != null) {
    if (caughtException != null) {
       await client.LogExceptionAsync();
    }
    await client.CloseAsync();
    if (caughtException != null) throw caughtException;
}
```

Tenga en cuenta que si espera una tarea no creada por `async` (p. Ej., Una tarea creada por `Task.Run` ), algunos depuradores pueden interrumpir las excepciones generadas por la tarea, incluso cuando parece ser manejada por el try / catch que la rodea. Esto sucede porque el depurador considera que no está manejado con respecto al código de usuario. En Visual Studio, hay una opción llamada "Just My Code" , que se puede desactivar para evitar que el depurador se rompa en tales situaciones.

## Configuración de Web.config para apuntar a 4.5 para un comportamiento asíncrono correcto.

El web.config system.web.httpRuntime debe apuntar a 4.5 para garantizar que el subproceso inquiline el contexto de la solicitud antes de reanudar su método asíncrono.

```
<httpRuntime targetFramework="4.5" />
```

Async y espera tienen un comportamiento indefinido en ASP.NET antes de 4.5. Async / await se reanudará en un hilo arbitrario que puede no tener el contexto de solicitud. Las aplicaciones bajo

carga fallarán aleatoriamente con excepciones de referencia nulas que accedan a HttpContext después de la espera. Usar HttpContext.Current en WebApi es peligroso debido a async

## Llamadas concurrentes

Es posible esperar múltiples llamadas simultáneamente invocando primero las tareas que *se pueden* esperar y *luego* esperándolas.

```
public async Task RunConcurrentTasks()
{
    var firstTask = DoSomethingAsync();
    var secondTask = DoSomethingElseAsync();

    await firstTask;
    await secondTask;
}
```

Alternativamente, `Task.WhenAll` se puede usar para agrupar múltiples tareas en una sola `Task` , que se completa cuando todas sus tareas pasadas están completas.

```
public async Task RunConcurrentTasks()
{
    var firstTask = DoSomethingAsync();
    var secondTask = DoSomethingElseAsync();

    await Task.WhenAll(firstTask, secondTask);
}
```

También puedes hacer esto dentro de un bucle, por ejemplo:

```
List<Task> tasks = new List<Task>();
while (something) {
    // do stuff
    Task someAsyncTask = someAsyncMethod();
    tasks.Add(someAsyncTask);
}

await Task.WhenAll(tasks);
```

Para obtener resultados de una tarea después de esperar varias tareas con Tarea. Cuando todo, simplemente vuelva a esperar la tarea. Ya que la tarea ya está completada, solo devolverá el resultado

```
var task1 = SomeOpAsync();
var task2 = SomeOtherOpAsync();

await Task.WhenAll(task1, task2);

var result = await task2;
```

Además, `Task.WhenAny` se puede usar para ejecutar múltiples tareas en paralelo, como `Task.WhenAll` arriba, con la diferencia de que este método se completará cuando se complete *cualquiera* de las tareas proporcionadas.

```
public async Task RunConcurrentTasksWhenAny()
{
    var firstTask = TaskOperation("#firstTask executed");
    var secondTask = TaskOperation("#secondTask executed");
    var thirdTask = TaskOperation("#thirdTask executed");
    await Task.WhenAny(firstTask, secondTask, thirdTask);
}
```

La `Task` devuelta por `RunConcurrentTasksWhenAny` se completará cuando se complete cualquiera de `firstTask` , `secondTask` o `thirdTask` .

## Espera operador y palabra clave asíncrona.

`await` operador y palabra clave `async` se unen:

> El método asíncrono en el que **esperan** se utiliza debe ser modificado por la palabra clave **asíncrono.**

Lo contrario no siempre es cierto: puede marcar un método como `async` sin usar `await` en su cuerpo.

Lo que `await` realmente es suspender la ejecución del código hasta que se complete la tarea esperada; Cualquier tarea puede ser esperada.

**Nota:** no puede esperar por un método asíncrono que no devuelve nada (vacío).

En realidad, la palabra 'suspender' es un poco engañosa porque no solo se detiene la ejecución, sino que el hilo puede quedar libre para ejecutar otras operaciones. Bajo el capó, `await` se implementa con un poco de magia de compilación: divide un método en dos partes: antes y después de `await` . La última parte se ejecuta cuando se completa la tarea esperada.

Si ignoramos algunos detalles importantes, el compilador aproximadamente lo hace por usted:

```
public async Task<TResult> DoIt()
{
    // do something and acquire someTask of type Task<TSomeResult>
    var awaitedResult = await someTask;
    // ... do something more and produce result of type TResult
    return result;
}
```

se convierte en:

```
public Task<TResult> DoIt()
{
    // ...
    return someTask.ContinueWith(task => {
        var result = ((Task<TSomeResult>)task).Result;
        return DoIt_Continuation(result);
    });
}

private TResult DoIt_Continuation(TSomeResult awaitedResult)
```

```
{
    // ...
}
```

Cualquier método habitual puede convertirse en asíncrono de la siguiente manera:

```
await Task.Run(() => YourSyncMethod());
```

Esto puede ser ventajoso cuando necesita ejecutar un método de ejecución prolongada en el subproceso de la interfaz de usuario sin congelar la interfaz de usuario.

Pero hay un comentario muy importante aquí: **asíncrono no siempre significa concurrente (paralelo o incluso multiproceso).** Incluso en un solo hilo, `async` - `await` todavía permite el código asíncrono. Por ejemplo, vea este programador de tareas personalizado. Un programador de tareas tan "loco" puede simplemente convertir las tareas en funciones que se llaman dentro del procesamiento del bucle de mensajes.

Necesitamos preguntarnos: ¿Qué hilo ejecutará la continuación de nuestro método `DoIt_Continuation` ?

Por defecto, el operador a la `await` programa la ejecución de continuación con el contexto de sincronización actual. Esto significa que, de forma predeterminada, WinForms y WPF se ejecutan en el subproceso de la interfaz de usuario. Si, por algún motivo, necesita cambiar este comportamiento, use el método `Task.ConfigureAwait()` :

```
await Task.Run(() => YourSyncMethod()).ConfigureAwait(continueOnCapturedContext: false);
```

## Devolviendo una tarea sin esperar

Los métodos que realizan operaciones asíncronas no tienen que usar `await` si:

- Solo hay una llamada asíncrona dentro del método.
- La llamada asíncrona se encuentra al final del método.
- La excepción de captura / manejo que puede ocurrir dentro de la Tarea no es necesaria

Considere este método que devuelve una `Task` :

```
public async Task<User> GetUserAsync(int id)
{
    var lookupKey = "Users" + id;

    return await dataStore.GetByKeyAsync(lookupKey);
}
```

Si `GetByKeyAsync` tiene la misma firma que `GetUserAsync` (que devuelve una `Task<User>` ), el método se puede simplificar:

```
public Task<User> GetUserAsync(int id)
{
```

```
    var lookupKey = "Users" + id;

    return dataStore.GetByKeyAsync(lookupKey);
}
```

En este caso, el método no necesita estar marcado como `async` , a pesar de que está realizando una operación asíncrona. La tarea devuelta por `GetByKeyAsync` se pasa directamente al método de llamada, donde se `await` .

**Importante** : Devolver la `Task` lugar de esperarla, cambia el comportamiento de excepción del método, ya que no lanzará la excepción dentro del método que inicia la tarea sino en el método que la espera.

```
public Task SaveAsync()
{
    try {
        return dataStore.SaveChangesAsync();
    }
    catch(Exception ex)
    {
        // this will never be called
        logger.LogException(ex);
    }
}

// Some other code calling SaveAsync()

// If exception happens, it will be thrown here, not inside SaveAsync()
await SaveAsync();
```

Esto mejorará el rendimiento, ya que ahorrará al compilador la generación de una máquina de estado **asíncrono** adicional.

## El bloqueo en el código asíncrono puede causar interbloqueos

Es una mala práctica bloquear las llamadas asíncronas, ya que puede provocar interbloqueos en entornos que tienen un contexto de sincronización. La mejor práctica es usar async / await "hasta el final". Por ejemplo, el siguiente código de Windows Forms provoca un interbloqueo:

```
private async Task<bool> TryThis()
{
    Trace.TraceInformation("Starting TryThis");
    await Task.Run(() =>
    {
        Trace.TraceInformation("In TryThis task");
        for (int i = 0; i < 100; i++)
        {
            // This runs successfully - the loop runs to completion
            Trace.TraceInformation("For loop " + i);
            System.Threading.Thread.Sleep(10);
        }
    });

    // This never happens due to the deadlock
    Trace.TraceInformation("About to return");
```

```
    return true;
}

// Button click event handler
private void button1_Click(object sender, EventArgs e)
{
    // .Result causes this to block on the asynchronous call
    bool result = TryThis().Result;
    // Never actually gets here
    Trace.TraceInformation("Done with result");
}
```

Esencialmente, una vez que se completa la llamada asíncrona, espera que el contexto de sincronización esté disponible. Sin embargo, el controlador de eventos "se mantiene" en el contexto de sincronización mientras espera que se `TryThis()` método `TryThis()` , lo que provoca una espera circular.

Para arreglar esto, el código debe ser modificado para

```
private async void button1_Click(object sender, EventArgs e)
{
  bool result = await TryThis();
  Trace.TraceInformation("Done with result");
}
```

Nota: los controladores de eventos son el único lugar donde se debe usar el `async void` (porque no puede esperar un método de `async void` ).

## Async / await solo mejorará el rendimiento si permite que la máquina realice trabajo adicional

Considere el siguiente código:

```
public async Task MethodA()
{
     await MethodB();
     // Do other work
}

public async Task MethodB()
{
     await MethodC();
     // Do other work
}

public async Task MethodC()
{
     // Or await some other async work
     await Task.Delay(100);
}
```

Esto no funcionará mejor que

```
public void MethodA()
```

```
{
     MethodB();
     // Do other work
}

public void MethodB()
{
     MethodC();
     // Do other work
}

public void MethodC()
{
     Thread.Sleep(100);
}
```

El propósito principal de async / await es permitir que la máquina realice trabajo adicional, por ejemplo, para permitir que el subproceso que realiza la llamada realice otro trabajo mientras espera el resultado de alguna operación de E / S. En este caso, al subproceso de llamada nunca se le permite hacer más trabajo de lo que hubiera podido hacer de otra manera, por lo que no hay ganancia de rendimiento al simplemente llamar a `MethodA()` , `MethodB()` y `MethodC()` sincrónica.

Lea Asíncrono-espera en línea: https://riptutorial.com/es/csharp/topic/48/asincrono-espera

# Capítulo 14: Async / await, Backgroundworker, tareas y ejemplos de subprocesos

## Observaciones

Para ejecutar cualquiera de estos ejemplos, llámelos así:

```
static void Main()
{
    new Program().ProcessDataAsync();
    Console.ReadLine();
}
```

## Examples

### ASP.NET Configure Await

Cuando ASP.NET maneja una solicitud, se asigna un subproceso desde el grupo de subprocesos y se crea un **contexto de solicitud** . El contexto de solicitud contiene información sobre la solicitud actual a la que se puede acceder a través de la propiedad estática `HttpContext.Current` . El contexto de solicitud para la solicitud se asigna al hilo que maneja la solicitud.

Un contexto de solicitud dado **solo puede estar activo en un hilo a la vez** .

Cuando la ejecución llega a la `await` , el subproceso que maneja una solicitud se devuelve al grupo de subprocesos mientras se ejecuta el método asíncrono y el contexto de la solicitud es libre para que otro subproceso lo utilice.

```
public async Task<ActionResult> Index()
{
    // Execution on the initially assigned thread
    var products = await dbContext.Products.ToListAsync();

    // Execution resumes on a "random" thread from the pool
    // Execution continues using the original request context.
    return View(products);
}
```

Cuando la tarea se completa, el grupo de hilos asigna otro hilo para continuar la ejecución de la solicitud. El contexto de solicitud se asigna a este hilo. Este puede o no ser el hilo original.

### Bloqueando

Cuando se espera el resultado de una llamada a un método `async` , pueden surgir puntos muertos

**sincrónicos** . Por ejemplo, el siguiente código resultará en un interbloqueo cuando se `IndexSync()` :

```
public async Task<ActionResult> Index()
{
    // Execution on the initially assigned thread
    List<Product> products = await dbContext.Products.ToListAsync();

    // Execution resumes on a "random" thread from the pool
    return View(products);
}

public ActionResult IndexSync()
{
    Task<ActionResult> task = Index();

    // Block waiting for the result synchronously
    ActionResult result = Task.Result;

    return result;
}
```

Esto se debe a que, de forma predeterminada, la tarea esperada, en este caso `db.Products.ToListAsync()` capturará el contexto (en el caso de ASP.NET el contexto de la solicitud) e intentará usarlo una vez que se haya completado.

Cuando toda la pila de llamadas es asíncrona, no hay problema porque, una vez `await` se alcanza la `await` el hilo original se libera, liberando el contexto de la solicitud.

Cuando `Task.Result` sincrónica utilizando `Task.Result` o `Task.Wait()` (u otros métodos de bloqueo), el subproceso original todavía está activo y conserva el contexto de la solicitud. El método esperado aún funciona de forma asíncrona y una vez que la devolución de llamada intenta ejecutarse, es decir, una vez que la tarea esperada ha regresado, intenta obtener el contexto de la solicitud.

Por lo tanto, el interbloqueo surge porque mientras el subproceso de bloqueo con el contexto de la solicitud está esperando a que se complete la operación asíncrona, la operación asíncrona está tratando de obtener el contexto de la solicitud para poder completarla.

## ConfigureAwait

De forma predeterminada, las llamadas a una tarea esperada capturarán el contexto actual e intentarán reanudar la ejecución en el contexto una vez completado.

Al utilizar `ConfigureAwait(false)` esto se puede evitar y los puntos muertos se pueden evitar.

```
public async Task<ActionResult> Index()
{
    // Execution on the initially assigned thread
    List<Product> products = await dbContext.Products.ToListAsync().ConfigureAwait(false);

    // Execution resumes on a "random" thread from the pool without the original request
context
```

```
    return View(products);
}

public ActionResult IndexSync()
{
    Task<ActionResult> task = Index();

    // Block waiting for the result synchronously
    ActionResult result = Task.Result;

    return result;
}
```

Esto puede evitar puntos muertos cuando es necesario bloquear en código asíncrono, sin embargo, esto conlleva el costo de perder el contexto en la continuación (código después de la llamada a la espera).

En ASP.NET, esto significa que si su código después de una llamada `await someTask.ConfigureAwait(false);` intenta acceder a la información desde el contexto, por ejemplo, `HttpContext.Current.User` entonces la información se ha perdido. En este caso, el `HttpContext.Current` es nulo. Por ejemplo:

```
public async Task<ActionResult> Index()
{
    // Contains information about the user sending the request
    var user = System.Web.HttpContext.Current.User;

    using (var client = new HttpClient())
    {
        await client.GetAsync("http://google.com").ConfigureAwait(false);
    }

    // Null Reference Exception, Current is null
    var user2 = System.Web.HttpContext.Current.User;

    return View();
}
```

Si se usa `ConfigureAwait(true)` (equivalente a no tener ningún ConfigureAwait), entonces el `user` y el `user2` se llenan con los mismos datos.

Por este motivo, a menudo se recomienda usar `ConfigureAwait(false)` en el código de la biblioteca donde ya no se usa el contexto.

## Asíncrono / espera

Vea a continuación un ejemplo sencillo de cómo usar async / await para hacer cosas que requieren mucho tiempo en un proceso en segundo plano, al tiempo que mantiene la opción de hacer otras cosas que no necesitan esperar en las cosas que requieren mucho tiempo para completarse.

Sin embargo, si necesita trabajar con el resultado del método intensivo de tiempo más adelante, puede hacerlo esperando la ejecución.

```
public async Task ProcessDataAsync()
{
    // Start the time intensive method
    Task<int> task = TimeintensiveMethod(@"PATH_TO_SOME_FILE");

    // Control returns here before TimeintensiveMethod returns
    Console.WriteLine("You can read this while TimeintensiveMethod is still running.");

    // Wait for TimeintensiveMethod to complete and get its result
    int x = await task;
    Console.WriteLine("Count: " + x);
}

private async Task<int> TimeintensiveMethod(object file)
{
    Console.WriteLine("Start TimeintensiveMethod.");

    // Do some time intensive calculations...
    using (StreamReader reader = new StreamReader(file.ToString()))
    {
        string s = await reader.ReadToEndAsync();

        for (int i = 0; i < 10000; i++)
            s.GetHashCode();
    }
    Console.WriteLine("End TimeintensiveMethod.");

    // return something as a "result"
    return new Random().Next(100);
}
```

## Trabajador de fondo

Vea a continuación un ejemplo simple de cómo usar un objeto `BackgroundWorker` para realizar operaciones que requieren mucho tiempo en un hilo de fondo.

Necesitas:

1. Defina un método de trabajo que haga el trabajo intensivo en tiempo y llámelo desde un controlador de eventos para el evento `DoWork` de un `BackgroundWorker` .
2. Inicie la ejecución con `RunWorkerAsync` . Cualquier argumento requerido por el método de obrero unido a `DoWork` se puede pasar en a través de la `DoWorkEventArgs` parámetro para `RunWorkerAsync` .

Además del evento `DoWork` , la clase `BackgroundWorker` también define dos eventos que deben usarse para interactuar con la interfaz de usuario. Estos son opcionales.

- El evento `RunWorkerCompleted` se activa cuando los controladores de `DoWork` han completado.
- El evento `ProgressChanged` se activa cuando se llama al método `ReportProgress` .

```
public void ProcessDataAsync()
{
    // Start the time intensive method
    BackgroundWorker bw = new BackgroundWorker();
    bw.DoWork += BwDoWork;
```

```
    bw.RunWorkerCompleted += BwRunWorkerCompleted;
    bw.RunWorkerAsync(@"PATH_TO_SOME_FILE");

    // Control returns here before TimeintensiveMethod returns
    Console.WriteLine("You can read this while TimeintensiveMethod is still running.");
}

// Method that will be called after BwDoWork exits
private void BwRunWorkerCompleted(object sender, RunWorkerCompletedEventArgs e)
{
    // we can access possible return values of our Method via the Parameter e
    Console.WriteLine("Count: " + e.Result);
}

// execution of our time intensive Method
private void BwDoWork(object sender, DoWorkEventArgs e)
{
    e.Result = TimeintensiveMethod(e.Argument);
}

private int TimeintensiveMethod(object file)
{
    Console.WriteLine("Start TimeintensiveMethod.");

    // Do some time intensive calculations...
    using (StreamReader reader = new StreamReader(file.ToString()))
    {
        string s = reader.ReadToEnd();

       for (int i = 0; i < 10000; i++)
            s.GetHashCode();
    }
    Console.WriteLine("End TimeintensiveMethod.");

    // return something as a "result"
    return new Random().Next(100);
}
```

## Tarea

Vea a continuación un ejemplo simple de cómo usar una `Task` para hacer cosas que requieren mucho tiempo en un proceso en segundo plano.

Todo lo que necesita hacer es envolver su método intensivo en tiempo en una llamada `Task.Run()` .

```
public void ProcessDataAsync()
{
    // Start the time intensive method
    Task<int> t = Task.Run(() => TimeintensiveMethod(@"PATH_TO_SOME_FILE"));

    // Control returns here before TimeintensiveMethod returns
    Console.WriteLine("You can read this while TimeintensiveMethod is still running.");

    Console.WriteLine("Count: " + t.Result);
}

private int TimeintensiveMethod(object file)
```

```
{
    Console.WriteLine("Start TimeintensiveMethod.");

    // Do some time intensive calculations...
    using (StreamReader reader = new StreamReader(file.ToString()))
    {
        string s = reader.ReadToEnd();

        for (int i = 0; i < 10000; i++)
            s.GetHashCode();
    }
    Console.WriteLine("End TimeintensiveMethod.");

    // return something as a "result"
    return new Random().Next(100);
}
```

## Hilo

Vea a continuación un ejemplo simple de cómo usar un `Thread` para hacer cosas que requieren mucho tiempo en un proceso en segundo plano.

```
public async void ProcessDataAsync()
{
    // Start the time intensive method
    Thread t = new Thread(TimeintensiveMethod);

    // Control returns here before TimeintensiveMethod returns
    Console.WriteLine("You can read this while TimeintensiveMethod is still running.");
}

private void TimeintensiveMethod()
{
    Console.WriteLine("Start TimeintensiveMethod.");

    // Do some time intensive calculations...
    using (StreamReader reader = new StreamReader(@"PATH_TO_SOME_FILE"))
    {
        string v = reader.ReadToEnd();

        for (int i = 0; i < 10000; i++)
            v.GetHashCode();
    }
    Console.WriteLine("End TimeintensiveMethod.");
}
```

Como puede ver, no podemos devolver un valor de nuestro `TimeIntensiveMethod` porque `Thread` espera un método vacío como parámetro.

Para obtener un valor de retorno de un `Thread` use un evento o lo siguiente:

```
int ret;
Thread t= new Thread(() =>
{
    Console.WriteLine("Start TimeintensiveMethod.");

    // Do some time intensive calculations...
```

```
    using (StreamReader reader = new StreamReader(file))
    {
        string s = reader.ReadToEnd();

        for (int i = 0; i < 10000; i++)
            s.GetHashCode();
    }
    Console.WriteLine("End TimeintensiveMethod.");

    // return something to demonstrate the coolness of await-async
    ret = new Random().Next(100);
});

t.Start();
t.Join(1000);
Console.Writeline("Count: " + ret);
```

## Tarea "Ejecutar y olvidar" extensión

En ciertos casos (por ejemplo, el registro) puede ser útil ejecutar una tarea y no esperar el resultado. La siguiente extensión permite ejecutar la tarea y continuar la ejecución del código de descanso:

```
public static class TaskExtensions
{
    public static async void RunAndForget(
        this Task task, Action<Exception> onException = null)
    {
        try
        {
            await task;
        }
        catch (Exception ex)
        {
            onException?.Invoke(ex);
        }
    }
}
```

El resultado se espera solo dentro del método de extensión. Como se utiliza `async` / `await` , es posible detectar una excepción y llamar a un método opcional para manejarlo.

Un ejemplo de cómo usar la extensión:

```
var task = Task.FromResult(0); // Or any other task from e.g. external lib.
task.RunAndForget(
    e =>
    {
        // Something went wrong, handle it.
    });
```

Lea Async / await, Backgroundworker, tareas y ejemplos de subprocesos en línea: https://riptutorial.com/es/csharp/topic/3824/async---await--backgroundworker--tareas-y-ejemplos-de-subprocesos

# Capítulo 15: Atributos

## Examples

### Creando un atributo personalizado

```
//1) All attributes should be inherited from System.Attribute
//2) You can customize your attribute usage (e.g. place restrictions) by using
System.AttributeUsage Attribute
//3) You can use this attribute only via reflection in the way it is supposed to be used
//4) MethodMetadataAttribute is just a name. You can use it without "Attribute" postfix - e.g.
[MethodMetadata("This text could be retrieved via reflection")].
//5) You can overload an attribute constructors
[System.AttributeUsage(System.AttributeTargets.Method | System.AttributeTargets.Class)]
public class MethodMetadataAttribute : System.Attribute
{
    //this is custom field given just for an example
    //you can create attribute without any fields
    //even an empty attribute can be used - as marker
    public string Text { get; set; }

    //this constructor could be used as [MethodMetadata]
    public MethodMetadataAttribute ()
    {
    }

    //This constructor could be used as [MethodMetadata("String")]
    public MethodMetadataAttribute (string text)
    {
        Text = text;
    }
}
```

### Usando un atributo

```
[StackDemo(Text = "Hello, World!")]
public class MyClass
{
    [StackDemo("Hello, World!")]
    static void MyMethod()
    {
    }
}
```

### Leyendo un atributo

El método `GetCustomAttributes` devuelve una matriz de atributos personalizados aplicados al miembro. Después de recuperar esta matriz, puede buscar uno o más atributos específicos.

```
var attribute = typeof(MyClass).GetCustomAttributes().OfType<MyCustomAttribute>().Single();
```

O iterar a través de ellos.

```
foreach(var attribute in typeof(MyClass).GetCustomAttributes()) {
    Console.WriteLine(attribute.GetType());
}
```

`GetCustomAttribute` método de extensión `GetCustomAttribute` de `System.Reflection.CustomAttributeExtensions` recupera un atributo personalizado de un tipo específico, puede aplicarse a cualquier `MemberInfo` .

```
var attribute = (MyCustomAttribute)
typeof(MyClass).GetCustomAttribute(typeof(MyCustomAttribute));
```

`GetCustomAttribute` también tiene una firma genérica para especificar el tipo de atributo para buscar.

```
var attribute = typeof(MyClass).GetCustomAttribute<MyCustomAttribute>();
```

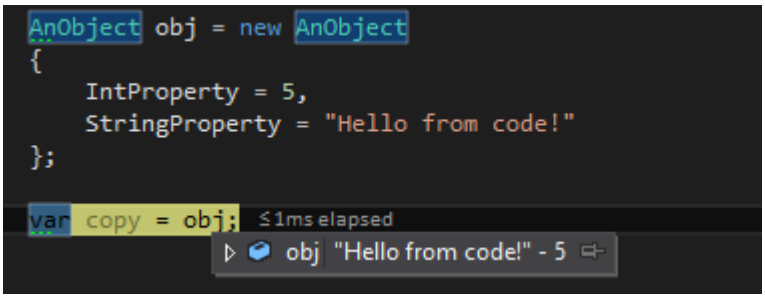
El argumento booleano `inherit` se puede pasar a ambos métodos. Si este valor se establece en `true` los antepasados del elemento también se inspeccionarán.

## DebuggerDisplay Attribute

Agregar el atributo `DebuggerDisplay` cambiará la forma en que el depurador muestra la clase cuando se pasa el cursor.

Las expresiones que están envueltas en `{}` serán evaluadas por el depurador. Esta puede ser una propiedad simple como en la siguiente muestra o una lógica más compleja.

```
[DebuggerDisplay("{StringProperty} - {IntProperty}")]
public class AnObject
{
   public int ObjectId { get; set; }
   public string StringProperty { get; set; }
   public int IntProperty { get; set; }
}
```




Agregando `,nq` antes del corchete de cierre, se eliminan las comillas al generar una cadena.

```
[DebuggerDisplay("{StringProperty,nq} - {IntProperty}")]
```

Aunque las expresiones generales están permitidas en `{}` no se recomiendan. El atributo `DebuggerDisplay` se escribirá en los metadatos del conjunto como una cadena. Las expresiones en `{}`

no se verifican para validez. Por lo tanto, un atributo `DebuggerDisplay` contenga una lógica más compleja que, por ejemplo, alguna aritmética simple, podría funcionar bien en C #, pero la misma expresión evaluada en VB.NET probablemente no será sintácticamente válida y producirá un error al depurar.

Una forma de hacer que `DebuggerDisplay` más `DebuggerDisplay` lenguaje es escribir la expresión en un método o propiedad y llamarla en su lugar.

```
[DebuggerDisplay("{DebuggerDisplay(),nq}")]
public class AnObject
{
   public int ObjectId { get; set; }
   public string StringProperty { get; set; }
   public int IntProperty { get; set; }

   private string DebuggerDisplay()
    {
        return $"{StringProperty} - {IntProperty}"";
    }
}
```

Uno podría querer que `DebuggerDisplay` muestre todas o solo algunas de las propiedades y al depurar e inspeccionar también el tipo del objeto.
El ejemplo a continuación también rodea el método auxiliar con `#if DEBUG` ya que `DebuggerDisplay` se usa en los entornos de depuración.

```
[DebuggerDisplay("{DebuggerDisplay(),nq}")]
public class AnObject
{
   public int ObjectId { get; set; }
   public string StringProperty { get; set; }
   public int IntProperty { get; set; }

#if DEBUG
   private string DebuggerDisplay()
    {
        return
            $"ObjectId:{this.ObjectId}, StringProperty:{this.StringProperty},
Type:{this.GetType()}";
    }
    #endif
}
```

## Atributos de información del llamante

Los atributos de la información de la persona que llama se pueden usar para transmitir información sobre el invocador al método invocado. La declaración se ve así:

```
using System.Runtime.CompilerServices;

public void LogException(Exception ex,
                         [CallerMemberName]string callerMemberName = "",
                         [CallerLineNumber]int callerLineNumber = 0,
                         [CallerFilePath]string callerFilePath = "")
```

```
{
    //perform logging
}
```

Y la invocación se ve así:

```
public void Save(DBContext context)
{
    try
    {
        context.SaveChanges();
    }
    catch (Exception ex)
    {
        LogException(ex);
    }
}
```

Observe que solo el primer parámetro se pasa explícitamente al método `LogException` , mientras que el resto se proporcionará en el momento de la compilación con los valores relevantes.

El parámetro `callerMemberName` recibirá el valor `"Save"` : el nombre del método de llamada.

El parámetro `callerLineNumber` recibirá el número de la línea en la que se `LogException` llamada al método `LogException` .

Y el parámetro 'callerFilePath' recibirá la ruta completa del archivo `Save` método se declara en.

## Leyendo un atributo desde la interfaz

No hay una forma sencilla de obtener atributos de una interfaz, ya que las clases no heredan atributos de una interfaz. Siempre que implemente una interfaz o anule miembros en una clase derivada, debe volver a declarar los atributos. Entonces, en el siguiente ejemplo, la salida sería `True` en los tres casos.

```
using System;
using System.Linq;
using System.Reflection;

namespace InterfaceAttributesDemo {

    [AttributeUsage(AttributeTargets.Interface, Inherited = true)]
    class MyCustomAttribute : Attribute {
        public string Text { get; set; }
    }

    [MyCustomAttribute(Text = "Hello from interface attribute")]
    interface IMyClass {
        void MyMethod();
    }

    class MyClass : IMyClass {
        public void MyMethod() { }
    }
```

```
    public class Program {
        public static void Main(string[] args) {
            GetInterfaceAttributeDemo();
        }

        private static void GetInterfaceAttributeDemo() {
            var attribute1 = (MyCustomAttribute)
typeof(MyClass).GetCustomAttribute(typeof(MyCustomAttribute), true);
            Console.WriteLine(attribute1 == null); // True

            var attribute2 =
typeof(MyClass).GetCustomAttributes(true).OfType<MyCustomAttribute>().SingleOrDefault();
            Console.WriteLine(attribute2 == null); // True

            var attribute3 = typeof(MyClass).GetCustomAttribute<MyCustomAttribute>(true);
            Console.WriteLine(attribute3 == null); // True
        }
    }
}
```

Una forma de recuperar los atributos de la interfaz es buscarlos en todas las interfaces implementadas por una clase.

```
var attribute = typeof(MyClass).GetInterfaces().SelectMany(x =>
x.GetCustomAttributes().OfType<MyCustomAttribute>()).SingleOrDefault();
Console.WriteLine(attribute == null); // False
Console.WriteLine(attribute.Text); // Hello from interface attribute
```

## Atributo obsoleto

System.Obsolete es un atributo que se usa para marcar un tipo o un miembro que tiene una mejor versión y, por lo tanto, no se debe usar.

```
[Obsolete("This class is obsolete. Use SomeOtherClass instead.")]
class SomeClass
{
    //
}
```

En caso de que se use la clase anterior, el compilador mostrará la advertencia "Esta clase está obsoleta. Use SomeOtherClass en su lugar".

Lea Atributos en línea: https://riptutorial.com/es/csharp/topic/1062/atributos

# Capítulo 16: Biblioteca paralela de tareas

## Examples

### Paralelo.para cada

Un ejemplo que utiliza el bucle Parallel.ForEach para hacer ping a una matriz determinada de urls de sitios web.

```
static void Main()
{
    string [] urls =
    {
        "www.stackoverflow.com",
        "www.google.net",
        "www.facebook.com",
        "www.twitter.com"
    };

    System.Threading.Tasks.Parallel.ForEach(urls, url =>
    {
        var ping = new System.Net.NetworkInformation.Ping();

        var result = ping.Send(url);

        if (result.Status == System.Net.NetworkInformation.IPStatus.Success)
        {
            Console.WriteLine(string.Format("{0} is online", url));
        }
    });
}
```

### Paralelo.para

Un ejemplo que utiliza el bucle Parallel.For para hacer ping a una matriz determinada de urls de sitios web.

```
static void Main()
{
    string [] urls =
    {
        "www.stackoverflow.com",
        "www.google.net",
        "www.facebook.com",
        "www.twitter.com"
    };

    System.Threading.Tasks.Parallel.For(0, urls.Length, i =>
    {
        var ping = new System.Net.NetworkInformation.Ping();

        var result = ping.Send(urls[i]);
```

```
        if (result.Status == System.Net.NetworkInformation.IPStatus.Success)
        {
            Console.WriteLine(string.Format("{0} is online", urls[i]));
        }
    });
}
```

## Paralelo.Invocar

Invocar métodos o acciones en paralelo (región paralela)

```
static void Main()
{
    string [] urls =
    {
        "www.stackoverflow.com",
        "www.google.net",
        "www.facebook.com",
        "www.twitter.com"
    };

    System.Threading.Tasks.Parallel.Invoke(
        () => PingUrl(urls[0]),
        () => PingUrl(urls[1]),
        () => PingUrl(urls[2]),
        () => PingUrl(urls[3])
    );
}

void PingUrl(string url)
{
    var ping = new System.Net.NetworkInformation.Ping();

    var result = ping.Send(url);

    if (result.Status == System.Net.NetworkInformation.IPStatus.Success)
    {
        Console.WriteLine(string.Format("{0} is online", url));
    }
}
```

## Una tarea de sondeo cancelable asíncrono que espera entre iteraciones

```
public class Foo
{
    private const int TASK_ITERATION_DELAY_MS = 1000;
    private CancellationTokenSource _cts;

    public Foo()
    {
        this._cts = new CancellationTokenSource();
    }

    public void StartExecution()
    {
        Task.Factory.StartNew(this.OwnCodeCancelableTask_EveryNSeconds, this._cts.Token);
    }
```

```
    public void CancelExecution()
    {
        this._cts.Cancel();
    }

    /// <summary>
    /// "Infinite" loop that runs every N seconds. Good for checking for a heartbeat or
updates.
    /// </summary>
    /// <param name="taskState">The cancellation token from our _cts field, passed in the
StartNew call</param>
    private async void OwnCodeCancelableTask_EveryNSeconds(object taskState)
    {
        var token = (CancellationToken)taskState;

        while (!token.IsCancellationRequested)
        {
            Console.WriteLine("Do the work that needs to happen every N seconds in this
loop");

            // Passing token here allows the Delay to be cancelled if your task gets
cancelled.
            await Task.Delay(TASK_ITERATION_DELAY_MS, token);
        }
    }
}
```

## Una tarea de sondeo cancelable utilizando CancelaciónTokenSource

```
public class Foo
{
    private CancellationTokenSource _cts;

    public Foo()
    {
        this._cts = new CancellationTokenSource();
    }

    public void StartExecution()
    {
        Task.Factory.StartNew(this.OwnCodeCancelableTask, this._cts.Token);
    }

    public void CancelExecution()
    {
        this._cts.Cancel();
    }

    /// <summary>
    /// "Infinite" loop with no delays. Writing to a database while pulling from a buffer for
example.
    /// </summary>
    /// <param name="taskState">The cancellation token from our _cts field, passed in the
StartNew call</param>
    private void OwnCodeCancelableTask(object taskState)
    {
        var token = (CancellationToken) taskState; //Our cancellation token passed from
StartNew();
```

```
        while ( !token.IsCancellationRequested )
        {
            Console.WriteLine("Do your task work in this loop");
        }
    }
}
```

## Versión asíncrona de PingUrl

```
    static void Main(string[] args)
    {
        string url = "www.stackoverflow.com";
        var pingTask = PingUrlAsync(url);
        Console.WriteLine($"Waiting for response from {url}");
        Task.WaitAll(pingTask);
        Console.WriteLine(pingTask.Result);
    }

    static async Task<string> PingUrlAsync(string url)
    {
        string response = string.Empty;
        var ping = new System.Net.NetworkInformation.Ping();

        var result = await ping.SendPingAsync(url);

        await Task.Delay(5000); //simulate slow internet

        if (result.Status == System.Net.NetworkInformation.IPStatus.Success)
        {
            response = $"{url} is online";
        }

        return response;
    }
```

Lea Biblioteca paralela de tareas en línea: https://riptutorial.com/es/csharp/topic/1010/biblioteca-paralela-de-tareas

# Capítulo 17: BigInteger

## Observaciones

## Cuándo usar

`BigInteger` objetos `BigInteger` son, por su propia naturaleza, muy pesados en la memoria RAM. En consecuencia, solo deben usarse cuando sea absolutamente necesario, es decir, para números en una escala verdaderamente astronómica.

Además de esto, todas las operaciones aritméticas en estos objetos son un orden de magnitud más lento que sus contrapartes primitivas, este problema se complica aún más a medida que el número crece, ya que no son de un tamaño fijo. Por lo tanto, es factible que un `BigInteger` deshonesto provoque un bloqueo al consumir toda la RAM disponible.

## Alternativas

Si la velocidad es imperativa para su solución, puede ser más eficiente implementar esta funcionalidad utilizando una clase que contenga un `Byte[]` y sobrecargue los operadores necesarios. Sin embargo, esto requiere una cantidad significativa de esfuerzo extra.

## Examples

### Calcule el primer número de Fibonacci de 1,000 dígitos

Incluya `using System.Numerics` y agregue una referencia a `System.Numerics` al proyecto.

```
using System;
using System.Numerics;

namespace Euler_25
{
    class Program
    {
        static void Main(string[] args)
        {
            BigInteger l1 = 1;
            BigInteger l2 = 1;
            BigInteger current = l1 + l2;
            while (current.ToString().Length < 1000)
            {
                l2 = l1;
                l1 = current;
                current = l1 + l2;
            }
            Console.WriteLine(current);
        }
    }
}
```

Este algoritmo simple se itera a través de los números de Fibonacci hasta que alcanza una longitud de al menos 1000 dígitos decimales, y luego lo imprime. Este valor es significativamente mayor de lo que incluso un `ulong` podría mantener.

Teóricamente, el único límite en la clase `BigInteger` es la cantidad de RAM que su aplicación puede consumir.

Nota: `BigInteger` solo está disponible en .NET 4.0 y superior.

Lea BigInteger en línea: https://riptutorial.com/es/csharp/topic/5654/biginteger

# Capítulo 18: Bucle

## Examples

### Estilos de bucle

#### Mientras

El tipo de bucle más trivial. El único inconveniente es que no hay una pista intrínseca para saber dónde se encuentra en el bucle.

```
/// loop while the condition satisfies
while(condition)
{
    /// do something
}
```

#### Hacer

Similar a `while` , pero la condición se evalúa al final del bucle en lugar del principio. Esto resulta en la ejecución de los bucles al menos una vez.

```
do
{
    /// do something
} while(condition) /// loop while the condition satisfies
```

#### por

Otro estilo de bucle trivial. Mientras se hace un bucle en un índice ( `i` ) aumenta y puede usarlo. Se suele utilizar para la manipulación de matrices.

```
for ( int i = 0; i < array.Count; i++ )
{
    var currentItem = array[i];
    /// do something with "currentItem"
}
```

#### Para cada

Manera modernizada de bucle a través de objetos `IEnumarable` . Menos mal que no tienes que pensar en el índice del artículo o en el recuento de elementos de la lista.

```
foreach ( var item in someList )
{
    /// do something with "item"
}
```

#### Método Foreach

Mientras que los otros estilos se usan para seleccionar o actualizar los elementos de las colecciones, este estilo se usa generalmente para *llamar a un método de* inmediato para todos los elementos de una colección.

```
list.ForEach(item => item.DoSomething());

// or
list.ForEach(item => DoSomething(item));

// or using a method group
list.ForEach(Console.WriteLine);

// using an array
Array.ForEach(myArray, Console.WriteLine);
```

Es importante señalar que este método sólo está disponible en `List<T>` casos y también como un método estático en la `Array` - **no** es parte de LINQ.

**Linq paralelo foreach**

Al igual que Linq Foreach, excepto que este hace el trabajo de manera paralela. Lo que significa que todos los elementos de la colección ejecutarán la acción dada al mismo tiempo, simultáneamente.

```
collection.AsParallel().ForAll(item => item.DoSomething());

/// or
collection.AsParallel().ForAll(item => DoSomething(item));
```

## descanso

A veces, la condición del bucle se debe verificar en el medio del bucle. El primero es posiblemente más elegante que el segundo:

```
for (;;)
{
    // precondition code that can change the value of should_end_loop expression

    if (should_end_loop)
        break;

    // do something
}
```

Alternativa:

```
bool endLoop = false;
for (; !endLoop;)
{
    // precondition code that can set endLoop flag

    if (!endLoop)
    {
```

```
        // do something
    }
}
```

Nota: en los bucles anidados y / o el `switch` debe usar más que un simple `break` .

## Foreach Loop

foreach iterará sobre cualquier objeto de una clase que implemente `IEnumerable` (tenga en cuenta que `IEnumerable<T>` hereda de él). Dichos objetos incluyen algunos incorporados, pero no se limitan a: `List<T>` , `T[]` (arreglos de cualquier tipo), `Dictionary<TKey, TSource>` , así como interfaces como `IQueryable` e `ICollection` , etc.

**sintaxis**

```
foreach(ItemType itemVariable in enumerableObject)
    statement;
```

**observaciones**

1. El tipo `ItemType` no necesita coincidir con el tipo exacto de los elementos, solo necesita ser asignable del tipo de los elementos.
2. En lugar de `ItemType` , alternativamente se puede usar `var ItemType` el tipo de elementos del objeto enumerable al inspeccionar el argumento genérico de la implementación `IEnumerable`
3. La declaración puede ser un bloque, una sola instrucción o incluso una declaración vacía ( `;` )
4. Si `enumerableObject` no implementa `IEnumerable` , el código no se compilará
5. Durante cada iteración, el elemento actual se `ItemType` en `ItemType` (incluso si no se especifica, pero el compilador se deduce a través de `var` ) y si el elemento no se puede convertir, se lanzará una `InvalidCastException` .

Considera este ejemplo:

```
var list = new List<string>();
list.Add("Ion");
list.Add("Andrei");
foreach(var name in list)
{
    Console.WriteLine("Hello " + name);
}
```

es equivalente a:

```
var list = new List<string>();
list.Add("Ion");
list.Add("Andrei");
IEnumerator enumerator;
try
{
    enumerator = list.GetEnumerator();
    while(enumerator.MoveNext())
```

```
    {
        string name = (string)enumerator.Current;
        Console.WriteLine("Hello " + name);
    }
}
finally
{
    if (enumerator != null)
        enumerator.Dispose();
}
```

## Mientras bucle

```
int n = 0;
while (n < 5)
{
    Console.WriteLine(n);
    n++;
}
```

Salida:

0
1
2
3
4

IEnumerators se puede iterar con un bucle while:

```
// Call a custom method that takes a count, and returns an IEnumerator for a list
// of strings with the names of theh largest city metro areas.
IEnumerator<string> largestMetroAreas = GetLargestMetroAreas(4);

while (largestMetroAreas.MoveNext())
{
    Console.WriteLine(largestMetroAreas.Current);
}
```

Salida de muestra:

Tokio / Yokohama
Metro de nueva york
Sao Paulo
Seúl / Incheon

## En bucle

A For Loop es ideal para hacer cosas durante un cierto tiempo. Es como un bucle While, pero el incremento se incluye con la condición.

A For Loop se configura así:

```
for (Initialization; Condition; Increment)
{
    // Code
}
```

Inicialización: crea una nueva variable local que solo se puede utilizar en el bucle.
Condición: el bucle solo se ejecuta cuando la condición es verdadera.
Incremento: cómo cambia la variable cada vez que se ejecuta el bucle.

Un ejemplo:

```
for (int i = 0; i < 5; i++)
{
    Console.WriteLine(i);
}
```

Salida:

0
1
2
3
4

También puede dejar espacios en For Loop, pero debe tener todos los puntos y coma para que funcione.

```
int input = Console.ReadLine();

for ( ; input < 10; input + 2)
{
    Console.WriteLine(input);
}
```

Salida para 3:

3
5
7
9
11

## Do - While Loop

Es similar a un `while` bucle, excepto que prueba la condición en el *extremo* del cuerpo del bucle. El bucle Do - While ejecuta el bucle una vez, independientemente de si la condición es verdadera o no.

```
int[] numbers = new int[] { 6, 7, 8, 10 };
```

```
// Sum values from the array until we get a total that's greater than 10,
// or until we run out of values.
int sum = 0;
int i = 0;
do
{
    sum += numbers[i];
    i++;
} while (sum <= 10 && i < numbers.Length);

System.Console.WriteLine(sum); // 13
```

## Bucles anidados

```
// Print the multiplication table up to 5s
for (int i = 1; i <= 5; i++)
{
    for (int j = 1; j <= 5; j++)
    {
        int product = i * j;
        Console.WriteLine("{0} times {1} is {2}", i, j, product);
    }
}
```

## continuar

Además de `break` , también está la palabra clave `continue` . En lugar de romper por completo el bucle, simplemente saltará la iteración actual. Podría ser útil si no desea que se ejecute algún código si se establece un valor particular.

Aquí hay un ejemplo simple:

```
for (int i = 1; i <= 10; i++)
{
    if (i < 9)
        continue;

    Console.WriteLine(i);
}
```

Resultará en:

```
9
10
```

**Nota:** `Continue` es a menudo más útil en los bucles while o do-while. Los bucles for, con condiciones de salida bien definidas, pueden no beneficiarse tanto.

Lea Bucle en línea: https://riptutorial.com/es/csharp/topic/2064/bucle

# Capítulo 19: C # Script

## Examples

### Evaluación de código simple

Puede evaluar cualquier código de C # válido:

```
int value = await CSharpScript.EvaluateAsync<int>("15 * 89 + 95");
var span = await CSharpScript.EvaluateAsync<TimeSpan>("new DateTime(2016,1,1) -
DateTime.Now");
```

Si no se especifica el tipo, el resultado es `object` :

```
object value = await CSharpScript.EvaluateAsync("15 * 89 + 95");
```

Lea C # Script en línea: https://riptutorial.com/es/csharp/topic/3780/c-sharp-script

# Capítulo 20: Cadena.Formato

## Introducción

Los métodos de `Format` son un conjunto de sobrecargas en la clase `System.String` usa para crear cadenas que combinan objetos en representaciones de cadenas específicas. Esta información se puede aplicar a `String.Format` , a varios métodos de `WriteLine` así como a otros métodos en el marco de .NET.

## Sintaxis

- string.Format (formato de cadena, objeto params [] args)
- string.Format (proveedor de IFormatProvider, formato de cadena, objeto params [] args)
- $ "string {text} blablabla" // Desde C # 6

## Parámetros

| Parámetro | Detalles |
|---|---|
| formato | Una cadena de formato compuesto , que define la forma en que los *argumentos* deben combinarse en una cadena. |
| args | Una secuencia de objetos para ser combinados en una cadena. Como esto usa un argumento de `params` , puede usar una lista de argumentos separados por comas o una matriz de objetos real. |
| proveedor | Una colección de formas de formatear objetos a cadenas. Los valores típicos incluyen CultureInfo.InvariantCulture y CultureInfo.CurrentCulture . |

## Observaciones

Notas:

- `String.Format()` maneja `null` argumentos `null` sin lanzar una excepción.
- Hay sobrecargas que reemplazan el parámetro `args` con uno, dos o tres parámetros de objeto.

## Examples

### Lugares donde String.Format está 'incrustado' en el marco

Hay varios lugares donde puede usar `String.Format` *indirectamente* : el secreto es buscar la sobrecarga con el `string format, params object[] args` firma `string format, params object[] args` ,

por ejemplo:

```
Console.WriteLine(String.Format("{0} - {1}", name, value));
```

Puede ser reemplazado por una versión más corta:

```
Console.WriteLine("{0} - {1}", name, value);
```

Hay otros métodos que también utilizan `String.Format` por ejemplo:

```
Debug.WriteLine(); // and Print()
StringBuilder.AppendFormat();
```

## Usando formato de número personalizado

`NumberFormatInfo` se puede usar para formatear números enteros y flotantes.

```
// invariantResult is "1,234,567.89"
var invarianResult = string.Format(CultureInfo.InvariantCulture, "{0:#,###,##}", 1234567.89);

// NumberFormatInfo is one of classes that implement IFormatProvider
var customProvider = new NumberFormatInfo
{
    NumberDecimalSeparator = "_NS_", // will be used instead of ','
    NumberGroupSeparator = "_GS_", // will be used instead of '.'
};

// customResult is "1_GS_234_GS_567_NS_89"
var customResult = string.Format(customProvider, "{0:#,###.##}", 1234567.89);
```

## Crear un proveedor de formato personalizado

```
public class CustomFormat : IFormatProvider, ICustomFormatter
{
    public string Format(string format, object arg, IFormatProvider formatProvider)
    {
        if (!this.Equals(formatProvider))
        {
            return null;
        }

        if (format == "Reverse")
        {
            return String.Join("", arg.ToString().Reverse());
        }

        return arg.ToString();
    }

    public object GetFormat(Type formatType)
    {
        return formatType==typeof(ICustomFormatter) ? this:null;
    }
}
```

Uso:

```
String.Format(new CustomFormat(), "-> {0:Reverse} <-", "Hello World");
```

Salida:

```
-> dlroW olleH <-
```

## Alinear izquierda / derecha, pad con espacios

El segundo valor en las llaves indica la longitud de la cadena de reemplazo. Al ajustar el segundo valor para que sea positivo o negativo, se puede cambiar la alineación de la cadena.

```
string.Format("LEFT:  string: ->{0,-5}<- int: ->{1,-5}<-", "abc", 123);
string.Format("RIGHT: string: ->{0,5}<- int: ->{1,5}<-", "abc", 123);
```

Salida:

```
LEFT:  string: ->abc  <- int: ->123  <-
RIGHT: string: ->  abc<- int: ->  123<-
```

## Formatos numericos

```
// Integral types as hex
string.Format("Hexadecimal: byte2: {0:x2}; byte4: {0:X4}; char: {1:x2}", 123, (int)'A');

// Integers with thousand separators
string.Format("Integer, thousand sep.: {0:#,#}; fixed length: >{0,10:#,#}<", 1234567);

// Integer with leading zeroes
string.Format("Integer, leading zeroes: {0:00}; ", 1);

// Decimals
string.Format("Decimal, fixed precision: {0:0.000}; as percents: {0:0.00%}", 0.12);
```

Salida:

```
Hexadecimal: byte2: 7b; byte4: 007B; char: 41
Integer, thousand sep.: 1,234,567; fixed length: > 1,234,567<
Integer, leading zeroes: 01;
Decimal, fixed precision: 0.120; as percents: 12.00%
```

## Formato de moneda

El especificador de formato "c" (o moneda) convierte un número en una cadena que representa una cantidad de moneda.

```
string.Format("{0:c}", 112.236677) // $112.23 - defaults to system
```

## Precisión

El valor predeterminado es 2. Use c1, c2, c3, etc. para controlar la precisión.

```
string.Format("{0:C1}", 112.236677) //$112.2
string.Format("{0:C3}", 112.236677) //$112.237
string.Format("{0:C4}", 112.236677) //$112.2367
string.Format("{0:C9}", 112.236677) //$112.236677000
```

## Símbolo de moneda

1. Pase la instancia de `CultureInfo` para usar el símbolo de cultura personalizado

```
string.Format(new CultureInfo("en-US"), "{0:c}", 112.236677); //$112.24
string.Format(new CultureInfo("de-DE"), "{0:c}", 112.236677); //112,24 €
string.Format(new CultureInfo("hi-IN"), "{0:c}", 112.236677); //₹ 112.24
```

2. Utilice cualquier cadena como símbolo de moneda. Utilice `NumberFormatInfo` para personalizar el símbolo de moneda.

```
NumberFormatInfo nfi = new CultureInfo( "en-US", false ).NumberFormat;
nfi = (NumberFormatInfo) nfi.Clone();
nfi.CurrencySymbol = "?";
string.Format(nfi, "{0:C}", 112.236677); //?112.24
nfi.CurrencySymbol = "?%^&";
string.Format(nfi, "{0:C}", 112.236677); //?%^&112.24
```

## Posición del símbolo de moneda

Use CurrencyPositivePattern para valores positivos y CurrencyNegativePattern para valores negativos.

```
NumberFormatInfo nfi = new CultureInfo( "en-US", false ).NumberFormat;
nfi.CurrencyPositivePattern = 0;
string.Format(nfi, "{0:C}", 112.236677); //$112.24 - default
nfi.CurrencyPositivePattern = 1;
string.Format(nfi, "{0:C}", 112.236677); //112.24$
nfi.CurrencyPositivePattern = 2;
string.Format(nfi, "{0:C}", 112.236677); //$ 112.24
nfi.CurrencyPositivePattern = 3;
string.Format(nfi, "{0:C}", 112.236677); //112.24 $
```

El uso negativo del patrón es el mismo que el patrón positivo. Muchos más casos de uso, por favor consulte el enlace original.

## Separador decimal personalizado

```
NumberFormatInfo nfi = new CultureInfo( "en-US", false ).NumberFormat;
```

```
nfi.CurrencyPositivePattern = 0;
nfi.CurrencyDecimalSeparator = "..";
string.Format(nfi, "{0:C}", 112.236677); //$112..24
```

## Desde C # 6.0

6.0

Desde C # 6.0 es posible usar la interpolación de cadenas en lugar de `String.Format` .

```
string name = "John";
string lastname = "Doe";
Console.WriteLine($"Hello {name} {lastname}!");
```

Hola John Doe!

Más ejemplos para esto en el tema C # 6.0 características: Interpolación de cadenas .

## Escape de llaves dentro de una expresión String.Format ()

```
string outsidetext = "I am outside of bracket";
string.Format("{{I am in brackets!}} {0}", outsidetext);

//Outputs "{I am in brackets!} I am outside of bracket"
```

## Formato de fecha

```
DateTime date = new DateTime(2016, 07, 06, 18, 30, 14);
// Format: year, month, day hours, minutes, seconds

Console.Write(String.Format("{0:dd}",date));

//Format by Culture info
String.Format(new System.Globalization.CultureInfo("mn-MN"),"{0:dddd}",date);
```

6.0

```
Console.Write($"{date:ddd}");
```

salida:

```
06
Лхагва
06
```

| Especificador | Sentido | Muestra | Resultado |
|---|---|---|---|
| re | Fecha | `{0:d}` | 6/6/2016 |
| dd | Día, sin relleno | `{0:dd}` | 06 |

| Especificador | Sentido | Muestra | Resultado |
|---|---|---|---|
| ddd | Nombre del día corto | `{0:ddd}` | Mie |
| dddd | Nombre de dia completo | `{0:dddd}` | miércoles |
| re | Fecha larga | `{0:D}` | Miércoles 6 de julio de 2016 |
| F | Fecha y hora completas, cortas | `{0:f}` | Miércoles 6 de julio de 2016 6:30 PM |
| ff | Segundas fracciones, 2 digitos | `{0:ff}` | 20 |
| fff | Segundas fracciones, 3 dígitos | `{0:fff}` | 201 |
| ffff | Segundas fracciones, 4 digitos | `{0:ffff}` | 2016 |
| F | Fecha y hora completas, largas | `{0:F}` | Miércoles 6 de julio de 2016 6:30:14 PM |
| sol | Fecha y hora por defecto | `{0:g}` | 7/6/2016 6:30 PM |
| gg | Era | `{0:gg}` | ANUNCIO |
| S.S | Hora (2 dígitos, 12H) | `{0:hh}` | 06 |
| S.S | Hora (2 dígitos, 24H) | `{0:HH}` | 18 |
| METRO | Mes y dia | `{0:M}` | 6 de julio |
| mm | Minutos, cero rellenado | `{0:mm}` | 30 |
| MM | Mes, rellenado a cero | `{0:MM}` | 07 |
| MMM | Nombre del mes de 3 letras | `{0:MMM}` | jul |
| MMMM | Nombre del mes completo | `{0:MMMM}` | julio |
| ss | Segundos | `{0:ss}` | 14 |
| r | Fecha RFC1123 | `{0:r}` | Mié, 06 jul 2016 18:30:14 GMT |
| s | Cadena de fecha clasificable | `{0:s}` | 2016-07-06T18: 30: 14 |
| t | Poco tiempo | `{0:t}` | 6:30 PM |
| T | Largo tiempo | `{0:T}` | 6:30:14 PM |
| tt | AM PM | `{0:tt}` | PM |
| tu | Hora local seleccionable | `{0:u}` | 2016-07-06 18: 30: 14Z |

| Especificador | Sentido | Muestra | Resultado |
|---|---|---|---|
| | universal | | |
| U | GMT universal | `{0:U}` | Miércoles 6 de julio de 2016 9:30:14 AM |
| Y | Mes y año | `{0:Y}` | Julio 2016 |
| yy | Año de 2 dígitos | `{0:yy}` | dieciséis |
| aaaa | Año de 4 dígitos | `{0:yyyy}` | 2016 |
| zz | Desplazamiento de zona horaria de 2 dígitos | `{0:zz}` | +09 |
| zzz | desplazamiento de zona horaria completa | `{0:zzz}` | +09: 00 |

## Encadenar()

El método ToString () está presente en todos los tipos de objetos de referencia. Esto se debe a que todos los tipos de referencia se derivan de Object que tiene el método ToString (). El método ToString () en la clase base del objeto devuelve el nombre del tipo. El fragmento a continuación imprimirá "Usuario" en la consola.

```
public class User
{
    public string Name { get; set; }
    public int Id { get; set; }
}

...

var user = new User {Name = "User1", Id = 5};
Console.WriteLine(user.ToString());
```

Sin embargo, la clase Usuario también puede anular ToString () para alterar la cadena que devuelve. El fragmento de código a continuación imprime "Id: 5, Name: User1" en la consola.

```
public class User
{
    public string Name { get; set; }
    public int Id { get; set; }
    public override ToString()
    {
        return string.Format("Id: {0}, Name: {1}", Id, Name);
    }
}

...

var user = new User {Name = "User1", Id = 5};
```

```
Console.WriteLine(user.ToString());
```

### Relación con ToString ()

Si bien el método `String.Format()` es ciertamente útil para formatear datos como cadenas, a menudo puede ser un poco excesivo, especialmente cuando se trata de un solo objeto como se ve a continuación:

```
String.Format("{0:C}", money);  // yields "$42.00"
```

Un enfoque más fácil podría ser simplemente usar el método `ToString()` disponible en todos los objetos dentro de C #. Admite todas las mismas cadenas de formato estándar y personalizadas , pero no requiere la asignación de parámetros necesaria, ya que solo habrá un único argumento:

```
money.ToString("C");  // yields "$42.00"
```

## Advertencias y restricciones de formato

Si bien este enfoque puede ser más simple en algunos escenarios, el enfoque `ToString()` está limitado con respecto a la adición del relleno izquierdo o derecho, como lo haría con el método `String.Format()` :

```
String.Format("{0,10:C}", money);  // yields "    $42.00"
```

Para lograr este mismo comportamiento con el método `ToString()` , necesitarías usar otro método como `PadLeft()` o `PadRight()` respectivamente:

```
money.ToString("C").PadLeft(10);  // yields "    $42.00"
```

Lea Cadena.Formato en línea: https://riptutorial.com/es/csharp/topic/79/cadena-formato

# Capítulo 21: Características de C # 3.0

## Observaciones

La versión 3.0 de C # se lanzó como parte de la versión 3.5 de .Net. Muchas de las funciones agregadas con esta versión eran compatibles con LINQ (Language INtegrated Queries).

Lista de características añadidas:

- LINQ
- Expresiones lambda
- Metodos de extension
- Tipos anónimos
- Variables implícitamente tipificadas
- Inicializadores de objetos y colecciones
- Propiedades implementadas automáticamente
- Arboles de expresion

## Examples

### Variables implícitamente escritas (var)

La palabra clave `var` permite que un programador escriba implícitamente una variable en el momento de la compilación. `var` declaraciones `var` tienen el mismo tipo que las variables declaradas explícitamente.

```
var squaredNumber = 10 * 10;
var squaredNumberDouble = 10.0 * 10.0;
var builder = new StringBuilder();
var anonymousObject = new
{
    One = SquaredNumber,
    Two = SquaredNumberDouble,
    Three = Builder
}
```

Los tipos de las variables anteriores son `int` , `double` , `StringBuilder` y un tipo anónimo, respectivamente.

Es importante tener en cuenta que una variable `var` no se escribe dinámicamente. `SquaredNumber = Builder` no es válido porque está intentando establecer un `int` en una instancia de `StringBuilder`

### Consultas integradas de idiomas (LINQ)

```
//Example 1
int[] array = { 1, 5, 2, 10, 7 };
```

```
// Select squares of all odd numbers in the array sorted in descending order
IEnumerable<int> query = from x in array
                         where x % 2 == 1
                         orderby x descending
                         select x * x;
// Result: 49, 25, 1
```

### Ejemplo del artículo de wikipedia sobre la sub-sección LINQ de C # 3.0

El ejemplo 1 usa una sintaxis de consulta que fue diseñada para verse similar a las consultas SQL.

```
//Example 2
IEnumerable<int> query = array.Where(x => x % 2 == 1)
    .OrderByDescending(x => x)
    .Select(x => x * x);
// Result: 49, 25, 1 using 'array' as defined in previous example
```

### Ejemplo del artículo de wikipedia sobre la sub-sección LINQ de C # 3.0

El ejemplo 2 usa la sintaxis del método para lograr el mismo resultado que el ejemplo 1.

Es importante tener en cuenta que, en C #, la sintaxis de consulta LINQ es azúcar sintáctica para la sintaxis del método LINQ. El compilador traduce las consultas en llamadas de método en tiempo de compilación. Algunas consultas deben expresarse en la sintaxis del método. De MSDN : "Por ejemplo, debe usar una llamada de método para expresar una consulta que recupera el número de elementos que coinciden con una condición específica".

## Expresiones lambda

Lambda Expresions es una extensión de métodos anónimos que permiten parámetros tipificados implícitamente y valores de retorno. Su sintaxis es menos detallada que los métodos anónimos y sigue un estilo de programación funcional.

```
using System;
using System.Collections.Generic;
using System.Linq;

public class Program
{
    public static void Main()
    {
        var numberList = new List<int> {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
        var sumOfSquares = numberList.Select( number => number * number )
            .Aggregate( (int first, int second) => { return first + second; } );
        Console.WriteLine( sumOfSquares );
    }
}
```

El código anterior generará la suma de los cuadrados de los números del 1 al 10 en la consola.

La primera expresión lambda cuadra los números de la lista. Dado que solo hay 1 parámetro, el paréntesis puede omitirse. Puede incluir paréntesis si lo desea:

```
.Select( (number) => number * number);
```

o escriba explícitamente el parámetro pero luego se requieren paréntesis:

```
.Select( (int number) => number * number);
```

El cuerpo lambda es una expresión y tiene un retorno implícito. Puedes usar un cuerpo de declaración si lo deseas también. Esto es útil para las lambdas más complejas.

```
.Select( number => { return number * number; } );
```

El método de selección devuelve un nuevo IEnumerable con los valores calculados.

La segunda expresión lambda suma los números en la lista devueltos por el método de selección. Se requieren paréntesis ya que hay múltiples parámetros. Los tipos de parámetros se escriben explícitamente, pero esto no es necesario. El siguiente método es equivalente.

```
.Aggregate( (first, second) => { return first + second; } );
```

Como es este:

```
.Aggregate( (int first, int second) => first + second );
```

## Tipos anónimos

Los tipos anónimos proporcionan una manera conveniente de encapsular un conjunto de propiedades de solo lectura en un solo objeto sin tener que definir explícitamente un tipo primero. El nombre del tipo lo genera el compilador y no está disponible en el nivel del código fuente. El tipo de cada propiedad es inferido por el compilador.

Puede hacer tipos anónimos utilizando la `new` palabra clave seguida de una llave *( `{` )* . Dentro de las llaves, puede definir propiedades como en el código a continuación.

```
var v = new { Amount = 108, Message = "Hello" };
```

También es posible crear una matriz de tipos anónimos. Ver código a continuación:

```
var a = new[] {
    new {
        Fruit = "Apple",
        Color = "Red"
    },
    new {
        Fruit = "Banana",
        Color = "Yellow"
    }
};
```

O usarlo con las consultas LINQ:

```
var productQuery = from prod in products
                   select new { prod.Color, prod.Price };
```

Lea Características de C # 3.0 en línea:
https://riptutorial.com/es/csharp/topic/3820/caracteristicas-de-c-sharp-3-0

# Capítulo 22: Características de C # 4.0

## Examples

### Parámetros opcionales y argumentos nombrados

Podemos omitir el argumento en la llamada si ese argumento es un argumento opcional. Cada argumento opcional tiene su propio valor predeterminado. Tomará el valor predeterminado si no proporcionamos el valor. El valor predeterminado de un argumento opcional debe ser un

1. Expresión constante.
2. Debe ser un tipo de valor como enum o struct.
3. Debe ser una expresión del formulario por defecto (valueType)

Debe establecerse al final de la lista de parámetros

Parámetros del método con valores por defecto:

```
public void ExampleMethod(int required, string optValue = "test", int optNum = 42)
{
    //...
}
```

Según lo dicho por MSDN, un argumento con nombre,

Le permite pasar el argumento a la función al asociar el nombre del parámetro. No es necesario recordar la posición de los parámetros que no conocemos siempre. No es necesario mirar el orden de los parámetros en la lista de parámetros de la función llamada. Podemos especificar el parámetro para cada argumento por su nombre.

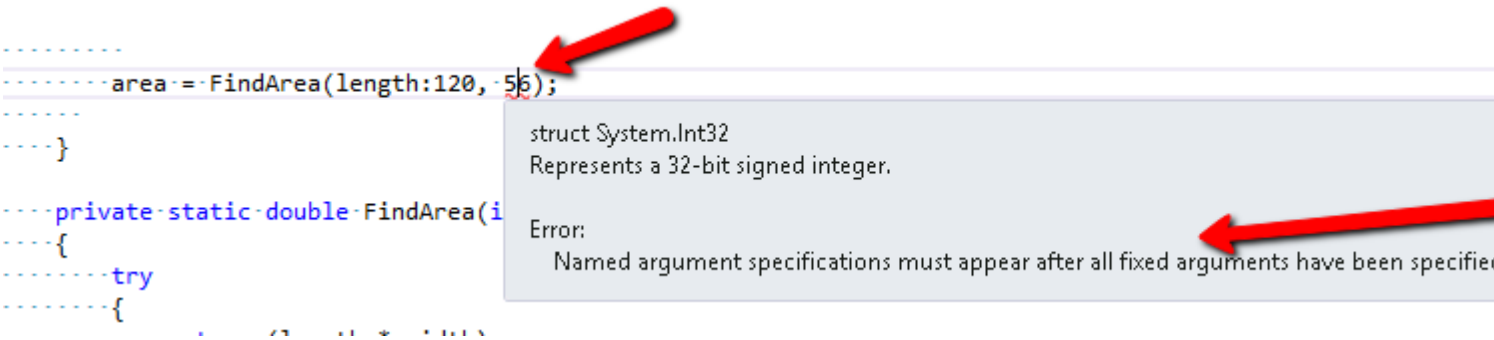
Argumentos con nombre:

```
// required = 3, optValue = "test", optNum = 4
ExampleMethod(3, optNum: 4);
// required = 2, optValue = "foo", optNum = 42
ExampleMethod(2, optValue: "foo");
// required = 6, optValue = "bar", optNum = 1
ExampleMethod(optNum: 1, optValue: "bar", required: 6);
```

**Limitación de usar un argumento con nombre**

La especificación de argumento con nombre debe aparecer después de que se hayan especificado todos los argumentos fijos.

Si usa un argumento con nombre antes de un argumento fijo, obtendrá un error de tiempo de compilación de la siguiente manera.




La especificación del argumento con nombre debe aparecer después de que se hayan especificado todos los argumentos fijos

## Diferencia

Las interfaces genéricas y los delegados pueden tener sus parámetros de tipo marcados como *covariantes* o *contravariantes* usando las palabras clave de `out` y `in` palabras clave respectivamente. Estas declaraciones se respetan para las conversiones de tipo, tanto implícitas como explícitas, y tanto el tiempo de compilación como el tiempo de ejecución.

Por ejemplo, la interfaz existente `IEnumerable<T>` se ha redefinido como covariante:

```
interface IEnumerable<out T>
{
    IEnumerator<T> GetEnumerator();
}
```

La interfaz existente IComparer se ha redefinido como contravariante:

```
public interface IComparer<in T>
{
    int Compare(T x, T y);
}
```

## Palabra clave de referencia opcional al utilizar COM

La palabra clave ref para quienes llaman a los métodos ahora es opcional al llamar a los métodos proporcionados por las interfaces COM. Dado un método COM con la firma

```
void Increment(ref int x);
```

La invocación ahora se puede escribir como cualquiera

```
Increment(0); // no need for "ref" or a place holder variable any more
```

## Búsqueda dinámica de miembros

Se introduce una nueva `dynamic` pseudo tipo en el sistema de tipo C #. Se trata como `System.Object` , pero además, se permite el acceso de cualquier miembro (llamada de método,

campo, propiedad o acceso de indexador, o una invocación de delegado) o la aplicación de un operador a un valor de tal tipo sin ningún tipo de verificación de tipo, y su resolución se pospone hasta el tiempo de ejecución. Esto se conoce como tipificación de pato o encuadernación tardía. Por ejemplo:

```
// Returns the value of Length property or field of any object
int GetLength(dynamic obj)
{
    return obj.Length;
}

GetLength("Hello, world");        // a string has a Length property,
GetLength(new int[] { 1, 2, 3 }); // and so does an array,
GetLength(42);                    // but not an integer - an exception will be thrown
                                  // in GetLength method at run-time
```

En este caso, el tipo dinámico se utiliza para evitar una Reflexión más detallada. Todavía usa Reflection bajo el capó, pero generalmente es más rápido gracias al almacenamiento en caché.

Esta función está dirigida principalmente a la interoperabilidad con lenguajes dinámicos.

```
// Initialize the engine and execute a file
var runtime = ScriptRuntime.CreateFromConfiguration();
dynamic globals = runtime.Globals;
runtime.ExecuteFile("Calc.rb");

// Use Calc type from Ruby
dynamic calc = globals.Calc.@new();
calc.valueA = 1337;
calc.valueB = 666;
dynamic answer = calc.Calculate();
```

El tipo dinámico tiene aplicaciones incluso en el código de tipo estático en su mayoría, por ejemplo, hace posible el doble envío sin implementar el patrón de visitante.

Lea Características de C # 4.0 en línea:
https://riptutorial.com/es/csharp/topic/3093/caracteristicas-de-c-sharp-4-0

# Capítulo 23: Características de C # 5.0

## Sintaxis

- ***Async y espera***
- **Tarea** pública MyTask **Async** () {doSomething (); }

  aguarda MyTaskAsync ();
- public **Task** `<string>` MyStringTask **Async** () {return getSomeString (); }

  cadena MyString = aguarda MyStringTaskAsync ();
- ***Atributos de información del llamante***
- public void MyCallerAttributes (cadena MyMessage,

  [CallerMemberName] string MemberName = "",

  [CallerFilePath] cadena SourceFilePath = "",

  [CallerLineNumber] int LineNumber = 0)
- Trace.WriteLine ("Mi mensaje:" + MiMensaje);

  Trace.WriteLine ("Member:" + MemberName);

  Trace.WriteLine ("Ruta del archivo de origen:" + SourceFilePath);

  Trace.WriteLine ("Número de línea:" + Número de línea);

## Parámetros

| Método / Modificador con Parámetro | Detalles |
|---|---|
| `Type<T>` | `T` es el tipo de retorno |

## Observaciones

C # 5.0 está acoplado con Visual Studio .NET 2012

## Examples

### Async y espera

`async` y `await` son dos operadores que intentan mejorar el rendimiento al liberar Threads y esperar a que se completen las operaciones antes de seguir adelante.

Aquí hay un ejemplo de cómo obtener una cadena antes de devolver su longitud:

```
//This method is async because:
//1. It has async and Task or Task<T> as modifiers
//2. It ends in "Async"
async Task<int> GetStringLengthAsync(string URL){
    HttpClient client = new HttpClient();
    //Sends a GET request and returns the response body as a string
    Task<string> getString = client.GetStringAsync(URL);
    //Waits for getString to complete before returning its length
    string contents = await getString;
    return contents.Length;
}

private async void doProcess(){
    int length = await GetStringLengthAsync("http://example.com/");
    //Waits for all the above to finish before printing the number
    Console.WriteLine(length);
}
```

Este es otro ejemplo de la descarga de un archivo y el manejo de lo que sucede cuando su progreso ha cambiado y cuando se completa la descarga (hay dos formas de hacerlo):

Método 1:

```
//This one using async event handlers, but not async coupled with await
private void DownloadAndUpdateAsync(string uri, string DownloadLocation){
    WebClient web = new WebClient();
    //Assign the event handler
    web.DownloadProgressChanged += new DownloadProgressChangedEventHandler(ProgressChanged);
    web.DownloadFileCompleted += new AsyncCompletedEventHandler(FileCompleted);
    //Download the file asynchronously
    web.DownloadFileAsync(new Uri(uri), DownloadLocation);
}

//event called for when download progress has changed
private void ProgressChanged(object sender, DownloadProgressChangedEventArgs e){
    //example code
    int i = 0;
    i++;
    doSomething();
}

//event called for when download has finished
private void FileCompleted(object sender, AsyncCompletedEventArgs e){
    Console.WriteLine("Completed!")
}
```

Método 2:

```
//however, this one does
//Refer to first example on why this method is async
private void DownloadAndUpdateAsync(string uri, string DownloadLocation){
    WebClient web = new WebClient();
```

```
    //Assign the event handler
    web.DownloadProgressChanged += new DownloadProgressChangedEventHandler(ProgressChanged);
    //Download the file async
    web.DownloadFileAsync(new Uri(uri), DownloadLocation);
    //Notice how there is no complete event, instead we're using techniques from the first
example
}
private void ProgressChanged(object sender, DownloadProgressChangedEventArgs e){
    int i = 0;
    i++;
    doSomething();
}
private void doProcess(){
    //Wait for the download to finish
    await DownloadAndUpdateAsync(new Uri("http://example.com/file"))
    doSomething();
}
```

## Atributos de información del llamante

Las CIA están pensadas como una forma simple de obtener atributos de lo que sea que esté llamando el método objetivo. Realmente solo hay 1 forma de usarlos y solo hay 3 atributos.

Ejemplo:

```
//This is the "calling method": the method that is calling the target method
public void doProcess()
{
    GetMessageCallerAttributes("Show my attributes.");
}
//This is the target method
//There are only 3 caller attributes
public void GetMessageCallerAttributes(string message,
    //gets the name of what is calling this method
    [System.Runtime.CompilerServices.CallerMemberName] string memberName = "",
    //gets the path of the file in which the "calling method" is in
    [System.Runtime.CompilerServices.CallerFilePath] string sourceFilePath = "",
    //gets the line number of the "calling method"
    [System.Runtime.CompilerServices.CallerLineNumber] int sourceLineNumber = 0)
{
    //Writes lines of all the attributes
    System.Diagnostics.Trace.WriteLine("Message: " + message);
    System.Diagnostics.Trace.WriteLine("Member: " + memberName);
    System.Diagnostics.Trace.WriteLine("Source File Path: " + sourceFilePath);
    System.Diagnostics.Trace.WriteLine("Line Number: " + sourceLineNumber);
}
```

Ejemplo de salida:

```
//Message: Show my attributes.
//Member: doProcess
//Source File Path: c:\Path\To\The\File
//Line Number: 13
```

Lea Características de C # 5.0 en línea:
https://riptutorial.com/es/csharp/topic/4584/caracteristicas-de-c-sharp-5-0

# Capítulo 24: Características de C # 6.0

## Introducción

Esta sexta iteración del lenguaje C # es proporcionada por el compilador de Roslyn. Este compilador salió con la versión 4.6 de .NET Framework, sin embargo, puede generar código de una manera compatible con versiones anteriores para permitir el destino de versiones anteriores del marco. El código de la versión 6 de C # se puede compilar de una manera totalmente compatible con .NET 4.0. También se puede usar para marcos anteriores, sin embargo, algunas características que requieren un soporte de marco adicional pueden no funcionar correctamente.

## Observaciones

La sexta versión de C # se lanzó en julio de 2015 junto con Visual Studio 2015 y .NET 4.6.

Además de agregar algunas características nuevas de lenguaje, incluye una reescritura completa del compilador. Anteriormente, `csc.exe` era una aplicación Win32 nativa escrita en C ++, con C # 6 ahora es una aplicación administrada .NET escrita en C #. Esta reescritura se conoció como proyecto "Roslyn" y el código ahora es de código abierto y está disponible en GitHub .

## Examples

### Nombre del operador

El operador `nameof` devuelve el nombre de un elemento de código como una `string` . Esto es útil cuando se lanzan excepciones relacionadas con los argumentos de los métodos y también cuando se implementa `INotifyPropertyChanged` .

```
public string SayHello(string greeted)
{
    if (greeted == null)
        throw new ArgumentNullException(nameof(greeted));

    Console.WriteLine("Hello, " + greeted);
}
```

El operador `nameof` se evalúa en el momento de la compilación y cambia la expresión a una cadena literal. Esto también es útil para las cadenas que llevan el nombre de su miembro que las expone. Considera lo siguiente:

```
public static class Strings
{
    public const string Foo = nameof(Foo); // Rather than Foo = "Foo"
    public const string Bar = nameof(Bar); // Rather than Bar = "Bar"
}
```

Dado que `nameof` expresiones `nameof` expresiones son constantes en tiempo de compilación, se pueden usar en atributos, etiquetas de `case` , declaraciones de `switch` , etc.

---

Es conveniente usar `nameof` con `Enum` s. En lugar de:

```
Console.WriteLine(Enum.One.ToString());
```

Es posible utilizar:

```
Console.WriteLine(nameof(Enum.One))
```

La salida será `One` en ambos casos.

---

El operador `nameof` puede acceder a miembros no estáticos utilizando una sintaxis similar a la estática. En lugar de hacer:

```
string foo = "Foo";
string lengthName = nameof(foo.Length);
```

Puede ser reemplazado por:

```
string lengthName = nameof(string.Length);
```

La salida será `Length` en ambos ejemplos. Sin embargo, este último impide la creación de instancias innecesarias.

---

Aunque el operador `nameof` funciona con la mayoría de las construcciones de lenguaje, existen algunas limitaciones. Por ejemplo, no puede usar el operador `nameof` en tipos genéricos abiertos o valores de retorno de método:

```
public static int Main()
{
    Console.WriteLine(nameof(List<>)); // Compile-time error
    Console.WriteLine(nameof(Main())); // Compile-time error
}
```

Además, si lo aplica a un tipo genérico, el parámetro de tipo genérico se ignorará:

```
Console.WriteLine(nameof(List<int>));  // "List"
Console.WriteLine(nameof(List<bool>)); // "List"
```

Para más ejemplos, vea este tema dedicado a `nameof` .

---

# Solución para versiones anteriores ( más

# detalles )

Aunque el operador `nameof` no existe en C # para las versiones anteriores a 6.0, se puede tener una funcionalidad similar utilizando `MemberExpression` como se `MemberExpression` a continuación:

6.0

Expresión:

```
public static string NameOf<T>(Expression<Func<T>> propExp)
{
    var memberExpression = propExp.Body as MemberExpression;
    return memberExpression != null ? memberExpression.Member.Name : null;
}

public static string NameOf<TObj, T>(Expression<Func<TObj, T>> propExp)
{
    var memberExpression = propExp.Body as MemberExpression;
    return memberExpression != null ? memberExpression.Member.Name : null;
}
```

Uso:

```
string variableName = NameOf(() => variable);
string propertyName = NameOf((Foo o) => o.Bar);
```

Tenga en cuenta que este enfoque hace que se cree un árbol de expresiones en cada llamada, por lo que el rendimiento es mucho peor en comparación con el `nameof` operador, que se evalúa en el momento de la compilación y no tiene sobrecarga en el tiempo de ejecución.

### Miembros de la función de cuerpo expresivo

Los miembros de la función con cuerpo de expresión permiten el uso de expresiones lambda como cuerpos miembros. Para miembros simples, puede resultar en un código más limpio y más legible.

Las funciones con cuerpo de expresión se pueden usar para propiedades, indizadores, métodos y operadores.

---

# Propiedades

```
public decimal TotalPrice => BasePrice + Taxes;
```

Es equivalente a:

```
public decimal TotalPrice
{
```

```
    get
    {
        return BasePrice + Taxes;
    }
}
```

Cuando se utiliza una función de expresión con una propiedad, la propiedad se implementa como una propiedad solo para el captador.

Ver demostración

---

# Indexadores

```
public object this[string key] => dictionary[key];
```

Es equivalente a:

```
public object this[string key]
{
    get
    {
        return dictionary[key];
    }
}
```

---

# Métodos

```
static int Multiply(int a, int b) => a * b;
```

Es equivalente a:

```
static int Multiply(int a, int b)
{
    return a * b;
}
```

Que también se puede utilizar con métodos de `void` :

```
public void Dispose() => resource?.Dispose();
```

Se podría agregar una anulación de `ToString` a la clase `Pair<T>` :

```
public override string ToString() => $"{First}, {Second}";
```

Además, este enfoque simplista funciona con la palabra clave de `override` :

```
public class Foo
{
    public int Bar { get; }

    public string override ToString() => $"Bar: {Bar}";
}
```

---

# Los operadores

Esto también puede ser utilizado por los operadores:

```
public class Land
{
    public double Area { get; set; }

    public static Land operator +(Land first, Land second) =>
        new Land { Area = first.Area + second.Area };
}
```

---

# Limitaciones

Los miembros de la función con cuerpo de expresión tienen algunas limitaciones. No pueden contener declaraciones de bloque y cualquier otra declaración que contenga bloques: `if` , `switch` , `for` , `foreach` , `while` , `do` , `try` , etc.

Algunas `if` declaraciones pueden ser reemplazadas por operadores ternarios. Algunas declaraciones `for` y `foreach` se pueden convertir en consultas LINQ, por ejemplo:

```
IEnumerable<string> Digits
{
    get
    {
        for (int i = 0; i < 10; i++)
            yield return i.ToString();
    }
}
```

```
IEnumerable<string> Digits => Enumerable.Range(0, 10).Select(i => i.ToString());
```

En todos los demás casos, se puede usar la sintaxis antigua para miembros de funciones.

Los miembros de la función con cuerpo de expresión pueden contener `async` / `await` , pero a menudo es redundante:

```
async Task<int> Foo() => await Bar();
```

Puede ser reemplazado por:

```
Task<int> Foo() => Bar();
```

### Filtros de excepción

Los filtros de excepción les dan a los desarrolladores la capacidad de agregar una condición (en forma de expresión `boolean` ) a un bloque catch , permitiendo que la `catch` ejecute solo si la condición se evalúa como `true` .

Los filtros de excepción permiten la propagación de información de depuración en la excepción original, mientras que al usar una instrucción `if` dentro de un bloque `catch` y volver a lanzar la excepción, se detiene la propagación de información de depuración en la excepción original. Con los filtros de excepción, la excepción continúa propagándose hacia arriba en la pila de llamadas a *menos* que se cumpla la condición. Como resultado, los filtros de excepción hacen que la experiencia de depuración sea mucho más fácil. En lugar de detenerse en la declaración de `throw` , el depurador se detendrá en la instrucción de lanzar la excepción, con el estado actual y todas las variables locales conservadas. Los vertederos se ven afectados de manera similar.

> Los filtros de excepción han sido admitidos por el **CLR** desde el principio y han sido accesibles desde VB.NET y F # durante más de una década al exponer una parte del modelo de manejo de excepciones del CLR. Solo después del lanzamiento de C # 6.0, la funcionalidad también estuvo disponible para los desarrolladores de C #.

---

## Usando filtros de excepción

Los filtros de excepción se utilizan agregando una cláusula `when` a la expresión `catch` . Es posible usar cualquier expresión que devuelva un `bool` en una cláusula `when` (excepto en espera ). La variable de excepción declarada `ex` es accesible desde dentro de la cláusula `when` :

```
var SqlErrorToIgnore = 123;
try
{
    DoSQLOperations();
}
catch (SqlException ex) when (ex.Number != SqlErrorToIgnore)
{
    throw new Exception("An error occurred accessing the database", ex);
}
```

Se pueden combinar múltiples bloques `catch` con las cláusulas `when` . La primera `when` cláusula devuelva `true` provocará que se detecte la excepción. Se ingresará su bloque `catch` , mientras que las otras cláusulas `catch` se ignorarán (no se evaluarán sus cláusulas `when` ). Por ejemplo:

```
try
{ ... }
catch (Exception ex) when (someCondition) //If someCondition evaluates to true,
                                          //the rest of the catches are ignored.
{ ... }
catch (NotImplementedException ex) when (someMethod()) //someMethod() will only run if
                                                       //someCondition evaluates to false
```

```
{ ... }
catch(Exception ex) // If both when clauses evaluate to false
{ ... }
```

---

# Arriesgado cuando la cláusula

> **Precaución**
>
> Puede ser arriesgado usar filtros de excepción: cuando se lanza una `Exception` desde dentro de la cláusula `when` , la `Exception` de la cláusula `when` se ignora y se trata como `false` . Este enfoque permite a los desarrolladores escribir `when` cláusula sin tener en cuenta los casos no válidos.

El siguiente ejemplo ilustra tal escenario:

```
public static void Main()
{
    int a = 7;
    int b = 0;
    try
    {
        DoSomethingThatMightFail();
    }
    catch (Exception ex) when (a / b == 0)
    {
        // This block is never reached because a / b throws an ignored
        // DivideByZeroException which is treated as false.
    }
    catch (Exception ex)
    {
        // This block is reached since the DivideByZeroException in the
        // previous when clause is ignored.
    }
}

public static void DoSomethingThatMightFail()
{
    // This will always throw an ArgumentNullException.
    Type.GetType(null);
}
```

Ver demostración

Tenga en cuenta que los filtros de excepción evitan los confusos problemas de número de línea asociados con el uso de `throw` cuando el código que falla está dentro de la misma función. Por ejemplo, en este caso, el número de línea se reporta como 6 en lugar de 3:

```
1. int a = 0, b = 0;
2. try {
3.     int c = a / b;
4. }
5. catch (DivideByZeroException) {
6.     throw;
7. }
```

El número de línea de excepción se informa como 6 porque el error se detectó y se volvió a lanzar con la declaración de `throw` en la línea 6.

Lo mismo no ocurre con los filtros de excepción:

```
1. int a = 0, b = 0;
2. try {
3.     int c = a / b;
4. }
5. catch (DivideByZeroException) when (a != 0) {
6.     throw;
7. }
```

En este ejemplo `a` es 0, la cláusula `catch` se ignora, pero 3 se reporta como número de línea. Esto se debe a que **no desenrollan la pila** . Más específicamente, la excepción *no se detecta* en la línea 5, porque `a` de hecho es igual a `0` y por lo tanto no hay oportunidad para la excepción que ser re-lanzado en la línea 6, porque la línea 6 no se ejecuta.

---

## La tala como efecto secundario

Las llamadas de método en la condición pueden causar efectos secundarios, por lo que los filtros de excepción se pueden usar para ejecutar código en excepciones sin detectarlos. Un ejemplo común que aprovecha esto es un método de `Log` que siempre devuelve `false` . Esto permite rastrear la información de registro mientras se realiza la depuración sin la necesidad de volver a lanzar la excepción.

> **Tenga en cuenta que** si bien esto parece ser una forma cómoda de registro, puede ser riesgoso, especialmente si se utilizan ensamblajes de registro de terceros. Estos pueden generar excepciones al iniciar sesión en situaciones no obvias que pueden no detectarse fácilmente (consulte **Risky `when(...)` cláusula** anterior).

```
try
{
    DoSomethingThatMightFail(s);
}
catch (Exception ex) when (Log(ex, "An error occurred"))
{
    // This catch block will never be reached
}

// ...

static bool Log(Exception ex, string message, params object[] args)
{
    Debug.Print(message, args);
    return false;
}
```

Ver demostración

El enfoque común en versiones anteriores de C # era registrar y volver a lanzar la excepción.

6.0

```
try
{
    DoSomethingThatMightFail(s);
}
catch (Exception ex)
{
     Log(ex, "An error occurred");
     throw;
}

// ...

static void Log(Exception ex, string message, params object[] args)
{
    Debug.Print(message, args);
}
```

Ver demostración

---

# El bloque `finally`

El bloque `finally` se ejecuta cada vez si se lanza la excepción o no. Una sutileza con expresiones en `when` se ejecutan los filtros de excepción más arriba en la pila *antes de* ingresar a los bloques internos por `finally` . Esto puede provocar que los resultados y comportamientos inesperados cuando el código intenta modificar el estado global (como usuario o la cultura del hilo actual) y poner de nuevo en un `finally` de bloquear.

## Ejemplo: `finally` bloque

```
private static bool Flag = false;

static void Main(string[] args)
{
    Console.WriteLine("Start");
    try
    {
        SomeOperation();
    }
    catch (Exception) when (EvaluatesTo())
    {
        Console.WriteLine("Catch");
    }
    finally
    {
        Console.WriteLine("Outer Finally");
    }
}

private static bool EvaluatesTo()
{
    Console.WriteLine($"EvaluatesTo: {Flag}");
    return true;
```

```
}

private static void SomeOperation()
{
    try
    {
        Flag = true;
        throw new Exception("Boom");
    }
    finally
    {
        Flag = false;
        Console.WriteLine("Inner Finally");
    }
}
```

Salida producida:

comienzo
EvaluatesTo: True
Interior finalmente
Captura
Exterior finalmente

Ver demostración

En el ejemplo anterior, si el método `SomeOperation` no desea a "fugas" de los cambios de estado globales a quien haya llamado `when` cláusulas, sino que también deben contener un `catch` bloque para modificar el estado. Por ejemplo:

```
private static void SomeOperation()
{
    try
    {
        Flag = true;
        throw new Exception("Boom");
    }
    catch
    {
       Flag = false;
       throw;
    }
    finally
    {
        Flag = false;
        Console.WriteLine("Inner Finally");
    }
}
```

También es común ver clases de ayuda `IDisposable` aprovechan la semántica del uso de bloques para lograr el mismo objetivo, ya que `IDisposable.Dispose` siempre se llamará antes de que una excepción llamada dentro de un bloque `using` comience a burbujear la pila.

## Inicializadores de propiedad automática

# Introducción

Las propiedades pueden inicializarse con el operador `=` después del cierre `}` . La clase de `Coordinate` continuación muestra las opciones disponibles para inicializar una propiedad:

6.0

```
public class Coordinate
{
    public int X { get; set; } = 34; // get or set auto-property with initializer

    public int Y { get; } = 89;      // read-only auto-property with initializer
}
```

---

## Accesorios con visibilidad diferente

Puede inicializar propiedades automáticas que tienen visibilidad diferente en sus accesores. Aquí hay un ejemplo con un setter protegido:

```
    public string Name { get; protected set; } = "Cheeze";
```

El accesorio también puede ser `internal` , `internal protected` o `private` .

---

## Propiedades de solo lectura

Además de la flexibilidad con la visibilidad, también puede inicializar propiedades automáticas de solo lectura. Aquí hay un ejemplo:

```
    public List<string> Ingredients { get; } =
        new List<string> { "dough", "sauce", "cheese" };
```

Este ejemplo también muestra cómo inicializar una propiedad con un tipo complejo. Además, las propiedades automáticas no pueden ser de solo escritura, por lo que también impide la inicialización de solo escritura.

---

# Estilo antiguo (pre C # 6.0)

Antes de C # 6, esto requería un código mucho más detallado. Estábamos usando una variable adicional llamada propiedad de respaldo de la propiedad para dar un valor predeterminado o para inicializar la propiedad pública como se muestra a continuación,

6.0

```
public class Coordinate
{
    private int _x = 34;
    public int X { get { return _x; } set { _x = value; } }

    private readonly int _y = 89;
    public int Y { get { return _y; } }

    private readonly int _z;
    public int Z { get { return _z; } }

    public Coordinate()
    {
        _z = 42;
    }
}
```

***Nota:*** *Antes de C # 6.0, aún podría inicializar* ***las propiedades de*** *lectura y escritura* ***implementadas automáticamente*** *(propiedades con un captador y un definidor) desde el constructor, pero no pudo inicializar la propiedad en línea con su declaración*

Ver demostración

---

# Uso

Los inicializadores deben evaluar las expresiones estáticas, al igual que los inicializadores de campo. Si necesita hacer referencia a miembros no estáticos, puede inicializar propiedades en constructores como antes o usar propiedades con expresión. Las expresiones no estáticas, como la de abajo (comentadas), generarán un error de compilación:

```
// public decimal X { get; set; } = InitMe();  // generates compiler error

decimal InitMe() { return 4m; }
```

Pero los métodos estáticos **se** pueden utilizar para inicializar propiedades automáticas:

```
public class Rectangle
{
    public double Length { get; set; } = 1;
    public double Width { get; set; } = 1;
    public double Area { get; set; } = CalculateArea(1, 1);

    public static double CalculateArea(double length, double width)
    {
        return length * width;
    }
}
```

Este método también se puede aplicar a propiedades con diferentes niveles de accesores:

```
public short Type { get; private set; } = 15;
```

El inicializador automático de propiedades permite la asignación de propiedades directamente dentro de su declaración. Para propiedades de solo lectura, se ocupa de todos los requisitos necesarios para garantizar que la propiedad sea inmutable. Considere, por ejemplo, la clase `FingerPrint` en el siguiente ejemplo:

```
public class FingerPrint
{
  public DateTime TimeStamp { get; } = DateTime.UtcNow;

  public string User { get; } =
    System.Security.Principal.WindowsPrincipal.Current.Identity.Name;

  public string Process { get; } =
    System.Diagnostics.Process.GetCurrentProcess().ProcessName;
}
```

Ver demostración

---

# Notas de precaución

Tenga cuidado de no confundir las propiedades automáticas o los inicializadores de campo con métodos de expresión similar que usan `=>` en lugar de `=` , y los campos que no incluyen `{ get; }` .

Por ejemplo, cada una de las siguientes declaraciones son diferentes.

```
public class UserGroupDto
{
    // Read-only auto-property with initializer:
    public ICollection<UserDto> Users1 { get; } = new HashSet<UserDto>();

    // Read-write field with initializer:
    public ICollection<UserDto> Users2 = new HashSet<UserDto>();

    // Read-only auto-property with expression body:
    public ICollection<UserDto> Users3 => new HashSet<UserDto>();
}
```

Falta `{ get; }` en los resultados de la declaración de propiedad en un campo público. Tanto la propiedad automática de solo lectura `Users1` como el campo de lectura-escritura `Users2` se inicializan solo una vez, pero un campo público permite cambiar la instancia de colección desde fuera de la clase, lo que generalmente no es deseable. Cambiar una propiedad automática de solo lectura con cuerpo de expresión a propiedad de solo lectura con inicializador requiere no solo eliminar `>` from `=>` , sino agregar `{ get; }` .

El símbolo diferente ( `=>` lugar de `=` ) en `Users3` da `Users3` resultado que cada acceso a la propiedad devuelva una nueva instancia del `HashSet<UserDto>` que, aunque C # válido (desde el punto de vista del compilador) es poco probable que sea el comportamiento deseado cuando utilizado para un miembro de la colección.

El código anterior es equivalente a:

```
public class UserGroupDto
{
    // This is a property returning the same instance
    // which was created when the UserGroupDto was instantiated.
    private ICollection<UserDto> _users1 = new HashSet<UserDto>();
    public ICollection<UserDto> Users1 { get { return _users1; } }

    // This is a field returning the same instance
    // which was created when the UserGroupDto was instantiated.
    public virtual ICollection<UserDto> Users2 = new HashSet<UserDto>();

    // This is a property which returns a new HashSet<UserDto> as
    // an ICollection<UserDto> on each call to it.
    public ICollection<UserDto> Users3 { get { return new HashSet<UserDto>(); } }
}
```

## Inicializadores de índice

Los inicializadores de índice permiten crear e inicializar objetos con índices al mismo tiempo.

Esto hace que inicializar diccionarios sea muy fácil:

```
var dict = new Dictionary<string, int>()
{
    ["foo"] = 34,
    ["bar"] = 42
};
```

Cualquier objeto que tenga un getter o setter indexado se puede usar con esta sintaxis:

```
class Program
{
    public class MyClassWithIndexer
    {
        public int this[string index]
        {
            set
            {
                Console.WriteLine($"Index: {index}, value: {value}");
            }
        }
    }

    public static void Main()
    {
        var x = new MyClassWithIndexer()
        {
            ["foo"] = 34,
            ["bar"] = 42
        };

        Console.ReadKey();
    }
}
```

Salida:

Índice: foo, valor: 34
Índice: barra, valor: 42

Ver demostración

Si la clase tiene varios indizadores, es posible asignarlos a todos en un solo grupo de declaraciones:

```
class Program
{
    public class MyClassWithIndexer
    {
        public int this[string index]
        {
            set
            {
                Console.WriteLine($"Index: {index}, value: {value}");
            }
        }
        public string this[int index]
        {
            set
            {
                Console.WriteLine($"Index: {index}, value: {value}");
            }
        }
    }

    public static void Main()
    {
        var x = new MyClassWithIndexer()
        {
            ["foo"] = 34,
            ["bar"] = 42,
            [10] = "Ten",
            [42] = "Meaning of life"
        };
    }
}
```

Salida:

Índice: foo, valor: 34
Índice: barra, valor: 42
Índice: 10, valor: diez
Índice: 42, valor: Significado de la vida.

Se debe tener en cuenta que el descriptor de acceso al `set` del indexador podría comportarse de manera diferente en comparación con un método `Add` (utilizado en inicializadores de colección).

Por ejemplo:

```
var d = new Dictionary<string, int>
{
    ["foo"] = 34,
    ["foo"] = 42,
```

```
}; // does not throw, second value overwrites the first one
```

versus:

```
var d = new Dictionary<string, int>
{
    { "foo", 34 },
    { "foo", 42 },
}; // run-time ArgumentException: An item with the same key has already been added.
```

### Interpolación de cuerdas

La interpolación de cadenas permite al desarrollador combinar `variables` y texto para formar una cadena.

---

# Ejemplo básico

Se crean dos variables `int` : `foo` y `bar` .

```
int foo = 34;
int bar = 42;

string resultString = $"The foo is {foo}, and the bar is {bar}.";

Console.WriteLine(resultString);
```

**Salida** :

> El foo es 34, y la barra es 42.

Ver demostración

Todavía se pueden usar llaves dentro de las cuerdas, como esto:

```
var foo = 34;
var bar = 42;

// String interpolation notation (new style)
Console.WriteLine($"The foo is {{foo}}, and the bar is {{bar}}.");
```

Esto produce el siguiente resultado:

> El foo es {foo}, y la barra es {bar}.

---

# Usando la interpolación con literales de

# cadena textual.

Usar `@` antes de la cadena hará que la cadena sea interpretada textualmente. Entonces, por ejemplo, los caracteres Unicode o los saltos de línea se mantendrán exactamente como se han escrito. Sin embargo, esto no afectará las expresiones en una cadena interpolada como se muestra en el siguiente ejemplo:

```
Console.WriteLine($@"In case it wasn't clear:
\u00B9
The foo
is {foo},
and the bar
is {bar}.");
```

Salida:

> En caso de que no estuviera claro:
> \ u00B9
> El foo
> es 34,
> y el bar
> es 42

Ver demostración

---

# Expresiones

Con la interpolación de cadenas, las *expresiones* entre llaves `{}` también se pueden evaluar. El resultado se insertará en la ubicación correspondiente dentro de la cadena. Por ejemplo, para calcular el máximo de `foo` y `bar` e insertarlo, use `Math.Max` dentro de las llaves:

```
Console.WriteLine($"And the greater one is: { Math.Max(foo, bar) }");
```

Salida:

> Y el mayor es: 42.

*Nota: Cualquier espacio en blanco inicial o final (incluidos el espacio, la pestaña y CRLF / nueva línea) entre la llave y la expresión se ignoran por completo y no se incluyen en la salida*

Ver demostración

Como otro ejemplo, las variables se pueden formatear como una moneda:

```
Console.WriteLine($"Foo formatted as a currency to 4 decimal places: {foo:c4}");
```

Salida:

Foo formateado como una moneda con 4 decimales: $ 34.0000

Ver demostración

O se pueden formatear como fechas:

```
Console.WriteLine($"Today is: {DateTime.Today:dddd, MMMM dd - yyyy}");
```

Salida:

Hoy es: lunes 20 de julio de 2015.

Ver demostración

Las declaraciones con un operador condicional (ternario) también se pueden evaluar dentro de la interpolación. Sin embargo, estos deben estar envueltos entre paréntesis, ya que los dos puntos se utilizan para indicar el formato como se muestra arriba:

```
Console.WriteLine($"{(foo > bar ? "Foo is larger than bar!" : "Bar is larger than foo!")}");
```

Salida:

Bar es más grande que foo!

Ver demostración

Se pueden mezclar expresiones condicionales y especificadores de formato:

```
Console.WriteLine($"Environment: {(Environment.Is64BitProcess ? 64 : 32):00'-bit'} process");
```

Salida:

Medio ambiente: proceso de 32 bits

---

# Secuencias de escape

Los caracteres de barra diagonal inversa ( `\` ) y comillas ( `"` ) funcionan exactamente igual en las cadenas interpoladas que en las cadenas no interpoladas, tanto para literales de cadena textuales como no literales:

```
Console.WriteLine($"Foo is: {foo}. In a non-verbatim string, we need to escape \" and \\ with backslashes.");
Console.WriteLine($@"Foo is: {foo}. In a verbatim string, we need to escape "" with an extra quote, but we don't need to escape \");
```

Salida:

Foo es 34. En una cadena no verbal, necesitamos escapar "y \ con barras invertidas.
Foo tiene 34. En una cadena textual, necesitamos escapar "con una cita extra, pero no necesitamos escapar \

Para incluir un corchete `{` o `}` en una cadena interpolada, use dos corchetes `{{` o `}}` :

```
$"{{foo}} is: {foo}"
```

Salida:

{foo} es: 34

Ver demostración

---

# Tipo FormattableString

El tipo de expresión de interpolación de cadena `$"..."` no siempre es una cadena simple. El compilador decide qué tipo asignar según el contexto:

```
string s = $"hello, {name}";
System.FormattableString s = $"Hello, {name}";
System.IFormattable s = $"Hello, {name}";
```

Este es también el orden de preferencia de tipo cuando el compilador necesita elegir a qué método sobrecargado se llamará.

Un nuevo tipo , `System.FormattableString` , representa una cadena de formato compuesto, junto con los argumentos a formatear. Use esto para escribir aplicaciones que manejen los argumentos de interpolación específicamente:

```
public void AddLogItem(FormattableString formattableString)
{
    foreach (var arg in formattableString.GetArguments())
    {
        // do something to interpolation argument 'arg'
    }

    // use the standard interpolation and the current culture info
    // to get an ordinary String:
    var formatted = formattableString.ToString();

    // ...
}
```

Llame al método anterior con:

```
AddLogItem($"The foo is {foo}, and the bar is {bar}.");
```

Por ejemplo, uno podría elegir no incurrir en el costo de rendimiento de formatear la cadena si el nivel de registro ya iba a filtrar el elemento de registro.

# Conversiones implícitas

Hay conversiones de tipo implícitas de una cadena interpolada:

```
var s = $"Foo: {foo}";
System.IFormattable s = $"Foo: {foo}";
```

También puede producir una variable `IFormattable` que le permita convertir la cadena con un contexto invariante:

```
var s = $"Bar: {bar}";
System.FormattableString s = $"Bar: {bar}";
```

---

# Métodos de cultivo actuales e invariantes

Si el análisis de código está activado, todas las cadenas interpoladas producirán una advertencia CA1305 (Especifique `IFormatProvider` ). Se puede utilizar un método estático para aplicar la cultura actual.

```
public static class Culture
{
    public static string Current(FormattableString formattableString)
    {
        return formattableString?.ToString(CultureInfo.CurrentCulture);
    }
    public static string Invariant(FormattableString formattableString)
    {
        return formattableString?.ToString(CultureInfo.InvariantCulture);
    }
}
```

Luego, para producir una cadena correcta para la cultura actual, solo use la expresión:

```
Culture.Current($"interpolated {typeof(string).Name} string.")
Culture.Invariant($"interpolated {typeof(string).Name} string.")
```

**Nota** : `Current` e `Invariant` no se pueden crear como métodos de extensión porque, de forma predeterminada, el compilador asigna el tipo `String` a la *expresión de cadena interpolada, lo* que hace que el siguiente código no se compile:

```
$"interpolated {typeof(string).Name} string.".Current();
```

`FormattableString` clase `FormattableString` ya contiene el método `Invariant()` , por lo que la forma más sencilla de cambiar a una cultura invariante es `using static` :

```
using static System.FormattableString;
```

```
string invariant = Invariant($"Now = {DateTime.Now}");
string current = $"Now = {DateTime.Now}";
```

---

# Entre bastidores

Las cadenas interpoladas son solo un azúcar sintáctico para `String.Format()` . El compilador ( Roslyn ) lo convertirá en un `String.Format` detrás de escena:

```
var text = $"Hello {name + lastName}";
```

Lo anterior se convertirá en algo como esto:

```
string text = string.Format("Hello {0}", new object[] {
    name + lastName
});
```

---

# Interpolación de cuerdas y Linq

Es posible usar cadenas interpoladas en las declaraciones de Linq para aumentar aún más la legibilidad.

```
var fooBar = (from DataRow x in fooBarTable.Rows
          select string.Format("{0}{1}", x["foo"], x["bar"])).ToList();
```

Puede ser reescrito como:

```
var fooBar = (from DataRow x in fooBarTable.Rows
          select $"{x["foo"]}{x["bar"]}").ToList();
```

# Cuerdas interpoladas reutilizables

Con `string.Format` , puede crear cadenas de formato reutilizables:

```
public const string ErrorFormat = "Exception caught:\r\n{0}";

// ...

Logger.Log(string.Format(ErrorFormat, ex));
```

Sin embargo, las cadenas interpoladas no se compilarán con los marcadores de posición que se refieren a variables no existentes. Lo siguiente no se compilará:

```
public const string ErrorFormat = $"Exception caught:\r\n{error}";
// CS0103: The name 'error' does not exist in the current context
```

En su lugar, cree un `Func<>` que consume variables y devuelve una `String` :

```
public static Func<Exception, string> FormatError =
    error => $"Exception caught:\r\n{error}";

// ...

Logger.Log(FormatError(ex));
```

---

# Interpolación de cuerdas y localización.

Si está localizando su aplicación, puede preguntarse si es posible utilizar la interpolación de cadenas junto con la localización. De hecho, sería bueno tener la posibilidad de almacenar en archivos de recursos `String` s como:

```
"My name is {name} {middlename} {surname}"
```

En lugar de mucho menos legible:

```
"My name is {0} {1} {2}"
```

`String` proceso de interpolación de `String` se produce *en tiempo de compilación* , a diferencia del formato de cadena con cadena. `string.Format` que ocurre *en tiempo de ejecución* . Las expresiones en una cadena interpolada deben hacer referencia a los nombres en el contexto actual y deben almacenarse en archivos de recursos. Eso significa que si quieres usar la localización tienes que hacerlo como:

```
var FirstName = "John";

// method using different resource file "strings"
// for French ("strings.fr.resx"), German ("strings.de.resx"),
// and English ("strings.en.resx")
void ShowMyNameLocalized(string name, string middlename = "", string surname = "")
{
    // get localized string
    var localizedMyNameIs = Properties.strings.Hello;
    // insert spaces where necessary
    name = (string.IsNullOrWhiteSpace(name) ? "" : name + " ");
    middlename = (string.IsNullOrWhiteSpace(middlename) ? "" : middlename + " ");
    surname = (string.IsNullOrWhiteSpace(surname) ? "" : surname + " ");
    // display it
    Console.WriteLine($"{localizedMyNameIs} {name}{middlename}{surname}".Trim());
}

// switch to French and greet John
Thread.CurrentThread.CurrentUICulture = CultureInfo.GetCultureInfo("fr-FR");
ShowMyNameLocalized(FirstName);
```

```
// switch to German and greet John
Thread.CurrentThread.CurrentUICulture = CultureInfo.GetCultureInfo("de-DE");
ShowMyNameLocalized(FirstName);

// switch to US English and greet John
Thread.CurrentThread.CurrentUICulture = CultureInfo.GetCultureInfo("en-US");
ShowMyNameLocalized(FirstName);
```

Si las cadenas de recursos para los idiomas utilizados anteriormente se almacenan correctamente en los archivos de recursos individuales, debe obtener el siguiente resultado:

> Bonjour, mon nom est John
> Hola, mein Nombre ist John
> Hola, mi nombre es John

**Tenga en cuenta** que esto implica que el nombre sigue la cadena localizada en todos los idiomas. Si ese no es el caso, debe agregar marcadores de posición a las cadenas de recursos y modificar la función anterior o debe consultar la información de cultura en la función y proporcionar una declaración de cambio de caso que contenga los diferentes casos. Para obtener más detalles sobre los archivos de recursos, consulte Cómo usar la localización en C # .

Es una buena práctica usar un idioma alternativo predeterminado que la mayoría de la gente entenderá, en caso de que no haya una traducción disponible. Sugiero utilizar el inglés como idioma alternativo predeterminado.

# Interpolación recursiva

Aunque no es muy útil, se permite usar una `string` interpolada recursivamente dentro de los corchetes de otra persona:

```
Console.WriteLine($"String has {$"My class is called {nameof(MyClass)}.".Length} chars:");
Console.WriteLine($"My class is called {nameof(MyClass)}.");
```

Salida:

> La cadena tiene 27 caracteres:
>
> Mi clase se llama MyClass.

## Esperar en la captura y finalmente

Es posible usar la expresión de `await` para aplicar el operador de espera a Tareas o Tarea (de resultados) en la `catch` y, `finally` bloques en C # 6.

No fue posible usar la expresión de `await` en los bloqueos de `catch` y `finally` en versiones anteriores debido a las limitaciones del compilador. C # 6 hace que la espera de tareas asíncronas sea mucho más fácil al permitir la expresión de `await` .

```
try
{
    //since C#5
    await service.InitializeAsync();
}
catch (Exception e)
{
    //since C#6
    await logger.LogAsync(e);
}
finally
{
    //since C#6
    await service.CloseAsync();
}
```

En C # 5 se requería usar un `bool` o declarar una `Exception` fuera del catch catch para realizar operaciones asíncronas. Este método se muestra en el siguiente ejemplo:

```
bool error = false;
Exception ex = null;

try
{
    // Since C#5
    await service.InitializeAsync();
}
catch (Exception e)
{
    // Declare bool or place exception inside variable
    error = true;
    ex = e;
}

// If you don't use the exception
if (error)
{
    // Handle async task
}

// If want to use information from the exception
if (ex != null)
{
    await logger.LogAsync(e);
}

// Close the service, since this isn't possible in the finally
await service.CloseAsync();
```

## Propagación nula

El `?.` operador y `?[...]` operador se llaman el operador condicional nulo . A veces también se hace referencia a otros nombres, como el operador de navegación segura .

Esto es útil, porque si el `.` El operador (miembro de acceso) se aplica a una expresión que se evalúa como `null` , el programa lanzará una `NullReferenceException` . Si el desarrollador usa en su lugar el `?.` Operador (condicional nulo), la expresión se evaluará como nula en lugar de lanzar

una excepción.

Tenga en cuenta que si el `?.` se usa el operador y la expresión es no nula, `?.` y `.` son equivalentes

# Lo esencial

```
var teacherName = classroom.GetTeacher().Name;
// throws NullReferenceException if GetTeacher() returns null
```

Ver demostración

Si el `classroom` no tiene un profesor, `GetTeacher()` puede devolver `null` . Cuando es `null` y se accede a la propiedad `Name` , se lanzará una `NullReferenceException` .

Si modificamos esta declaración para utilizar el `?.` Sintaxis, el resultado de toda la expresión será `null` :

```
var teacherName = classroom.GetTeacher()?.Name;
// teacherName is null if GetTeacher() returns null
```

Ver demostración

Posteriormente, si el `classroom` también pudiera ser `null` , también podríamos escribir esta declaración como:

```
var teacherName = classroom?.GetTeacher()?.Name;
// teacherName is null if GetTeacher() returns null OR classroom is null
```

Ver demostración

Este es un ejemplo de cortocircuito: cuando cualquier operación de acceso condicional que utiliza el operador condicional nulo se evalúa como nulo, la expresión completa se evalúa como nula inmediatamente, sin procesar el resto de la cadena.

Cuando el miembro terminal de una expresión que contiene el operador condicional nulo es de un tipo de valor, la expresión se evalúa como un `Nullable<T>` de ese tipo y, por lo tanto, no puede usarse como un reemplazo directo de la expresión sin `?.` .

```
bool hasCertification = classroom.GetTeacher().HasCertification;
// compiles without error but may throw a NullReferenceException at runtime

bool hasCertification = classroom?.GetTeacher()?.HasCertification;
// compile time error: implicit conversion from bool? to bool not allowed

bool? hasCertification = classroom?.GetTeacher()?.HasCertification;
// works just fine, hasCertification will be null if any part of the chain is null

bool hasCertification = classroom?.GetTeacher()?.HasCertification.GetValueOrDefault();
// must extract value from nullable to assign to a value type variable
```

# Usar con el operador de unión nula (??)

Puede combinar el operador de condición nula con el operador de unión nula ( `??` ) para devolver un valor predeterminado si la expresión se resuelve en `null` . Usando nuestro ejemplo anterior:

```
var teacherName = classroom?.GetTeacher()?.Name ?? "No Name";
// teacherName will be "No Name" when GetTeacher()
// returns null OR classroom is null OR Name is null
```

---

# Usar con indexadores

El operador condicional nulo se puede utilizar con los indizadores :

```
var firstStudentName = classroom?.Students?[0]?.Name;
```

En el ejemplo anterior:

- ¿El primero `?.` Asegura que el `classroom` no sea `null` .
- El segundo `?` asegura que toda la colección `Students` no sea `null` .
- ¿El tercero `?.` después de que el indexador se asegure de que el indexador `[0]` no devolvió un objeto `null` . Se debe tener en cuenta que esta operación **aún** puede lanzar una `IndexOutOfRangeException` .

---

# Usar con funciones vacias

El operador de condición nula también se puede utilizar con funciones de `void` . Sin embargo, en este caso, la declaración no se evaluará como `null` . Simplemente evitará una `NullReferenceException` .

```
List<string> list = null;
list?.Add("hi");          // Does not evaluate to null
```

---

# Utilizar con la invocación de eventos

Asumiendo la siguiente definición de evento:

```
private event EventArgs OnCompleted;
```

Cuando se invoca un evento, tradicionalmente, es una buena práctica verificar si el evento es `null` en caso de que no haya suscriptores presentes:

```
var handler = OnCompleted;
if (handler != null)
{
    handler(EventArgs.Empty);
}
```

Dado que se ha introducido el operador condicional nulo, la invocación se puede reducir a una sola línea:

```
OnCompleted?.Invoke(EventArgs.Empty);
```

---

# Limitaciones

El operador condicional nulo produce rvalue, no lvalue, es decir, no se puede usar para la asignación de propiedades, suscripción de eventos, etc. Por ejemplo, el siguiente código no funcionará:

```
// Error: The left-hand side of an assignment must be a variable, property or indexer
Process.GetProcessById(1337)?.EnableRaisingEvents = true;
// Error: The event can only appear on the left hand side of += or -=
Process.GetProcessById(1337)?.Exited += OnProcessExited;
```

---

# Gotchas

Tenga en cuenta que:

```
int? nameLength = person?.Name.Length;    // safe if 'person' is null
```

**no** es **lo** mismo que

```
int? nameLength = (person?.Name).Length;  // avoid this
```

porque lo primero corresponde a:

```
int? nameLength = person != null ? (int?)person.Name.Length : null;
```

y este último corresponde a:

```
int? nameLength = (person != null ? person.Name : null).Length;
```

A pesar del operador ternario `?:` Se utiliza aquí para explicar la diferencia entre dos casos, estos operadores no son equivalentes. Esto se puede demostrar fácilmente con el siguiente ejemplo:

```
void Main()
```

```
{
    var foo = new Foo();
    Console.WriteLine("Null propagation");
    Console.WriteLine(foo.Bar?.Length);

    Console.WriteLine("Ternary");
    Console.WriteLine(foo.Bar != null ? foo.Bar.Length : (int?)null);
}

class Foo
{
    public string Bar
    {
        get
        {
            Console.WriteLine("I was read");
            return string.Empty;
        }
    }
}
```

Qué salidas:

Propagación nula
Fui leído
0
Ternario
Fui leído
Fui leído
0

Ver demostración

Para evitar múltiples invocaciones equivalentes sería:

```
var interimResult = foo.Bar;
Console.WriteLine(interimResult != null ? interimResult.Length : (int?)null);
```

Y esta diferencia explica en parte por qué el operador de propagación nula aún no se admite en los árboles de expresiones.

## Utilizando el tipo estático

El `using static [Namespace.Type]` directiva `using static [Namespace.Type]` permite la importación de miembros estáticos de tipos y valores de enumeración. Los métodos de extensión se importan como métodos de extensión (de un solo tipo), no al ámbito de nivel superior.

6.0

```
using static System.Console;
using static System.ConsoleColor;
using static System.Math;
```

```
class Program
{
    static void Main()
    {
        BackgroundColor = DarkBlue;
        WriteLine(Sqrt(2));
    }
}
```

Live Demo Fiddle

6.0

```
using System;

class Program
{
    static void Main()
    {
        Console.BackgroundColor = ConsoleColor.DarkBlue;
        Console.WriteLine(Math.Sqrt(2));
    }
}
```

## Resolución mejorada de sobrecarga

El siguiente fragmento de código muestra un ejemplo de cómo pasar un grupo de métodos (a diferencia de un lambda) cuando se espera un delegado. La resolución de sobrecarga ahora resolverá esto en lugar de generar un error de sobrecarga ambiguo debido a la capacidad de **C # 6** para verificar el tipo de retorno del método que se pasó.

```
using System;
public class Program
{
    public static void Main()
    {
        Overloaded(DoSomething);
    }

    static void Overloaded(Action action)
    {
       Console.WriteLine("overload with action called");
    }

    static void Overloaded(Func<int> function)
    {
       Console.WriteLine("overload with Func<int> called");
    }

    static int DoSomething()
    {
        Console.WriteLine(0);
        return 0;
    }
}
```

Resultados:

6.0

**Salida**

> sobrecarga con Func <int> llamada

Ver demostración

5.0

**Error**

> error CS0121: la llamada es ambigua entre los siguientes métodos o propiedades: 'Program.Overloaded (System.Action)' y 'Program.Overloaded (System.Func)'

**C # 6** también puede manejar bien el siguiente caso de coincidencia exacta para expresiones lambda que podría haber resultado en un error en **C # 5** .

```
using System;

class Program
{
    static void Foo(Func<Func<long>> func) {}
    static void Foo(Func<Func<int>> func) {}

    static void Main()
    {
        Foo(() => () => 7);
    }
}
```

## Cambios menores y correcciones de errores

Los paréntesis ahora están prohibidos alrededor de los parámetros nombrados. Lo siguiente se compila en C # 5, pero no en C # 6

5.0

```
Console.WriteLine((value: 23));
```

Los operandos de `is` y `as` ya no pueden ser grupos de métodos. Lo siguiente se compila en C # 5, pero no en C # 6

5.0

```
var result = "".Any is byte;
```

> El compilador nativo lo permitió (aunque mostró una advertencia) y, de hecho, ni siquiera `1.Any is string` compatibilidad del método de extensión, permitiendo cosas locas como `1.Any is string` o `IDisposable.Dispose is object` .

Vea esta referencia para actualizaciones sobre cambios.

## Usando un método de extensión para la inicialización de la colección

La sintaxis de inicialización de la colección se puede usar al crear instancias de cualquier clase que implemente `IEnumerable` y tenga un método llamado `Add` que tome un solo parámetro.

En versiones anteriores, este método `Add` tenía que ser un método de **instancia** en la clase que se estaba inicializando. En C # 6, también puede ser un método de extensión.

```
public class CollectionWithAdd : IEnumerable
{
    public void Add<T>(T item)
    {
        Console.WriteLine("Item added with instance add method: " + item);
    }

    public IEnumerator GetEnumerator()
    {
        // Some implementation here
    }
}

public class CollectionWithoutAdd : IEnumerable
{
    public IEnumerator GetEnumerator()
    {
        // Some implementation here
    }
}

public static class Extensions
{
    public static void Add<T>(this CollectionWithoutAdd collection, T item)
    {
        Console.WriteLine("Item added with extension add method: " + item);
    }
}

public class Program
{
    public static void Main()
    {
        var collection1 = new CollectionWithAdd{1,2,3}; // Valid in all C# versions
        var collection2 = new CollectionWithoutAdd{4,5,6}; // Valid only since C# 6
    }
}
```

Esto dará como resultado:

> Artículo añadido con el método de agregar instancia: 1
> Artículo añadido con el método de agregar instancia: 2
> Elemento agregado con instancia agregar método: 3
> Artículo agregado con extensión añadir método: 4
> Artículo agregado con extensión añadir método: 5
> Artículo agregado con extensión añadir método: 6

### Deshabilitar las mejoras de advertencias

En C # 5.0 y anteriores, el desarrollador solo podía suprimir las advertencias por número. Con la introducción de los analizadores Roslyn, C # necesita una forma de deshabilitar las advertencias emitidas desde bibliotecas específicas. Con C # 6.0, la directiva pragma puede suprimir las advertencias por nombre.

Antes de:

```
#pragma warning disable 0501
```

C # 6.0:

```
#pragma warning disable CS0501
```

Lea Características de C # 6.0 en línea: https://riptutorial.com/es/csharp/topic/24/caracteristicas-de-c-sharp-6-0

# Capítulo 25: Características de C # 7.0

## Introducción

C # 7.0 es la séptima versión de C #. Esta versión contiene algunas características nuevas: soporte de idioma para tuplas, funciones locales, declaraciones de `out var` , separadores de dígitos, literales binarios, coincidencia de patrones, expresiones de lanzamiento, `ref return` `ref local` y `ref local` . Lista de miembros con cuerpo `ref local` y de expresión extendida.

Referencia oficial: Novedades en C # 7

## Examples

### declaración de var

Un patrón común en C # es usar `bool TryParse(object input, out object value)` para analizar objetos de forma segura.

La declaración `out var` es una característica simple para mejorar la legibilidad. Permite que una variable se declare al mismo tiempo que se pasa como un parámetro de salida.

Una variable declarada de esta manera tiene el alcance del resto del cuerpo en el punto en el que se declara.

# Ejemplo

Al usar `TryParse` antes de C # 7.0, debe declarar una variable para recibir el valor antes de llamar a la función:

7.0

```
int value;
if (int.TryParse(input, out value))
{
    Foo(value); // ok
}
else
{
    Foo(value); // value is zero
}

Foo(value); // ok
```

En C # 7.0, puede alinear la declaración de la variable pasada al parámetro `out` , eliminando la necesidad de una declaración de variable separada:

7.0

```
if (int.TryParse(input, out var value))
{
    Foo(value); // ok
}
else
{
    Foo(value); // value is zero
}

Foo(value); // still ok, the value in scope within the remainder of the body
```

Si algunos de los parámetros que devuelve una función de `out` no se necesita se puede utilizar el operador *de descartes* `_` .

```
p.GetCoordinates(out var x, out _); // I only care about x
```

Un `out var` declaración se puede utilizar con cualquier función existente que ya tiene `out` parámetros. La sintaxis de la declaración de la función sigue siendo la misma, y no se necesitan requisitos adicionales para que la función sea compatible con una declaración de `out var` . Esta característica es simplemente el azúcar sintáctico.

Otra característica de `out var` declaración de `out var` es que puede usarse con tipos anónimos.

7.0

```
var a = new[] { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
var groupedByMod2 = a.Select(x => new
                                  {
                                      Source = x,
                                      Mod2 = x % 2
                                  })
                     .GroupBy(x => x.Mod2)
                     .ToDictionary(g => g.Key, g => g.ToArray());
if (groupedByMod2.TryGetValue(1, out var oddElements))
{
    Console.WriteLine(oddElements.Length);
}
```

En este código creamos un `Dictionary` con clave `int` y una matriz de valor de tipo anónimo. En la versión anterior de C # era imposible usar el método `TryGetValue` aquí ya que requería que declararas la variable de `out` (¡que es de tipo anónimo!). Sin embargo, con `out var` no necesitamos especificar explícitamente el tipo de la variable `out` .

# Limitaciones

Tenga en cuenta que las declaraciones var son de uso limitado en las consultas LINQ, ya que las expresiones se interpretan como cuerpos lambda de expresión, por lo que el alcance de las variables introducidas se limita a estos lambdas. Por ejemplo, el siguiente código no funcionará:

```
var nums =
    from item in seq
```

```
    let success = int.TryParse(item, out var tmp)
    select success ? tmp : 0; // Error: The name 'tmp' does not exist in the current context
```

# Referencias

- Original propuesta de declaración de var en GitHub

### Literales binarios

El prefijo **0b** se puede usar para representar literales binarios.

Los literales binarios permiten construir números a partir de ceros y unos, lo que hace que sea mucho más fácil ver qué bits se establecen en la representación binaria de un número. Esto puede ser útil para trabajar con banderas binarias.

Las siguientes son formas equivalentes de especificar un `int` con valor `34` (= $2^5 + 2^1$ ):

```
// Using a binary literal:
//   bits: 76543210
int a1 = 0b00100010;          // binary: explicitly specify bits

// Existing methods:
int a2 = 0x22;                // hexadecimal: every digit corresponds to 4 bits
int a3 = 34;                  // decimal: hard to visualise which bits are set
int a4 = (1 << 5) | (1 << 1); // bitwise arithmetic: combining non-zero bits
```

# Enumeración de banderas

Antes, la especificación de valores de `enum` para una `enum` solo se podía hacer usando uno de los tres métodos en este ejemplo:

```
[Flags]
public enum DaysOfWeek
{
    // Previously available methods:
    //          decimal        hex       bit shifting
    Monday    =  1,    //    = 0x01    = 1 << 0
    Tuesday   =  2,    //    = 0x02    = 1 << 1
    Wednesday =  4,    //    = 0x04    = 1 << 2
    Thursday  =  8,    //    = 0x08    = 1 << 3
    Friday    = 16,    //    = 0x10    = 1 << 4
    Saturday  = 32,    //    = 0x20    = 1 << 5
    Sunday    = 64,    //    = 0x40    = 1 << 6

    Weekdays = Monday | Tuesday | Wednesday | Thursday | Friday,
    Weekends = Saturday | Sunday
}
```

Con los literales binarios es más obvio qué bits se configuran, y su uso no requiere entender los números hexadecimales y la aritmética a nivel de bits:

```
[Flags]
public enum DaysOfWeek
{
    Monday    = 0b00000001,
    Tuesday   = 0b00000010,
    Wednesday = 0b00000100,
    Thursday  = 0b00001000,
    Friday    = 0b00010000,
    Saturday  = 0b00100000,
    Sunday    = 0b01000000,

    Weekdays = Monday | Tuesday | Wednesday | Thursday | Friday,
    Weekends = Saturday | Sunday
}
```

### Separadores de dígitos

El subrayado _ se puede utilizar como un separador de dígitos. Ser capaz de agrupar dígitos en grandes literales numéricos tiene un impacto significativo en la legibilidad.

El subrayado puede aparecer en cualquier lugar en un literal numérico, excepto como se indica a continuación. Diferentes agrupamientos pueden tener sentido en diferentes escenarios o con diferentes bases numéricas.

Cualquier secuencia de dígitos puede estar separada por uno o más guiones bajos. Se permite el _ en decimales así como exponentes. Los separadores no tienen impacto semántico, simplemente se ignoran.

```
int bin = 0b1001_1010_0001_0100;
int hex = 0x1b_a0_44_fe;
int dec = 33_554_432;
int weird = 1_2__3___4____5_____6______7_______8________9;
double real = 1_000.111_1e-1_000;
```

**Donde el separador de dígitos _ no puede ser utilizado:**

- al comienzo del valor ( `_121` )
- al final del valor ( `121_` o `121.05_` )
- al lado del decimal ( `10_.0` )
- junto al personaje exponente ( `1.1e_1` )
- junto al especificador de tipo ( `10_f` )
- inmediatamente después de `0x` o `0b` en literales binarios y hexadecimales ( puede cambiarse para permitir, por ejemplo, 0b_1001_1000 )

### Soporte de idioma para Tuplas

# Lo esencial

Una **tupla** es una lista ordenada y finita de elementos. Las tuplas se usan comúnmente en la programación como un medio para trabajar con una sola entidad colectivamente en lugar de

trabajar individualmente con cada uno de los elementos de la tupla, y para representar filas individuales (es decir, "registros") en una base de datos relacional.

En C # 7.0, los métodos pueden tener múltiples valores de retorno. Detrás de escena, el compilador utilizará la nueva estructura ValueTuple .

```
public (int sum, int count) GetTallies()
{
    return (1, 2);
}
```

*Nota* al `System.ValueTuple` : para que esto funcione en Visual Studio 2017, necesita obtener el paquete `System.ValueTuple` .

Si se asigna un resultado del método de devolución de la tupla a una sola variable, puede acceder a los miembros por sus nombres definidos en la firma del método:

```
var result = GetTallies();
// > result.sum
// 1
// > result.count
// 2
```

# Deconstrucción de tuplas

La deconstrucción de la tupla separa una tupla en sus partes.

Por ejemplo, invocar `GetTallies` y asignar el valor de retorno a dos variables separadas deconstruye la tupla en esas dos variables:

```
(int tallyOne, int tallyTwo) = GetTallies();
```

`var` también funciona:

```
(var s, var c) = GetTallies();
```

También puede usar una sintaxis más corta, con `var` fuera de `()` :

```
var (s, c) = GetTallies();
```

También puedes deconstruir en variables existentes:

```
int s, c;
(s, c) = GetTallies();
```

El intercambio es ahora mucho más simple (no se necesita una variable temporal):

```
(b, a) = (a, b);
```

Curiosamente, cualquier objeto se puede deconstruir definiendo un método `Deconstruct` en la clase:

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public void Deconstruct(out string firstName, out string lastName)
    {
        firstName = FirstName;
        lastName = LastName;
    }
}

var person = new Person { FirstName = "John", LastName = "Smith" };
var (localFirstName, localLastName) = person;
```

En este caso, la `(localFirstName, localLastName) = person` invoca a `Deconstruct` en la `person` .

La deconstrucción puede incluso definirse en un método de extensión. Esto es equivalente a lo anterior:

```
public static class PersonExtensions
{
    public static void Deconstruct(this Person person, out string firstName, out string
lastName)
    {
        firstName = person.FirstName;
        lastName = person.LastName;
    }
}

var (localFirstName, localLastName) = person;
```

Un enfoque alternativo para la clase `Person` es definir el `Name` sí mismo como un `Tuple` . Considera lo siguiente:

```
class Person
{
    public (string First, string Last) Name { get; }

    public Person((string FirstName, string LastName) name)
    {
        Name = name;
    }
}
```

Luego, puede crear una instancia de una persona así (donde podemos tomar una tupla como argumento):

```
var person = new Person(("Jane", "Smith"));

var firstName = person.Name.First; // "Jane"
var lastName = person.Name.Last;   // "Smith"
```

# Inicialización de la tupla

También puede crear arbitrariamente tuplas en el código:

```
var name = ("John", "Smith");
Console.WriteLine(name.Item1);
// Outputs John

Console.WriteLine(name.Item2);
// Outputs Smith
```

Al crear una tupla, puede asignar nombres de elementos ad-hoc a los miembros de la tupla:

```
var name = (first: "John", middle: "Q", last: "Smith");
Console.WriteLine(name.first);
// Outputs John
```

# Inferencia de tipos

Las múltiples tuplas definidas con la misma firma (tipos coincidentes y recuento) se deducirán como tipos coincidentes. Por ejemplo:

```
public (int sum, double average) Measure(List<int> items)
{
    var stats = (sum: 0, average: 0d);
    stats.sum = items.Sum();
    stats.average = items.Average();
    return stats;
}
```

`stats` se pueden devolver ya que la declaración de la variable de `stats` y la firma de devolución del método son una coincidencia.

# Reflexión y nombres de campos de tuplas

Los nombres de los miembros no existen en tiempo de ejecución. Reflexión considerará que las tuplas con el mismo número y tipos de miembros son iguales, incluso si los nombres de los miembros no coinciden. Convertir una tupla en un `object` y luego en una tupla con los mismos tipos de miembros, pero con nombres diferentes, tampoco causará una excepción.

Si bien la clase ValueTuple en sí misma no conserva la información de los nombres de los miembros, la información está disponible a través de la reflexión en un TupleElementNamesAttribute. Este atributo no se aplica a la tupla en sí, sino a los parámetros del método, valores de retorno, propiedades y campos. Esto permite que los nombres de los elementos de la tupla se conserven en todos los ensamblajes, es decir, si un método devuelve

(nombre de cadena, cuenta int), el nombre y el recuento estarán disponibles para los llamadores del método en otro ensamblaje porque el valor de retorno se marcará con TupleElementNameAttribute que contiene los valores "nombre" y "contar".

## Utilizar con genéricos y `async`

Las nuevas características de la tupla (que utilizan el tipo `ValueTuple` subyacente) son totalmente compatibles con los genéricos y se pueden utilizar como parámetro de tipo genérico. Eso hace que sea posible usarlos con el patrón `async` / `await` :

```
public async Task<(string value, int count)> GetValueAsync()
{
    string fooBar = await _stackoverflow.GetStringAsync();
    int num = await _stackoverflow.GetIntAsync();

    return (fooBar, num);
}
```

## Usar con colecciones

Puede ser beneficioso tener una colección de tuplas en (como ejemplo) un escenario en el que intenta encontrar una tupla que coincida con las condiciones para evitar la bifurcación de códigos.

Ejemplo:

```
private readonly List<Tuple<string, string, string>> labels = new List<Tuple<string, string,
string>>()
{
    new Tuple<string, string, string>("test1", "test2", "Value"),
    new Tuple<string, string, string>("test1", "test1", "Value2"),
    new Tuple<string, string, string>("test2", "test2", "Value3"),
};

public string FindMatchingValue(string firstElement, string secondElement)
{
    var result = labels
        .Where(w => w.Item1 == firstElement && w.Item2 == secondElement)
        .FirstOrDefault();

    if (result == null)
        throw new ArgumentException("combo not found");

    return result.Item3;
}
```

Con las nuevas tuplas pueden convertirse:

```
private readonly List<(string firstThingy, string secondThingyLabel, string foundValue)>
labels = new List<(string firstThingy, string secondThingyLabel, string foundValue)>()
{
    ("test1", "test2", "Value"),
```

```
    ("test1", "test1", "Value2"),
    ("test2", "test2", "Value3"),
}

public string FindMatchingValue(string firstElement, string secondElement)
{
    var result = labels
        .Where(w => w.firstThingy == firstElement && w.secondThingyLabel == secondElement)
        .FirstOrDefault();

    if (result == null)
        throw new ArgumentException("combo not found");

    return result.foundValue;
}
```

Aunque la denominación en la tupla de ejemplo anterior es bastante genérica, la idea de etiquetas relevantes permite una comprensión más profunda de lo que se intenta en el código que hace referencia a "item1", "item2", y "item3".

# Diferencias entre ValueTuple y Tuple

La razón principal para la introducción de `ValueTuple` es el rendimiento.

| Escribe un nombre | `ValueTuple` | `Tuple` |
|---|---|---|
| Clase o estructura | `struct` | `class` |
| Mutabilidad (cambio de valores después de la creación) | mudable | inmutable |
| Nombrando miembros y otro soporte de idioma | sí | no ( TBD ) |

# Referencias

- Propuesta original de la característica del lenguaje Tuples en GitHub
- Una solución ejecutable VS 15 para funciones de C # 7.0
- Paquete NuGet Tuple

### Funciones locales

Las funciones locales se definen dentro de un método y no están disponibles fuera de él. Tienen acceso a todas las variables locales y admiten iteradores, `async` / `await` y lambda sintaxis. De esta manera, las repeticiones específicas de una función se pueden funcionalizar sin sobrecargar a la clase. Como efecto secundario, esto mejora el rendimiento de la sugerencia inteligente.

# Ejemplo

```
double GetCylinderVolume(double radius, double height)
{
    return getVolume();

    double getVolume()
    {
        // You can declare inner-local functions in a local function
        double GetCircleArea(double r) => Math.PI * r * r;

        // ALL parents' variables are accessible even though parent doesn't have any input.
        return GetCircleArea(radius) * height;
    }
}
```

Las funciones locales simplifican considerablemente el código para los operadores LINQ, donde normalmente tiene que separar las verificaciones de argumentos de la lógica real para hacer que las verificaciones de los argumentos sean instantáneas, no se demore hasta después de que se inicie la iteración.

# Ejemplo

```
public static IEnumerable<TSource> Where<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, bool> predicate)
{
    if (source == null) throw new ArgumentNullException(nameof(source));
    if (predicate == null) throw new ArgumentNullException(nameof(predicate));

    return iterator();

    IEnumerable<TSource> iterator()
    {
        foreach (TSource element in source)
            if (predicate(element))
                yield return element;
    }
}
```

Las funciones locales también soportan el `async` y `await` palabras clave.

# Ejemplo

```
async Task WriteEmailsAsync()
{
    var emailRegex = new Regex(@"(?i)[a-z0-9_.+-]+@[a-z0-9-]+\.[a-z0-9-.]+");
    IEnumerable<string> emails1 = await getEmailsFromFileAsync("input1.txt");
    IEnumerable<string> emails2 = await getEmailsFromFileAsync("input2.txt");
    await writeLinesToFileAsync(emails1.Concat(emails2), "output.txt");

    async Task<IEnumerable<string>> getEmailsFromFileAsync(string fileName)
    {
        string text;
```

```
        using (StreamReader reader = File.OpenText(fileName))
        {
            text = await reader.ReadToEndAsync();
        }

        return from Match emailMatch in emailRegex.Matches(text) select emailMatch.Value;
    }

    async Task writeLinesToFileAsync(IEnumerable<string> lines, string fileName)
    {
        using (StreamWriter writer = File.CreateText(fileName))
        {
            foreach (string line in lines)
            {
                await writer.WriteLineAsync(line);
            }
        }
    }
}
```

Una cosa importante que puede haber notado es que las funciones locales se pueden definir bajo la declaración de `return` , no **es** necesario que estén definidas arriba. Además, las funciones locales suelen seguir la convención de denominación "lowerCamelCase" para diferenciarse más fácilmente de las funciones de alcance de clase.

### La coincidencia de patrones

Las extensiones de coincidencia de patrones para C # permiten muchos de los beneficios de la coincidencia de patrones de lenguajes funcionales, pero de una manera que se integra sin problemas con la sensación del lenguaje subyacente

## `switch` expresión

La coincidencia de patrones se extiende el `switch` declaración para encender tipos:

```
class Geometry {}

class Triangle : Geometry
{
    public int Width { get; set; }
    public int Height { get; set; }
    public int Base { get; set; }
}

class Rectangle : Geometry
{
    public int Width { get; set; }
    public int Height { get; set; }
}

class Square : Geometry
{
    public int Width { get; set; }
}
```

```
public static void PatternMatching()
{
    Geometry g = new Square { Width = 5 };

    switch (g)
    {
        case Triangle t:
            Console.WriteLine($"{t.Width} {t.Height} {t.Base}");
            break;
        case Rectangle sq when sq.Width == sq.Height:
            Console.WriteLine($"Square rectangle: {sq.Width} {sq.Height}");
            break;
        case Rectangle r:
            Console.WriteLine($"{r.Width} {r.Height}");
            break;
        case Square s:
            Console.WriteLine($"{s.Width}");
            break;
        default:
            Console.WriteLine("<other>");
            break;
    }
}
```

## `is` expresión

La coincidencia de patrones extiende el operador `is` para verificar un tipo y declarar una nueva variable al mismo tiempo.

# Ejemplo

7.0

```
string s = o as string;
if(s != null)
{
    // do something with s
}
```

Se puede reescribir como

7.0

```
if(o is string s)
{
    //Do something with s
};
```

También tenga en cuenta que el alcance de la variable de patrón `s` se extiende hacia fuera del bloque `if` que llega al final del alcance que lo rodea, por ejemplo:

```
if(someCondition)
{
```

```
   if(o is string s)
   {
      //Do something with s
   }
   else
   {
     // s is unassigned here, but accessible
   }

   // s is unassigned here, but accessible
}
// s is not accessible here
```

### ref retorno y ref local

Los retornos de referencia y los locales de referencia son útiles para manipular y devolver referencias a bloques de memoria en lugar de copiar memoria sin recurrir a punteros no seguros.

# Retorno de referencia

```
public static ref TValue Choose<TValue>(
    Func<bool> condition, ref TValue left, ref TValue right)
{
    return condition() ? ref left : ref right;
}
```

Con esto, puede pasar dos valores por referencia y uno de ellos se devuelve en función de alguna condición:

```
Matrix3D left = …, right = …;
Choose(chooser, ref left, ref right).M20 = 1.0;
```

# Ref Local

```
public static ref int Max(ref int first, ref int second, ref int third)
{
    ref int max = first > second ? ref first : ref second;
    return max > third ? ref max : ref third;
}
…
int a = 1, b = 2, c = 3;
Max(ref a, ref b, ref c) = 4;
Debug.Assert(a == 1); // true
Debug.Assert(b == 2); // true
Debug.Assert(c == 4); // true
```

# Operaciones de referencia inseguras

En `System.Runtime.CompilerServices.Unsafe` se ha definido un conjunto de operaciones no seguras que le permiten manipular los valores de `ref` como si fueran punteros, básicamente.

Por ejemplo, reinterpretando una dirección de memoria ( `ref` ) como un tipo diferente:

```
byte[] b = new byte[4] { 0x42, 0x42, 0x42, 0x42 };

ref int r = ref Unsafe.As<byte, int>(ref b[0]);
Assert.Equal(0x42424242, r);

0x0EF00EF0;
Assert.Equal(0xFE, b[0] | b[1] | b[2] | b[3]);
```

Sin embargo, `BitConverter.IsLittleEndian` cuidado con el endianness al hacer esto, por ejemplo, verifique `BitConverter.IsLittleEndian` si es necesario y maneje en consecuencia.

O iterar sobre una matriz de una manera insegura:

```
int[] a = new int[] { 0x123, 0x234, 0x345, 0x456 };

ref int r1 = ref Unsafe.Add(ref a[0], 1);
Assert.Equal(0x234, r1);

ref int r2 = ref Unsafe.Add(ref r1, 2);
Assert.Equal(0x456, r2);

ref int r3 = ref Unsafe.Add(ref r2, -3);
Assert.Equal(0x123, r3);
```

O la `Subtract` similar:

```
string[] a = new string[] { "abc", "def", "ghi", "jkl" };

ref string r1 = ref Unsafe.Subtract(ref a[0], -2);
Assert.Equal("ghi", r1);

ref string r2 = ref Unsafe.Subtract(ref r1, -1);
Assert.Equal("jkl", r2);

ref string r3 = ref Unsafe.Subtract(ref r2, 3);
Assert.Equal("abc", r3);
```

Además, uno puede verificar si dos valores de `ref` son iguales, es decir, la misma dirección:

```
long[] a = new long[2];

Assert.True(Unsafe.AreSame(ref a[0], ref a[0]));
Assert.False(Unsafe.AreSame(ref a[0], ref a[1]));
```

# Campo de golf

Tema de Roslyn Github

System.Runtime.CompilerServices.Unsafe en github

## lanzar expresiones

C # 7.0 permite lanzar como una expresión en ciertos lugares:

```
class Person
{
    public string Name { get; }

    public Person(string name) => Name = name ?? throw new
ArgumentNullException(nameof(name));

    public string GetFirstName()
    {
        var parts = Name.Split(' ');
        return (parts.Length > 0) ? parts[0] : throw new InvalidOperationException("No
name!");
    }

    public string GetLastName() => throw new NotImplementedException();
}
```

Antes de C # 7.0, si deseaba lanzar una excepción desde un cuerpo de expresión, tendría que:

```
var spoons = "dinner,desert,soup".Split(',');

var spoonsArray = spoons.Length > 0 ? spoons : null;

if (spoonsArray == null)
{
    throw new Exception("There are no spoons");
}
```

O

```
var spoonsArray = spoons.Length > 0
    ? spoons
    : new Func<string[]>(() =>
      {
          throw new Exception("There are no spoons");
      })();
```

En C # 7.0, lo anterior ahora está simplificado para:

```
var spoonsArray = spoons.Length > 0 ? spoons : throw new Exception("There are no spoons");
```

## Expresión extendida lista de miembros con cuerpo

C # 7.0 agrega accesores, constructores y finalizadores a la lista de cosas que pueden tener cuerpos de expresión:

```
class Person
```

```
{
    private static ConcurrentDictionary<int, string> names = new ConcurrentDictionary<int,
string>();

    private int id = GetId();

    public Person(string name) => names.TryAdd(id, name); // constructors

    ~Person() => names.TryRemove(id, out _);              // finalizers

    public string Name
    {
        get => names[id];                                 // getters
        set => names[id] = value;                         // setters
    }
}
```

También vea la sección de declaración de var para el operador de descarte.

### ValueTask

`Task<T>` es una **clase** y provoca una sobrecarga innecesaria de su asignación cuando el resultado está disponible de inmediato.

`ValueTask<T>` es una **estructura** y se ha introducido para evitar la asignación de un objeto `Task` en caso de que el resultado de la operación **asíncrona** ya esté disponible en el momento de la espera.

Entonces, `ValueTask<T>` proporciona dos beneficios:

# 1. Aumento de rendimiento

Aquí hay un ejemplo de `Task<T>` :

- Requiere asignación de montón
- Toma 120ns con JIT

```
async Task<int> TestTask(int d)
{
    await Task.Delay(d);
    return 10;
}
```

Aquí está el ejemplo de `ValueTask<T>` analógico:

- Sin asignación del montón si el resultado se conoce de forma sincrónica (que no lo es en este caso debido a la `Task.Delay` , pero a menudo se encuentra en muchos en el mundo real `async` / `await` escenarios)
- Toma 65ns con JIT

```
async ValueTask<int> TestValueTask(int d)
```

```
{
    await Task.Delay(d);
    return 10;
}
```

# 2. Mayor flexibilidad de implementación

De lo contrario, las implementaciones de una interfaz asíncrona que deseen ser síncronas se verían obligadas a utilizar `Task.Run` o `Task.FromResult` (que resulta en la penalización de rendimiento analizada anteriormente). Por lo tanto hay una cierta presión contra implementaciones sincrónicas.

Pero con `ValueTask<T>` , las implementaciones son más libres de elegir entre ser síncronas o asíncronas sin afectar a las personas que llaman.

Por ejemplo, aquí hay una interfaz con un método asíncrono:

```
interface IFoo<T>
{
    ValueTask<T> BarAsync();
}
```

... y así es como se podría llamar ese método:

```
IFoo<T> thing = getThing();
var x = await thing.BarAsync();
```

Con `ValueTask` , el código anterior funcionará con **implementaciones síncronas o asíncronas** :

## Implementación síncrona:

```
class SynchronousFoo<T> : IFoo<T>
{
    public ValueTask<T> BarAsync()
    {
        var value = default(T);
        return new ValueTask<T>(value);
    }
}
```

## Implementación asíncrona

```
class AsynchronousFoo<T> : IFoo<T>
{
    public async ValueTask<T> BarAsync()
    {
        var value = default(T);
        await Task.Delay(1);
        return value;
```

```
    }
}
```

# Notas

Aunque se estaba planeando `ValueTask` estructura `ValueTask` a C # 7.0 , por el momento se ha mantenido como otra biblioteca. ValueTask <T> `System.Threading.Tasks.Extensions` paquete se puede descargar desde la Galería Nuget

Lea Características de C # 7.0 en línea:
https://riptutorial.com/es/csharp/topic/1936/caracteristicas-de-c-sharp-7-0

# Capítulo 26: Clase parcial y metodos

## Introducción

Las clases parciales nos proporcionan una opción para dividir las clases en varias partes y en múltiples archivos de origen. Todas las partes se combinan en una sola clase durante el tiempo de compilación. Todas las partes deben contener la palabra clave `partial` , deben ser de la misma accesibilidad. Todas las partes deben estar presentes en el mismo ensamblaje para que se incluya durante el tiempo de compilación.

## Sintaxis

- clase **parcial** pública MyPartialClass {}

## Observaciones

- Las clases parciales se deben definir dentro del mismo ensamblado y espacio de nombres, como la clase que están extendiendo.
- Todas las partes de la clase deben usar la palabra clave `partial` .
- Todas las partes de la clase deben tener la misma accesibilidad; `public` / `protected` / `private` etc.
- Si alguna parte utiliza la palabra clave `abstract` , el tipo combinado se considera abstracto.
- Si alguna parte utiliza la palabra clave `sealed` , el tipo combinado se considera sellado.
- Si alguna parte utiliza un tipo base, el tipo combinado se hereda de ese tipo.
- El tipo combinado hereda todas las interfaces definidas en todas las clases parciales.

## Examples

### Clases parciales

Las clases parciales proporcionan la capacidad de dividir la declaración de clase (generalmente en archivos separados). Un problema común que puede resolverse con clases parciales es permitir a los usuarios modificar el código generado automáticamente sin temor a que sus cambios se sobrescriban si el código se regenera. También varios desarrolladores pueden trabajar en la misma clase o métodos.

```
using System;

namespace PartialClassAndMethods
{
```

```
    public partial class PartialClass
    {
        public void ExampleMethod() {
            Console.WriteLine("Method call from the first declaration.");
        }
    }

    public partial class PartialClass
    {
        public void AnotherExampleMethod()
        {
            Console.WriteLine("Method call from the second declaration.");
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            PartialClass partial = new PartialClass();
            partial.ExampleMethod(); // outputs "Method call from the first declaration."
            partial.AnotherExampleMethod(); // outputs "Method call from the second
declaration."
        }
    }
}
```

## Metodos parciales

El método parcial consiste en la definición en una declaración de clase parcial (como un escenario común, en la autogenerada) y la implementación en otra declaración de clase parcial.

```
using System;

namespace PartialClassAndMethods
{
    public partial class PartialClass // Auto-generated
    {
        partial void PartialMethod();
    }

    public partial class PartialClass // Human-written
    {
        public void PartialMethod()
        {
            Console.WriteLine("Partial method called.");
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            PartialClass partial = new PartialClass();
            partial.PartialMethod(); // outputs "Partial method called."
        }
    }
}
```

### Clases parciales heredadas de una clase base

Cuando se hereda de cualquier clase base, solo una clase parcial necesita tener la clase base especificada.

```
// PartialClass1.cs
public partial class PartialClass : BaseClass {}

// PartialClass2.cs
public partial class PartialClass {}
```

Se *puede* especificar la *misma* clase base en más de una clase parcial. Algunas herramientas IDE lo marcarán como redundante, pero se compila correctamente.

```
// PartialClass1.cs
public partial class PartialClass : BaseClass {}

// PartialClass2.cs
public partial class PartialClass : BaseClass {} // base class here is redundant
```

No *puede* especificar *diferentes* clases base en varias clases parciales, dará lugar a un error del compilador.

```
// PartialClass1.cs
public partial class PartialClass : BaseClass {} // compiler error

// PartialClass2.cs
public partial class PartialClass : OtherBaseClass {} // compiler error
```

Lea Clase parcial y metodos en línea: https://riptutorial.com/es/csharp/topic/3674/clase-parcial-y-metodos

# Capítulo 27: Clases estáticas

## Examples

### Palabra clave estática

La palabra clave estática significa 2 cosas:

1. Este valor no cambia de un objeto a otro, sino que cambia en una clase como un todo
2. Las propiedades y métodos estáticos no requieren una instancia.

```
public class Foo
{
    public Foo{
        Counter++;
        NonStaticCounter++;
    }

    public static int Counter { get; set; }
    public int NonStaticCounter { get; set; }
}

public class Program
{
    static void Main(string[] args)
    {
        //Create an instance
        var foo1 = new Foo();
        Console.WriteLine(foo1.NonStaticCounter); //this will print "1"

        //Notice this next call doesn't access the instance but calls by the class name.
        Console.WriteLine(Foo.Counter); //this will also print "1"

        //Create a second instance
        var foo2 = new Foo();

        Console.WriteLine(foo2.NonStaticCounter); //this will print "1"

        Console.WriteLine(Foo.Counter); //this will now print "2"
        //The static property incremented on both instances and can persist for the whole
class

    }
}
```

### Clases estáticas

La palabra clave "estática" cuando se refiere a una clase tiene tres efectos:

1. No **puede** crear una instancia de una clase estática (esto incluso elimina el constructor predeterminado)
2. Todas las propiedades y métodos en la clase también **deben** ser estáticos.

3. Una clase `static` es una clase `sealed` , lo que significa que no se puede heredar.

```
public static class Foo
{
    //Notice there is no constructor as this cannot be an instance
    public static int Counter { get; set; }
    public static int GetCount()
    {
        return Counter;
    }
}

public class Program
{
    static void Main(string[] args)
    {
        Foo.Counter++;
        Console.WriteLine(Foo.GetCount()); //this will print 1

        //var foo1 = new Foo();
        //this line would break the code as the Foo class does not have a constructor
    }
}
```

## Vida de clase estática

Una clase `static` se inicializa perezosamente en el acceso de miembros y vive durante la duración del dominio de la aplicación.

```
void Main()
{
    Console.WriteLine("Static classes are lazily initialized");
    Console.WriteLine("The static constructor is only invoked when the class is first
accessed");
    Foo.SayHi();

    Console.WriteLine("Reflecting on a type won't trigger its static .ctor");
    var barType = typeof(Bar);

    Console.WriteLine("However, you can manually trigger it with
System.Runtime.CompilerServices.RuntimeHelpers");
    RuntimeHelpers.RunClassConstructor(barType.TypeHandle);
}

// Define other methods and classes here
public static class Foo
{
    static Foo()
    {
        Console.WriteLine("static Foo.ctor");
    }
    public static void SayHi()
    {
        Console.WriteLine("Foo: Hi");
    }
}
public static class Bar
```

```
{
    static Bar()
    {
        Console.WriteLine("static Bar.ctor");
    }
}
```

Lea Clases estáticas en línea: https://riptutorial.com/es/csharp/topic/1653/clases-estaticas

# Capítulo 28: CLSCompliantAttribute

## Sintaxis

1. [ensamblado: CLSCompliant (verdadero)]
2. [CLSCompleta (verdadero)]

## Parámetros

| Constructor | Parámetro |
|---|---|
| CLSCompliantAttribute (booleano) | Inicializa una instancia de la clase CLSCompliantAttribute con un valor booleano que indica si el elemento del programa indicado es compatible con CLS. |

## Observaciones

La especificación de lenguaje común (CLS) es un conjunto de reglas básicas a las que cualquier idioma que se dirige a la CLI (idioma que confirma las especificaciones de la infraestructura de lenguaje común) debe confirmar para interoperar con otros idiomas compatibles con CLS.

Lista de idiomas CLI

Debe marcar su ensamblaje como CLSC Compliant en la mayoría de los casos cuando distribuye bibliotecas. Este atributo le garantizará que su código será utilizable por todos los idiomas compatibles con CLS. Esto significa que su código puede ser consumido por cualquier idioma que pueda compilarse y ejecutarse en CLR ( Common Language Runtime )

Cuando su ensamblaje está marcado con `CLSCompliantAttribute` , el compilador verificará si su código viola cualquiera de las reglas de CLS y devolverá una **advertencia** si es necesario.

## Examples

### Modificador de acceso al que se aplican las reglas de CLS

```
using System;

[assembly:CLSCompliant(true)]
namespace CLSDoc
{

    public class Cat
    {
        internal UInt16 _age = 0;
        private UInt16 _daysTillVacination = 0;
```

```
        //Warning CS3003  Type of 'Cat.DaysTillVacination' is not CLS-compliant
        protected UInt16 DaysTillVacination
        {
            get { return _daysTillVacination; }
        }

        //Warning    CS3003    Type of 'Cat.Age' is not CLS-compliant
        public UInt16 Age
        { get { return _age; } }

        //valid behaviour by CLS-compliant rules
        public int IncreaseAge()
        {
            int increasedAge = (int)_age + 1;

            return increasedAge;
        }

    }
}
```

Las reglas para el cumplimiento de CLS se aplican solo a componentes públicos / protegidos.

## Violación de la regla CLS: tipos sin firmar / sbyte

```
using System;

[assembly:CLSCompliant(true)]
namespace CLSDoc
{

    public class Car
    {
        internal UInt16 _yearOfCreation = 0;

        //Warning CS3008  Identifier '_numberOfDoors' is not CLS-compliant
        //Warning CS3003  Type of 'Car._numberOfDoors' is not CLS-compliant
        public UInt32 _numberOfDoors = 0;

        //Warning    CS3003    Type of 'Car.YearOfCreation' is not CLS-compliant
        public UInt16 YearOfCreation
        {
            get { return _yearOfCreation; }
        }


        //Warning CS3002  Return type of 'Car.CalculateDistance()' is not CLS-compliant
        public UInt64 CalculateDistance()
        {
            return 0;
        }


        //Warning CS3002  Return type of 'Car.TestDummyUnsignedPointerMethod()' is not CLS-
compliant
        public UIntPtr TestDummyUnsignedPointerMethod()
        {
            int[] arr = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
            UIntPtr ptr = (UIntPtr)arr[0];
```

```
            return ptr;
        }

        //Warning CS3003  Type of 'Car.age' is not CLS-compliant
        public sbyte age = 120;


    }
}
```

## Violación de la regla CLS: misma denominación

```
using System;

[assembly:CLSCompliant(true)]
namespace CLSDoc
{

    public class Car
    {
        //Warning    CS3005    Identifier 'Car.CALCULATEAge()' differing only in case is not
CLS-compliant
        public int CalculateAge()
        {
            return 0;
        }

        public int CALCULATEAge()
        {
            return 0;
        }

    }
}
```

Visual Basic no distingue entre mayúsculas y minúsculas

## Violación de la regla CLS: Identificador _

```
using System;

[assembly:CLSCompliant(true)]
namespace CLSDoc
{

    public class Car
    {
        //Warning CS3008  Identifier '_age' is not CLS-complian
        public int _age = 0;
    }

}
```

No se puede iniciar variable con _

## Violación de la regla CLS: Heredar de una clase que no sea CLSComplaint

```
using System;

[assembly:CLSCompliant(true)]
namespace CLSDoc
{

    [CLSCompliant(false)]
    public class Animal
    {
        public int age = 0;
    }

    //Warning    CS3009    'Dog': base type 'Animal' is not CLS-compliant
    public class Dog : Animal
    {
    }

}
```

Lea CLSCompliantAttribute en línea:
https://riptutorial.com/es/csharp/topic/7214/clscompliantattribute

# Capítulo 29: Código inseguro en .NET

## Observaciones

- Para poder utilizar la palabra clave `unsafe` en un proyecto .Net, debe marcar "Permitir código no seguro" en Propiedades del proyecto => Generar
- El uso de código inseguro puede mejorar el rendimiento, sin embargo, es a costa de la seguridad del código (de ahí el término `unsafe` ).

Por ejemplo, cuando usa un bucle for una matriz como esta:

```
for (int i = 0; i < array.Length; i++)
{
    array[i] = 0;
}
```

.NET Framework garantiza que no exceda los límites de la matriz, lanzando una `IndexOutOfRangeException` si el índice excede los límites.

Sin embargo, si usa un código no seguro, puede exceder los límites de la matriz de esta manera:

```
unsafe
{
    fixed (int* ptr = array)
    {
        for (int i = 0; i <= array.Length; i++)
        {
            *(ptr+i) = 0;
        }
    }
}
```

## Examples

### Índice de matriz insegura

```
void Main()
{
    unsafe
    {
        int[] a = {1, 2, 3};
        fixed(int* b = a)
        {
            Console.WriteLine(b[4]);
        }
    }
}
```

La ejecución de este código crea una matriz de longitud 3, pero luego intenta obtener el quinto elemento (índice 4). En mi máquina, esto imprimió `1910457872` , pero el comportamiento no está

definido.

Sin el bloque `unsafe` , no puede usar punteros y, por lo tanto, no puede acceder a los valores más allá del final de una matriz sin que se genere una excepción.

## Usando inseguro con matrices

Al acceder a matrices con punteros, no hay comprobación de límites y, por lo tanto, no se `IndexOutOfRangeException` una `IndexOutOfRangeException` . Esto hace que el código sea más rápido.

Asignando valores a una matriz con un puntero:

```
class Program
{
    static void Main(string[] args)
    {
        unsafe
        {
            int[] array = new int[1000];
            fixed (int* ptr = array)
            {
                for (int i = 0; i < array.Length; i++)
                {
                    *(ptr+i) = i; //assigning the value with the pointer
                }
            }
        }
    }
}
```

Mientras que la contraparte segura y normal sería:

```
class Program
{
    static void Main(string[] args)
    {
        int[] array = new int[1000];

        for (int i = 0; i < array.Length; i++)
        {
            array[i] = i;
        }
    }
}
```

La parte insegura generalmente será más rápida y la diferencia en el rendimiento puede variar según la complejidad de los elementos de la matriz, así como la lógica aplicada a cada uno. Aunque puede ser más rápido, debe usarse con cuidado ya que es más difícil de mantener y más fácil de romper.

## Usando inseguro con cuerdas

```
var s = "Hello";      // The string referenced by variable 's' is normally immutable, but
                      // since it is memory, we could change it if we can access it in an
```

```
                      // unsafe way.

unsafe                // allows writing to memory; methods on System.String don't allow this
{
  fixed (char* c = s) // get pointer to string originally stored in read only memory
    for (int i = 0; i < s.Length; i++)
      c[i] = 'a';     // change data in memory allocated for original string "Hello"
}
Console.WriteLine(s); // The variable 's' still refers to the same System.String
                      // value in memory, but the contents at that location were
                      // changed by the unsafe write above.
                      // Displays: "aaaaa"
```

Lea Código inseguro en .NET en línea: https://riptutorial.com/es/csharp/topic/81/codigo-inseguro-en--net

# Capítulo 30: Comentarios y regiones

## Examples

### Comentarios

El uso de comentarios en sus proyectos es una forma útil de dejar explicaciones de sus opciones de diseño, y debe apuntar a hacer su vida (o la de otra persona) más fácil al mantener o agregar al código.

Hay dos formas de agregar un comentario a su código.

# Comentarios de una sola línea

Cualquier texto colocado después de `//` será tratado como un comentario.

```
public class Program
{
    // This is the entry point of my program.
    public static void Main()
    {
        // Prints a message to the console. - This is a comment!
        System.Console.WriteLine("Hello, World!");

        // System.Console.WriteLine("Hello, World again!"); // You can even comment out code.
        System.Console.ReadLine();
    }
}
```

# Comentarios multilínea o delimitados.

Cualquier texto entre `/*` y `*/` será tratado como un comentario.

```
public class Program
{
    public static void Main()
    {
        /*
            This is a multi line comment
            it will be ignored by the compiler.
        */
        System.Console.WriteLine("Hello, World!");

        // It's also possible to make an inline comment with /* */
        // although it's rarely used in practice
        System.Console.WriteLine(/* Inline comment */ "Hello, World!");

        System.Console.ReadLine();
    }
```

```
}
```

## Regiones

Una región es un bloque de código plegable, que puede ayudar con la legibilidad y organización de su código.

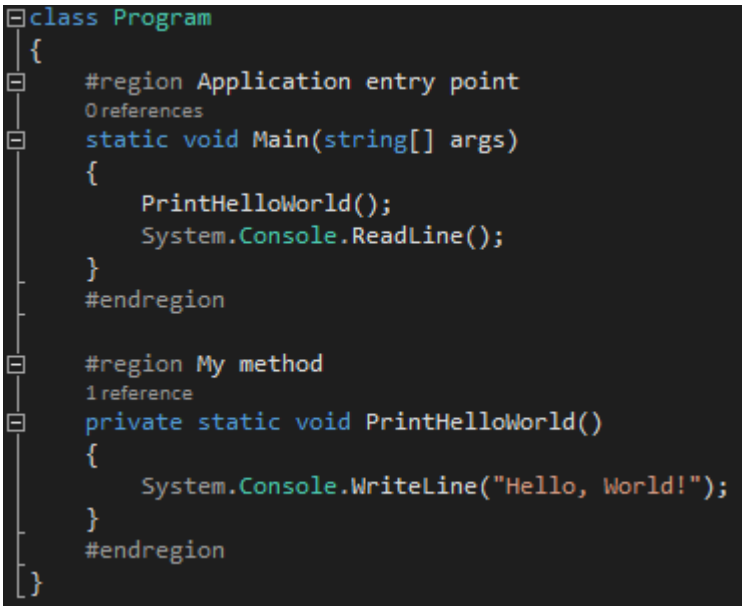
**NOTA:** La regla SA1124 DoNotUseRegions de StyleCop desalienta el uso de regiones. Por lo general, son un signo de código mal organizado, ya que C # incluye clases parciales y otras características que hacen que las regiones queden obsoletas.

Puedes usar las regiones de la siguiente manera:

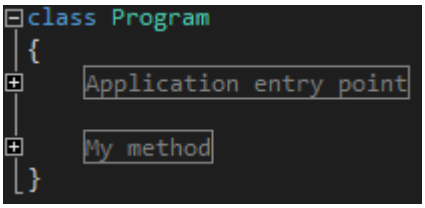
```
class Program
{
    #region Application entry point
    static void Main(string[] args)
    {
        PrintHelloWorld();
        System.Console.ReadLine();
    }
    #endregion

    #region My method
    private static void PrintHelloWorld()
    {
        System.Console.WriteLine("Hello, World!");
    }
    #endregion
}
```

Cuando el código anterior se ve en un IDE, podrá contraer y expandir el código utilizando los símbolos + y -.

### Expandido



**Colapsado**

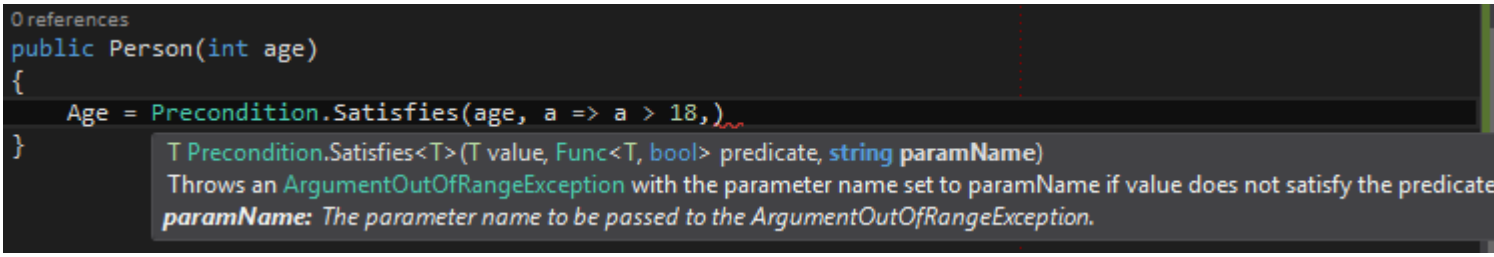



## Comentarios de documentación

Los comentarios de la documentación XML se pueden utilizar para proporcionar documentación de la API que se puede procesar fácilmente mediante herramientas:

```
/// <summary>
/// A helper class for validating method arguments.
/// </summary>
public static class Precondition
{
    /// <summary>
    ///     Throws an <see cref="ArgumentOutOfRangeException"/> with the parameter
    ///     name set to <c>paramName</c> if <c>value</c> does not satisfy the
    ///     <c>predicate</c> specified.
    /// </summary>
    /// <typeparam name="T">
    ///     The type of the argument checked
    /// </typeparam>
    /// <param name="value">
    ///     The argument to be checked
    /// </param>
    /// <param name="predicate">
    ///     The predicate the value is required to satisfy
    /// </param>
    /// <param name="paramName">
    ///     The parameter name to be passed to the
    ///     <see cref="ArgumentOutOfRangeException"/>.
    /// </param>
    /// <returns>The value specified</returns>
    public static T Satisfies<T>(T value, Func<T, bool> predicate, string paramName)
    {
        if (!predicate(value))
            throw new ArgumentOutOfRangeException(paramName);

        return value;
    }
}
```

La documentación es instantáneamente recogida por IntelliSense:



Lea Comentarios y regiones en línea: https://riptutorial.com/es/csharp/topic/5346/comentarios-y-regiones

# Capítulo 31: Comenzando: Json con C #

## Introducción

El siguiente tema presentará una forma de trabajar con Json utilizando el lenguaje C # y los conceptos de serialización y deserialización.

## Examples

### Ejemplo simple de Json

```
{
    "id": 89,
    "name": "Aldous Huxley",
    "type": "Author",
    "books":[{
                "name": "Brave New World",
                "date": 1932
             },
             {
                "name": "Eyeless in Gaza",
                "date": 1936
             },
             {
                "name": "The Genius and the Goddess",
                "date": 1955
             }]
}
```

Si eres nuevo en Json, aquí hay un tutorial ejemplificado .

### Lo primero es lo primero: la biblioteca para trabajar con Json

Para trabajar con Json usando C #, es necesario usar Newtonsoft (biblioteca .net). Esta biblioteca proporciona métodos que permiten al programador serializar y deserializar objetos y más. Hay un tutorial si desea conocer detalles sobre sus métodos y usos.

Si usa Visual Studio, vaya a *Herramientas / Nuget Package Manager / Manage Package to Solution /* y escriba "Newtonsoft" en la barra de búsqueda e instale el paquete. Si no tienes NuGet, este tutorial detallado puede ayudarte.

### Implementación de C #

Antes de leer algunos códigos, es importante comprender y comprender los conceptos principales que ayudarán a programar las aplicaciones utilizando json.

> **Serialización** : proceso de convertir un objeto en un flujo de bytes que se pueden enviar a través de las aplicaciones. El siguiente código puede ser serializado y

convertido al json anterior.

**Deserialización** : proceso de conversión de un json / flujo de bytes en un objeto. Es exactamente el proceso opuesto de serialización. El json anterior se puede deserializar en un objeto C # como se muestra en los ejemplos a continuación.

Para resolver esto, es importante convertir la estructura json en clases para poder utilizar los procesos ya descritos. Si usa Visual Studio, puede convertir un json en una clase automáticamente simplemente seleccionando *"Editar / Pegar Especial / Pegar JSON como Clases"* y pegando la estructura json.

```
using Newtonsoft.Json;

  class Author
{
    [JsonProperty("id")] // Set the variable below to represent the json attribute
    public int id;       //"id"
    [JsonProperty("name")]
    public string name;
    [JsonProperty("type")]
    public string type;
    [JsonProperty("books")]
    public Book[] books;

    public Author(int id, string name, string type, Book[] books) {
        this.id = id;
        this.name = name;
        this.type= type;
        this.books = books;
    }
}

 class Book
{
   [JsonProperty("name")]
   public string name;
   [JsonProperty("date")]
   public DateTime date;
}
```

## Publicación por entregas

```
 static void Main(string[] args)
    {
        Book[] books = new Book[3];
        Author author = new Author(89,"Aldous Huxley","Author",books);
        string objectDeserialized = JsonConvert.SerializeObject(author);
        //Converting author into json
    }
```

El método ".SerializeObject" recibe como parámetro un *objeto de tipo* , por lo que puede poner cualquier cosa en él.

## Deserialización

Puede recibir un json desde cualquier lugar, un archivo o incluso un servidor, por lo que no se incluye en el siguiente código.

```
static void Main(string[] args)
{
    string jsonExample; // Has the previous json
    Author author = JsonConvert.DeserializeObject<Author>(jsonExample);
}
```

El método ".DeserializeObject" deserializa ' *jsonExample* ' en un objeto " *Autor* ". Por esta razón, es importante establecer las variables json en la definición de las clases, de modo que el método acceda a ellas para completarlas.

## Función de utilidad de serialización y deserialización.

Este ejemplo se utiliza para la función común para todo tipo de serialización y deserialización de objetos.

```
using System.Runtime.Serialization.Formatters.Binary;
using System.Xml.Serialization;

namespace Framework
{
public static class IGUtilities
{
        public static string Serialization(this T obj)
        {
        string data = JsonConvert.SerializeObject(obj);
        return data;
        }

        public static T Deserialization(this string JsonData)
        {
        T copy = JsonConvert.DeserializeObject(JsonData);
        return copy;
        }

         public static T Clone(this T obj)
        {
            string data = JsonConvert.SerializeObject(obj);
            T copy = JsonConvert.DeserializeObject(data);
            return copy;
        }
}
}
```

Lea Comenzando: Json con C # en línea:
https://riptutorial.com/es/csharp/topic/9910/comenzando--json-con-c-sharp

# Capítulo 32: Cómo usar C # Structs para crear un tipo de unión (similar a los sindicatos C)

## Observaciones

Los tipos de unión se utilizan en varios idiomas, especialmente en lenguaje C, para contener varios tipos diferentes que pueden "superponerse" en el mismo espacio de memoria. En otras palabras, pueden contener diferentes campos, todos los cuales comienzan en el mismo desplazamiento de memoria, incluso cuando tienen diferentes longitudes y tipos. Esto tiene la ventaja de ahorrar memoria y hacer una conversión automática.

Por favor, tenga en cuenta los comentarios en el constructor de la Struct. El orden en que se inicializan los campos es extremadamente importante. Primero desea inicializar todos los otros campos y luego establecer el valor que pretende cambiar como la última declaración. Debido a que los campos se superponen, la configuración del último valor es la que cuenta.

## Examples

### Uniones de estilo C en C #

Los tipos de unión se utilizan en varios idiomas, como el lenguaje C, para contener varios tipos diferentes que pueden "superponerse". En otras palabras, pueden contener diferentes campos, todos los cuales comienzan en el mismo desplazamiento de memoria, incluso cuando tienen diferentes longitudes y tipos. Esto tiene la ventaja de ahorrar memoria y hacer una conversión automática. Piense en una dirección IP, como un ejemplo. Internamente, una dirección IP se representa como un entero, pero a veces queremos acceder al componente Byte diferente, como en Byte1.Byte2.Byte3.Byte4. Esto funciona para cualquier tipo de valor, ya sea primitivos como Int32 o largo, o para otras estructuras que defina usted mismo.

Podemos lograr el mismo efecto en C # usando estructuras de diseño explícitas.

```
using System;
using System.Runtime.InteropServices;

// The struct needs to be annotated as "Explicit Layout"
[StructLayout(LayoutKind.Explicit)]
struct IpAddress
{
    // The "FieldOffset" means that this Integer starts, an offset in bytes.
    // sizeof(int) 4, sizeof(byte) = 1
    [FieldOffset(0)] public int Address;
    [FieldOffset(0)] public byte Byte1;
    [FieldOffset(1)] public byte Byte2;
    [FieldOffset(2)] public byte Byte3;
    [FieldOffset(3)] public byte Byte4;
```

```
    public IpAddress(int address) : this()
    {
        // When we init the Int, the Bytes will change too.
        Address = address;
    }

    // Now we can use the explicit layout to access the
    // bytes separately, without doing any conversion.
    public override string ToString() => $"{Byte1}.{Byte2}.{Byte3}.{Byte4}";
}
```

Habiendo definido Struct de esta manera, podemos usarlo como usaríamos una Unión en C. Por ejemplo, creemos una dirección IP como un Entero Aleatorio y luego modifiquemos el primer token en la dirección a '100', al cambiarlo de 'ABCD' a '100.BCD':

```
var ip = new IpAddress(new Random().Next());
Console.WriteLine($"{ip} = {ip.Address}");
ip.Byte1 = 100;
Console.WriteLine($"{ip} = {ip.Address}");
```

Salida:

```
75.49.5.32 = 537211211
100.49.5.32 = 537211236
```

Ver demostración

## Los tipos de unión en C # también pueden contener campos Struct

Aparte de las primitivas, las estructuras de diseño explícito (Unions) en C #, también pueden contener otras estructuras. Mientras un campo sea un tipo de valor y no una referencia, puede estar contenido en una unión:

```
using System;
using System.Runtime.InteropServices;

// The struct needs to be annotated as "Explicit Layout"
[StructLayout(LayoutKind.Explicit)]
struct IpAddress
{
    // Same definition of IpAddress, from the example above
}

// Now let's see if we can fit a whole URL into a long

// Let's define a short enum to hold protocols
enum Protocol : short { Http, Https, Ftp, Sftp, Tcp }

// The Service struct will hold the Address, the Port and the Protocol
[StructLayout(LayoutKind.Explicit)]
struct Service
{
    [FieldOffset(0)] public IpAddress Address;
    [FieldOffset(4)] public ushort Port;
```

```
    [FieldOffset(6)] public Protocol AppProtocol;
    [FieldOffset(0)] public long Payload;

    public Service(IpAddress address, ushort port, Protocol protocol)
    {
        Payload = 0;
        Address = address;
        Port  = port;
        AppProtocol = protocol;
    }

    public Service(long payload)
    {
        Address = new IpAddress(0);
        Port = 80;
        AppProtocol = Protocol.Http;
        Payload = payload;
    }

    public Service Copy() => new Service(Payload);

    public override string ToString() => $"{AppProtocol}//{Address}:{Port}/";
}
```

Ahora podemos verificar que toda la Unión de Servicios se ajuste al tamaño de un largo (8 bytes).

```
var ip = new IpAddress(new Random().Next());
Console.WriteLine($"Size: {Marshal.SizeOf(ip)} bytes. Value: {ip.Address} = {ip}.");

var s1 = new Service(ip, 8080, Protocol.Https);
var s2 = new Service(s1.Payload);
s2.Address.Byte1 = 100;
s2.AppProtocol = Protocol.Ftp;

Console.WriteLine($"Size: {Marshal.SizeOf(s1)} bytes. Value: {s1.Address} = {s1}.");
Console.WriteLine($"Size: {Marshal.SizeOf(s2)} bytes. Value: {s2.Address} = {s2}.");
```

Ver demostración

Lea Cómo usar C # Structs para crear un tipo de unión (similar a los sindicatos C) en línea: https://riptutorial.com/es/csharp/topic/5626/como-usar-c-sharp-structs-para-crear-un-tipo-de-union--similar-a-los-sindicatos-c-

# Capítulo 33: Compilación de tiempo de ejecución

## Examples

### RoslynScript

`Microsoft.CodeAnalysis.CSharp.Scripting.CSharpScript` es un nuevo motor de script de C #.

```
var code = "(1 + 2).ToString()";
var run = await CSharpScript.RunAsync(code, ScriptOptions.Default);
var result = (string)run.ReturnValue;
Console.WriteLine(result); //output 3
```

Puede compilar y ejecutar cualquier declaración, variable, método, clase o segmento de código.

### CSharpCodeProvider

`Microsoft.CSharp.CSharpCodeProvider` se puede usar para compilar clases de C #.

```
var code = @"
    public class Abc {
       public string Get() { return ""abc""; }
    }
";

var options = new CompilerParameters();
options.GenerateExecutable = false;
options.GenerateInMemory = false;

var provider = new CSharpCodeProvider();
var compile = provider.CompileAssemblyFromSource(options, code);

var type = compile.CompiledAssembly.GetType("Abc");
var abc = Activator.CreateInstance(type);

var method = type.GetMethod("Get");
var result = method.Invoke(abc, null);

Console.WriteLine(result); //output: abc
```

Lea Compilación de tiempo de ejecución en línea:
https://riptutorial.com/es/csharp/topic/3139/compilacion-de-tiempo-de-ejecucion

# Capítulo 34: Comprobado y desactivado

## Sintaxis

- marcada (a + b) // expresión marcada
- deseleccionado (a + b) // expresión deseleccionada
- marcada {c = a + b; c + = 5; } // bloque marcado
- sin marcar {c = a + b; c + = 5; } // bloque sin marcar

## Examples

### Comprobado y desactivado

Las declaraciones de C # se ejecutan en el contexto marcado o no marcado. En un contexto comprobado, el desbordamiento aritmético genera una excepción. En un contexto no verificado, el desbordamiento aritmético se ignora y el resultado se trunca.

```
short m = 32767;
short n = 32767;
int result1 =  checked((short)(m + n));   //will throw an OverflowException
int result2 =  unchecked((short)(m + n)); // will return -2
```

Si no se especifica ninguno de estos, el contexto predeterminado se basará en otros factores, como las opciones del compilador.

### Comprobado y desactivado como un alcance

Las palabras clave también pueden crear ámbitos para (des) verificar múltiples operaciones.

```
short m = 32767;
short n = 32767;
checked
{
    int result1 = (short)(m + n); //will throw an OverflowException
}
unchecked
{
    int result2 = (short)(m + n); // will return -2
}
```

Lea Comprobado y desactivado en línea: https://riptutorial.com/es/csharp/topic/2394/comprobado-y-desactivado

# Capítulo 35: Concatenación de cuerdas

## Observaciones

Si está creando una cadena dinámica, es una buena práctica optar por la clase `StringBuilder` lugar de unir cadenas mediante el método + o `Concat` , ya que cada + / `Concat` crea un nuevo objeto de cadena cada vez que se ejecuta.

## Examples

### + Operador

```
string s1 = "string1";
string s2 = "string2";

string s3 = s1 + s2; // "string1string2"
```

### Concatenar cadenas utilizando System.Text.StringBuilder

La concatenación de cadenas con un StringBuilder puede ofrecer ventajas de rendimiento en comparación con la simple concatenación de cadenas con `+` . Esto se debe a la forma en que se asigna la memoria. Las cadenas se reasignan con cada concatenación, los StringBuilders asignan memoria en bloques y solo se reasignan cuando se agota el bloque actual. Esto puede hacer una gran diferencia cuando se hacen muchas concatenaciones pequeñas.

```
StringBuilder sb = new StringBuilder();
for (int i = 1; i <= 5; i++)
{
    sb.Append(i);
    sb.Append(" ");
}
Console.WriteLine(sb.ToString()); // "1 2 3 4 5 "
```

Las llamadas a `Append()` se pueden conectar en cadena, porque devuelve una referencia al `StringBuilder` :

```
StringBuilder sb = new StringBuilder();
sb.Append("some string ")
  .Append("another string");
```

### Elementos de matriz de cadena de concat utilizando String.Join

El método `String.Join` se puede usar para concatenar múltiples elementos de una matriz de cadenas.

```
string[] value = {"apple", "orange", "grape", "pear"};
```

```
string separator = ", ";

string result = String.Join(separator, value, 1, 2);
Console.WriteLine(result);
```

Produce la siguiente salida: "naranja, uva".

Este ejemplo utiliza la `String.Join(String, String[], Int32, Int32)` , que especifica el índice de inicio y el recuento sobre el separador y el valor.

Si no desea utilizar el índice de inicio y las sobrecargas de conteo, puede unir todas las cadenas dadas. Me gusta esto:

```
string[] value = {"apple", "orange", "grape", "pear"};
string separator = ", ";
string result = String.Join(separator, value);
Console.WriteLine(result);
```

que producirá;

manzana, naranja, uva, pera

## Concatenación de dos cuerdas usando $

$ proporciona un método fácil y conciso para concatenar múltiples cadenas.

```
var str1 = "text1";
var str2 = " ";
var str3 = "text3";
string result2 = $"{str1}{str2}{str3}"; //"text1 text3"
```

Lea Concatenación de cuerdas en línea:
https://riptutorial.com/es/csharp/topic/3616/concatenacion-de-cuerdas

# Capítulo 36: Construcciones de flujo de datos de la biblioteca paralela de tareas (TPL)

## Examples

### JoinBlock

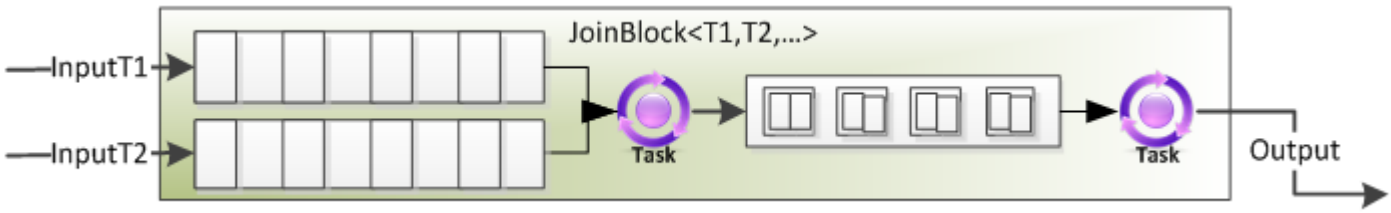
(Recoge 2-3 entradas y las combina en una tupla)

Al igual que BatchBlock, JoinBlock <T1, T2, ...> puede agrupar datos de múltiples fuentes de datos. De hecho, ese es el propósito principal de JoinBlock <T1, T2, ...>.

Por ejemplo, un JoinBlock <string, double, int> es un ISourceBlock <Tuple <string, double, int >>.

Al igual que con BatchBlock, JoinBlock <T1, T2, ...> es capaz de operar tanto en modo codicioso como no codicioso.

- En el modo codicioso predeterminado, se aceptan todos los datos ofrecidos a los objetivos, incluso si el otro objetivo no tiene los datos necesarios para formar una tupla.
- En el modo no ambicioso, los objetivos del bloque pospondrán los datos hasta que a todos los objetivos se les ofrezcan los datos necesarios para crear una tupla, momento en el que el bloque se involucrará en un protocolo de confirmación de dos fases para recuperar atómicamente todos los elementos necesarios de las fuentes. Este aplazamiento hace posible que otra entidad consuma los datos mientras tanto para permitir que el sistema en general avance hacia adelante.




**Procesamiento de solicitudes con un número limitado de objetos agrupados**

```
var throttle = new JoinBlock<ExpensiveObject, Request>();
for(int i=0; i<10; i++)
{
    requestProcessor.Target1.Post(new ExpensiveObject());
}

var processor = new Transform<Tuple<ExpensiveObject, Request>, ExpensiveObject>(pair =>
{
    var resource = pair.Item1;
    var request = pair.Item2;

    request.ProcessWith(resource);

    return resource;
```

```
});

throttle.LinkTo(processor);
processor.LinkTo(throttle.Target1);
```

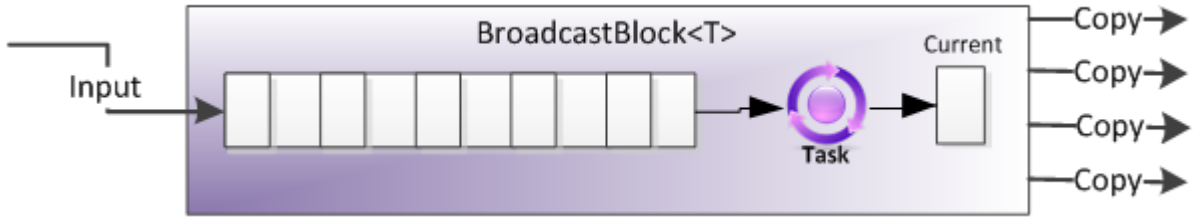
Introducción al flujo de datos TPL por Stephen Toub

## BroadcastBlock

(Copie un elemento y envíe las copias a cada bloque al que esté vinculado)

A diferencia de BufferBlock, la misión de BroadcastBlock en la vida es permitir que todos los objetivos vinculados desde el bloque obtengan una copia de cada elemento publicado, sobrescribiendo continuamente el valor "actual" con los propagados a él.

Además, a diferencia de BufferBlock, BroadcastBlock no retiene datos innecesariamente. Después de que se haya ofrecido un dato en particular a todos los destinos, ese elemento será sobrescrito por cualquier pieza de datos que se encuentre en la siguiente línea (como ocurre con todos los bloques de flujo de datos, los mensajes se manejan en orden FIFO). Ese elemento se ofrecerá a todos los objetivos, y así sucesivamente.




### Productor asincrónico / consumidor con un productor estrangulado

```
var ui = TaskScheduler.FromCurrentSynchronizationContext();
var bb = new BroadcastBlock<ImageData>(i => i);

var saveToDiskBlock = new ActionBlock<ImageData>(item =>
    item.Image.Save(item.Path)
);

var showInUiBlock = new ActionBlock<ImageData>(item =>
    imagePanel.AddImage(item.Image),
    new DataflowBlockOptions { TaskScheduler =
TaskScheduler.FromCurrentSynchronizationContext() }
);

bb.LinkTo(saveToDiskBlock);
bb.LinkTo(showInUiBlock);
```

### Exponer el estado de un agente

```
public class MyAgent
{
    public ISourceBlock<string> Status { get; private set; }
```

```
    public MyAgent()
    {
        Status = new BroadcastBlock<string>();
        Run();
    }

    private void Run()
    {
        Status.Post("Starting");
        Status.Post("Doing cool stuff");
        …
        Status.Post("Done");
    }
}
```

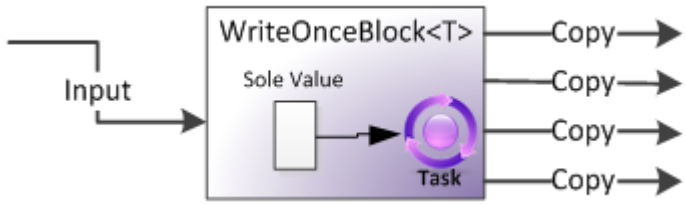
Introducción al flujo de datos TPL por Stephen Toub

## WriteOnceBlock

(Variable de solo lectura: Memoriza su primer elemento de datos y distribuye copias de él como salida. Ignora todos los demás elementos de datos)

Si BufferBlock es el bloque más fundamental en el flujo de datos TPL, WriteOnceBlock es el más simple.
Almacena a lo sumo un valor, y una vez que se ha establecido ese valor, nunca se reemplazará ni se sobrescribirá.

Puede pensar que WriteOnceBlock es similar a una variable miembro de solo lectura en C #, excepto que en lugar de ser solo configurable en un constructor y luego ser inmutable, solo se puede configurar una vez y luego es inmutable.




**Dividir las salidas potenciales de una tarea**

```
public static async void SplitIntoBlocks(this Task<T> task,
    out IPropagatorBlock<T> result,
    out IPropagatorBlock<Exception> exception)
{
    result = new WriteOnceBlock<T>(i => i);
    exception = new WriteOnceBlock<Exception>(i => i);

    try
    {
        result.Post(await task);
    }
    catch(Exception ex)
    {
        exception.Post(ex);
```

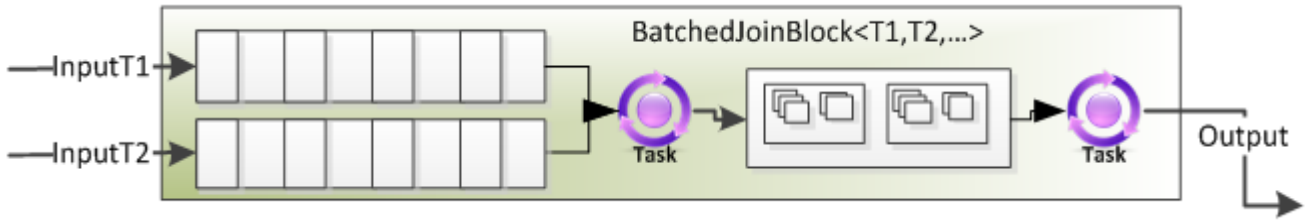
```
    }
}
```

Introducción al flujo de datos TPL por Stephen Toub

## BatchedJoinBlock

(Recopila una cierta cantidad de elementos totales de 2-3 entradas y los agrupa en una tupla de colecciones de elementos de datos)

BatchedJoinBlock <T1, T2, ...> es en cierto sentido una combinación de BatchBlock y JoinBlock <T1, T2, ...>.
Mientras que JoinBlock <T1, T2, ...> se usa para agregar una entrada de cada objetivo en una tupla, y BatchBlock se usa para agregar N entradas en una colección, BatchedJoinBlock <T1, T2, ...> se usa para reunir N entradas desde Todos los objetivos en tuplas de colecciones.




**Dispersión / reunión**

Considere un problema de dispersión / recopilación en el que se inician N operaciones, algunas de las cuales pueden tener éxito y producir resultados de cadena, y otras pueden fallar y producir Excepciones.

```
var batchedJoin = new BatchedJoinBlock<string, Exception>(10);

for (int i=0; i<10; i++)
{
    Task.Factory.StartNew(() => {
        try { batchedJoin.Target1.Post(DoWork()); }
        catch(Exception ex) { batchJoin.Target2.Post(ex); }
    });
}

var results = await batchedJoin.ReceiveAsync();

foreach(string s in results.Item1)
{
    Console.WriteLine(s);
}

foreach(Exception e in results.Item2)
{
    Console.WriteLine(e);
}
```

Introducción al flujo de datos TPL por Stephen Toub

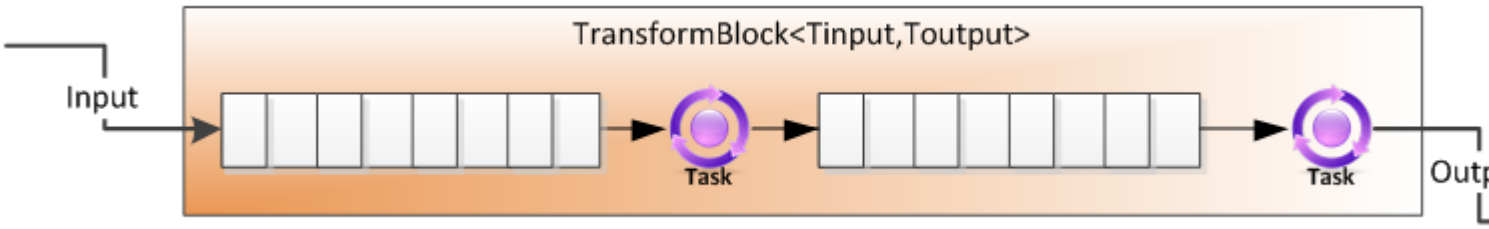
## TransformBlock

(Seleccione, uno a uno)

Al igual que con ActionBlock, TransformBlock <TInput, TOutput> permite la ejecución de un delegado para realizar alguna acción para cada dato de entrada; **a diferencia de ActionBlock, este procesamiento tiene una salida.** Este delegado puede ser un Func <TInput, TOutput>, en cuyo caso el procesamiento de ese elemento se considera completado cuando el delegado regresa, o puede ser un Func <TInput, Task>, en cuyo caso el procesamiento de ese elemento se considera completado no cuando el delegado regresa pero cuando la tarea devuelta se completa. Para aquellos familiarizados con LINQ, es algo similar a Select () en que toma una entrada, transforma esa entrada de alguna manera y luego produce una salida.

De forma predeterminada, TransformBlock <TInput, TOutput> procesa sus datos secuencialmente con un MaxDegreeOfParallelism igual a 1. Además de recibir una entrada almacenada en el búfer y procesarla, este bloque también tomará toda su salida procesada y el búfer (datos que no han sido procesados). procesados, y datos que han sido procesados).

Tiene 2 tareas: una para procesar los datos y otra para enviar los datos al siguiente bloque.




**Un oleoducto concurrente**

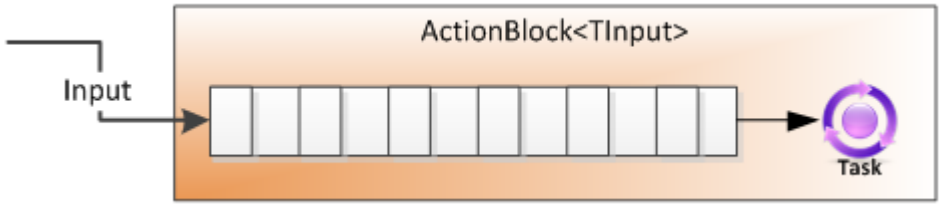
```
var compressor = new TransformBlock<byte[], byte[]>(input => Compress(input));
var encryptor = new TransformBlock<byte[], byte[]>(input => Encrypt(input));

compressor.LinkTo(Encryptor);
```

Introducción al flujo de datos TPL por Stephen Toub

## ActionBlock

(para cada)

Esta clase puede considerarse lógicamente como un búfer para que los datos se procesen combinados con tareas para procesar esos datos, con el "bloque de flujo de datos" gestionando ambos. En su uso más básico, podemos instanciar un ActionBlock y "publicar" datos en él; El delegado proporcionado en la construcción del ActionBlock se ejecutará de forma asincrónica para cada pieza de datos publicada.




**Computación sincrónica**

```
var ab = new ActionBlock<TInput>(i =>
{
    Compute(i);
});
…
ab.Post(1);
ab.Post(2);
ab.Post(3);
```

**Aceleración de descargas asíncronas a un máximo de 5 al mismo tiempo**

```
var downloader = new ActionBlock<string>(async url =>
{
    byte [] imageData = await DownloadAsync(url);
    Process(imageData);
}, new DataflowBlockOptions { MaxDegreeOfParallelism = 5 });

downloader.Post("http://website.com/path/to/images");
downloader.Post("http://another-website.com/path/to/images");
```

Introducción al flujo de datos TPL por Stephen Toub

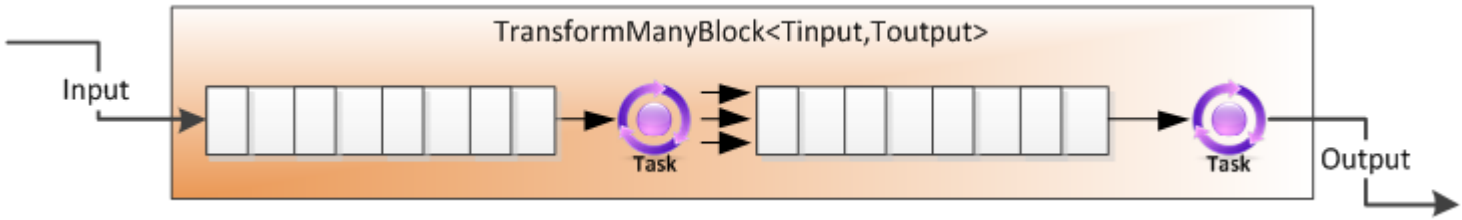
## TransformManyBlock

(SelectMany, 1-m: Los resultados de este mapeo son "aplanados", al igual que SelectMany de LINQ)

TransformManyBlock <TInput, TOutput> es muy similar a TransformBlock <TInput, TOutput>. La diferencia clave es que mientras un TransformBlock <TInput, TOutput> produce una y solo una salida para cada entrada, TransformManyBlock <TInput, TOutput> produce cualquier número de salidas (cero o más) para cada entrada. Al igual que con ActionBlock y TransformBlock <TInput, TOutput>, este procesamiento se puede especificar utilizando delegados, tanto para el procesamiento síncrono como para el asíncrono.

Un Func <TInput, IEnumerable> se usa para síncrono, y un Func <TInput, Task <IEnumerable>> se usa para asíncrono. Al igual que con ActionBlock y TransformBlock <TInput, TOutput>, TransformManyBlock <TInput, TOutput>, los valores predeterminados son procesadores secuenciales, pero pueden configurarse de otro modo.

El delegado de asignación vuelve a ejecutar una colección de elementos, que se insertan individualmente en el búfer de salida.




**Rastreador Web Asíncrono**

```
var downloader = new TransformManyBlock<string, string>(async url =>
{
    Console.WriteLine("Downloading " + url);
    try
    {
        return ParseLinks(await DownloadContents(url));
    }
    catch{}

    return Enumerable.Empty<string>();
});
downloader.LinkTo(downloader);
```

**Expandiendo un Enumerable en sus Elementos Constituyentes**

```
var expanded = new TransformManyBlock<T[], T>(array => array);
```

**Filtrado pasando de 1 a 0 o 1 elementos.**

```
public IPropagatorBlock<T> CreateFilteredBuffer<T>(Predicate<T> filter)
{
    return new TransformManyBlock<T, T>(item =>
        filter(item) ? new [] { item } : Enumerable.Empty<T>());
}
```

Introducción al flujo de datos TPL por Stephen Toub

## BatchBlock

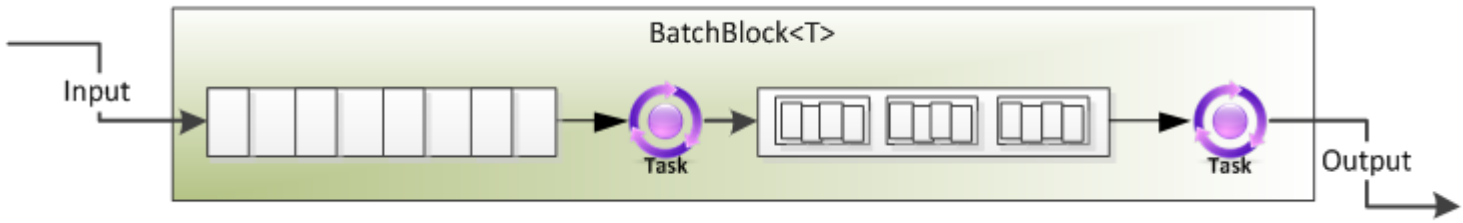
(Agrupa un cierto número de elementos de datos secuenciales en colecciones de elementos de datos)

BatchBlock combina N elementos únicos en un artículo por lotes, representado como una matriz de elementos. Se crea una instancia con un tamaño de lote específico, y luego el bloque crea un lote tan pronto como recibe ese número de elementos, enviando el lote de forma asíncrona al búfer de salida.

BatchBlock es capaz de ejecutarse en modo codicioso y no codicioso.

- En el modo codicioso predeterminado, todos los mensajes ofrecidos al bloque desde cualquier número de fuentes se aceptan y almacenan en búfer para convertirlos en lotes.
- ◦ En el modo no codicioso, todos los mensajes se posponen desde las fuentes hasta

que suficientes fuentes hayan ofrecido mensajes al bloque para crear un lote. Por lo tanto, se puede usar un BatchBlock para recibir 1 elemento de cada una de N fuentes, N elementos de 1 fuente y una gran cantidad de opciones entre ellas.




**Solicitudes por lotes en grupos de 100 para enviar a una base de datos**

```
var batchRequests = new BatchBlock<Request>(batchSize:100);
var sendToDb = new ActionBlock<Request[]>(reqs => SubmitToDatabase(reqs));

batchRequests.LinkTo(sendToDb);
```

**Creando un lote una vez por segundo**

```
var batch = new BatchBlock<T>(batchSize:Int32.MaxValue);
new Timer(() => { batch.TriggerBatch(); }).Change(1000, 1000);
```

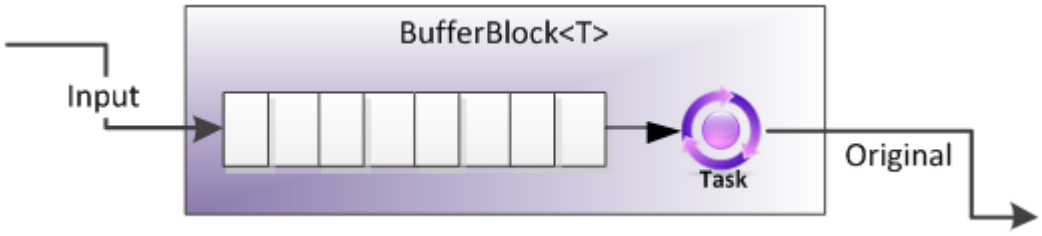
Introducción al flujo de datos TPL por Stephen Toub

## BufferBlock

(Cola FIFO: los datos que entran son los datos que salen)

En resumen, BufferBlock proporciona un búfer ilimitado o limitado para almacenar instancias de T.

Puede "publicar" instancias de T en el bloque, lo que hace que los datos que se están publicando se almacenen en un orden de primero en entrar, primero en salir (FIFO) por el bloque.
Puede "recibir" del bloque, lo que le permite obtener de forma sincrónica o asíncrona instancias de T previamente almacenadas o disponibles en el futuro (nuevamente, FIFO).




**Productor asincrónico / consumidor con un productor estrangulado**

```
// Hand-off through a bounded BufferBlock<T>
private static BufferBlock<int> _Buffer = new BufferBlock<int>(
    new DataflowBlockOptions { BoundedCapacity = 10 });
```

```
// Producer
private static async void Producer()
{
    while(true)
    {
        await _Buffer.SendAsync(Produce());
    }
}

// Consumer
private static async Task Consumer()
{
    while(true)
    {
        Process(await _Buffer.ReceiveAsync());
    }
}

// Start the Producer and Consumer
private static async Task Run()
{
    await Task.WhenAll(Producer(), Consumer());
}
```

Introducción al flujo de datos TPL por Stephen Toub

Lea Construcciones de flujo de datos de la biblioteca paralela de tareas (TPL) en línea: https://riptutorial.com/es/csharp/topic/3110/construcciones-de-flujo-de-datos-de-la-biblioteca-paralela-de-tareas--tpl-

# Capítulo 37: Constructores y finalizadores

## Introducción

Los constructores son métodos en una clase que se invocan cuando se crea una instancia de esa clase. Su principal responsabilidad es dejar el nuevo objeto en un estado útil y consistente.

Los destructores / finalizadores son métodos en una clase que se invocan cuando una instancia de eso se destruye. En C # rara vez se escriben / usan explícitamente.

## Observaciones

C # en realidad no tiene destructores, sino finalizadores que usan la sintaxis del destructor de estilo C ++. Especificar un destructor anula el método `Object.Finalize()` que no se puede llamar directamente.

A diferencia de otros idiomas con una sintaxis similar, estos métodos *no se* llaman cuando los objetos están fuera del alcance, sino que se llaman cuando se ejecuta el recolector de basura, lo que ocurre bajo ciertas condiciones . Como tales, *no* están garantizados para funcionar en un orden particular.

Los finalizadores deben ser responsables de limpiar **solo los** recursos no administrados (los punteros adquiridos a través de la clase Marshal, recibidos a través de p / Invoke (llamadas al sistema) o los punteros sin procesar utilizados en bloques inseguros). Para limpiar los recursos administrados, revise IDisposable, el patrón de Disposición y la declaración de `using` .

(Lectura adicional: ¿ Cuándo debo crear un destructor? )

## Examples

### Constructor predeterminado

Cuando un tipo se define sin un constructor:

```
public class Animal
{
}
```

entonces el compilador genera un constructor predeterminado equivalente a lo siguiente:

```
public class Animal
{
    public Animal() {}
}
```

La definición de cualquier constructor para el tipo suprimirá la generación del constructor por

defecto. Si el tipo se definiera de la siguiente manera:

```
public class Animal
{
    public Animal(string name) {}
}
```

entonces solo se puede crear un `Animal` llamando al constructor declarado.

```
// This is valid
var myAnimal = new Animal("Fluffy");
// This fails to compile
var unnamedAnimal = new Animal();
```

Para el segundo ejemplo, el compilador mostrará un mensaje de error:

> 'Animal' no contiene un constructor que tome 0 argumentos

Si desea que una clase tenga un constructor sin parámetros y un constructor que toma un parámetro, puede hacerlo implementando explícitamente ambos constructores.

```
public class Animal
{

    public Animal() {} //Equivalent to a default constructor.
    public Animal(string name) {}
}
```

El compilador no podrá generar un constructor predeterminado si la clase extiende a otra clase que no tiene un constructor sin parámetros. Por ejemplo, si tuviéramos una clase `Creature` :

```
public class Creature
{
    public Creature(Genus genus) {}
}
```

entonces `Animal` definido como `class Animal : Creature {}` no compilaría.

## Llamando a un constructor desde otro constructor

```
public class Animal
{
    public string Name { get; set; }

    public Animal() : this("Dog")
    {
    }

    public Animal(string name)
    {
        Name = name;
    }
}
```

```
var dog = new Animal();      // dog.Name will be set to "Dog" by default.
var cat = new Animal("Cat"); // cat.Name is "Cat", the empty constructor is not called.
```

## Constructor estático

Un constructor estático se llama la primera vez que se inicializa cualquier miembro de un tipo, se llama un miembro de clase estática o un método estático. El constructor estático es seguro para subprocesos. Un constructor estático se usa comúnmente para:

- Inicialice el estado estático, es decir, el estado que se comparte en diferentes instancias de la misma clase.
- Crear un singleton

**Ejemplo:**

```
class Animal
{
    // * A static constructor is executed only once,
    //   when a class is first accessed.
    // * A static constructor cannot have any access modifiers
    // * A static constructor cannot have any parameters
    static Animal()
    {
        Console.WriteLine("Animal initialized");
    }

    // Instance constructor, this is executed every time the class is created
    public Animal()
    {
        Console.WriteLine("Animal created");
    }

    public static void Yawn()
    {
        Console.WriteLine("Yawn!");
    }
}

var turtle = new Animal();
var giraffe = new Animal();
```

**Salida:**

Animal inicializado
Animal creado
Animal creado

Ver demostración

Si la primera llamada es a un método estático, el constructor estático se invoca sin el constructor de instancia. Esto está bien, porque el método estático no puede acceder al estado de la instancia de todos modos.

```
Animal.Yawn();
```

Esto dará como resultado:

> Animal inicializado
> ¡Bostezo!

Vea también Excepciones en constructores estáticos y Constructores estáticos genéricos .

Ejemplo de Singleton:

```
public class SessionManager
{
    public static SessionManager Instance;

    static SessionManager()
    {
        Instance = new SessionManager();
    }
}
```

## Llamando al constructor de la clase base

Se llama a un constructor de una clase base antes de que se ejecute un constructor de una clase derivada. Por ejemplo, si `Mammal` extiende `Animal` , entonces el código contenido en el constructor de `Animal` se llama primero cuando se crea una instancia de un `Mammal` .

Si una clase derivada no especifica explícitamente a qué constructor de la clase base se debe llamar, el compilador asume el constructor sin parámetros.

```
public class Animal
{
    public Animal() { Console.WriteLine("An unknown animal gets born."); }
    public Animal(string name) { Console.WriteLine(name + " gets born"); }
}

public class Mammal : Animal
{
    public Mammal(string name)
    {
        Console.WriteLine(name + " is a mammal.");
    }
}
```

En este caso, se imprimirá una instancia de un `Mammal` llamando al `new Mammal("George the Cat")`

> Nace un animal desconocido.
> George el gato es un mamífero.

Ver demostración

La llamada a un constructor diferente de la clase base se realiza colocando `: base(args)` entre la firma del constructor y su cuerpo:

```
public class Mammal : Animal
{
    public Mammal(string name) : base(name)
    {
        Console.WriteLine(name + " is a mammal.");
    }
}
```

Llamando `new Mammal("George the Cat")` ahora se imprimirá:

> George el gato nace
> George el gato es un mamífero.

Ver demostración

## Finalizadores en clases derivadas.

Cuando se finaliza un gráfico de objetos, el orden es el inverso de la construcción. Por ejemplo, el supertipo se finaliza antes que el tipo base como lo demuestra el siguiente código:

```
class TheBaseClass
{
    ~TheBaseClass()
    {
        Console.WriteLine("Base class finalized!");
    }
}

class TheDerivedClass : TheBaseClass
{
    ~TheDerivedClass()
    {
        Console.WriteLine("Derived class finalized!");
    }
}

//Don't assign to a variable
//to make the object unreachable
new TheDerivedClass();

//Just to make the example work;
//this is otherwise NOT recommended!
GC.Collect();

//Derived class finalized!
//Base class finalized!
```

## Patrón de constructor Singleton

```
public class SingletonClass
{
    public static SingletonClass Instance { get; } = new SingletonClass();

    private SingletonClass()
    {
```

```
        // Put custom constructor code here
    }
}
```

Debido a que el constructor es privado, no se pueden crear nuevas instancias de `SingletonClass` consumiendo código. La única forma de acceder a la instancia única de `SingletonClass` es mediante el uso de la propiedad estática `SingletonClass.Instance` .

La propiedad de `Instance` es asignada por un constructor estático que genera el compilador de C #. El tiempo de ejecución de .NET garantiza que el constructor estático se ejecute como máximo una vez y se ejecute antes de que se lea la `Instance` primera vez. Por lo tanto, todas las preocupaciones de sincronización e inicialización se llevan a cabo por el tiempo de ejecución.

Tenga en cuenta que si el constructor estático falla, la clase `Singleton` quedará permanentemente inutilizable durante la vida útil del dominio de aplicación.

Además, no se garantiza que el constructor estático se ejecute en el momento del primer acceso de la `Instance` . Más bien, se ejecutará *en algún momento antes de eso* . Esto hace que el momento en el que se produce la inicialización no sea determinista. En casos prácticos, el JIT a menudo llama al constructor estático durante la *compilación* (no la ejecución) de un método que hace referencia a la `Instance` . Esta es una optimización de rendimiento.

Vea la página de Implementaciones de Singleton para otras formas de implementar el patrón de singleton.

### Obligando a un constructor estático a ser llamado

Mientras que a los constructores estáticos siempre se les llama antes del primer uso de un tipo, a veces es útil poder forzarlos a ser llamados y la clase `RuntimeHelpers` proporciona un ayudante para ello:

```
using System.Runtime.CompilerServices;
// ...
RuntimeHelpers.RunClassConstructor(typeof(Foo).TypeHandle);
```

***Nota* :** se ejecutarán todas las inicializaciones estáticas (inicializadores de campos, por ejemplo), no solo el propio constructor.

***Usos potenciales* :** forzar la inicialización durante la pantalla de inicio en una aplicación de interfaz de usuario o asegurar que un constructor estático no falle en una prueba de unidad.

### Llamando a métodos virtuales en el constructor.

A diferencia de C ++ en C #, puede llamar a un método virtual desde el constructor de la clase (OK, también puede hacerlo en C ++, pero el comportamiento es sorprendente al principio). Por ejemplo:

```
abstract class Base
{
```

```
    protected Base()
    {
        _obj = CreateAnother();
    }

    protected virtual AnotherBase CreateAnother()
    {
        return new AnotherBase();
    }

    private readonly AnotherBase _obj;
}

sealed class Derived : Base
{
    public Derived() { }

    protected override AnotherBase CreateAnother()
    {
        return new AnotherDerived();
    }
}

var test = new Derived();
// test._obj is AnotherDerived
```

Si viene de un fondo de C ++, esto es sorprendente, ¡el constructor de clases base ya ve la tabla de métodos virtuales de clases derivadas!

**Tenga cuidado** : la clase derivada aún no se ha inicializado completamente (su constructor se ejecutará después del constructor de la clase base) y esta técnica es peligrosa (también existe una advertencia de StyleCop para esto). Por lo general, esto es considerado como una mala práctica.

## Constructores Estáticos Genéricos

Si el tipo en el que se declara el constructor estático es genérico, se llamará una vez al constructor estático para cada combinación única de argumentos genéricos.

```
class Animal<T>
{
    static Animal()
    {
        Console.WriteLine(typeof(T).FullName);
    }

    public static void Yawn() { }
}

Animal<Object>.Yawn();
Animal<String>.Yawn();
```

Esto dará como resultado:

    Sistema.Objeto

> System.String

Vea también ¿Cómo funcionan los constructores estáticos para tipos genéricos?

## Excepciones en constructores estáticos.

Si un constructor estático lanza una excepción, nunca se reintenta. El tipo no se puede utilizar durante la vida útil del dominio de aplicación. Cualquier uso posterior del tipo generará una `TypeInitializationException` alrededor de la excepción original.

```
public class Animal
{
    static Animal()
    {
        Console.WriteLine("Static ctor");
        throw new Exception();
    }

    public static void Yawn() {}
}

try
{
    Animal.Yawn();
}
catch (Exception e)
{
    Console.WriteLine(e.ToString());
}

try
{
    Animal.Yawn();
}
catch (Exception e)
{
    Console.WriteLine(e.ToString());
}
```

Esto dará como resultado:

> Ctor estático
>
> System.TypeInitializationException: el inicializador de tipo para 'Animal' lanzó una excepción. ---> System.Exception: Excepción del tipo 'System.Exception' fue lanzada.

[...]

> System.TypeInitializationException: el inicializador de tipo para 'Animal' lanzó una excepción. ---> System.Exception: Excepción del tipo 'System.Exception' fue lanzada.

donde puede ver que el constructor real solo se ejecuta una vez, y la excepción se reutiliza.

## Constructor y inicialización de propiedades

¿Se debe ejecutar la asignación del valor de la propiedad *antes* o *después* del constructor de la clase?

```
public class TestClass
{
    public int TestProperty { get; set; } = 2;

    public TestClass()
    {
        if (TestProperty == 1)
        {
            Console.WriteLine("Shall this be executed?");
        }

        if (TestProperty == 2)
        {
            Console.WriteLine("Or shall this be executed");
        }
    }
}

var testInstance = new TestClass() { TestProperty = 1 };
```

En el ejemplo anterior, ¿será el valor `TestProperty 1` en el constructor de la clase o después del constructor de la clase?

---

Asignando valores de propiedad en la creación de la instancia de esta manera:

```
var testInstance = new TestClass() {TestProperty = 1};
```

Se ejecutará ***después de*** ejecutar el constructor. Sin embargo, al inicializar el valor de la propiedad en la propiedad class 'en C # 6.0 de esta forma:

```
public class TestClass
{
    public int TestProperty { get; set; } = 2;

    public TestClass()
    {
    }
}
```

Se hará ***antes de que*** se ejecute el constructor.

---

Combinando los dos conceptos anteriores en un solo ejemplo:

```
public class TestClass
{
    public int TestProperty { get; set; } = 2;

    public TestClass()
    {
        if (TestProperty == 1)
```

```
        {
            Console.WriteLine("Shall this be executed?");
        }

        if (TestProperty == 2)
        {
            Console.WriteLine("Or shall this be executed");
        }
    }
}

static void Main(string[] args)
{
    var testInstance = new TestClass() { TestProperty = 1 };
    Console.WriteLine(testInstance.TestProperty); //resulting in 1
}
```

Resultado final:

```
"Or shall this be executed"
"1"
```

---

**Explicación:**

El valor `TestProperty` primero se asignará como `2` , luego se `TestClass` constructor `TestClass` dará como resultado la impresión de

```
"Or shall this be executed"
```

Y luego, `TestProperty` se asignará como `1` debido a la `new TestClass() { TestProperty = 1 }` , lo que hace que el valor final de `TestProperty` impreso por `Console.WriteLine(testInstance.TestProperty)` sea

```
"1"
```

Lea Constructores y finalizadores en línea: https://riptutorial.com/es/csharp/topic/25/constructores-y-finalizadores

# Capítulo 38: Consultas LINQ

## Introducción

LINQ es un acrónimo que significa **L** angguage **IN** tegrated **Q** uery. Es un concepto que integra un lenguaje de consulta al ofrecer un modelo consistente para trabajar con datos a través de varios tipos de fuentes de datos y formatos; utiliza los mismos patrones de codificación básicos para consultar y transformar datos en documentos XML, bases de datos SQL, conjuntos de datos ADO.NET, colecciones .NET y cualquier otro formato para el que esté disponible un proveedor LINQ.

## Sintaxis

- Sintaxis de consulta:
    - desde <rango variable> en <colección>
    - [desde <variable de rango> en <colección>, ...]
    - <filtro, unión, agrupación, operadores agregados, ...> <expresión lambda>
    - <seleccione o agrupe el operador> <formule el resultado>
- Sintaxis del método:
    - Enumerable.Aggregate (func)
    - Enumerable. Agregado (semilla, func)
    - Enumerable.Aggregate (seed, func, resultSelector)
    - Enumerable.Todo (predicado)
    - Enumerable.Any ()
    - Enumerable.Any (predicado)
    - Enumerable.AsEnumerable ()
    - Enumerable.Average ()
    - Enumerable.Average (selector)
    - Enumerable.Cast <Result> ()
    - Enumerable.Concat (segundo)
    - Enumerable.Contains (valor)
    - Enumerable.Contains (valor, comparador)
    - Enumerable. Cuenta ()
    - Enumerable. Cuenta (predicado)
    - Enumerable.DefaultIfEmpty ()
    - Enumerable.DefaultIfEmpty (defaultValue)
    - Enumerable.Distinto ()
    - Enumerable.Distinto (comparador)
    - Enumerable.ElementAt (index)
    - Enumerable.ElementAtOrDefault (índice)
    - Enumerable.Empty ()
    - Enumerable. Excepto (segundo)

- Enumerable. Excepto (segundo, comparador)
- Enumerable.Primera ()
- Enumerable.Primero (predicado)
- Enumerable.FirstOrDefault ()
- Enumerable.FirstOrDefault (predicado)
- Enumerable.GroupBy (keySelector)
- Enumerable.GroupBy (keySelector, resultSelector)
- Enumerable.GroupBy (keySelector, elementSelector)
- Enumerable.GroupBy (keySelector, comparer)
- Enumerable.GroupBy (keySelector, resultSelector, comparer)
- Enumerable.GroupBy (keySelector, elementSelector, resultSelector)
- Enumerable.GroupBy (keySelector, elementSelector, comparer)
- Enumerable.GroupBy (keySelector, elementSelector, resultSelector, comparer)
- Enumerable.Intersect (segundo)
- Enumerable.Intersect (segundo, comparador)
- Enumerable.Join (inner, outerKeySelector, innerKeySelector, resultSelector)
- Enumerable.Join (inner, outerKeySelector, innerKeySelector, resultSelector, comparer)
- Enumerable.Last ()
- Enumerable.Last (predicado)
- Enumerable.LastOrDefault ()
- Enumerable.LastOrDefault (predicado)
- Enumerable.LongCount ()
- Enumerable.LongCount (predicado)
- Enumerable.Max ()
- Enumerable.Max (selector)
- Enumerable.Min ()
- Enumerable.Min (selector)
- Enumerable.OfType <TResult> ()
- Enumerable.OrderBy (keySelector)
- Enumerable.OrderBy (keySelector, comparer)
- Enumerable.OrderByDescending (keySelector)
- Enumerable.OrderByDescending (keySelector, comparer)
- Enumerable. Rango (inicio, conteo)
- Enumerable.Repetir (elemento, contar)
- Enumerable.Reverse ()
- Enumerable.Seleccionar (selector)
- Enumerable.SelectMany (selector)
- Enumerable.SelectMany (collectionSelector, resultSelector)
- Enumerable.SequenceEqual (segundo)
- Enumerable.SequenceEqual (segundo, comparador)
- Enumerable.Single ()
- Enumerable.Single (predicado)
- Enumerable.SingleOrDefault ()
- Enumerable.SingleOrDefault (predicado)
- Enumerable.Skip (contar)
- Enumerable.SkipWhile (predicado)

- Enumerable.Sum ()
- Enumerable.Sum (selector)
- Enumerable. Tomar (contar)
- Enumerable.TakeWhile (predicado)
- orderEnumerable.ThenBy (keySelector)
- orderEnumerable.ThenBy (keySelector, comparer)
- orderEnumerable.ThenByDescending (keySelector)
- orderEnumerable.ThenByDescending (keySelector, comparer)
- Enumerable.ToArray ()
- Enumerable.ToDictionary (keySelector)
- Enumerable.ToDictionary (keySelector, elementSelector)
- Enumerable.ToDictionary (keySelector, comparer)
- Enumerable.ToDictionary (keySelector, elementSelector, comparer)
- Enumerable.ToList ()
- Enumerable.ToLookup (keySelector)
- Enumerable.ToLookup (keySelector, elementSelector)
- Enumerable.ToLookup (keySelector, comparer)
- Enumerable.ToLookup (keySelector, elementSelector, comparer)
- Enumerable.Union (segundo)
- Enumerable.Union (segundo, comparador)
- Enumerable.Donde (predicado)
- Enumerable.Zip (segundo, resultSelector)

# Observaciones

Para usar las consultas LINQ necesita importar `System.Linq` .

La sintaxis del método es más potente y flexible, pero la sintaxis de la consulta puede ser más simple y más familiar. Todas las consultas escritas en la sintaxis de consulta son traducidas a la sintaxis funcional por el compilador, por lo que el rendimiento es el mismo.

Los objetos de consulta no se evalúan hasta que se usan, por lo que se pueden cambiar o agregar sin una penalización de rendimiento.

# Examples

### Dónde

Devuelve un subconjunto de elementos cuyo verdadero predicado es verdadero para ellos.

```
List<string> trees = new List<string>{ "Oak", "Birch", "Beech", "Elm", "Hazel", "Maple" };
```

# Sintaxis del método

```
// Select all trees with name of length 3
```

```
var shortTrees = trees.Where(tree => tree.Length == 3); // Oak, Elm
```

# Sintaxis de consulta

```
var shortTrees = from tree in trees
                 where tree.Length == 3
                 select tree; // Oak, Elm
```

## Seleccionar - Transformar elementos

Seleccionar le permite aplicar una transformación a cada elemento en cualquier estructura de datos que implemente IEnumerable.

Obteniendo el primer carácter de cada cadena en la siguiente lista:

```
List<String> trees = new List<String>{ "Oak", "Birch", "Beech", "Elm", "Hazel", "Maple" };
```

Usando sintaxis regular (lambda)

```
//The below select stament transforms each element in tree into its first character.
IEnumerable<String> initials = trees.Select(tree => tree.Substring(0, 1));
foreach (String initial in initials) {
    System.Console.WriteLine(initial);
}
```

**Salida:**

O
segundo
segundo
mi
H
METRO

Demo en vivo en .NET Fiddle

Usando la sintaxis de consulta LINQ

```
initials = from tree in trees
           select tree.Substring(0, 1);
```

## Métodos de encadenamiento

Muchas funciones de LINQ operan en un `IEnumerable<TSource>` y también devuelven en `IEnumerable<TResult>` . Los parámetros de tipo `TSource` y `TResult` pueden o no referirse al mismo tipo, según el método en cuestión y las funciones que se le pasen.

Algunos ejemplos de esto son

```
public static IEnumerable<TResult> Select<TSource, TResult>(
    this IEnumerable<TSource> source,
    Func<TSource, TResult> selector
)

public static IEnumerable<TSource> Where<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, int, bool> predicate
)

public static IOrderedEnumerable<TSource> OrderBy<TSource, TKey>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector
)
```

Mientras que algunos métodos de encadenamiento pueden requerir que se trabaje un conjunto completo antes de continuar, LINQ aprovecha la ejecución diferida utilizando el rendimiento **MSDN** que crea un Enumerable y un Enumerador detrás de la escena. El proceso de encadenamiento en LINQ consiste esencialmente en construir un enumerable (iterador) para el conjunto original, que se difiere, hasta que se materialice al enumerar el enumerable .

Esto permite que estas funciones se encadenen con fluidez en el **wiki** , donde una función puede actuar directamente sobre el resultado de otra. Este estilo de código se puede usar para realizar muchas operaciones basadas en secuencias en una sola declaración.

Por ejemplo, es posible combinar `Select` , `Where` y `OrderBy` para transformar, filtrar y ordenar una secuencia en una sola declaración.

```
var someNumbers = { 4, 3, 2, 1 };

var processed = someNumbers
        .Select(n => n * 2)    // Multiply each number by 2
        .Where(n => n != 6)    // Keep all the results, except for 6
        .OrderBy(n => n);      // Sort in ascending order
```

**Salida:**

2
4
8

Demo en vivo en .NET Fiddle

Cualquier función que extienda y devuelva el tipo genérico `IEnumerable<T>` se puede usar como cláusulas encadenadas en una sola declaración. Este estilo de programación fluida es potente, y debe tenerse en cuenta al crear sus propios métodos de extensión .

## Alcance y repetición

Los métodos estáticos de `Range` y `Repeat` en `Enumerable` se pueden usar para generar secuencias simples.

# Distancia

`Enumerable.Range()` genera una secuencia de enteros dado un valor de inicio y un conteo.

```
// Generate a collection containing the numbers 1-100 ([1, 2, 3, ..., 98, 99, 100])
var range = Enumerable.Range(1,100);
```

Demo en vivo en .NET Fiddle

# Repetir

`Enumerable.Repeat()` genera una secuencia de elementos repetidos dado un elemento y el número de repeticiones requeridas.

```
// Generate a collection containing "a", three times (["a","a","a"])
var repeatedValues = Enumerable.Repeat("a", 3);
```

Demo en vivo en .NET Fiddle

## Omitir y tomar

El método Omitir devuelve una colección que excluye un número de elementos desde el principio de la colección de origen. El número de elementos excluidos es el número dado como argumento. Si hay menos elementos en la colección que los especificados en el argumento, se devuelve una colección vacía.

El método Take devuelve una colección que contiene una serie de elementos desde el principio de la colección de origen. El número de elementos incluidos es el número dado como argumento. Si hay menos elementos en la colección que los especificados en el argumento, la colección devuelta contendrá los mismos elementos que la colección de origen.

```
var values = new [] { 5, 4, 3, 2, 1 };

var skipTwo        = values.Skip(2);         // { 3, 2, 1 }
var takeThree      = values.Take(3);         // { 5, 4, 3 }
var skipOneTakeTwo = values.Skip(1).Take(2); // { 4, 3 }
var takeZero       = values.Take(0);         // An IEnumerable<int> with 0 items
```

Demo en vivo en .NET Fiddle

**Omitir y Tomar** se usan juntos para paginar los resultados, por ejemplo:

```
IEnumerable<T> GetPage<T>(IEnumerable<T> collection, int pageNumber, int resultsPerPage) {
    int startIndex = (pageNumber - 1) * resultsPerPage;
    return collection.Skip(startIndex).Take(resultsPerPage);
}
```

**Advertencia:** LINQ to Entities solo admite Omitir en consultas ordenadas . Si intenta

utilizar Omitir sin pedir, obtendrá una **excepción NotSupportedException** con el mensaje "El método 'Omitir' solo se admite para entradas ordenadas en LINQ a entidades. El método 'OrderBy' debe llamarse antes que el método 'Omitir'".

### Primero, FirstOrDefault, Last, LastOrDefault, Single y SingleOrDefault

Los seis métodos devuelven un solo valor del tipo de secuencia, y pueden llamarse con o sin un predicado.

Dependiendo de la cantidad de elementos que coincidan con el `predicate` o, si no se proporciona un `predicate` , la cantidad de elementos en la secuencia de origen, se comportan de la siguiente manera:

## Primero()

- Devuelve el primer elemento de una secuencia, o el primer elemento que coincide con el `predicate` proporcionado.
- Si la secuencia no contiene elementos, se `InvalidOperationException` una `InvalidOperationException` con el mensaje: "La secuencia no contiene elementos".
- Si la secuencia no contiene elementos que coincidan con el `predicate` proporcionado, se lanza una `InvalidOperationException` con el mensaje "La secuencia no contiene ningún elemento coincidente".

**Ejemplo**

```
// Returns "a":
new[] { "a" }.First();

// Returns "a":
new[] { "a", "b" }.First();

// Returns "b":
new[] { "a", "b" }.First(x => x.Equals("b"));

// Returns "ba":
new[] { "ba", "be" }.First(x => x.Contains("b"));

// Throws InvalidOperationException:
new[] { "ca", "ce" }.First(x => x.Contains("b"));

// Throws InvalidOperationException:
new string[0].First();
```

Demo en vivo en .NET Fiddle

## FirstOrDefault ()

- Devuelve el primer elemento de una secuencia, o el primer elemento que coincide con el `predicate` proporcionado.

- Si la secuencia no contiene elementos, o no hay elementos que coincidan con el `predicate` proporcionado, devuelve el valor predeterminado del tipo de secuencia utilizando el `default(T)` .

**Ejemplo**

```
// Returns "a":
new[] { "a" }.FirstOrDefault();

// Returns "a":
new[] { "a", "b" }.FirstOrDefault();

// Returns "b":
new[] { "a", "b" }.FirstOrDefault(x => x.Equals("b"));

// Returns "ba":
new[] { "ba", "be" }.FirstOrDefault(x => x.Contains("b"));

// Returns null:
new[] { "ca", "ce" }.FirstOrDefault(x => x.Contains("b"));

// Returns null:
new string[0].FirstOrDefault();
```

Demo en vivo en .NET Fiddle

# Último()

- Devuelve el último elemento de una secuencia, o el último elemento que coincide con el `predicate` proporcionado.
- Si la secuencia no contiene elementos, se lanza una `InvalidOperationException` con el mensaje "La secuencia no contiene elementos".
- Si la secuencia no contiene elementos que coincidan con el `predicate` proporcionado, se lanza una `InvalidOperationException` con el mensaje "La secuencia no contiene ningún elemento coincidente".

**Ejemplo**

```
// Returns "a":
new[] { "a" }.Last();

// Returns "b":
new[] { "a", "b" }.Last();

// Returns "a":
new[] { "a", "b" }.Last(x => x.Equals("a"));

// Returns "be":
new[] { "ba", "be" }.Last(x => x.Contains("b"));

// Throws InvalidOperationException:
new[] { "ca", "ce" }.Last(x => x.Contains("b"));
```

```
// Throws InvalidOperationException:
new string[0].Last();
```

# LastOrDefault ()

- Devuelve el último elemento de una secuencia, o el último elemento que coincide con el `predicate` proporcionado.
- Si la secuencia no contiene elementos, o no hay elementos que coincidan con el `predicate` proporcionado, devuelve el valor predeterminado del tipo de secuencia utilizando el `default(T)` .

**Ejemplo**

```
// Returns "a":
new[] { "a" }.LastOrDefault();

// Returns "b":
new[] { "a", "b" }.LastOrDefault();

// Returns "a":
new[] { "a", "b" }.LastOrDefault(x => x.Equals("a"));

 // Returns "be":
new[] { "ba", "be" }.LastOrDefault(x => x.Contains("b"));

// Returns null:
new[] { "ca", "ce" }.LastOrDefault(x => x.Contains("b"));

// Returns null:
new string[0].LastOrDefault();
```

# Soltero()

- Si la secuencia contiene exactamente un elemento, o exactamente un elemento que coincide con el `predicate` proporcionado, se devuelve ese elemento.
- Si la secuencia no contiene elementos, o no hay elementos que coincidan con el `predicate` proporcionado, se lanza una `InvalidOperationException` con el mensaje "La secuencia no contiene elementos".
- Si la secuencia contiene más de un elemento, o más de un elemento que coincide con el `predicate` proporcionado, se `InvalidOperationException` una `InvalidOperationException` con el mensaje "La secuencia contiene más de un elemento".
- **Nota:** para evaluar si la secuencia contiene exactamente un elemento, como máximo se deben enumerar dos elementos.

**Ejemplo**

```
// Returns "a":
new[] { "a" }.Single();
```

```
// Throws InvalidOperationException because sequence contains more than one element:
new[] { "a", "b" }.Single();

// Returns "b":
new[] { "a", "b" }.Single(x => x.Equals("b"));

// Throws InvalidOperationException:
new[] { "a", "b" }.Single(x => x.Equals("c"));

// Throws InvalidOperationException:
new string[0].Single();

// Throws InvalidOperationException because sequence contains more than one element:
new[] { "a", "a" }.Single();
```

# SingleOrDefault ()

- Si la secuencia contiene exactamente un elemento, o exactamente un elemento que coincide con el `predicate` proporcionado, se devuelve ese elemento.
- Si la secuencia no contiene elementos, o no hay elementos que coincidan con el `predicate` proporcionado, se devuelve el `default(T)` .
- Si la secuencia contiene más de un elemento, o más de un elemento que coincide con el `predicate` proporcionado, se `InvalidOperationException` una `InvalidOperationException` con el mensaje "La secuencia contiene más de un elemento".
- Si la secuencia no contiene elementos que coincidan con el `predicate` proporcionado, devuelve el valor predeterminado del tipo de secuencia utilizando el `default(T)` .
- **Nota:** para evaluar si la secuencia contiene exactamente un elemento, como máximo se deben enumerar dos elementos.

### Ejemplo

```
// Returns "a":
new[] { "a" }.SingleOrDefault();

// returns "a"
new[] { "a", "b" }.SingleOrDefault(x => x == "a");

// Returns null:
new[] { "a", "b" }.SingleOrDefault(x => x == "c");

// Throws InvalidOperationException:
new[] { "a", "a" }.SingleOrDefault(x => x == "a");

// Throws InvalidOperationException:
new[] { "a", "b" }.SingleOrDefault();

// Returns null:
new string[0].SingleOrDefault();
```

# Recomendaciones

- Aunque puede usar `FirstOrDefault` , `LastOrDefault` o `SingleOrDefault` para verificar si una secuencia contiene algún elemento, `Any` o `Count` son más confiables. Esto se debe a que el valor de retorno del valor `default(T)` de uno de estos tres métodos no prueba que la secuencia esté vacía, ya que el valor del primer / último / único elemento de la secuencia podría ser también `default(T)`

- Decida qué métodos se adaptan mejor al propósito de su código. Por ejemplo, use `Single` solo si debe asegurarse de que haya un solo elemento en la colección que coincida con su predicado; de lo contrario, use `First` ; como `Single` lanza una excepción si la secuencia tiene más de un elemento coincidente. Esto, por supuesto, se aplica también a las contrapartes "* OrDefault".

- Respecto a la eficiencia: aunque a menudo es apropiado asegurarse de que haya solo un elemento ( `Single` ) o, solo uno o cero ( `SingleOrDefault` ) devueltos por una consulta, ambos métodos requieren más, y con frecuencia la totalidad, de la colección. para ser examinado para asegurar que no haya una segunda coincidencia con la consulta. Esto es diferente del comportamiento del `First` método, por ejemplo, que puede satisfacerse después de encontrar la primera coincidencia.

## Excepto

El método Except devuelve el conjunto de elementos que están contenidos en la primera colección, pero no están contenidos en la segunda. El `IEqualityComparer` predeterminado se utiliza para comparar los elementos dentro de los dos conjuntos. Hay una sobrecarga que acepta un `IEqualityComparer` como argumento.

**Ejemplo:**

```
int[] first = { 1, 2, 3, 4 };
int[] second = { 0, 2, 3, 5 };

IEnumerable<int> inFirstButNotInSecond = first.Except(second);
// inFirstButNotInSecond = { 1, 4 }
```

**Salida:**

1
4

Demo en vivo en .NET Fiddle

En este caso. `.Except(second)` excluye los elementos contenidos en la matriz `second` , es decir, 2 y 3 (0 y 5 no están contenidos en la `first` matriz y se omiten).

Tenga en cuenta que `Except` implica `Distinct` (es decir, elimina elementos repetidos). Por ejemplo:

```
int[] third = { 1, 1, 1, 2, 3, 4 };

IEnumerable<int> inThirdButNotInSecond = third.Except(second);
// inThirdButNotInSecond = { 1, 4 }
```

**Salida:**

1
4

Demo en vivo en .NET Fiddle

En este caso, los elementos 1 y 4 se devuelven solo una vez.

---

Implementar `IEquatable` o proporcionar la función un `IEqualityComparer` permitirá usar un método diferente para comparar los elementos. Tenga en cuenta que el método `GetHashCode` también debe anularse para que devuelva un código hash idéntico para el `object` que sea idéntico según la implementación de `IEquatable` .

***Ejemplo con IEquatable:***

```
class Holiday : IEquatable<Holiday>
{
    public string Name { get; set; }

    public bool Equals(Holiday other)
    {
        return Name == other.Name;
    }

    // GetHashCode must return true whenever Equals returns true.
    public override int GetHashCode()
    {
        //Get hash code for the Name field if it is not null.
        return Name?.GetHashCode() ?? 0;
    }
}

public class Program
{
    public static void Main()
    {
        List<Holiday> holidayDifference = new List<Holiday>();

        List<Holiday> remoteHolidays = new List<Holiday>
        {
            new Holiday { Name = "Xmas" },
            new Holiday { Name = "Hanukkah" },
            new Holiday { Name = "Ramadan" }
        };

        List<Holiday> localHolidays = new List<Holiday>
        {
            new Holiday { Name = "Xmas" },
            new Holiday { Name = "Ramadan" }
        };

        holidayDifference = remoteHolidays
            .Except(localHolidays)
            .ToList();

        holidayDifference.ForEach(x => Console.WriteLine(x.Name));
```

```
    }
}
```

Salida:

```
Hanukkah
```

Demo en vivo en .NET Fiddle

## SelectMany: aplanando una secuencia de secuencias

```
var sequenceOfSequences = new [] { new [] { 1, 2, 3 }, new [] { 4, 5 }, new [] { 6 } };
var sequence = sequenceOfSequences.SelectMany(x => x);
// returns { 1, 2, 3, 4, 5, 6 }
```

Utilice `SelectMany()` si tiene, o si está creando una secuencia de secuencias, pero desea que el resultado sea una secuencia larga.

En la sintaxis de consulta LINQ:

```
var sequence = from subSequence in sequenceOfSequences
               from item in subSequence
               select item;
```

Si tiene una colección de colecciones y le gustaría poder trabajar en los datos de la colección principal y secundaria al mismo tiempo, también es posible con `SelectMany` .

Definamos clases simples

```
public class BlogPost
{
    public int Id { get; set; }
    public string Content { get; set; }
    public List<Comment> Comments { get; set; }
}

public class Comment
{
    public int Id { get; set; }
    public string Content { get; set; }
}
```

Supongamos que tenemos la siguiente colección.

```
List<BlogPost> posts = new List<BlogPost>()
{
    new BlogPost()
    {
        Id = 1,
        Comments = new List<Comment>()
        {
            new Comment()
            {
```

```
                Id = 1,
                Content = "It's really great!",
            },
            new Comment()
            {
                Id = 2,
                Content = "Cool post!"
            }
        }
    },
    new BlogPost()
    {
        Id = 2,
        Comments = new List<Comment>()
        {
            new Comment()
            {
                Id = 3,
                Content = "I don't think you're right",
            },
            new Comment()
            {
                Id = 4,
                Content = "This post is a complete nonsense"
            }
        }
    }
};
```

Ahora queremos seleccionar el `Content` comentarios junto con la `Id` de `BlogPost` asociada con este comentario. Para hacerlo, podemos usar la sobrecarga de `SelectMany` apropiada.

```
var commentsWithIds = posts.SelectMany(p => p.Comments, (post, comment) => new { PostId =
post.Id, CommentContent = comment.Content });
```

Nuestros `commentsWithIds` ve así

```
{
    PostId = 1,
    CommentContent = "It's really great!"
},
{
    PostId = 1,
    CommentContent = "Cool post!"
},
{
    PostId = 2,
    CommentContent = "I don't think you're right"
},
{
    PostId = 2,
    CommentContent = "This post is a complete nonsense"
}
```

## SelectMany

El método linq de SelectMany 'aplana' un `IEnumerable<IEnumerable<T>>` en un `IEnumerable<T>` .

Todos los elementos T dentro de las instancias `IEnumerable` contenidas en la fuente `IEnumerable` se combinarán en una sola `IEnumerable` .

```
var words = new [] { "a,b,c", "d,e", "f" };
var splitAndCombine = words.SelectMany(x => x.Split(','));
// returns { "a", "b", "c", "d", "e", "f" }
```

Si utiliza una función de selección que convierte los elementos de entrada en secuencias, el resultado serán los elementos de esas secuencias que se devuelven uno por uno.

Tenga en cuenta que, a diferencia de `Select()` , el número de elementos en la salida no tiene que ser el mismo que el de la entrada.

**Más ejemplo del mundo real.**

```
class School
{
    public Student[] Students { get; set; }
}

class Student
{
    public string Name { get; set; }
}

var schools = new [] {
    new School(){ Students = new [] { new Student { Name="Bob"}, new Student { Name="Jack"}
}},
    new School(){ Students = new [] { new Student { Name="Jim"}, new Student { Name="John"} }}
};

var allStudents = schools.SelectMany(s=> s.Students);

foreach(var student in allStudents)
{
    Console.WriteLine(student.Name);
}
```

Salida:

Mover
Jack
Jim
Juan

Demo en vivo en .NET Fiddle

## Todos

`All` se utiliza para verificar, si todos los elementos de una colección coinciden con una condición o no.
*ver también: .cualquier*

# 1. Parámetro vacío

**Todos** : no está permitido ser usado con un parámetro vacío.

# 2. Expresión lambda como parámetro

**Todos** : Devuelve `true` si todos los elementos de la colección satisfacen la expresión lambda y, de lo contrario, `false` :

```
var numbers = new List<int>(){ 1, 2, 3, 4, 5};
bool result = numbers.All(i => i < 10); // true
bool result = numbers.All(i => i >= 3); // false
```

# 3. Colección vacía

**Todos** : Devuelve `true` si la colección está vacía y se suministra una expresión lambda:

```
var numbers = new List<int>();
bool result = numbers.All(i => i >= 0); // true
```

**Nota:** `All` detendrán la iteración de la colección tan pronto como encuentre un elemento que **no** coincide con la condición. Esto significa que la colección no necesariamente será enumerada completamente; solo se enumerará lo suficiente como para encontrar el primer elemento que **no coincide con** la condición.

### Consulta la colección por tipo / cast elementos para escribir

```
interface IFoo { }
class Foo : IFoo { }
class Bar : IFoo { }
```

---

```
var item0 = new Foo();
var item1 = new Foo();
var item2 = new Bar();
var item3 = new Bar();
var collection = new IFoo[] { item0, item1, item2, item3 };
```

Usando `OfType`

```
var foos = collection.OfType<Foo>(); // result: IEnumerable<Foo> with item0 and item1
var bars = collection.OfType<Bar>(); // result: IEnumerable<Bar> item item2 and item3
var foosAndBars = collection.OfType<IFoo>(); // result: IEnumerable<IFoo> with all four items
```

Usando `Where`

```
var foos = collection.Where(item => item is Foo); // result: IEnumerable<IFoo> with item0 and
item1
```

```
var bars = collection.Where(item => item is Bar); // result: IEnumerable<IFoo> with item2 and
item3
```

Utilizando `Cast`

```
var bars = collection.Cast<Bar>();                // throws InvalidCastException on the 1st
item
var foos = collection.Cast<Foo>();                // throws InvalidCastException on the 3rd
item
var foosAndBars = collection.Cast<IFoo>();        // OK
```

### Unión

Fusiona dos colecciones para crear una colección distinta utilizando el comparador de igualdad predeterminado

```
int[] numbers1 = { 1, 2, 3 };
int[] numbers2 = { 2, 3, 4, 5 };

var allElement = numbers1.Union(numbers2);   // AllElement now contains 1,2,3,4,5
```

Demo en vivo en .NET Fiddle

### Se une

Las combinaciones se utilizan para combinar diferentes listas o tablas que contienen datos a través de una clave común.

Al igual que en SQL, los siguientes tipos de combinaciones son compatibles con LINQ: **Inner, Left, Right, Cross** y **Full Outer** Joins.

Las siguientes dos listas se utilizan en los siguientes ejemplos:

```
var first = new List<string>(){ "a","b","c"}; // Left data
var second = new List<string>(){ "a", "c", "d"}; // Right data
```

## (Unir internamente

```
var result = from f in first
             join s in second on f equals s
             select new { f, s };

var result = first.Join(second,
                        f => f,
                        s => s,
                        (f, s) => new { f, s });

// Result: {"a","a"}
//         {"c","c"}
```

## Izquierda combinación externa

```
var leftOuterJoin = from f in first
                    join s in second on f equals s into temp
                    from t in temp.DefaultIfEmpty()
                    select new { First = f, Second = t};

// Or can also do:
var leftOuterJoin = from f in first
                    from s in second.Where(x => x == f).DefaultIfEmpty()
                    select new { First = f, Second = s};

// Result: {"a","a"}
//         {"b", null}
//         {"c","c"}


// Left outer join method syntax
var leftOuterJoinFluentSyntax = first.GroupJoin(second,
                                      f => f,
                                      s => s,
                                      (f, s) => new { First = f, Second = s })
                                    .SelectMany(temp => temp.Second.DefaultIfEmpty(),
                                      (f, s) => new { First = f.First, Second = s });
```

## Unión externa derecha

```
var rightOuterJoin = from s in second
                     join f in first on s equals f into temp
                     from t in temp.DefaultIfEmpty()
                     select new {First=t,Second=s};

// Result: {"a","a"}
//         {"c","c"}
//         {null,"d"}
```

## Cruzar

```
var CrossJoin = from f in first
                from s in second
                select new { f, s };

// Result: {"a","a"}
//         {"a","c"}
//         {"a","d"}
//         {"b","a"}
//         {"b","c"}
//         {"b","d"}
//         {"c","a"}
//         {"c","c"}
//         {"c","d"}
```

## Unión externa completa

```
var fullOuterjoin = leftOuterJoin.Union(rightOuterJoin);

// Result: {"a","a"}
//         {"b", null}
//         {"c","c"}
//         {null,"d"}
```

# Ejemplo practico

Los ejemplos anteriores tienen una estructura de datos simple para que pueda concentrarse en entender las diferentes uniones LINQ técnicamente, pero en el mundo real tendría tablas con columnas a las que necesita unirse.

En el siguiente ejemplo, solo se utiliza una clase de `Region` ; en realidad, uniría dos o más tablas diferentes que contienen la misma clave (en este ejemplo, la `first` y la `second` se unen mediante la `ID` clave común).

**Ejemplo:** Considere la siguiente estructura de datos:

```
public class Region
{
    public Int32 ID;
    public string RegionDescription;

    public Region(Int32 pRegionID, string pRegionDescription=null)
    {
        ID = pRegionID; RegionDescription = pRegionDescription;
    }
}
```

Ahora prepare los datos (es decir, rellene con datos):

```
// Left data
var first = new List<Region>()
                { new Region(1), new Region(3), new Region(4) };
// Right data
var second = new List<Region>()
                {
                   new Region(1, "Eastern"),  new Region(2, "Western"),
                   new Region(3, "Northern"), new Region(4, "Southern")
                };
```

Puede ver que en este ejemplo, en `first` , no contiene ninguna descripción de región, por lo que desea unirse a ellos desde el `second` . Entonces la unión interna se vería como:

```
// do the inner join
var result = from f in first
             join s in second on f.ID equals s.ID
             select new { f.ID, s.RegionDescription };


 // Result: {1,"Eastern"}
 //         {3, Northern}
```

```
//         {4,"Southern"}
```

Este resultado ha creado objetos anónimos sobre la marcha, lo cual está bien, pero ya hemos creado una clase adecuada, por lo que podemos especificarlo: en lugar de `select new { f.ID, s.RegionDescription };` podemos decir `select new Region(f.ID, s.RegionDescription);` , que devolverá los mismos datos pero creará objetos de tipo `Region` , que mantendrán la compatibilidad con los otros objetos.

Demostración en vivo en .NET Fiddle

## Distinto

Devuelve valores únicos de un `IEnumerable` . La singularidad se determina utilizando el comparador de igualdad predeterminado.

```
int[] array = { 1, 2, 3, 4, 2, 5, 3, 1, 2 };

var distinct = array.Distinct();
// distinct = { 1, 2, 3, 4, 5 }
```

Para comparar un tipo de datos personalizado, necesitamos implementar la `IEquatable<T>` y proporcionar los métodos `GetHashCode` y `Equals` para el tipo. O el comparador de igualdad puede ser anulado:

```
class SSNEqualityComparer : IEqualityComparer<Person> {
    public bool Equals(Person a, Person b) => return a.SSN == b.SSN;
    public int GetHashCode(Person p) => p.SSN;
}

List<Person> people;

distinct = people.Distinct(SSNEqualityComparer);
```

## Grupo por uno o varios campos

Asumamos que tenemos algún modelo de película:

```
public class Film {
    public string Title { get; set; }
    public string Category { get; set; }
    public int Year { get; set; }
}
```

Grupo por propiedad de categoría:

```
foreach (var grp in films.GroupBy(f => f.Category)) {
    var groupCategory = grp.Key;
    var numberOfFilmsInCategory = grp.Count();
}
```

Grupo por categoría y año:

```
foreach (var grp in films.GroupBy(f => new { Category = f.Category, Year = f.Year })) {
    var groupCategory = grp.Key.Category;
    var groupYear = grp.Key.Year;
    var numberOfFilmsInCategory = grp.Count();
}
```

## Usando Range con varios métodos Linq

Puede usar la clase Enumerable junto con las consultas de Linq para convertir los bucles en forros de Linq one.

**Seleccione Ejemplo**

Opuesto a hacer esto:

```
var asciiCharacters = new List<char>();
for (var x = 0; x < 256; x++)
{
    asciiCharacters.Add((char)x);
}
```

Puedes hacerlo:

```
var asciiCharacters = Enumerable.Range(0, 256).Select(a => (char) a);
```

**Donde ejemplo**

En este ejemplo, se generarán 100 números e incluso se extraerán

```
var evenNumbers = Enumerable.Range(1, 100).Where(a => a % 2 == 0);
```

## Pedidos de consultas: OrderBy () ThenBy () OrderByDescending () ThenByDescending ()

```
string[] names= { "mark", "steve", "adam" };
```

**Ascendente**

*Sintaxis de consulta*

```
var sortedNames =
    from name in names
    orderby name
    select name;
```

*Sintaxis del método*

```
var sortedNames = names.OrderBy(name => name);
```

sortedNames contiene los nombres en el siguiente orden: "adam", "mark", "steve"

**Descendente**

*Sintaxis de consulta*

```
var sortedNames =
    from name in names
    orderby name descending
    select name;
```

*Sintaxis del método*

```
var sortedNames = names.OrderByDescending(name => name);
```

sortedNames contiene los nombres en el siguiente orden: "steve", "mark", "adam"

**Ordenar por varios campos**

```
Person[] people =
{
    new Person { FirstName = "Steve", LastName = "Collins", Age = 30},
    new Person { FirstName = "Phil" , LastName = "Collins", Age = 28},
    new Person { FirstName = "Adam" , LastName = "Ackerman", Age = 29},
    new Person { FirstName = "Adam" , LastName = "Ackerman", Age = 15}
};
```

*Sintaxis de consulta*

```
var sortedPeople = from person in people
                   orderby person.LastName, person.FirstName, person.Age descending
                   select person;
```

*Sintaxis del método*

```
 sortedPeople = people.OrderBy(person => person.LastName)
                      .ThenBy(person => person.FirstName)
                      .ThenByDescending(person => person.Age);
```

*Resultado*

```
1. Adam Ackerman 29
2. Adam Ackerman 15
3. Phil Collins  28
4. Steve Collins 30
```

## Lo esencial

LINQ es en gran parte beneficioso para consultar colecciones (o matrices).

Por ejemplo, dados los siguientes datos de muestra:

```
var classroom = new Classroom
{
    new Student { Name = "Alice", Grade = 97, HasSnack = true  },
    new Student { Name = "Bob",   Grade = 82, HasSnack = false },
    new Student { Name = "Jimmy", Grade = 71, HasSnack = true  },
    new Student { Name = "Greg",  Grade = 90, HasSnack = false },
    new Student { Name = "Joe",   Grade = 59, HasSnack = false }
}
```

Podemos "consultar" sobre estos datos usando la sintaxis de LINQ. Por ejemplo, para recuperar todos los estudiantes que tienen una merienda hoy:

```
var studentsWithSnacks = from s in classroom.Students
                         where s.HasSnack
                         select s;
```

O, para recuperar estudiantes con una calificación de 90 o más, y solo devolver sus nombres, no el objeto de `Student` completo:

```
var topStudentNames = from s in classroom.Students
                      where s.Grade >= 90
                      select s.Name;
```

La característica LINQ consta de dos sintaxis que realizan las mismas funciones, tienen un rendimiento casi idéntico, pero se escriben de manera muy diferente. La sintaxis en el ejemplo anterior se llama **sintaxis de consulta** . El siguiente ejemplo, sin embargo, ilustra la **sintaxis del método** . Se devolverán los mismos datos que en el ejemplo anterior, pero la forma en que se escribe la consulta es diferente.

```
var topStudentNames = classroom.Students
                              .Where(s => s.Grade >= 90)
                              .Select(s => s.Name);
```

### Agrupar por

GroupBy es una forma fácil de ordenar una colección de elementos `IEnumerable<T>` en grupos distintos.

## Ejemplo simple

En este primer ejemplo, terminamos con dos grupos, elementos pares e impares.

```
List<int> iList = new List<int>() { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
var grouped = iList.GroupBy(x => x % 2 == 0);

//Groups iList into odd [13579] and even[2468] items

foreach(var group in grouped)
{
    foreach (int item in group)
    {
```

```
        Console.Write(item); // 135792468  (first odd then even)
    }
}
```

## Ejemplo más complejo

Tomemos como ejemplo agrupar una lista de personas por edad. Primero, crearemos un objeto Person que tiene dos propiedades, Nombre y Edad.

```
public class Person
{
    public int Age {get; set;}
    public string Name {get; set;}
}
```

Luego creamos nuestra lista de muestra de personas con diferentes nombres y edades.

```
List<Person> people = new List<Person>();
people.Add(new Person{Age = 20, Name = "Mouse"});
people.Add(new Person{Age = 30, Name = "Neo"});
people.Add(new Person{Age = 40, Name = "Morpheus"});
people.Add(new Person{Age = 30, Name = "Trinity"});
people.Add(new Person{Age = 40, Name = "Dozer"});
people.Add(new Person{Age = 40, Name = "Smith"});
```

Luego creamos una consulta LINQ para agrupar nuestra lista de personas por edad.

```
var query = people.GroupBy(x => x.Age);
```

Al hacerlo, podemos ver la edad de cada grupo y tener una lista de cada persona en el grupo.

```
foreach(var result in query)
{
    Console.WriteLine(result.Key);

    foreach(var person in result)
        Console.WriteLine(person.Name);
}
```

Esto resulta en el siguiente resultado:

```
20
Mouse
30
Neo
Trinity
40
Morpheus
Dozer
Smith
```

Puedes jugar con la demostración en vivo en .NET Fiddle

### Alguna

`Any` se utiliza para comprobar si **algún** elemento de una colección coincide con una condición o no.
*vea también: .All , Any y FirstOrDefault: mejores prácticas*

## 1. Parámetro vacío

**Cualquiera** : Devuelve `true` si la colección tiene algún elemento y `false` si la colección está vacía:

```
var numbers = new List<int>();
bool result = numbers.Any(); // false

var numbers = new List<int>(){ 1, 2, 3, 4, 5};
bool result = numbers.Any(); //true
```

## 2. Expresión lambda como parámetro

**Cualquiera** : Devuelve `true` si la colección tiene uno o más elementos que cumplen la condición en la expresión lambda:

```
var arrayOfStrings = new string[] { "a", "b", "c" };
arrayOfStrings.Any(item => item == "a");    // true
arrayOfStrings.Any(item => item == "d");    // false
```

## 3. Colección vacía

**Cualquiera** : Devuelve `false` si la colección está vacía y se suministra una expresión lambda:

```
var numbers = new List<int>();
bool result = numbers.Any(i => i >= 0); // false
```

**Nota:** `Any` detendrá la iteración de la colección tan pronto como encuentre un elemento que coincida con la condición. Esto significa que la colección no necesariamente será enumerada completamente; solo se enumerará lo suficiente como para encontrar el primer elemento que coincida con la condición.

Demo en vivo en .NET Fiddle

### Al diccionario

El método LINQ de `ToDictionary()` se puede usar para generar una colección `Dictionary<TKey, TElement>` basada en una fuente `IEnumerable<T>` .

```
IEnumerable<User> users = GetUsers();
Dictionary<int, User> usersById = users.ToDictionary(x => x.Id);
```

En este ejemplo, el único argumento pasado a `ToDictionary` es de tipo `Func<TSource, TKey>` , que devuelve la clave para cada elemento.

Esta es una forma concisa de realizar la siguiente operación:

```
Dictionary<int, User> usersById = new Dictionary<int User>();
foreach (User u in users)
{
  usersById.Add(u.Id, u);
}
```

También puede pasar un segundo parámetro al método `ToDictionary` , que es de tipo `Func<TSource, TElement>` y devuelve el `Value` que se agregará para cada entrada.

```
IEnumerable<User> users = GetUsers();
Dictionary<int, string> userNamesById = users.ToDictionary(x => x.Id, x => x.Name);
```

También es posible especificar el `IComparer` que se utiliza para comparar valores clave. Esto puede ser útil cuando la clave es una cadena y desea que coincida con mayúsculas y minúsculas.

```
IEnumerable<User> users = GetUsers();
Dictionary<string, User> usersByCaseInsenstiveName = users.ToDictionary(x => x.Name,
StringComparer.InvariantCultureIgnoreCase);

var user1 = usersByCaseInsenstiveName["john"];
var user2 = usersByCaseInsenstiveName["JOHN"];
user1 == user2; // Returns true
```

Nota: el método `ToDictionary` requiere que todas las claves sean únicas, no debe haber claves duplicadas. Si los hay, se lanza una excepción: `ArgumentException: An item with the same key has already been added.` Si tiene un escenario en el que sabe que tendrá varios elementos con la misma clave, entonces es mejor que utilice `ToLookup` .

## Agregar

`Aggregate` Aplica una función de acumulador sobre una secuencia.

```
int[] intList = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = intList.Aggregate((prevSum, current) => prevSum + current);
// sum = 55
```

- En el primer paso `prevSum = 1`
- En la segunda `prevSum = prevSum(at the first step) + 2`
- En el paso i-th `prevSum = prevSum(at the (i-1) step) + i-th element of the array`

```
string[] stringList = { "Hello", "World", "!" };
string joinedString = stringList.Aggregate((prev, current) => prev + " " + current);
// joinedString = "Hello World !"
```

---

Una segunda sobrecarga de `Aggregate` también recibe un parámetro `seed` que es el valor inicial del

acumulador. Esto se puede usar para calcular múltiples condiciones en una colección sin iterarla más de una vez.

```
List<int> items = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };
```

Para la colección de `items` que queremos calcular.

1. El total `.Count`
2. La cantidad de números pares.
3. Recoge cada artículo

Usando `Aggregate` se puede hacer así:

```
var result = items.Aggregate(new { Total = 0, Even = 0, FourthItems = new List<int>() },
                (accumelative,item) =>
                new {
                    Total = accumelative.Total + 1,
                    Even = accumelative.Even + (item % 2 == 0 ? 1 : 0),
                    FourthItems = (accumelative.Total + 1)%4 == 0 ?
                        new List<int>(accumelative.FourthItems) { item } :
                        accumelative.FourthItems
                });
// Result:
// Total = 12
// Even = 6
// FourthItems = [4, 8, 12]
```

*Tenga en cuenta que el uso de un tipo anónimo como semilla tiene que crear una instancia de un nuevo objeto para cada elemento porque las propiedades son de solo lectura. El uso de una clase personalizada puede simplemente asignar la información y ninguna `new` se necesita (sólo cuando se da la primera `seed` de parámetros*

## Definir una variable dentro de una consulta Linq (dejar palabra clave)

Para definir una variable dentro de una expresión linq, puede usar la palabra clave **let** . Esto generalmente se hace para almacenar los resultados de las subconsultas intermedias, por ejemplo:

```
 int[] numbers = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

 var aboveAverages = from number in numbers
                     let average = numbers.Average()
                     let nSquared = Math.Pow(number,2)
                     where nSquared > average
                     select number;

 Console.WriteLine("The average of the numbers is {0}.", numbers.Average());

 foreach (int n in aboveAverages)
 {
   Console.WriteLine("Query result includes number {0} with square of {1}.", n,
Math.Pow(n,2));
 }
```

**Salida:**

El promedio de los números es 4.5.
El resultado de la consulta incluye el número 3 con el cuadrado de 9.
El resultado de la consulta incluye el número 4 con el cuadrado de 16.
El resultado de la consulta incluye el número 5 con el cuadrado de 25.
El resultado de la consulta incluye el número 6 con el cuadrado de 36.
El resultado de la consulta incluye el número 7 con el cuadrado de 49.
El resultado de la consulta incluye el número 8 con el cuadrado de 64.
El resultado de la consulta incluye el número 9 con el cuadrado de 81.

Ver demostración

### SkipWhile

`SkipWhile()` se usa para excluir elementos hasta la primera no coincidencia (esto puede ser contra intuitivo para la mayoría)

```
int[] list = { 42, 42, 6, 6, 6, 42 };
var result = list.SkipWhile(i => i == 42);
// Result: 6, 6, 6, 42
```

### DefaultIfEmpty

DefaultIfEmpty se utiliza para devolver un elemento predeterminado si la secuencia no contiene elementos. Este elemento puede ser el valor predeterminado del tipo o una instancia definida por el usuario de ese tipo. Ejemplo:

```
var chars = new List<string>() { "a", "b", "c", "d" };

chars.DefaultIfEmpty("N/A").FirstOrDefault(); // returns "a";

chars.Where(str => str.Length > 1)
     .DefaultIfEmpty("N/A").FirstOrDefault(); // return "N/A"

chars.Where(str => str.Length > 1)
        .DefaultIfEmpty().First(); // returns null;
```

## Uso en uniones izquierdas :

Con `DefaultIfEmpty` la `DefaultIfEmpty` tradicional de Linq puede devolver un objeto predeterminado si no se encuentra ninguna coincidencia. Actuando así como una unión izquierda de SQL. Ejemplo:

```
var leftSequence = new List<int>() { 99, 100, 5, 20, 102, 105 };
var rightSequence = new List<char>() { 'a', 'b', 'c', 'i', 'd' };

var numbersAsChars = from l in leftSequence
                     join r in rightSequence
                     on l equals (int)r into leftJoin
```

```
                      from result in leftJoin.DefaultIfEmpty('?')
                      select new
                      {
                          Number = l,
                          Character = result
                      };

foreach(var item in numbersAsChars)
{
    Console.WriteLine("Num = {0} ** Char = {1}", item.Number, item.Character);
}

ouput:

Num = 99         Char = c
Num = 100        Char = d
Num = 5          Char = ?
Num = 20         Char = ?
Num = 102        Char = ?
Num = 105        Char = i
```

En el caso de que se `DefaultIfEmpty` un `DefaultIfEmpty` (sin especificar un valor predeterminado) y que resulte, no habrá elementos coincidentes en la secuencia correcta, uno debe asegurarse de que el objeto no sea `null` antes de acceder a sus propiedades. De lo contrario, dará lugar a una `NullReferenceException` . Ejemplo:

```
var leftSequence = new List<int> { 1, 2, 5 };
var rightSequence = new List<dynamic>()
    {
        new { Value = 1 },
        new { Value = 2 },
        new { Value = 3 },
        new { Value = 4 },
    };

var numbersAsChars = (from l in leftSequence
                      join r in rightSequence
                      on l equals r.Value into leftJoin
                      from result in leftJoin.DefaultIfEmpty()
                      select new
                      {
                          Left = l,
                          // 5 will not have a matching object in the right so result
                          // will be equal to null.
                          // To avoid an error use:
                          //    -  C# 6.0 or above - ?.
                          //    -  Under           - result == null ? 0 : result.Value
                          Right = result?.Value
                      }).ToList();
```

## SecuenciaEqual

`SequenceEqual` se usa para comparar dos `IEnumerable<T>` entre sí.

```
int[] a = new int[] {1, 2, 3};
int[] b = new int[] {1, 2, 3};
int[] c = new int[] {1, 3, 2};
```

```
bool returnsTrue = a.SequenceEqual(b);
bool returnsFalse = a.SequenceEqual(c);
```

## Count y LongCount

`Count` devuelve el número de elementos en un `IEnumerable<T>` . `Count` también expone un parámetro de predicado opcional que le permite filtrar los elementos que desea contar.

```
int[] array = { 1, 2, 3, 4, 2, 5, 3, 1, 2 };

int n = array.Count(); // returns the number of elements in the array
int x = array.Count(i => i > 2); // returns the number of elements in the array greater than 2
```

`LongCount` funciona de la misma manera que `Count` pero tiene un tipo de retorno de `long` y se usa para contar `IEnumerable<T>` que son más largas que `int.MaxValue`

```
int[] array = GetLargeArray();

long n = array.LongCount(); // returns the number of elements in the array
long x = array.LongCount(i => i > 100); // returns the number of elements in the array greater
than 100
```

## Incrementando una consulta

Debido a que LINQ utiliza la **ejecución diferida** , podemos tener un objeto de consulta que no contiene realmente los valores, pero devolverá los valores cuando se evalúe. De este modo, podemos construir dinámicamente la consulta en función de nuestro flujo de control y evaluarla una vez que hayamos terminado:

```
IEnumerable<VehicleModel> BuildQuery(int vehicleType, SearchModel search, int start = 1, int
count = -1) {
    IEnumerable<VehicleModel> query = _entities.Vehicles
        .Where(x => x.Active && x.Type == vehicleType)
        .Select(x => new VehicleModel {
            Id = v.Id,
            Year = v.Year,
            Class = v.Class,
            Make = v.Make,
            Model = v.Model,
            Cylinders = v.Cylinders ?? 0
        });
```

Podemos aplicar condicionalmente filtros:

```
    if (!search.Years.Contains("all", StringComparer.OrdinalIgnoreCase))
        query = query.Where(v => search.Years.Contains(v.Year));

    if (!search.Makes.Contains("all", StringComparer.OrdinalIgnoreCase)) {
        query = query.Where(v => search.Makes.Contains(v.Make));
    }
```

```
    if (!search.Models.Contains("all", StringComparer.OrdinalIgnoreCase)) {
        query = query.Where(v => search.Models.Contains(v.Model));
    }

    if (!search.Cylinders.Equals("all", StringComparer.OrdinalIgnoreCase)) {
        decimal minCylinders = 0;
        decimal maxCylinders = 0;
        switch (search.Cylinders) {
            case "2-4":
                maxCylinders = 4;
                break;
            case "5-6":
                minCylinders = 5;
                maxCylinders = 6;
                break;
            case "8":
                minCylinders = 8;
                maxCylinders = 8;
                break;
            case "10+":
                minCylinders = 10;
                break;
        }
        if (minCylinders > 0) {
            query = query.Where(v => v.Cylinders >= minCylinders);
        }
        if (maxCylinders > 0) {
            query = query.Where(v => v.Cylinders <= maxCylinders);
        }
    }
```

Podemos agregar un orden de clasificación a la consulta en función de una condición:

```
    switch (search.SortingColumn.ToLower()) {
        case "make_model":
            query = query.OrderBy(v => v.Make).ThenBy(v => v.Model);
            break;
        case "year":
            query = query.OrderBy(v => v.Year);
            break;
        case "engine_size":
            query = query.OrderBy(v => v.EngineSize).ThenBy(v => v.Cylinders);
            break;
        default:
            query = query.OrderBy(v => v.Year); //The default sorting.
    }
```

Nuestra consulta se puede definir para comenzar desde un punto dado:

```
    query = query.Skip(start - 1);
```

y definido para devolver un número específico de registros:

```
    if (count > -1) {
        query = query.Take(count);
    }
    return query;
```

```
}
```

---

Una vez que tenemos el objeto de consulta, podemos evaluar los resultados con un bucle `foreach` o uno de los métodos LINQ que devuelve un conjunto de valores, como `ToList` o `ToArray` :

```
SearchModel sm;

// populate the search model here
// ...

List<VehicleModel> list = BuildQuery(5, sm).ToList();
```

## Cremallera

El método de extensión `Zip` actúa sobre dos colecciones. Se empareja cada elemento en las dos series en función de la posición. Con una instancia de `Func` , usamos `Zip` para manejar elementos de las dos colecciones de C # en pares. Si las series difieren en tamaño, los elementos adicionales de las series más grandes se ignorarán.

Para tomar un ejemplo del libro "C # en pocas palabras",

```
int[] numbers = { 3, 5, 7 };
string[] words = { "three", "five", "seven", "ignored" };
IEnumerable<string> zip = numbers.Zip(words, (n, w) => n + "=" + w);
```

**Salida:**

3 = tres
5 = cinco
7 = siete

Ver demostración

## GroupJoin con rango externo variable

```
Customer[] customers = Customers.ToArray();
Purchase[] purchases = Purchases.ToArray();

var groupJoinQuery =
    from c in customers
    join p in purchases on c.ID equals p.CustomerID
    into custPurchases
    select new
    {
        CustName = c.Name,
        custPurchases
    };
```

## ElementAt y ElementAtOrDefault

`ElementAt`

devolverá el artículo en el índice `n` . Si `n` no está dentro del rango de lo enumerable, lanza una `ArgumentOutOfRangeException` .

```
int[] numbers  = { 1, 2, 3, 4, 5 };
numbers.ElementAt(2);  // 3
numbers.ElementAt(10); // throws ArgumentOutOfRangeException
```

`ElementAtOrDefault` devolverá el artículo en el índice `n` . Si `n` no está dentro del rango de lo enumerable, devuelve un `default(T)` .

```
int[] numbers  = { 1, 2, 3, 4, 5 };
numbers.ElementAtOrDefault(2);  // 3
numbers.ElementAtOrDefault(10); // 0 = default(int)
```

Tanto `ElementAt` como `ElementAtOrDefault` están optimizados para cuando la fuente es un `IList<T>` y se usará la indexación normal en esos casos.

Tenga en cuenta que para `ElementAt` , si el índice proporcionado es mayor que el tamaño de `IList<T>` , la lista debería (pero técnicamente no está garantizado) lanzar una `ArgumentOutOfRangeException` .

## Cuantificadores Linq

Las operaciones de cuantificación devuelven un valor booleano si algunos o todos los elementos en una secuencia satisfacen una condición. En este artículo, veremos algunos escenarios comunes de LINQ to Objects donde podemos usar estos operadores. Hay 3 operaciones de cuantificadores que se pueden usar en LINQ:

`All` : se utiliza para determinar si todos los elementos de una secuencia satisfacen una condición. P.ej:

```
int[] array = { 10, 20, 30 };

// Are all elements >= 10? YES
array.All(element => element >= 10);

// Are all elements >= 20? NO
array.All(element => element >= 20);

// Are all elements < 40? YES
array.All(element => element < 40);
```

`Any` : se utiliza para determinar si algún elemento de una secuencia satisface una condición. P.ej:

```
int[] query=new int[] { 2, 3, 4 }
query.Any (n => n == 3);
```

`Contains` : se utiliza para determinar si una secuencia contiene un elemento específico. P.ej:

```
//for int array
```

```
int[] query =new int[] { 1,2,3 };
query.Contains(1);

//for string array
string[] query={"Tom","grey"};
query.Contains("Tom");

//for a string
var stringValue="hello";
stringValue.Contains("h");
```

## Uniendo múltiples secuencias

Considere las entidades `Customer` , `Purchase` y `PurchaseItem` siguiente manera:

```
public class Customer
{
   public string Id { get; set } // A unique Id that identifies customer
   public string Name  {get; set; }
}

public class Purchase
{
   public string Id { get; set }
   public string CustomerId {get; set; }
   public string Description { get; set; }
}

public class PurchaseItem
{
   public string Id { get; set }
   public string PurchaseId {get; set; }
   public string Detail { get; set; }
}
```

Considere los siguientes datos de muestra para las entidades anteriores:

```
var customers = new List<Customer>()
 {
    new Customer() {
        Id = Guid.NewGuid().ToString(),
        Name = "Customer1"
    },

    new Customer() {
        Id = Guid.NewGuid().ToString(),
        Name = "Customer2"
    }
 };

 var purchases = new List<Purchase>()
 {
     new Purchase() {
         Id = Guid.NewGuid().ToString(),
         CustomerId = customers[0].Id,
         Description = "Customer1-Purchase1"
     },
```

```
    new Purchase() {
        Id = Guid.NewGuid().ToString(),
        CustomerId = customers[0].Id,
        Description = "Customer1-Purchase2"
    },

    new Purchase() {
        Id = Guid.NewGuid().ToString(),
        CustomerId = customers[1].Id,
        Description = "Customer2-Purchase1"
    },

    new Purchase() {
        Id = Guid.NewGuid().ToString(),
        CustomerId = customers[1].Id,
        Description = "Customer2-Purchase2"
    }
 };

var purchaseItems = new List<PurchaseItem>()
{
    new PurchaseItem() {
        Id = Guid.NewGuid().ToString(),
        PurchaseId= purchases[0].Id,
        Detail = "Purchase1-PurchaseItem1"
    },

    new PurchaseItem() {
        Id = Guid.NewGuid().ToString(),
        PurchaseId= purchases[1].Id,
        Detail = "Purchase2-PurchaseItem1"
    },

    new PurchaseItem() {
        Id = Guid.NewGuid().ToString(),
        PurchaseId= purchases[1].Id,
        Detail = "Purchase2-PurchaseItem2"
    },

    new PurchaseItem() {
        Id = Guid.NewGuid().ToString(),
        PurchaseId= purchases[3].Id,
        Detail = "Purchase3-PurchaseItem1"
    }
};
```

Ahora, considere la siguiente consulta de linq:

```
var result = from c in customers
             join p in purchases on c.Id equals p.CustomerId           // first join
             join pi in purchaseItems on p.Id equals pi.PurchaseId     // second join
             select new
             {
                c.Name, p.Description, pi.Detail
             };
```

Para dar salida al resultado de la consulta anterior:

```
foreach(var resultItem in result)
```

```
{
    Console.WriteLine($"{resultItem.Name}, {resultItem.Description}, {resultItem.Detail}");
}
```

El resultado de la consulta sería:

Cliente1, Cliente1-Purchase1, Purchase1-PurchaseItem1

Cliente1, Cliente1-Purchase2, Purchase2-PurchaseItem1

Cliente1, Cliente1-Purchase2, Purchase2-PurchaseItem2

Cliente2, Cliente2-Purchase2, Purchase3-PurchaseItem1

Demo en vivo en .NET Fiddle

## Uniéndose en múltiples claves

```
  PropertyInfo[] stringProps = typeof (string).GetProperties();//string properties
  PropertyInfo[] builderProps = typeof(StringBuilder).GetProperties();//stringbuilder
properties

    var query =
        from s in stringProps
        join b in builderProps
            on new { s.Name, s.PropertyType } equals new { b.Name, b.PropertyType }
        select new
        {
            s.Name,
            s.PropertyType,
            StringToken = s.MetadataToken,
            StringBuilderToken = b.MetadataToken
        };
```

Tenga en cuenta que los tipos anónimos en la `join` anterior deben contener las mismas propiedades, ya que los objetos se consideran iguales solo si todas sus propiedades son iguales. De lo contrario la consulta no se compilará.

## Seleccionar con Func selector - Se usa para obtener ranking de elementos

Una de las sobrecargas de los métodos de extensión `Select` también pasa el `index` del elemento actual en la colección que se está `select` . Estos son algunos usos de la misma.

### Obtener el "número de fila" de los artículos

```
var rowNumbers = collection.OrderBy(item => item.Property1)
                           .ThenBy(item => item.Property2)
                           .ThenByDescending(item => item.Property3)
                           .Select((item, index) => new { Item = item, RowNumber = index })
                           .ToList();
```

### Obtener el rango de un elemento *dentro de* su grupo

```
var rankInGroup = collection.GroupBy(item => item.Property1)
                            .OrderBy(group => group.Key)
                            .SelectMany(group => group.OrderBy(item => item.Property2)
                                                      .ThenByDescending(item => item.Property3)
                                                      .Select((item, index) => new
                                                      {
                                                          Item = item,
                                                          RankInGroup = index
                                                      })).ToList();
```

**Obtener el ranking de grupos (también conocido en Oracle como dense_rank)**

```
var rankOfBelongingGroup = collection.GroupBy(item => item.Property1)
                            .OrderBy(group => group.Key)
                            .Select((group, index) => new
                            {
                                Items = group,
                                Rank = index
                            })
                            .SelectMany(v => v.Items, (s, i) => new
                            {
                                Item = i,
                                DenseRank = s.Rank
                            }).ToList();
```

Para probar esto puedes usar:

```
public class SomeObject
{
    public int Property1 { get; set; }
    public int Property2 { get; set; }
    public int Property3 { get; set; }

    public override string ToString()
    {
        return string.Join(", ", Property1, Property2, Property3);
    }
}
```

Y datos:

```
List<SomeObject> collection = new List<SomeObject>
{
    new SomeObject { Property1 = 1, Property2 = 1, Property3 = 1},
    new SomeObject { Property1 = 1, Property2 = 2, Property3 = 1},
    new SomeObject { Property1 = 1, Property2 = 2, Property3 = 2},
    new SomeObject { Property1 = 2, Property2 = 1, Property3 = 1},
    new SomeObject { Property1 = 2, Property2 = 2, Property3 = 1},
    new SomeObject { Property1 = 2, Property2 = 2, Property3 = 1},
    new SomeObject { Property1 = 2, Property2 = 3, Property3 = 1}
};
```

## TakeWhile

`TakeWhile` devuelve elementos de una secuencia siempre que la condición sea verdadera

```
int[] list = { 1, 10, 40, 50, 44, 70, 4 };
var result = list.TakeWhile(item => item < 50).ToList();
// result = { 1, 10, 40 }
```

## Suma

El método de extensión `Enumerable.Sum` calcula la suma de los valores numéricos.

En caso de que los elementos de la colección sean en sí mismos números, puede calcular la suma directamente.

```
int[] numbers = new int[] { 1, 4, 6 };
Console.WriteLine( numbers.Sum() ); //outputs 11
```

En caso de que el tipo de los elementos sea un tipo complejo, puede usar una expresión lambda para especificar el valor que debe calcularse:

```
var totalMonthlySalary = employees.Sum( employee => employee.MonthlySalary );
```

El método de extensión de suma puede calcularse con los siguientes tipos:

- Int32
- Int64
- Soltero
- Doble
- Decimal

En caso de que su colección contenga tipos anulables, puede usar el operador de unión nula para establecer un valor predeterminado para los elementos nulos:

```
int?[] numbers = new int?[] { 1, null, 6 };
Console.WriteLine( numbers.Sum( number => number ?? 0 ) ); //outputs 7
```

## Para buscar

> ToLookup devuelve una estructura de datos que permite la indexación. Es un método de extensión. Produce una instancia de ILookup que se puede indexar o enumerar usando un bucle foreach. Las entradas se combinan en grupos en cada tecla. - dotnetperls

```
string[] array = { "one", "two", "three" };
//create lookup using string length as key
var lookup = array.ToLookup(item => item.Length);

//join the values whose lengths are 3
Console.WriteLine(string.Join(",",lookup[3]));
//output: one,two
```

Otro ejemplo:

```
int[] array = { 1,2,3,4,5,6,7,8 };
//generate lookup for odd even numbers (keys will be 0 and 1)
var lookup = array.ToLookup(item => item % 2);

//print even numbers after joining
Console.WriteLine(string.Join(",",lookup[0]));
//output: 2,4,6,8

//print odd numbers after joining
Console.WriteLine(string.Join(",",lookup[1]));
//output: 1,3,5,7
```

## Construye tus propios operadores Linq para IEnumerable

Una de las mejores cosas de Linq es que es muy fácil de extender. Solo necesita crear un método de extensión cuyo argumento sea `IEnumerable<T>` .

```
public namespace MyNamespace
{
    public static class LinqExtensions
    {
        public static IEnumerable<List<T>> Batch<T>(this IEnumerable<T> source, int batchSize)
        {
            var batch = new List<T>();
            foreach (T item in source)
            {
                batch.Add(item);
                if (batch.Count == batchSize)
                {
                    yield return batch;
                    batch = new List<T>();
                }
            }
            if (batch.Count > 0)
                yield return batch;
        }
    }
}
```

Este ejemplo divide los elementos en una `IEnumerable<T>` en listas de un tamaño fijo, la última lista que contiene el resto de los elementos. Observe cómo se pasa el objeto al que se aplica el método de extensión ( `source` argumento) como el argumento inicial usando `this` palabra clave. Luego, la palabra clave de `yield` se utiliza para generar el siguiente elemento en la salida `IEnumerable<T>` antes de continuar con la ejecución desde ese punto (consulte la palabra clave de rendimiento ).

Este ejemplo se usaría en su código de esta manera:

```
//using MyNamespace;
var items = new List<int> { 2, 3, 4, 5, 6 };
foreach (List<int> sublist in items.Batch(3))
{
    // do something
}
```

En el primer bucle, la lista secundaria sería `{2, 3, 4}` y en el segundo `{5, 6}` .

Los métodos LinQ personalizados se pueden combinar con métodos LinQ estándar también. p.ej:

```
//using MyNamespace;
var result = Enumerable.Range(0, 13)          // generate a list
                       .Where(x => x%2 == 0) // filter the list or do something other
                       .Batch(3)              // call our extension method
                       .ToList()              // call other standard methods
```

Esta consulta devolverá números pares agrupados en lotes con un tamaño de 3: `{0, 2, 4}, {6, 8, 10}, {12}`

Recuerda que necesitas `using MyNamespace;` Línea para poder acceder al método de extensión.

## Usando SelectMany en lugar de bucles anidados

Dadas 2 listas

```
var list1 = new List<string> { "a", "b", "c" };
var list2 = new List<string> { "1", "2", "3", "4" };
```

Si desea generar todas las permutaciones, puede utilizar bucles anidados como

```
var result = new List<string>();
foreach (var s1 in list1)
    foreach (var s2 in list2)
        result.Add($"{s1}{s2}");
```

Usando SelectMany puedes hacer la misma operación que

```
var result = list1.SelectMany(x => list2.Select(y => $"{x}{y}", x, y)).ToList();
```

## Any and First (OrDefault) - Mejores prácticas

No explicaré lo que `Any` y `FirstOrDefault` hacen porque ya hay dos buenos ejemplos sobre ellos. Para obtener más información, consulte Cualquiera y primero, FirstOrDefault, Last, LastOrDefault, Single y SingleOrDefault .

Un patrón que a menudo veo en el código que **debe evitarse** es

```
if (myEnumerable.Any(t=>t.Foo == "Bob"))
{
    var myFoo = myEnumerable.First(t=>t.Foo == "Bob");
    //Do stuff
}
```

Podría escribirse de manera más eficiente así

```
var myFoo = myEnumerable.FirstOrDefault(t=>t.Foo == "Bob");
```

```
if (myFoo != null)
{
    //Do stuff
}
```

Al usar el segundo ejemplo, la colección se busca solo una vez y da el mismo resultado que el primero. La misma idea se puede aplicar a `Single` .

## GroupBy Sum y Count

Tomemos una clase de muestra:

```
public class Transaction
{
    public string Category { get; set; }
    public DateTime Date { get; set; }
    public decimal Amount { get; set; }
}
```

Ahora, consideremos una lista de transacciones:

```
var transactions = new List<Transaction>
{
   new Transaction { Category = "Saving Account", Amount = 56, Date =
DateTime.Today.AddDays(1) },
   new Transaction { Category = "Saving Account", Amount = 10, Date = DateTime.Today.AddDays(-
10) },
   new Transaction { Category = "Credit Card", Amount = 15, Date = DateTime.Today.AddDays(1)
},
   new Transaction { Category = "Credit Card", Amount = 56, Date = DateTime.Today },
   new Transaction { Category = "Current Account", Amount = 100, Date =
DateTime.Today.AddDays(5) },
};
```

Si desea calcular la suma sabia de la categoría y el recuento, puede usar GroupBy de la siguiente manera:

```
var summaryApproach1 = transactions.GroupBy(t => t.Category)
                           .Select(t => new
                           {
                               Category = t.Key,
                               Count = t.Count(),
                               Amount = t.Sum(ta => ta.Amount),
                           }).ToList();

Console.WriteLine("-- Summary: Approach 1 --");
summaryApproach1.ForEach(
            row => Console.WriteLine($"Category: {row.Category}, Amount: {row.Amount}, Count:
{row.Count}"));
```

Alternativamente, puedes hacer esto en un solo paso:

```
var summaryApproach2 = transactions.GroupBy(t => t.Category, (key, t) =>
{
```

```
        var transactionArray = t as Transaction[] ?? t.ToArray();
        return new
        {
            Category = key,
            Count = transactionArray.Length,
            Amount = transactionArray.Sum(ta => ta.Amount),
        };
}).ToList();

Console.WriteLine("-- Summary: Approach 2 --");
summaryApproach2.ForEach(
row => Console.WriteLine($"Category: {row.Category}, Amount: {row.Amount}, Count:
{row.Count}"));
```

La salida para ambas consultas anteriores sería la misma:

> Categoría: Cuenta de Ahorro, Cantidad: 66, Cuenta: 2
>
> Categoría: Tarjeta de crédito, Cantidad: 71, Cuenta: 2
>
> Categoría: Cuenta Corriente, Cantidad: 100, Cuenta: 1

Demo en vivo en .NET Fiddle

## Marcha atrás

- Invierte el orden de los elementos en una secuencia.
- Si no hay elementos lanza una `ArgumentNullException: source is null.`

***Ejemplo:***

```
// Create an array.
int[] array = { 1, 2, 3, 4 };                          //Output:
// Call reverse extension method on the array.         //4
var reverse = array.Reverse();                         //3
// Write contents of array to screen.                  //2
foreach (int value in reverse)                         //1
    Console.WriteLine(value);
```

Ejemplo de código en vivo

Recuerde que `Reverse()` puede funcionar de manera diferente según el orden de la cadena de sus declaraciones LINQ.

```
        //Create List of chars
        List<int> integerlist = new List<int>() { 1, 2, 3, 4, 5, 6 };

        //Reversing the list then taking the two first elements
        IEnumerable<int> reverseFirst = integerlist.Reverse<int>().Take(2);

        //Taking 2 elements and then reversing only thos two
        IEnumerable<int> reverseLast = integerlist.Take(2).Reverse();

        //reverseFirst output: 6, 5
        //reverseLast output:  2, 1
```

Ejemplo de código en vivo

*Reverse ()* funciona amortiguando todo y luego lo recorre hacia atrás, lo que no es muy eficiente, pero tampoco lo es OrderBy desde esa perspectiva.

En LINQ-to-Objects, hay operaciones de almacenamiento en búfer (Reverse, OrderBy, GroupBy, etc.) y operaciones de no almacenamiento en búfer (Where, Take, Skip, etc.).

***Ejemplo: Extensión inversa sin buffering***

```
public static IEnumerable<T> Reverse<T>(this IList<T> list) {
    for (int i = list.Count - 1; i >= 0; i--)
        yield return list[i];
}
```

Ejemplo de código en vivo

Este método puede encontrar problemas si muta la lista mientras está iterando.

## Enumerar lo Enumerable

La interfaz IEnumerable <T> es la interfaz base para todos los enumeradores genéricos y es una parte esencial de la comprensión de LINQ. En su núcleo, representa la secuencia.

Esta interfaz subyacente es heredada por todas las colecciones genéricas, como Collection <T> , Array , List <T> , Dictionary <TKey, TValue> Class y HashSet <T> .

Además de representar la secuencia, cualquier clase que herede de IEnumerable <T> debe proporcionar un IEnumerator <T>. El enumerador expone el iterador para el enumerable, y estas dos interfaces e ideas interconectadas son la fuente del dicho "enumerar el enumerable".

"Enumerar lo enumerable" es una frase importante. Lo enumerable es simplemente una estructura para iterar, no contiene ningún objeto materializado. Por ejemplo, al ordenar, un enumerable puede contener los criterios del campo a ordenar, pero usar `.OrderBy()` en sí mismo devolverá un IEnumerable <T> que solo sabe *cómo* ordenar. El uso de una llamada que materializará los objetos, como en iterar el conjunto, se conoce como enumeración (por ejemplo, `.ToList()` ). El proceso de enumeración utilizará la definición enumerable de *cómo* para moverse a través de la serie y devolver los objetos relevantes (en orden, filtrado, proyectado, etc.).

Solo una vez que se ha enumerado la enumeración provoca la materialización de los objetos, que es cuando las métricas como la complejidad del tiempo (el tiempo que debe tomar en relación con el tamaño de la serie) y la complejidad espacial (la cantidad de espacio que debe usar en relación con el tamaño de la serie) pueden medirse.

Crear su propia clase que hereda de IEnumerable <T> puede ser un poco complicado dependiendo de la serie subyacente que debe ser enumerable. En general, es mejor utilizar una de las colecciones genéricas existentes. Dicho esto, también es posible heredar de la interfaz IEnumerable <T> sin tener una matriz definida como la estructura subyacente.

Por ejemplo, usando la serie de Fibonacci como la secuencia subyacente. Tenga en cuenta que

la llamada a `Where` simplemente construye un `IEnumerable` , y no es hasta que se realiza una llamada para enumerar que enumerable se materializa cualquiera de los valores.

```
void Main()
{
    Fibonacci Fibo = new Fibonacci();
    IEnumerable<long> quadrillionplus = Fibo.Where(i => i > 1000000000000);
    Console.WriteLine("Enumerable built");
    Console.WriteLine(quadrillionplus.Take(2).Sum());
    Console.WriteLine(quadrillionplus.Skip(2).First());

    IEnumerable<long> fibMod612 = Fibo.OrderBy(i => i % 612);
    Console.WriteLine("Enumerable built");
    Console.WriteLine(fibMod612.First());//smallest divisible by 612
}

public class Fibonacci : IEnumerable<long>
{
    private int max = 90;

    //Enumerator called typically from foreach
    public IEnumerator GetEnumerator() {
        long n0 = 1;
        long n1 = 1;
        Console.WriteLine("Enumerating the Enumerable");
        for(int i = 0; i < max; i++){
            yield return n0+n1;
            n1 += n0;
            n0 = n1-n0;
        }
    }

    //Enumerable called typically from linq
    IEnumerator<long> IEnumerable<long>.GetEnumerator() {
        long n0 = 1;
        long n1 = 1;
        Console.WriteLine("Enumerating the Enumerable");
        for(int i = 0; i < max; i++){
            yield return n0+n1;
            n1 += n0;
            n0 = n1-n0;
        }
    }
}
```

Salida

```
Enumerable built
Enumerating the Enumerable
4052739537881
Enumerating the Enumerable
4052739537881
Enumerable built
Enumerating the Enumerable
14930352
```

La fortaleza en el segundo conjunto (el fibMod612) es que a pesar de que hicimos la llamada para solicitar nuestro conjunto completo de números de Fibonacci, ya que solo se tomó un valor

usando `.First()` la complejidad del tiempo fue O (n) como solo 1 valor Necesitaba ser comparado durante la ejecución del algoritmo de ordenamiento. Esto se debe a que nuestro enumerador solo solicitó 1 valor, por lo que no fue necesario materializar todo el enumerable. Si hubiéramos usado `.Take(5)` lugar de `.First()` el enumerador habría pedido 5 valores y, como máximo, deberían materializarse 5 valores. Comparado con la necesidad de ordenar un conjunto completo *y luego* tomar los primeros 5 valores, el principio de ahorrar mucho tiempo y espacio de ejecución.

## Orden por

> Ordena una colección por un valor especificado.

Cuando el valor es un **número entero** , **doble** o **flotante** , comienza con el *valor mínimo* , lo que significa que primero obtiene los valores negativos, que cero y después los valores positivos (vea el Ejemplo 1).

Cuando ordena por un **char,** el método compara los *valores ascii* de los caracteres para ordenar la colección (vea el Ejemplo 2).

Cuando ordena las **cadenas,** el método OrderBy las compara echando un vistazo a su CultureInfo, pero normalmente comienza con la *primera letra* del alfabeto (a, b, c ...).

Este tipo de orden se denomina ascendente, si lo desea al revés, necesita descender (consulte OrderByDescending).

**Ejemplo 1:**

```
int[] numbers = {2, 1, 0, -1, -2};
IEnumerable<int> ascending = numbers.OrderBy(x => x);
// returns {-2, -1, 0, 1, 2}
```

**Ejemplo 2:**

```
 char[] letters = {' ', '!', '?', '[', '{', '+', '1', '9', 'a', 'A', 'b', 'B', 'y', 'Y', 'z',
'Z'};
 IEnumerable<char> ascending = letters.OrderBy(x => x);
 // returns { ' ', '!', '+', '1', '9', '?', 'A', 'B', 'Y', 'Z', '[', 'a', 'b', 'y', 'z', '{' }
```

**Ejemplo:**

```
class Person
{
   public string Name { get; set; }
   public int Age { get; set; }
}

var people = new[]
{
    new Person {Name = "Alice", Age = 25},
    new Person {Name = "Bob", Age = 21},
    new Person {Name = "Carol", Age = 43}
};
var youngestPerson = people.OrderBy(x => x.Age).First();
```

```
var name = youngestPerson.Name; // Bob
```

## OrderByDescending

> Ordena una colección por un valor especificado.

Cuando el valor es un **número entero** , **doble** o **flotante** , comienza con el *valor máximo* , lo que significa que primero obtiene los valores positivos, que cero y después los valores negativos (consulte el Ejemplo 1).

Cuando ordena por un **char,** el método compara los *valores ascii* de los caracteres para ordenar la colección (vea el Ejemplo 2).

Cuando ordena las **cadenas,** el método OrderBy las compara echando un vistazo a CultureInfo, pero normalmente comienza con la *última letra* del alfabeto (z, y, x, ...).

Este tipo de orden se denomina descendente, si lo desea al revés, necesita ascender (consulte Orden por).

**Ejemplo 1:**

```
int[] numbers = {-2, -1, 0, 1, 2};
IEnumerable<int> descending = numbers.OrderByDescending(x => x);
// returns {2, 1, 0, -1, -2}
```

**Ejemplo 2:**

```
char[] letters = {' ', '!', '?', '[', '{', '+', '1', '9', 'a', 'A', 'b', 'B', 'y', 'Y', 'z',
'Z'};
IEnumerable<char> descending = letters.OrderByDescending(x => x);
// returns { '{', 'z', 'y', 'b', 'a', '[', 'Z', 'Y', 'B', 'A', '?', '9', '1', '+', '!', ' ' }
```

**Ejemplo 3:**

```
class Person
{
   public  string Name { get; set; }
   public  int Age { get; set; }
}

var people = new[]
{
    new Person {Name = "Alice", Age = 25},
    new Person {Name = "Bob", Age = 21},
    new Person {Name = "Carol", Age = 43}
};
var oldestPerson = people.OrderByDescending(x => x.Age).First();
var name = oldestPerson.Name; // Carol
```

## Concat

Fusiona dos colecciones (sin eliminar duplicados)

```
List<int> foo = new List<int> { 1, 2, 3 };
List<int> bar = new List<int> { 3, 4, 5 };

// Through Enumerable static class
var result = Enumerable.Concat(foo, bar).ToList(); // 1,2,3,3,4,5

// Through extension method
var result = foo.Concat(bar).ToList(); // 1,2,3,3,4,5
```

## Contiene

MSDN:

> Determina si una secuencia contiene un elemento específico utilizando un `IEqualityComparer<T>`

```
List<int> numbers = new List<int> { 1, 2, 3, 4, 5 };
var result1 = numbers.Contains(4); // true
var result2 = numbers.Contains(8); // false

List<int> secondNumberCollection = new List<int> { 4, 5, 6, 7 };
// Note that can use the Intersect method in this case
var result3 = secondNumberCollection.Where(item => numbers.Contains(item)); // will be true
only for 4,5
```

Usando un objeto definido por el usuario:

```
public class Person
{
   public string Name { get; set; }
}

List<Person> objects = new List<Person>
{
    new Person { Name = "Nikki"},
    new Person { Name = "Gilad"},
    new Person { Name = "Phil"},
    new Person { Name = "John"}
};

//Using the Person's Equals method - override Equals() and GetHashCode() - otherwise it
//will compare by reference and result will be false
var result4 = objects.Contains(new Person { Name = "Phil" }); // true
```

Usando la sobrecarga `Enumerable.Contains(value, comparer)` :

```
public class Compare : IEqualityComparer<Person>
{
    public bool Equals(Person x, Person y)
    {
        return x.Name == y.Name;
    }
    public int GetHashCode(Person codeh)
    {
        return codeh.Name.GetHashCode();
    }
```

```
}

var result5 = objects.Contains(new Person { Name = "Phil" }, new Compare()); // true
```

**Un uso inteligente de `Contains` sería reemplazar varias cláusulas `if` de una llamada `Contains` .**

Así que en lugar de hacer esto:

```
if(status == 1 || status == 3 || status == 4)
{
    //Do some business operation
}
else
{
    //Do something else
}
```

Hacer esto:

```
if(new int[] {1, 3, 4 }.Contains(status)
{
    //Do some business operaion
}
else
{
    //Do something else
}
```

Lea Consultas LINQ en línea: https://riptutorial.com/es/csharp/topic/68/consultas-linq

# Capítulo 39: Contexto de sincronización en Async-Await

## Examples

### Pseudocódigo para palabras clave async / await

Considere un método asíncrono simple:

```
async Task Foo()
{
    Bar();
    await Baz();
    Qux();
}
```

Simplificando, podemos decir que este código en realidad significa lo siguiente:

```
Task Foo()
{
    Bar();
    Task t = Baz();
    var context = SynchronizationContext.Current;
    t.ContinueWith(task) =>
    {
        if (context == null)
            Qux();
        else
            context.Post((obj) => Qux(), null);
    }, TaskScheduler.Current);

    return t;
}
```

Significa que las palabras clave `async` / `await` usan el contexto de sincronización actual si existe. Es decir, puede escribir un código de biblioteca que funcione correctamente en las aplicaciones de UI, Web y Consola.

Artículo fuente .

### Deshabilitando el contexto de sincronización

Para deshabilitar el contexto de sincronización, debe llamar al método `ConfigureAwait` :

```
async Task() Foo()
{
    await Task.Run(() => Console.WriteLine("Test"));
}

. . .
```

```
Foo().ConfigureAwait(false);
```

> ConfigureAwait proporciona un medio para evitar el comportamiento de captura predeterminado de SynchronizationContext; pasar falso para el parámetro flowContext evita que se use SynchronizationContext para reanudar la ejecución después de la espera.

Cita de Todo se trata del SynchronizationContext .

## ¿Por qué SynchronizationContext es tan importante?

Considera este ejemplo:

```
private void button1_Click(object sender, EventArgs e)
{
    label1.Text = RunTooLong();
}
```

Este método congelará la aplicación de la interfaz de usuario hasta que se complete el `RunTooLong` . La aplicación no responderá.

Puede intentar ejecutar código interno de forma asíncrona:

```
private void button1_Click(object sender, EventArgs e)
{
    Task.Run(() => label1.Text = RunTooLong());
}
```

Pero este código no se ejecutará porque el cuerpo interno puede ejecutarse en un subproceso que no pertenece a la IU y no debería cambiar las propiedades de la IU directamente :

```
private void button1_Click(object sender, EventArgs e)
{
    Task.Run(() =>
    {
        var label1Text = RunTooLong();

        if (label1.InvokeRequired)
            lable1.BeginInvoke((Action) delegate() { label1.Text = label1Text; });
        else
            label1.Text = label1Text;
    });
}
```

Ahora no olvides usar siempre este patrón. O intente `SynchronizationContext.Post` que lo hará por usted:

```
private void button1_Click(object sender, EventArgs e)
{
    Task.Run(() =>
    {
```

```
        var label1Text = RunTooLong();
        SynchronizationContext.Current.Post((obj) =>
        {
            label1.Text = label1    Text);
        }, null);
    });
}
```

Lea Contexto de sincronización en Async-Await en línea:
https://riptutorial.com/es/csharp/topic/7381/contexto-de-sincronizacion-en-async-await

# Capítulo 40: Contratos de código

## Sintaxis

1. Contrato.Requisitos (Condición, UserMessage)

   Contrato.Requisitos (Condición, UserMessage)

   Contrato.Resultado <T>

   Contrato.Aseguros ()

   Contract.Invariants ()

## Observaciones

.NET es compatible con la idea de diseño por contrato a través de su clase de contratos que se encuentra en el espacio de nombres System.Diagnostics e introducido en .NET 4.0. La API de contratos de código incluye clases para verificaciones estáticas y en tiempo de ejecución del código y le permite definir condiciones previas, condiciones posteriores e invariantes dentro de un método. Las condiciones previas especifican las condiciones que deben cumplir los parámetros antes de que un método pueda ejecutarse, las condiciones posteriores que se verifican al completar un método, y las invariantes definen las condiciones que no cambian durante la ejecución de un método.

**¿Por qué son necesarios los contratos de código?**

El seguimiento de los problemas de una aplicación cuando la aplicación se está ejecutando, es una de las principales preocupaciones de todos los desarrolladores y administradores. El seguimiento se puede realizar de muchas maneras. Por ejemplo -

- Puede aplicar el rastreo en nuestra aplicación y obtener los detalles de una aplicación cuando la aplicación se está ejecutando.
- Puede utilizar el mecanismo de registro de eventos cuando está ejecutando la aplicación. Los mensajes se pueden ver usando el Visor de Eventos.
- Puede aplicar Performance Monitoring después de un intervalo de tiempo específico y escribir datos en vivo desde su aplicación.

Los contratos de código utilizan un enfoque diferente para el seguimiento y la gestión de problemas dentro de una aplicación. En lugar de validar todo lo que se devuelve de una llamada de método, los Contratos de código con la ayuda de condiciones previas, postcondiciones e invariantes de métodos, aseguran que todo lo que ingresa y salga de sus métodos sea correcto.

## Examples

## Precondiciones

```
namespace CodeContractsDemo
{
    using System;
    using System.Collections.Generic;
    using System.Diagnostics.Contracts;

    public class PaymentProcessor
    {
        private List<Payment> _payments = new List<Payment>();

        public void Add(Payment payment)
        {
            Contract.Requires(payment != null);
            Contract.Requires(!string.IsNullOrEmpty(payment.Name));
            Contract.Requires(payment.Date <= DateTime.Now);
            Contract.Requires(payment.Amount > 0);

            this._payments.Add(payment);
        }
    }
}
```

## Postcondiciones

```
public double GetPaymentsTotal(string name)
{
    Contract.Ensures(Contract.Result<double>() >= 0);

    double total = 0.0;

    foreach (var payment in this._payments) {
        if (string.Equals(payment.Name, name)) {
            total += payment.Amount;
        }
    }

    return total;
}
```

## Invariantes

```
namespace CodeContractsDemo
{
    using System;
    using System.Diagnostics.Contracts;

    public class Point
    {
        public int X { get; set; }
        public int Y { get; set; }

        public Point()
        {
        }
```

```
        public Point(int x, int y)
        {
            this.X = x;
            this.Y = y;
        }

        public void Set(int x, int y)
        {
            this.X = x;
            this.Y = y;
        }

        public void Test(int x, int y)
        {
            for (int dx = -x; dx <= x; dx++) {
                this.X = dx;
                Console.WriteLine("Current X = {0}", this.X);
            }

            for (int dy = -y; dy <= y; dy++) {
                this.Y = dy;
                Console.WriteLine("Current Y = {0}", this.Y);
            }

            Console.WriteLine("X = {0}", this.X);
            Console.WriteLine("Y = {0}", this.Y);
        }

        [ContractInvariantMethod]
        private void ValidateCoordinates()
        {
            Contract.Invariant(this.X >= 0);
            Contract.Invariant(this.Y >= 0);
        }
    }
}
```

## Definición de contratos en la interfaz

```
[ContractClass(typeof(ValidationContract))]
interface IValidation
{
    string CustomerID{get;set;}
    string Password{get;set;}
}

[ContractClassFor(typeof(IValidation))]
sealed class ValidationContract:IValidation
{
    string IValidation.CustomerID
    {
        [Pure]
        get
        {
            return Contract.Result<string>();
        }
        set
        {
```

```
            Contract.Requires<ArgumentNullException>(!string.IsNullOrEmpty(value), "Customer
ID cannot be null!!");
        }
    }

    string IValidation.Password
    {
        [Pure]
        get
        {
            return Contract.Result<string>();
        }
        set
        {
            Contract.Requires<ArgumentNullException>(!string.IsNullOrEmpty(value), "Password
cannot be null!!");
        }
    }
}

class Validation:IValidation
{
    public string GetCustomerPassword(string customerID)
    {
        Contract.Requires(!string.IsNullOrEmpty(customerID),"Customer ID cannot be Null");
        Contract.Requires<ArgumentNullException>(!string.IsNullOrEmpty(customerID),
"Exception!!");
        Contract.Ensures(Contract.Result<string>() != null);
        string password="AAA@1234";
        if (customerID!=null)
        {
            return password;
        }
        else
        {
            return null;
        }

    }

    private string m_custID, m_PWD;

    public string CustomerID
    {
        get
        {
            return m_custID;
        }
        set
        {
            m_custID = value;
        }
    }

    public string Password
    {
        get
        {
            return m_PWD;
        }
        set
```

```
        {
            m_PWD = value;
        }
    }
}
```

En el código anterior, hemos definido una interfaz llamada `IValidation` con un atributo `[ContractClass]` . Este atributo toma una dirección de una clase en la que hemos implementado un contrato para una interfaz. La clase `ValidationContract` hace uso de las propiedades definidas en la interfaz y comprueba los valores nulos mediante `Contract.Requires<T>` . `Contract.Requires<T>` . `T` es una clase de excepción.

También hemos marcado el acceso get con un atributo `[Pure]` . El atributo puro garantiza que el método o una propiedad no cambie el estado de instancia de una clase en la que se implementa la interfaz de `IValidation` .

Lea Contratos de código en línea: https://riptutorial.com/es/csharp/topic/4241/contratos-de-codigo

# Capítulo 41: Contratos de Código y Afirmaciones

## Examples

### Las afirmaciones para verificar la lógica siempre deben ser ciertas

Las afirmaciones se utilizan no para realizar pruebas de los parámetros de entrada, sino para verificar que el flujo del programa sea correcto, es decir, que puede hacer ciertas suposiciones acerca de su código en un momento determinado. En otras palabras: una prueba realizada con `Debug.Assert` *siempre* debe asumir que el valor probado es `true` .

Debug.Assert solo se ejecuta en construcciones DEBUG; Se filtra de las versiones RELEASE. Debe considerarse una herramienta de depuración además de las pruebas unitarias y no como un reemplazo de los contratos de código o los métodos de validación de entrada.

Por ejemplo, esta es una buena afirmación:

```
var systemData = RetrieveSystemConfiguration();
Debug.Assert(systemData != null);
```

Aquí afirmar es una buena opción porque podemos suponer que RetrieveSystemConfiguration () devolverá un valor válido y nunca devolverá el valor nulo.

Aquí hay otro buen ejemplo:

```
UserData user = RetrieveUserData();
Debug.Assert(user != null);
Debug.Assert(user.Age > 0);
int year = DateTime.Today.Year - user.Age;
```

Primero, podemos asumir que RetrieveUserData () devolverá un valor válido. Luego, antes de usar la propiedad Age, verificamos la suposición (que siempre debería ser cierta) de que la edad del usuario es estrictamente positiva.

Este es un mal ejemplo de aseverar:

```
string input = Console.ReadLine();
int age = Convert.ToInt32(input);
Debug.Assert(age > 16);
Console.WriteLine("Great, you are over 16");
```

Assert no es para validación de entrada porque es incorrecto suponer que esta aserción siempre será verdadera. Debe utilizar métodos de validación de entrada para eso. En el caso anterior, también debe verificar que el valor de entrada sea un número en primer lugar.

Lea Contratos de Código y Afirmaciones en línea:
https://riptutorial.com/es/csharp/topic/4349/contratos-de-codigo-y-afirmaciones

# Capítulo 42: Convenciones de nombres

## Introducción

Este tema describe algunas convenciones básicas de nomenclatura utilizadas al escribir en el lenguaje C #. Como todas las convenciones, el compilador no las impone, pero garantizará la legibilidad entre los desarrolladores.

Para conocer las pautas de diseño del marco .NET, consulte docs.microsoft.com/dotnet/standard/design-guidelines .

## Observaciones

## Elija nombres de identificadores fácilmente legibles

Por ejemplo, una propiedad llamada HorizontalAlignment es más legible en inglés que AlignmentHorizontal.

## Favorecer la legibilidad sobre la brevedad.

El nombre de la propiedad `CanScrollHorizontally` es mejor que `ScrollableX` (una referencia poco `ScrollableX` al eje X).

Evite usar guiones bajos, guiones o cualquier otro carácter no alfanumérico.

## No utilice la notación húngara

La notación húngara es la práctica de incluir un prefijo en los identificadores para codificar algunos metadatos sobre el parámetro, como el tipo de datos del identificador, por ejemplo, `string strName` .

Además, evite utilizar identificadores que entren en conflicto con las palabras clave que ya se utilizan en C #.

## Abreviaciones y acronimos

En general, no debe utilizar abreviaturas o acrónimos; esto hace que sus nombres sean menos legibles. Del mismo modo, es difícil saber cuándo es seguro asumir que un acrónimo es ampliamente reconocido.

## Examples

### Convenios de capitalización

Los siguientes términos describen diferentes maneras de identificar los casos.

## Pascal Casing

La primera letra del identificador y la primera letra de cada palabra concatenada subsiguiente están en mayúscula. Puede usar el caso de Pascal para identificadores de tres o más caracteres. Por ejemplo: `BackColor`

## Carcasa de camello

La primera letra de un identificador es minúscula y la primera letra de cada palabra concatenada subsiguiente se escribe con mayúscula. Por ejemplo: `backColor`

## Mayúsculas

Todas las letras en el identificador están en mayúscula. Por ejemplo: `IO`

---

## Reglas

Cuando un identificador consta de varias palabras, no use separadores, como guiones bajos ("_") o guiones ("-"), entre palabras. En su lugar, utilice la caja para indicar el comienzo de cada palabra.

La siguiente tabla resume las reglas de uso de mayúsculas para los identificadores y proporciona ejemplos para los diferentes tipos de identificadores:

| Identificador | Caso | Ejemplo |
|---|---|---|
| Variable local | Camello | nombre del coche |
| Clase | Pascal | AppDomain |
| Tipo de enumeración | Pascal | Nivel de errores |
| Valores de enumeración | Pascal | Error fatal |
| Evento | Pascal | ValueChanged |
| Clase de excepcion | Pascal | WebException |
| Campo estático de solo lectura | Pascal | RedValue |
| Interfaz | Pascal | IDisponible |
| Método | Pascal | Encadenar |

| Identificador | Caso | Ejemplo |
|---|---|---|
| Espacio de nombres | Pascal | Sistema.Dibujo |
| Parámetro | Camello | escribe un nombre |
| Propiedad | Pascal | BackColor |

Se puede encontrar más información en MSDN .

### Interfaces

Las interfaces deben nombrarse con sustantivos o frases nominales, o adjetivos que describan el comportamiento. Por ejemplo, `IComponent` usa un nombre descriptivo, `ICustomAttributeProvider` usa una frase de nombre e `IPersistable` usa un adjetivo.

Los nombres de la interfaz deben tener un prefijo con la letra `I` , para indicar que el tipo es una interfaz, y se debe usar el caso Pascal.

A continuación se muestran correctamente las interfaces con nombre:

```
public interface IServiceProvider
public interface IFormatable
```

### Campos privados

Hay dos convenciones comunes para los campos privados: `camelCase` y `_camelCaseWithLeadingUnderscore` .

## El caso de Carmel

```
public class Rational
{
    private readonly int numerator;
    private readonly int denominator;

    public Rational(int numerator, int denominator)
    {
        // "this" keyword is required to refer to the class-scope field
        this.numerator = numerator;
        this.denominator = denominator;
    }
}
```

## Funda de camello con subrayado.

```
public class Rational
{
    private readonly int _numerator;
    private readonly int _denominator;
```

```
    public Rational(int numerator, int denominator)
    {
        // Names are unique, so "this" keyword is not required
        _numerator = numerator;
        _denominator = denominator;
    }
}
```

### Espacios de nombres

El formato general para los espacios de nombres es:

```
<Company>.(<Product>|<Technology>)[.<Feature>][.<Subnamespace>].
```

Ejemplos incluyen:

```
Fabrikam.Math
Litware.Security
```

El prefijo de nombres de espacios de nombres con un nombre de compañía evita que los espacios de nombres de diferentes compañías tengan el mismo nombre.

### Enums

## Usa un nombre singular para la mayoría de Enums

```
public enum Volume
{
   Low,
   Medium,
   High
}
```

## Utilice un nombre plural para los tipos Enum que son campos de bits

```
[Flags]
public enum MyColors
{
    Yellow = 1,
    Green = 2,
    Red = 4,
    Blue = 8
}
```

*Nota: siempre agregue el* `FlagsAttribute` *a un tipo de bit de campo de bit.*

## No agregue 'enumeración' como sufijo

```
public enum VolumeEnum // Incorrect
```

## No utilice el nombre de enumeración en cada entrada

```
public enum Color
{
    ColorBlue, // Remove Color, unnecessary
    ColorGreen,
}
```

### Excepciones

## Añadir 'excepción' como sufijo

Los nombres de excepción personalizados deben tener el sufijo "-Exception".

Debajo están las excepciones correctamente nombradas:

```
public class MyCustomException : Exception
public class FooException : Exception
```

Lea Convenciones de nombres en línea:
https://riptutorial.com/es/csharp/topic/2330/convenciones-de-nombres

# Capítulo 43: Corriente

## Examples

### Usando Streams

Un flujo es un objeto que proporciona un medio de bajo nivel para transferir datos. Ellos mismos no actúan como contenedores de datos.

Los datos con los que tratamos están en forma de matriz de bytes ( `byte []` ). Las funciones para leer y escribir están todas orientadas a bytes, por ejemplo, `WriteByte()` .

No hay funciones para tratar con enteros, cadenas, etc. Esto hace que el flujo sea muy general, pero menos fácil de trabajar si, por ejemplo, solo desea transferir texto. Las transmisiones pueden ser particularmente útiles cuando se trata de una gran cantidad de datos.

Tendremos que usar diferentes tipos de Stream en función de dónde se debe escribir / leer (es decir, la tienda de respaldo). Por ejemplo, si la fuente es un archivo, necesitamos usar `FileStream` :

```
string filePath = @"c:\Users\exampleuser\Documents\userinputlog.txt";
using (FileStream fs = new FileStream(filePath, FileMode.Open, FileAccess.Read,
FileShare.ReadWrite))
{
    // do stuff here...

    fs.Close();
}
```

De manera similar, `MemoryStream` se usa si el almacén de respaldo es memoria:

```
// Read all bytes in from a file on the disk.
byte[] file = File.ReadAllBytes(“C:\\file.txt”);

// Create a memory stream from those bytes.
using (MemoryStream memory = new MemoryStream(file))
{
   // do stuff here...
}
```

Del mismo modo, `System.Net.Sockets.NetworkStream` se utiliza para el acceso a la red.

Todos los flujos se derivan de la clase genérica `System.IO.Stream` . Los datos no se pueden leer ni escribir directamente desde secuencias. .NET Framework proporciona clases auxiliares como `StreamReader` , `StreamWriter` , `BinaryReader` y `BinaryWriter` que convierten entre los tipos nativos y la interfaz de flujo de bajo nivel, y transfieren los datos hacia o desde el flujo para usted.

La lectura y escritura de secuencias se puede hacer a través de `StreamReader` y `StreamWriter` . Se debe tener cuidado al cerrar estos. De forma predeterminada, el cierre también cerrará el flujo contenido y lo hará inutilizable para usos posteriores. Este comportamiento predeterminado se

puede cambiar utilizando un constructor que tenga el parámetro `bool leaveOpen` y establezca su valor como `true` .

`StreamWriter` :

```
FileStream fs = new FileStream("sample.txt", FileMode.Create);
StreamWriter sw = new StreamWriter(fs);
string NextLine = "This is the appended line.";
sw.Write(NextLine);
sw.Close();
//fs.Close(); There is no need to close fs. Closing sw will also close the stream it contains.
```

`StreamReader` :

```
using (var ms = new MemoryStream())
{
    StreamWriter sw = new StreamWriter(ms);
    sw.Write(123);
    //sw.Close();     This will close ms and when we try to use ms later it will cause an
exception
    sw.Flush();     //You can send the remaining data to stream. Closing will do this
automatically
    // We need to set the position to 0 in order to read
    // from the beginning.
    ms.Position = 0;
    StreamReader sr = new StreamReader(ms);
    var myStr = sr.ReadToEnd();
    sr.Close();
    ms.Close();
}
```

Dado que Classes `Stream` , `StreamReader` , `StreamWriter` , etc. implementan la interfaz `IDisposable` , podemos llamar al método `Dispose()` en objetos de estas clases.

Lea Corriente en línea: https://riptutorial.com/es/csharp/topic/3114/corriente

# Capítulo 44: Creación de una aplicación de consola con un editor de texto sin formato y el compilador de C # (csc.exe)

## Examples

### Creación de una aplicación de consola con un editor de texto sin formato y el compilador de C #

Para utilizar un editor de texto sin formato para crear una aplicación de consola escrita en C #, necesitará el compilador de C #. El compilador de C # (csc.exe) se puede encontrar en la siguiente ubicación: `%WINDIR%\Microsoft.NET\Framework64\v4.0.30319\csc.exe`

**NB** Dependiendo de la versión de .NET Framework que esté instalada en su sistema, es posible que deba cambiar la ruta anterior, según corresponda.

---

# Guardando el Código

El propósito de este tema no es enseñarle *cómo* escribir una aplicación de la Consola, sino enseñarle cómo *compilar* uno [para producir un solo archivo ejecutable], con nada más que el compilador C # y cualquier editor de texto sin formato (como Bloc).

1. Abra el cuadro de diálogo Ejecutar, utilizando el método abreviado de teclado `Windows` + `R`
2. Escribe `notepad` , luego pulsa `Intro`
3. Pegue el código de ejemplo a continuación, en el Bloc de notas
4. Guarde el archivo como `ConsoleApp.cs` , vaya a **Archivo** → **Guardar como ...** , luego ingrese `ConsoleApp.cs` en el campo de texto 'Nombre de archivo', luego seleccione `All Files` como tipo de archivo.
5. Haga clic en `Save`

# Compilando el código fuente

1. Abra el cuadro de diálogo Ejecutar, usando la `tecla de Windows` + `R`
2. Ingrese:

```
%WINDIR%\Microsoft.NET\Framework64\v4.0.30319\csc.exe /t:exe
/out:"C:\Users\yourUserName\Documents\ConsoleApp.exe"
"C:\Users\yourUserName\Documents\ConsoleApp.cs"
```

Ahora, vuelva a donde guardó originalmente su archivo `ConsoleApp.cs` . Ahora debería ver un archivo ejecutable ( `ConsoleApp.exe` ). Haga doble clic en `ConsoleApp.exe` para abrirlo.

¡Eso es! Su aplicación de consola ha sido compilada. Se ha creado un archivo ejecutable y ahora tiene una aplicación de consola que funciona.

```
using System;

namespace ConsoleApp
{
    class Program
    {
        private static string input = String.Empty;

        static void Main(string[] args)
        {
            goto DisplayGreeting;

            DisplayGreeting:
            {
                Console.WriteLine("Hello! What is your name?");

                input = Console.ReadLine();

                if (input.Length >= 1)
                {
                    Console.WriteLine(
                        "Hello, " +
                        input +
                        ", enter 'Exit' at any time to exit this app.");

                    goto AwaitFurtherInstruction;
                }
                else
                {
                    goto DisplayGreeting;
                }
            }

            AwaitFurtherInstruction:
            {
                input = Console.ReadLine();

                if(input.ToLower() == "exit")
                {
                    input = String.Empty;

                    Environment.Exit(0);
                }
                else
                {
                    goto AwaitFurtherInstruction;
                }
            }
        }
    }
}
```

Lea Creación de una aplicación de consola con un editor de texto sin formato y el compilador de

C # (csc.exe) en línea: https://riptutorial.com/es/csharp/topic/6676/creacion-de-una-aplicacion-de-consola-con-un-editor-de-texto-sin-formato-y-el-compilador-de-c-sharp--csc-exe-

# Capítulo 45: Creando un cuadro de mensaje propio en la aplicación Windows Form

## Introducción

Primero necesitamos saber qué es un MessageBox ...

El control MessageBox muestra un mensaje con el texto especificado y se puede personalizar especificando una imagen personalizada, títulos y conjuntos de botones (estos conjuntos de botones permiten al usuario elegir más de una respuesta básica de sí / no).

Al crear nuestro propio MessageBox, podemos reutilizar ese Control MessageBox en cualquier aplicación nueva simplemente usando la dll generada o copiando el archivo que contiene la clase.

## Sintaxis

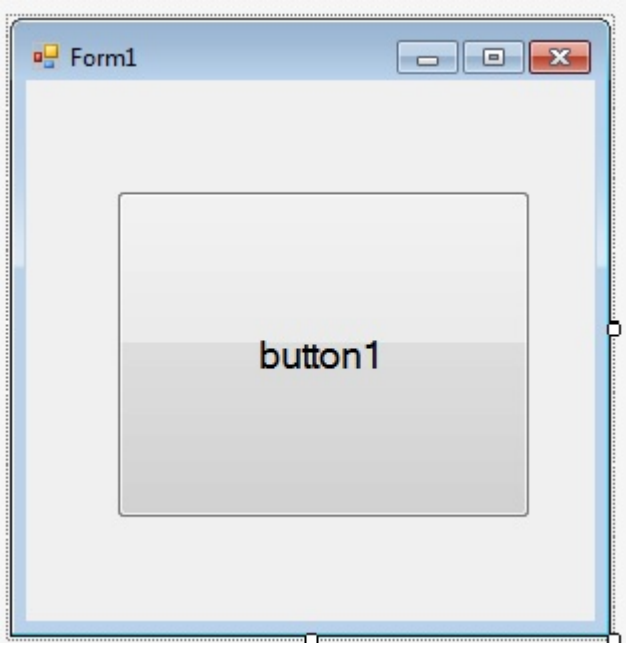
- 'static DialogResult result = DialogResult.No; // DialogResult es devuelto por los diálogos después del despido. '

## Examples

### Creando Control de MessageBox Propio.

Para crear nuestro propio control MessageBox, simplemente siga la guía a continuación ...

1. Abre tu instancia de Visual Studio (VS 2008/2010/2012/2015/2017)
2. Vaya a la barra de herramientas en la parte superior y haga clic en Archivo -> Nuevo proyecto -> Aplicación de Windows Forms -> Asigne un nombre al proyecto y luego haga clic en Aceptar.
3. Una vez cargado, arrastre y suelte un control de botón de la Caja de herramientas (que se encuentra a la izquierda) en el formulario (como se muestra a continuación).



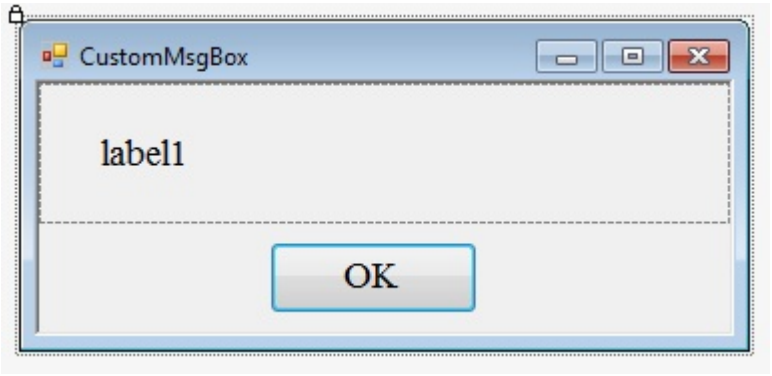

4. Haga doble clic en el botón y el Entorno de desarrollo integrado generará automáticamente el controlador de eventos de clic para usted.

5. Edite el código del formulario para que tenga el siguiente aspecto (puede hacer clic con el botón derecho en el formulario y hacer clic en Editar código):

```
namespace MsgBoxExample {
    public partial class MsgBoxExampleForm : Form {
        //Constructor, called when the class is initialised.
        public MsgBoxExampleForm() {
            InitializeComponent();
        }

        //Called whenever the button is clicked.
        private void btnShowMessageBox_Click(object sender, EventArgs e) {
           CustomMsgBox.Show($"I'm a {nameof(CustomMsgBox)}!", "MSG", "OK");
        }
    }
}
```

6. Explorador de soluciones -> Haga clic derecho en su proyecto -> Agregar -> Windows Form y establezca el nombre como "CustomMsgBox.cs"

7. Arrastre un control de botón y etiqueta desde la Caja de herramientas hasta el formulario (después de hacerlo tendrá un aspecto similar al formulario siguiente):




8. Ahora escriba el siguiente código en el formulario recién creado:

```
private DialogResult result = DialogResult.No;
public static DialogResult Show(string text, string caption, string btnOkText) {
    var msgBox = new CustomMsgBox();
    msgBox.lblText.Text = text; //The text for the label...
    msgBox.Text = caption; //Title of form
    msgBox.btnOk.Text = btnOkText; //Text on the button
    //This method is blocking, and will only return once the user
    //clicks ok or closes the form.
    msgBox.ShowDialog();
    return result;
}

private void btnOk_Click(object sender, EventArgs e) {
    result = DialogResult.Yes;
    MsgBox.Close();
}
```

9. Ahora ejecuta el programa simplemente presionando la tecla F5. Felicitaciones, has hecho un control reutilizable.

## Cómo usar el control MessageBox creado en otra aplicación de Windows Form.

Para encontrar sus archivos .cs existentes, haga clic con el botón derecho en el proyecto en su instancia de Visual Studio y haga clic en Abrir carpeta en el Explorador de archivos.

1. Visual Studio -> Su proyecto actual (Windows Form) -> Explorador de soluciones -> Nombre del proyecto -> Clic derecho -> Agregar -> Elemento existente -> Luego localice su archivo .cs existente.

2. Ahora hay una última cosa que hacer para usar el control. Agregue una declaración de uso a su código, para que su ensamblaje conozca sus dependencias.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
.
.
```

```
.
using CustomMsgBox; //Here's the using statement for our dependency.
```

3. Para mostrar el cuadro de mensaje, simplemente use lo siguiente ...

   CustomMsgBox.Show ("Tu mensaje para el cuadro de mensaje ...", "MSG", "OK");

Lea Creando un cuadro de mensaje propio en la aplicación Windows Form en línea: https://riptutorial.com/es/csharp/topic/9788/creando-un-cuadro-de-mensaje-propio-en-la-aplicacion-windows-form

# Capítulo 46: Criptografía (System.Security.Cryptography)

## Examples

### Ejemplos modernos de cifrado autenticado simétrico de una cadena

La criptografía es algo muy difícil y después de pasar mucho tiempo leyendo diferentes ejemplos y viendo lo fácil que es introducir algún tipo de vulnerabilidad, encontré una respuesta escrita originalmente por @jbtule que creo que es muy buena. Disfruta leyendo:

"La mejor práctica general para el cifrado simétrico es utilizar el cifrado autenticado con datos asociados (AEAD), sin embargo, esto no forma parte de las bibliotecas criptográficas .net estándar. Por lo tanto, el primer ejemplo utiliza AES256 y luego HMAC256 , un cifrado de dos pasos. MAC , que requiere más sobrecarga y más teclas.

El segundo ejemplo utiliza la práctica más sencilla de AES256- GCM utilizando el código abierto Bouncy Castle (a través de nuget).

Ambos ejemplos tienen una función principal que toma una cadena de mensajes secretas, una (s) clave (s) y una carga útil no secreta opcional y devuelve una cadena cifrada autenticada opcionalmente con los datos no secretos. Lo ideal sería utilizarlos con claves de 256 bits generadas al azar, ver `NewKey()` .

Ambos ejemplos también tienen métodos de ayuda que utilizan una contraseña de cadena para generar las claves. Estos métodos de ayuda se proporcionan como una conveniencia para coincidir con otros ejemplos, sin embargo, son *mucho menos seguros* porque la fortaleza de la contraseña será *mucho más débil que una clave de 256 bits* .

**Actualización: Se** agregaron las sobrecargas de `byte[]` , y solo el Gist tiene el formato completo con 4 espacios de sangría y api docs debido a los límites de respuesta de StackOverflow ".

---

**.NET incorporado cifrado (AES) -Entonces-MAC (HMAC) [Gist]**

```
/*
 * This work (Modern Encryption of a String C#, by James Tuley),
 * identified by James Tuley, is free of known copyright restrictions.
 * https://gist.github.com/4336842
 * http://creativecommons.org/publicdomain/mark/1.0/
 */

using System;
using System.IO;
using System.Security.Cryptography;
using System.Text;

namespace Encryption
```

```
{
  public static class AESThenHMAC
  {
    private static readonly RandomNumberGenerator Random = RandomNumberGenerator.Create();

    //Preconfigured Encryption Parameters
    public static readonly int BlockBitSize = 128;
    public static readonly int KeyBitSize = 256;

    //Preconfigured Password Key Derivation Parameters
    public static readonly int SaltBitSize = 64;
    public static readonly int Iterations = 10000;
    public static readonly int MinPasswordLength = 12;

    /// <summary>
    /// Helper that generates a random key on each call.
    /// </summary>
    /// <returns></returns>
    public static byte[] NewKey()
    {
      var key = new byte[KeyBitSize / 8];
      Random.GetBytes(key);
      return key;
    }

    /// <summary>
    /// Simple Encryption (AES) then Authentication (HMAC) for a UTF8 Message.
    /// </summary>
    /// <param name="secretMessage">The secret message.</param>
    /// <param name="cryptKey">The crypt key.</param>
    /// <param name="authKey">The auth key.</param>
    /// <param name="nonSecretPayload">(Optional) Non-Secret Payload.</param>
    /// <returns>
    /// Encrypted Message
    /// </returns>
    /// <exception cref="System.ArgumentException">Secret Message
Required!;secretMessage</exception>
    /// <remarks>
    /// Adds overhead of (Optional-Payload + BlockSize(16) + Message-Padded-To-Blocksize +
HMac-Tag(32)) * 1.33 Base64
    /// </remarks>
    public static string SimpleEncrypt(string secretMessage, byte[] cryptKey, byte[] authKey,
                        byte[] nonSecretPayload = null)
    {
      if (string.IsNullOrEmpty(secretMessage))
        throw new ArgumentException("Secret Message Required!", "secretMessage");

      var plainText = Encoding.UTF8.GetBytes(secretMessage);
      var cipherText = SimpleEncrypt(plainText, cryptKey, authKey, nonSecretPayload);
      return Convert.ToBase64String(cipherText);
    }

    /// <summary>
    /// Simple Authentication (HMAC) then Decryption (AES) for a secrets UTF8 Message.
    /// </summary>
    /// <param name="encryptedMessage">The encrypted message.</param>
    /// <param name="cryptKey">The crypt key.</param>
    /// <param name="authKey">The auth key.</param>
    /// <param name="nonSecretPayloadLength">Length of the non secret payload.</param>
    /// <returns>
    /// Decrypted Message
```

```
    /// </returns>
    /// <exception cref="System.ArgumentException">Encrypted Message
Required!;encryptedMessage</exception>
    public static string SimpleDecrypt(string encryptedMessage, byte[] cryptKey, byte[]
authKey,
                        int nonSecretPayloadLength = 0)
    {
      if (string.IsNullOrWhiteSpace(encryptedMessage))
        throw new ArgumentException("Encrypted Message Required!", "encryptedMessage");

      var cipherText = Convert.FromBase64String(encryptedMessage);
      var plainText = SimpleDecrypt(cipherText, cryptKey, authKey, nonSecretPayloadLength);
      return plainText == null ? null : Encoding.UTF8.GetString(plainText);
    }

    /// <summary>
    /// Simple Encryption (AES) then Authentication (HMAC) of a UTF8 message
    /// using Keys derived from a Password (PBKDF2).
    /// </summary>
    /// <param name="secretMessage">The secret message.</param>
    /// <param name="password">The password.</param>
    /// <param name="nonSecretPayload">The non secret payload.</param>
    /// <returns>
    /// Encrypted Message
    /// </returns>
    /// <exception cref="System.ArgumentException">password</exception>
    /// <remarks>
    /// Significantly less secure than using random binary keys.
    /// Adds additional non secret payload for key generation parameters.
    /// </remarks>
    public static string SimpleEncryptWithPassword(string secretMessage, string password,
                            byte[] nonSecretPayload = null)
    {
      if (string.IsNullOrEmpty(secretMessage))
        throw new ArgumentException("Secret Message Required!", "secretMessage");

      var plainText = Encoding.UTF8.GetBytes(secretMessage);
      var cipherText = SimpleEncryptWithPassword(plainText, password, nonSecretPayload);
      return Convert.ToBase64String(cipherText);
    }

    /// <summary>
    /// Simple Authentication (HMAC) and then Descryption (AES) of a UTF8 Message
    /// using keys derived from a password (PBKDF2).
    /// </summary>
    /// <param name="encryptedMessage">The encrypted message.</param>
    /// <param name="password">The password.</param>
    /// <param name="nonSecretPayloadLength">Length of the non secret payload.</param>
    /// <returns>
    /// Decrypted Message
    /// </returns>
    /// <exception cref="System.ArgumentException">Encrypted Message
Required!;encryptedMessage</exception>
    /// <remarks>
    /// Significantly less secure than using random binary keys.
    /// </remarks>
    public static string SimpleDecryptWithPassword(string encryptedMessage, string password,
                            int nonSecretPayloadLength = 0)
    {
      if (string.IsNullOrWhiteSpace(encryptedMessage))
        throw new ArgumentException("Encrypted Message Required!", "encryptedMessage");
```

```
      var cipherText = Convert.FromBase64String(encryptedMessage);
      var plainText = SimpleDecryptWithPassword(cipherText, password, nonSecretPayloadLength);
      return plainText == null ? null : Encoding.UTF8.GetString(plainText);
    }

    public static byte[] SimpleEncrypt(byte[] secretMessage, byte[] cryptKey, byte[] authKey,
byte[] nonSecretPayload = null)
    {
      //User Error Checks
      if (cryptKey == null || cryptKey.Length != KeyBitSize / 8)
        throw new ArgumentException(String.Format("Key needs to be {0} bit!", KeyBitSize),
"cryptKey");

      if (authKey == null || authKey.Length != KeyBitSize / 8)
        throw new ArgumentException(String.Format("Key needs to be {0} bit!", KeyBitSize),
"authKey");

      if (secretMessage == null || secretMessage.Length < 1)
        throw new ArgumentException("Secret Message Required!", "secretMessage");

      //non-secret payload optional
      nonSecretPayload = nonSecretPayload ?? new byte[] { };

      byte[] cipherText;
      byte[] iv;

      using (var aes = new AesManaged
      {
        KeySize = KeyBitSize,
        BlockSize = BlockBitSize,
        Mode = CipherMode.CBC,
        Padding = PaddingMode.PKCS7
      })
      {

        //Use random IV
        aes.GenerateIV();
        iv = aes.IV;

        using (var encrypter = aes.CreateEncryptor(cryptKey, iv))
        using (var cipherStream = new MemoryStream())
        {
          using (var cryptoStream = new CryptoStream(cipherStream, encrypter,
CryptoStreamMode.Write))
          using (var binaryWriter = new BinaryWriter(cryptoStream))
          {
            //Encrypt Data
            binaryWriter.Write(secretMessage);
          }

          cipherText = cipherStream.ToArray();
        }

      }

      //Assemble encrypted message and add authentication
      using (var hmac = new HMACSHA256(authKey))
      using (var encryptedStream = new MemoryStream())
      {
        using (var binaryWriter = new BinaryWriter(encryptedStream))
```

```
        {
          //Prepend non-secret payload if any
          binaryWriter.Write(nonSecretPayload);
          //Prepend IV
          binaryWriter.Write(iv);
          //Write Ciphertext
          binaryWriter.Write(cipherText);
          binaryWriter.Flush();

          //Authenticate all data
          var tag = hmac.ComputeHash(encryptedStream.ToArray());
          //Postpend tag
          binaryWriter.Write(tag);
        }
        return encryptedStream.ToArray();
      }

    }

    public static byte[] SimpleDecrypt(byte[] encryptedMessage, byte[] cryptKey, byte[]
authKey, int nonSecretPayloadLength = 0)
    {

      //Basic Usage Error Checks
      if (cryptKey == null || cryptKey.Length != KeyBitSize / 8)
        throw new ArgumentException(String.Format("CryptKey needs to be {0} bit!",
KeyBitSize), "cryptKey");

      if (authKey == null || authKey.Length != KeyBitSize / 8)
        throw new ArgumentException(String.Format("AuthKey needs to be {0} bit!", KeyBitSize),
"authKey");

      if (encryptedMessage == null || encryptedMessage.Length == 0)
        throw new ArgumentException("Encrypted Message Required!", "encryptedMessage");

      using (var hmac = new HMACSHA256(authKey))
      {
        var sentTag = new byte[hmac.HashSize / 8];
        //Calculate Tag
        var calcTag = hmac.ComputeHash(encryptedMessage, 0, encryptedMessage.Length -
sentTag.Length);
        var ivLength = (BlockBitSize / 8);

        //if message length is to small just return null
        if (encryptedMessage.Length < sentTag.Length + nonSecretPayloadLength + ivLength)
          return null;

        //Grab Sent Tag
        Array.Copy(encryptedMessage, encryptedMessage.Length - sentTag.Length, sentTag, 0,
sentTag.Length);

        //Compare Tag with constant time comparison
        var compare = 0;
        for (var i = 0; i < sentTag.Length; i++)
          compare |= sentTag[i] ^ calcTag[i];

        //if message doesn't authenticate return null
        if (compare != 0)
          return null;

        using (var aes = new AesManaged
```

```
        {
          KeySize = KeyBitSize,
          BlockSize = BlockBitSize,
          Mode = CipherMode.CBC,
          Padding = PaddingMode.PKCS7
        })
        {

          //Grab IV from message
          var iv = new byte[ivLength];
          Array.Copy(encryptedMessage, nonSecretPayloadLength, iv, 0, iv.Length);

          using (var decrypter = aes.CreateDecryptor(cryptKey, iv))
          using (var plainTextStream = new MemoryStream())
          {
            using (var decrypterStream = new CryptoStream(plainTextStream, decrypter,
CryptoStreamMode.Write))
            using (var binaryWriter = new BinaryWriter(decrypterStream))
            {
              //Decrypt Cipher Text from Message
              binaryWriter.Write(
                encryptedMessage,
                nonSecretPayloadLength + iv.Length,
                encryptedMessage.Length - nonSecretPayloadLength - iv.Length - sentTag.Length
              );
            }
            //Return Plain Text
            return plainTextStream.ToArray();
          }
        }
      }
    }

    public static byte[] SimpleEncryptWithPassword(byte[] secretMessage, string password,
byte[] nonSecretPayload = null)
    {
      nonSecretPayload = nonSecretPayload ?? new byte[] {};

      //User Error Checks
      if (string.IsNullOrWhiteSpace(password) || password.Length < MinPasswordLength)
        throw new ArgumentException(String.Format("Must have a password of at least {0}
characters!", MinPasswordLength), "password");

      if (secretMessage == null || secretMessage.Length ==0)
        throw new ArgumentException("Secret Message Required!", "secretMessage");

      var payload = new byte[((SaltBitSize / 8) * 2) + nonSecretPayload.Length];

      Array.Copy(nonSecretPayload, payload, nonSecretPayload.Length);
      int payloadIndex = nonSecretPayload.Length;

      byte[] cryptKey;
      byte[] authKey;
      //Use Random Salt to prevent pre-generated weak password attacks.
      using (var generator = new Rfc2898DeriveBytes(password, SaltBitSize / 8, Iterations))
      {
        var salt = generator.Salt;

        //Generate Keys
        cryptKey = generator.GetBytes(KeyBitSize / 8);
```

```
        //Create Non Secret Payload
        Array.Copy(salt, 0, payload, payloadIndex, salt.Length);
        payloadIndex += salt.Length;
      }

      //Deriving separate key, might be less efficient than using HKDF,
      //but now compatible with RNEncryptor which had a very similar wireformat and requires
less code than HKDF.
      using (var generator = new Rfc2898DeriveBytes(password, SaltBitSize / 8, Iterations))
      {
        var salt = generator.Salt;

        //Generate Keys
        authKey = generator.GetBytes(KeyBitSize / 8);

        //Create Rest of Non Secret Payload
        Array.Copy(salt, 0, payload, payloadIndex, salt.Length);
      }

      return SimpleEncrypt(secretMessage, cryptKey, authKey, payload);
    }

    public static byte[] SimpleDecryptWithPassword(byte[] encryptedMessage, string password,
int nonSecretPayloadLength = 0)
    {
      //User Error Checks
      if (string.IsNullOrWhiteSpace(password) || password.Length < MinPasswordLength)
        throw new ArgumentException(String.Format("Must have a password of at least {0}
characters!", MinPasswordLength), "password");

      if (encryptedMessage == null || encryptedMessage.Length == 0)
        throw new ArgumentException("Encrypted Message Required!", "encryptedMessage");

      var cryptSalt = new byte[SaltBitSize / 8];
      var authSalt = new byte[SaltBitSize / 8];

      //Grab Salt from Non-Secret Payload
      Array.Copy(encryptedMessage, nonSecretPayloadLength, cryptSalt, 0, cryptSalt.Length);
      Array.Copy(encryptedMessage, nonSecretPayloadLength + cryptSalt.Length, authSalt, 0,
authSalt.Length);

      byte[] cryptKey;
      byte[] authKey;

      //Generate crypt key
      using (var generator = new Rfc2898DeriveBytes(password, cryptSalt, Iterations))
      {
        cryptKey = generator.GetBytes(KeyBitSize / 8);
      }
      //Generate auth key
      using (var generator = new Rfc2898DeriveBytes(password, authSalt, Iterations))
      {
        authKey = generator.GetBytes(KeyBitSize / 8);
      }

      return SimpleDecrypt(encryptedMessage, cryptKey, authKey, cryptSalt.Length +
authSalt.Length + nonSecretPayloadLength);
    }
  }
}
```

### Castillo hinchable AES-GCM [Gist]

```
/*
 * This work (Modern Encryption of a String C#, by James Tuley),
 * identified by James Tuley, is free of known copyright restrictions.
 * https://gist.github.com/4336842
 * http://creativecommons.org/publicdomain/mark/1.0/
 */

using System;
using System.IO;
using System.Text;
using Org.BouncyCastle.Crypto;
using Org.BouncyCastle.Crypto.Engines;
using Org.BouncyCastle.Crypto.Generators;
using Org.BouncyCastle.Crypto.Modes;
using Org.BouncyCastle.Crypto.Parameters;
using Org.BouncyCastle.Security;
namespace Encryption
{

  public static class AESGCM
  {
    private static readonly SecureRandom Random = new SecureRandom();

    //Preconfigured Encryption Parameters
    public static readonly int NonceBitSize = 128;
    public static readonly int MacBitSize = 128;
    public static readonly int KeyBitSize = 256;

    //Preconfigured Password Key Derivation Parameters
    public static readonly int SaltBitSize = 128;
    public static readonly int Iterations = 10000;
    public static readonly int MinPasswordLength = 12;


    /// <summary>
    /// Helper that generates a random new key on each call.
    /// </summary>
    /// <returns></returns>
    public static byte[] NewKey()
    {
      var key = new byte[KeyBitSize / 8];
      Random.NextBytes(key);
      return key;
    }

    /// <summary>
    /// Simple Encryption And Authentication (AES-GCM) of a UTF8 string.
    /// </summary>
    /// <param name="secretMessage">The secret message.</param>
    /// <param name="key">The key.</param>
    /// <param name="nonSecretPayload">Optional non-secret payload.</param>
    /// <returns>
    /// Encrypted Message
    /// </returns>
    /// <exception cref="System.ArgumentException">Secret Message
Required!;secretMessage</exception>
    /// <remarks>
    /// Adds overhead of (Optional-Payload + BlockSize(16) + Message +  HMac-Tag(16)) * 1.33
Base64
```

```
    /// </remarks>
    public static string SimpleEncrypt(string secretMessage, byte[] key, byte[]
nonSecretPayload = null)
    {
      if (string.IsNullOrEmpty(secretMessage))
        throw new ArgumentException("Secret Message Required!", "secretMessage");

      var plainText = Encoding.UTF8.GetBytes(secretMessage);
      var cipherText = SimpleEncrypt(plainText, key, nonSecretPayload);
      return Convert.ToBase64String(cipherText);
    }


    /// <summary>
    /// Simple Decryption & Authentication (AES-GCM) of a UTF8 Message
    /// </summary>
    /// <param name="encryptedMessage">The encrypted message.</param>
    /// <param name="key">The key.</param>
    /// <param name="nonSecretPayloadLength">Length of the optional non-secret
payload.</param>
    /// <returns>Decrypted Message</returns>
    public static string SimpleDecrypt(string encryptedMessage, byte[] key, int
nonSecretPayloadLength = 0)
    {
      if (string.IsNullOrEmpty(encryptedMessage))
        throw new ArgumentException("Encrypted Message Required!", "encryptedMessage");

      var cipherText = Convert.FromBase64String(encryptedMessage);
      var plainText = SimpleDecrypt(cipherText, key, nonSecretPayloadLength);
      return plainText == null ? null : Encoding.UTF8.GetString(plainText);
    }

    /// <summary>
    /// Simple Encryption And Authentication (AES-GCM) of a UTF8 String
    /// using key derived from a password (PBKDF2).
    /// </summary>
    /// <param name="secretMessage">The secret message.</param>
    /// <param name="password">The password.</param>
    /// <param name="nonSecretPayload">The non secret payload.</param>
    /// <returns>
    /// Encrypted Message
    /// </returns>
    /// <remarks>
    /// Significantly less secure than using random binary keys.
    /// Adds additional non secret payload for key generation parameters.
    /// </remarks>
    public static string SimpleEncryptWithPassword(string secretMessage, string password,
                            byte[] nonSecretPayload = null)
    {
      if (string.IsNullOrEmpty(secretMessage))
        throw new ArgumentException("Secret Message Required!", "secretMessage");

      var plainText = Encoding.UTF8.GetBytes(secretMessage);
      var cipherText = SimpleEncryptWithPassword(plainText, password, nonSecretPayload);
      return Convert.ToBase64String(cipherText);
    }


    /// <summary>
    /// Simple Decryption and Authentication (AES-GCM) of a UTF8 message
    /// using a key derived from a password (PBKDF2)
```

```
    /// </summary>
    /// <param name="encryptedMessage">The encrypted message.</param>
    /// <param name="password">The password.</param>
    /// <param name="nonSecretPayloadLength">Length of the non secret payload.</param>
    /// <returns>
    /// Decrypted Message
    /// </returns>
    /// <exception cref="System.ArgumentException">Encrypted Message
Required!;encryptedMessage</exception>
    /// <remarks>
    /// Significantly less secure than using random binary keys.
    /// </remarks>
    public static string SimpleDecryptWithPassword(string encryptedMessage, string password,
                            int nonSecretPayloadLength = 0)
    {
      if (string.IsNullOrWhiteSpace(encryptedMessage))
        throw new ArgumentException("Encrypted Message Required!", "encryptedMessage");

      var cipherText = Convert.FromBase64String(encryptedMessage);
      var plainText = SimpleDecryptWithPassword(cipherText, password, nonSecretPayloadLength);
      return plainText == null ? null : Encoding.UTF8.GetString(plainText);
    }

    public static byte[] SimpleEncrypt(byte[] secretMessage, byte[] key, byte[]
nonSecretPayload = null)
    {
      //User Error Checks
      if (key == null || key.Length != KeyBitSize / 8)
        throw new ArgumentException(String.Format("Key needs to be {0} bit!", KeyBitSize),
"key");

      if (secretMessage == null || secretMessage.Length == 0)
        throw new ArgumentException("Secret Message Required!", "secretMessage");

      //Non-secret Payload Optional
      nonSecretPayload = nonSecretPayload ?? new byte[] { };

      //Using random nonce large enough not to repeat
      var nonce = new byte[NonceBitSize / 8];
      Random.NextBytes(nonce, 0, nonce.Length);

      var cipher = new GcmBlockCipher(new AesFastEngine());
      var parameters = new AeadParameters(new KeyParameter(key), MacBitSize, nonce,
nonSecretPayload);
      cipher.Init(true, parameters);

      //Generate Cipher Text With Auth Tag
      var cipherText = new byte[cipher.GetOutputSize(secretMessage.Length)];
      var len = cipher.ProcessBytes(secretMessage, 0, secretMessage.Length, cipherText, 0);
      cipher.DoFinal(cipherText, len);

      //Assemble Message
      using (var combinedStream = new MemoryStream())
      {
        using (var binaryWriter = new BinaryWriter(combinedStream))
        {
          //Prepend Authenticated Payload
          binaryWriter.Write(nonSecretPayload);
          //Prepend Nonce
          binaryWriter.Write(nonce);
          //Write Cipher Text
```

```
          binaryWriter.Write(cipherText);
        }
        return combinedStream.ToArray();
      }
    }

    public static byte[] SimpleDecrypt(byte[] encryptedMessage, byte[] key, int
nonSecretPayloadLength = 0)
    {
      //User Error Checks
      if (key == null || key.Length != KeyBitSize / 8)
        throw new ArgumentException(String.Format("Key needs to be {0} bit!", KeyBitSize),
"key");

      if (encryptedMessage == null || encryptedMessage.Length == 0)
        throw new ArgumentException("Encrypted Message Required!", "encryptedMessage");

      using (var cipherStream = new MemoryStream(encryptedMessage))
      using (var cipherReader = new BinaryReader(cipherStream))
      {
        //Grab Payload
        var nonSecretPayload = cipherReader.ReadBytes(nonSecretPayloadLength);

        //Grab Nonce
        var nonce = cipherReader.ReadBytes(NonceBitSize / 8);

        var cipher = new GcmBlockCipher(new AesFastEngine());
        var parameters = new AeadParameters(new KeyParameter(key), MacBitSize, nonce,
nonSecretPayload);
        cipher.Init(false, parameters);

        //Decrypt Cipher Text
        var cipherText = cipherReader.ReadBytes(encryptedMessage.Length -
nonSecretPayloadLength - nonce.Length);
        var plainText = new byte[cipher.GetOutputSize(cipherText.Length)];

        try
        {
          var len = cipher.ProcessBytes(cipherText, 0, cipherText.Length, plainText, 0);
          cipher.DoFinal(plainText, len);

        }
        catch (InvalidCipherTextException)
        {
          //Return null if it doesn't authenticate
          return null;
        }

        return plainText;
      }

    }

    public static byte[] SimpleEncryptWithPassword(byte[] secretMessage, string password,
byte[] nonSecretPayload = null)
    {
      nonSecretPayload = nonSecretPayload ?? new byte[] {};

      //User Error Checks
      if (string.IsNullOrWhiteSpace(password) || password.Length < MinPasswordLength)
        throw new ArgumentException(String.Format("Must have a password of at least {0}
```

```
characters!", MinPasswordLength), "password");

      if (secretMessage == null || secretMessage.Length == 0)
        throw new ArgumentException("Secret Message Required!", "secretMessage");

      var generator = new Pkcs5S2ParametersGenerator();

      //Use Random Salt to minimize pre-generated weak password attacks.
      var salt = new byte[SaltBitSize / 8];
      Random.NextBytes(salt);

      generator.Init(
        PbeParametersGenerator.Pkcs5PasswordToBytes(password.ToCharArray()),
        salt,
        Iterations);

      //Generate Key
      var key = (KeyParameter)generator.GenerateDerivedMacParameters(KeyBitSize);

      //Create Full Non Secret Payload
      var payload = new byte[salt.Length + nonSecretPayload.Length];
      Array.Copy(nonSecretPayload, payload, nonSecretPayload.Length);
      Array.Copy(salt,0, payload,nonSecretPayload.Length, salt.Length);

      return SimpleEncrypt(secretMessage, key.GetKey(), payload);
    }

    public static byte[] SimpleDecryptWithPassword(byte[] encryptedMessage, string password,
int nonSecretPayloadLength = 0)
    {
      //User Error Checks
      if (string.IsNullOrWhiteSpace(password) || password.Length < MinPasswordLength)
        throw new ArgumentException(String.Format("Must have a password of at least {0}
characters!", MinPasswordLength), "password");

      if (encryptedMessage == null || encryptedMessage.Length == 0)
        throw new ArgumentException("Encrypted Message Required!", "encryptedMessage");

      var generator = new Pkcs5S2ParametersGenerator();

      //Grab Salt from Payload
      var salt = new byte[SaltBitSize / 8];
      Array.Copy(encryptedMessage, nonSecretPayloadLength, salt, 0, salt.Length);

      generator.Init(
        PbeParametersGenerator.Pkcs5PasswordToBytes(password.ToCharArray()),
        salt,
        Iterations);

      //Generate Key
      var key = (KeyParameter)generator.GenerateDerivedMacParameters(KeyBitSize);

      return SimpleDecrypt(encryptedMessage, key.GetKey(), salt.Length +
nonSecretPayloadLength);
    }
  }
}
```

## Introducción al cifrado simétrico y asimétrico

Puede mejorar la seguridad para el tránsito o almacenamiento de datos implementando técnicas de cifrado. Básicamente, existen dos enfoques cuando se utiliza *System.Security.Cryptography* : **simétrico** y **asimétrico.**

---

## Cifrado simétrico

Este método utiliza una clave privada para realizar la transformación de datos.

Pros:

- Los algoritmos simétricos consumen menos recursos y son más rápidos que los asimétricos.
- La cantidad de datos que puede cifrar es ilimitada.

Contras:

- El cifrado y el descifrado utilizan la misma clave. Alguien podrá descifrar sus datos si la clave está comprometida.
- Podría terminar con muchas claves secretas diferentes para administrar si elige usar una clave secreta diferente para datos diferentes.

Bajo System.Security.Cryptography tiene diferentes clases que realizan cifrado simétrico, se conocen como cifrados de bloque :

- AesManaged (algoritmo AES ).
- AesCryptoServiceProvider ( AES algoritmo FIPS 140-2 queja ).
- DESCryptoServiceProvider (algoritmo DES ).
- RC2CryptoServiceProvider (algoritmo Rivest Cipher 2 ).
- RijndaelManaged (algoritmo AES ). *Nota* : RijndaelManaged **no** es una queja FIPS-197 .
- TripleDES (algoritmo TripleDES ).

---

## Cifrado asimétrico

Este método utiliza una combinación de claves públicas y privadas para realizar la transformación de datos.

Pros:

- Utiliza claves más grandes que los algoritmos simétricos, por lo que son menos susceptibles de romperse con el uso de la fuerza bruta.
- Es más fácil garantizar quién puede cifrar y descifrar los datos porque se basa en dos claves (pública y privada).

Contras:

- Hay un límite en la cantidad de datos que puede cifrar. El límite es diferente para cada algoritmo y suele ser proporcional al tamaño de la clave del algoritmo. Por ejemplo, un

objeto RSACryptoServiceProvider con una longitud de clave de 1.024 bits solo puede cifrar un mensaje que sea más pequeño que 128 bytes.
- Los algoritmos asimétricos son muy lentos en comparación con los algoritmos simétricos.

En System.Security.Cryptography tiene acceso a diferentes clases que realizan cifrado asimétrico:

- DSACryptoServiceProvider ( algoritmo de algoritmo de firma digital )
- RSACryptoServiceProvider ( algoritmo de algoritmo RSA )

## Hash de contraseña

¡Las contraseñas nunca deben almacenarse como texto plano! Deben incluirse un sal generado aleatoriamente (para defenderse de los ataques de la tabla arco iris) utilizando un algoritmo de hashing de contraseña lento. Se puede usar un alto número de iteraciones (> 10k) para ralentizar los ataques de fuerza bruta. Un retraso de ~ 100ms es aceptable para un usuario que inicia sesión, pero dificulta la ruptura de una contraseña larga. Al elegir una serie de iteraciones, debe utilizar el valor máximo tolerable para su aplicación y aumentarla a medida que mejore el rendimiento del equipo. También deberá considerar detener las solicitudes repetidas que podrían usarse como un ataque DoS.

Cuando hash por primera vez se puede generar una sal para ti, el hash y la sal resultantes se pueden almacenar en un archivo.

```
private void firstHash(string userName, string userPassword, int numberOfItterations)
{
    Rfc2898DeriveBytes PBKDF2 = new Rfc2898DeriveBytes(userPassword, 8, numberOfItterations);
//Hash the password with a 8 byte salt
    byte[] hashedPassword = PBKDF2.GetBytes(20);    //Returns a 20 byte hash
    byte[] salt = PBKDF2.Salt;
    writeHashToFile(userName, hashedPassword, salt, numberOfItterations); //Store the hashed
password with the salt and number of itterations to check against future password entries
}
```

Verificando una contraseña de usuario existente, lea su hash y sal de un archivo y compare con el hash de la contraseña ingresada

```
private bool checkPassword(string userName, string userPassword, int numberOfItterations)
{
    byte[] usersHash = getUserHashFromFile(userName);
    byte[] userSalt = getUserSaltFromFile(userName);
    Rfc2898DeriveBytes PBKDF2 = new Rfc2898DeriveBytes(userPassword, userSalt,
numberOfItterations);    //Hash the password with the users salt
    byte[] hashedPassword = PBKDF2.GetBytes(20);    //Returns a 20 byte hash
    bool passwordsMach = comparePasswords(usersHash, hashedPassword);    //Compares byte
arrays
    return passwordsMach;
}
```

## Cifrado simple de archivos simétricos

El siguiente ejemplo de código muestra un medio rápido y fácil de cifrar y descifrar archivos utilizando el algoritmo de cifrado simétrico AES.

El código genera aleatoriamente los Vectores de Sal e Inicialización cada vez que se encripta un archivo, lo que significa que cifrar el mismo archivo con la misma contraseña siempre dará lugar a una salida diferente. La sal y el IV se escriben en el archivo de salida de modo que solo se requiere la contraseña para descifrarlo.

```
public static void ProcessFile(string inputPath, string password, bool encryptMode, string
outputPath)
{
    using (var cypher = new AesManaged())
    using (var fsIn = new FileStream(inputPath, FileMode.Open))
    using (var fsOut = new FileStream(outputPath, FileMode.Create))
    {
        const int saltLength = 256;
        var salt = new byte[saltLength];
        var iv = new byte[cypher.BlockSize / 8];

        if (encryptMode)
        {
            // Generate random salt and IV, then write them to file
            using (var rng = new RNGCryptoServiceProvider())
            {
                rng.GetBytes(salt);
                rng.GetBytes(iv);
            }
            fsOut.Write(salt, 0, salt.Length);
            fsOut.Write(iv, 0, iv.Length);
        }
        else
        {
            // Read the salt and IV from the file
            fsIn.Read(salt, 0, saltLength);
            fsIn.Read(iv, 0, iv.Length);
        }

        // Generate a secure password, based on the password and salt provided
        var pdb = new Rfc2898DeriveBytes(password, salt);
        var key = pdb.GetBytes(cypher.KeySize / 8);

        // Encrypt or decrypt the file
        using (var cryptoTransform = encryptMode
            ? cypher.CreateEncryptor(key, iv)
            : cypher.CreateDecryptor(key, iv))
        using (var cs = new CryptoStream(fsOut, cryptoTransform, CryptoStreamMode.Write))
        {
            fsIn.CopyTo(cs);
        }
    }
}
```

## Datos aleatorios criptográficamente seguros

Hay ocasiones en que la clase Random () del marco puede no considerarse suficientemente aleatoria, dado que se basa en un generador de números psuedo-aleatorios. Sin embargo, las clases Crypto del marco proporcionan algo más robusto en forma de RNGCryptoServiceProvider.

Los siguientes ejemplos de código demuestran cómo generar arrays, cadenas y números de bytes criptográficamente seguros.

**Matriz de bytes aleatorios**

```
public static byte[] GenerateRandomData(int length)
{
    var rnd = new byte[length];
    using (var rng = new RNGCryptoServiceProvider())
        rng.GetBytes(rnd);
    return rnd;
}
```

**Entero aleatorio** (con distribución uniforme)

```
public static int GenerateRandomInt(int minVal=0, int maxVal=100)
{
    var rnd = new byte[4];
    using (var rng = new RNGCryptoServiceProvider())
        rng.GetBytes(rnd);
    var i = Math.Abs(BitConverter.ToInt32(rnd, 0));
    return Convert.ToInt32(i % (maxVal - minVal + 1) + minVal);
}
```

**Cadena aleatoria**

```
public static string GenerateRandomString(int length, string allowableChars=null)
{
    if (string.IsNullOrEmpty(allowableChars))
        allowableChars = @"ABCDEFGHIJKLMNOPQRSTUVWXYZ";

    // Generate random data
    var rnd = new byte[length];
    using (var rng = new RNGCryptoServiceProvider())
        rng.GetBytes(rnd);

    // Generate the output string
    var allowable = allowableChars.ToCharArray();
    var l = allowable.Length;
    var chars = new char[length];
    for (var i = 0; i < length; i++)
        chars[i] = allowable[rnd[i] % l];

    return new string(chars);
}
```

## Cifrado rápido de archivos asimétricos

El cifrado asimétrico a menudo se considera preferible al cifrado simétrico para transferir mensajes a otras partes. Esto se debe principalmente a que niega muchos de los riesgos relacionados con el intercambio de una clave compartida y garantiza que, aunque cualquier persona con la clave pública pueda cifrar un mensaje para el destinatario deseado, solo ese destinatario puede descifrarlo. Desafortunadamente, el principal inconveniente de los algoritmos de cifrado asimétrico es que son significativamente más lentos que sus primos simétricos. Como

tal, el cifrado asimétrico de archivos, especialmente los grandes, a menudo puede ser un proceso muy computacional.

Para proporcionar seguridad y rendimiento, se puede adoptar un enfoque híbrido. Esto implica la generación aleatoria criptográfica de una clave y un vector de inicialización para el cifrado *simétrico* . Estos valores se encriptan luego utilizando un algoritmo *asimétrico* y se escriben en el archivo de salida, antes de usarlos para cifrar los datos de origen de forma *simétrica* y adjuntarlos a la salida.

Este enfoque proporciona un alto grado de rendimiento y seguridad, ya que los datos se cifran mediante un algoritmo simétrico (rápido) y la clave y iv, ambos generados aleatoriamente (seguro) se cifran mediante un algoritmo asimétrico (seguro). También tiene la ventaja adicional de que la misma carga útil cifrada en diferentes ocasiones tendrá un texto cifrado muy diferente, ya que las claves simétricas se generan aleatoriamente cada vez.

La siguiente clase muestra el cifrado asimétrico de cadenas y matrices de bytes, así como el cifrado de archivos híbridos.

```
public static class AsymmetricProvider
{
    #region Key Generation
    public class KeyPair
    {
        public string PublicKey { get; set; }
        public string PrivateKey { get; set; }
    }

    public static KeyPair GenerateNewKeyPair(int keySize = 4096)
    {
        // KeySize is measured in bits. 1024 is the default, 2048 is better, 4096 is more
robust but takes a fair bit longer to generate.
        using (var rsa = new RSACryptoServiceProvider(keySize))
        {
            return new KeyPair {PublicKey = rsa.ToXmlString(false), PrivateKey =
rsa.ToXmlString(true)};
        }
    }

    #endregion

    #region Asymmetric Data Encryption and Decryption

    public static byte[] EncryptData(byte[] data, string publicKey)
    {
        using (var asymmetricProvider = new RSACryptoServiceProvider())
        {
            asymmetricProvider.FromXmlString(publicKey);
            return asymmetricProvider.Encrypt(data, true);
        }
    }

    public static byte[] DecryptData(byte[] data, string publicKey)
    {
        using (var asymmetricProvider = new RSACryptoServiceProvider())
        {
            asymmetricProvider.FromXmlString(publicKey);
```

```
            if (asymmetricProvider.PublicOnly)
                throw new Exception("The key provided is a public key and does not contain the
private key elements required for decryption");
            return asymmetricProvider.Decrypt(data, true);
        }
    }

    public static string EncryptString(string value, string publicKey)
    {
        return Convert.ToBase64String(EncryptData(Encoding.UTF8.GetBytes(value), publicKey));
    }

    public static string DecryptString(string value, string privateKey)
    {
        return Encoding.UTF8.GetString(EncryptData(Convert.FromBase64String(value),
privateKey));
    }

    #endregion

    #region Hybrid File Encryption and Decription

    public static void EncryptFile(string inputFilePath, string outputFilePath, string
publicKey)
    {
        using (var symmetricCypher = new AesManaged())
        {
            // Generate random key and IV for symmetric encryption
            var key = new byte[symmetricCypher.KeySize / 8];
            var iv = new byte[symmetricCypher.BlockSize / 8];
            using (var rng = new RNGCryptoServiceProvider())
            {
                rng.GetBytes(key);
                rng.GetBytes(iv);
            }

            // Encrypt the symmetric key and IV
            var buf = new byte[key.Length + iv.Length];
            Array.Copy(key, buf, key.Length);
            Array.Copy(iv, 0, buf, key.Length, iv.Length);
            buf = EncryptData(buf, publicKey);

            var bufLen = BitConverter.GetBytes(buf.Length);

            // Symmetrically encrypt the data and write it to the file, along with the
encrypted key and iv
            using (var cypherKey = symmetricCypher.CreateEncryptor(key, iv))
            using (var fsIn = new FileStream(inputFilePath, FileMode.Open))
            using (var fsOut = new FileStream(outputFilePath, FileMode.Create))
            using (var cs = new CryptoStream(fsOut, cypherKey, CryptoStreamMode.Write))
            {
                fsOut.Write(bufLen,0, bufLen.Length);
                fsOut.Write(buf, 0, buf.Length);
                fsIn.CopyTo(cs);
            }
        }
    }

    public static void DecryptFile(string inputFilePath, string outputFilePath, string
privateKey)
    {
```

```
        using (var symmetricCypher = new AesManaged())
        using (var fsIn = new FileStream(inputFilePath, FileMode.Open))
        {
            // Determine the length of the encrypted key and IV
            var buf = new byte[sizeof(int)];
            fsIn.Read(buf, 0, buf.Length);
            var bufLen = BitConverter.ToInt32(buf, 0);

            // Read the encrypted key and IV data from the file and decrypt using the
asymmetric algorithm
            buf = new byte[bufLen];
            fsIn.Read(buf, 0, buf.Length);
            buf = DecryptData(buf, privateKey);

            var key = new byte[symmetricCypher.KeySize / 8];
            var iv = new byte[symmetricCypher.BlockSize / 8];
            Array.Copy(buf, key, key.Length);
            Array.Copy(buf, key.Length, iv, 0, iv.Length);

            // Decript the file data using the symmetric algorithm
            using (var cypherKey = symmetricCypher.CreateDecryptor(key, iv))
            using (var fsOut = new FileStream(outputFilePath, FileMode.Create))
            using (var cs = new CryptoStream(fsOut, cypherKey, CryptoStreamMode.Write))
            {
                fsIn.CopyTo(cs);
            }
        }
    }

    #endregion

    #region Key Storage

    public static void WritePublicKey(string publicKeyFilePath, string publicKey)
    {
        File.WriteAllText(publicKeyFilePath, publicKey);
    }
    public static string ReadPublicKey(string publicKeyFilePath)
    {
        return File.ReadAllText(publicKeyFilePath);
    }

    private const string SymmetricSalt = "Stack_Overflow!"; // Change me!

    public static string ReadPrivateKey(string privateKeyFilePath, string password)
    {
        var salt = Encoding.UTF8.GetBytes(SymmetricSalt);
        var cypherText = File.ReadAllBytes(privateKeyFilePath);

        using (var cypher = new AesManaged())
        {
            var pdb = new Rfc2898DeriveBytes(password, salt);
            var key = pdb.GetBytes(cypher.KeySize / 8);
            var iv = pdb.GetBytes(cypher.BlockSize / 8);

            using (var decryptor = cypher.CreateDecryptor(key, iv))
            using (var msDecrypt = new MemoryStream(cypherText))
            using (var csDecrypt = new CryptoStream(msDecrypt, decryptor,
CryptoStreamMode.Read))
            using (var srDecrypt = new StreamReader(csDecrypt))
            {
```

```
                return srDecrypt.ReadToEnd();
            }
        }
    }

    public static void WritePrivateKey(string privateKeyFilePath, string privateKey, string
password)
    {
        var salt = Encoding.UTF8.GetBytes(SymmetricSalt);
        using (var cypher = new AesManaged())
        {
            var pdb = new Rfc2898DeriveBytes(password, salt);
            var key = pdb.GetBytes(cypher.KeySize / 8);
            var iv = pdb.GetBytes(cypher.BlockSize / 8);

            using (var encryptor = cypher.CreateEncryptor(key, iv))
            using (var fsEncrypt = new FileStream(privateKeyFilePath, FileMode.Create))
            using (var csEncrypt = new CryptoStream(fsEncrypt, encryptor,
CryptoStreamMode.Write))
            using (var swEncrypt = new StreamWriter(csEncrypt))
            {
                swEncrypt.Write(privateKey);
            }
        }
    }

    #endregion
}
```

Ejemplo de uso:

```
private static void HybridCryptoTest(string privateKeyPath, string privateKeyPassword, string
inputPath)
{
    // Setup the test
    var publicKeyPath = Path.ChangeExtension(privateKeyPath, ".public");
    var outputPath = Path.Combine(Path.ChangeExtension(inputPath, ".enc"));
    var testPath = Path.Combine(Path.ChangeExtension(inputPath, ".test"));

    if (!File.Exists(privateKeyPath))
    {
        var keys = AsymmetricProvider.GenerateNewKeyPair(2048);
        AsymmetricProvider.WritePublicKey(publicKeyPath, keys.PublicKey);
        AsymmetricProvider.WritePrivateKey(privateKeyPath, keys.PrivateKey,
privateKeyPassword);
    }

    // Encrypt the file
    var publicKey = AsymmetricProvider.ReadPublicKey(publicKeyPath);
    AsymmetricProvider.EncryptFile(inputPath, outputPath, publicKey);

    // Decrypt it again to compare against the source file
    var privateKey = AsymmetricProvider.ReadPrivateKey(privateKeyPath, privateKeyPassword);
    AsymmetricProvider.DecryptFile(outputPath, testPath, privateKey);

    // Check that the two files match
    var source = File.ReadAllBytes(inputPath);
    var dest = File.ReadAllBytes(testPath);

    if (source.Length != dest.Length)
```

```
        throw new Exception("Length does not match");

    if (source.Where((t, i) => t != dest[i]).Any())
        throw new Exception("Data mismatch");
}
```

Lea Criptografía (System.Security.Cryptography) en línea:
https://riptutorial.com/es/csharp/topic/2988/criptografia--system-security-cryptography-

# Capítulo 47: Cronómetros

## Sintaxis

- stopWatch.Start () - inicia el cronómetro.
- stopWatch.Stop () - Detiene el cronómetro.
- stopWatch.Elapsed: obtiene el tiempo total transcurrido medido por el intervalo actual.

## Observaciones

Los cronómetros a menudo se usan en programas de evaluación comparativa para calcular el tiempo y ver cómo se ejecutan los diferentes segmentos de código óptimos.

## Examples

### Creando una instancia de un cronómetro

Una instancia de Cronómetro puede medir el tiempo transcurrido en varios intervalos, y el tiempo total transcurrido es la suma de todos los intervalos individuales. Esto proporciona un método confiable para medir el tiempo transcurrido entre dos o más eventos.

```
Stopwatch stopWatch = new Stopwatch();
stopWatch.Start();

double d = 0;
for (int i = 0; i < 1000 * 1000 * 1000; i++)
{
    d += 1;
}

stopWatch.Stop();
Console.WriteLine("Time elapsed: {0:hh\\:mm\\:ss\\.fffffff}", stopWatch.Elapsed);
```

`Stopwach` está en `System.Diagnostics` por lo que necesita agregar `using System.Diagnostics;` a su archivo.

### IsHighResolution

- La propiedad IsHighResolution indica si el temporizador se basa en un contador de rendimiento de alta resolución o en la clase DateTime.
- Este campo es de solo lectura.

```
// Display the timer frequency and resolution.
if (Stopwatch.IsHighResolution)
{
    Console.WriteLine("Operations timed using the system's high-resolution performance
counter.");
```

```
}
else
{
    Console.WriteLine("Operations timed using the DateTime class.");
}

long frequency = Stopwatch.Frequency;
Console.WriteLine("  Timer frequency in ticks per second = {0}",
    frequency);
long nanosecPerTick = (1000L*1000L*1000L) / frequency;
Console.WriteLine("  Timer is accurate within {0} nanoseconds",
    nanosecPerTick);
}
```

https://dotnetfiddle.net/ckrWUo

El temporizador utilizado por la clase de cronómetro depende del hardware del sistema y del sistema operativo. IsHighResolution se cumple si el temporizador del cronómetro se basa en un contador de rendimiento de alta resolución. De lo contrario, IsHighResolution es falso, lo que indica que el temporizador de cronómetro se basa en el temporizador del sistema.

Las marcas en el cronómetro dependen de la máquina / sistema operativo, por lo tanto, nunca debe contar con la ración de tictac de cronómetro a segundos para que sean iguales entre dos sistemas, y posiblemente incluso en el mismo sistema después de un reinicio. Por lo tanto, nunca se puede contar con que los tics de cronómetro tengan el mismo intervalo que los tics DateTime / TimeSpan.

Para obtener un tiempo independiente del sistema, asegúrese de usar las propiedades de tiempo transcurrido o de milisegundos transcurridos del cronómetro, que ya tienen en cuenta el cronómetro. Frecuencia (tics por segundo).

El cronómetro siempre se debe usar en Datetime para los procesos de sincronización, ya que es más liviano y usa Dateime si no puede usar un contador de rendimiento de alta resolución.

Fuente

Lea Cronómetros en línea: https://riptutorial.com/es/csharp/topic/3676/cronometros

# Capítulo 48: Cuerdas verbatim

## Sintaxis

- @ "las cadenas verbales son cadenas cuyo contenido no se escapa, por lo que en este caso \ n no representa el carácter de nueva línea sino dos caracteres individuales: \ y n. Las cadenas Verbatim se crean prefijando los contenidos de la cadena con el carácter @"
- @ "Para escapar de las comillas", se utilizan "comillas dobles".

## Observaciones

Para concatenar literales de cadena, use el símbolo @ al principio de cada cadena.

```
var combinedString = @"\t means a tab" + @" and \n means a newline";
```

## Examples

### Cuerdas multilínea

```
var multiLine = @"This is a

multiline paragraph";
```

**Salida:**

Esto es un

párrafo multilínea

Demo en vivo en .NET Fiddle

Las cadenas de varias líneas que contienen comillas dobles también pueden escaparse tal como estaban en una sola línea, porque son cadenas literales.

```
var multilineWithDoubleQuotes = @"I went to a city named

                        ""San Diego""

                    during summer vacation.";
```

Demo en vivo en .NET Fiddle

*Cabe señalar que los espacios / tabulaciones al comienzo de las líneas 2 y 3 aquí están realmente presentes en el valor de la variable; Compruebe esta pregunta para posibles soluciones.*

## Escapando cotizaciones dobles

Las comillas dobles dentro de las cadenas textuales pueden escaparse utilizando 2 comillas dobles secuenciales `""` para representar una comilla doble `"` en la cadena resultante.

```
var str = @"""I don't think so,"" he said.";
Console.WriteLine(str);
```

**Salida:**

> "No lo creo", dijo.

Demo en vivo en .NET Fiddle

## Cuerdas verbales interpoladas

Las cadenas verbales se pueden combinar con las nuevas funciones de interpolación de cadenas que se encuentran en C # 6.

```
Console.WriteLine($@"Testing \n 1 2 {5 - 2}
New line");
```

**Salida:**

> Pruebas \ n 1 2 3
> Nueva línea

Demo en vivo en .NET Fiddle

Como se esperaba de una cadena textual, las barras invertidas se ignoran como caracteres de escape. Y como se esperaba de una cadena interpolada, cualquier expresión dentro de llaves se evalúa antes de insertarse en la cadena en esa posición.

## Las cadenas Verbatim ordenan al compilador que no use escapes de caracteres

En una cadena normal, el carácter de barra diagonal inversa es el carácter de escape, que le indica al compilador que mire los caracteres siguientes para determinar el carácter real de la cadena. ( Lista completa de escapes de personajes )

En las cadenas textuales, no hay escapes de caracteres (excepto `""` que se convierte en un `"` ). Para usar una cadena textual, simplemente anteponga una `@` antes de las comillas iniciales.

Esta cadena textual

```
var filename = @"c:\temp\newfile.txt"
```

**Salida:**

c: \ temp \ newfile.txt

A diferencia de usar una cadena ordinaria (no verbal):

```
var filename = "c:\temp\newfile.txt"
```

que dará salida:

```
c:    emp
ewfile.txt
```

utilizando el carácter de escapar. (La `\t` se reemplaza con un carácter de tabulación y la `\n` se reemplaza con una nueva línea).

Demo en vivo en .NET Fiddle

Lea Cuerdas verbatim en línea: https://riptutorial.com/es/csharp/topic/16/cuerdas-verbatim

# Capítulo 49: Declaración de bloqueo

## Sintaxis

- bloqueo (obj) {}

## Observaciones

Usando la instrucción de `lock` , puede controlar el acceso de diferentes hilos al código dentro del bloque de código. Se usa comúnmente para prevenir condiciones de carrera, por ejemplo, varios subprocesos que leen y eliminan elementos de una colección. Como el bloqueo hace que los subprocesos esperen a que otros suban para salir de un bloque de código, puede causar retrasos que podrían resolverse con otros métodos de sincronización.

MSDN

> La palabra clave de bloqueo marca un bloque de declaración como una sección crítica al obtener el bloqueo de exclusión mutua para un objeto determinado, ejecutar una declaración y luego liberar el bloqueo.
>
> La palabra clave de bloqueo garantiza que un subproceso no ingrese a una sección crítica del código mientras que otro subproceso está en la sección crítica. Si otro hilo intenta ingresar un código bloqueado, esperará, bloqueará, hasta que se libere el objeto.
>
> La mejor práctica es definir un objeto **privado** para bloquear, o una variable de objeto **estática privada** para proteger los datos comunes a todas las instancias.

---

En C # 5.0 y versiones posteriores, la instrucción de `lock` es equivalente a:

```
bool lockTaken = false;
try
{
    System.Threading.Monitor.Enter(refObject, ref lockTaken);
    // code
}
finally
{
    if (lockTaken)
        System.Threading.Monitor.Exit(refObject);
}
```

Para C # 4.0 y anteriores, la declaración de `lock` es equivalente a:

```
System.Threading.Monitor.Enter(refObject);
try
{
    // code
```

```
}
finally
{
     System.Threading.Monitor.Exit(refObject);
}
```

# Examples

## Uso simple

El uso común de la `lock` es una sección crítica.

En el siguiente ejemplo, se supone que `ReserveRoom` debe llamarse desde diferentes subprocesos. La sincronización con `lock` es la forma más sencilla de evitar las condiciones de carrera aquí. El cuerpo del método está rodeado de un `lock` que garantiza que dos o más subprocesos no puedan ejecutarlo simultáneamente.

```
public class Hotel
{
    private readonly object _roomLock = new object();

    public void ReserveRoom(int roomNumber)
    {
        // lock keyword ensures that only one thread executes critical section at once
        // in this case, reserves a hotel room of given number
        // preventing double bookings
        lock (_roomLock)
        {
            // reserve room logic goes here
        }
    }
}
```

Si un hilo llega a `lock` bloque bloqueado mientras otro se ejecuta dentro de él, el primero esperará a otro para salir del bloque.

> La mejor práctica es definir un objeto privado para bloquear, o una variable de objeto estática privada para proteger los datos comunes a todas las instancias.

## Lanzar excepción en una sentencia de bloqueo

El siguiente código liberará el bloqueo. No habrá problema. Detrás de la escena de bloqueo de escenas funciona como `try finally`

```
lock(locker)
{
    throw new Exception();
}
```

Se puede ver más en la especificación de C # 5.0 :

Una declaración de `lock` de la forma

```
lock (x) ...
```

donde `x` es una expresión de un *tipo de referencia* , es exactamente equivalente a

```
bool __lockWasTaken = false;
try {
    System.Threading.Monitor.Enter(x, ref __lockWasTaken);
    ...
}
finally {
    if (__lockWasTaken) System.Threading.Monitor.Exit(x);
}
```

excepto que `x` sólo se evalúa una vez.

## Volver en una declaración de bloqueo

El siguiente código liberará el bloqueo.

```
lock(locker)
{
    return 5;
}
```

Para una explicación detallada, se recomienda esta respuesta SO .

## Usando instancias de Object para bloqueo

Cuando se utiliza la instrucción de `lock` incorporada de C #, se necesita una instancia de algún tipo, pero su estado no importa. Una instancia de `object` es perfecta para esto:

```
public class ThreadSafe {
  private static readonly object locker = new object();


  public void SomeThreadSafeMethod() {
    lock (locker) {
      // Only one thread can be here at a time.
    }
  }
}
```

**NB** . las instancias de `Type` no deben usarse para esto (en el código anterior a `typeof(ThreadSafe)` ) porque las instancias de `Type` se comparten entre los `typeof(ThreadSafe)` y, por lo tanto, la extensión del bloqueo puede incluir código que no debería (por ejemplo, si `ThreadSafe` está cargado en dos AppDomains en el mismo proceso y luego el bloqueo en su instancia de `Type` se bloquearían mutuamente).

## Anti-patrones y gotchas

# Bloqueo en una variable local / asignada a la pila

Una de las falacias al usar el `lock` es el uso de objetos locales como casillero en una función. Dado que estas instancias de objetos locales diferirán en cada llamada de la función, el `lock` no funcionará como se esperaba.

```
List<string> stringList = new List<string>();

public void AddToListNotThreadSafe(string something)
{
    // DO NOT do this, as each call to this method
    // will lock on a different instance of an Object.
    // This provides no thread safety, it only degrades performance.
    var localLock = new Object();
    lock(localLock)
    {
        stringList.Add(something);
    }
}

// Define object that can be used for thread safety in the AddToList method
readonly object classLock = new object();

public void AddToList(List<string> stringList, string something)
{
    // USE THE classLock instance field to achieve a
    // thread-safe lock before adding to stringList
    lock(classLock)
    {
        stringList.Add(something);
    }
}
```

# Suponiendo que el bloqueo restringe el acceso al objeto de sincronización en sí

Si un subproceso llama: el `lock(obj)` y otro subproceso llama a `obj.ToString()` segundo subproceso no se va a bloquear.

```
object obj = new Object();

public void SomeMethod()
{
     lock(obj)
    {
       //do dangerous stuff
    }
 }
```

```
 //Meanwhile on other tread
 public void SomeOtherMethod()
 {
   var objInString = obj.ToString(); //this does not block
 }
```

# Esperando que las subclases sepan cuándo bloquear

A veces, las clases base están diseñadas de tal manera que sus subclases deben usar un bloqueo al acceder a ciertos campos protegidos:

```
public abstract class Base
{
    protected readonly object padlock;
    protected readonly List<string> list;

    public Base()
    {
        this.padlock = new object();
        this.list = new List<string>();
    }

    public abstract void Do();
}

public class Derived1 : Base
{
    public override void Do()
    {
        lock (this.padlock)
        {
            this.list.Add("Derived1");
        }
    }
}

public class Derived2 : Base
{
    public override void Do()
    {
        this.list.Add("Derived2"); // OOPS! I forgot to lock!
    }
}
```

Es mucho más seguro *encapsular el bloqueo* mediante el uso de un método de plantilla :

```
public abstract class Base
{
    private readonly object padlock; // This is now private
    protected readonly List<string> list;

    public Base()
    {
```

```
        this.padlock = new object();
        this.list = new List<string>();
    }

    public void Do()
    {
        lock (this.padlock) {
            this.DoInternal();
        }
    }

    protected abstract void DoInternal();
}

public class Derived1 : Base
{
    protected override void DoInternal()
    {
        this.list.Add("Derived1"); // Yay! No need to lock
    }
}
```

# El bloqueo en una variable ValueType en caja no se sincroniza

En el siguiente ejemplo, una variable privada está encuadrada implícitamente ya que se suministra como un argumento de `object` a una función, esperando que un recurso de monitor se bloquee. El boxeo ocurre justo antes de llamar a la función IncInSync, por lo que la instancia en caja corresponde a un objeto de pila diferente cada vez que se llama a la función.

```
public int Count { get; private set; }

private readonly int counterLock = 1;

public void Inc()
{
    IncInSync(counterLock);
}

private void IncInSync(object monitorResource)
{
    lock (monitorResource)
    {
        Count++;
    }
}
```

El boxeo se produce en la función `Inc` :

```
BulemicCounter.Inc:
IL_0000:  nop
IL_0001:  ldarg.0
IL_0002:  ldarg.0
IL_0003:  ldfld       UserQuery+BulemicCounter.counterLock
```

```
IL_0008:  box         System.Int32**
IL_000D:  call        UserQuery+BulemicCounter.IncInSync
IL_0012:  nop
IL_0013:  ret
```

No significa que no se pueda utilizar un ValueType en caja para el bloqueo del monitor:

```
private readonly object counterLock = 1;
```

Ahora el boxeo ocurre en el constructor, lo cual está bien para bloquear:

```
IL_0001:  ldc.i4.1
IL_0002:  box         System.Int32
IL_0007:  stfld       UserQuery+BulemicCounter.counterLock
```

# Usar cerraduras innecesariamente cuando existe una alternativa más segura

Un patrón muy común es usar una `List` o `Dictionary` privado en una clase segura para subprocesos y bloquear cada vez que se accede a él:

```
public class Cache
{
    private readonly object padlock;
    private readonly Dictionary<string, object> values;

    public WordStats()
    {
        this.padlock = new object();
        this.values = new Dictionary<string, object>();
    }

    public void Add(string key, object value)
    {
        lock (this.padlock)
        {
            this.values.Add(key, value);
        }
    }

    /* rest of class omitted */
}
```

Si hay varios métodos para acceder al diccionario de `values` , el código puede ser muy largo y, lo que es más importante, bloquear todo el tiempo oculta su *intención* . El bloqueo también es muy fácil de olvidar y la falta de un bloqueo adecuado puede causar errores muy difíciles de encontrar.

Mediante el uso de un `ConcurrentDictionary` , podemos evitar el bloqueo por completo:

```
public class Cache
{
```

```
    private readonly ConcurrentDictionary<string, object> values;

    public WordStats()
    {
        this.values = new ConcurrentDictionary<string, object>();
    }

    public void Add(string key, object value)
    {
        this.values.Add(key, value);
    }

    /* rest of class omitted */
}
```

El uso de colecciones simultáneas también mejora el rendimiento, ya que todas emplean técnicas de bloqueo en cierta medida.

Lea Declaración de bloqueo en línea: https://riptutorial.com/es/csharp/topic/1495/declaracion-de-bloqueo

# Capítulo 50: Declaraciones condicionales

## Examples

### Declaración If-Else

La programación en general a menudo requiere una `decision` o una `branch` dentro del código para tener en cuenta cómo funciona el código bajo diferentes entradas o condiciones. En el lenguaje de programación C # (y en la mayoría de los lenguajes de programación para esta materia), la forma más sencilla y, a veces, la más útil de crear una rama dentro de su programa es a través de una declaración `If-Else` .

Asumamos que tenemos un método (también conocido como una función) que toma un parámetro int que representará un puntaje de hasta 100, y el método imprimirá un mensaje que indica si aprobamos o no.

```
static void PrintPassOrFail(int score)
{
    if (score >= 50) // If score is greater or equal to 50
    {
        Console.WriteLine("Pass!");
    }
    else // If score is not greater or equal to 50
    {
        Console.WriteLine("Fail!");
    }
}
```

Al observar este método, puede observar esta línea de código ( `score >= 50` ) dentro de la instrucción `If` . Esto se puede ver como una condición `boolean` , donde si la condición se evalúa para que sea igual a `true` , entonces se ejecuta el código que está entre `if` se ejecuta `{ }` .

Por ejemplo, si este método fue llamado así: `PrintPassOrFail(60);` , la salida del método sería una Consola de Impresión que dice ***¡Pase!*** ya que el valor del parámetro de 60 es mayor o igual a 50.

Sin embargo, si el método fue llamado como: `PrintPassOrFail(30);` , la salida del método se imprimirá diciendo ***Fail!*** . Esto se debe a que el valor 30 no es mayor o igual a 50, por lo tanto, el código que se encuentra entre `else { }` se ejecuta en lugar de la instrucción `If` .

En este ejemplo, hemos dicho que la *puntuación* debería aumentar a 100, lo que no se ha contabilizado en absoluto. Para tener en cuenta que el *puntaje* no supera los 100 *puntos* o posiblemente esté por debajo de 0, consulte el ejemplo de la **instrucción If-Else If-Else** .

### Declaración If-Else If-Else

Siguiendo con el ejemplo de la **Declaración If-Else** , ahora es el momento de introducir la instrucción `Else If` . La instrucción `Else If` sigue directamente después de la instrucción `If` en la estructura **If-Else If-Else** , pero intrínsecamente tiene una sintaxis similar a la instrucción `If` . Se

utiliza para agregar más ramas al código de lo que puede hacer una simple instrucción **If-Else** .

En el ejemplo de **If-Else Statement** , el ejemplo especificó que el puntaje sube a 100; Sin embargo, nunca hubo controles contra esto. Para solucionar esto, modifiquemos el método de **If-Else Statement** para que se vea así:

```
static void PrintPassOrFail(int score)
{
    if (score > 100) // If score is greater than 100
    {
        Console.WriteLine("Error: score is greater than 100!");
    }
    else if (score < 0) // Else If score is less than 0
    {
        Console.WriteLine("Error: score is less than 0!");
    }
    else if (score >= 50) // Else if score is greater or equal to 50
    {
        Console.WriteLine("Pass!");
    }
    else // If none above, then score must be between 0 and 49
    {
        Console.WriteLine("Fail!");
    }
}
```

Todas estas declaraciones se ejecutarán en orden desde la parte superior hasta la parte inferior hasta que se cumpla una condición. En esta nueva actualización del método, hemos agregado dos nuevas sucursales para acomodar ahora la puntuación que está *saliendo de los límites* .

Por ejemplo, si ahora llamamos al método en nuestro código como `PrintPassOFail(110);` , la salida sería un mensaje de ***error de*** Impresión de la Consola ***: ¡la puntuación es mayor que 100!*** ; y si llamamos al método en nuestro código como `PrintPassOrFail(-20);` , la salida diría ***Error: la puntuación es menor que 0!*** .

## Cambiar declaraciones

Una declaración de cambio permite que una variable se pruebe para determinar su igualdad frente a una lista de valores. Cada valor se llama un caso, y la variable que se está activando se comprueba para cada caso de interruptor.

Una declaración de `switch` es a menudo más concisa y comprensible que `if...else if... else..` declaraciones cuando se prueban varios valores posibles para una sola variable.

La sintaxis es la siguiente

```
switch(expression) {
   case constant-expression:
      statement(s);
      break;
   case constant-expression:
      statement(s);
      break;
```

```
   // you can have any number of case statements
   default : // Optional
      statement(s);
      break;
}
```

Hay varias cosas que se deben tener en cuenta al usar la instrucción switch

- La expresión utilizada en una instrucción de conmutación debe tener un tipo integral o enumerado, o debe ser de un tipo de clase en el que la clase tenga una única función de conversión a un tipo integral o enumerado.
- Puede tener cualquier número de declaraciones de casos dentro de un interruptor. Cada caso va seguido del valor que se va a comparar y dos puntos. Los valores para comparar deben ser únicos dentro de cada instrucción de conmutación.
- Una instrucción de cambio puede tener un caso predeterminado opcional. El caso predeterminado se puede usar para realizar una tarea cuando ninguno de los casos es verdadero.
- Cada caso debe terminar con una declaración de `break` , a menos que sea una declaración vacía. En ese caso la ejecución continuará en el caso que se encuentra debajo. La declaración de ruptura también se puede omitir cuando se utiliza una `return` , `throw` o `goto case` .

Se puede dar ejemplo con los grados sabios.

```
char grade = 'B';

switch (grade)
{
    case 'A':
        Console.WriteLine("Excellent!");
        break;
    case 'B':
    case 'C':
        Console.WriteLine("Well done");
        break;
    case 'D':
        Console.WriteLine("You passed");
        break;
    case 'F':
        Console.WriteLine("Better try again");
        break;
    default:
        Console.WriteLine("Invalid grade");
        break;
}
```

## Si las condiciones de la declaración son expresiones y valores booleanos estándar

La siguiente declaración

```
if (conditionA && conditionB && conditionC) //...
```

es exactamente equivalente a

```
bool conditions = conditionA && conditionB && conditionC;
if (conditions) // ...
```

en otras palabras, las condiciones dentro de la declaración "if" solo forman una expresión booleana ordinaria.

Un error común al escribir sentencias condicionales es comparar explícitamente con `true` y `false` :

```
if (conditionA == true && conditionB == false && conditionC == true) // ...
```

Esto se puede reescribir como

```
if (conditionA && !conditionB && conditionC)
```

Lea Declaraciones condicionales en línea:
https://riptutorial.com/es/csharp/topic/3144/declaraciones-condicionales

# Capítulo 51: Delegados

## Observaciones

# Resumen

Un **tipo de delegado** es un tipo que representa una firma de método particular. Una instancia de este tipo se refiere a un método particular con una firma coincidente. Los parámetros de los métodos pueden tener tipos de delegado, por lo que este único método puede pasar una referencia a otro método, que luego puede invocarse

# Tipos de delegados incorporados: `Action<...>` , `Predicate<T>` y `Func<...,TResult>`

El espacio de nombres del `System` contiene delegados de `Action<...>` , `Predicate<T>` y `Func<...,TResult>` , donde "..." representa entre 0 y 16 parámetros de tipo genérico (para 0 parámetros, `Action` no. genérico).

`Func` representa métodos con un tipo de retorno que coincide con `TResult` , y `Action` representa métodos sin un valor de retorno (vacío). En ambos casos, los parámetros de tipo genérico adicionales coinciden, en orden, con los parámetros del método.

`Predicate` representa el método con el tipo de retorno booleano, T es el parámetro de entrada.

# Tipos de delegados personalizados

Los tipos de delegado con nombre se pueden declarar utilizando la palabra clave `delegate` .

# Invocando delegados

Los delegados pueden invocarse utilizando la misma sintaxis que los métodos: el nombre de la instancia del delegado, seguido de paréntesis que contienen cualquier parámetro.

# Asignación a delegados

Los delegados pueden ser asignados de las siguientes maneras:

- Asignando un método nombrado
- Asignar un método anónimo utilizando un lambda

- Asignación de un método nombrado utilizando la palabra clave `delegate` .

# Combinando delegados

Se pueden asignar varios objetos delegados a una instancia de delegado utilizando el operador `+` . El operador `-` se puede usar para eliminar un componente delegado de otro delegado.

## Examples

### Referencias subyacentes de los delegados de métodos nombrados

Al asignar métodos nombrados a los delegados, se referirán al mismo objeto subyacente si:

- Son el mismo método de instancia, en la misma instancia de una clase
- Son el mismo método estático en una clase.

```
public class Greeter
{
    public void WriteInstance()
    {
        Console.WriteLine("Instance");
    }

    public static void WriteStatic()
    {
        Console.WriteLine("Static");
    }
}

// ...

Greeter greeter1 = new Greeter();
Greeter greeter2 = new Greeter();

Action instance1 = greeter1.WriteInstance;
Action instance2 = greeter2.WriteInstance;
Action instance1Again = greeter1.WriteInstance;

Console.WriteLine(instance1.Equals(instance2)); // False
Console.WriteLine(instance1.Equals(instance1Again)); // True

Action @static = Greeter.WriteStatic;
Action staticAgain = Greeter.WriteStatic;

Console.WriteLine(@static.Equals(staticAgain)); // True
```

### Declarar un tipo de delegado

La siguiente sintaxis crea un tipo de `delegate` con el nombre `NumberInOutDelegate` , que representa un método que toma un `int` y devuelve un `int` .

```
public delegate int NumberInOutDelegate(int input);
```

Esto se puede utilizar de la siguiente manera:

```
public static class Program
{
    static void Main()
    {
        NumberInOutDelegate square = MathDelegates.Square;
        int answer1 = square(4);
        Console.WriteLine(answer1); // Will output 16

        NumberInOutDelegate cube = MathDelegates.Cube;
        int answer2 = cube(4);
        Console.WriteLine(answer2); // Will output 64
    }
}

public static class MathDelegates
{
    static int Square (int x)
    {
        return x*x;
    }

    static int Cube (int x)
    {
        return x*x*x;
    }
}
```

El `example` instancia delegado se ejecuta de la misma manera como el `Square` método. Una instancia de delegado actúa literalmente como un delegado para la persona que llama: la persona que llama invoca al delegado, y luego el delegado llama al método de destino. Esta indirección desacopla al llamante del método de destino.

---

Puede declarar un tipo de delegado **genérico** y, en ese caso, puede especificar que el tipo sea covariante ( `out` ) o contravariante ( `in` ) en algunos de los argumentos de tipo. Por ejemplo:

```
public delegate TTo Converter<in TFrom, out TTo>(TFrom input);
```

Al igual que otros tipos genéricos, los tipos de delegados genéricos pueden tener restricciones, como `where TFrom : struct, IConvertible where TTo : new()` .

Evite la covarianza y la contravarianza para los tipos de delegados que están destinados a ser utilizados para delegados de multidifusión, como los tipos de controlador de eventos. Esto se debe a que la concatenación ( `+` ) puede fallar si el tipo de tiempo de ejecución es diferente del tipo de tiempo de compilación debido a la varianza. Por ejemplo, evita:

```
public delegate void EventHandler<in TEventArgs>(object sender, TEventArgs e);
```

En su lugar, utilice un tipo genérico invariante:

```
public delegate void EventHandler<TEventArgs>(object sender, TEventArgs e);
```

---

También se admiten delegados donde algunos parámetros se modifican por `ref` o `out` , como en:

```
public delegate bool TryParser<T>(string input, out T result);
```

(uso de la muestra `TryParser<decimal> example = decimal.TryParse;` ), o delegados donde el último parámetro tiene el `params` modificador. Los tipos de delegado pueden tener parámetros opcionales (suministrar valores predeterminados). Los tipos de delegados pueden usar tipos de punteros como `int*` o `char*` en sus firmas o tipos de devolución (use palabras clave `unsafe` ). Un tipo de delegado y sus parámetros pueden llevar atributos personalizados.

## El funcional Acción y Predicado tipos de delegado

El espacio de nombres del sistema contiene tipos de delegado `Func<..., TResult>` con entre 0 y 15 parámetros genéricos, devolviendo el tipo `TResult` .

```
private void UseFunc(Func<string> func)
{
    string output = func(); // Func with a single generic type parameter returns that type
    Console.WriteLine(output);
}

private void UseFunc(Func<int, int, string> func)
{
    string output = func(4, 2); // Func with multiple generic type parameters takes all but
the first as parameters of that type
    Console.WriteLine(output);
}
```

El espacio de nombres del sistema también contiene los tipos de delegados de la `Action<...>` con un número diferente de parámetros genéricos (de 0 a 16). Es similar a `Func<T1, .., Tn>` , pero siempre devuelve `void` .

```
private void UseAction(Action action)
{
    action(); // The non-generic Action has no parameters
}

private void UseAction(Action<int, string> action)
{
    action(4, "two"); // The generic action is invoked with parameters matching its type
arguments
}
```

`Predicate<T>` también es una forma de `Func` pero siempre devolverá `bool` . Un predicado es una forma de especificar un criterio personalizado. Dependiendo del valor de la entrada y la lógica definida dentro del predicado, devolverá `true` o `false` . `Predicate<T>` por lo tanto, se comporta de la misma manera que `Func<T, bool>` y ambos pueden inicializarse y usarse de la misma manera.

```
Predicate<string> predicate = s => s.StartsWith("a");
```

```
Func<string, bool> func = s => s.StartsWith("a");

// Both of these return true
var predicateReturnsTrue = predicate("abc");
var funcReturnsTrue = func("abc");

// Both of these return false
var predicateReturnsFalse = predicate("xyz");
var funcReturnsFalse = func("xyz");
```

La elección de utilizar `Predicate<T>` o `Func<T, bool>` es realmente una cuestión de opinión. `Predicate<T>` es posiblemente más expresivo de la intención del autor, mientras que `Func<T, bool>` es probable que sea familiar para una mayor proporción de desarrolladores de C #.

Además de eso, hay algunos casos en los que solo una de las opciones está disponible, especialmente al interactuar con otra API. Por ejemplo, `List<T>` y `Array<T>` generalmente toman `Predicate<T>` para sus métodos, mientras que la mayoría de las extensiones LINQ solo aceptan `Func<T, bool>` .

## Asignar un método nombrado a un delegado

Los métodos nombrados se pueden asignar a delegados con firmas coincidentes:

```
public static class Example
{
    public static int AddOne(int input)
    {
        return input + 1;
    }
}


Func<int,int> addOne = Example.AddOne
```

`Example.AddOne` toma un `int` y devuelve un `int` , su firma coincide con el delegado `Func<int,int>` . `Example.AddOne` puede asignarse directamente a `addOne` porque tienen firmas coincidentes.

## Igualdad de delegados

Al llamar a `.Equals()` en un delegado se compara por igualdad de referencia:

```
Action action1 = () => Console.WriteLine("Hello delegates");
Action action2 = () => Console.WriteLine("Hello delegates");
Action action1Again = action1;

Console.WriteLine(action1.Equals(action1)) // True
Console.WriteLine(action1.Equals(action2)) // False
Console.WriteLine(action1Again.Equals(action1)) // True
```

Estas reglas también se aplican al hacer `+=` o `-=` en un delegado de multidifusión, por ejemplo, al suscribirse y cancelar la suscripción a eventos.

## Asignar a un delegado por lambda

Lambdas se puede usar para crear métodos anónimos para asignar a un delegado:

```
Func<int,int> addOne = x => x+1;
```

Tenga en cuenta que la declaración explícita de tipo se requiere al crear una variable de esta manera:

```
var addOne = x => x+1; // Does not work
```

## Pasando delegados como parámetros

Los delegados se pueden utilizar como punteros de función escritos:

```
class FuncAsParameters
{
  public void Run()
  {
    DoSomething(ErrorHandler1);
    DoSomething(ErrorHandler2);
  }

  public bool ErrorHandler1(string message)
  {
    Console.WriteLine(message);
    var shouldWeContinue = ...
    return shouldWeContinue;
  }

  public bool ErrorHandler2(string message)
  {
    // ...Write message to file...
    var shouldWeContinue = ...
    return shouldWeContinue;
  }

  public void DoSomething(Func<string, bool> errorHandler)
  {
    // In here, we don't care what handler we got passed!
    ...
    if (...error...)
    {
      if (!errorHandler("Some error occurred!"))
      {
        // The handler decided we can't continue
        return;
      }
    }
  }
}
```

## Combinar delegados (delegados de multidifusión)

Operaciones de suma `+` y resta `-` se pueden utilizar para combinar instancias delegadas. El delegado contiene una lista de los delegados asignados.

```
using System;
using System.Reflection;
using System.Reflection.Emit;

namespace DelegatesExample {
    class MainClass {
        private delegate void MyDelegate(int a);

        private static void PrintInt(int a) {
            Console.WriteLine(a);
        }

        private static void PrintType<T>(T a) {
            Console.WriteLine(a.GetType());
        }

        public static void Main (string[] args)
        {
            MyDelegate d1 = PrintInt;
            MyDelegate d2 = PrintType;

            // Output:
            // 1
            d1(1);

            // Output:
            // System.Int32
            d2(1);

            MyDelegate d3 = d1 + d2;
            // Output:
            // 1
            // System.Int32
            d3(1);

            MyDelegate d4 = d3 - d2;
            // Output:
            // 1
            d4(1);

            // Output:
            // True
            Console.WriteLine(d1 == d4);
        }
    }
}
```

En este ejemplo, `d3` es una combinación de delegados `d1` y `d2` , por lo tanto, cuando se llama, el programa produce las cadenas `1` y `System.Int32` .

---

Combinando delegados con tipos de retorno **no nulos** :

Si un delegado de multidifusión tiene un tipo de retorno no `nonvoid` , la persona que llama recibe el valor de retorno del último método a invocar. Los métodos anteriores todavía se llaman, pero sus

valores de retorno se descartan.

```
    class Program
    {
        public delegate int Transformer(int x);

        static void Main(string[] args)
        {
            Transformer t = Square;
            t += Cube;
            Console.WriteLine(t(2));  // O/P 8
        }

        static int Square(int x) { return x * x; }

        static int Cube(int x) { return x*x*x; }
    }
```

`t(2)` llamará primero a `Square` y luego a `Cube` . El valor de retorno de Cuadrado se descarta y el valor de retorno del último método, es decir, el `Cube` se retiene.

## Seguro invocar delegado de multidifusión

Alguna vez quiso llamar a un delegado de multidifusión, pero desea que se llame a toda la lista de llamadas, incluso si se produce una excepción en alguna de las cadenas. Entonces estás de suerte, he creado un método de extensión que hace precisamente eso, lanzando una `AggregateException` solo después de que se completa la ejecución de la lista completa:

```
public static class DelegateExtensions
{
    public static void SafeInvoke(this Delegate del,params object[] args)
    {
        var exceptions = new List<Exception>();

        foreach (var handler in del.GetInvocationList())
        {
            try
            {
                handler.Method.Invoke(handler.Target, args);
            }
            catch (Exception ex)
            {
                exceptions.Add(ex);
            }
        }

        if(exceptions.Any())
        {
            throw new AggregateException(exceptions);
        }
    }
}

public class Test
{
    public delegate void SampleDelegate();
```

```
    public void Run()
    {
        SampleDelegate delegateInstance = this.Target2;
        delegateInstance += this.Target1;

        try
        {
            delegateInstance.SafeInvoke();
        }
        catch(AggregateException ex)
        {
            // Do any exception handling here
        }
    }

    private void Target1()
    {
        Console.WriteLine("Target 1 executed");
    }

    private void Target2()
    {
        Console.WriteLine("Target 2 executed");
        throw new Exception();
    }
}
```

Esto produce:

```
Target 2 executed
Target 1 executed
```

Invocar directamente, sin `SaveInvoke` , solo ejecutaría Target 2.

## Cierre dentro de un delegado

Los cierres son métodos anónimos en línea que tienen la capacidad de usar variables de métodos `Parent` y otros métodos anónimos que se definen en el alcance de los padres.

> En esencia, un cierre es un bloque de código que puede ejecutarse en un momento posterior, pero que mantiene el entorno en el que se creó por primera vez, es decir, aún puede usar las variables locales, etc., del método que lo creó, incluso después de eso. El método ha terminado de ejecutarse. **- Jon Skeet**

```
delegate int testDel();
static void Main(string[] args)
{
    int foo = 4;
    testDel myClosure = delegate()
    {
        return foo;
    };
    int bar = myClosure();

}
```

Ejemplo tomado de Closures en .NET .

### Encapsulando transformaciones en funciones.

```
public class MyObject{
    public DateTime? TestDate { get; set; }

    public Func<MyObject, bool> DateIsValid = myObject => myObject.TestDate.HasValue &&
myObject.TestDate > DateTime.Now;

    public void DoSomething(){
        //We can do this:
        if(this.TestDate.HasValue && this.TestDate > DateTime.Now){
            CallAnotherMethod();
        }

        //or this:
        if(DateIsValid(this)){
            CallAnotherMethod();
        }
    }
}
```

En el espíritu de la codificación limpia, las comprobaciones y transformaciones encapsuladas como la de arriba como Func pueden hacer que su código sea más fácil de leer y entender. Si bien el ejemplo anterior es muy simple, ¿qué pasaría si hubiera varias propiedades de DateTime con sus propias reglas de validación diferentes y quisiéramos verificar diferentes combinaciones? Las funciones simples, de una línea, cada una de las cuales ha establecido una lógica de retorno, pueden ser legibles y reducir la complejidad aparente de su código. Considere las siguientes llamadas a funciones e imagínese cuánto más código estaría saturando el método:

```
public void CheckForIntegrity(){
    if(ShipDateIsValid(this) && TestResultsHaveBeenIssued(this) && !TestResultsFail(this)){
        SendPassingTestNotification();
    }
}
```

Lea Delegados en línea: https://riptutorial.com/es/csharp/topic/1194/delegados

# Capítulo 52: Delegados funcionales

## Sintaxis

- `public delegate TResult Func<in T, out TResult>(T arg)`
- `public delegate TResult Func<in T1, in T2, out TResult>(T1 arg1, T2 arg2)`
- `public delegate TResult Func<in T1, in T2, in T3, out TResult>(T1 arg1, T2 arg2, T3 arg3)`
- `public delegate TResult Func<in T1, in T2, in T3, in T4, out TResult>(T1 arg1, T2 arg2, T3 arg3, T4 arg4)`

## Parámetros

| Parámetro | Detalles |
|---|---|
| `arg` o `arg1` | el (primer) parámetro del método |
| `arg2` | El segundo parámetro del método. |
| `arg3` | El tercer parámetro del método. |
| `arg4` | El cuarto parámetro del método. |
| `T` o `T1` | El tipo del (primer) parámetro del método. |
| `T2` | El tipo del segundo parámetro del método. |
| `T3` | El tipo del tercer parámetro del método. |
| `T4` | El tipo del cuarto parámetro del método. |
| `TResult` | el tipo de retorno del método |

## Examples

### Sin parametros

Este ejemplo muestra cómo crear un delegado que encapsula el método que devuelve la hora actual

```
static DateTime UTCNow()
{
    return DateTime.UtcNow;
}

static DateTime LocalNow()
{
    return DateTime.Now;
}
```

```
static void Main(string[] args)
{
    Func<DateTime> method = UTCNow;
    // method points to the UTCNow method
    // that retuns current UTC time
    DateTime utcNow = method();

    method = LocalNow;
    // now method points to the LocalNow method
    // that returns local time

    DateTime localNow = method();
}
```

### Con multiples variables

```
static int Sum(int a, int b)
{
    return a + b;
}

static int Multiplication(int a, int b)
{
    return a * b;
}

static void Main(string[] args)
{
    Func<int, int, int> method = Sum;
    // method points to the Sum method
    // that retuns 1 int variable and takes 2 int variables
    int sum = method(1, 1);

    method = Multiplication;
    // now method points to the Multiplication method

    int multiplication = method(1, 1);
}
```

### Lambda y métodos anónimos

Se puede asignar un método anónimo donde se espera un delegado:

```
Func<int, int> square = delegate (int x) { return x * x; }
```

Las expresiones Lambda se pueden usar para expresar lo mismo:

```
Func<int, int> square = x => x * x;
```

En cualquier caso, ahora podemos invocar el método almacenado dentro de un `square` como este:

```
var sq = square.Invoke(2);
```

O como una taquigrafía:

```
var sq = square(2);
```

Observe que para que la asignación sea segura para el tipo, los tipos de parámetros y el tipo de retorno del método anónimo deben coincidir con los del tipo delegado:

```
Func<int, int> sum = delegate (int x, int y) { return x + y; } // error
Func<int, int> sum = (x, y) => x + y; // error
```

## Parámetros de Tipo Covariante y Contravariante

`Func` también soporta Covariant & Contravariant

```
// Simple hierarchy of classes.
public class Person { }
public class Employee : Person { }

class Program
{
    static Employee FindByTitle(String title)
    {
        // This is a stub for a method that returns
        // an employee that has the specified title.
        return new Employee();
    }

    static void Test()
    {
        // Create an instance of the delegate without using variance.
        Func<String, Employee> findEmployee = FindByTitle;

        // The delegate expects a method to return Person,
        // but you can assign it a method that returns Employee.
        Func<String, Person> findPerson = FindByTitle;

        // You can also assign a delegate
        // that returns a more derived type
        // to a delegate that returns a less derived type.
        findPerson = findEmployee;

    }
}
```

Lea Delegados funcionales en línea: https://riptutorial.com/es/csharp/topic/2769/delegados-funcionales

# Capítulo 53: Diagnósticos

## Examples

### Debug.WriteLine

Escribe a los escuchas de seguimiento en la colección de escuchas cuando la aplicación se compila en la configuración de depuración.

```
public static void Main(string[] args)
{
    Debug.WriteLine("Hello");
}
```

En Visual Studio o Xamarin Studio, esto aparecerá en la ventana de resultados de la aplicación. Esto se debe a la presencia del agente de escucha de seguimiento predeterminado en TraceListenerCollection.

### Redireccionando la salida del registro con TraceListeners

Puede redirigir la salida de depuración a un archivo de texto agregando un TextWriterTraceListener a la colección Debug.Listeners.

```
public static void Main(string[] args)
{
    TextWriterTraceListener myWriter = new TextWriterTraceListener(@"debug.txt");
    Debug.Listeners.Add(myWriter);
    Debug.WriteLine("Hello");

    myWriter.Flush();
}
```

Puede redirigir el resultado de la depuración a la salida de una aplicación de consola utilizando ConsoleTraceListener.

```
public static void Main(string[] args)
{
    ConsoleTraceListener myWriter = new ConsoleTraceListener();
    Debug.Listeners.Add(myWriter);
    Debug.WriteLine("Hello");
}
```

Lea Diagnósticos en línea: https://riptutorial.com/es/csharp/topic/2147/diagnosticos

# Capítulo 54: Directiva de uso

## Observaciones

La palabra clave `using` es tanto una directiva (este tema) como una declaración.

Para la declaración de `using` (es decir, para encapsular el alcance de un objeto `IDisposable` , asegurándose de que fuera de ese alcance el objeto se elimine de forma limpia), consulte Uso de la declaración .

## Examples

### Uso básico

```
using System;
using BasicStuff = System;
using Sayer = System.Console;
using static System.Console;  //From C# 6

class Program
{
    public static void Main()
    {
        System.Console.WriteLine("Ignoring usings and specifying full type name");
        Console.WriteLine("Thanks to the 'using System' directive");
        BasicStuff.Console.WriteLine("Namespace aliasing");
        Sayer.WriteLine("Type aliasing");
        WriteLine("Thanks to the 'using static' directive (from C# 6)");
    }
}
```

### Referencia a un espacio de nombres

```
using System.Text;
//allows you to access classes within this namespace such as StringBuilder
//without prefixing them with the namespace.  i.e:

//...
var sb = new StringBuilder();
//instead of
var sb = new System.Text.StringBuilder();
```

### Asociar un alias con un espacio de nombres

```
using st = System.Text;
//allows you to access classes within this namespace such as StringBuilder
//prefixing them with only the defined alias and not the full namespace.  i.e:

//...
var sb = new st.StringBuilder();
```

```
//instead of
var sb = new System.Text.StringBuilder();
```

## Acceder a los miembros estáticos de una clase

6.0

Le permite importar un tipo específico y usar los miembros estáticos del tipo sin calificarlos con el nombre del tipo. Esto muestra un ejemplo usando métodos estáticos:

```
using static System.Console;

// ...

string GetName()
{
    WriteLine("Enter your name.");
    return ReadLine();
}
```

Y esto muestra un ejemplo utilizando propiedades y métodos estáticos:

```
using static System.Math;

namespace Geometry
{
    public class Circle
    {
        public double Radius { get; set; };

        public double Area => PI * Pow(Radius, 2);
    }
}
```

## Asociar un alias para resolver conflictos

Si está utilizando varios espacios de nombres que pueden tener clases con el mismo nombre (como `System.Random` y `UnityEngine.Random` ), puede usar un alias para especificar que `Random` proviene de uno u otro sin tener que usar todo el espacio de nombres en la llamada .

Por ejemplo:

```
using UnityEngine;
using System;

Random rnd = new Random();
```

Esto hará que el compilador no esté seguro de qué `Random` evalúa la nueva variable como. En su lugar, puedes hacer:

```
using UnityEngine;
using System;
```

```
using Random = System.Random;

Random rnd = new Random();
```

Esto no le impide llamar al otro por su espacio de nombres totalmente calificado, como este:

```
using UnityEngine;
using System;
using Random = System.Random;

Random rnd = new Random();
int unityRandom = UnityEngine.Random.Range(0,100);
```

`rnd` será una variable `System.Random` y `unityRandom` será una variable `UnityEngine.Random` .

## Usando directivas de alias

Puede usar `using` para establecer un alias para un espacio de nombres o tipo. Más detalles se pueden encontrar aquí .

Sintaxis:

```
using <identifier> = <namespace-or-type-name>;
```

Ejemplo:

```
using NewType = Dictionary<string, Dictionary<string,int>>;
NewType multiDictionary = new NewType();
//Use instances as you are using the original one
multiDictionary.Add("test", new Dictionary<string,int>());
```

Lea Directiva de uso en línea: https://riptutorial.com/es/csharp/topic/52/directiva-de-uso

# Capítulo 55: Directivas del pre procesador

## Sintaxis

- #define *[symbol]* // Define un símbolo compilador.
- #undef *[symbol]* // Undefines un símbolo compilador.
- #warning *[mensaje de advertencia]* // Genera una advertencia del compilador. Útil con #if.
- #error *[mensaje de error]* // Genera un error del compilador. Útil con #if.
- #line *[número de línea] (nombre de archivo)* // Anula el número de línea del compilador (y opcionalmente el nombre del archivo de origen). Utilizado con plantillas de texto T4 .
- #pragma warning [disable | restore] *[números de advertencia]* // Desactiva / restaura las advertencias del compilador.
- #pragma checksum " *[filename]* " " *[guid]* " " *[checksum]* " // Valida el contenido de un archivo fuente.
- #region *[nombre de la región]* // Define una región de código plegable.
- #endregion // Finaliza un bloque de región de código.
- #if *[condición]* // Ejecuta el código siguiente si la condición es verdadera.
- #else // Se usa después de un #if.
- #elif *[condición]* // Se usa después de un #if.
- #endif // Finaliza un bloque condicional iniciado con #if.

## Observaciones

Las directivas de preprocesador se utilizan normalmente para hacer que los programas de origen sean fáciles de cambiar y compilar en diferentes entornos de ejecución. Las directivas en el archivo de origen le indican al preprocesador que realice acciones específicas. Por ejemplo, el preprocesador puede reemplazar tokens en el texto, insertar el contenido de otros archivos en el archivo de origen o suprimir la compilación de parte del archivo eliminando secciones de texto. Las líneas del preprocesador se reconocen y se ejecutan antes de la expansión de macros. Por lo tanto, si una macro se expande en algo que parece un comando de preprocesador, ese comando no es reconocido por el preprocesador.

Las instrucciones del preprocesador utilizan el mismo conjunto de caracteres que las declaraciones del archivo de origen, con la excepción de que las secuencias de escape no son compatibles. El conjunto de caracteres utilizado en las instrucciones del preprocesador es el mismo que el conjunto de caracteres de ejecución. El preprocesador también reconoce valores de caracteres negativos.

## Expresiones condicionales

Las expresiones condicionales ( `#if` , `#elif` , etc.) admiten un subconjunto limitado de operadores booleanos. Son:

- `==` y `!=` . Solo se pueden usar para comprobar si el símbolo es verdadero (definido) o falso

(no definido)

- `&&` , `||` , `!`
- `()`

Por ejemplo:

```
#if !DEBUG && (SOME_SYMBOL || SOME_OTHER_SYMBOL) && RELEASE == true
Console.WriteLine("OK!");
#endif
```

compilaría el código que imprime "OK!" a la consola si `DEBUG` no está definido, ya sea `SOME_SYMBOL` o `SOME_OTHER_SYMBOL` , y `RELEASE` está definido.

Nota: estas sustituciones se realizan *en tiempo de compilación* y, por lo tanto, no están disponibles para inspección en tiempo de ejecución. El código eliminado mediante el uso de `#if` no es parte de la salida del compilador.

Vea también: Directivas de preprocesador de C # en MSDN.

# Examples

## Expresiones condicionales

Cuando se compila lo siguiente, devolverá un valor diferente según las directivas definidas.

```
// Compile with /d:A or /d:B to see the difference
string SomeFunction()
{
#if A
    return "A";
#elif B
    return "B";
#else
    return "C";
#endif
}
```

Las expresiones condicionales se utilizan normalmente para registrar información adicional para las construcciones de depuración.

```
void SomeFunc()
{
    try
    {
        SomeRiskyMethod();
    }
    catch (ArgumentException ex)
    {
        #if DEBUG
        log.Error("SomeFunc", ex);
        #endif

        HandleException(ex);
```

```
    }
}
```

## Generando advertencias y errores del compilador

Las advertencias del compilador se pueden generar usando la directiva `#warning` , y los errores también se pueden generar usando la directiva `#error` .

```
#if SOME_SYMBOL
#error This is a compiler Error.
#elif SOME_OTHER_SYMBOL
#warning This is a compiler Warning.
#endif
```

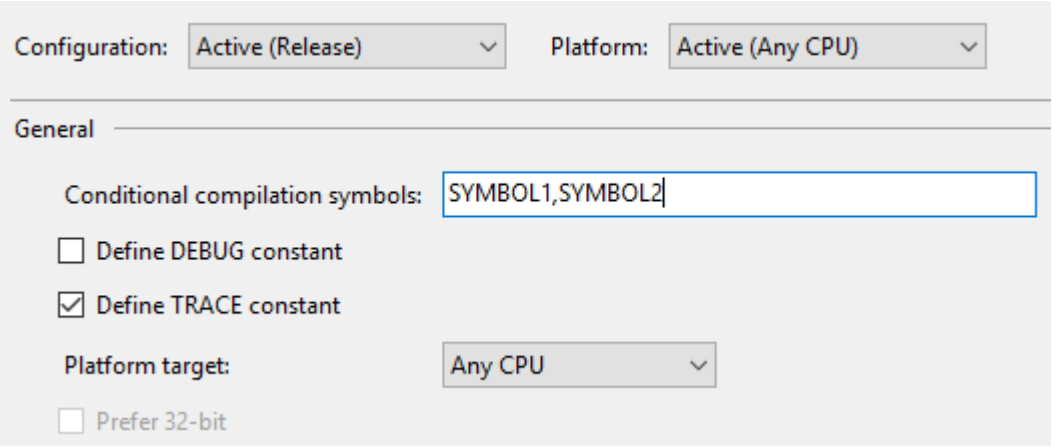
## Definir y no definir símbolos

Un símbolo de compilador es una palabra clave que se define en tiempo de compilación y se puede verificar para ejecutar condicionalmente secciones específicas de código.

Hay tres formas de definir un símbolo compilador. Se pueden definir vía código:

```
#define MYSYMBOL
```

Se pueden definir en Visual Studio, en Propiedades del proyecto> Generar> Símbolos de compilación condicional:




*(Tenga en cuenta que `DEBUG` y `TRACE` tienen sus propias casillas de verificación y no es necesario que se especifiquen explícitamente).*

O se pueden definir en tiempo de compilación utilizando el `csc.exe /define:[name]` en el compilador de C #, `csc.exe` .

También puede definir símbolos no definidos utilizando la directiva `#undefine` .

El ejemplo más frecuente de esto es el símbolo `DEBUG` , que Visual Studio define cuando una aplicación se compila en modo Debug (versus modo Release).

```
public void DoBusinessLogic()
{
    try
    {
        AuthenticateUser();
        LoadAccount();
        ProcessAccount();
        FinalizeTransaction();
    }
    catch (Exception ex)
    {
#if DEBUG
        System.Diagnostics.Trace.WriteLine("Unhandled exception!");
        System.Diagnostics.Trace.WriteLine(ex);
        throw;
#else
        LoggingFramework.LogError(ex);
        DisplayFriendlyErrorMessage();
#endif
    }
}
```

En el ejemplo anterior, cuando se produce un error en la lógica de negocios de la aplicación, si la aplicación se compila en modo de depuración (y se establece el símbolo `DEBUG` ), el error se escribirá en el registro de seguimiento y la excepción se volverá a realizar. -fruto de la depuración. Sin embargo, si la aplicación se compila en modo Release (y no se establece ningún símbolo `DEBUG` ), se utiliza un marco de registro para registrar el error de manera silenciosa, y se muestra un mensaje de error al usuario final.

### Bloques regionales

Utilice `#region` y `#endregion` para definir una región de código plegable.

```
#region Event Handlers

public void Button_Click(object s, EventArgs e)
{
    // ...
}

public void DropDown_SelectedIndexChanged(object s, EventArgs e)
{
    // ...
}

#endregion
```

Estas directivas solo son beneficiosas cuando se utiliza un IDE que admite regiones plegables (como Visual Studio ) para editar el código.

### Otras instrucciones del compilador

# Línea

`#line` controla el número de línea y el nombre de archivo reportados por el compilador al emitir advertencias y errores.

```
void Test()
{
    #line 42 "Answer"
    #line filename "SomeFile.cs"
    int life; // compiler warning CS0168 in "SomeFile.cs" at Line 42
    #line default
    // compiler warnings reset to default
}
```

# Pragma Checksum

`#pragma checksum` permite la especificación de una suma de comprobación específica para una base de datos de programa generada (PDB) para la depuración.

```
#pragma checksum "MyCode.cs" "{00000000-0000-0000-0000-000000000000}" "{0123456789A}"
```

## Usando el atributo condicional

Agregar un atributo `Conditional` desde el espacio de nombres de `System.Diagnostics` a un método es una forma limpia de controlar qué métodos se llaman en las compilaciones y cuáles no.

```
#define EXAMPLE_A

using System.Diagnostics;
class Program
{
    static void Main()
    {
        ExampleA(); // This method will be called
        ExampleB(); // This method will not be called
    }

    [Conditional("EXAMPLE_A")]
    static void ExampleA() {...}

    [Conditional("EXAMPLE_B")]
    static void ExampleB() {...}
}
```

## Desactivación y restauración de las advertencias del compilador

Puede deshabilitar las advertencias del compilador usando `#pragma warning disable` y restaurarlas usando `#pragma warning restore` :

```
#pragma warning disable CS0168

// Will not generate the "unused variable" compiler warning since it was disabled
var x = 5;
```

```
#pragma warning restore CS0168

// Will generate a compiler warning since the warning was just restored
var y = 8;
```

Se permiten números de advertencia separados por comas:

```
#pragma warning disable CS0168, CS0219
```

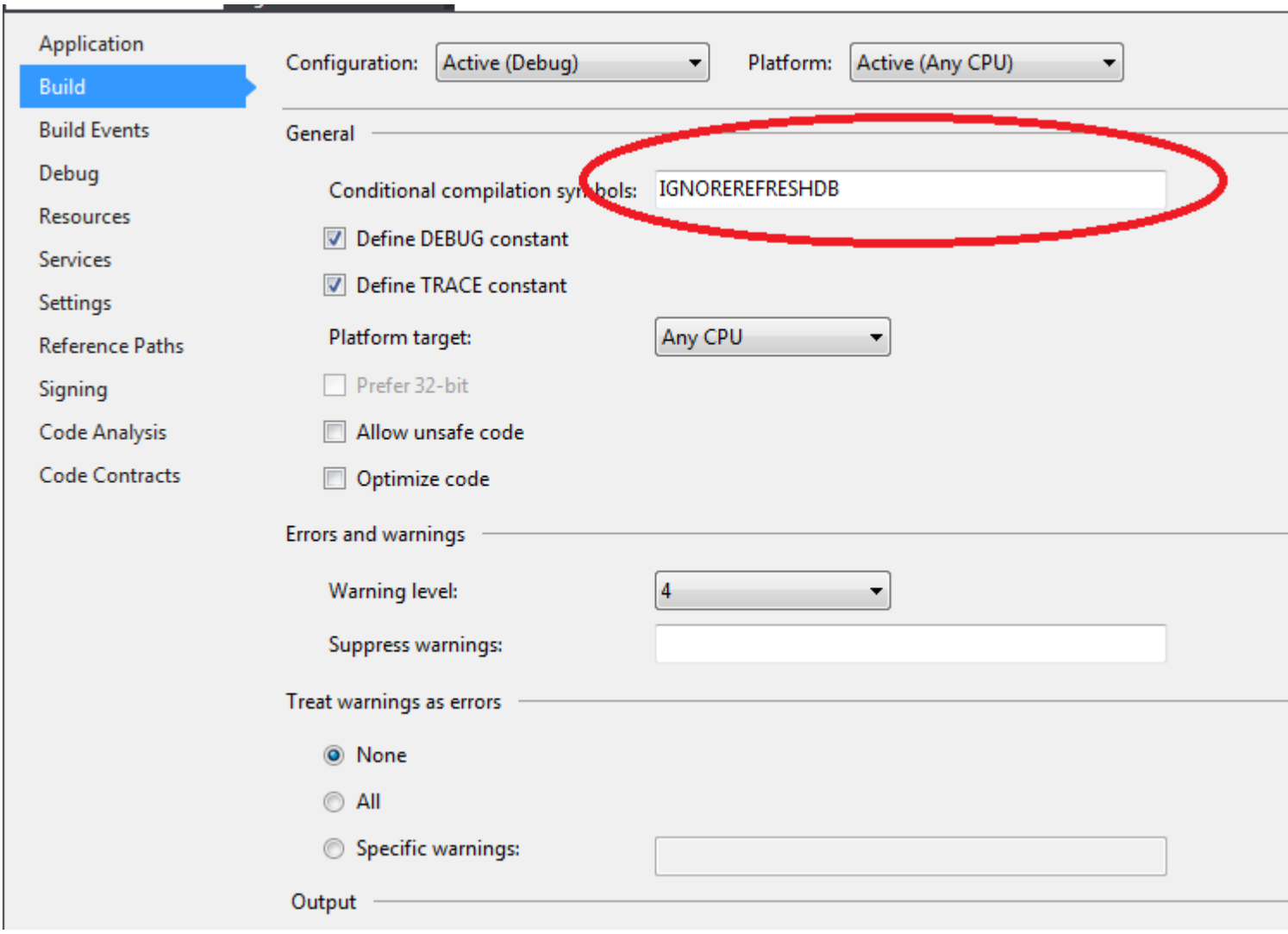
El prefijo `CS` es opcional, e incluso se puede mezclar (aunque no es una buena práctica):

```
#pragma warning disable 0168, 0219, CS0414
```

## Preprocesadores personalizados a nivel de proyecto

Es conveniente establecer un preprocesamiento condicional personalizado a nivel de proyecto cuando es necesario omitir algunas acciones, por ejemplo, para las pruebas.

Vaya al `Solution Explorer` -> Haga clic con el `botón derecho del mouse` en el proyecto en el que desea establecer la variable en -> `Properties` -> `Build` -> En el campo General de búsqueda `Conditional compilation symbols` e ingrese su variable condicional aquí




Ejemplo de código que saltará algún código:

```
public void Init()
{
    #if !IGNOREREFRESHDB
    // will skip code here
     db.Initialize();
    #endif
}
```

Lea Directivas del pre procesador en línea: https://riptutorial.com/es/csharp/topic/755/directivas-del-pre-procesador

# Capítulo 56: Documentación XML Comentarios

## Observaciones

Algunas veces necesitas **crear documentación de texto extendido a** partir de tus comentarios xml. Desafortunadamente ***no hay una forma estándar para ello*** .

Pero hay algunos proyectos separados que puedes usar para este caso:

- Castillo de arena
- Docu
- NDoc
- DocFX

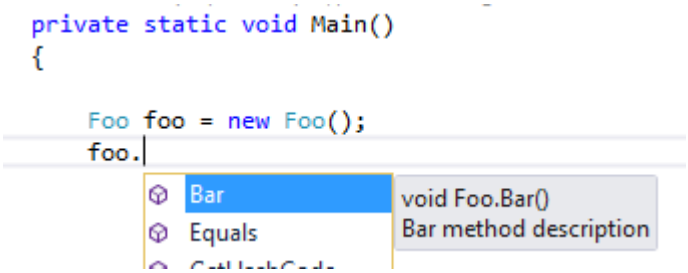
## Examples

### Anotación de método simple

Los comentarios de la documentación se colocan directamente sobre el método o la clase que describen. Comienzan con tres barras diagonales `///` , y permiten que la información meta se almacene a través de XML.

```
/// <summary>
/// Bar method description
/// </summary>
public void Bar()
{

}
```

Visual Studio y otras herramientas pueden utilizar la información dentro de las etiquetas para proporcionar servicios como IntelliSense:



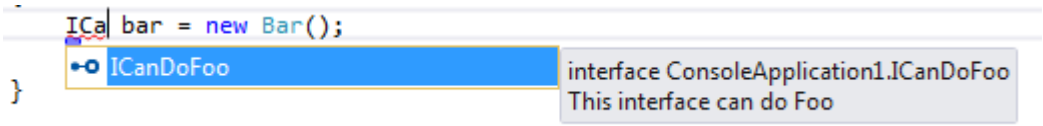

Vea también la lista de etiquetas comunes de documentación de Microsoft .

### Comentarios de interfaz y documentación de clase.

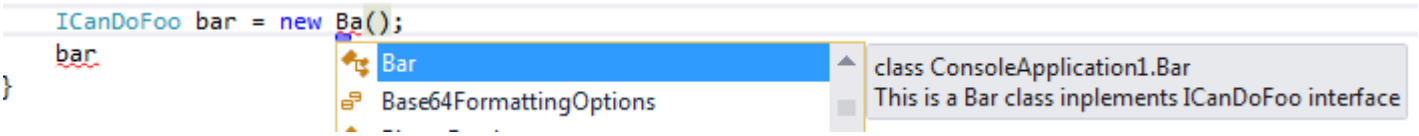
```
/// <summary>
/// This interface can do Foo
/// </summary>
public interface ICanDoFoo
{
    // ...
}

/// <summary>
/// This Bar class implements ICanDoFoo interface
/// </summary>
public class Bar : ICanDoFoo
{
    // ...
}
```

**Resultado**

Resumen de la interfaz



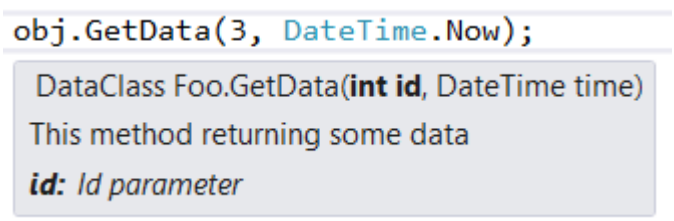

Resumen de la clase




## Documentación del método comentar con param y devuelve elementos.

```
/// <summary>
/// Returns the data for the specified ID and timestamp.
/// </summary>
/// <param name="id">The ID for which to get data. </param>
/// <param name="time">The DateTime for which to get data. </param>
/// <returns>A DataClass instance with the result. </returns>
public DataClass GetData(int id, DateTime time)
{
   // ...
}
```

**IntelliSense le** muestra la descripción de cada parámetro:

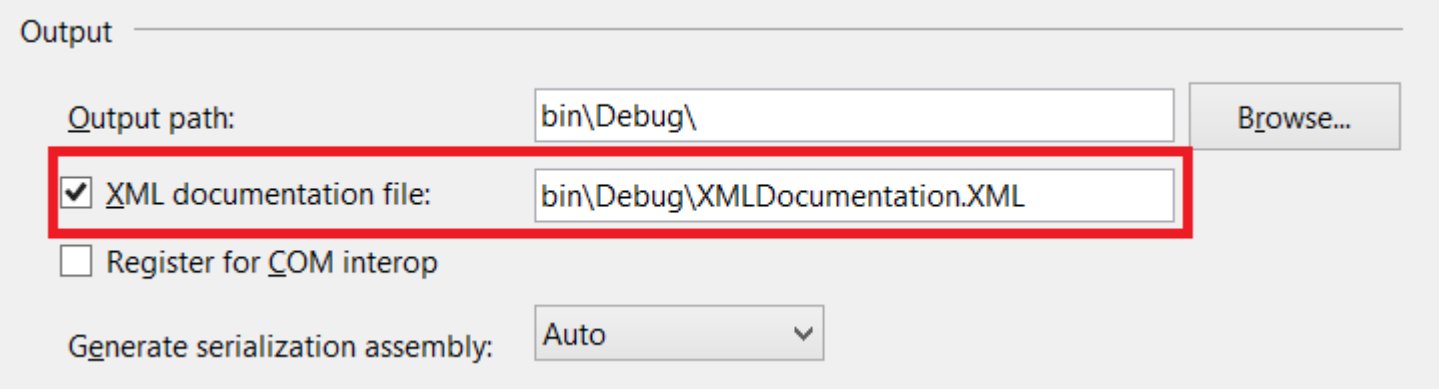



Consejo: si Intellisense no se muestra en Visual Studio, elimine el primer corchete o coma y luego vuelva a escribirlo.

## Generando XML a partir de comentarios de documentación.

Para generar un archivo de documentación XML a partir de los comentarios de la documentación en el código, use la opción `/doc` con el compilador `csc.exe` C #.

En Visual Studio 2013/2015, en **Proyecto** -> **Propiedades** -> **Generar** -> **Salida** , marque la casilla de verificación del `XML documentation file` :




Cuando compile el proyecto, el compilador producirá un archivo XML con un nombre que corresponde al nombre del proyecto (por ejemplo, `XMLDocumentation.dll` -> `XMLDocumentation.xml` ).

Cuando use el ensamblaje en otro proyecto, asegúrese de que el archivo XML esté en el mismo directorio que la DLL a la que se hace referencia.

Este ejemplo:

```
/// <summary>
/// Data class description
/// </summary>
public class DataClass
{
    /// <summary>
    /// Name property description
    /// </summary>
    public string Name { get; set; }
}


/// <summary>
/// Foo function
/// </summary>
public class Foo
{
    /// <summary>
    /// This method returning some data
    /// </summary>
    /// <param name="id">Id parameter</param>
    /// <param name="time">Time parameter</param>
    /// <returns>Data will be returned</returns>
    public DataClass GetData(int id, DateTime time)
    {
        return new DataClass();
    }
}
```

Produce este xml en la compilación:

```
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>XMLDocumentation</name>
    </assembly>
    <members>
        <member name="T:XMLDocumentation.DataClass">
            <summary>
            Data class description
            </summary>
        </member>
        <member name="P:XMLDocumentation.DataClass.Name">
            <summary>
            Name property description
            </summary>
        </member>
        <member name="T:XMLDocumentation.Foo">
            <summary>
            Foo function
            </summary>
        </member>
        <member name="M:XMLDocumentation.Foo.GetData(System.Int32,System.DateTime)">
            <summary>
            This method returning some data
            </summary>
            <param name="id">Id parameter</param>
            <param name="time">Time parameter</param>
            <returns>Data will be returned</returns>
        </member>
    </members>
</doc>
```

## Haciendo referencia a otra clase en documentación

La etiqueta `<see>` se puede utilizar para vincular a otra clase. Contiene el miembro `cref` que debe contener el nombre de la clase a la que se va a hacer referencia. Visual Studio proporcionará Intellsense al escribir esta etiqueta y dichas referencias se procesarán al cambiar el nombre de la clase a la que se hace referencia, también.

```
/// <summary>
/// You might also want to check out <see cref="SomeOtherClass"/>.
/// </summary>
public class SomeClass
{
}
```

En las ventanas emergentes de Visual Studio Intellisense, estas referencias también se mostrarán en color en el texto.

Para hacer referencia a una clase genérica, use algo similar a lo siguiente:

```
/// <summary>
/// An enhanced version of <see cref="List{T}"/>.
```

```
/// </summary>
public class SomeGenericClass<T>
{
}
```

Lea Documentación XML Comentarios en línea:
https://riptutorial.com/es/csharp/topic/740/documentacion-xml-comentarios

# Capítulo 57: Ejemplos de AssemblyInfo.cs

## Observaciones

El nombre de archivo `AssemblyInfo.cs` se usa por convención como el archivo de origen donde los desarrolladores colocan atributos de metadatos que describen el ensamblaje completo que están construyendo.

## Examples

### [Título de la asamblea]

Este atributo se utiliza para dar un nombre a este conjunto particular.

```
[assembly: AssemblyTitle("MyProduct")]
```

### [Producto de la Asamblea]

Este atributo se utiliza para describir el producto para el que se utiliza este ensamblaje en particular. Los conjuntos múltiples pueden ser componentes del mismo producto, en cuyo caso todos pueden compartir el mismo valor para este atributo.

```
[assembly: AssemblyProduct("MyProduct")]
```

### AssemblyInfo global y local

Al tener un permiso global para un mejor DRYness, solo necesita poner valores que sean diferentes en AssemblyInfo.cs para proyectos que tengan variaciones. Este uso asume que su producto tiene más de un proyecto de estudio visual.

GlobalAssemblyInfo.cs

```
using System.Reflection;
using System.Runtime.InteropServices;
//using Stackoverflow domain as a made up example

// It is common, and mostly good, to use one GlobalAssemblyInfo.cs that is added
// as a link to many projects of the same product, details below
// Change these attribute values in local assembly info to modify the information.
[assembly: AssemblyProduct("Stackoverflow Q&A")]
[assembly: AssemblyCompany("Stackoverflow")]
[assembly: AssemblyCopyright("Copyright © Stackoverflow 2016")]

// The following GUID is for the ID of the typelib if this project is exposed to COM
[assembly: Guid("4e4f2d33-aaab-48ea-a63d-1f0a8e3c935f")]
[assembly: ComVisible(false)] //not going to expose ;)

// Version information for an assembly consists of the following four values:
```

```
// roughly translated from I reckon it is for SO, note that they most likely
// dynamically generate this file
//      Major Version  - Year 6 being 2016
//      Minor Version  - The month
//      Day Number     - Day of month
//      Revision       - Build number
// You can specify all the values or you can default the Build and Revision Numbers
// by using the '*' as shown below: [assembly: AssemblyVersion("year.month.day.*")]
[assembly: AssemblyVersion("2016.7.00.00")]
[assembly: AssemblyFileVersion("2016.7.27.3839")]
```

AssemblyInfo.cs - uno para cada proyecto

```
//then the following might be put into a separate Assembly file per project, e.g.
[assembly: AssemblyTitle("Stackoveflow.Redis")]
```

Puede agregar GlobalAssemblyInfo.cs al proyecto local mediante el siguiente procedimiento :

1. Seleccione Agregar / artículo existente ... en el menú contextual del proyecto
2. Seleccione GlobalAssemblyInfo.cs
3. Expanda el botón Agregar haciendo clic en esa pequeña flecha hacia abajo en la mano derecha
4. Seleccione "Agregar como enlace" en la lista desplegable de botones

## [AssemblyVersion]

Este atributo aplica una versión al ensamblaje.

```
[assembly: AssemblyVersion("1.0.*")]
```

El carácter `*` se usa para incrementar automáticamente una parte de la versión automáticamente cada vez que compila (a menudo se usa para el número de "compilación")

## Leyendo los atributos de la asamblea

Usando las API de reflexión enriquecida de .NET, puede obtener acceso a los metadatos de un ensamblaje. Por ejemplo, puede obtener el atributo de título de `this` ensamblaje con el siguiente código

```
using System.Linq;
using System.Reflection;

...

Assembly assembly = typeof(this).Assembly;
var titleAttribute = assembly.GetCustomAttributes<AssemblyTitleAttribute>().FirstOrDefault();

Console.WriteLine($"This assembly title is {titleAttribute?.Title}");
```

## Control de versiones automatizado

Su código en el control de origen tiene números de versión de forma predeterminada (identificadores de SVN o hash Git SHA1) o explícitamente (etiquetas Git). En lugar de actualizar manualmente las versiones en AssemblyInfo.cs, puede utilizar un proceso de compilación para escribir la versión de su sistema de control de origen en sus archivos AssemblyInfo.cs y, por lo tanto, en sus ensamblajes.

Los paquetes GitVersionTask o SemVer.Git.Fody NuGet son ejemplos de lo anterior. Para usar GitVersionTask, por ejemplo, después de instalar el paquete en su proyecto, elimine los atributos de la `Assembly*Version` de sus archivos AssemblyInfo.cs. Esto pone a GitVersionTask a cargo de la versión de sus ensamblajes.

Tenga en cuenta que el control de versiones semántico es cada vez más el estándar *de facto* , por lo que estos métodos recomiendan el uso de etiquetas de control de origen que siguen a SemVer.

## Campos comunes

Es una buena práctica completar los campos predeterminados de AssemblyInfo. La información puede ser recogida por los instaladores y luego aparecerá cuando utilice Programas y características (Windows 10) para desinstalar o cambiar un programa.

El mínimo debe ser:

- AssemblyTitle: normalmente el espacio de nombres, *es decir* , MyCompany.MySolution.MyProject
- AssemblyCompany - el nombre completo de las personas jurídicas
- Producto de la asamblea - la comercialización puede tener una visión aquí
- AssemblyCopyright - Manténgalo actualizado ya que se ve desaliñado de lo contrario

'AssemblyTitle' se convierte en la 'Descripción del archivo' al examinar la pestaña de Detalles de propiedades de la DLL.

## [Configuración de la Asamblea]

AssemblyConfiguration: El atributo AssemblyConfiguration debe tener la configuración que se usó para construir el ensamblaje. Utilice la compilación condicional para incluir correctamente diferentes configuraciones de ensamblaje. Usa el bloque similar al siguiente ejemplo. Agregue tantas configuraciones diferentes como comúnmente usa.

```
#if (DEBUG)

[assembly: AssemblyConfiguration("Debug")]

#else

[assembly: AssemblyConfiguration("Release")]

#endif
```

## [InternalsVisibleTo]

Si desea que `internal` clases o funciones `internal` de un conjunto sean accesibles desde otro conjunto, declare esto mediante `InternalsVisibleTo` y el nombre del conjunto al que se permite acceder.

En este ejemplo, el código en el ensamblaje `MyAssembly.UnitTests` puede llamar a elementos `internal` desde `MyAssembly` .

```
[assembly: InternalsVisibleTo("MyAssembly.UnitTests")]
```

Esto es especialmente útil para pruebas de unidad para prevenir declaraciones `public` innecesarias.

## [AssemblyKeyFile]

Cuando queremos que nuestro ensamblaje se instale en GAC, es necesario tener un nombre sólido. Para un conjunto de denominación fuerte tenemos que crear una clave pública. Para generar el archivo `.snk` .

Para crear un archivo de clave de nombre seguro

1. Símbolo del sistema de desarrolladores para VS2015 (con acceso de administrador)
2. En el símbolo del sistema, escriba cd C: \ Directory_Name y presione ENTRAR.
3. En el símbolo del sistema, escriba sn -k KeyFileName.snk y, a continuación, presione ENTRAR.

una vez que se haya creado el keyFileName.snk en el directorio especificado, déle una referencia en su proyecto. del atributo `AssemblyKeyFileAttribute` la ruta al archivo `snk` para generar la clave cuando construimos nuestra biblioteca de clases.

Propiedades -> AssemblyInfo.cs

```
[assembly: AssemblyKeyFile(@"c:\Directory_Name\KeyFileName.snk")]
```

Thi creará un ensamblaje de nombre seguro después de la construcción. Después de crear su conjunto de nombre seguro, puede instalarlo en GAC

Feliz codificación :)

Lea Ejemplos de AssemblyInfo.cs en línea: https://riptutorial.com/es/csharp/topic/4264/ejemplos-de-assemblyinfo-cs

# Capítulo 58: Encuadernación

## Examples

### Evitando la iteración N * 2

Esto se coloca en un controlador de eventos de Windows Forms

```
var nameList = new BindingList<string>();
ComboBox1.DataSource = nameList;
for(long i = 0; i < 10000; i++ ) {
    nameList.AddRange(new [] {"Alice", "Bob", "Carol" });
}
```

Esto demora mucho tiempo en ejecutarse, arreglarse, hacer lo siguiente:

```
var nameList = new BindingList<string>();
ComboBox1.DataSource = nameList;
nameList.RaiseListChangedEvents = false;
for(long i = 0; i < 10000; i++ ) {
    nameList.AddRange(new [] {"Alice", "Bob", "Carol" });
}
nameList.RaiseListChangedEvents = true;
nameList.ResetBindings();
```

### Añadir elemento a la lista

```
BindingList<string> listOfUIItems = new BindingList<string>();
listOfUIItems.Add("Alice");
listOfUIItems.Add("Bob");
```

Lea Encuadernación en línea: https://riptutorial.com/es/csharp/topic/182/encuadernacion--t-

# Capítulo 59: Enhebrado

## Observaciones

Un **hilo** es una parte de un programa que puede ejecutarse independientemente de otras partes. Puede realizar tareas simultáneamente con otros hilos. **El subprocesamiento múltiple** es una función que permite a los programas realizar un procesamiento simultáneo para que se pueda realizar más de una operación a la vez.

Por ejemplo, puede utilizar subprocesos para actualizar un temporizador o contador en segundo plano mientras realiza simultáneamente otras tareas en primer plano.

Las aplicaciones de subprocesos múltiples responden mejor a las aportaciones de los usuarios y también son fácilmente escalables, porque el desarrollador puede agregar subprocesos a medida que aumenta la carga de trabajo.

Por defecto, un programa C # tiene un hilo: el hilo principal del programa. Sin embargo, los subprocesos secundarios se pueden crear y utilizar para ejecutar código en paralelo con el subproceso principal. Tales hilos se llaman hilos de trabajo.

Para controlar el funcionamiento de un subproceso, el CLR delega una función al sistema operativo conocido como Thread Scheduler. Un programador de hilos asegura que a todos los hilos se les asigna el tiempo de ejecución adecuado. También comprueba que los subprocesos que están bloqueados o bloqueados no consumen mucho tiempo de CPU.

El .NET Framework `System.Threading` espacio de nombres facilita el uso de subprocesos. System.Threading permite el subprocesamiento múltiple al proporcionar una serie de clases e interfaces. Además de proporcionar tipos y clases para un subproceso en particular, también define tipos para contener una colección de subprocesos, una clase de temporizador, etc. También proporciona su soporte al permitir el acceso sincronizado a los datos compartidos.

`Thread` es la clase principal en el `System.Threading` nombres `System.Threading` . Otras clases incluyen `AutoResetEvent` , `Interlocked` , `Monitor` , `Mutex` y `ThreadPool` .

Algunos de los delegados que están presentes en el `System.Threading` nombres de `System.Threading` incluyen `ThreadStart` , `TimerCallback` y `WaitCallback` .

Las enumeraciones en el `System.Threading` nombres de `System.Threading` incluyen `ThreadPriority` , `ThreadState` y `EventResetMode` .

En .NET Framework 4 y versiones posteriores, la programación multihilo se hace más fácil y sencilla a través de los `System.Threading.Tasks.Parallel` y `System.Threading.Tasks.Task` nuevas clases de colección concurrentes clases, Parallel LINQ (PLINQ), en los `System.Collections.Concurrent` nombres `System.Collections.Concurrent` y un nuevo modelo de programación basado en tareas.

# Examples

## Demostración de subprocesos completa simple

```
class Program
{
    static void Main(string[] args)
    {
        // Create 2 thread objects.  We're using delegates because we need to pass
        // parameters to the threads.
        var thread1 = new Thread(new ThreadStart(() => PerformAction(1)));
        var thread2 = new Thread(new ThreadStart(() => PerformAction(2)));

        // Start the threads running
        thread1.Start();
        // NB: as soon as the above line kicks off the thread, the next line starts;
        // even if thread1 is still processing.
        thread2.Start();

        // Wait for thread1 to complete before continuing
        thread1.Join();
        // Wait for thread2 to complete before continuing
        thread2.Join();

        Console.WriteLine("Done");
        Console.ReadKey();
    }

    // Simple method to help demonstrate the threads running in parallel.
    static void PerformAction(int id)
    {
        var rnd = new Random(id);
        for (int i = 0; i < 100; i++)
        {
            Console.WriteLine("Thread: {0}: {1}", id, i);
            Thread.Sleep(rnd.Next(0, 1000));
        }
    }
}
```

## Demostración de subprocesos completa simple usando tareas

```
class Program
{
    static void Main(string[] args)
    {
        // Run 2 Tasks.
        var task1 = Task.Run(() => PerformAction(1));
        var task2 = Task.Run(() => PerformAction(2));

        // Wait (i.e. block this thread) until both Tasks are complete.
        Task.WaitAll(new [] { task1, task2 });

        Console.WriteLine("Done");
        Console.ReadKey();
    }
```

```
    // Simple method to help demonstrate the threads running in parallel.
    static void PerformAction(int id)
    {
        var rnd = new Random(id);
        for (int i = 0; i < 100; i++)
        {
            Console.WriteLine("Task: {0}: {1}", id, i);
            Thread.Sleep(rnd.Next(0, 1000));
        }
    }
}
```

## Paralismo explícito de tareas

```
    private static void explicitTaskParallism()
    {
        Thread.CurrentThread.Name = "Main";

        // Create a task and supply a user delegate by using a lambda expression.
        Task taskA = new Task(() => Console.WriteLine($"Hello from task {nameof(taskA)}."));
        Task taskB = new Task(() => Console.WriteLine($"Hello from task {nameof(taskB)}."));

        // Start the task.
        taskA.Start();
        taskB.Start();

        // Output a message from the calling thread.
        Console.WriteLine("Hello from thread '{0}'.",
                          Thread.CurrentThread.Name);
        taskA.Wait();
        taskB.Wait();
        Console.Read();
    }
```

## Paralelismo implícito de tareas

```
    private static void Main(string[] args)
    {
        var a = new A();
        var b = new B();
        //implicit task parallelism
        Parallel.Invoke(
            () => a.DoSomeWork(),
            () => b.DoSomeOtherWork()
            );

      }
```

## Creando e iniciando un segundo hilo

Si está realizando varios cálculos largos, puede ejecutarlos al mismo tiempo en diferentes subprocesos en su computadora. Para hacer esto, hacemos un nuevo **hilo** y lo apuntamos a un método diferente.

```
using System.Threading;
```

```
class MainClass {
    static void Main() {
        var thread = new Thread(Secondary);
        thread.Start();
    }

    static void Secondary() {
        System.Console.WriteLine("Hello World!");
    }
}
```

## Comenzando un hilo con parámetros

utilizando System.Threading;

```
class MainClass {
    static void Main() {
        var thread = new Thread(Secondary);
        thread.Start("SecondThread");
    }

    static void Secondary(object threadName) {
        System.Console.WriteLine("Hello World from thread: " + threadName);
    }
}
```

## Creando un hilo por procesador

> `Environment.ProcessorCount` Obtiene el número de procesadores **lógicos** en la máquina actual.

El CLR luego programará cada subproceso a un procesador lógico, esto teóricamente podría significar cada subproceso en un procesador lógico diferente, todos los subprocesos en un solo procesador lógico o alguna otra combinación.

```
using System;
using System.Threading;

class MainClass {
    static void Main() {
        for (int i = 0; i < Environment.ProcessorCount; i++) {
            var thread = new Thread(Secondary);
            thread.Start(i);
        }

    }

    static void Secondary(object threadNumber) {
        System.Console.WriteLine("Hello World from thread: " + threadNumber);
    }
}
```

## Evitar leer y escribir datos simultáneamente

A veces, quieres que tus hilos compartan datos simultáneamente. Cuando esto sucede, es importante conocer el código y bloquear cualquier parte que pueda salir mal. Un ejemplo simple de dos hilos contando se muestra a continuación.

Aquí hay un código peligroso (incorrecto):

```
using System.Threading;

class MainClass
{
    static int count { get; set; }

    static void Main()
    {
        for (int i = 1; i <= 2; i++)
        {
            var thread = new Thread(ThreadMethod);
            thread.Start(i);
            Thread.Sleep(500);
        }
    }

    static void ThreadMethod(object threadNumber)
    {
        while (true)
        {
            var temp = count;
            System.Console.WriteLine("Thread " + threadNumber + ": Reading the value of
count.");
            Thread.Sleep(1000);
            count = temp + 1;
            System.Console.WriteLine("Thread " + threadNumber + ": Incrementing the value of
count to:" + count);
            Thread.Sleep(1000);
        }
    }
}
```

Notarás, en lugar de contar 1,2,3,4,5 ... contamos 1,1,2,2,3 ...

Para solucionar este problema, debemos **bloquear** el valor de conteo, de modo que varios subprocesos diferentes no puedan leer y escribir en él al mismo tiempo. Con la adición de un candado y una llave, podemos evitar que los hilos accedan a los datos simultáneamente.

```
using System.Threading;

class MainClass
{

    static int count { get; set; }
    static readonly object key = new object();

    static void Main()
    {
        for (int i = 1; i <= 2; i++)
        {
            var thread = new Thread(ThreadMethod);
```

```
            thread.Start(i);
            Thread.Sleep(500);
        }
    }

    static void ThreadMethod(object threadNumber)
    {
        while (true)
        {
            lock (key)
            {
                var temp = count;
                System.Console.WriteLine("Thread " + threadNumber + ": Reading the value of
count.");
                Thread.Sleep(1000);
                count = temp + 1;
                System.Console.WriteLine("Thread " + threadNumber + ": Incrementing the value
of count to:" + count);
            }
            Thread.Sleep(1000);
        }
    }
}
```

## Parallel.ForEach Loop

Si tiene un bucle foreach que desea acelerar y no le importa en qué orden está la salida, puede convertirlo en un bucle foreach paralelo haciendo lo siguiente:

```
using System;
using System.Threading;
using System.Threading.Tasks;

public class MainClass {

    public static void Main() {
        int[] Numbers = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
        // Single-threaded
        Console.WriteLine("Normal foreach loop: ");
        foreach (var number in Numbers) {
            Console.WriteLine(longCalculation(number));
        }
        // This is the Parallel (Multi-threaded solution)
        Console.WriteLine("Parallel foreach loop: ");
        Parallel.ForEach(Numbers, number => {
            Console.WriteLine(longCalculation(number));
        });
    }

    private static int longCalculation(int number) {
        Thread.Sleep(1000); // Sleep to simulate a long calculation
        return number * number;
    }
}
```

## Puntos muertos (dos hilos esperando el uno al otro)

Un interbloqueo es lo que ocurre cuando dos o más subprocesos están esperando que se completen o liberen un recurso de tal manera que esperen para siempre.

Un escenario típico de dos subprocesos que esperan que se completen entre sí es cuando un subproceso de la GUI de Windows Forms espera un subproceso de trabajo y el subproceso de trabajo intenta invocar un objeto administrado por el subproceso de la GUI. Observe que con este ejemplo de código, hacer clic en el botón 1 hará que el programa se bloquee.

```
private void button1_Click(object sender, EventArgs e)
{
    Thread workerthread= new Thread(dowork);
    workerthread.Start();
    workerthread.Join();
    // Do something after
}

private void dowork()
{
    // Do something before
    textBox1.Invoke(new Action(() => textBox1.Text = "Some Text"));
    // Do something after
}
```

`workerthread.Join()` es una llamada que bloquea el subproceso de llamada hasta que workerthread se complete. `textBox1.Invoke(invoke_delegate)` es una llamada que bloquea el subproceso de llamada hasta que el subproceso de la GUI haya procesado invoke_delegate, pero esta llamada provoca interbloqueos si el subproceso de la GUI ya está esperando a que se complete el subproceso de la llamada.

Para solucionar esto, uno puede usar una forma no bloqueante de invocar el cuadro de texto en su lugar:

```
private void dowork()
{
    // Do work
    textBox1.BeginInvoke(new Action(() => textBox1.Text = "Some Text"));
    // Do work that is not dependent on textBox1 being updated first
}
```

Sin embargo, esto causará problemas si necesita ejecutar un código que depende de que el cuadro de texto se actualice primero. En ese caso, ejecute eso como parte de la invocación, pero tenga en cuenta que esto hará que se ejecute en el hilo de la GUI.

```
private void dowork()
{
    // Do work
    textBox1.BeginInvoke(new Action(() => {
        textBox1.Text = "Some Text";
        // Do work dependent on textBox1 being updated first,
        // start another worker thread or raise an event
    }));
    // Do work that is not dependent on textBox1 being updated first
}
```

Alternativamente, inicie un hilo completamente nuevo y deje que ese haga la espera en el hilo de la GUI, para que ese hilo de trabajo pueda completarse.

```
private void dowork()
{
    // Do work
    Thread workerthread2 = new Thread(() =>
    {
        textBox1.Invoke(new Action(() => textBox1.Text = "Some Text"));
        // Do work dependent on textBox1 being updated first,
        // start another worker thread or raise an event
    });
    workerthread2.Start();
    // Do work that is not dependent on textBox1 being updated first
}
```

Para minimizar el riesgo de encontrarse con un interbloqueo de espera mutua, siempre evite referencias circulares entre hilos cuando sea posible. Una jerarquía de subprocesos en la que los subprocesos de menor rango solo dejan mensajes para los subprocesos de mayor rango y nunca los espera, no se encontrará con este tipo de problema. Sin embargo, aún sería vulnerable a puntos muertos en función del bloqueo de recursos.

## Puntos muertos (mantener el recurso y esperar)

Un interbloqueo es lo que ocurre cuando dos o más subprocesos están esperando que se completen o liberen un recurso de tal manera que esperen para siempre.

Si thread1 mantiene un bloqueo en el recurso A y está esperando que se libere el recurso B, mientras que thread2 mantiene el recurso B y está esperando que se libere el recurso A, están bloqueados.

Al hacer clic en el botón 1 para el siguiente código de ejemplo, su aplicación entrará en el estado de interbloqueo mencionado anteriormente y se bloqueará

```
private void button_Click(object sender, EventArgs e)
{
    DeadlockWorkers workers = new DeadlockWorkers();
    workers.StartThreads();
    textBox.Text = workers.GetResult();
}

private class DeadlockWorkers
{
    Thread thread1, thread2;

    object resourceA = new object();
    object resourceB = new object();

    string output;

    public void StartThreads()
    {
        thread1 = new Thread(Thread1DoWork);
        thread2 = new Thread(Thread2DoWork);
```

```
        thread1.Start();
        thread2.Start();
    }

    public string GetResult()
    {
        thread1.Join();
        thread2.Join();
        return output;
    }

    public void Thread1DoWork()
    {
        Thread.Sleep(100);
        lock (resourceA)
        {
            Thread.Sleep(100);
            lock (resourceB)
            {
                output += "T1#";
            }
        }
    }

    public void Thread2DoWork()
    {
        Thread.Sleep(100);
        lock (resourceB)
        {
            Thread.Sleep(100);
            lock (resourceA)
            {
                output += "T2#";
            }
        }
    }
}
```

Para evitar estar bloqueado de esta manera, se puede usar Monitor.TryEnter (lock_object, timeout_in_milliseconds) para verificar si ya hay un bloqueo en un objeto. Si Monitor.TryEnter no logra adquirir un bloqueo en lock_object antes de timeout_in_milliseconds, devuelve falso, lo que le da al hilo la oportunidad de liberar otros recursos retenidos y el rendimiento, dando así a otros hilos la oportunidad de completar, como en esta versión ligeramente modificada del anterior :

```
private void button_Click(object sender, EventArgs e)
{
    MonitorWorkers workers = new MonitorWorkers();
    workers.StartThreads();
    textBox.Text = workers.GetResult();
}

private class MonitorWorkers
{
    Thread thread1, thread2;

    object resourceA = new object();
    object resourceB = new object();

    string output;
```

```
    public void StartThreads()
    {
        thread1 = new Thread(Thread1DoWork);
        thread2 = new Thread(Thread2DoWork);
        thread1.Start();
        thread2.Start();
    }

    public string GetResult()
    {
        thread1.Join();
        thread2.Join();
        return output;
    }

    public void Thread1DoWork()
    {
        bool mustDoWork = true;
        Thread.Sleep(100);
        while (mustDoWork)
        {
            lock (resourceA)
            {
                Thread.Sleep(100);
                if (Monitor.TryEnter(resourceB, 0))
                {
                    output += "T1#";
                    mustDoWork = false;
                    Monitor.Exit(resourceB);
                }
            }
            if (mustDoWork) Thread.Yield();
        }
    }

    public void Thread2DoWork()
    {
        Thread.Sleep(100);
        lock (resourceB)
        {
            Thread.Sleep(100);
            lock (resourceA)
            {
                output += "T2#";
            }
        }
    }
}
```

Tenga en cuenta que esta solución temporal se basa en que el subproceso2 es obstinado en cuanto a sus bloqueos y el subproceso1 está dispuesto a ceder, de modo que el subproceso2 siempre tiene prioridad. También tenga en cuenta que el subproceso1 debe rehacer el trabajo que realizó después de bloquear el recurso A, cuando cede. Por lo tanto, tenga cuidado al implementar este enfoque con más de un subproceso de rendimiento, ya que correrá el riesgo de ingresar a un llamado Livelock, un estado que se produciría si dos subprocesos siguieran haciendo el primer bit de su trabajo y luego se rindieran mutuamente. , volviendo a empezar repetidamente.

Lea Enhebrado en línea: https://riptutorial.com/es/csharp/topic/51/enhebrado

# Capítulo 60: Enumerable

## Introducción

`IEnumerable` es la interfaz base para todas las colecciones no genéricas como ArrayList que se pueden enumerar. `IEnumerator<T>` es la interfaz base para todos los enumeradores genéricos como List <>.

`IEnumerable` es una interfaz que implementa el método `GetEnumerator` . El método `GetEnumerator` devuelve un `IEnumerator` que proporciona opciones para iterar a través de la colección como foreach.

## Observaciones

IEnumerable es la interfaz base para todas las colecciones no genéricas que se pueden enumerar

## Examples

### Enumerable

En su forma más básica, un objeto que implementa IEnumerable representa una serie de objetos. Los objetos en cuestión se pueden iterar usando la palabra clave c # `foreach` .

En el siguiente ejemplo, el objeto `sequenceOfNumbers` implementa IEnumerable. Representa una serie de enteros. El bucle `foreach` recorre cada uno de ellos.

```
int AddNumbers(IEnumerable<int> sequenceOfNumbers) {
    int returnValue = 0;
    foreach(int i in sequenceOfNumbers) {
        returnValue += i;
    }
    return returnValue;
}
```

### Enumerable con enumerador personalizado

La implementación de la interfaz IEnumerable permite que las clases se enumeren de la misma manera que las colecciones BCL. Esto requiere extender la clase Enumerator que rastrea el estado de la enumeración.

Aparte de iterar sobre una colección estándar, los ejemplos incluyen:

- Uso de rangos de números basados en una función en lugar de una colección de objetos
- Implementar diferentes algoritmos de iteración sobre colecciones, como DFS o BFS en una colección de gráficos

```
public static void Main(string[] args) {

    foreach (var coffee in new CoffeeCollection()) {
        Console.WriteLine(coffee);
    }
}

public class CoffeeCollection : IEnumerable {
    private CoffeeEnumerator enumerator;

    public CoffeeCollection() {
        enumerator = new CoffeeEnumerator();
    }

    public IEnumerator GetEnumerator() {
        return enumerator;
    }

    public class CoffeeEnumerator : IEnumerator {
        string[] beverages = new string[3] { "espresso", "macchiato", "latte" };
        int currentIndex = -1;

        public object Current {
            get {
                return beverages[currentIndex];
            }
        }

        public bool MoveNext() {
            currentIndex++;

            if (currentIndex < beverages.Length) {
                return true;
            }

            return false;
        }

        public void Reset() {
            currentIndex = 0;
        }
    }
}
```

Lea Enumerable en línea: https://riptutorial.com/es/csharp/topic/2220/enumerable

# Capítulo 61: Enumerar

## Introducción

Una enumeración puede derivar de cualquiera de los siguientes tipos: byte, sbyte, short, ushort, int, uint, long, ulong. El valor predeterminado es int, y se puede cambiar especificando el tipo en la definición de enumeración:

enumeración pública Día de la semana: byte {lunes = 1, martes = 2, miércoles = 3, jueves = 4, viernes = 5}

Esto es útil cuando P / Invoking a código nativo, mapeo a fuentes de datos y circunstancias similares. En general, se debe usar el int predeterminado, porque la mayoría de los desarrolladores esperan que una enumeración sea un int.

## Sintaxis

- enumeración Colores {rojo, verde, azul} // declaración de enumeración
- enum Colores: byte {Red, Green, Blue} // Declaración con un tipo específico
- enumeración Colores {rojo = 23, verde = 45, azul = 12} // Declaración con valores definidos
- Colors.Red // Accede a un elemento de un Enum
- int value = (int) Colors.Red // Obtener el valor int de un elemento enum
- Colors color = (Colors) intValue // Obtener un elemento de enumeración de int

## Observaciones

Un Enum (abreviatura de "tipo enumerado") es un tipo que consiste en un conjunto de constantes con nombre, representado por un identificador específico del tipo.

Las enumeraciones son más útiles para representar conceptos que tienen un número (generalmente pequeño) de posibles valores discretos. Por ejemplo, se pueden usar para representar un día de la semana o un mes del año. También pueden usarse como indicadores que pueden combinarse o comprobarse mediante operaciones a nivel de bits.

## Examples

### Obtener todos los valores de los miembros de una enumeración

```
enum MyEnum
{
    One,
    Two,
    Three
}

foreach(MyEnum e in Enum.GetValues(typeof(MyEnum)))
```

```
    Console.WriteLine(e);
```

Esto imprimirá:

```
One
Two
Three
```

## Enumerar como banderas

El `FlagsAttribute` se puede aplicar a una enumeración que cambia el comportamiento de `ToString()` para que coincida con la naturaleza de la enumeración:

```
[Flags]
enum MyEnum
{
    //None = 0, can be used but not combined in bitwise operations
    FlagA = 1,
    FlagB = 2,
    FlagC = 4,
    FlagD = 8
    //you must use powers of two or combinations of powers of two
    //for bitwise operations to work
}

var twoFlags = MyEnum.FlagA | MyEnum.FlagB;

// This will enumerate all the flags in the variable: "FlagA, FlagB".
Console.WriteLine(twoFlags);
```

Debido a que `FlagsAttribute` basa en las constantes de enumeración para ser potencias de dos (o sus combinaciones) y los valores de enumeración son valores numéricos en última instancia, está limitado por el tamaño del tipo numérico subyacente. El tipo numérico más grande disponible que puede usar es `UInt64` , que le permite especificar 64 constantes de enumeración de bandera distintas (no combinadas). La palabra clave `enum` predeterminada en el tipo subyacente `int` , que es `Int32` . El compilador permitirá la declaración de valores mayores a 32 bit. Estos se envolverán sin previo aviso y darán como resultado dos o más miembros del mismo valor. Por lo tanto, si se pretende que una enumeración acomode un conjunto de bits de más de 32 indicadores, debe especificar un tipo más grande explícitamente:

```
public enum BigEnum : ulong
{
    BigValue = 1 << 63
}
```

Aunque las banderas suelen ser solo un bit, se pueden combinar en "conjuntos" con nombre para facilitar su uso.

```
[Flags]
enum FlagsEnum
{
    None = 0,
```

```
    Option1 = 1,
    Option2 = 2,
    Option3 = 4,

    Default = Option1 | Option3,
    All = Option1 | Option2 | Option3,
}
```

Para evitar deletrear los valores decimales de potencias de dos, el operador de desplazamiento a la izquierda (<<) también se puede utilizar para declarar la misma enumeración

```
[Flags]
enum FlagsEnum
{
    None = 0,
    Option1 = 1 << 0,
    Option2 = 1 << 1,
    Option3 = 1 << 2,

    Default = Option1 | Option3,
    All = Option1 | Option2 | Option3,
}
```

A partir de C # 7.0, también se pueden utilizar literales binarios .

Para verificar si el valor de la variable enum tiene un determinado conjunto de indicadores, se puede usar el método `HasFlag` . Digamos que tenemos

```
[Flags]
enum MyEnum
{
    One = 1,
    Two = 2,
    Three = 4
}
```

Y un `value`

```
var value = MyEnum.One | MyEnum.Two;
```

Con `HasFlag` podemos comprobar si alguna de las banderas está `HasFlag` .

```
if(value.HasFlag(MyEnum.One))
    Console.WriteLine("Enum has One");

if(value.HasFlag(MyEnum.Two))
    Console.WriteLine("Enum has Two");

if(value.HasFlag(MyEnum.Three))
    Console.WriteLine("Enum has Three");
```

También podemos iterar a través de todos los valores de enumeración para obtener todos los indicadores que se establecen

```
var type = typeof(MyEnum);
var names = Enum.GetNames(type);

foreach (var name in names)
{
    var item = (MyEnum)Enum.Parse(type, name);

    if (value.HasFlag(item))
        Console.WriteLine("Enum has " + name);
}
```

O

```
foreach(MyEnum flagToCheck in Enum.GetValues(typeof(MyEnum)))
{
    if(value.HasFlag(flagToCheck))
    {
         Console.WriteLine("Enum has " + flagToCheck);
    }
}
```

Los tres ejemplos se imprimirán:

```
Enum has One
Enum has Two
```

## Probar los valores de enumeración de estilo de los indicadores con lógica bitwise

Un valor de enumeración de estilo de indicadores debe probarse con lógica bitwise porque puede no coincidir con ningún valor único.

```
[Flags]
enum FlagsEnum
{
    Option1 = 1,
    Option2 = 2,
    Option3 = 4,
    Option2And3 = Option2 | Option3;

    Default = Option1 | Option3,
}
```

El valor `Default` es en realidad una combinación de otros dos *combinados* con un OR a nivel de bits. Por lo tanto, para probar la presencia de una bandera necesitamos usar un AND a nivel de bits.

```
var value = FlagsEnum.Default;

bool isOption2And3Set = (value & FlagsEnum.Option2And3) == FlagsEnum.Option2And3;

Assert.True(isOption2And3Set);
```

## Enumerar a la cuerda y la espalda.

```
public enum DayOfWeek
{
    Sunday,
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday
}


// Enum to string
string thursday = DayOfWeek.Thursday.ToString(); // "Thursday"

string seventhDay = Enum.GetName(typeof(DayOfWeek), 6); // "Saturday"

string monday = Enum.GetName(typeof(DayOfWeek), DayOfWeek.Monday); // "Monday"


// String to enum (.NET 4.0+ only - see below for alternative syntax for earlier .NET
versions)
DayOfWeek tuesday;
Enum.TryParse("Tuesday", out tuesday); // DayOfWeek.Tuesday

DayOfWeek sunday;
bool matchFound1 = Enum.TryParse("SUNDAY", out sunday); // Returns false (case-sensitive
match)

DayOfWeek wednesday;
bool matchFound2 = Enum.TryParse("WEDNESDAY", true, out wednesday); // Returns true;
DayOfWeek.Wednesday (case-insensitive match)


// String to enum (all .NET versions)
DayOfWeek friday = (DayOfWeek)Enum.Parse(typeof(DayOfWeek), "Friday"); // DayOfWeek.Friday

DayOfWeek caturday = (DayOfWeek)Enum.Parse(typeof(DayOfWeek), "Caturady"); // Thows
ArgumentException

// All names of an enum type as strings
string[] weekdays = Enum.GetNames(typeof(DayOfWeek));
```

## Valor predeterminado para enumeración == CERO

**El valor predeterminado para una enumeración es cero** . Si una enumeración no define un elemento con un valor de cero, su valor predeterminado será cero.

```
public class Program
{
    enum EnumExample
    {
        one = 1,
        two = 2
    }
```

```
    public void Main()
    {
        var e = default(EnumExample);

        if (e == EnumExample.one)
            Console.WriteLine("defaults to one");
        else
            Console.WriteLine("Unknown");
    }
}
```

Ejemplo: https://dotnetfiddle.net/l5Rwie

## Conceptos básicos de enumeración

Desde MSDN :

> Un tipo de enumeración (también denominado enumeración o enumeración) proporciona una manera eficiente de definir un conjunto de **constantes integrales con** nombre que pueden **asignarse a una variable** .

Esencialmente, una enumeración es un tipo que solo permite un conjunto de opciones finitas, y cada opción corresponde a un número. Por defecto, esos números aumentan en el orden en que se declaran los valores, comenzando desde cero. Por ejemplo, uno podría declarar una enumeración para los días de la semana:

```
public enum Day
{
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday,
    Sunday
}
```

Esa enumeración se podría utilizar de esta manera:

```
// Define variables with values corresponding to specific days
Day myFavoriteDay = Day.Friday;
Day myLeastFavoriteDay = Day.Monday;

// Get the int that corresponds to myFavoriteDay
// Friday is number 4
int myFavoriteDayIndex = (int)myFavoriteDay;

// Get the day that represents number 5
Day dayFive = (Day)5;
```

Por defecto, el tipo subyacente de cada elemento en la `enum` es `int` , pero también se pueden usar `byte` , `sbyte` , `short` , `ushort` , `uint` , `long` y `ulong` . Si usa un tipo que no sea `int` , debe especificar el tipo usando dos puntos después del nombre enum:

```
public enum Day : byte
{
    // same as before
}
```

Los números después del nombre ahora son bytes en lugar de números enteros. Puede obtener el tipo subyacente de la enumeración de la siguiente manera:

```
Enum.GetUnderlyingType(typeof(Days)));
```

Salida:

```
System.Byte
```

Demostración: .NET violín

## Manipulación bitwise usando enumeraciones

FlagsAttribute se debe usar siempre que la enumeración represente una colección de banderas, en lugar de un solo valor. El valor numérico asignado a cada valor de enumeración ayuda a la hora de manipular las enumeraciones con operadores bitwise.

**Ejemplo 1: Con [Flags]**

```
[Flags]
enum Colors
{
    Red=1,
    Blue=2,
    Green=4,
    Yellow=8
}

var color = Colors.Red | Colors.Blue;
Console.WriteLine(color.ToString());
```

estampados rojo, azul

**Ejemplo 2: Sin [Flags]**

```
enum Colors
{
    Red=1,
    Blue=2,
    Green=4,
    Yellow=8
}
var color = Colors.Red | Colors.Blue;
Console.WriteLine(color.ToString());
```

impresiones 3

## Usando << notación para banderas

El operador de desplazamiento a la izquierda ( `<<` ) se puede utilizar en las declaraciones de enumeración de bandera para garantizar que cada bandera tenga exactamente un `1` en representación binaria, como deben hacerlo las banderas.

Esto también ayuda a mejorar la legibilidad de grandes enums con un montón de banderas en ellos.

```
[Flags]
public enum MyEnum
{
    None  = 0,
    Flag1 = 1 << 0,
    Flag2 = 1 << 1,
    Flag3 = 1 << 2,
    Flag4 = 1 << 3,
    Flag5 = 1 << 4,
    ...
    Flag31 = 1 << 30
}
```

Es obvio ahora que `MyEnum` contiene solo indicadores apropiados y no elementos desordenados como `Flag30 = 1073741822` (o 11111111111111111111111111110 en binario) lo cual es inapropiado.

## Agregar información de descripción adicional a un valor de enumeración

En algunos casos, es posible que desee agregar una descripción adicional a un valor de enumeración, por ejemplo, cuando el propio valor de enumeración es menos legible de lo que le gustaría mostrarle al usuario. En tales casos, puede usar la clase `System.ComponentModel.DescriptionAttribute` .

Por ejemplo:

```
public enum PossibleResults
{
    [Description("Success")]
    OK = 1,
    [Description("File not found")]
    FileNotFound = 2,
    [Description("Access denied")]
    AccessDenied = 3
}
```

Ahora, si desea devolver la descripción de un valor de enumeración específico, puede hacer lo siguiente:

```
public static string GetDescriptionAttribute(PossibleResults result)
{
        return
((DescriptionAttribute)Attribute.GetCustomAttribute((result.GetType().GetField(result.ToString())),
```

```
typeof(DescriptionAttribute))).Description;
}

static void Main(string[] args)
{
    PossibleResults result = PossibleResults.FileNotFound;
    Console.WriteLine(result); // Prints "FileNotFound"
    Console.WriteLine(GetDescriptionAttribute(result)); // Prints "File not found"
}
```

Esto también se puede transformar fácilmente en un método de extensión para todas las enumeraciones:

```
static class EnumExtensions
{
    public static string GetDescription(this Enum enumValue)
    {
        return
((DescriptionAttribute)Attribute.GetCustomAttribute((enumValue.GetType().GetField(enumValue.ToString())
typeof(DescriptionAttribute))).Description;
    }
}
```

Y luego se usa fácilmente de esta manera: `Console.WriteLine(result.GetDescription());`

## Agregar y eliminar valores de enumeración marcada

Este código es para agregar y eliminar un valor de una instancia de enumeración marcada:

```
[Flags]
public enum MyEnum
{
    Flag1 = 1 << 0,
    Flag2 = 1 << 1,
    Flag3 = 1 << 2
}

var value = MyEnum.Flag1;

// set additional value
value |= MyEnum.Flag2;  //value is now Flag1, Flag2
value |= MyEnum.Flag3;  //value is now Flag1, Flag2, Flag3

// remove flag
value &= ~MyEnum.Flag2; //value is now Flag1, Flag3
```

## Las enumeraciones pueden tener valores inesperados.

Dado que una enumeración se puede convertir desde y hacia su tipo de integral subyacente, el valor puede caer fuera del rango de valores que figura en la definición del tipo de enumeración.

Aunque el siguiente tipo de enumeración `DaysOfWeek` solo tiene 7 valores definidos, aún puede contener cualquier valor `int` .

```
public enum DaysOfWeek
{
    Monday = 1,
    Tuesday = 2,
    Wednesday = 3,
    Thursday = 4,
    Friday = 5,
    Saturday = 6,
    Sunday = 7
}

DaysOfWeek d = (DaysOfWeek)31;
Console.WriteLine(d); // prints 31

DaysOFWeek s = DaysOfWeek.Sunday;
s++; // No error
```

Actualmente no hay manera de definir una enumeración que no tenga este comportamiento.

Sin embargo, los valores de enumeración indefinidos se pueden detectar utilizando el método `Enum.IsDefined` . Por ejemplo,

```
DaysOfWeek d = (DaysOfWeek)31;
Console.WriteLine(Enum.IsDefined(typeof(DaysOfWeek),d)); // prints False
```

Lea Enumerar en línea: https://riptutorial.com/es/csharp/topic/931/enumerar

# Capítulo 62: Equals y GetHashCode

## Observaciones

Cada implementación de `Equals` debe cumplir los siguientes requisitos:

- **Reflexivo** : un objeto debe igualarse a si mismo.
  `x.Equals(x)` devuelve `true` .
- **Simétrico** : no hay diferencia si comparo x con y o con y con x - el resultado es el mismo.
  `x.Equals(y)` devuelve el mismo valor que `y.Equals(x)` .
- **Transitivo** : si un objeto es igual a otro objeto y este es igual a un tercero, el primero debe ser igual al tercero.
  si `(x.Equals(y) && y.Equals(z))` devuelve `true` , entonces `x.Equals(z)` devuelve `true` .
- **Consistente** : si compara un objeto con otro varias veces, el resultado es siempre el mismo.

  Las invocaciones sucesivas de `x.Equals(y)` devuelven el mismo valor siempre que los objetos a los que se hace referencia con x e y no se modifiquen.
- **Comparación a nulo** : Ningún objeto es igual a `null` .
  `x.Equals(null)` devuelve `false` .

Implementaciones de `GetHashCode` :

- **Compatible con `Equals`** : si dos objetos son iguales (lo que significa que `Equals` devuelve true), `GetHashCode` **debe** devolver el mismo valor para cada uno de ellos.
- **Rango grande** : si dos objetos no son iguales ( `Equals` falso), debería haber una **alta probabilidad de que** sus códigos hash sean distintos. *El* hashing *perfecto a* menudo no es posible ya que hay un número limitado de valores para elegir.
- **Barato** : debería ser barato calcular el código hash en todos los casos.

Consulte: Pautas para la sobrecarga de Equals () y Operator ==

## Examples

### Por defecto es igual al comportamiento.

`Equals` se declara en la propia clase de `Object` .

```
public virtual bool Equals(Object obj);
```

Por defecto, `Equals` tiene el siguiente comportamiento:

- Si la instancia es un tipo de referencia, entonces `Equals` devolverá verdadero solo si las referencias son las mismas.
- Si la instancia es un tipo de valor, entonces `Equals` devolverá verdadero solo si el tipo y el valor son los mismos.
- `string` es un caso especial. Se comporta como un tipo de valor.

```
namespace ConsoleApplication
{
    public class Program
    {
        public static void Main(string[] args)
        {
            //areFooClassEqual: False
            Foo fooClass1 = new Foo("42");
            Foo fooClass2 = new Foo("42");
            bool areFooClassEqual = fooClass1.Equals(fooClass2);
            Console.WriteLine("fooClass1 and fooClass2 are equal: {0}", areFooClassEqual);
            //False

            //areFooIntEqual: True
            int fooInt1 = 42;
            int fooInt2 = 42;
            bool areFooIntEqual = fooInt1.Equals(fooInt2);
            Console.WriteLine("fooInt1 and fooInt2 are equal: {0}", areFooIntEqual);

            //areFooStringEqual: True
            string fooString1 = "42";
            string fooString2 = "42";
            bool areFooStringEqual = fooString1.Equals(fooString2);
            Console.WriteLine("fooString1 and fooString2 are equal: {0}", areFooStringEqual);
        }
    }

    public class Foo
    {
        public string Bar { get; }

        public Foo(string bar)
        {
            Bar = bar;
        }
    }
}
```

## Escribiendo un buen reemplazo de GetHashCode

`GetHashCode` tiene efectos de rendimiento importantes en Dictionary <> y HashTable.

Buenos métodos `GetHashCode`

- debe tener una distribución uniforme
  - cada entero debe tener una probabilidad aproximadamente igual de regresar para una instancia aleatoria
  - Si su método devuelve el mismo entero (por ejemplo, la constante '999') para cada

instancia, tendrá un mal rendimiento.
- debería ser rápido
    - Estos NO son hashes criptográficos, donde la lentitud es una característica
    - Cuanto más lenta sea la función hash, más lento será su diccionario.
- debe devolver el mismo HashCode en dos instancias que `Equals` evalúa como verdadero
    - si no lo hacen (por ejemplo, porque `GetHashCode` devuelve un número aleatorio), es posible que los elementos no se encuentren en una `List` , `Dictionary` o similar.

Un buen método para implementar `GetHashCode` es usar un número primo como valor de inicio y agregar los hashcodes de los campos del tipo multiplicado por otros números primos a eso:

```
public override int GetHashCode()
{
    unchecked // Overflow is fine, just wrap
    {
        int hash = 3049; // Start value (prime number).

        // Suitable nullity checks etc, of course :)
        hash = hash * 5039 + field1.GetHashCode();
        hash = hash * 883 + field2.GetHashCode();
        hash = hash * 9719 + field3.GetHashCode();
        return hash;
    }
}
```

Solo los campos que se usan en el método `Equals` deben usarse para la función hash.

Si necesita tratar el mismo tipo de diferentes maneras para Dictionary / HashTables, puede usar IEqualityComparer.

## Sobrescribe Equals y GetHashCode en tipos personalizados

Para una `Person` clase como:

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    public string Clothes { get; set; }
}

var person1 = new Person { Name = "Jon", Age = 20, Clothes = "some clothes" };
var person2 = new Person { Name = "Jon", Age = 20, Clothes = "some other clothes" };

bool result = person1.Equals(person2); //false because it's reference Equals
```

Pero definiendo `Equals` y `GetHashCode` como sigue:

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    public string Clothes { get; set; }
```

```
    public override bool Equals(object obj)
    {
        var person = obj as Person;
        if(person == null) return false;
        return Name == person.Name && Age == person.Age; //the clothes are not important when
comparing two persons
    }

    public override int GetHashCode()
    {
        return Name.GetHashCode()*Age;
    }
}

var person1 = new Person { Name = "Jon", Age = 20, Clothes = "some clothes" };
var person2 = new Person { Name = "Jon", Age = 20, Clothes = "some other clothes" };

bool result = person1.Equals(person2); // result is true
```

También al usar LINQ para hacer diferentes consultas sobre personas, se verificará `Equals` y `GetHashCode` :

```
var persons = new List<Person>
{
     new Person{ Name = "Jon", Age = 20, Clothes = "some clothes"},
     new Person{ Name = "Dave", Age = 20, Clothes = "some other clothes"},
     new Person{ Name = "Jon", Age = 20, Clothes = ""}
};

var distinctPersons = persons.Distinct().ToList();//distinctPersons has Count = 2
```

## Equals y GetHashCode en IEqualityComparator

Para el tipo de `Person` dado:

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    public string Clothes { get; set; }
}

List<Person> persons = new List<Person>
{
    new Person{ Name = "Jon", Age = 20, Clothes = "some clothes"},
    new Person{ Name = "Dave", Age = 20, Clothes = "some other clothes"},
    new Person{ Name = "Jon", Age = 20, Clothes = ""}
};

var distinctPersons = persons.Distinct().ToList();// distinctPersons has Count = 3
```

Pero definiendo `Equals` y `GetHashCode` en un `IEqualityComparator` :

```
public class PersonComparator : IEqualityComparer<Person>
{
```

```
    public bool Equals(Person x, Person y)
    {
        return x.Name == y.Name && x.Age == y.Age; //the clothes are not important when
comparing two persons;
    }

    public int GetHashCode(Person obj) { return obj.Name.GetHashCode() * obj.Age; }
}

var distinctPersons = persons.Distinct(new PersonComparator()).ToList();// distinctPersons has
Count = 2
```

Tenga en cuenta que para esta consulta, dos objetos se han considerado iguales si tanto el valor `Equals` devuelto como el código `GetHashCode` han devuelto el mismo código hash para las dos personas.

Lea Equals y GetHashCode en línea: https://riptutorial.com/es/csharp/topic/3429/equals-y-gethashcode

# Capítulo 63: Estructuras

## Observaciones

A diferencia de las clases, una `struct` es un tipo de valor, y se crea en la pila local y no en el montón administrado, *de forma predeterminada* . Esto significa que una vez que la pila específica queda fuera del alcance, la `struct` se desasigna. Los tipos de referencia contenidos de las `struct` también se barren, una vez que el GC determina que ya no están referenciados por la `struct` .

`struct` no pueden heredar y no pueden ser bases para la herencia, están implícitamente selladas y tampoco pueden incluir miembros `protected` . Sin embargo, una `struct` puede implementar una interfaz, como lo hacen las clases.

## Examples

### Declarando una estructura

```
public struct Vector
{
    public int X;
    public int Y;
    public int Z;
}

public struct Point
{
    public decimal x, y;

    public Point(decimal pointX, decimal pointY)
    {
        x = pointX;
        y = pointY;
    }
}
```

- `struct` campos de instancia de `struct` se pueden establecer a través de un constructor parametrizado o individualmente después de la construcción de `struct` .
- Los miembros privados solo pueden ser inicializados por el constructor.
- `struct` define un tipo sellado que hereda implícitamente de System.ValueType.
- Las estructuras no pueden heredar de ningún otro tipo, pero pueden implementar interfaces.
- Las estructuras se copian en la asignación, lo que significa que todos los datos se copian en la nueva instancia y los cambios en una de ellas no se reflejan en la otra.
- Una estructura no puede ser `null` , aunque se *puede* usar como un tipo anulable:

```
Vector v1 = null; //illegal
Vector? v2 = null; //OK
Nullable<Vector> v3 = null // OK
```

- Las estructuras se pueden crear instancias con o sin el uso del `new` operador.

```
//Both of these are acceptable
Vector v1 = new Vector();
v1.X = 1;
v1.Y = 2;
v1.Z = 3;

Vector v2;
v2.X = 1;
v2.Y = 2;
v2.Z = 3;
```

Sin embargo, el `new` operador debe usarse para utilizar un inicializador:

```
Vector v1 = new MyStruct { X=1, Y=2, Z=3 }; // OK
Vector v2 { X=1, Y=2, Z=3 }; // illegal
```

Una estructura puede declarar todo lo que una clase puede declarar, con algunas excepciones:

- Una estructura no puede declarar un constructor sin parámetros. `struct` campos de instancia de `struct` se pueden establecer a través de un constructor parametrizado o individualmente después de la construcción de `struct` . Los miembros privados solo pueden ser inicializados por el constructor.
- Una estructura no puede declarar miembros como protegidos, ya que está implícitamente sellada.
- Los campos Struct solo se pueden inicializar si son const o static.

## Uso estricto

**Con constructor:**

```
Vector v1 = new Vector();
v1.X = 1;
v1.Y = 2;
v1.Z = 3;

Console.WriteLine("X = {0}, Y = {1}, Z = {2}",v1.X,v1.Y,v1.Z);
// Output X=1,Y=2,Z=3

Vector v1 = new Vector();
//v1.X is not assigned
v1.Y = 2;
v1.Z = 3;

Console.WriteLine("X = {0}, Y = {1}, Z = {2}",v1.X,v1.Y,v1.Z);
// Output X=0,Y=2,Z=3

Point point1 = new Point();
```

```
point1.x = 0.5;
point1.y = 0.6;

Point point2 = new Point(0.5, 0.6);
```

**Sin constructor:**

```
Vector v1;
v1.Y = 2;
v1.Z = 3;

Console.WriteLine("X = {0}, Y = {1}, Z = {2}",v1.X,v1.Y,v1.Z);
//Output ERROR "Use of possibly unassigned field 'X'

Vector v1;
v1.X = 1;
v1.Y = 2;
v1.Z = 3;

Console.WriteLine("X = {0}, Y = {1}, Z = {2}",v1.X,v1.Y,v1.Z);
// Output X=1,Y=2,Z=3

Point point3;
point3.x = 0.5;
point3.y = 0.6;
```

Si usamos una estructura con su constructor, no tendremos problemas con el campo sin asignar (cada campo sin asignar tiene un valor nulo).

A diferencia de las clases, no es necesario construir una estructura, es decir, no es necesario utilizar la nueva palabra clave, a menos que necesite llamar a uno de los constructores. Una estructura no requiere la nueva palabra clave porque es un tipo de valor y, por lo tanto, no puede ser nula.

## Implementar la interfaz

```
public interface IShape
{
    decimal Area();
}

public struct Rectangle : IShape
{
    public decimal Length { get; set; }
    public decimal Width { get; set; }

    public decimal Area()
    {
        return Length * Width;
    }
}
```

## Las estructuras se copian en la asignación

Sinse structs son tipos de valor, todos los datos se *copian* en la asignación y cualquier

modificación de la nueva copia no modifica los datos de la copia original. El fragmento de código a continuación muestra que `p1` se *copia* a `p2` y los cambios realizados en `p1` no afectan la instancia de `p2` .

```
var p1 = new Point {
    x = 1,
    y = 2
};

Console.WriteLine($"{p1.x} {p1.y}"); // 1 2

var p2 = p1;
Console.WriteLine($"{p2.x} {p2.y}"); // Same output: 1 2

p1.x = 3;
Console.WriteLine($"{p1.x} {p1.y}"); // 3 2
Console.WriteLine($"{p2.x} {p2.y}"); // p2 remain the same: 1 2
```

Lea Estructuras en línea: https://riptutorial.com/es/csharp/topic/778/estructuras

# Capítulo 64: Eventos

## Introducción

Un evento es una notificación de que algo ocurrió (como un clic del mouse) o, en algunos casos, está a punto de ocurrir (como un cambio de precio).

Las clases pueden definir eventos y sus instancias (objetos) pueden provocar estos eventos. Por ejemplo, un botón puede contener un evento de clic que se genera cuando un usuario lo ha hecho clic.

Los controladores de eventos son métodos que se llaman cuando se genera el evento correspondiente. Un formulario puede contener un controlador de eventos de clic para cada botón que contiene, por ejemplo.

## Parámetros

| Parámetro | Detalles |
|---|---|
| EventArgsT | El tipo que se deriva de EventArgs y contiene los parámetros del evento. |
| Nombre del evento | El nombre del evento. |
| Nombre del controlador | El nombre del controlador de eventos. |
| SenderObject | El objeto que invoca el evento. |
| EventosArgumentos | Una instancia del tipo EventArgsT que contiene los parámetros del evento. |

## Observaciones

Al plantear un evento:

- Compruebe siempre si el delegado es `null` . Un delegado nulo significa que el evento no tiene suscriptores. Si se genera un evento sin suscriptores, se obtendrá una `NullReferenceException` .

6.0

- Copie el delegado (por ejemplo, `EventName` ) en una variable local (por ejemplo, `eventName` ) antes de verificar si el evento es nulo / `eventName` . Esto evita las condiciones de carrera en entornos de subprocesos múltiples:

**Mal**

```
    if(Changed != null)      // Changed has 1 subscriber at this point
                             // In another thread, that one subscriber decided to unsubscribe
        Changed(this, args); // `Changed` is now null, `NullReferenceException` is thrown.
```

**A la derecha**

```
    // Cache the "Changed" event as a local. If it is not null, then use
    // the LOCAL variable (handler) to raise the event, NOT the event itself.
    var handler = Changed;
    if(handler != null)
        handler(this, args);
```

6.0

- Use el operador condicional nulo (?.) Para elevar el método en lugar de verificar nulo al delegado en busca de suscriptores en una instrucción `if` : `EventName?.Invoke(SenderObject, new EventArgsT());`
- Cuando se usa Acción <> para declarar tipos de delegados, la firma del controlador de eventos / eventos anónimo debe ser la misma que el tipo de delegado anónimo declarado en la declaración de eventos.

# Examples

### Declarar y levantar eventos

# Declarar un evento

Puedes declarar un evento en cualquier `class` o `struct` usando la siguiente sintaxis:

```
public class MyClass
{
    // Declares the event for MyClass
    public event EventHandler MyEvent;

    // Raises the MyEvent event
    public void RaiseEvent()
    {
        OnMyEvent();
    }
}
```

Hay una sintaxis expandida para declarar eventos, donde se tiene una instancia privada del evento, y se define una instancia pública utilizando los accesores de `add` y `set` . La sintaxis es muy similar a las propiedades de C #. En todos los casos, se debe preferir la sintaxis demostrada anteriormente, ya que el compilador emite código para ayudar a garantizar que varios subprocesos puedan agregar y eliminar de forma segura los controladores de eventos al evento en su clase.

# Elevando el evento

6.0

```
private void OnMyEvent()
{
    EventName?.Invoke(this, EventArgs.Empty);
}
```

6.0

```
private void OnMyEvent()
{
    // Use a local for EventName, because another thread can modify the
    // public EventName between when we check it for null, and when we
    // raise the event.
    var eventName = EventName;

    // If eventName == null, then it means there are no event-subscribers,
    // and therefore, we cannot raise the event.
    if(eventName != null)
        eventName(this, EventArgs.Empty);

}
```

Tenga en cuenta que los eventos solo pueden ser provocados por el tipo declarante. Los clientes solo pueden suscribirse / darse de baja.

Para versiones de C # anteriores a 6.0, donde `EventName?.Invoke` no es compatible, es una buena práctica asignar el evento a una variable temporal antes de la invocación, como se muestra en el ejemplo, que garantiza la seguridad de subprocesos en los casos en que varios subprocesos ejecutan el mismo código. Si no lo hace, puede provocar una `NullReferenceException` en ciertos casos en los que varios subprocesos utilizan la misma instancia de objeto. En C # 6.0, el compilador emite un código similar al que se muestra en el ejemplo de código para C # 6.

## Declaración de evento estándar

Declaración de evento:

```
public event EventHandler<EventArgsT> EventName;
```

Declaración del manejador de eventos:

```
public void HandlerName(object sender, EventArgsT args) { /* Handler logic */ }
```

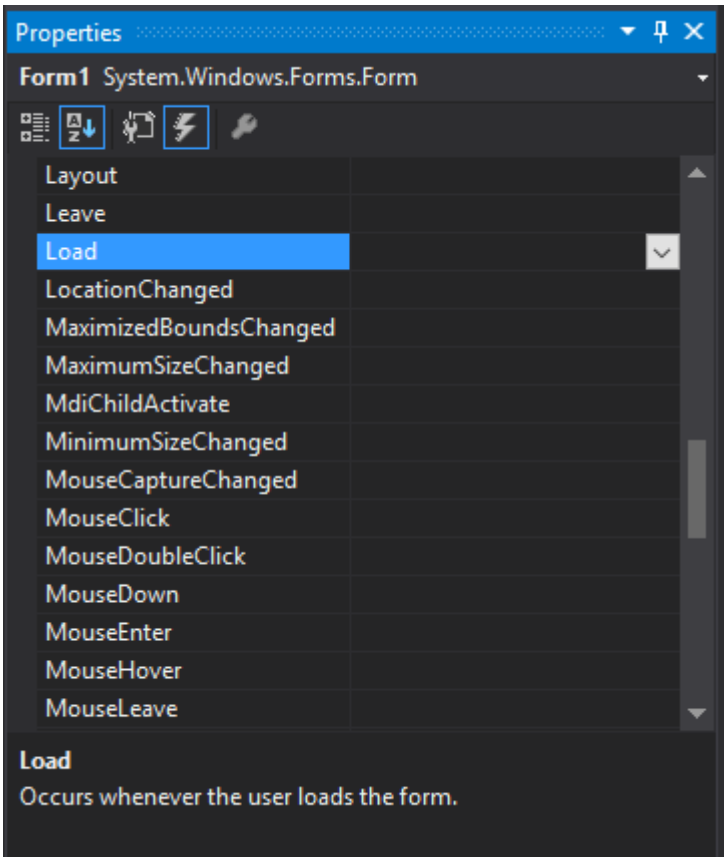
Suscribiéndose al evento:

*Dinamicamente:*

```
EventName += HandlerName;
```

*A través del Diseñador:*

1. Haga clic en el botón Eventos en la ventana de propiedades del control (Lightening bolt)
2. Haga doble clic en el nombre del evento:




3. Visual Studio generará el código del evento:

```
private void Form1_Load(object sender, EventArgs e)
{

}
```

Invocando el método:

```
EventName(SenderObject, EventArguments);
```

## Declaración de manejador de eventos anónimos

Declaración de evento:

```
public event EventHandler<EventArgsType> EventName;
```

Declaración del manejador de eventos utilizando el operador lambda => y suscribiéndose al evento:

```
EventName += (obj, eventArgs) => { /* Handler logic */ };
```

Declaración del controlador de eventos mediante la sintaxis de métodos anónimos delegados :

```
EventName += delegate(object obj, EventArgsType eventArgs) { /* Handler Logic */ };
```

Declaración y suscripción de un controlador de eventos que no usa el parámetro del evento, por lo que puede usar la sintaxis anterior sin necesidad de especificar parámetros:

```
EventName += delegate { /* Handler Logic */ }
```

Invocando el evento:

```
EventName?.Invoke(SenderObject, EventArguments);
```

## Declaración de evento no estándar

Los eventos pueden ser de cualquier tipo de delegado, no solo `EventHandler` y `EventHandler<T>` . Por ejemplo:

```
//Declaring an event
public event Action<Param1Type, Param2Type, ...> EventName;
```

Esto se usa de manera similar a los eventos de `EventHandler` estándar:

```
//Adding a named event handler
public void HandlerName(Param1Type parameter1, Param2Type parameter2, ...) {
    /* Handler logic */
}
EventName += HandlerName;

//Adding an anonymous event handler
EventName += (parameter1, parameter2, ...) => { /* Handler Logic */ };

//Invoking the event
EventName(parameter1, parameter2, ...);
```

---

Es posible declarar varios eventos del mismo tipo en una sola declaración, similar a los campos y las variables locales (aunque esto puede ser una mala idea):

```
public event EventHandler Event1, Event2, Event3;
```

Esto declara tres eventos separados ( `Event1` , `Event2` y `Event3` ) todos de tipo `EventHandler` .
*Nota: aunque algunos compiladores pueden aceptar esta sintaxis en las interfaces y en las clases, la especificación C # (v5.0 §13.2.3) proporciona una gramática para las interfaces que no lo permiten, por lo que su uso en las interfaces puede ser poco confiable con diferentes compiladores.*

## Creación de EventArgs personalizados que contienen datos adicionales

Los eventos personalizados generalmente necesitan argumentos de eventos personalizados que contengan información sobre el evento. Por ejemplo, `MouseEventArgs` utilizado por eventos del mouse como `MouseDown` o `MouseUp` , contiene información sobre la `Location` o los `Buttons` que se utilizan para generar el evento.

Al crear nuevos eventos, para crear un evento personalizado arg:

- Cree una clase derivada de `EventArgs` y defina las propiedades para los datos necesarios.
- Como convención, el nombre de la clase debe terminar con `EventArgs` .

**Ejemplo**

En el siguiente ejemplo, creamos un evento `PriceChangingEventArgs` para la propiedad `Price` de una clase. La clase de datos de evento contiene un `CurrentPrice` y un `NewPrice` . El evento aumenta cuando asigna un nuevo valor a la propiedad `Price` y le informa al consumidor que el valor está cambiando y le informa sobre el precio actual y el nuevo precio:

*PriceChangingEventArgs*

```
public class PriceChangingEventArgs : EventArgs
{
    public PriceChangingEventArgs(int currentPrice, int newPrice)
    {
        this.CurrentPrice = currentPrice;
        this.NewPrice = newPrice;
    }

    public int CurrentPrice { get; private set; }
    public int NewPrice { get; private set; }
}
```

*Producto*

```
public class Product
{
    public event EventHandler<PriceChangingEventArgs> PriceChanging;

    int price;
    public int Price
    {
        get { return price; }
        set
        {
            var e = new PriceChangingEventArgs(price, value);
            OnPriceChanging(e);
            price = value;
        }
    }

    protected void OnPriceChanging(PriceChangingEventArgs e)
    {
        var handler = PriceChanging;
        if (handler != null)
            handler(this, e);
    }
```

```
}
```

Puede mejorar el ejemplo permitiendo que el consumidor cambie el nuevo valor y luego el valor se usará para la propiedad. Para hacerlo es suficiente aplicar estos cambios en las clases.

Cambie la definición de `NewPrice` para ser configurable:

```
public int NewPrice { get; set; }
```

Cambie la definición de `Price` para usar `e.NewPrice` como valor de propiedad, después de llamar a `OnPriceChanging` :

```
int price;
public int Price
{
    get { return price; }
    set
    {
        var e = new PriceChangingEventArgs(price, value);
        OnPriceChanging(e);
        price = e.NewPrice;
    }
}
```

## Creando evento cancelable

Un evento cancelable puede ser generado por una clase cuando está a punto de realizar una acción que puede cancelarse, como el evento `FormClosing` de un `Form` .

Para crear tal evento:

- Cree un nuevo evento `CancelEventArgs` en `CancelEventArgs` y agregue propiedades adicionales para los datos del evento.
- Cree un evento usando `EventHandler<T>` y use la nueva clase arg de evento de cancelación que creó.

**Ejemplo**

En el siguiente ejemplo, creamos un evento `PriceChangingEventArgs` para la propiedad `Price` de una clase. La clase de datos del evento contiene un `Value` que le permite al consumidor conocer la nueva. El evento aumenta cuando asigna un nuevo valor a la propiedad `Price` y le informa al consumidor que el valor está cambiando y le permite cancelar el evento. Si el consumidor cancela el evento, se utilizará el valor anterior para `Price` :

*PriceChangingEventArgs*

```
public class PriceChangingEventArgs : CancelEventArgs
{
    int value;
    public int Value
    {
```

```
        get { return value; }
    }
    public PriceChangingEventArgs(int value)
    {
        this.value = value;
    }
}
```

*Producto*

```
public class Product
{
    int price;
    public int Price
    {
        get { return price; }
        set
        {
            var e = new PriceChangingEventArgs(value);
            OnPriceChanging(e);
            if (!e.Cancel)
                price = value;
        }
    }

    public event EventHandler<PriceChangingEventArgs> PropertyChanging;
    protected void OnPriceChanging(PriceChangingEventArgs e)
    {
        var handler = PropertyChanging;
        if (handler != null)
            PropertyChanging(this, e);
    }
}
```

## Propiedades del evento

Si una clase aumenta la cantidad de eventos, el costo de almacenamiento de un campo por delegado puede no ser aceptable. .NET Framework proporciona propiedades de eventos para estos casos. De esta manera, puede utilizar otra estructura de datos como `EventHandlerList` para almacenar delegados de eventos:

```
public class SampleClass
{
    // Define the delegate collection.
    protected EventHandlerList eventDelegates = new EventHandlerList();

    // Define a unique key for each event.
    static readonly object someEventKey = new object();

    // Define the SomeEvent event property.
    public event EventHandler SomeEvent
    {
        add
        {
            // Add the input delegate to the collection.
            eventDelegates.AddHandler(someEventKey, value);
        }
```

```
        remove
        {
            // Remove the input delegate from the collection.
            eventDelegates.RemoveHandler(someEventKey, value);
        }
    }

    // Raise the event with the delegate specified by someEventKey
    protected void OnSomeEvent(EventArgs e)
    {
        var handler = (EventHandler)eventDelegates[someEventKey];
        if (handler != null)
            handler(this, e);
    }
}
```

Este enfoque se usa ampliamente en marcos de GUI como WinForms, donde los controles pueden tener docenas e incluso cientos de eventos.

Tenga en cuenta que `EventHandlerList` no es seguro para subprocesos, por lo que si espera que su clase se use desde varios subprocesos, deberá agregar declaraciones de bloqueo u otro mecanismo de sincronización (o usar un almacenamiento que proporcione seguridad de subprocesos).

Lea Eventos en línea: https://riptutorial.com/es/csharp/topic/64/eventos

# Capítulo 65: Excepcion de referencia nula

## Examples

### NullReferenceException explicado

Se `NullReferenceException` una `NullReferenceException` cuando intenta acceder a un miembro no estático (propiedad, método, campo o evento) de un objeto de referencia pero es nulo.

```
Car myFirstCar = new Car();
Car mySecondCar = null;
Color myFirstColor = myFirstCar.Color; // No problem as myFirstCar exists / is not null
Color mySecondColor = mySecondCar.Color; // Throws a NullReferenceException
// as mySecondCar is null and yet we try to access its color.
```

Para depurar tal excepción, es bastante fácil: en la línea donde se lanza la excepción, solo hay que mirar antes de cada ' . 'o' [ ', o en raras ocasiones' ( '.

```
myGarage.CarCollection[currentIndex.Value].Color = theCarInTheStreet.Color;
```

¿De dónde viene mi excepción? Ya sea:

- `myGarage` es `null`
- `myGarage.CarCollection` es `null`
- `currentIndex` es `null`
- `myGarage.CarCollection[currentIndex.Value]` es `null`
- `theCarInTheStreet` es `null`

En el modo de depuración, solo tiene que colocar el cursor del mouse en cada uno de estos elementos y encontrará su referencia nula. Entonces, lo que tienes que hacer es entender por qué no tiene un valor. La corrección depende totalmente del objetivo de tu método.

¿Te has olvidado de instanciarlo / inicializarlo?

```
myGarage.CarCollection = new Car[10];
```

¿Se supone que debes hacer algo diferente si el objeto es nulo?

```
if (myGarage == null)
{
    Console.WriteLine("Maybe you should buy a garage first!");
}
```

O tal vez alguien te dio un argumento nulo, y se suponía que no:

```
if (theCarInTheStreet == null)
{
```

```
    throw new ArgumentNullException("theCarInTheStreet");
}
```

En cualquier caso, recuerde que un método nunca debe lanzar una NullReferenceException. Si lo hace, eso significa que has olvidado revisar algo.

Lea Excepcion de referencia nula en línea: https://riptutorial.com/es/csharp/topic/2702/excepcion-de-referencia-nula

# Capítulo 66: Expresiones lambda

## Observaciones

Una expresión lambda es una sintaxis para crear funciones anónimas en línea. Más formalmente, de la Guía de programación de C # :

> Una expresión lambda es una función anónima que puede usar para crear delegados o tipos de árbol de expresiones. Al utilizar expresiones lambda, puede escribir funciones locales que pueden pasarse como argumentos o devolverse como el valor de las llamadas a funciones.

Una expresión lambda se crea utilizando el operador `=>` . Ponga los parámetros en el lado izquierdo del operador. En el lado derecho, ponga una expresión que pueda usar esos parámetros; esta expresión se resolverá como el valor de retorno de la función. Más raramente, si es necesario, se puede usar un `{code block}` completo en el lado derecho. Si el tipo de retorno no es nulo, el bloque contendrá una declaración de retorno.

## Examples

### Pasar una expresión Lambda como un parámetro a un método

```
List<int> l2 = l1.FindAll(x => x > 6);
```

Aquí `x => x > 6` es una expresión lambda que actúa como un predicado que se asegura de que solo se devuelvan los elementos por encima de 6.

### Expresiones de Lambda como taquigrafía para la inicialización de delegados

```
public delegate int ModifyInt(int input);
ModifyInt multiplyByTwo = x => x * 2;
```

La sintaxis de la expresión Lambda anterior es equivalente al siguiente código detallado:

```
public delegate int ModifyInt(int input);

ModifyInt multiplyByTwo = delegate(int x){
    return x * 2;
};
```

### Lambdas tanto para `Func` como para` Action`

Normalmente, las lambdas se usan para definir *funciones* simples (generalmente en el contexto de una expresión linq):

```
var incremented = myEnumerable.Select(x => x + 1);
```

Aquí el `return` es implícito.

Sin embargo, también es posible pasar *acciones* como lambdas:

```
myObservable.Do(x => Console.WriteLine(x));
```

## Expresiones de Lambda con múltiples parámetros o sin parámetros

Use paréntesis alrededor de la expresión a la izquierda del operador `=>` para indicar múltiples parámetros.

```
delegate int ModifyInt(int input1, int input2);
ModifyInt multiplyTwoInts = (x,y) => x * y;
```

De manera similar, un conjunto vacío de paréntesis indica que la función no acepta parámetros.

```
delegate string ReturnString();
ReturnString getGreeting = () => "Hello world.";
```

## Poner múltiples declaraciones en una declaración Lambda

A diferencia de una expresión lambda, una declaración lambda puede contener varias declaraciones separadas por punto y coma.

```
delegate void ModifyInt(int input);

ModifyInt addOneAndTellMe = x =>
{
    int result = x + 1;
    Console.WriteLine(result);
};
```

Tenga en cuenta que las declaraciones están entre llaves `{}` .

Recuerde que la declaración lambdas no se puede utilizar para crear árboles de expresión.

## Lambdas se puede emitir como `Func` y` Expresión`

Suponiendo la siguiente clase de `Person` :

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
}
```

La siguiente lambda:

```
p => p.Age > 18
```

Se puede pasar como argumento a ambos métodos:

```
public void AsFunc(Func<Person, bool> func)
public void AsExpression(Expression<Func<Person, bool>> expr)
```

Porque el compilador es capaz de transformar lambdas tanto a delegados como a `Expression` s.

Obviamente, los proveedores de LINQ dependen en gran medida de `Expression` s (expuesto principalmente a través de la interfaz `IQueryable<T>` ) para poder analizar las consultas y traducirlas a las consultas almacenadas.

## Expresión Lambda como un controlador de eventos

Las expresiones Lambda se pueden usar para manejar eventos, lo cual es útil cuando:

- El manejador es corto.
- El manejador nunca necesita ser cancelado.

A continuación se presenta una buena situación en la que se podría usar un controlador de eventos lambda:

```
smtpClient.SendCompleted += (sender, args) => Console.WriteLine("Email sent");
```

Si es necesario cancelar la suscripción de un controlador de eventos registrado en algún punto futuro del código, la expresión del controlador de eventos se debe guardar en una variable y el registro / anulación de registro a través de esa variable:

```
EventHandler handler = (sender, args) => Console.WriteLine("Email sent");

smtpClient.SendCompleted += handler;
smtpClient.SendCompleted -= handler;
```

La razón por la que se hace esto en lugar de simplemente volver a escribir la expresión lambda literalmente para cancelar la suscripción ( `-=` ) es que el compilador de C # no necesariamente considerará iguales a las dos expresiones:

```
EventHandler handlerA = (sender, args) => Console.WriteLine("Email sent");
EventHandler handlerB = (sender, args) => Console.WriteLine("Email sent");
Console.WriteLine(handlerA.Equals(handlerB)); // May return "False"
```

Tenga en cuenta que si se agregan sentencias adicionales a la expresión lambda, entonces se pueden omitir accidentalmente las llaves circundantes requeridas, sin causar un error de tiempo de compilación. Por ejemplo:

```
smtpClient.SendCompleted += (sender, args) => Console.WriteLine("Email sent");
emailSendButton.Enabled = true;
```

Esto se compilará, pero dará como resultado la adición de la expresión lambda `(sender, args) => Console.WriteLine("Email sent");` como un controlador de eventos, y ejecutando la declaración

`emailSendButton.Enabled = true;` inmediatamente. Para solucionar esto, el contenido de la lambda debe estar rodeado de llaves. Esto se puede evitar usando llaves desde el principio, teniendo cuidado al agregar declaraciones adicionales a un controlador de eventos lambda, o rodeando la lambda entre paréntesis desde el principio:

```
smtpClient.SendCompleted += ((sender, args) => Console.WriteLine("Email sent"));
//Adding an extra statement will result in a compile-time error
```

Lea Expresiones lambda en línea: https://riptutorial.com/es/csharp/topic/46/expresiones-lambda

# Capítulo 67: Expresiones lambda

## Observaciones

## Cierres

Las expresiones Lambda capturarán implícitamente las variables utilizadas y crearán un cierre . Un cierre es una función junto con algún contexto de estado. El compilador generará un cierre cada vez que una expresión lambda "encierre" un valor de su contexto circundante.

Ej. Cuando se ejecuta lo siguiente

```
Func<object, bool> safeApplyFiltererPredicate = o => (o != null) && filterer.Predicate(i);
```

`safeApplyFilterPredicate` se refiere a un nuevo objeto creado que tiene una referencia privado al valor actual de `filterer` , y cuya `Invoke` método comporta como

```
o => (o != null) && filterer.Predicate(i);
```

Esto puede ser importante, ya que mientras se mantenga la referencia al valor ahora en `safeApplyFilterPredicate` , habrá una referencia al objeto al que se refiere actualmente el `filterer` . Esto tiene un efecto en la recolección de basura y puede causar un comportamiento inesperado si el objeto al que se refiere el `filterer` actualidad está mutado.

Por otro lado, los cierres se pueden usar para deliberar un efecto para encapsular un comportamiento que involucra referencias a otros objetos.

P.ej

```
var logger = new Logger();
Func<int, int> Add1AndLog = i => {
    logger.Log("adding 1 to " + i);
    return (i + 1);
};
```

Los cierres también se pueden utilizar para modelar máquinas de estado:

```
Func<int, int> MyAddingMachine() {
    var i = 0;
    return x => i += x;
};
```

## Examples

### Expresiones lambda basicas

```
Func<int, int> add1 = i => i + 1;

Func<int, int, int> add = (i, j) => i + j;

// Behaviourally equivalent to:

int Add1(int i)
{
    return i + 1;
}

int Add(int i, int j)
{
    return i + j;
}

...

Console.WriteLine(add1(42)); //43
Console.WriteLine(Add1(42)); //43
Console.WriteLine(add(100, 250)); //350
Console.WriteLine(Add(100, 250)); //350
```

## Expresiones lambda básicas con LINQ

```
// assume source is {0, 1, 2, ..., 10}

var evens = source.Where(n => n%2 == 0);
// evens = {0, 2, 4, ... 10}

var strings = source.Select(n => n.ToString());
// strings = {"0", "1", ..., "10"}
```

## Usando la sintaxis lambda para crear un cierre

Ver comentarios para la discusión de los cierres. Supongamos que tenemos una interfaz:

```
public interface IMachine<TState, TInput>
{
    TState State { get; }
    public void Input(TInput input);
}
```

y luego se ejecuta lo siguiente:

```
IMachine<int, int> machine = ...;
Func<int, int> machineClosure = i => {
    machine.Input(i);
    return machine.State;
};
```

Ahora `machineClosure` refiere a una función de `int` a `int` , que detrás de escena usa la instancia de `IMachine` que se refiere la `machine` para llevar a cabo el cálculo. Incluso si la `machine` referencia queda fuera del alcance, siempre que se `machineClosure` objeto `machineClosure` , la instancia original

de `IMachine` se mantendrá como parte de un 'cierre', definido automáticamente por el compilador.

Advertencia: esto puede significar que la misma llamada de función devuelve valores diferentes en momentos diferentes (por ejemplo, en este ejemplo si la máquina mantiene una suma de sus entradas). En muchos casos, esto puede ser inesperado y debe evitarse para cualquier código en un estilo funcional: los cierres accidentales e inesperados pueden ser una fuente de errores.

### Sintaxis Lambda con cuerpo de bloque de declaración

```
Func<int, string> doubleThenAddElevenThenQuote = i => {
    var doubled = 2 * i;
    var addedEleven = 11 + doubled;
    return $"'{addedEleven}'";
};
```

### Expresiones Lambda con System.Linq.Expressions

```
Expression<Func<int, bool>> checkEvenExpression = i => i%2 == 0;
// lambda expression is automatically converted to an Expression<Func<int, bool>>
```

Lea Expresiones lambda en línea: https://riptutorial.com/es/csharp/topic/7057/expresiones-lambda

# Capítulo 68: Extensiones reactivas (Rx)

## Examples

### Observando el evento TextChanged en un TextBox

Se crea un observable a partir del evento TextChanged del TextBox. Además, cualquier entrada solo se selecciona si es diferente de la última entrada y si no hubo entrada en 0,5 segundos. La salida en este ejemplo se envía a la consola.

```
Observable
     .FromEventPattern(textBoxInput, "TextChanged")
     .Select(s => ((TextBox) s.Sender).Text)
     .Throttle(TimeSpan.FromSeconds(0.5))
     .DistinctUntilChanged()
     .Subscribe(text => Console.WriteLine(text));
```

### Streaming de datos de la base de datos con observable

Supongamos que tiene un método que devuelve `IEnumerable<T>` , fe

```
private IEnumerable<T> GetData()
{
    try
    {
        // return results from database
    }
    catch(Exception exception)
    {
        throw;
    }
}
```

Crea un Observable e inicia un método de forma asíncrona. `SelectMany` aplana la colección y la suscripción se `SelectMany` cada 200 elementos a través de `Buffer` .

```
int bufferSize = 200;

Observable
    .Start(() => GetData())
    .SelectMany(s => s)
    .Buffer(bufferSize)
    .ObserveOn(SynchronizationContext.Current)
    .Subscribe(items =>
    {
        Console.WriteLine("Loaded {0} elements", items.Count);

        // do something on the UI like incrementing a ProgressBar
    },
    () => Console.WriteLine("Completed loading"));
```

Lea Extensiones reactivas (Rx) en línea: https://riptutorial.com/es/csharp/topic/5770/extensiones-reactivas--rx-

# Capítulo 69: FileSystemWatcher

## Sintaxis

- FileSystemWatcher público ()
- FileSystemWatcher público (ruta de cadena)
- FileSystemWatcher público (ruta de la cadena, filtro de cadena)

## Parámetros

| camino | filtrar |
| --- | --- |
| El directorio para monitorear, en notación estándar o Convención de nomenclatura universal (UNC). | El tipo de archivos para ver. Por ejemplo, "* .txt" busca cambios en todos los archivos de texto. |

## Examples

### FileWatcher básico

El siguiente ejemplo crea un `FileSystemWatcher` para ver el directorio especificado en el tiempo de ejecución. El componente está configurado para observar los cambios en el tiempo de **LastWrite** y **LastAccess** , la creación, eliminación o cambio de nombre de los archivos de texto en el directorio. Si un archivo se cambia, se crea o se elimina, la ruta al archivo se imprime en la consola. Cuando se cambia el nombre de un archivo, las rutas antiguas y nuevas se imprimen en la consola.

Utilice los espacios de nombres System.Diagnostics y System.IO para este ejemplo.

```
FileSystemWatcher watcher;

private void watch()
{
  // Create a new FileSystemWatcher and set its properties.
  watcher = new FileSystemWatcher();
  watcher.Path = path;

 /* Watch for changes in LastAccess and LastWrite times, and
       the renaming of files or directories. */
  watcher.NotifyFilter = NotifyFilters.LastAccess | NotifyFilters.LastWrite
                         | NotifyFilters.FileName | NotifyFilters.DirectoryName;

  // Only watch text files.
  watcher.Filter = "*.txt*";

  // Add event handler.
  watcher.Changed += new FileSystemEventHandler(OnChanged);
  // Begin watching.
```

```
  watcher.EnableRaisingEvents = true;
}

// Define the event handler.
private void OnChanged(object source, FileSystemEventArgs e)
{
  //Copies file to another directory or another action.
  Console.WriteLine("File: " +  e.FullPath + " " + e.ChangeType);
}
```

## IsFileReady

Un error común que mucha gente que comienza con FileSystemWatcher no tiene en cuenta que el evento FileWatcher se genera tan pronto como se crea el archivo. Sin embargo, puede llevar algo de tiempo completar el archivo.

*Ejemplo* :

Tome un tamaño de archivo de 1 GB, por ejemplo. El archivo apr pregunto creado por otro programa (Explorer.exe copiándolo desde algún lugar) pero tomará unos minutos completar este proceso. El evento se genera en el momento de la creación y debe esperar a que el archivo esté listo para copiarse.

Este es un método para verificar si el archivo está listo.

```
 public static bool IsFileReady(String sFilename)
{
    // If the file can be opened for exclusive access it means that the file
    // is no longer locked by another process.
    try
    {
        using (FileStream inputStream = File.Open(sFilename, FileMode.Open, FileAccess.Read,
FileShare.None))
        {
            if (inputStream.Length > 0)
            {
                return true;
            }
            else
            {
                return false;
            }

        }
    }
    catch (Exception)
    {
        return false;
    }
}
```

Lea FileSystemWatcher en línea: https://riptutorial.com/es/csharp/topic/5061/filesystemwatcher

# Capítulo 70: Filtros de acción

## Examples

### Filtros de acción personalizados

Escribimos filtros de acción personalizados por varias razones. Es posible que tengamos un filtro de acción personalizado para el registro o para guardar datos en la base de datos antes de cualquier ejecución de acción. También podríamos tener uno para obtener datos de la base de datos y establecerlos como los valores globales de la aplicación.

Para crear un filtro de acción personalizado, necesitamos realizar las siguientes tareas:

1. Crear una clase
2. Heredarlo de la clase ActionFilterAttribute

**Reemplace al menos uno de los siguientes métodos:**

**OnActionExecuting** : este método se llama antes de que se ejecute una acción del controlador.

**OnActionExecuted** : este método se llama después de ejecutar una acción del controlador.

**OnResultExecuting** : este método se llama antes de que se ejecute un resultado de acción del controlador.

**OnResultExecuted** : este método se llama después de que se ejecuta un resultado de acción del controlador.

**El filtro se puede crear como se muestra en el listado a continuación:**

```
using System;

using System.Diagnostics;

using System.Web.Mvc;



namespace WebApplication1
{

    public class MyFirstCustomFilter : ActionFilterAttribute
    {
        public override void OnResultExecuting(ResultExecutingContext filterContext)
        {
            //You may fetch data from database here
            filterContext.Controller.ViewBag.GreetMesssage = "Hello Foo";
            base.OnResultExecuting(filterContext);
        }

        public override void OnActionExecuting(ActionExecutingContext filterContext)
        {
```

```
                var controllerName = filterContext.RouteData.Values["controller"];
                var actionName = filterContext.RouteData.Values["action"];
                var message = String.Format("{0} controller:{1} action:{2}",
"onactionexecuting", controllerName, actionName);
                Debug.WriteLine(message, "Action Filter Log");
                base.OnActionExecuting(filterContext);
            }
        }
    }
```

Lea Filtros de acción en línea: https://riptutorial.com/es/csharp/topic/1505/filtros-de-accion

# Capítulo 71: Función con múltiples valores de retorno.

## Observaciones

No hay una respuesta inherente en C # a esta necesidad, así llamada. Sin embargo, hay soluciones para satisfacer esta necesidad.

La razón por la que califico la necesidad como "así llamada" es que solo necesitamos métodos con 2 o más de 2 valores para devolver cuando violamos los principios de buena programación. Especialmente el principio de responsabilidad única .

Por lo tanto, sería mejor recibir una alerta cuando necesitemos funciones que devuelvan 2 o más valores, y mejorar nuestro diseño.

## Examples

### Solución de "objeto anónimo" + "palabra clave dinámica"

Puedes devolver un objeto anónimo desde tu función

```
public static object FunctionWithUnknowReturnValues ()
{
    /// anonymous object
    return new { a = 1, b = 2 };
}
```

Y asigne el resultado a un objeto dinámico y lea los valores en él.

```
/// dynamic object
dynamic x = FunctionWithUnknowReturnValues();

Console.WriteLine(x.a);
Console.WriteLine(x.b);
```

### Solución de tupla

Puede devolver una instancia de la clase `Tuple` desde su función con dos parámetros de plantilla como `Tuple<string, MyClass>` :

```
public Tuple<string, MyClass> FunctionWith2ReturnValues ()
{
    return Tuple.Create("abc", new MyClass());
}
```

Y lee los valores como abajo:

```
Console.WriteLine(x.Item1);
Console.WriteLine(x.Item2);
```

## Parámetros de referencia y salida

La palabra clave `ref` se utiliza para pasar un argumento como referencia . `out` hará lo mismo que `ref` pero no requiere un valor asignado por la persona que llama antes de llamar a la función.

**Parámetro de referencia** : si desea pasar una variable como parámetro de referencia, debe inicializarla antes de pasarla como parámetro de referencia al método.

**Parámetro de salida: -** Si desea pasar una variable como parámetro de salida, no necesita inicializarla antes de pasarla como parámetro a método.

```
static void Main(string[] args)
{
    int a = 2;
    int b = 3;
    int add = 0;
    int mult= 0;
    AddOrMult(a, b, ref add, ref mult); //AddOrMult(a, b, out add, out mult);
    Console.WriteLine(add); //5
    Console.WriteLine(mult); //6
}

private static void AddOrMult(int a, int b, ref int add, ref int mult) //AddOrMult(int a, int
b, out int add, out int mult)
{
    add = a + b;
    mult = a * b;
}
```

Lea Función con múltiples valores de retorno. en línea:
https://riptutorial.com/es/csharp/topic/3908/funcion-con-multiples-valores-de-retorno-

# Capítulo 72: Funciones hash

## Observaciones

MD5 y SHA1 son inseguros y deben evitarse. Los ejemplos existen con fines educativos y debido al hecho de que el software heredado todavía puede utilizar estos algoritmos.

## Examples

### MD5

Las funciones hash asignan cadenas binarias de una longitud arbitraria a pequeñas cadenas binarias de una longitud fija.

El algoritmo `MD5` es una función hash ampliamente utilizada que produce un valor de hash de 128 bits (16 bytes, 32 caracteres hexadecimales).

El método `ComputeHash` de la clase `System.Security.Cryptography.MD5` devuelve el hash como una matriz de 16 bytes.

---

**Ejemplo:**

```
using System;
using System.Security.Cryptography;
using System.Text;

internal class Program
{
    private static void Main()
    {
        var source = "Hello World!";

        // Creates an instance of the default implementation of the MD5 hash algorithm.
        using (var md5Hash = MD5.Create())
        {
            // Byte array representation of source string
            var sourceBytes = Encoding.UTF8.GetBytes(source);

            // Generate hash value(Byte Array) for input data
            var hashBytes = md5Hash.ComputeHash(sourceBytes);

            // Convert hash byte array to string
            var hash = BitConverter.ToString(hashBytes).Replace("-", string.Empty);

            // Output the MD5 hash
            Console.WriteLine("The MD5 hash of " + source + " is: " + hash);
        }
    }
}
```

**Salida:** El hash MD5 de Hello World! es: ED076287532E86365E841E92BFC50D8C

---

***Temas de seguridad:***

Como la mayoría de las funciones hash, MD5 no es cifrado ni codificación. Puede ser revertido por un ataque de fuerza bruta y sufre de amplias vulnerabilidades contra los ataques de colisión y preimagen.

## SHA1

```
using System;
using System.Security.Cryptography;
using System.Text;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            string source = "Hello World!";
            using (SHA1 sha1Hash = SHA1.Create())
            {
                //From String to byte array
                byte[] sourceBytes = Encoding.UTF8.GetBytes(source);
                byte[] hashBytes = sha1Hash.ComputeHash(sourceBytes);
                string hash = BitConverter.ToString(hashBytes).Replace("-",String.Empty);

                Console.WriteLine("The SHA1 hash of " + source + " is: " + hash);
            }
        }
    }
 }
```

**Salida:**

El hash SHA1 de Hello Word! es: 2EF7BDE608CE5404E97D5F042F95F89F1C232871

## SHA256

```
using System;
using System.Security.Cryptography;
using System.Text;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            string source = "Hello World!";
            using (SHA256 sha256Hash = SHA256.Create())
            {
                //From String to byte array
                byte[] sourceBytes = Encoding.UTF8.GetBytes(source);
```

```
                byte[] hashBytes = sha256Hash.ComputeHash(sourceBytes);
                string hash = BitConverter.ToString(hashBytes).Replace("-", String.Empty);

                Console.WriteLine("The SHA256 hash of " + source + " is: " + hash);
            }
        }
    }
}
```

**Salida:**

El hash SHA256 de Hello World! es:
7F83B1657FF1FC53B92DC18148A1D65DFC2D4B1FA3D677284ADDD200126D9069

## SHA384

```
using System;
using System.Security.Cryptography;
using System.Text;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            string source = "Hello World!";
            using (SHA384 sha384Hash = SHA384.Create())
            {
                //From String to byte array
                byte[] sourceBytes = Encoding.UTF8.GetBytes(source);
                byte[] hashBytes = sha384Hash.ComputeHash(sourceBytes);
                string hash = BitConverter.ToString(hashBytes).Replace("-", String.Empty);

                Console.WriteLine("The SHA384 hash of " + source + " is: " + hash);
            }
        }
    }
}
```

**Salida:**

El hash SHA384 de Hello World! es:
BFD76C0EBBD006FEE583410547C1887B0292BE76D582D96C242D2A792723E3FD6FD061F9D5CFD

## SHA512

```
using System;
using System.Security.Cryptography;
using System.Text;

namespace ConsoleApplication1
{
    class Program
    {
```

```
        static void Main(string[] args)
        {
            string source = "Hello World!";
            using (SHA512 sha512Hash = SHA512.Create())
            {
                //From String to byte array
                byte[] sourceBytes = Encoding.UTF8.GetBytes(source);
                byte[] hashBytes = sha512Hash.ComputeHash(sourceBytes);
                string hash = BitConverter.ToString(hashBytes).Replace("-", String.Empty);

                Console.WriteLine("The SHA512 hash of " + source + " is: " + hash);
            }
        }
    }
}
```

**Salida:** El hash SHA512 de Hello World! es:
861844D6704E8573FEC34D967E20BCFEF3D424CF48BE04E6DC08F2BD58C729743371015EAD891C

## PBKDF2 para el hash de contraseña

**PBKDF2** ("Función de derivación de clave basada en contraseña 2") es una de las funciones hash recomendadas para el hashing de contraseña. Es parte de rfc-2898 .

`Rfc2898DeriveBytes` .NET se basa en HMACSHA1.

```
    using System.Security.Cryptography;

    ...

    public const int SALT_SIZE = 24; // size in bytes
    public const int HASH_SIZE = 24; // size in bytes
    public const int ITERATIONS = 100000; // number of pbkdf2 iterations

    public static byte[] CreateHash(string input)
    {
        // Generate a salt
        RNGCryptoServiceProvider provider = new RNGCryptoServiceProvider();
        byte[] salt = new byte[SALT_SIZE];
        provider.GetBytes(salt);

        // Generate the hash
        Rfc2898DeriveBytes pbkdf2 = new Rfc2898DeriveBytes(input, salt, ITERATIONS);
        return pbkdf2.GetBytes(HASH_SIZE);
    }
```

PBKDF2 requiere una sal y el número de iteraciones.

**Iteraciones:**

Un alto número de iteraciones ralentizará el algoritmo, lo que dificulta mucho el descifrado de contraseñas. Por lo tanto, se recomienda un alto número de iteraciones. PBKDF2 es un orden de magnitudes más lento que MD5, por ejemplo.

**Sal:**

Una sal evitará la búsqueda de valores de hash en tablas de arco iris. Debe almacenarse junto con el hash de contraseña. Se recomienda una sal por contraseña (no una sal global).

## Solución de hash de contraseña completa utilizando Pbkdf2

```
using System;
using System.Linq;
using System.Security.Cryptography;

namespace YourCryptoNamespace
{
    /// <summary>
    /// Salted password hashing with PBKDF2-SHA1.
  /// Compatibility: .NET 3.0 and later.
  /// </summary>
  /// <remarks>See http://crackstation.net/hashing-security.htm for much more on password
hashing.</remarks>
  public static class PasswordHashProvider
  {
    /// <summary>
    /// The salt byte size, 64 length ensures safety but could be increased / decreased
    /// </summary>
    private const int SaltByteSize = 64;
    /// <summary>
    /// The hash byte size,
    /// </summary>
    private const int HashByteSize = 64;
    /// <summary>
    ///  High iteration count is less likely to be cracked
    /// </summary>
    private const int Pbkdf2Iterations = 10000;

    /// <summary>
    /// Creates a salted PBKDF2 hash of the password.
    /// </summary>
    /// <remarks>
    /// The salt and the hash have to be persisted side by side for the password. They could
be persisted as bytes or as a string using the convenience methods in the next class to
convert from byte[] to string and later back again when executing password validation.
    /// </remarks>
    /// <param name="password">The password to hash.</param>
    /// <returns>The hash of the password.</returns>
    public static PasswordHashContainer CreateHash(string password)
    {
      // Generate a random salt
      using (var csprng = new RNGCryptoServiceProvider())
      {
        // create a unique salt for every password hash to prevent rainbow and dictionary
based attacks
        var salt = new byte[SaltByteSize];
        csprng.GetBytes(salt);

        // Hash the password and encode the parameters
        var hash = Pbkdf2(password, salt, Pbkdf2Iterations, HashByteSize);

        return new PasswordHashContainer(hash, salt);
      }
    }
    /// <summary>
```

```
    /// Recreates a password hash based on the incoming password string and the stored salt
    /// </summary>
    /// <param name="password">The password to check.</param>
    /// <param name="salt">The salt existing.</param>
    /// <returns>the generated hash based on the password and salt</returns>
    public static byte[] CreateHash(string password, byte[] salt)
    {
      // Extract the parameters from the hash
      return Pbkdf2(password, salt, Pbkdf2Iterations, HashByteSize);
    }

    /// <summary>
    /// Validates a password given a hash of the correct one.
    /// </summary>
    /// <param name="password">The password to check.</param>
    /// <param name="salt">The existing stored salt.</param>
    /// <param name="correctHash">The hash of the existing password.</param>
    /// <returns><c>true</c> if the password is correct. <c>false</c> otherwise. </returns>
    public static bool ValidatePassword(string password, byte[] salt, byte[] correctHash)
    {
      // Extract the parameters from the hash
      byte[] testHash = Pbkdf2(password, salt, Pbkdf2Iterations, HashByteSize);
      return CompareHashes(correctHash, testHash);
    }
    /// <summary>
    /// Compares two byte arrays (hashes)
    /// </summary>
    /// <param name="array1">The array1.</param>
    /// <param name="array2">The array2.</param>
    /// <returns><c>true</c> if they are the same, otherwise <c>false</c></returns>
    public static bool CompareHashes(byte[] array1, byte[] array2)
    {
      if (array1.Length != array2.Length) return false;
      return !array1.Where((t, i) => t != array2[i]).Any();
    }

    /// <summary>
    /// Computes the PBKDF2-SHA1 hash of a password.
    /// </summary>
    /// <param name="password">The password to hash.</param>
    /// <param name="salt">The salt.</param>
    /// <param name="iterations">The PBKDF2 iteration count.</param>
    /// <param name="outputBytes">The length of the hash to generate, in bytes.</param>
    /// <returns>A hash of the password.</returns>
    private static byte[] Pbkdf2(string password, byte[] salt, int iterations, int
outputBytes)
    {
      using (var pbkdf2 = new Rfc2898DeriveBytes(password, salt))
      {
        pbkdf2.IterationCount = iterations;
        return pbkdf2.GetBytes(outputBytes);
      }
    }
  }

  /// <summary>
  /// Container for password hash and salt and iterations.
  /// </summary>
  public sealed class PasswordHashContainer
  {
    /// <summary>
```

```
    /// Gets the hashed password.
    /// </summary>
    public byte[] HashedPassword { get; private set; }
    /// <summary>
    /// Gets the salt.
    /// </summary>
    public byte[] Salt { get; private set; }

    /// <summary>
    /// Initializes a new instance of the <see cref="PasswordHashContainer" /> class.
    /// </summary>
    /// <param name="hashedPassword">The hashed password.</param>
    /// <param name="salt">The salt.</param>
    public PasswordHashContainer(byte[] hashedPassword, byte[] salt)
    {
      this.HashedPassword = hashedPassword;
      this.Salt = salt;
    }
  }

  /// <summary>
  /// Convenience methods for converting between hex strings and byte array.
  /// </summary>
  public static class ByteConverter
  {
    /// <summary>
    /// Converts the hex representation string to an array of bytes
    /// </summary>
    /// <param name="hexedString">The hexed string.</param>
    /// <returns></returns>
    public static byte[] GetHexBytes(string hexedString)
    {
      var bytes = new byte[hexedString.Length / 2];
      for (var i = 0; i < bytes.Length; i++)
      {
        var strPos = i * 2;
        var chars = hexedString.Substring(strPos, 2);
        bytes[i] = Convert.ToByte(chars, 16);
      }
      return bytes;
    }
    /// <summary>
    /// Gets a hex string representation of the byte array passed in.
    /// </summary>
    /// <param name="bytes">The bytes.</param>
    public static string GetHexString(byte[] bytes)
    {
      return BitConverter.ToString(bytes).Replace("-", "").ToUpper();
    }
  }
}

/*
 * Password Hashing With PBKDF2 (http://crackstation.net/hashing-security.htm).
 * Copyright (c) 2013, Taylor Hornby
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are met:
 *
 * 1. Redistributions of source code must retain the above copyright notice,
```

```
* this list of conditions and the following disclaimer.
*
* 2. Redistributions in binary form must reproduce the above copyright notice,
* this list of conditions and the following disclaimer in the documentation
* and/or other materials provided with the distribution.
*
* THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
* AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
* ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE
* LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
* CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
* SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
* INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
* CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
* ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
* POSSIBILITY OF SUCH DAMAGE.
*/
```

---

Por favor, vea este excelente recurso Crackstation - Hashing de contraseña con sal - Hacerlo bien para obtener más información. Parte de esta solución (la función de hashing) se basó en el código de ese sitio.

Lea Funciones hash en línea: https://riptutorial.com/es/csharp/topic/2774/funciones-hash

# Capítulo 73: Fundación de comunicación de Windows

## Introducción

Windows Communication Foundation (WCF) es un marco para crear aplicaciones orientadas a servicios. Usando WCF, puede enviar datos como mensajes asíncronos de un punto final de servicio a otro. Un punto final de servicio puede ser parte de un servicio disponible continuamente alojado por IIS, o puede ser un servicio alojado en una aplicación. Los mensajes pueden ser tan simples como un solo carácter o palabra enviados como XML, o tan complejos como un flujo de datos binarios.

## Examples

### Muestra de inicio

El servicio describe las operaciones que realiza en un contrato de servicio que expone públicamente como metadatos.

```
// Define a service contract.
[ServiceContract(Namespace="http://StackOverflow.ServiceModel.Samples")]
public interface ICalculator
{
    [OperationContract]
    double Add(double n1, double n2);
}
```

La implementación del servicio calcula y devuelve el resultado apropiado, como se muestra en el siguiente código de ejemplo.

```
// Service class that implements the service contract.
public class CalculatorService : ICalculator
{
    public double Add(double n1, double n2)
    {
        return n1 + n2;
    }
}
```

El servicio expone un punto final para comunicarse con el servicio, definido mediante un archivo de configuración (Web.config), como se muestra en la siguiente configuración de muestra.

```
<services>
    <service
        name="StackOverflow.ServiceModel.Samples.CalculatorService"
        behaviorConfiguration="CalculatorServiceBehavior">
        <!-- ICalculator is exposed at the base address provided by
```

```
         host: http://localhost/servicemodelsamples/service.svc.  -->
       <endpoint address=""
             binding="wsHttpBinding"
             contract="StackOverflow.ServiceModel.Samples.ICalculator" />
       ...
    </service>
</services>
```

El marco no expone los metadatos de forma predeterminada. Como tal, el servicio activa ServiceMetadataBehavior y expone un punto final de intercambio de metadatos (MEX) en http://localhost/servicemodelsamples/service.svc/mex . La siguiente configuración demuestra esto.

```
<system.serviceModel>
  <services>
    <service
        name="StackOverflow.ServiceModel.Samples.CalculatorService"
        behaviorConfiguration="CalculatorServiceBehavior">
      ...
      <!-- the mex endpoint is explosed at
       http://localhost/servicemodelsamples/service.svc/mex -->
      <endpoint address="mex"
                binding="mexHttpBinding"
                contract="IMetadataExchange" />
    </service>
  </services>

  <!--For debugging purposes set the includeExceptionDetailInFaults
   attribute to true-->
  <behaviors>
    <serviceBehaviors>
      <behavior name="CalculatorServiceBehavior">
        <serviceMetadata httpGetEnabled="True"/>
        <serviceDebug includeExceptionDetailInFaults="False" />
      </behavior>
    </serviceBehaviors>
  </behaviors>
</system.serviceModel>
```

El cliente se comunica mediante un tipo de contrato determinado mediante una clase de cliente generada por la herramienta de utilidad de metadatos de ServiceModel (Svcutil.exe).

Ejecute el siguiente comando desde el indicador de comandos del SDK en el directorio del cliente para generar el proxy escrito:

```
svcutil.exe /n:"http://StackOverflow.ServiceModel.Samples,StackOverflow.ServiceModel.Samples"
http://localhost/servicemodelsamples/service.svc/mex /out:generatedClient.cs
```

Al igual que el servicio, el cliente utiliza un archivo de configuración (App.config) para especificar el punto final con el que desea comunicarse. La configuración del punto final del cliente consiste en una dirección absoluta para el punto final del servicio, el enlace y el contrato, como se muestra en el siguiente ejemplo.

```
<client>
     <endpoint
         address="http://localhost/servicemodelsamples/service.svc"
```

```
        binding="wsHttpBinding"
        contract="StackOverflow.ServiceModel.Samples.ICalculator" />
</client>
```

La implementación del cliente crea una instancia del cliente y utiliza la interfaz escrita para comenzar a comunicarse con el servicio, como se muestra en el siguiente código de ejemplo.

```
// Create a client.
CalculatorClient client = new CalculatorClient();

// Call the Add service operation.
double value1 = 100.00D;
double value2 = 15.99D;
double result = client.Add(value1, value2);
Console.WriteLine("Add({0},{1}) = {2}", value1, value2, result);

//Closing the client releases all communication resources.
client.Close();
```

Lea Fundación de comunicación de Windows en línea:
https://riptutorial.com/es/csharp/topic/10760/fundacion-de-comunicacion-de-windows

# Capítulo 74: Fundición

## Observaciones

*Casting* no es lo mismo que *convertir* . Es posible convertir el valor de cadena `"-1"` en un valor entero ( `-1` ), pero esto debe hacerse a través de métodos de biblioteca como `Convert.ToInt32()` o `Int32.Parse()` . No se puede hacer usando la sintaxis de casting directamente.

## Examples

### Lanzar un objeto a un tipo de base

Dadas las siguientes definiciones:

```
public interface IMyInterface1
{
    string GetName();
}

public interface IMyInterface2
{
    string GetName();
}

public class MyClass : IMyInterface1, IMyInterface2
{
    string IMyInterface1.GetName()
    {
        return "IMyInterface1";
    }

    string IMyInterface2.GetName()
    {
        return "IMyInterface2";
    }
}
```

Casting de un objeto a un ejemplo de tipo base:

```
    MyClass obj = new MyClass();

    IMyInterface1 myClass1 = (IMyInterface1)obj;
    IMyInterface2 myClass2 = (IMyInterface2)obj;

    Console.WriteLine("I am : {0}", myClass1.GetName());
    Console.WriteLine("I am : {0}", myClass2.GetName());

    // Outputs :
    // I am : IMyInterface1
    // I am : IMyInterface2
```

## Casting explícito

Si sabe que un valor es de un tipo específico, puede convertirlo explícitamente en ese tipo para usarlo en un contexto donde ese tipo sea necesario.

```
object value = -1;
int number = (int) value;
Console.WriteLine(Math.Abs(number));
```

Si intentamos pasar un `value` directamente a `Math.Abs()` , obtendríamos una excepción en tiempo de compilación porque `Math.Abs()` no tiene una sobrecarga que toma un `object` como parámetro.

Si el `value` no se puede convertir a un `int` , entonces la segunda línea en este ejemplo lanzaría una `InvalidCastException`

## Casting explícito seguro (operador `as`)

Si no está seguro de si un valor es del tipo que cree que es, puede emitirlo de forma segura utilizando el operador `as` . Si el valor no es de ese tipo, el valor resultante será `null` .

```
object value = "-1";
int? number = value as int?;
if(number != null)
{
    Console.WriteLine(Math.Abs(number.Value));
}
```

Tenga en cuenta que `null` valores `null` no tienen tipo, por lo que la palabra clave `as` dará `as` resultado un `null` cuando se emita cualquier valor `null` .

## Casting implícito

Un valor se convertirá automáticamente al tipo apropiado si el compilador sabe que siempre se puede convertir a ese tipo.

```
int number = -1;
object value = number;
Console.WriteLine(value);
```

En este ejemplo, no necesitamos utilizar la sintaxis de conversión explícita típica porque el compilador sabe que todas las `int` se pueden convertir a los `object` s. De hecho, podríamos evitar crear variables y pasar `-1` directamente como el argumento de `Console.WriteLine()` que espera un `object` .

```
Console.WriteLine(-1);
```

## Comprobación de compatibilidad sin colada.

Si necesita saber si el tipo de un valor se extiende o implementa un tipo dado, pero no quiere

convertirlo en ese tipo, puede usar el operador `is` .

```
if(value is int)
{
   Console.WriteLine(value + "is an int");
}
```

## Conversiones numéricas explícitas

Los operadores de conversión explícita se pueden utilizar para realizar conversiones de tipos numéricos, aunque no se extiendan o implementen entre sí.

```
double value = -1.1;
int number = (int) value;
```

Tenga en cuenta que en los casos en que el tipo de destino tenga menos precisión que el tipo original, la precisión se perderá. Por ejemplo, `-1.1` como valor doble en el ejemplo anterior se convierte en `-1` como valor entero.

Además, las conversiones numéricas dependen de los tipos de tiempo de compilación, por lo que no funcionarán si los tipos numéricos se han "encajonado" en objetos.

```
object value = -1.1;
int number = (int) value; // throws InvalidCastException
```

## Operadores de conversión

En C #, los tipos pueden definir *operadores de conversión* personalizados, que permiten que los valores se conviertan ay desde otros tipos utilizando *conversiones* explícitas o implícitas. Por ejemplo, considere una clase que pretende representar una expresión de JavaScript:

```
public class JsExpression
{
    private readonly string expression;
    public JsExpression(string rawExpression)
    {
        this.expression = rawExpression;
    }
    public override string ToString()
    {
        return this.expression;
    }
    public JsExpression IsEqualTo(JsExpression other)
    {
        return new JsExpression("(" + this + " == " + other + ")");
    }
}
```

Si quisiéramos crear una JsExpression que representara una comparación de dos valores de JavaScript, podríamos hacer algo como esto:

```
JsExpression intExpression = new JsExpression("-1");
JsExpression doubleExpression = new JsExpression("-1.0");
Console.WriteLine(intExpression.IsEqualTo(doubleExpression)); // (-1 == -1.0)
```

Pero podemos agregar algunos *operadores de conversión explícitos* a `JsExpression` , para permitir una conversión simple cuando se usa la conversión explícita.

```
public static explicit operator JsExpression(int value)
{
    return new JsExpression(value.ToString());
}
public static explicit operator JsExpression(double value)
{
    return new JsExpression(value.ToString());
}

// Usage:
JsExpression intExpression = (JsExpression)(-1);
JsExpression doubleExpression = (JsExpression)(-1.0);
Console.WriteLine(intExpression.IsEqualTo(doubleExpression)); // (-1 == -1.0)
```

O bien, podríamos cambiar estos operadores por *implícitos* para hacer la sintaxis mucho más simple.

```
public static implicit operator JsExpression(int value)
{
    return new JsExpression(value.ToString());
}
public static implicit operator JsExpression(double value)
{
    return new JsExpression(value.ToString());
}

// Usage:
JsExpression intExpression = -1;
Console.WriteLine(intExpression.IsEqualTo(-1.0)); // (-1 == -1.0)
```

## Operaciones de fundición LINQ

Supongamos que tienes tipos como los siguientes:

```
interface IThing {  }
class Thing : IThing {  }
```

LINQ le permite crear una proyección que cambia el tipo genérico en tiempo de compilación de un `IEnumerable<>` través de los métodos de extensión `Enumerable.Cast<>()` y `Enumerable.OfType<>()` .

```
IEnumerable<IThing> things = new IThing[] {new Thing()};
IEnumerable<Thing> things2 = things.Cast<Thing>();
IEnumerable<Thing> things3 = things.OfType<Thing>();
```

Cuando se evalúa `things2` , el método `Cast<>()` intentará convertir todos los valores de las `things` en `Thing` s. Si encuentra un valor que no se puede convertir, se lanzará una `InvalidCastException` .

Cuando se evalúa `things3` , el `OfType<>()` hará lo mismo, excepto que si encuentra un valor que no se puede convertir, simplemente omitirá ese valor en lugar de lanzar una excepción.

Debido al tipo genérico de estos métodos, no pueden invocar operadores de conversión o realizar conversiones numéricas.

```
double[] doubles = new[]{1,2,3}.Cast<double>().ToArray(); // Throws InvalidCastException
```

Simplemente puede realizar una conversión dentro de un `.Select()` como solución alternativa:

```
double[] doubles = new[]{1,2,3}.Select(i => (double)i).ToArray();
```

Lea Fundición en línea: https://riptutorial.com/es/csharp/topic/2690/fundicion

# Capítulo 75: Generación de Código T4

## Sintaxis

- **Sintaxis de T4**
- `<#@...#>` // Declarar propiedades que incluyen plantillas, ensamblajes y espacios de nombres y el idioma que usa la plantilla
- `Plain Text` formato // Declaración de texto que se puede recorrer en bucle para los archivos generados
- `<#=...#>` // Declarar scripts
- `<#+...#>` // Declarar scriptlets
- `<#...#>` // Declarar bloques de texto

## Examples

### Generación de código de tiempo de ejecución

```
<#@ template language="C#" #> //Language of your project
<#@ assembly name="System.Core" #>
<#@ import namespace="System.Linq" #>
<#@ import namespace="System.Text" #>
<#@ import namespace="System.Collections.Generic" #>
```

Lea Generación de Código T4 en línea: https://riptutorial.com/es/csharp/topic/4824/generacion-de-codigo-t4

# Capítulo 76: Generador de consultas Lambda genérico

## Observaciones

La clase se llama `ExpressionBuilder` . Tiene tres propiedades:

```
 private static readonly MethodInfo ContainsMethod = typeof(string).GetMethod("Contains",
new[] { typeof(string) });
 private static readonly MethodInfo StartsWithMethod = typeof(string).GetMethod("StartsWith",
new[] { typeof(string) });
 private static readonly MethodInfo EndsWithMethod = typeof(string).GetMethod("EndsWith",
new[] { typeof(string) });
```

Un método público `GetExpression` que devuelve la expresión lambda y tres métodos privados:

- `Expression GetExpression<T>`
- `BinaryExpression GetExpression<T>`
- `ConstantExpression GetConstant`

Todos los métodos se explican en detalle en los ejemplos.

## Examples

### Clase QueryFilter

Esta clase mantiene valores de filtros de predicado.

```
public class QueryFilter
{
    public string PropertyName { get; set; }
    public string Value { get; set; }
    public Operator Operator { get; set; }

    // In the query {a => a.Name.Equals("Pedro")}
    // Property name to filter - propertyName = "Name"
    // Filter value - value = "Pedro"
    // Operation to perform - operation = enum Operator.Equals
    public QueryFilter(string propertyName, string value, Operator operatorValue)
    {
        PropertyName = propertyName;
        Value = value;
        Operator = operatorValue;
    }
}
```

Enumerar para mantener los valores de las operaciones:

```
 public enum Operator
```

```
{
    Contains,
    GreaterThan,
    GreaterThanOrEqual,
    LessThan,
    LessThanOrEqualTo,
    StartsWith,
    EndsWith,
    Equals,
    NotEqual
}
```

## Método GetExpression

```
public static Expression<Func<T, bool>> GetExpression<T>(IList<QueryFilter> filters)
{
    Expression exp = null;

    // Represents a named parameter expression. {parm => parm.Name.Equals()}, it is the param
part
    // To create a ParameterExpression need the type of the entity that the query is against
an a name
    // The type is possible to find with the generic T and the name is fixed parm
    ParameterExpression param = Expression.Parameter(typeof(T), "parm");

    // It is good parctice never trust in the client, so it is wise to validate.
    if (filters.Count == 0)
        return null;

    // The expression creation differ if there is one, two or more filters.
    if (filters.Count != 1)
    {
        if (filters.Count == 2)
            // It is result from direct call.
            // For simplicity sake the private overloads will be explained in another example.
            exp = GetExpression<T>(param, filters[0], filters[1]);
        else
        {
            // As there is no method for more than two filters,
            // I iterate through all the filters and put I in the query two at a time
            while (filters.Count > 0)
            {
                // Retreive the first two filters
                var f1 = filters[0];
                var f2 = filters[1];

                // To build a expression with a conditional AND operation that evaluates
                // the second operand only if the first operand evaluates to true.
                // It needed to use the BinaryExpression a Expression derived class
                // That has the AndAlso method that join two expression together
                exp = exp == null ? GetExpression<T>(param, filters[0], filters[1]) :
Expression.AndAlso(exp, GetExpression<T>(param, filters[0], filters[1]));

                // Remove the two just used filters, for the method in the next iteration
finds the next filters
                filters.Remove(f1);
                filters.Remove(f2);

                // If it is that last filter, add the last one and remove it
```

```
                if (filters.Count == 1)
                {
                    exp = Expression.AndAlso(exp, GetExpression<T>(param, filters[0]));
                    filters.RemoveAt(0);
                }
            }
        }
    }
    else
        // It is result from direct call.
        exp = GetExpression<T>(param, filters[0]);

           // converts the Expression into Lambda and retuns the query
    return Expression.Lambda<Func<T, bool>>(exp, param);
}
```

### GetExpression sobrecarga privada

# Para un filtro:

Aquí es donde se crea la consulta, recibe un parámetro de expresión y un filtro.

```
private static Expression GetExpression<T>(ParameterExpression param, QueryFilter queryFilter)
{
    // Represents accessing a field or property, so here we are accessing for example:
    // the property "Name" of the entity
    MemberExpression member = Expression.Property(param, queryFilter.PropertyName);

    //Represents an expression that has a constant value, so here we are accessing for
example:
    // the values of the Property "Name".
    // Also for clarity sake the GetConstant will be explained in another example.
    ConstantExpression constant = GetConstant(member.Type, queryFilter.Value);

    // With these two, now I can build the expression
    // every operator has it one way to call, so the switch will do.
    switch (queryFilter.Operator)
    {
        case Operator.Equals:
            return Expression.Equal(member, constant);

        case Operator.Contains:
            return Expression.Call(member, ContainsMethod, constant);

        case Operator.GreaterThan:
            return Expression.GreaterThan(member, constant);

        case Operator.GreaterThanOrEqual:
            return Expression.GreaterThanOrEqual(member, constant);

        case Operator.LessThan:
            return Expression.LessThan(member, constant);

        case Operator.LessThanOrEqualTo:
            return Expression.LessThanOrEqual(member, constant);

        case Operator.StartsWith:
```

```
            return Expression.Call(member, StartsWithMethod, constant);

        case Operator.EndsWith:
            return Expression.Call(member, EndsWithMethod, constant);
    }

    return null;
}
```

# Para dos filtros:

Devuelve la instancia de BinaryExpresion en lugar de la expresión simple.

```
private static BinaryExpression GetExpression<T>(ParameterExpression param, QueryFilter
filter1, QueryFilter filter2)
{
    // Built two separated expression and join them after.
    Expression result1 = GetExpression<T>(param, filter1);
    Expression result2 = GetExpression<T>(param, filter2);
    return Expression.AndAlso(result1, result2);
}
```

## Método de expresión constante

`ConstantExpression` debe ser del mismo tipo que `MemberExpression` . El valor en este ejemplo es una cadena, que se convierte antes de crear la instancia de `ConstantExpression` .

```
private static ConstantExpression GetConstant(Type type, string value)
{
    // Discover the type, convert it, and create ConstantExpression
    ConstantExpression constant = null;
    if (type == typeof(int))
    {
        int num;
        int.TryParse(value, out num);
        constant = Expression.Constant(num);
    }
    else if(type == typeof(string))
    {
        constant = Expression.Constant(value);
    }
    else if (type == typeof(DateTime))
    {
        DateTime date;
        DateTime.TryParse(value, out date);
        constant = Expression.Constant(date);
    }
    else if (type == typeof(bool))
    {
        bool flag;
        if (bool.TryParse(value, out flag))
        {
            flag = true;
        }
        constant = Expression.Constant(flag);
```

```
    }
    else if (type == typeof(decimal))
    {
        decimal number;
        decimal.TryParse(value, out number);
        constant = Expression.Constant(number);
    }
    return constant;
}
```

### Uso

Filtros de colección = nueva Lista (); Filtro QueryFilter = nuevo QueryFilter ("Nombre", "Burger", Operator.StartsWith); filtros. añadir (filtro);

```
 Expression<Func<Food, bool>> query = ExpressionBuilder.GetExpression<Food>(filters);
```

En este caso, es una consulta en contra de la entidad de Alimentos, que desea encontrar todos los alimentos que comienzan con "Hamburguesa" en el nombre.

# Salida:

```
query = {parm => a.parm.StartsWith("Burger")}


Expression<Func<T, bool>> GetExpression<T>(IList<QueryFilter> filters)
```

Lea Generador de consultas Lambda genérico en línea:
https://riptutorial.com/es/csharp/topic/6721/generador-de-consultas-lambda-generico

# Capítulo 77: Generando números aleatorios en C #

## Sintaxis

- Aleatorio()
- Aleatorio (int Seed)
- int siguiente ()
- int Next (int maxValue)
- int Next (int minValue, int maxValue)

## Parámetros

| Parámetros | Detalles |
|---|---|
| Semilla | Un valor para generar números aleatorios. Si no se establece, el valor predeterminado está determinado por la hora actual del sistema. |
| minValue | Los números generados no serán más pequeños que este valor. Si no se establece, el valor predeterminado es 0. |
| valor máximo | Los números generados serán más pequeños que este valor. Si no se establece, el valor predeterminado es `Int32.MaxValue` . |
| valor de retorno | Devuelve un número con valor aleatorio. |

## Observaciones

La semilla aleatoria generada por el sistema no es la misma en cada ejecución diferente.

Las semillas generadas al mismo tiempo pueden ser las mismas.

## Examples

### Generar un int al azar

Este ejemplo genera valores aleatorios entre 0 y 2147483647.

```
Random rnd = new Random();
int randomNumber = rnd.Next();
```

## Generar un doble aleatorio

Genera un número aleatorio entre 0 y 1.0. (sin incluir 1.0)

```
Random rnd = new Random();
var randomDouble = rnd.NextDouble();
```

## Generar un int aleatorio en un rango dado

Genere un número aleatorio entre `minValue` y `maxValue - 1` .

```
Random rnd = new Random();
var randomBetween10And20 = rnd.Next(10, 20);
```

## Generando la misma secuencia de números aleatorios una y otra vez

Al crear instancias `Random` con la misma semilla, se generarán los mismos números.

```
int seed = 5;
for (int i = 0; i < 2; i++)
{
   Console.WriteLine("Random instance " + i);
   Random rnd = new Random(seed);
   for (int j = 0; j < 5; j++)
   {
      Console.Write(rnd.Next());
      Console.Write(" ");
   }

   Console.WriteLine();
}
```

Salida:

```
Random instance 0
726643700 610783965 564707973 1342984399 995276750
Random instance 1
726643700 610783965 564707973 1342984399 995276750
```

## Crea múltiples clases aleatorias con diferentes semillas simultáneamente.

Dos clases aleatorias creadas al mismo tiempo tendrán el mismo valor semilla.

Usando `System.Guid.NewGuid().GetHashCode()` puede obtener una semilla diferente incluso al mismo tiempo.

```
Random rnd1 = new Random();
```

```
Random rnd2 = new Random();
Console.WriteLine("First 5 random number in rnd1");
for (int i = 0; i < 5; i++)
    Console.WriteLine(rnd1.Next());

Console.WriteLine("First 5 random number in rnd2");
for (int i = 0; i < 5; i++)
    Console.WriteLine(rnd2.Next());

rnd1 = new Random(Guid.NewGuid().GetHashCode());
rnd2 = new Random(Guid.NewGuid().GetHashCode());
Console.WriteLine("First 5 random number in rnd1 using Guid");
for (int i = 0; i < 5; i++)
    Console.WriteLine(rnd1.Next());
Console.WriteLine("First 5 random number in rnd2 using Guid");
for (int i = 0; i < 5; i++)
    Console.WriteLine(rnd2.Next());
```

Otra forma de lograr diferentes semillas es usar otra instancia `Random` para recuperar los valores de semillas.

```
Random rndSeeds = new Random();
Random rnd1 = new Random(rndSeeds.Next());
Random rnd2 = new Random(rndSeeds.Next());
```

Esto también hace posible controlar el resultado de todas las instancias `Random` al establecer solo el valor semilla para `rndSeeds` . Todas las demás instancias se derivarán determinísticamente de ese único valor semilla.

## Generar un carácter aleatorio.

Genere una letra aleatoria entre `a` y `z` usando la sobrecarga `Next()` para un rango de números dado, luego convierta el `int` resultante en un `char`

```
Random rnd = new Random();
char randomChar = (char)rnd.Next('a','z');
//'a' and 'z' are interpreted as ints for parameters for Next()
```

## Generar un número que sea un porcentaje de un valor máximo

Una necesidad común de números aleatorios es generar un número que sea `X%` de algún valor máximo. Esto se puede hacer tratando el resultado de `NextDouble()` como un porcentaje:

```
var rnd = new Random();
var maxValue = 5000;
var percentage = rnd.NextDouble();
var result = maxValue * percentage;
//suppose NextDouble() returns .65, result will hold 65% of 5000: 3250.
```

Lea Generando números aleatorios en C # en línea:
https://riptutorial.com/es/csharp/topic/1975/generando-numeros-aleatorios-en-c-sharp

# Capítulo 78: Genéricos

## Sintaxis

- `public void SomeMethod <T> () { }`
- `public void SomeMethod<T, V>() { }`
- `public T SomeMethod<T>(IEnumerable<T> sequence) { ... }`
- `public void SomeMethod<T>() where T : new() { }`
- `public void SomeMethod<T, V>() where T : new() where V : struct { }`
- `public void SomeMethod<T>() where T: IDisposable { }`
- `public void SomeMethod<T>() where T: Foo { }`
- `public class MyClass<T> { public T Data {get; set; } }`

## Parámetros

| Parámetro (s) | Descripción |
|---|---|
| TELEVISIÓN | Tipo de marcadores de posición para declaraciones genéricas |

## Observaciones

Los genéricos en C # son compatibles hasta el tiempo de ejecución: los tipos genéricos creados con C # tendrán su semántica genérica conservada incluso después de compilarse en CIL .

Esto significa efectivamente que, en C #, es posible reflexionar sobre tipos genéricos y verlos como fueron declarados o verificar si un objeto es una instancia de un tipo genérico, por ejemplo. Esto contrasta con el borrado de tipo , donde la información de tipo genérico se elimina durante la compilación. También está en contraste con el enfoque de la plantilla para los genéricos, donde varios tipos genéricos concretos se convierten en múltiples tipos no genéricos en tiempo de ejecución, y se pierde cualquier metadato requerido para crear una instancia más precisa de las definiciones de los tipos genéricos originales.

Sin embargo, tenga cuidado al reflexionar sobre los tipos genéricos: los nombres de los tipos genéricos se modificarán en la compilación, sustituyendo los corchetes angulares y los nombres de los parámetros de tipo por una comilla invertida seguida del número de parámetros de tipo genérico. Por lo tanto, un `Dictionary<TKey, Tvalue>` se traducirá al `Dictionary`2` .

## Examples

### Parámetros de tipo (clases)

Declaración:

```
class MyGenericClass<T1, T2, T3, ...>
{
```

```
    // Do something with the type parameters.
}
```

Inicialización:

```
var x = new MyGenericClass<int, char, bool>();
```

Uso (como el tipo de un parámetro):

```
void AnotherMethod(MyGenericClass<float, byte, char> arg) { ... }
```

## Tipo de parámetros (métodos)

Declaración:

```
void MyGenericMethod<T1, T2, T3>(T1 a, T2 b, T3 c)
{
    // Do something with the type parameters.
}
```

Invocación:

No es necesario proporcionar argumentos de tipo a un método genric, porque el compilador puede inferir implícitamente el tipo.

```
int x =10;
int y =20;
string z = "test";
MyGenericMethod(x,y,z);
```

Sin embargo, si existe una ambigüedad, los métodos genéricos deben llamarse con argumentos de tipo como

```
MyGenericMethod<int, int, string>(x,y,z);
```

## Parámetros de tipo (interfaces)

Declaración:

```
interface IMyGenericInterface<T1, T2, T3, ...> { ... }
```

Uso (en herencia):

```
class ClassA<T1, T2, T3> : IMyGenericInterface<T1, T2, T3> { ... }

class ClassB<T1, T2> : IMyGenericInterface<T1, T2, int> { ... }

class ClassC<T1> : IMyGenericInterface<T1, char, int> { ... }
```

```
class ClassD : IMyGenericInterface<bool, char, int> { ... }
```

Uso (como el tipo de un parámetro):

```
void SomeMethod(IMyGenericInterface<int, char, bool> arg) { ... }
```

## Inferencia de tipo implícita (métodos)

Cuando se pasan argumentos formales a un método genérico, los argumentos de tipo genérico relevantes se pueden inferir implícitamente. Si se puede inferir todo tipo genérico, especificarlo en la sintaxis es opcional.

Considere el siguiente método genérico. Tiene un parámetro formal y un parámetro de tipo genérico. Existe una relación muy obvia entre ellos: el tipo que se pasa como argumento al parámetro de tipo genérico debe ser el mismo que el tipo de tiempo de compilación del argumento que se pasa al parámetro formal.

```
void M<T>(T obj)
{
}
```

Estas dos llamadas son equivalentes:

```
M<object>(new object());
M(new object());
```

Estas dos llamadas también son equivalentes:

```
M<string>("");
M("");
```

Y así son estas tres llamadas:

```
M<object>("");
M((object) "");
M("" as object);
```

---

Observe que si al menos un argumento de tipo no se puede inferir, entonces todos deben especificarse.

Considere el siguiente método genérico. El primer argumento de tipo genérico es el mismo que el tipo del argumento formal. Pero no existe tal relación para el segundo argumento de tipo genérico. Por lo tanto, el compilador no tiene forma de inferir el segundo argumento de tipo genérico en ninguna llamada a este método.

```
void X<T1, T2>(T1 obj)
{
}
```

Esto ya no funciona:

```
X("");
```

Esto tampoco funciona, porque el compilador no está seguro de si estamos especificando el primer o el segundo parámetro genérico (ambos serían válidos como `object` ):

```
X<object>("");
```

Estamos obligados a escribir ambos, como este:

```
X<string, object>("");
```

## Tipo de restricciones (clases e interfaces)

Las restricciones de tipo pueden forzar un parámetro de tipo para implementar una determinada interfaz o clase.

```
interface IType;
interface IAnotherType;

// T must be a subtype of IType
interface IGeneric<T>
    where T : IType
{
}

// T must be a subtype of IType
class Generic<T>
    where T : IType
{
}

class NonGeneric
{
    // T must be a subtype of IType
    public void DoSomething<T>(T arg)
        where T : IType
    {
    }
}

// Valid definitions and expressions:
class Type : IType { }
class Sub : IGeneric<Type> { }
class Sub : Generic<Type> { }
new NonGeneric().DoSomething(new Type());

// Invalid definitions and expressions:
class AnotherType : IAnotherType { }
class Sub : IGeneric<AnotherType> { }
class Sub : Generic<AnotherType> { }
new NonGeneric().DoSomething(new AnotherType());
```

Sintaxis para múltiples restricciones:

```
class Generic<T, T1>
    where T : IType
    where T1 : Base, new()
{
}
```

Las restricciones de tipo funcionan de la misma manera que la herencia, ya que es posible especificar múltiples interfaces como restricciones en el tipo genérico, pero solo una clase:

```
class A { /* ... */ }
class B { /* ... */ }

interface I1 { }
interface I2 { }

class Generic<T>
    where T : A, I1, I2
{
}

class Generic2<T>
    where T : A, B //Compilation error
{
}
```

Otra regla es que la clase debe agregarse como la primera restricción y luego las interfaces:

```
class Generic<T>
    where T : A, I1
{
}

class Generic2<T>
    where T : I1, A //Compilation error
{
}
```

Todas las restricciones declaradas deben satisfacerse simultáneamente para que funcione una instanciación genérica particular. No hay manera de especificar dos o más conjuntos alternativos de restricciones.

## Tipo de restricciones (clase y estructura)

Es posible especificar si el argumento de tipo debe ser un tipo de referencia o un tipo de valor utilizando la `class` o `struct` restricciones respectivas. Si se usan estas restricciones, *deben* definirse *antes de que* se puedan enumerar *todas las* demás restricciones (por ejemplo, un tipo principal o `new()` ).

```
// TRef must be a reference type, the use of Int32, Single, etc. is invalid.
// Interfaces are valid, as they are reference types
class AcceptsRefType<TRef>
    where TRef : class
{
    // TStruct must be a value type.
```

```
    public void AcceptStruct<TStruct>()
        where TStruct : struct
    {
    }

    // If multiple constraints are used along with class/struct
    // then the class or struct constraint MUST be specified first
    public void Foo<TComparableClass>()
        where TComparableClass : class, IComparable
    {
    }
}
```

## Restricciones de tipo (nueva palabra clave)

Al usar la restricción `new()` , es posible imponer parámetros de tipo para definir un constructor vacío (predeterminado).

```
class Foo
{
    public Foo () { }
}

class Bar
{
    public Bar (string s) { ... }
}

class Factory<T>
    where T : new()
{
    public T Create()
    {
        return new T();
    }
}

Foo f = new Factory<Foo>().Create(); // Valid.
Bar b = new Factory<Bar>().Create(); // Invalid, Bar does not define a default/empty
constructor.
```

La segunda llamada a to `Create()` dará un error de tiempo de compilación con el siguiente mensaje:

> 'Barra' debe ser un tipo no abstracto con un constructor público sin parámetros para usarlo como parámetro 'T' en el tipo o método genérico 'Fábrica'

No hay ninguna restricción para un constructor con parámetros, solo se admiten constructores sin parámetros.

## Inferencia de tipos (clases)

Los desarrolladores pueden verse atrapados por el hecho de que la inferencia de tipos *no funciona* para los constructores:

```
class Tuple<T1,T2>
{
   public Tuple(T1 value1, T2 value2)
   {
   }
}

var x = new Tuple(2, "two");              // This WON'T work...
var y = new Tuple<int, string>(2, "two"); // even though the explicit form will.
```

La primera forma de crear una instancia sin especificar explícitamente los parámetros de tipo causará un error de tiempo de compilación que diría:

> El uso del tipo genérico 'Tuple <T1, T2>' requiere 2 argumentos de tipo

Una solución común es agregar un método auxiliar en una clase estática:

```
static class Tuple
{
    public static Tuple<T1, T2> Create<T1, T2>(T1 value1, T2 value2)
    {
         return new Tuple<T1, T2>(value1, value2);
    }
}

var x = Tuple.Create(2, "two");  // This WILL work...
```

## Reflexionando sobre los parámetros de tipo.

El operador `typeof` trabaja en parámetros de tipo.

```
class NameGetter<T>
{
    public string GetTypeName()
    {
        return typeof(T).Name;
    }
}
```

## Parámetros de tipo explícito

Hay diferentes casos en los que debe especificar explícitamente los parámetros de tipo para un método genérico. En los dos casos siguientes, el compilador no puede inferir todos los parámetros de tipo de los parámetros de método especificados.

Un caso es cuando no hay parámetros:

```
public void SomeMethod<T, V>()
{
   // No code for simplicity
}

SomeMethod(); // doesn't compile
```

```
SomeMethod<int, bool>(); // compiles
```

El segundo caso es cuando uno (o más) de los parámetros de tipo no es parte de los parámetros del método:

```
public K SomeMethod<K, V>(V input)
{
    return default(K);
}

int num1 = SomeMethod(3); // doesn't compile
int num2 = SomeMethod<int>("3"); // doesn't compile
int num3 = SomeMethod<int, string>("3"); // compiles.
```

## Utilizando método genérico con una interfaz como tipo de restricción.

Este es un ejemplo de cómo usar el tipo genérico TFood dentro del método Eat en la clase Animal

```
public interface IFood
{
    void EatenBy(Animal animal);
}

public class Grass: IFood
{
    public void EatenBy(Animal animal)
    {
        Console.WriteLine("Grass was eaten by: {0}", animal.Name);
    }
}

public class Animal
{
    public string Name { get; set; }

    public void Eat<TFood>(TFood food)
        where TFood : IFood
    {
        food.EatenBy(this);
    }
}

public class Carnivore : Animal
{
    public Carnivore()
    {
        Name = "Carnivore";
    }
}

public class Herbivore : Animal, IFood
{
    public Herbivore()
    {
        Name = "Herbivore";
    }
```

```
    public void EatenBy(Animal animal)
    {
        Console.WriteLine("Herbivore was eaten by: {0}", animal.Name);
    }
}
```

Puedes llamar al método Eat así:

```
var grass = new Grass();
var sheep = new Herbivore();
var lion = new Carnivore();

sheep.Eat(grass);
//Output: Grass was eaten by: Herbivore

lion.Eat(sheep);
//Output: Herbivore was eaten by: Carnivore
```

En este caso si intentas llamar:

```
sheep.Eat(lion);
```

No será posible porque el objeto lion no implementa la interfaz IFood. Intentar realizar la llamada anterior generará un error de compilación: "El tipo 'Carnivore' no se puede usar como parámetro de tipo 'TFood' en el tipo genérico o método 'Animal.Eat (TFood)'. No hay una conversión de referencia implícita desde ' Carnívoro 'a' IFood '".

## Covarianza

¿Cuándo es un `IEnumerable<T>` un subtipo de un `IEnumerable<T1>` diferente `IEnumerable<T1>` ? Cuando `T` es un subtipo de `T1` . `IEnumerable` es *covariante* en su parámetro `T` , lo que significa que la relación de subtipo de `IEnumerable` va en *la misma dirección* que `T` 's.

```
class Animal { /* ... */ }
class Dog : Animal { /* ... */ }

IEnumerable<Dog> dogs = Enumerable.Empty<Dog>();
IEnumerable<Animal> animals = dogs;  // IEnumerable<Dog> is a subtype of IEnumerable<Animal>
// dogs = animals;  // Compilation error - IEnumerable<Animal> is not a subtype of
IEnumerable<Dog>
```

Una instancia de un tipo genérico covariante con un parámetro de tipo dado se puede convertir implícitamente al mismo tipo genérico con un parámetro de tipo menos derivado.

Esta relación se mantiene porque `IEnumerable` *produce* `T` s pero no las consume. Un objeto que produce `Dog` s puede usarse como si produjera `Animal` s.

Los parámetros de tipo covariante se declaran usando la palabra clave `out` , porque el parámetro debe usarse solo como una *salida* .

```
interface IEnumerable<out T> { /* ... */ }
```

Un parámetro de tipo declarado como covariante puede no aparecer como entrada.

```
interface Bad<out T>
{
    void SetT(T t);  // type error
}
```

Aquí hay un ejemplo completo:

```
using NUnit.Framework;

namespace ToyStore
{
   enum Taste { Bitter, Sweet };

   interface IWidget
   {
      int Weight { get; }
   }

   interface IFactory<out TWidget>
       where TWidget : IWidget
   {
      TWidget Create();
   }

   class Toy : IWidget
   {
      public int Weight { get; set; }
      public Taste Taste { get; set; }
   }

   class ToyFactory : IFactory<Toy>
   {
      public const int StandardWeight = 100;
      public const Taste StandardTaste = Taste.Sweet;

      public Toy Create() { return new Toy { Weight = StandardWeight, Taste = StandardTaste };
}
   }

   [TestFixture]
   public class GivenAToyFactory
   {
      [Test]
      public static void WhenUsingToyFactoryToMakeWidgets()
      {
         var toyFactory = new ToyFactory();

         //// Without out keyword, note the verbose explicit cast:
         // IFactory<IWidget> rustBeltFactory = (IFactory<IWidget>)toyFactory;

         // covariance: concrete being assigned to abstract (shiny and new)
         IFactory<IWidget> widgetFactory = toyFactory;
         IWidget anotherToy = widgetFactory.Create();
         Assert.That(anotherToy.Weight, Is.EqualTo(ToyFactory.StandardWeight)); // abstract
contract
         Assert.That(((Toy)anotherToy).Taste, Is.EqualTo(ToyFactory.StandardTaste)); //
concrete contract
```

```
        }
    }
}
```

## Contravarianza

¿Cuándo es un `IComparer<T>` de un `IComparer<T1>` diferente `IComparer<T1>` ? Cuando `T1` es un subtipo de `T` `IComparer` es *contravariante* en su parámetro `T` , lo que significa que la relación de subtipo de `IComparer` va en la *dirección opuesta a la de* `T`

```
class Animal { /* ... */ }
class Dog : Animal { /* ... */ }

IComparer<Animal> animalComparer = /* ... */;
IComparer<Dog> dogComparer = animalComparer;  // IComparer<Animal> is a subtype of
IComparer<Dog>
// animalComparer = dogComparer;  // Compilation error - IComparer<Dog> is not a subtype of
IComparer<Animal>
```

Una instancia de un tipo genérico contravariante con un parámetro de tipo dado se puede convertir implícitamente al mismo tipo genérico con un parámetro de tipo más derivado.

Esta relación se mantiene porque `IComparer` *consume* `T` s pero no las produce. Un objeto que puede comparar cualesquiera dos `Animal` puede usarse para comparar dos `Dog` .

Los parámetros de tipo contravariante se declaran usando la palabra clave `in` , porque el parámetro se debe usar solo como *entrada* .

```
interface IComparer<in T> { /* ... */ }
```

Un parámetro de tipo declarado como contravariante puede no aparecer como una salida.

```
interface Bad<in T>
{
    T GetT();  // type error
}
```

## Invariancia

`IList<T>` nunca es un subtipo de un `IList<T1>` diferente `IList<T1>` . `IList` es *invariante* en su parámetro de tipo.

```
class Animal { /* ... */ }
class Dog : Animal { /* ... */ }

IList<Dog> dogs = new List<Dog>();
IList<Animal> animals = dogs;  // type error
```

No hay una relación de subtipo para las listas porque puede poner valores en una lista *y* sacar valores de una lista.

Si `IList` fuera covariante, podría agregar elementos del *subtipo incorrecto* a una lista determinada.

```
IList<Animal> animals = new List<Dog>();  // supposing this were allowed...
animals.Add(new Giraffe());  // ... then this would also be allowed, which is bad!
```

Si `IList` fuera contravariante, podría extraer valores del subtipo incorrecto de una lista determinada.

```
IList<Dog> dogs = new List<Animal> { new Dog(), new Giraffe() };  // if this were allowed...
Dog dog = dogs[1];  // ... then this would be allowed, which is bad!
```

Los parámetros de tipo invariantes se declaran omitiendo tanto el `in` y `out` palabras clave.

```
interface IList<T> { /* ... */ }
```

## Interfaces de variante

Las interfaces pueden tener parámetros de tipo variante.

```
interface IEnumerable<out T>
{
    // ...
}
interface IComparer<in T>
{
    // ...
}
```

pero las clases y las estructuras no pueden

```
class BadClass<in T1, out T2>  // not allowed
{
}

struct BadStruct<in T1, out T2>  // not allowed
{
}
```

ni tampoco declaraciones de método genérico

```
class MyClass
{
    public T Bad<out T, in T1>(T1 t1)  // not allowed
    {
        // ...
    }
}
```

El siguiente ejemplo muestra múltiples declaraciones de varianza en la misma interfaz

```
interface IFoo<in T1, out T2, T3>
//  T1 : Contravariant type
//  T2 : Covariant type
//  T3 : Invariant type
{
    // ...
}

IFoo<Animal, Dog, int> foo1 = /* ... */;
IFoo<Dog, Animal, int> foo2 = foo1;
// IFoo<Animal, Dog, int> is a subtype of IFoo<Dog, Animal, int>
```

## Delegados variantes

Los delegados pueden tener parámetros de tipo variante.

```
delegate void Action<in T>(T t);    // T is an input
delegate T Func<out T>();           // T is an output
delegate T2 Func<in T1, out T2>();  // T1 is an input, T2 is an output
```

Esto se desprende del Principio de Sustitución de Liskov , que establece (entre otras cosas) que un método D puede considerarse más derivado que un método B si:

- D tiene un tipo de retorno igual o más derivado que B
- D tiene tipos de parámetros correspondientes iguales o más generales que B

Por lo tanto las siguientes asignaciones son todas de tipo seguro:

```
Func<object, string> original = SomeMethod;
Func<object, object> d1 = original;
Func<string, string> d2 = original;
Func<string, object> d3 = original;
```

## Tipos de variantes como parámetros y valores de retorno.

Si un tipo covariante aparece como una salida, el tipo que contiene es covariante. Producir un productor de `T` s es como producir `T` s.

```
interface IReturnCovariant<out T>
{
    IEnumerable<T> GetTs();
}
```

Si un tipo contravariante aparece como una salida, el tipo que contiene es contravariante. Producir un consumidor de `T` s es como consumir `T` s.

```
interface IReturnContravariant<in T>
{
    IComparer<T> GetTComparer();
}
```

Si un tipo covariante aparece como una entrada, el tipo que contiene es contravariante. Consumir

un productor de `T` s es como consumir `T` s.

```
interface IAcceptCovariant<in T>
{
    void ProcessTs(IEnumerable<T> ts);
}
```

Si un tipo contravariante aparece como una entrada, el tipo que contiene es covariante. Consumir un consumidor de `T` s es como producir `T` s.

```
interface IAcceptContravariant<out T>
{
    void CompareTs(IComparer<T> tComparer);
}
```

## Comprobando la igualdad de valores genéricos.

Si la lógica de una clase o método genérico requiere verificar la igualdad de valores que tienen un tipo genérico, use la propiedad `EqualityComparer<TType>.Default` :

```
public void Foo<TBar>(TBar arg1, TBar arg2)
{
    var comparer = EqualityComparer<TBar>.Default;
    if (comparer.Equals(arg1,arg2)
    {
        ...
    }
}
```

Este enfoque es mejor que simplemente llamando `Object.Equals()` método, debido a los controles de aplicación comparador predeterminado, ya sea `TBar` tipo implementa `IEquatale<TBar>` interfaz en caso afirmativo, las llamadas y `IEquatable<TBar>.Equals(TBar other)` del método. Esto permite evitar el boxeo / unboxing de tipos de valor.

## Tipo genérico de fundición

```
    /// <summary>
    /// Converts a data type to another data type.
    /// </summary>
    public static class Cast
    {
        /// <summary>
        /// Converts input to Type of default value or given as typeparam T
        /// </summary>
        /// <typeparam name="T">typeparam is the type in which value will be returned, it
could be any type eg. int, string, bool, decimal etc.</typeparam>
        /// <param name="input">Input that need to be converted to specified type</param>
        /// <param name="defaultValue">defaultValue will be returned in case of value is null
or any exception occures</param>
        /// <returns>Input is converted in Type of default value or given as typeparam T and
returned</returns>
        public static T To<T>(object input, T defaultValue)
        {
```

```
            var result = defaultValue;
            try
            {
                if (input == null || input == DBNull.Value) return result;
                if (typeof (T).IsEnum)
                {
                    result = (T) Enum.ToObject(typeof (T), To(input,
Convert.ToInt32(defaultValue)));
                }
                else
                {
                    result = (T) Convert.ChangeType(input, typeof (T));
                }
            }
            catch (Exception ex)
            {
                Tracer.Current.LogException(ex);
            }

            return result;
        }

        /// <summary>
        /// Converts input to Type of typeparam T
        /// </summary>
        /// <typeparam name="T">typeparam is the type in which value will be returned, it
could be any type eg. int, string, bool, decimal etc.</typeparam>
        /// <param name="input">Input that need to be converted to specified type</param>
        /// <returns>Input is converted in Type of default value or given as typeparam T and
returned</returns>
        public static T To<T>(object input)
        {
            return To(input, default(T));
        }



    }
```

Usos:

```
std.Name = Cast.To<string>(drConnection["Name"]);
std.Age = Cast.To<int>(drConnection["Age"]);
std.IsPassed = Cast.To<bool>(drConnection["IsPassed"]);


// Casting type using default value
//Following both ways are correct
// Way 1 (In following style input is converted into type of default value)
std.Name = Cast.To(drConnection["Name"], "");
std.Marks = Cast.To(drConnection["Marks"], 0);
// Way 2
std.Name = Cast.To<string>(drConnection["Name"], "");
std.Marks = Cast.To<int>(drConnection["Marks"], 0);
```

## Lector de configuración con conversión de tipo genérico

```
    /// <summary>
```

```
    /// Read configuration values from app.config and convert to specified types
    /// </summary>
    public static class ConfigurationReader
    {
        /// <summary>
        /// Get value from AppSettings by key, convert to Type of default value or typeparam T
and return
        /// </summary>
        /// <typeparam name="T">typeparam is the type in which value will be returned, it
could be any type eg. int, string, bool, decimal etc.</typeparam>
        /// <param name="strKey">key to find value from AppSettings</param>
        /// <param name="defaultValue">defaultValue will be returned in case of value is null
or any exception occures</param>
        /// <returns>AppSettings value against key is returned in Type of default value or
given as typeparam T</returns>
        public static T GetConfigKeyValue<T>(string strKey, T defaultValue)
        {
            var result = defaultValue;
            try
            {
                if (ConfigurationManager.AppSettings[strKey] != null)
                    result = (T)Convert.ChangeType(ConfigurationManager.AppSettings[strKey],
typeof(T));
            }
            catch (Exception ex)
            {
                Tracer.Current.LogException(ex);
            }

            return result;
        }
        /// <summary>
        /// Get value from AppSettings by key, convert to Type of default value or typeparam T
and return
        /// </summary>
        /// <typeparam name="T">typeparam is the type in which value will be returned, it
could be any type eg. int, string, bool, decimal etc.</typeparam>
        /// <param name="strKey">key to find value from AppSettings</param>
        /// <returns>AppSettings value against key is returned in Type given as typeparam
T</returns>
        public static T GetConfigKeyValue<T>(string strKey)
        {
            return GetConfigKeyValue(strKey, default(T));
        }

    }
```

Usos:

```
var timeOut = ConfigurationReader.GetConfigKeyValue("RequestTimeout", 2000);
var url = ConfigurationReader.GetConfigKeyValue("URL", "www.someurl.com");
var enabled = ConfigurationReader.GetConfigKeyValue("IsEnabled", false);
```

Lea Genéricos en línea: https://riptutorial.com/es/csharp/topic/27/genericos

# Capítulo 79: Guid

## Introducción

GUID (o UUID) es un acrónimo de 'Identificador global único' (o 'Identificador universal único'). Es un número entero de 128 bits que se utiliza para identificar recursos.

## Observaciones

`Guid` son *identificadores únicos a nivel mundial* , también conocidos como *identificadores únicos universales de UUID* .

Son valores pseudoaleatorios de 128 bits. Hay tantos `Guid` s válidos (aproximadamente 10 ^ 18 `Guid` s para cada célula de cada persona en la Tierra) que si son generados por un buen algoritmo pseudoaleatorio, pueden considerarse únicos en todo el universo por todos los medios prácticos.

`Guid` s son los más utilizados como claves primarias en bases de datos. Su ventaja es que no tiene que llamar a la base de datos para obtener una nueva identificación que (casi) garantiza ser única.

## Examples

### Obteniendo la representación de cadena de un Guid

Se puede obtener una representación de cadena de un Guid utilizando el método integrado en `ToString`

```
string myGuidString = myGuid.ToString();
```

Dependiendo de sus necesidades, también puede formatear el Guid, agregando un argumento de tipo de formato a la llamada `ToString` .

```
var guid = new Guid("7febf16f-651b-43b0-a5e3-0da8da49e90d");

// None          "7febf16f651b43b0a5e30da8da49e90d"
Console.WriteLine(guid.ToString("N"));

// Hyphens       "7febf16f-651b-43b0-a5e3-0da8da49e90d"
Console.WriteLine(guid.ToString("D"));

// Braces        "{7febf16f-651b-43b0-a5e3-0da8da49e90d}"
Console.WriteLine(guid.ToString("B"));

// Parentheses   "(7febf16f-651b-43b0-a5e3-0da8da49e90d)"
Console.WriteLine(guid.ToString("P"));

// Hex           "{0x7febf16f,0x651b,0x43b0{0xa5,0xe3,0x0d,0xa8,0xda,0x49,0xe9,0x0d}}"
Console.WriteLine(guid.ToString("X"));
```

## Creando un Guid

Estas son las formas más comunes de crear una instancia de Guid:

- Creación de un guid vacío ( `00000000-0000-0000-0000-000000000000` ):

```
Guid g = Guid.Empty;
Guid g2 = new Guid();
```

- Creando un nuevo Guid (pseudoaleatorio):

```
Guid g = Guid.NewGuid();
```

- Creación de Guids con un valor específico:

```
Guid g = new Guid("0b214de7-8958-4956-8eed-28f9ba2c47c6");
Guid g2 = new Guid("0b214de7895849568eed28f9ba2c47c6");
Guid g3 = Guid.Parse("0b214de7-8958-4956-8eed-28f9ba2c47c6");
```

## Declarar un GUID que acepta nulos

Al igual que otros tipos de valor, GUID también tiene un tipo anulable que puede tomar un valor nulo.

Declaración:

```
Guid? myGuidVar = null;
```

Esto es particularmente útil cuando se recuperan datos de la base de datos cuando existe la posibilidad de que el valor de una tabla sea NULL.

Lea Guid en línea: https://riptutorial.com/es/csharp/topic/1153/guid

# Capítulo 80: Haciendo un hilo variable seguro

## Examples

### Controlar el acceso a una variable en un bucle Parallel.For

```
using System;
using System.Threading;
using System.Threading.Tasks;

class Program
{
    static void Main( string[] args )
    {
        object sync = new object();
        int sum = 0;
        Parallel.For( 1, 1000, ( i ) => {
            lock( sync ) sum = sum + i; // lock is necessary

            // As a practical matter, ensure this `parallel for` executes
            // on multiple threads by simulating a lengthy operation.
            Thread.Sleep( 1 );
        } );
        Console.WriteLine( "Correct answer should be 499500.  sum is: {0}", sum );
    }
}
```

No es suficiente hacer simplemente `sum = sum + i` sin el bloqueo porque la operación de lectura-modificación-escritura no es atómica. Un hilo sobrescribirá cualquier modificación externa a la `sum` que ocurra después de que haya leído el valor actual de `sum` , pero antes de que almacene el valor modificado de `sum + i` nuevamente en `sum` .

Lea Haciendo un hilo variable seguro en línea:
https://riptutorial.com/es/csharp/topic/4140/haciendo-un-hilo-variable-seguro

# Capítulo 81: Herencia

## Sintaxis

- clase DerivedClass: BaseClass
- clase DerivedClass: BaseClass, IExampleInterface
- clase DerivedClass: BaseClass, IExampleInterface, IAnotherInterface

## Observaciones

Las clases pueden heredar directamente de una sola clase, pero (en lugar o al mismo tiempo) pueden implementar una o más interfaces.

Structs puede implementar interfaces pero no puede heredar explícitamente de ningún tipo. `System.ValueType` implícitamente de `System.ValueType` , que a su vez hereda directamente de `System.Object` .

Las clases estáticas no pueden implementar interfaces.

## Examples

### Heredando de una clase base

Para evitar la duplicación de código, defina métodos y atributos comunes en una clase general como base:

```
public class Animal
{
    public string Name { get; set; }
    // Methods and attributes common to all animals
    public void Eat(Object dinner)
    {
        // ...
    }
    public void Stare()
    {
        // ...
    }
    public void Roll()
    {
        // ...
    }
}
```

Ahora que tiene una clase que representa a `Animal` en general, puede definir una clase que describa las peculiaridades de animales específicos:

```
public class Cat : Animal
```

```
{
    public Cat()
    {
        Name = "Cat";
    }
    // Methods for scratching furniture and ignoring owner
    public void Scratch(Object furniture)
    {
        // ...
    }
}
```

La clase Cat tiene acceso no solo a los métodos descritos en su definición explícitamente, sino también a todos los métodos definidos en la clase base general de `Animal` . Cualquier animal (sea o no un gato) podría comer, mirar fijamente o rodar. Sin embargo, un animal no podría rascarse, a menos que también fuera un gato. A continuación, podría definir otras clases que describen otros animales. (Como Gopher con un método para destruir jardines de flores y Sloth sin ningún método adicional).

### Heredar de una clase e implementar una interfaz.

```
public class Animal
{
    public string Name { get; set; }
}

public interface INoiseMaker
{
    string MakeNoise();
}

//Note that in C#, the base class name must come before the interface names
public class Cat : Animal, INoiseMaker
{
    public Cat()
    {
        Name = "Cat";
    }

    public string MakeNoise()
    {
        return "Nyan";
    }
}
```

### Heredando de una clase e implementando múltiples interfaces

```
public class LivingBeing
{
    string Name { get; set; }
}

public interface IAnimal
{
    bool HasHair { get; set; }
```

```
}

public interface INoiseMaker
{
    string MakeNoise();
}

//Note that in C#, the base class name must come before the interface names
public class Cat : LivingBeing, IAnimal, INoiseMaker
{
    public Cat()
    {
        Name = "Cat";
        HasHair = true;
    }

    public bool HasHair { get; set; }

    public string Name { get; set; }

    public string MakeNoise()
    {
        return "Nyan";
    }
}
```

## Herencia de prueba y navegación

```
interface BaseInterface {}
class BaseClass : BaseInterface {}

interface DerivedInterface {}
class DerivedClass : BaseClass, DerivedInterface {}

var baseInterfaceType = typeof(BaseInterface);
var derivedInterfaceType = typeof(DerivedInterface);
var baseType = typeof(BaseClass);
var derivedType = typeof(DerivedClass);

var baseInstance = new BaseClass();
var derivedInstance = new DerivedClass();

Console.WriteLine(derivedInstance is DerivedClass); //True
Console.WriteLine(derivedInstance is DerivedInterface); //True
Console.WriteLine(derivedInstance is BaseClass); //True
Console.WriteLine(derivedInstance is BaseInterface); //True
Console.WriteLine(derivedInstance is object); //True

Console.WriteLine(derivedType.BaseType.Name);  //BaseClass
Console.WriteLine(baseType.BaseType.Name);  //Object
Console.WriteLine(typeof(object).BaseType);  //null

Console.WriteLine(baseType.IsInstanceOfType(derivedInstance));  //True
Console.WriteLine(derivedType.IsInstanceOfType(baseInstance));  //False

Console.WriteLine(
    string.Join(",",
    derivedType.GetInterfaces().Select(t => t.Name).ToArray()));
//BaseInterface,DerivedInterface
```

```
Console.WriteLine(baseInterfaceType.IsAssignableFrom(derivedType)); //True
Console.WriteLine(derivedInterfaceType.IsAssignableFrom(derivedType)); //True
Console.WriteLine(derivedInterfaceType.IsAssignableFrom(baseType)); //False
```

## Extendiendo una clase base abstracta

A diferencia de las interfaces, que se pueden describir como contratos de implementación, las clases abstractas actúan como contratos de extensión.

Una clase abstracta no puede ser instanciada, debe ser extendida y la clase resultante (o clase derivada) puede ser instanciada.

Las clases abstractas se utilizan para proporcionar implementaciones genéricas.

```
public abstract class Car
{
    public void HonkHorn() {
        // Implementation of horn being honked
    }
}

public class Mustang : Car
{
    // Simply by extending the abstract class Car, the Mustang can HonkHorn()
    // If Car were an interface, the HonkHorn method would need to be included
    // in every class that implemented it.
}
```

El ejemplo anterior muestra cómo cualquier clase de Automóviles que se extienden automáticamente recibirá el método HonkHorn con la implementación. Esto significa que cualquier desarrollador que cree un nuevo Coche no tendrá que preocuparse por cómo sonará su bocina.

## Constructores en una subclase

Cuando creas una subclase de una clase base, puedes construir la clase base usando `: base` después de los parámetros del constructor de la subclase.

```
class Instrument
{
    string type;
    bool clean;

    public Instrument (string type, bool clean)
    {
        this.type = type;
        this.clean = clean;
    }
}

class Trumpet : Instrument
{
    bool oiled;
```

```
    public Trumpet(string type, bool clean, bool oiled) : base(type, clean)
    {
        this.oiled = oiled;
    }
}
```

## Herencia. Secuencia de llamadas de los constructores.

Considera que tenemos un `Animal` clase que tiene un `Dog` clase infantil

```
class Animal
{
    public Animal()
    {
        Console.WriteLine("In Animal's constructor");
    }
}

class Dog : Animal
{
    public Dog()
    {
        Console.WriteLine("In Dog's constructor");
    }
}
```

Por defecto, cada clase hereda implícitamente la clase `Object` .

Esto es lo mismo que el código anterior.

```
class Animal : Object
{
    public Animal()
    {
        Console.WriteLine("In Animal's constructor");
    }
}
```

Al crear una instancia de la clase `Dog` , **se llamará al constructor predeterminado de las clases base (sin parámetros) si no hay una llamada explícita a otro constructor en la clase principal** . En nuestro caso, primero se llamará constructor `Object's` , luego constructor de `Animal's` y al final de `Dog's` .

```
public class Program
{
    public static void Main()
    {
        Dog dog = new Dog();
    }
}
```

La salida será

En constructor de animales.
En constructor de perros.

Ver demostración

**Llame al constructor de los padres explícitamente.**

En los ejemplos anteriores, nuestro constructor de clase `Dog` llama al constructor **predeterminado** de la clase `Animal` . Si lo desea, puede especificar a qué constructor se debe llamar: es posible llamar a cualquier constructor que esté definido en la clase principal.

Consideremos que tenemos estas dos clases.

```
class Animal
{
    protected string name;

    public Animal()
    {
        Console.WriteLine("Animal's default constructor");
    }

    public Animal(string name)
    {
        this.name = name;
        Console.WriteLine("Animal's constructor with 1 parameter");
        Console.WriteLine(this.name);
    }
}

class Dog : Animal
{
    public Dog() : base()
    {
        Console.WriteLine("Dog's default constructor");
    }

    public Dog(string name) : base(name)
    {
        Console.WriteLine("Dog's constructor with 1 parameter");
        Console.WriteLine(this.name);
    }
}
```

### ¿Qué está pasando aquí?

Tenemos 2 constructores en cada clase.

### ¿Qué significa la `base` ?

`base` es una referencia a la clase padre. En nuestro caso, cuando creamos una instancia de clase `Dog` como esta.

```
Dog dog = new Dog();
```

El tiempo de ejecución primero llama al `Dog()` , que es el constructor sin parámetros. Pero su cuerpo no funciona de inmediato. Después de los paréntesis del constructor tenemos una llamada de este tipo: `base()` , lo que significa que cuando llamamos al constructor de `Dog` predeterminado, a su vez llamará al constructor **predeterminado** del padre. Después de que se ejecute el constructor principal, este regresará y, finalmente, ejecutará el cuerpo del constructor `Dog()` .

Así que la salida será así:

> Constructor por defecto del animal
> Constructor por defecto del perro

Ver demostración

**¿Y ahora si llamamos al constructor `Dog's` con un parámetro?**

```
Dog dog = new Dog("Rex");
```

Usted sabe que los miembros de la clase principal que no son privados son heredados por la clase secundaria, lo que significa que `Dog` también tendrá el campo de `name` .
En este caso pasamos un argumento a nuestro constructor. A su vez, pasa el argumento al **constructor de la** clase padre **con un parámetro** , que inicializa el campo de `name` .

La salida será

```
Animal's constructor with 1 parameter
Rex
Dog's constructor with 1 parameter
Rex
```

**Resumen:**

Cada creación de objetos comienza a partir de la clase base. En la herencia, las clases que están en la jerarquía están encadenadas. Como todas las clases se derivan de `Object` , el primer constructor al que se llama cuando se crea un objeto es el constructor de la clase `Object` ; Luego se llama al siguiente constructor en la cadena y solo después de que todos se llaman, se crea el objeto

**palabra clave base**

1. La palabra clave base se utiliza para acceder a los miembros de la clase base desde una clase derivada:
2. Llame a un método en la clase base que ha sido anulado por otro método. Especifique a qué constructor de clase base se debe llamar al crear instancias de la clase derivada.

## Heredando metodos

Hay varias formas de heredar los métodos.

```
public abstract class Car
```

```
{
    public void HonkHorn() {
        // Implementation of horn being honked
    }

    // virtual methods CAN be overridden in derived classes
    public virtual void ChangeGear() {
        // Implementation of gears being changed
    }

    // abstract methods MUST be overridden in derived classes
    public abstract void Accelerate();
}

public class Mustang : Car
{
    // Before any code is added to the Mustang class, it already contains
    // implementations of HonkHorn and ChangeGear.

    // In order to compile, it must be given an implementation of Accelerate,
    // this is done using the override keyword
    public override void Accelerate() {
        // Implementation of Mustang accelerating
    }

    // If the Mustang changes gears differently to the implementation in Car
    // this can be overridden using the same override keyword as above
    public override void ChangeGear() {
        // Implementation of Mustang changing gears
    }
}
```

### Herencias Anti-patrones

# Herencia impropia

Digamos que hay 2 clases de clase `Foo` y `Bar` . `Foo` tiene dos características `Do1` y `Do2` . `Bar` necesita usar `Do1` de `Foo` , pero no necesita `Do2` o necesita una función que sea equivalente a `Do2` pero hace algo completamente diferente.

**Mala manera** : hacer `Do2()` en `Foo` virtual, anularlo en `Bar` o simplemente `throw Exception` en `Bar` para `Do2()`

```
public class Bar : Foo
{
    public override void Do2()
    {
        //Does something completely different that you would expect Foo to do
        //or simply throws new Exception
    }
}
```

#### Buen camino

Saque `Do1()` de `Foo` y póngalo en la nueva clase `Baz` luego herede tanto a `Foo` como a `Bar` de `Baz` e

implemente `Do2()` separado

```
public class Baz
{
    public void Do1()
    {
        // magic
    }
}

public class Foo : Baz
{
    public void Do2()
    {
        // foo way
    }
}

public class Bar : Baz
{
    public void Do2()
    {
        // bar way or not have Do2 at all
    }
}
```

Ahora, por qué el primer ejemplo es malo y el segundo es bueno: cuando el desarrollador nr2 tiene que hacer un cambio en `Foo` , es probable que rompa la implementación de `Bar` porque ahora `Bar` es inseparable de `Foo` . Al hacerlo mediante el último ejemplo, `Foo` y `Bar` commonalty se han trasladado a `Baz` y no se afectan entre sí (como no debería).

## Clase base con especificación de tipo recursivo

Definición única de una clase base genérica con especificador de tipo recursivo. Cada nodo tiene un padre y varios hijos.

```
/// <summary>
/// Generic base class for a tree structure
/// </summary>
/// <typeparam name="T">The node type of the tree</typeparam>
public abstract class Tree<T> where T : Tree<T>
{
    /// <summary>
    /// Constructor sets the parent node and adds this node to the parent's child nodes
    /// </summary>
    /// <param name="parent">The parent node or null if a root</param>
    protected Tree(T parent)
    {
        this.Parent=parent;
        this.Children=new List<T>();
        if(parent!=null)
        {
            parent.Children.Add(this as T);
        }
    }
    public T Parent { get; private set; }
    public List<T> Children { get; private set; }
```

```
    public bool IsRoot { get { return Parent==null; } }
    public bool IsLeaf { get { return Children.Count==0; } }
    /// <summary>
    /// Returns the number of hops to the root object
    /// </summary>
    public int Level { get { return IsRoot ? 0 : Parent.Level+1; } }
}
```

Lo anterior se puede reutilizar cada vez que se necesita definir una jerarquía de objetos de árbol. El objeto de nodo en el árbol tiene que heredar de la clase base con

```
public class MyNode : Tree<MyNode>
{
    // stuff
}
```

cada clase de nodo sabe dónde se encuentra en la jerarquía, qué es el objeto principal y qué son los objetos secundarios. Varios tipos incorporados utilizan una estructura de árbol, como `Control` o `XmlElement` y el `Tree<T>` se puede usar como una clase base de *cualquier* tipo en su código.

---

Por ejemplo, para crear una jerarquía de partes donde el peso total se calcula a partir del peso de *todos* los hijos, haga lo siguiente:

```
public class Part : Tree<Part>
{
    public static readonly Part Empty = new Part(null) { Weight=0 };
    public Part(Part parent) : base(parent) { }
    public Part Add(float weight)
    {
        return new Part(this) { Weight=weight };
    }
    public float Weight { get; set; }

    public float TotalWeight { get { return Weight+Children.Sum((part) => part.TotalWeight); }
}
}
```

para ser utilizado como

```
// [Q:2.5] -- [P:4.2] -- [R:0.4]
//    \
//      - [Z:0.8]
var Q = Part.Empty.Add(2.5f);
var P = Q.Add(4.2f);
var R = P.Add(0.4f);
var Z = Q.Add(0.9f);

// 2.5+(4.2+0.4)+0.9 = 8.0
float weight = Q.TotalWeight;
```

---

Otro ejemplo sería en la definición de marcos de coordenadas relativas. En este caso, la posición verdadera del marco de coordenadas depende de las posiciones de *todos* los marcos de

coordenadas principales.

```
public class RelativeCoordinate : Tree<RelativeCoordinate>
{
    public static readonly RelativeCoordinate Start = new RelativeCoordinate(null,
PointF.Empty) { };
    public RelativeCoordinate(RelativeCoordinate parent, PointF local_position)
        : base(parent)
    {
        this.LocalPosition=local_position;
    }
    public PointF LocalPosition { get; set; }
    public PointF GlobalPosition
    {
        get
        {
            if(IsRoot) return LocalPosition;
            var parent_pos = Parent.GlobalPosition;
            return new PointF(parent_pos.X+LocalPosition.X, parent_pos.Y+LocalPosition.Y);
        }
    }
    public float TotalDistance
    {
        get
        {
            float dist =
(float)Math.Sqrt(LocalPosition.X*LocalPosition.X+LocalPosition.Y*LocalPosition.Y);
            return IsRoot ? dist : Parent.TotalDistance+dist;
        }
    }
    public RelativeCoordinate Add(PointF local_position)
    {
        return new RelativeCoordinate(this, local_position);
    }
    public RelativeCoordinate Add(float x, float y)
    {
        return Add(new PointF(x, y));
    }
}
```

para ser utilizado como

```
// Define the following coordinate system hierarchy
//
// o--> [A1] --+--> [B1] -----> [C1]
//              |
//              +--> [B2] --+--> [C2]
//                          |
//                          +--> [C3]

var A1 = RelativeCoordinate.Start;
var B1 = A1.Add(100, 20);
var B2 = A1.Add(160, 10);

var C1 = B1.Add(120, -40);
var C2 = B2.Add(80, -20);
var C3 = B2.Add(60, -30);

double dist1 = C1.TotalDistance;
```

Lea Herencia en línea: https://riptutorial.com/es/csharp/topic/29/herencia

# Capítulo 82: ICloneable

## Sintaxis

- objeto ICloneable.Clone () {return Clone (); } // Implementación privada del método de interfaz que utiliza nuestra función Clone () pública personalizada.
- público Foo Clone () {devolver nuevo Foo (esto); } // El método de clonación público debe utilizar la lógica del constructor de copia.

## Observaciones

El `CLR` requiere un `object Clone()` definición de método `object Clone()` que no es de tipo seguro. Es una práctica común anular este comportamiento y definir un método de tipo seguro que devuelva una copia de la clase contenedora.

Es decisión del autor decidir si la clonación significa solo una copia superficial o copia profunda. Para estructuras inmutables que contienen referencias, se recomienda hacer una copia profunda. Para las clases que son referencias en sí mismas, es probable que esté bien implementar una copia superficial.

NOTA: En `C#` un método de interfaz se puede implementar de forma privada con la sintaxis que se muestra arriba.

## Examples

### Implementación de ICloneable en una clase.

Implementar `ICloneable` en una clase con un toque. Expone un tipo público seguro `Clone()` e implementa el `object Clone()` privada.

```
public class Person : ICloneable
{
    // Contents of class
    public string Name { get; set; }
    public int Age { get; set; }
    // Constructor
    public Person(string name, int age)
    {
        this.Name=name;
        this.Age=age;
    }
    // Copy Constructor
    public Person(Person other)
    {
        this.Name=other.Name;
        this.Age=other.Age;
    }

    #region ICloneable Members
    // Type safe Clone
```

```
    public Person Clone() { return new Person(this); }
    // ICloneable implementation
    object ICloneable.Clone()
    {
        return Clone();
    }
    #endregion
}
```

Más tarde para ser utilizado de la siguiente manera:

```
{
    Person bob=new Person("Bob", 25);
    Person bob_clone=bob.Clone();
    Debug.Assert(bob_clone.Name==bob.Name);

    bob.Age=56;
    Debug.Assert(bob.Age!=bob.Age);
}
```

Tenga en cuenta que cambiar la edad de `bob` no cambia la edad de `bob_clone` . Esto se debe a que el diseño utiliza la clonación en lugar de asignar variables (de referencia).

## Implementación de ICloneable en una estructura.

La implementación de ICloneable para una estructura generalmente no es necesaria porque las estructuras hacen una copia de memberwise con el operador de asignación `=` . Pero el diseño puede requerir la implementación de otra interfaz que herede de `ICloneable` .

Otra razón sería si la estructura contiene un tipo de referencia (o una matriz) que también necesitaría copiarse.

```
// Structs are recommended to be immutable objects
[ImmutableObject(true)]
public struct Person : ICloneable
{
    // Contents of class
    public string Name { get; private set; }
    public int Age { get; private set; }
    // Constructor
    public Person(string name, int age)
    {
        this.Name=name;
        this.Age=age;
    }
    // Copy Constructor
    public Person(Person other)
    {
        // The assignment operator copies all members
        this=other;
    }

    #region ICloneable Members
    // Type safe Clone
    public Person Clone() { return new Person(this); }
    // ICloneable implementation
```

```
    object ICloneable.Clone()
    {
        return Clone();
    }
    #endregion
}
```

Más tarde para ser utilizado de la siguiente manera:

```
static void Main(string[] args)
{
    Person bob=new Person("Bob", 25);
    Person bob_clone=bob.Clone();
    Debug.Assert(bob_clone.Name==bob.Name);
}
```

Lea ICloneable en línea: https://riptutorial.com/es/csharp/topic/7917/icloneable

# Capítulo 83: Identidad ASP.NET

## Introducción

Tutoriales relacionados con la identidad de asp.net, como la administración de usuarios, la administración de roles, la creación de tokens y más.

## Examples

### Cómo implementar el token de restablecimiento de contraseña en la identidad de asp.net mediante el administrador de usuarios.

1. Cree una nueva carpeta llamada MyClasses y cree y agregue la siguiente clase

```
public class GmailEmailService:SmtpClient
{
    // Gmail user-name
    public string UserName { get; set; }

    public GmailEmailService() :
        base(ConfigurationManager.AppSettings["GmailHost"],
Int32.Parse(ConfigurationManager.AppSettings["GmailPort"]))
    {
        //Get values from web.config file:
        this.UserName = ConfigurationManager.AppSettings["GmailUserName"];
        this.EnableSsl = Boolean.Parse(ConfigurationManager.AppSettings["GmailSsl"]);
        this.UseDefaultCredentials = false;
        this.Credentials = new System.Net.NetworkCredential(this.UserName,
ConfigurationManager.AppSettings["GmailPassword"]);
    }
}
```

2. Configure su clase de identidad

```
public async Task SendAsync(IdentityMessage message)
{
    MailMessage email = new MailMessage(new MailAddress("youremailadress@domain.com",
"(any subject here)"),
    new MailAddress(message.Destination));
    email.Subject = message.Subject;
    email.Body = message.Body;

    email.IsBodyHtml = true;

    GmailEmailService mailClient = new GmailEmailService();
    await mailClient.SendMailAsync(email);
}
```

3. Agregue sus credenciales al web.config. No utilicé gmail en esta parte porque el uso de gmail está bloqueado en mi lugar de trabajo y todavía funciona perfectamente.

```
<add key="GmailUserName" value="youremail@yourdomain.com"/>
<add key="GmailPassword" value="yourPassword"/>
<add key="GmailHost" value="yourServer"/>
<add key="GmailPort" value="yourPort"/>
<add key="GmailSsl" value="chooseTrueOrFalse"/>
<!--Smptp Server (confirmations emails)-->
```

4. Realice los cambios necesarios en el controlador de su cuenta. Agregue el siguiente código resaltado.

```
using Microsoft.AspNet.Identity;
using Microsoft.AspNet.Identity.EntityFramework;
using Microsoft.AspNet.Identity.Owin;
using Microsoft.Owin.Security;
using Microsoft.Owin.Security.DataProtection;
using System;
using System.Linq;
using System.Threading.Tasks;
using System.Web;
using System.Web.Mvc;
using MYPROJECT.Models;

namespace MYPROJECT.Controllers
{
    [Authorize]
    public class AccountController : Controller
    {
        private ApplicationSignInManager _signInManager;
        private ApplicationUserManager _userManager;
        ApplicationDbContext _context;
        DpapiDataProtectionProvider provider = new DpapiDataProtectionProvider("MYPROJECT");
        EmailService EmailService = new EmailService();


        public AccountController()
            : this(new UserManager<ApplicationUser>(new UserStore<ApplicationUser>(new ApplicationDbContext
        {
            _context = new ApplicationDbContext();

        }

        public AccountController(UserManager<ApplicationUser> userManager, ApplicationSignInManager signIn
        {
            UserManager = userManager;
            SignInManager = signInManager;
            UserManager.UserTokenProvider = new DataProtectorTokenProvider<ApplicationUser>(
                    provider.Create("EmailConfirmation"));

        }

        private AccountController(UserManager<ApplicationUser> userManager)
        {
            UserManager = userManager;
            UserManager.UserTokenProvider = new DataProtectorTokenProvider<ApplicationUser>(
                    provider.Create("EmailConfirmation"));
        }

        public UserManager<ApplicationUser> UserManager { get; private set; }

        public ApplicationSignInManager SignInManager
        {
            get
            {
                return _signInManager ?? HttpContext.GetOwinContext().Get<ApplicationSignInManager>();
            }
            private set
            {
                _signInManager = value;
            }
        }
```

```
//
// GET: /Account/ForgotPassword
[AllowAnonymous]
public ActionResult ForgotPassword()
{
    return View();
}

//
// POST: /Account/ForgotPassword
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<ActionResult> ForgotPassword(ForgotPasswordViewModel model)
{
    if (ModelState.IsValid)
    {

        var user = await UserManager.FindByNameAsync(model.UserName);

        if (user == null || !(await UserManager.IsEmailConfirmedAsync(user.Id)))
        {
            // Don't reveal that the user does not exist or is not confirmed
            return View("ForgotPasswordConfirmation");
        }

        // For more information on how to enable account confirmation and password reset please visit http
        // Send an email with this link
        string code = await UserManager.GeneratePasswordResetTokenAsync(user.Id);
        code = HttpUtility.UrlEncode(code);
        var callbackUrl = Url.Action("ResetPassword", "Account", new { userId = user.Id, code = HttpUtility
        await EmailService.SendAsync(new IdentityMessage
        {
            Body = "Please reset your password by clicking <a href=\"" + callbackUrl + "\">here</a>",
            Destination = user.Email,
            Subject = "Reset Password"
        });
        //await UserManager.SendEmailAsync(user.Id, "Reset Password", "Please reset your password by clicki
        return RedirectToAction("ForgotPasswordConfirmation", "Account");
    }

    // If we got this far, something failed, redisplay form
    return View(model);
}
```

Compila luego corre. ¡Aclamaciones!

Lea Identidad ASP.NET en línea: https://riptutorial.com/es/csharp/topic/9577/identidad-asp-net

# Capítulo 84: ILGenerador

## Examples

### Crea un DynamicAssembly que contiene un método auxiliar de UnixTimestamp

Este ejemplo muestra el uso de ILGenerator mediante la generación de código que hace uso de miembros ya existentes y nuevos creados, así como el manejo básico de excepciones. El siguiente código emite un DynamicAssembly que contiene un equivalente a este código c #:

```
public static class UnixTimeHelper
{
    private readonly static DateTime EpochTime = new DateTime(1970, 1, 1);

    public static int UnixTimestamp(DateTime input)
    {
        int totalSeconds;
        try
        {
            totalSeconds =
checked((int)input.Subtract(UnixTimeHelper.EpochTime).TotalSeconds);
        }
        catch (OverflowException overflowException)
        {
            throw new InvalidOperationException("It's to late for an Int32 timestamp.",
overflowException);
        }
        return totalSeconds;
    }
}
```

---

```
//Get the required methods
var dateTimeCtor = typeof (DateTime)
    .GetConstructor(new[] {typeof (int), typeof (int), typeof (int)});
var dateTimeSubstract = typeof (DateTime)
    .GetMethod(nameof(DateTime.Subtract), new[] {typeof (DateTime)});
var timeSpanSecondsGetter = typeof (TimeSpan)
    .GetProperty(nameof(TimeSpan.TotalSeconds)).GetGetMethod();
var invalidOperationCtor = typeof (InvalidOperationException)
    .GetConstructor(new[] {typeof (string), typeof (Exception)});

if (dateTimeCtor == null || dateTimeSubstract == null ||
    timeSpanSecondsGetter == null || invalidOperationCtor == null)
{
    throw new Exception("Could not find a required Method, can not create Assembly.");
}

//Setup the required members
var an = new AssemblyName("UnixTimeAsm");
var dynAsm = AppDomain.CurrentDomain.DefineDynamicAssembly(an,
AssemblyBuilderAccess.RunAndSave);
var dynMod = dynAsm.DefineDynamicModule(an.Name, an.Name + ".dll");
```

```
var dynType = dynMod.DefineType("UnixTimeHelper",
    TypeAttributes.Abstract | TypeAttributes.Sealed | TypeAttributes.Public);

var epochTimeField = dynType.DefineField("EpochStartTime", typeof (DateTime),
    FieldAttributes.Private | FieldAttributes.Static | FieldAttributes.InitOnly);

var cctor =
    dynType.DefineConstructor(
        MethodAttributes.Private | MethodAttributes.HideBySig | MethodAttributes.SpecialName |
        MethodAttributes.RTSpecialName | MethodAttributes.Static, CallingConventions.Standard,
        Type.EmptyTypes);

var cctorGen = cctor.GetILGenerator();
cctorGen.Emit(OpCodes.Ldc_I4, 1970); //Load the DateTime constructor arguments onto the stack
cctorGen.Emit(OpCodes.Ldc_I4_1);
cctorGen.Emit(OpCodes.Ldc_I4_1);
cctorGen.Emit(OpCodes.Newobj, dateTimeCtor); //Call the constructor
cctorGen.Emit(OpCodes.Stsfld, epochTimeField); //Store the object in the static field
cctorGen.Emit(OpCodes.Ret);

var unixTimestampMethod = dynType.DefineMethod("UnixTimestamp",
    MethodAttributes.Public | MethodAttributes.HideBySig | MethodAttributes.Static,
    CallingConventions.Standard, typeof (int), new[] {typeof (DateTime)});

unixTimestampMethod.DefineParameter(1, ParameterAttributes.None, "input");

var methodGen = unixTimestampMethod.GetILGenerator();
methodGen.DeclareLocal(typeof (TimeSpan));
methodGen.DeclareLocal(typeof (int));
methodGen.DeclareLocal(typeof (OverflowException));

methodGen.BeginExceptionBlock(); //Begin the try block
methodGen.Emit(OpCodes.Ldarga_S, (byte) 0); //To call a method on a struct we need to load the
address of it
methodGen.Emit(OpCodes.Ldsfld, epochTimeField);
    //Load the object of the static field we created as argument for the following call
methodGen.Emit(OpCodes.Call, dateTimeSubstract); //Call the substract method on the input
DateTime
methodGen.Emit(OpCodes.Stloc_0); //Store the resulting TimeSpan in a local
methodGen.Emit(OpCodes.Ldloca_S, (byte) 0); //Load the locals address to call a method on it
methodGen.Emit(OpCodes.Call, timeSpanSecondsGetter); //Call the TotalSeconds Get method on the
TimeSpan
methodGen.Emit(OpCodes.Conv_Ovf_I4); //Convert the result to Int32; throws an exception on
overflow
methodGen.Emit(OpCodes.Stloc_1); //store the result for returning later
//The leave instruction to jump behind the catch block will be automatically emitted
methodGen.BeginCatchBlock(typeof (OverflowException)); //Begin the catch block
//When we are here, an OverflowException was thrown, that is now on the stack
methodGen.Emit(OpCodes.Stloc_2); //Store the exception in a local.
methodGen.Emit(OpCodes.Ldstr, "It's to late for an Int32 timestamp.");
    //Load our error message onto the stack
methodGen.Emit(OpCodes.Ldloc_2); //Load the exception again
methodGen.Emit(OpCodes.Newobj, invalidOperationCtor);
    //Create an InvalidOperationException with our message and inner Exception
methodGen.Emit(OpCodes.Throw); //Throw the created exception
methodGen.EndExceptionBlock(); //End the catch block
//When we are here, everything is fine
methodGen.Emit(OpCodes.Ldloc_1); //Load the result value
methodGen.Emit(OpCodes.Ret); //Return it
```

```
dynType.CreateType();

dynAsm.Save(an.Name + ".dll");
```

## Crear anulación de método

Este ejemplo muestra cómo anular el método `ToString` en la clase generada

```
// create an Assembly and new type
var name = new AssemblyName("MethodOverriding");
var dynAsm = AppDomain.CurrentDomain.DefineDynamicAssembly(name,
AssemblyBuilderAccess.RunAndSave);
var dynModule = dynAsm.DefineDynamicModule(name.Name, $"{name.Name}.dll");
var typeBuilder = dynModule.DefineType("MyClass", TypeAttributes.Public |
TypeAttributes.Class);

// define a new method
var toStr = typeBuilder.DefineMethod(
    "ToString", // name
    MethodAttributes.Public | MethodAttributes.Virtual, // modifiers
    typeof(string), // return type
    Type.EmptyTypes); // argument types
var ilGen = toStr.GetILGenerator();
ilGen.Emit(OpCodes.Ldstr, "Hello, world!");
ilGen.Emit(OpCodes.Ret);

// set this method as override of object.ToString
typeBuilder.DefineMethodOverride(toStr, typeof(object).GetMethod("ToString"));
var type = typeBuilder.CreateType();

// now test it:
var instance = Activator.CreateInstance(type);
Console.WriteLine(instance.ToString());
```

Lea ILGenerador en línea: https://riptutorial.com/es/csharp/topic/667/ilgenerador

# Capítulo 85: Implementación Singleton

## Examples

### Singleton estáticamente inicializado

```
public class Singleton
{
    private readonly static Singleton instance = new Singleton();
    private Singleton() { }
    public static Singleton Instance => instance;
}
```

Esta implementación es segura para subprocesos porque en este caso `instance` objeto de `instance` se inicializa en el constructor estático. El CLR ya garantiza que todos los constructores estáticos se ejecuten seguros para subprocesos.

La `instance` mutante no es una operación segura para subprocesos, por lo tanto, el atributo de `readonly` garantiza la inmutabilidad después de la inicialización.

### Singleton perezoso, seguro para subprocesos (con bloqueo de doble control)

Esta versión segura para subprocesos de un singleton era necesaria en las versiones anteriores de .NET donde no se garantizaba que `static` inicialización `static` fuera segura para subprocesos. En versiones más modernas del marco, generalmente se prefiere un singleton estáticamente inicializado porque es muy fácil cometer errores de implementación en el siguiente patrón.

```
public sealed class ThreadSafeSingleton
{
   private static volatile ThreadSafeSingleton instance;
   private static object lockObject = new Object();

   private ThreadSafeSingleton()
   {
   }

   public static ThreadSafeSingleton Instance
   {
      get
      {
         if (instance == null)
         {
            lock (lockObject)
            {
               if (instance == null)
               {
                  instance = new ThreadSafeSingleton();
               }
            }
         }

         return instance;
```

```
        }
    }
}
```

Observe que la verificación `if (instance == null)` se realiza dos veces: una vez antes de que se adquiera el bloqueo y una vez después. Esta implementación aún sería segura para subprocesos incluso sin la primera comprobación nula. Sin embargo, eso significaría que se adquirirá un bloqueo *cada vez que* se solicite la instancia, y eso causaría un deterioro en el rendimiento. La primera comprobación nula se agrega para que el bloqueo no se obtenga a menos que sea necesario. La segunda comprobación nula se asegura de que solo la primera hebra para adquirir el bloqueo cree la instancia. Los otros subprocesos encontrarán la instancia que se va a rellenar y seguir adelante.

### Singleton perezoso, seguro para hilos (usando perezoso )

El tipo .Net 4.0 Lazy garantiza la inicialización de objetos seguros para subprocesos, por lo que este tipo podría usarse para hacer Singletons.

```
public class LazySingleton
{
    private static readonly Lazy<LazySingleton> _instance =
        new Lazy<LazySingleton>(() => new LazySingleton());

    public static LazySingleton Instance
    {
        get { return _instance.Value; }
    }

    private LazySingleton() { }
}
```

El uso de `Lazy<T>` se asegurará de que el objeto solo se ejecute cuando se use en algún lugar del código de llamada.

Un uso simple será como:

```
using System;

public class Program
{
    public static void Main()
    {
        var instance = LazySingleton.Instance;
    }
}
```

Demo en vivo en .NET Fiddle

### Singleton seguro, seguro para subprocesos (para .NET 3.5 o anterior, implementación alternativa)

Debido a que en .NET 3.5 y versiones anteriores no tienes la clase `Lazy<T>` , usas el siguiente

patrón:

```
public class Singleton
{
    private Singleton() // prevents public instantiation
    {
    }

    public static Singleton Instance
    {
        get
        {
            return Nested.instance;
        }
    }

    private class Nested
    {
        // Explicit static constructor to tell C# compiler
        // not to mark type as beforefieldinit
        static Nested()
        {
        }

        internal static readonly Singleton instance = new Singleton();
    }
}
```

Esto está inspirado en la publicación del blog de Jon Skeet .

Debido a que el `Nested` clase anidada es privada y la creación de instancias de la instancia singleton no se activará mediante el acceso a otros miembros de la `Sigleton` clase (como una propiedad pública de sólo lectura, por ejemplo).

## Eliminación de la instancia de Singleton cuando ya no sea necesaria.

La mayoría de los ejemplos muestran la `LazySingleton` instancias y la retención de un objeto `LazySingleton` hasta que la aplicación propietaria haya finalizado, incluso si la aplicación ya no lo necesita. Una solución para esto es implementar `IDisposable` y establecer la instancia del objeto en nulo de la siguiente manera:

```
public class LazySingleton : IDisposable
{
    private static volatile Lazy<LazySingleton> _instance;
    private static volatile int _instanceCount = 0;
    private bool _alreadyDisposed = false;

public static LazySingleton Instance
{
    get
    {
        if (_instance == null)
            _instance = new Lazy<LazySingleton>(() => new LazySingleton());
        _instanceCount++;
        return _instance.Value;
    }
```

```
}

private LazySingleton() { }

// Public implementation of Dispose pattern callable by consumers.
public void Dispose()
{
    if (--_instanceCount == 0) // No more references to this object.
    {
       Dispose(true);
       GC.SuppressFinalize(this);
    }
}

// Protected implementation of Dispose pattern.
protected virtual void Dispose(bool disposing)
{
    if (_alreadyDisposed) return;

    if (disposing)
    {
        _instance = null; // Allow GC to dispose of this instance.
        // Free any other managed objects here.
    }

    // Free any unmanaged objects here.
    _alreadyDisposed = true;
}
```

El código anterior elimina la instancia antes de la finalización de la aplicación, pero solo si los consumidores llaman a `Dispose()` en el objeto después de cada uso. Dado que no hay garantía de que esto suceda o una forma de forzarlo, tampoco hay garantía de que la instancia se pueda disponer. Pero si esta clase se usa internamente, es más fácil asegurar que se llame al método `Dispose()` después de cada uso. Un ejemplo sigue:

```
public class Program
{
    public static void Main()
    {
        using (var instance = LazySingleton.Instance)
        {
            // Do work with instance
        }
    }
}
```

Tenga en cuenta que este ejemplo **no** es **seguro para subprocesos** .

Lea Implementación Singleton en línea:
https://riptutorial.com/es/csharp/topic/1192/implementacion-singleton

# Capítulo 86: Implementando el patrón de diseño de peso mosca

## Examples

### Implementando mapa en juego de rol

El peso mosca es uno de los patrones de diseño estructural. Se utiliza para disminuir la cantidad de memoria utilizada al compartir la mayor cantidad de datos posible con objetos similares. Este documento le enseñará cómo usar Flyweight DP correctamente.

Déjame explicarte la idea de ello en un ejemplo simple. Imagina que estás trabajando en un juego de rol y necesitas cargar un archivo enorme que contenga algunos personajes. Por ejemplo:

- `#` es la hierba. Puedes caminar sobre ella.
- `$` es el punto de partida
- `@` es rock. No puedes caminar sobre eso.
- `%` es el cofre del tesoro

Muestra de un mapa:

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

@##############@@@@@######@#$@@@

@##############@@@@@######@###@@@

@########%######@############@@@

@############################@

@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
```

Dado que esos objetos tienen características similares, no es necesario crear un objeto separado para cada campo del mapa. Te mostraré cómo usar el peso mosca.

Definamos una interfaz que nuestros campos implementarán:

```
public interface IField
{
    string Name { get; }
    char Mark { get; }
    bool CanWalk { get; }
    FieldType Type { get; }
}
```

Ahora podemos crear clases que representen nuestros campos. También tenemos que identificarlos de alguna manera (usé una enumeración):

```
public enum FieldType
{
    GRASS,
    ROCK,
    START,
    CHEST
}
public class Grass : IField
{
    public string Name { get { return "Grass"; } }
    public char Mark { get { return '#'; } }
    public bool CanWalk { get { return true; } }
    public FieldType Type { get { return FieldType.GRASS; } }
}
public class StartingPoint : IField
{
    public string Name { get { return "Starting Point"; } }
    public char Mark { get { return '$'; } }
    public bool CanWalk { get { return true; } }
    public FieldType Type { get { return FieldType.START; } }
}
public class Rock : IField
{
    public string Name { get { return "Rock"; } }
    public char Mark { get { return '@'; } }
    public bool CanWalk { get { return false; } }
    public FieldType Type { get { return FieldType.ROCK; } }
}
public class TreasureChest : IField
{
    public string Name { get { return "Treasure Chest"; } }
    public char Mark { get { return '%'; } }
    public bool CanWalk { get { return true; } } // you can approach it
    public FieldType Type { get { return FieldType.CHEST; } }
}
```

Como dije, no necesitamos crear una instancia separada para cada campo. Tenemos que crear un **repositorio** de campos. La esencia de Flyweight DP es que creamos dinámicamente un objeto solo si lo necesitamos y aún no existe en nuestro repositorio, o lo devolvemos si ya existe. Escribamos una clase simple que manejará esto por nosotros:

```
public class FieldRepository
{
    private List<IField> lstFields = new List<IField>();

    private IField AddField(FieldType type)
    {
        IField f;
        switch(type)
        {
            case FieldType.GRASS: f = new Grass(); break;
            case FieldType.ROCK: f = new Rock(); break;
            case FieldType.START: f = new StartingPoint(); break;
            case FieldType.CHEST:
            default: f = new TreasureChest(); break;
        }
        lstFields.Add(f); //add it to repository
        Console.WriteLine("Created new instance of {0}", f.Name);
        return f;
```

```
    }
    public IField GetField(FieldType type)
    {
        IField f = lstFields.Find(x => x.Type == type);
        if (f != null) return f;
        else return AddField(type);
    }
}
```

¡Genial! Ahora podemos probar nuestro código:

```
public class Program
{
    public static void Main(string[] args)
    {
        FieldRepository f = new FieldRepository();
        IField grass = f.GetField(FieldType.GRASS);
        grass = f.GetField(FieldType.ROCK);
        grass = f.GetField(FieldType.GRASS);
    }
}
```

El resultado en la consola debe ser:

> Crea una nueva instancia de Grass.
>
> Crea una nueva instancia de rock.

Pero, ¿por qué el pasto aparece solo una vez si queremos hacerlo dos veces? Esto se debe a que la primera vez que llamamos a la instancia de `GetField` grass no existe en nuestro **repositorio** , por lo que se creó, pero la próxima vez que necesitamos Grass ya existe, por lo que solo la devolvemos.

Lea Implementando el patrón de diseño de peso mosca en línea:
https://riptutorial.com/es/csharp/topic/4619/implementando-el-patron-de-diseno-de-peso-mosca

# Capítulo 87: Implementando un patrón de diseño de decorador

## Observaciones

Ventajas de usar Decorador:

- Puedes agregar nuevas funcionalidades en tiempo de ejecución en diferentes configuraciones
- buena alternativa para la herencia
- el cliente puede elegir la configuración que quiere usar

## Examples

### Simulando cafeteria

Decorador es uno de los patrones de diseño estructural. Se utiliza para agregar, eliminar o cambiar el comportamiento del objeto. Este documento le enseñará cómo usar Decorator DP correctamente.

Déjame explicarte la idea de ello en un ejemplo simple. Imagina que ahora estás en Starbobs, famosa compañía de café. ¡Puede hacer un pedido de cualquier café que desee, con crema y azúcar, con crema y cobertura y muchas más combinaciones! Pero, la base de todas las bebidas es café: bebida oscura y amarga, que puede modificar. Vamos a escribir un programa simple que simule la máquina de café.

Primero, necesitamos crear una clase abstracta que describa nuestra bebida base:

```
public abstract class AbstractCoffee
{
    protected AbstractCoffee k = null;

    public AbstractCoffee(AbstractCoffee k)
    {
        this.k = k;
    }

    public abstract string ShowCoffee();
}
```

Ahora, vamos a crear algunos extras, como azúcar, leche y topping. Las clases creadas deben implementar `AbstractCoffee` - lo decorarán:

```
public class Milk : AbstractCoffee
{
    public Milk(AbstractCoffee c) : base(c) { }
    public override string ShowCoffee()
```

```
    {
        if (k != null)
            return k.ShowCoffee() + " with Milk";
        else return "Milk";
    }
}
public class Sugar : AbstractCoffee
{
    public Sugar(AbstractCoffee c) : base(c) { }

    public override string ShowCoffee()
    {
        if (k != null) return k.ShowCoffee() + " with Sugar";
        else return "Sugar";
    }
}
public class Topping : AbstractCoffee
{
    public Topping(AbstractCoffee c) : base(c) { }

    public override string ShowCoffee()
    {
        if (k != null) return k.ShowCoffee() + " with Topping";
        else return "Topping";
    }
}
```

Ahora podemos crear nuestro café favorito:

```
public class Program
{
    public static void Main(string[] args)
    {
        AbstractCoffee coffee = null; //we cant create instance of abstract class
        coffee = new Topping(coffee); //passing null
        coffee = new Sugar(coffee); //passing topping instance
        coffee = new Milk(coffee);  //passing sugar
        Console.WriteLine("Coffee with " + coffee.ShowCoffee());

    }
}
```

Ejecutar el código producirá el siguiente resultado:

> Café Con Topping Con Azúcar Con Leche

Lea Implementando un patrón de diseño de decorador en línea:
https://riptutorial.com/es/csharp/topic/4798/implementando-un-patron-de-diseno-de-decorador

# Capítulo 88: Importar contactos de Google

## Observaciones

Los datos de los contactos de los usuarios se recibirán en formato JSON, los extraemos y, finalmente, hacemos un bucle a través de estos datos y así obtenemos los contactos de Google.

## Examples

### Requerimientos

Para importar contactos de Google (Gmail) en la aplicación MVC de ASP.NET, primero descargue "Configuración de API de Google". Esto le otorgará las siguientes referencias:

```
using Google.Contacts;
using Google.GData.Client;
using Google.GData.Contacts;
using Google.GData.Extensions;
```

Añadir estos a la aplicación correspondiente.

### Código fuente en el controlador

```
using Google.Contacts;
using Google.GData.Client;
using Google.GData.Contacts;
using Google.GData.Extensions;
using Newtonsoft.Json;
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Net;
using System.Text;
using System.Web;
using System.Web.Mvc;

namespace GoogleContactImport.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            return View();
        }

        public ActionResult Import()
        {
            string clientId = "";  // here you need to add your google client id
            string redirectUrl = "http://localhost:1713/Home/AddGoogleContacts"; // here your
redirect action method NOTE: you need to configure same url in google console
            Response.Redirect("https://accounts.google.com/o/oauth2/auth?redirect_uri=" +
```

```
redirectUrl + "&amp&response_type=code&amp&client_id=" + clientId +
"&amp&scope=https://www.google.com/m8/feeds/&amp;approval_prompt=force&amp;access_type=offline");

            return View();
        }

        public ActionResult AddGoogleContacts()
        {
            string code = Request.QueryString["code"];
            if (!string.IsNullOrEmpty(code))
            {
                var contacts = GetAccessToken().ToArray();
                if (contacts.Length > 0)
                {
                  // You will get all contacts here
                    return View("Index",contacts);
                }
                else
                {
                    return RedirectToAction("Index","Home");
                }
            }
            else
            {
                return RedirectToAction("Index", "Home");
            }
        }
        public List<GmailContacts> GetAccessToken()
        {
            string code = Request.QueryString["code"];
            string google_client_id = ""; //your google client Id
            string google_client_sceret = ""; // your google secret key
            string google_redirect_url = "http://localhost:1713/MyContact/AddGoogleContacts";


            HttpWebRequest webRequest =
(HttpWebRequest)WebRequest.Create("https://accounts.google.com/o/oauth2/token");
            webRequest.Method = "POST";
            string parameters = "code=" + code + "&client_id=" + google_client_id +
"&client_secret=" + google_client_sceret + "&redirect_uri=" + google_redirect_url +
"&grant_type=authorization_code";
            byte[] byteArray = Encoding.UTF8.GetBytes(parameters);
            webRequest.ContentType = "application/x-www-form-urlencoded";
            webRequest.ContentLength = byteArray.Length;
            Stream postStream = webRequest.GetRequestStream();
            // Add the post data to the web request
            postStream.Write(byteArray, 0, byteArray.Length);
            postStream.Close();
            WebResponse response = webRequest.GetResponse();
            postStream = response.GetResponseStream();
            StreamReader reader = new StreamReader(postStream);
            string responseFromServer = reader.ReadToEnd();
            GooglePlusAccessToken serStatus =
JsonConvert.DeserializeObject<GooglePlusAccessToken>(responseFromServer);
            /*End*/
            return GetContacts(serStatus);
        }

        public List<GmailContacts> GetContacts(GooglePlusAccessToken serStatus)
        {
            string google_client_id = "";  //client id
```

```
            string google_client_sceret = ""; //secret key
            /*Get Google Contacts From Access Token and Refresh Token*/
            // string refreshToken = serStatus.refresh_token;
            string accessToken = serStatus.access_token;
            string scopes = "https://www.google.com/m8/feeds/contacts/default/full/";
            OAuth2Parameters oAuthparameters = new OAuth2Parameters()
            {
                ClientId = google_client_id,
                ClientSecret = google_client_sceret,
                RedirectUri = "http://localhost:1713/Home/AddGoogleContacts",
                Scope = scopes,
                AccessToken = accessToken,
                //  RefreshToken = refreshToken
            };

            RequestSettings settings = new RequestSettings("App Name", oAuthparameters);
            ContactsRequest cr = new ContactsRequest(settings);
            ContactsQuery query = new
ContactsQuery(ContactsQuery.CreateContactsUri("default"));
            query.NumberToRetrieve = 5000;
            Feed<Contact> ContactList = cr.GetContacts();

            List<GmailContacts> olist = new List<GmailContacts>();
            foreach (Contact contact in ContactList.Entries)
            {
                foreach (EMail email in contact.Emails)
                {
                    GmailContacts gc = new GmailContacts();
                    gc.EmailID = email.Address;
                    var a = contact.Name.FullName;
                    olist.Add(gc);
                }
            }
            return olist;
        }

        public class GmailContacts
        {
            public string EmailID
            {
                get { return _EmailID; }
                set { _EmailID = value; }
            }
            private string _EmailID;
        }

        public class GooglePlusAccessToken
        {

            public GooglePlusAccessToken()
            { }


            public string access_token
            {
                get { return _access_token; }
                set { _access_token = value; }
            }
            private string _access_token;

            public string token_type
```

```
            {
                get { return _token_type; }
                set { _token_type = value; }
            }
            private string _token_type;

            public string expires_in
            {
                get { return _expires_in; }
                set { _expires_in = value; }
            }
            private string _expires_in;

        }
    }
}
```

## Código fuente en la vista.

El único método de acción que necesita agregar es agregar un enlace de acción presente a continuación

```
<a href='@Url.Action("Import", "Home")'>Import Google Contacts</a>
```

Lea Importar contactos de Google en línea: https://riptutorial.com/es/csharp/topic/6744/importar-contactos-de-google

# Capítulo 89: Incluyendo recursos de fuentes

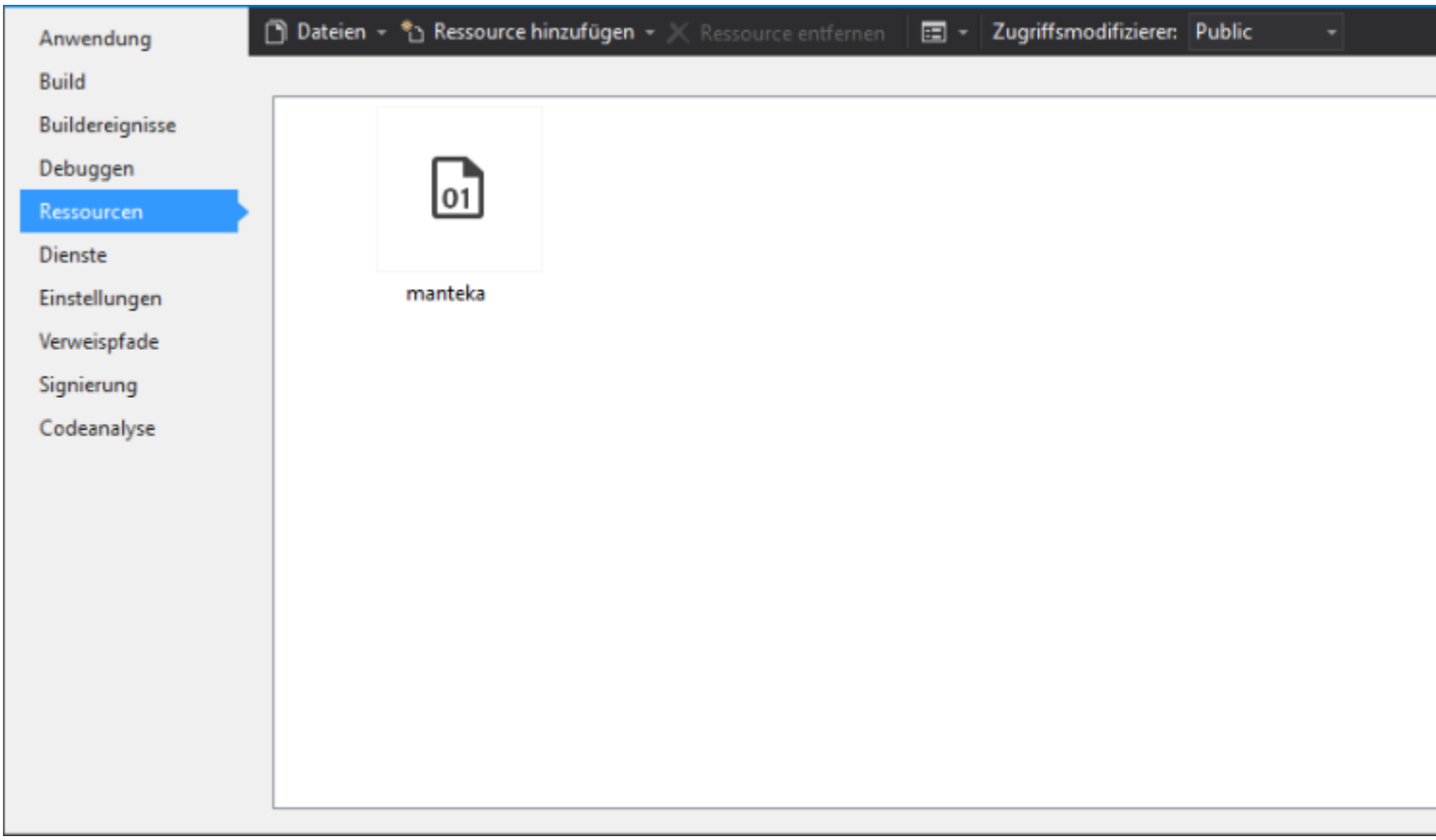
## Parámetros

| Parámetro | Detalles |
|---|---|
| fontbytes | byte array desde el binario .ttf |

## Examples

### Instancia 'Fontfamily' de los recursos

```
public FontFamily Maneteke = GetResourceFontFamily(Properties.Resources.manteka);
```




### Metodo de integracion

```
public static FontFamily GetResourceFontFamily(byte[] fontbytes)
{
    PrivateFontCollection pfc = new PrivateFontCollection();
    IntPtr fontMemPointer = Marshal.AllocCoTaskMem(fontbytes.Length);
    Marshal.Copy(fontbytes, 0, fontMemPointer, fontbytes.Length);
    pfc.AddMemoryFont(fontMemPointer, fontbytes.Length);
    Marshal.FreeCoTaskMem(fontMemPointer);
```

```
    return pfc.Families[0];
}
```

## Uso con un 'botón'

```
    public static class Res
    {
        /// <summary>
        /// URL: https://www.behance.net/gallery/2846011/Manteka
        /// </summary>
        public static FontFamily Maneteke =
GetResourceFontFamily(Properties.Resources.manteka);

        public static FontFamily GetResourceFontFamily(byte[] fontbytes)
        {
            PrivateFontCollection pfc = new PrivateFontCollection();
            IntPtr fontMemPointer = Marshal.AllocCoTaskMem(fontbytes.Length);
            Marshal.Copy(fontbytes, 0, fontMemPointer, fontbytes.Length);
            pfc.AddMemoryFont(fontMemPointer, fontbytes.Length);
            Marshal.FreeCoTaskMem(fontMemPointer);
            return pfc.Families[0];
        }
    }

    public class FlatButton : Button
    {
        public FlatButton() : base()
        {
            Font = new Font(Res.Maneteke, Font.Size);
        }

        protected override void OnFontChanged(EventArgs e)
        {
            base.OnFontChanged(e);
            this.Font = new Font(Res.Maneteke, this.Font.Size);
        }
    }
```

Lea Incluyendo recursos de fuentes en línea:
https://riptutorial.com/es/csharp/topic/9789/incluyendo-recursos-de-fuentes

# Capítulo 90: Incomparables

## Examples

### Ordenar versiones

**Clase:**

```
public class Version : IComparable<Version>
{
    public int[] Parts { get; }

    public Version(string value)
    {
        if (value == null)
            throw new ArgumentNullException();
        if (!Regex.IsMatch(value, @"^[0-9]+(\.[0-9]+)*$"))
            throw new ArgumentException("Invalid format");
        var parts = value.Split('.');
        Parts = new int[parts.Length];
        for (var i = 0; i < parts.Length; i++)
            Parts[i] = int.Parse(parts[i]);
    }

    public override string ToString()
    {
        return string.Join(".", Parts);
    }

    public int CompareTo(Version that)
    {
        if (that == null) return 1;
        var thisLength = this.Parts.Length;
        var thatLength = that.Parts.Length;
        var maxLength = Math.Max(thisLength, thatLength);
        for (var i = 0; i < maxLength; i++)
        {
            var thisPart = i < thisLength ? this.Parts[i] : 0;
            var thatPart = i < thatLength ? that.Parts[i] : 0;
            if (thisPart < thatPart) return -1;
            if (thisPart > thatPart) return 1;
        }
        return 0;
    }
}
```

**Prueba:**

```
Version a, b;

a = new Version("4.2.1");
b = new Version("4.2.6");
a.CompareTo(b); // a < b : -1

a = new Version("2.8.4");
```

```
b = new Version("2.8.0");
a.CompareTo(b); // a > b : 1

a = new Version("5.2");
b = null;
a.CompareTo(b); // a > b : 1

a = new Version("3");
b = new Version("3.6");
a.CompareTo(b); // a < b : -1

var versions = new List<Version>
{
    new Version("2.0"),
    new Version("1.1.5"),
    new Version("3.0.10"),
    new Version("1"),
    null,
    new Version("1.0.1")
};

versions.Sort();

foreach (var version in versions)
    Console.WriteLine(version?.ToString() ?? "NULL");
```

**Salida:**

NULO
1
1.0.1
1.1.5
2.0
3.0.10

**Manifestación:**

Demo en vivo en Ideone

Lea Incomparables en línea: https://riptutorial.com/es/csharp/topic/4222/incomparables

# Capítulo 91: Indexador

## Sintaxis

- retorno públicoEscriba este [índice de IndexType] {obtener {...} establecer {...}}

## Observaciones

El indexador permite que la sintaxis similar a una matriz acceda a una propiedad de un objeto con un índice.

- Se puede utilizar en una clase, estructura o interfaz.
- Puede ser sobrecargado.
- Puede utilizar múltiples parámetros.
- Puede ser utilizado para acceder y establecer valores.
- Puede usar cualquier tipo para su índice.

## Examples

### Un indexador simple

```
class Foo
{
    private string[] cities = new[] { "Paris", "London", "Berlin" };

    public string this[int index]
    {
        get {
            return cities[index];
        }
        set {
            cities[index] = value;
        }
    }
}
```

---

**Uso:**

```
    var foo = new Foo();

    // access a value
    string berlin = foo[2];

    // assign a value
    foo[0] = "Rome";
```

Ver demostración

## Indexador con 2 argumentos e interfaz.

```
interface ITable {
    // an indexer can be declared in an interface
    object this[int x, int y] { get; set; }
}

class DataTable : ITable
{
    private object[,] cells = new object[10, 10];

    /// <summary>
    /// implementation of the indexer declared in the interface
    /// </summary>
    /// <param name="x">X-Index</param>
    /// <param name="y">Y-Index</param>
    /// <returns>Content of this cell</returns>
    public object this[int x, int y]
    {
        get
        {
            return cells[x, y];
        }
        set
        {
            cells[x, y] = value;
        }
    }
}
```

## Sobrecargar el indexador para crear un SparseArray

Al sobrecargar el indexador, puede crear una clase que se vea y se sienta como una matriz pero no lo sea. Tendrá métodos O (1) de obtención y configuración, puede acceder a un elemento en el índice 100 y aún así tendrá el tamaño de los elementos dentro de él. La clase SparseArray

```
class SparseArray
    {
        Dictionary<int, string> array = new Dictionary<int, string>();

        public string this[int i]
        {
            get
            {
                if(!array.ContainsKey(i))
                {
                    return null;
                }
                return array[i];
            }
            set
            {
                if(!array.ContainsKey(i))
                    array.Add(i, value);
            }
        }
    }
```

Lea Indexador en línea: https://riptutorial.com/es/csharp/topic/1660/indexador

# Capítulo 92: Inicializadores de colección

## Observaciones

El único requisito para que un objeto se inicialice utilizando este azúcar sintáctico es que el tipo implemente `System.Collections.IEnumerable` y el método `Add` . Aunque lo llamamos un inicializador de colección, el objeto *no* tiene que ser una colección.

## Examples

### Inicializadores de colección

Inicializar un tipo de colección con valores:

```
var stringList = new List<string>
{
    "foo",
    "bar",
};
```

Los inicializadores de colección son azúcar sintáctica para llamadas `Add()` . El código anterior es equivalente a:

```
var temp = new List<string>();
temp.Add("foo");
temp.Add("bar");
var stringList = temp;
```

Tenga en cuenta que la inicialización se realiza de forma atómica utilizando una variable temporal, para evitar las condiciones de carrera.

Para los tipos que ofrecen múltiples parámetros en su método `Add()` , incluya los argumentos separados por comas entre llaves:

```
var numberDictionary = new Dictionary<int, string>
{
    { 1, "One" },
    { 2, "Two" },
};
```

Esto es equivalente a:

```
var temp = new Dictionary<int, string>();
temp.Add(1, "One");
temp.Add(2, "Two");
var numberDictionarynumberDictionary = temp;
```

### Inicializadores de índice C # 6

A partir de C # 6, las colecciones con indizadores se pueden inicializar especificando el índice a asignar entre corchetes, seguido de un signo igual, seguido del valor a asignar.

# Inicialización del diccionario

Un ejemplo de esta sintaxis usando un Diccionario:

```
var dict = new Dictionary<string, int>
{
    ["key1"] = 1,
    ["key2"] = 50
};
```

Esto es equivalente a:

```
var dict = new Dictionary<string, int>();
dict["key1"] = 1;
dict["key2"] = 50
```

La sintaxis de inicialización de colección para hacer esto antes de C # 6 era:

```
var dict = new Dictionary<string, int>
{
    { "key1", 1 },
    { "key2", 50 }
};
```

Que correspondería a:

```
var dict = new Dictionary<string, int>();
dict.Add("key1", 1);
dict.Add("key2", 50);
```

Por lo tanto, existe una diferencia significativa en la funcionalidad, ya que la nueva sintaxis utiliza el *indexador* del objeto inicializado para asignar valores en lugar de usar su método `Add()` . Esto significa que la nueva sintaxis solo requiere un indexador disponible públicamente y funciona para cualquier objeto que tenga uno.

```
public class IndexableClass
{
    public int this[int index]
    {
        set
        {
            Console.WriteLine("{0} was assigned to index {1}", value, index);
        }
    }
}
```

```
var foo = new IndexableClass
{
    [0] = 10,
    [1] = 20
}
```

Esto daría como resultado:

```
10 was assigned to index 0
20 was assigned to index 1
```

## Colección de inicializadores en clases personalizadas.

Para hacer que una clase sea compatible con los inicializadores de colección, debe implementar la interfaz `IEnumerable` y tener al menos un método `Add` . Desde C # 6, cualquier colección que implemente `IEnumerable` puede extenderse con métodos personalizados de `Add` utilizando métodos de extensión.

```
class Program
{
    static void Main()
    {
        var col = new MyCollection {
            "foo",
            { "bar", 3 },
            "baz",
            123.45d,
        };
    }
}

class MyCollection : IEnumerable
{
    private IList list = new ArrayList();

    public void Add(string item)
    {
        list.Add(item)
    }

    public void Add(string item, int count)
    {
        for(int i=0;i< count;i++) {
            list.Add(item);
        }
    }

    public IEnumerator GetEnumerator()
    {
        return list.GetEnumerator();
    }
}

static class MyCollectionExtensions
{
    public static void Add(this MyCollection @this, double value) =>
```

```
        @this.Add(value.ToString());
}
```

## Inicializadores de colección con matrices de parámetros

Puedes mezclar parámetros normales y matrices de parámetros:

```
public class LotteryTicket : IEnumerable{
    public int[] LuckyNumbers;
    public string UserName;

    public void Add(string userName, params int[] luckyNumbers){
        UserName = userName;
        Lottery = luckyNumbers;
    }
}
```

Esta sintaxis es ahora posible:

```
var Tickets = new List<LotteryTicket>{
    {"Mr Cool"  , 35663, 35732, 12312, 75685},
    {"Bruce"    , 26874, 66677, 24546, 36483, 46768, 24632, 24527},
    {"John Cena", 25446, 83356, 65536, 23783, 24567, 89337}
}
```

## Usando el inicializador de colección dentro del inicializador de objeto

```
public class Tag
{
    public IList<string> Synonyms { get; set; }
}
```

`Synonyms` es una propiedad de tipo colección. Cuando el objeto `Tag` se crea utilizando la sintaxis del inicializador de objetos, los `Synonyms` también se pueden inicializar con la sintaxis del inicializador de colecciones:

```
Tag t = new Tag
{
    Synonyms = new List<string> {"c#", "c-sharp"}
};
```

La propiedad de colección puede ser de solo lectura y aún así admite la sintaxis del inicializador de colección. Considere este ejemplo modificado (la propiedad `Synonyms` ahora tiene un setter privado):

```
public class Tag
{
    public Tag()
    {
        Synonyms = new List<string>();
    }
```

```
    public IList<string> Synonyms { get; private set; }
}
```

Un nuevo objeto `Tag` se puede crear así:

```
Tag t = new Tag
{
    Synonyms = {"c#", "c-sharp"}
};
```

Esto funciona porque los inicializadores de la colección son solo azúcar sintética sobre las llamadas a `Add()` . No se está creando una lista nueva aquí, el compilador está generando llamadas a `Add()` en el objeto que sale.

Lea Inicializadores de colección en línea: https://riptutorial.com/es/csharp/topic/21/inicializadores-de-coleccion

# Capítulo 93: Inicializadores de objetos

## Sintaxis

- SomeClass sc = new SomeClass {Property1 = value1, Property2 = value2, ...};
- SomeClass sc = new SomeClass (param1, param2, ...) {Property1 = value1, Property2 = value2, ...}

## Observaciones

Los paréntesis del constructor solo se pueden omitir si el tipo que se está creando tiene un constructor predeterminado (sin parámetros) disponible.

## Examples

### Uso simple

Los inicializadores de objetos son útiles cuando necesita crear un objeto y establecer un par de propiedades de inmediato, pero los constructores disponibles no son suficientes. Di que tienes una clase

```
public class Book
{
    public string Title { get; set; }
    public string Author { get; set; }

    // the rest of class definition
}
```

Para inicializar una nueva instancia de la clase con un inicializador:

```
Book theBook = new Book { Title = "Don Quixote", Author = "Miguel de Cervantes" };
```

Esto es equivalente a

```
Book theBook = new Book();
theBook.Title = "Don Quixote";
theBook.Author = "Miguel de Cervantes";
```

### Uso con tipos anónimos

Los inicializadores de objetos son la única forma de inicializar tipos anónimos, que son tipos generados por el compilador.

```
var album = new { Band = "Beatles", Title = "Abbey Road" };
```

Por esa razón, los inicializadores de objetos se utilizan ampliamente en las consultas de selección de LINQ, ya que proporcionan una manera conveniente de especificar qué partes de un objeto consultado le interesan.

```
var albumTitles = from a in albums
                  select new
                  {
                     Title = a.Title,
                     Artist = a.Band
                  };
```

## Uso con constructores no predeterminados

Puede combinar inicializadores de objetos con constructores para inicializar tipos si es necesario. Tomemos por ejemplo una clase definida como tal:

```
public class Book {
    public string Title { get; set; }
    public string Author { get; set; }

    public Book(int id) {
        //do things
    }

    // the rest of class definition
}

var someBook = new Book(16) { Title = "Don Quixote", Author = "Miguel de Cervantes" }
```

Esto primero creará una instancia de un `Book` con el constructor `Book(int)` , luego establecerá cada propiedad en el inicializador. Es equivalente a:

```
var someBook = new Book(16);
someBook.Title = "Don Quixote";
someBook.Author = "Miguel de Cervantes";
```

Lea Inicializadores de objetos en línea: https://riptutorial.com/es/csharp/topic/738/inicializadores-de-objetos

# Capítulo 94: Inicializando propiedades

## Observaciones

Cuando decida cómo crear una propiedad, comience con una propiedad auto-implementada por simplicidad y brevedad.

Cambie a una propiedad con un campo de respaldo solo cuando las circunstancias lo exijan. Si necesita otras manipulaciones más allá de un conjunto simple y obtener, puede que necesite introducir un campo de respaldo.

## Examples

### C # 6.0: Inicializar una propiedad auto-implementada

Cree una propiedad con getter y / o setter e inicialice todo en una línea:

```
public string Foobar { get; set; } = "xyz";
```

### Inicializando la propiedad con un campo de respaldo

```
public string Foobar {
    get { return _foobar; }
    set { _foobar = value; }
}
private string _foobar = "xyz";
```

### Inicializando propiedad en constructor

```
class Example
{
    public string Foobar { get; set; }
    public List<string> Names { get; set; }
    public Example()
    {
        Foobar = "xyz";
        Names = new List<string>(){"carrot","fox","ball"};
    }
}
```

### Inicialización de propiedades durante la instanciación de objetos

Las propiedades se pueden establecer cuando se crea una instancia de un objeto.

```
var redCar = new Car
{
    Wheels = 2,
```

```
    Year = 2016,
    Color = Color.Red
};
```

Lea Inicializando propiedades en línea: https://riptutorial.com/es/csharp/topic/82/inicializando-propiedades

# Capítulo 95: Inmutabilidad

## Examples

### Clase System.String

En C # (y .NET) una cadena está representada por la clase System.String. La palabra clave de `string` es un alias para esta clase.

La clase System.String es inmutable, es decir, una vez creado, su estado no puede alterarse.

Por lo tanto, todas las operaciones que realice en una cadena como Subcadena, Eliminar, Reemplazar, concatenación utilizando el operador `+` , etc. crearán una nueva cadena y la devolverán.

Vea el siguiente programa de demostración:

```
string str = "mystring";
string newString = str.Substring(3);
Console.WriteLine(newString);
Console.WriteLine(str);
```

Esto imprimirá `string` y `mystring` respectivamente.

### Cuerdas e inmutabilidad.

Los tipos inmutables son tipos que, cuando se modifican, crean una nueva versión del objeto en la memoria, en lugar de cambiar el objeto existente en la memoria. El ejemplo más simple de esto es el tipo de `string` incorporado.

Tomando el siguiente código, que agrega "mundo" a la palabra "Hola"

```
string myString = "hello";
myString += " world";
```

Lo que sucede en la memoria en este caso es que se crea un nuevo objeto cuando se agrega a la `string` en la segunda línea. Si hace esto como parte de un bucle grande, existe la posibilidad de que esto cause problemas de rendimiento en su aplicación.

El equivalente mutable para una `string` es un `StringBuilder`

Tomando el siguiente código

```
StringBuilder myStringBuilder = new StringBuilder("hello");
myStringBuilder.append(" world");
```

Cuando ejecuta esto, está modificando el objeto `StringBuilder` en la memoria.

Lea Inmutabilidad en línea: https://riptutorial.com/es/csharp/topic/1863/inmutabilidad

# Capítulo 96: Interfaces

## Examples

### Implementando una interfaz

Se utiliza una interfaz para imponer la presencia de un método en cualquier clase que lo "implementa". La interfaz se define con la palabra clave `interface` y una clase puede "implementarla" agregando `: InterfaceName` después del nombre de la clase. Una clase puede implementar múltiples interfaces al separar cada interfaz con una coma.
`: InterfaceName, ISecondInterface`

```
public interface INoiseMaker
{
    string MakeNoise();
}

public class Cat : INoiseMaker
{
    public string MakeNoise()
    {
        return "Nyan";
    }
}

public class Dog : INoiseMaker
{
    public string MakeNoise()
    {
        return "Woof";
    }
}
```

Debido a que implementan `INoiseMaker` , tanto el `cat` como el `dog` deben incluir el `string MakeNoise()` y no podrán compilarse sin él.

### Implementando multiples interfaces

```
public interface IAnimal
{
    string Name { get; set; }
}

public interface INoiseMaker
{
    string MakeNoise();
}

public class Cat : IAnimal, INoiseMaker
{
    public Cat()
    {
        Name = "Cat";
```

```
    }

    public string Name { get; set; }

    public string MakeNoise()
    {
        return "Nyan";
    }
}
```

## Implementación de interfaz explícita

La implementación explícita de la interfaz es necesaria cuando implementa varias interfaces que definen un método común, pero se requieren diferentes implementaciones dependiendo de qué interfaz se está utilizando para llamar al método (tenga en cuenta que no necesita implementaciones explícitas si varias interfaces comparten el mismo método y una implementación común es posible).

```
interface IChauffeur
{
    string Drive();
}

interface IGolfPlayer
{
    string Drive();
}

class GolfingChauffeur : IChauffeur, IGolfPlayer
{
    public string Drive()
    {
        return "Vroom!";
    }

    string IGolfPlayer.Drive()
    {
        return "Took a swing...";
    }
}


GolfingChauffeur obj = new GolfingChauffeur();
IChauffeur chauffeur = obj;
IGolfPlayer golfer = obj;

Console.WriteLine(obj.Drive()); // Vroom!
Console.WriteLine(chauffeur.Drive()); // Vroom!
Console.WriteLine(golfer.Drive()); // Took a swing...
```

La implementación no se puede llamar desde ningún otro lugar excepto mediante la interfaz:

```
public class Golfer : IGolfPlayer
{
    string IGolfPlayer.Drive()
    {
```

```
        return "Swinging hard...";
    }
    public void Swing()
    {
        Drive(); // Compiler error: No such method
    }
}
```

Debido a esto, puede ser ventajoso colocar el código de implementación complejo de una interfaz implementada explícitamente en un método privado y separado.

Por supuesto, una implementación de interfaz explícita solo se puede usar para los métodos que realmente existen para esa interfaz:

```
public class ProGolfer : IGolfPlayer
{
    string IGolfPlayer.Swear() // Error
    {
        return "The ball is in the pit";
    }
}
```

De manera similar, el uso de una implementación de interfaz explícita sin declarar esa interfaz en la clase también causa un error.

# Insinuación:

La implementación de interfaces explícitamente también se puede utilizar para evitar el código muerto. Cuando ya no se necesita un método y se elimina de la interfaz, el compilador se quejará de cada implementación existente.

# Nota:

Los programadores esperan que el contrato sea el mismo independientemente del contexto del tipo y la implementación explícita no debe exponer un comportamiento diferente cuando se llama. Así que, a diferencia del ejemplo anterior, `IGolfPlayer.Drive` y `Drive` deberían hacer lo mismo cuando sea posible.

### Por qué usamos interfaces

Una interfaz es una definición de un contrato entre el usuario de la interfaz y la clase que la implementa. Una forma de pensar en una interfaz es como una declaración de que un objeto puede realizar ciertas funciones.

Digamos que definimos una interfaz `IShape` para representar diferentes tipos de formas, esperamos que una forma tenga un área, así que definiremos un método para forzar a las implementaciones de la interfaz a devolver su área:

```
public interface IShape
{
    double ComputeArea();
}
```

Supongamos que tenemos las siguientes dos formas: un `Rectangle` y un `Circle`

```
public class Rectangle : IShape
{
    private double length;
    private double width;

    public Rectangle(double length, double width)
    {
        this.length = length;
        this.width = width;
    }

    public double ComputeArea()
    {
        return length * width;
    }
}

public class Circle : IShape
{
    private double radius;

    public Circle(double radius)
    {
        this.radius = radius;
    }

    public double ComputeArea()
    {
        return Math.Pow(radius, 2.0) * Math.PI;
    }
}
```

Cada uno de ellos tiene su propia definición de su área, pero ambos son formas. Así que es lógico verlos como `IShape` en nuestro programa:

```
private static void Main(string[] args)
{
    var shapes = new List<IShape>() { new Rectangle(5, 10), new Circle(5) };
    ComputeArea(shapes);

    Console.ReadKey();
}

private static void ComputeArea(IEnumerable<IShape> shapes)
{
    foreach (shape in shapes)
    {
        Console.WriteLine("Area: {0:N}, shape.ComputeArea());
    }
}
```

```
// Output:
// Area : 50.00
// Area : 78.54
```

## Fundamentos de la interfaz

Una función de interfaz conocida como un "contrato" de funcionalidad. Significa que declara propiedades y métodos pero no los implementa.

Así que a diferencia de las clases de interfaces:

- No puede ser instanciado
- No puedo tener ninguna funcionalidad.
- Solo puede contener métodos * *(Las propiedades y los eventos son métodos internamente)*
- La herencia de una interfaz se llama "Implementación"
- Puede heredar de 1 clase, pero puede "Implementar" múltiples interfaces

```
public interface ICanDoThis{
    void TheThingICanDo();
    int SomeValueProperty { get; set; }
}
```

Cosas para notar:

- El prefijo "I" es una convención de nomenclatura utilizada para las interfaces.
- El cuerpo de la función se sustituye por un punto y coma ";".
- Las propiedades también están permitidas porque internamente también son métodos.

```
public class MyClass : ICanDoThis {
    public void TheThingICanDo(){
        // do the thing
    }

    public int SomeValueProperty { get; set; }
    public int SomeValueNotImplemtingAnything { get; set; }
}
```

.

```
ICanDoThis obj = new MyClass();

// ok
obj.TheThingICanDo();

// ok
obj.SomeValueProperty = 5;

// Error, this member doesn't exist in the interface
obj.SomeValueNotImplemtingAnything = 5;

// in order to access the property in the class you must "down cast" it
((MyClass)obj).SomeValueNotImplemtingAnything = 5; // ok
```

Esto es especialmente útil cuando se trabaja con marcos de IU como WinForms o WPF porque es obligatorio heredar de una clase base para crear control de usuario y perder la capacidad de crear abstracción sobre diferentes tipos de control. ¿Un ejemplo? Subiendo:

```
public class MyTextBlock : TextBlock {
    public void SetText(string str){
        this.Text = str;
    }
}

public class MyButton : Button {
    public void SetText(string str){
        this.Content = str;
    }
}
```

El problema propuesto es que ambos contienen algún concepto de "Texto" pero los nombres de las propiedades difieren. Y no puede crear crear una *clase base abstracta* porque tienen una herencia obligatoria de 2 clases diferentes. Una interfaz puede aliviar eso

```
public interface ITextControl{
    void SetText(string str);
}

public class MyTextBlock : TextBlock, ITextControl {
    public void SetText(string str){
        this.Text = str;
    }
}

public class MyButton : Button, ITextControl {
    public void SetText(string str){
        this.Content = str;
    }

    public int Clicks { get; set; }
}
```

Ahora MyButton y MyTextBlock son intercambiables.

```
var controls = new List<ITextControls>{
    new MyTextBlock(),
    new MyButton()
};

foreach(var ctrl in controls){
    ctrl.SetText("This text will be applied to both controls despite them being different");


    // Compiler Error, no such member in interface
    ctrl.Clicks = 0;

    // Runtime Error because 1 class is in fact not a button which makes this cast invalid
    ((MyButton)ctrl).Clicks = 0;


    /* the solution is to check the type first.
```

```
    This is usually considered bad practice since
    it's a symptom of poor abstraction */
    var button = ctrl as MyButton;
    if(button != null)
        button.Clicks = 0; // no errors


}
```

## Miembros "ocultos" con implementación explícita

¿No odias que las interfaces contaminen tu clase con demasiados miembros que ni siquiera te importan? Bueno tengo una solución! Implementaciones explícitas

```
public interface IMessageService {
    void OnMessageRecieve();
    void SendMessage();
    string Result { get; set; }
    int Encoding { get; set; }
    // yadda yadda
}
```

Normalmente implementarías la clase de esta manera.

```
public class MyObjectWithMessages : IMessageService {
     public void OnMessageRecieve(){

     }

     public void SendMessage(){

     }

     public string Result { get; set; }
     public int Encoding { get; set; }
}
```

Cada miembro es público.

```
var obj = new MyObjectWithMessages();

// why would i want to call this function?
obj.OnMessageRecieve();
```

Respuesta: Yo no. Así que tampoco se debe declarar público, sino que simplemente declarar a los miembros como privados hará que el compilador arroje un error.

La solución es usar implementación explícita:

```
public class MyObjectWithMessages : IMessageService{
    void IMessageService.OnMessageRecieve() {

    }
```

```
    void IMessageService.SendMessage() {

    }

    string IMessageService.Result { get; set; }
    int IMessageService.Encoding { get; set; }
}
```

Así que ahora ha implementado los miembros según lo requerido y no expondrán a ningún miembro como público.

```
var obj = new MyObjectWithMessages();

/* error member does not exist on type MyObjectWithMessages.
 * We've succesfully made it "private" */
obj.OnMessageRecieve();
```

Si de verdad quieres seguir accediendo al miembro, aunque está implementado explícitamente, todo lo que tienes que hacer es enviar el objeto a la interfaz y listo.

```
((IMessageService)obj).OnMessageRecieve();
```

## Incomparables Como ejemplo de implementación de una interfaz

Las interfaces pueden parecer abstractas hasta que las parezcas en la práctica. `IComparable` e `IComparable<T>` son excelentes ejemplos de por qué las interfaces pueden ser útiles para nosotros.

Digamos que en un programa para una tienda en línea, tenemos una variedad de artículos que puedes comprar. Cada artículo tiene un nombre, un número de identificación y un precio.

```
public class Item {

    public string name; // though public variables are generally bad practice,
    public int idNumber; // to keep this example simple we will use them instead
    public decimal price; // of a property.

    // body omitted for brevity

}
```

Tenemos nuestros `Item` almacenados dentro de una `List<Item>` , y en nuestro programa en algún lugar, queremos ordenar nuestra lista por número de identificación de menor a mayor. En lugar de escribir nuestro propio algoritmo de clasificación, podemos usar el método `Sort()` que `List<T>` ya tiene. Sin embargo, como nuestra clase de `Item` es ahora, no hay forma de que la `List<T>` entienda el orden en que se ordenará la lista. Aquí es donde entra en `IComparable` interfaz de `IComparable` .

Para implementar correctamente el método `CompareTo` , `CompareTo` debe devolver un número positivo si el parámetro es "menor que" el actual, cero si son iguales y un número negativo si el parámetro es "mayor que".

```
Item apple = new Item();
```

```
apple.idNumber = 15;
Item banana = new Item();
banana.idNumber = 4;
Item cow = new Item();
cow.idNumber = 15;
Item diamond = new Item();
diamond.idNumber = 18;

Console.WriteLine(apple.CompareTo(banana)); // 11
Console.WriteLine(apple.CompareTo(cow)); // 0
Console.WriteLine(apple.CompareTo(diamond)); // -3
```

Aquí está la implementación de la interfaz del `Item` ejemplo:

```
public class Item : IComparable<Item> {

    private string name;
    private int idNumber;
    private decimal price;

    public int CompareTo(Item otherItem) {

        return (this.idNumber - otherItem.idNumber);

    }

    // rest of code omitted for brevity

}
```

En un nivel de superficie, el método `CompareTo` en nuestro artículo simplemente devuelve la diferencia en sus números de identificación, pero ¿qué hace lo anterior en la práctica?

Ahora, cuando llamamos a `Sort()` en un objeto `List<Item>` , la `List` llamará automáticamente al método `CompareTo` del `Item` cuando necesite determinar en qué orden colocar los objetos. Además, además de `List<T>` , cualquier otro objeto la necesidad de la capacidad de comparar dos objetos funcionará con el `Item` porque hemos definido la capacidad de comparar dos `Item` diferentes entre sí.

Lea Interfaces en línea: https://riptutorial.com/es/csharp/topic/2208/interfaces

# Capítulo 97: Interfaz IDisponible

## Observaciones

- `IDisposable` de los clientes de la clase que implementan `IDisposable` asegurarse de que llamen al método `Dispose` cuando hayan terminado de usar el objeto. No hay nada en el CLR que busque directamente los objetos para invocar un método `Dispose` .
- No es necesario implementar un finalizador si su objeto solo contiene recursos administrados. Asegúrese de llamar a `Dispose` en todos los objetos que usa su clase cuando implemente su propio método `Dispose` .
- Se recomienda hacer que la clase sea segura contra múltiples llamadas a `Dispose` , aunque lo ideal sería que solo se llame una vez. Esto se puede lograr agregando una variable `private bool` a su clase y estableciendo el valor en `true` cuando se haya ejecutado el método `Dispose` .

## Examples

### En una clase que contiene solo recursos gestionados

Los recursos administrados son recursos que el recolector de basura del tiempo de ejecución conoce y tiene bajo control. Hay muchas clases disponibles en el BCL, por ejemplo, como una `SqlConnection` que es una clase de contenedor para un recurso no administrado. Estas clases ya implementan la interfaz `IDisposable` : depende de su código limpiarlas cuando haya terminado.

No es necesario implementar un finalizador si su clase solo contiene recursos administrados.

```
public class ObjectWithManagedResourcesOnly : IDisposable
{
    private SqlConnection sqlConnection = new SqlConnection();

    public void Dispose()
    {
        sqlConnection.Dispose();
    }
}
```

### En una clase con recursos gestionados y no gestionados.

Es importante dejar que la finalización ignore los recursos administrados. El finalizador se ejecuta en otro subproceso: es posible que los objetos gestionados ya no existan en el momento en que se ejecuta el finalizador. La implementación de un método protegido de `Dispose(bool)` es una práctica común para garantizar que los recursos administrados no tengan su método `Dispose` llamado desde un finalizador.

```
public class ManagedAndUnmanagedObject : IDisposable
```

```
{
    private SqlConnection sqlConnection = new SqlConnection();
    private UnmanagedHandle unmanagedHandle = Win32.SomeUnmanagedResource();
    private bool disposed;

    public void Dispose()
    {
        Dispose(true); // client called dispose
        GC.SuppressFinalize(this); // tell the GC to not execute the Finalizer
    }

    protected virtual void Dispose(bool disposeManaged)
    {
        if (!disposed)
        {
            if (disposeManaged)
            {
                if (sqlConnection != null)
                {
                    sqlConnection.Dispose();
                }
            }

            unmanagedHandle.Release();

            disposed = true;
        }
    }

    ~ManagedAndUnmanagedObject()
    {
        Dispose(false);
    }
}
```

## IDisponible, Disponer

.NET Framework define una interfaz para los tipos que requieren un método de desmontaje:

```
public interface IDisposable
{
  void Dispose();
}
```

`Dispose()` se usa principalmente para limpiar recursos, como referencias no administradas. Sin embargo, también puede ser útil forzar la disposición de otros recursos incluso si se administran. En lugar de esperar a que el GC finalmente limpie también la conexión de su base de datos, puede asegurarse de que se haga en su propia implementación de `Dispose()` .

```
public void Dispose()
{
   if (null != this.CurrentDatabaseConnection)
   {
       this.CurrentDatabaseConnection.Dispose();
       this.CurrentDatabaseConnection = null;
   }
}
```

Cuando necesite acceder directamente a recursos no administrados, como punteros no administrados o recursos win32, cree una clase `SafeHandle` de `SafeHandle` y use las convenciones / herramientas de esa clase para hacerlo.

### En una clase heredada con recursos gestionados

Es bastante común que pueda crear una clase que implemente `IDisposable` , y luego derivar clases que también contengan recursos administrados. Se recomienda marcar el método de `Dispose` con la palabra clave `virtual` para que los clientes puedan limpiar los recursos que posean.

```
public class Parent : IDisposable
{
    private ManagedResource parentManagedResource = new ManagedResource();

    public virtual void Dispose()
    {
        if (parentManagedResource != null)
        {
            parentManagedResource.Dispose();
        }
    }
}

public class Child : Parent
{
    private ManagedResource childManagedResource = new ManagedResource();

    public override void Dispose()
    {
        if (childManagedResource != null)
        {
            childManagedResource.Dispose();
        }
        //clean up the parent's resources
        base.Dispose();
    }
}
```

### usando palabras clave

Cuando un objeto implementa la interfaz `IDisposable` , se puede crear dentro de la sintaxis de `using` :

```
using (var foo = new Foo())
{
    // do foo stuff
} // when it reaches here foo.Dispose() will get called

public class Foo : IDisposable
{
    public void Dispose()
    {
        Console.WriteLine("dispose called");
    }
}
```

Ver demo

`using` azúcar sintáctica es para un bloque `try/finally` ; el uso anterior se traduciría aproximadamente en:

```
{
    var foo = new Foo();
    try
    {
        // do foo stuff
    }
    finally
    {
        if (foo != null)
            ((IDisposable)foo).Dispose();
    }
}
```

Lea Interfaz IDisponible en línea: https://riptutorial.com/es/csharp/topic/1795/interfaz-idisponible

# Capítulo 98: Interfaz INotifyPropertyChanged

## Observaciones

La interfaz `INotifyPropertyChanged` es necesaria siempre que necesite hacer que su clase informe los cambios que ocurren en sus propiedades. La interfaz define un único evento `PropertyChanged` .

Con el enlace XAML, el evento `PropertyChanged` se conecta automáticamente, por lo que solo necesita implementar la interfaz INotifyPropertyChanged en su modelo de vista o clases de contexto de datos para trabajar con el enlace XAML.

## Examples

### Implementando INotifyPropertyChanged en C # 6

La implementación de `INotifyPropertyChange` puede ser propensa a errores, ya que la interfaz requiere especificar el nombre de la propiedad como una cadena. Para hacer que la implementación sea más robusta, se puede usar un atributo `CallerMemberName` .

```
class C : INotifyPropertyChanged
{
    // backing field
    int offset;
    // property
    public int Offset
    {
        get
        {
            return offset;
        }
        set
        {
            if (offset == value)
                return;
            offset = value;
            RaisePropertyChanged();
        }
    }

    // helper method for raising PropertyChanged event
    void RaisePropertyChanged([CallerMemberName] string propertyName = null) =>
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));

    // interface implemetation
    public event PropertyChangedEventHandler PropertyChanged;
}
```

Si tiene varias clases que implementan `INotifyPropertyChanged` , puede que le resulte útil refactorizar la implementación de la interfaz y el método auxiliar a la clase base común:

```
class NotifyPropertyChangedImpl : INotifyPropertyChanged
```

```
{
    protected void RaisePropertyChanged([CallerMemberName] string propertyName = null) =>
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));

    // interface implemetation
    public event PropertyChangedEventHandler PropertyChanged;
}

class C : NotifyPropertyChangedImpl
{
    int offset;
    public int Offset
    {
        get { return offset; }
        set { if (offset != value) { offset = value; RaisePropertyChanged(); } }
    }
}
```

## INotifyPropertyChanged con método de conjunto genérico

La clase `NotifyPropertyChangedBase` continuación define un método de conjunto genérico al que se puede llamar desde cualquier tipo derivado.

```
public class NotifyPropertyChangedBase : INotifyPropertyChanged
{
    protected void RaisePropertyChanged([CallerMemberName] string propertyName = null) =>
    PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));

    public event PropertyChangedEventHandler PropertyChanged;

    public virtual bool Set<T>(ref T field, T value, [CallerMemberName] string propertyName =
null)
    {
        if (Equals(field, value))
            return false;
        storage = value;
        RaisePropertyChanged(propertyName);
        return true;
    }
}
```

Para usar este método de set genérico, simplemente necesita crear una clase que derive de NotifyPropertyChangedBase.

```
public class SomeViewModel : NotifyPropertyChangedBase
{
    private string _foo;
    private int _bar;

    public string Foo
    {
        get { return _foo; }
        set { Set(ref _foo, value); }
    }

    public int Bar
    {
```

```
        get { return _bar; }
        set { Set(ref _bar, value); }
    }
}
```

Como se muestra arriba, puede llamar a `Set(ref _fieldName, value);` en el establecedor de una propiedad y automáticamente generará un evento PropertyChanged si es necesario.

Luego, puede registrarse en el evento PropertyChanged de otra clase que necesita para manejar los cambios de propiedad.

```
public class SomeListener
{
    public SomeListener()
    {
        _vm = new SomeViewModel();
        _vm.PropertyChanged += OnViewModelPropertyChanged;
    }

    private void OnViewModelPropertyChanged(object sender, PropertyChangedEventArgs e)
    {
        Console.WriteLine($"Property {e.PropertyName} was changed.");
    }

    private readonly SomeViewModel _vm;

}
```

Lea Interfaz INotifyPropertyChanged en línea: https://riptutorial.com/es/csharp/topic/2990/interfaz-inotifypropertychanged

# Capítulo 99: Interfaz IQueryable

## Examples

### Traducir una consulta LINQ a una consulta SQL

Las interfaces `IQueryable` e `IQueryable<T>` permiten a los desarrolladores traducir una consulta LINQ (una consulta 'integrada en el idioma') a una fuente de datos específica, por ejemplo, una base de datos relacional. Tome esta consulta LINQ escrita en C #:

```
var query = from book in books
            where book.Author == "Stephen King"
            select book;
```

Si los `books` variables son de un tipo que implementa `IQueryable<Book>` , la consulta anterior se pasa al proveedor (establecido en la propiedad `IQueryable.Provider` ) en forma de árbol de expresión, una estructura de datos que refleja la estructura del código. .

El proveedor puede inspeccionar el árbol de expresiones en tiempo de ejecución para determinar:

- que hay un predicado para la propiedad `Author` de la clase `Book` ;
- que el método de comparación utilizado es 'igual a' ( `==` );
- que el valor que debe igualar es `"Stephen King"` .

Con esta información, el proveedor puede traducir la consulta de C # a una consulta de SQL en tiempo de ejecución y pasar la consulta a una base de datos relacional para obtener solo los libros que coinciden con el predicado:

```
select *
from Books
where Author = 'Stephen King'
```

Se llama al proveedor cuando la variable de `query` se itera sobre ( `IEnumerable` implementa `IQueryable` ).

(El proveedor utilizado en este ejemplo requeriría algunos metadatos adicionales para saber qué tabla consultar y cómo hacer coincidir las propiedades de la clase C # con las columnas de la tabla, pero dichos metadatos están fuera del alcance de la interfaz `IQueryable` ).

Lea Interfaz IQueryable en línea: https://riptutorial.com/es/csharp/topic/3094/interfaz-iqueryable

# Capítulo 100: Interoperabilidad

## Observaciones

**Trabajando con la API de Win32 usando C #**

Windows expone muchas funcionalidades en forma de API de Win32. Usando estas API, puede realizar operaciones directas en Windows, lo que aumenta el rendimiento de su aplicación. Fuente Haga clic aquí

Windows expone una amplia gama de API. Para obtener información sobre varias API, puede consultar sitios como Pinvoke .

## Examples

### Función de importación desde DLL de C ++ no administrado

Este es un ejemplo de cómo importar una función que está definida en una DLL de C ++ no administrada. En el código fuente de C ++ para "myDLL.dll", la función `add` está definida:

```
extern "C" __declspec(dllexport) int __stdcall add(int a, int b)
{
    return a + b;
}
```

Luego se puede incluir en un programa de C # de la siguiente manera:

```
class Program
{
    // This line will import the C++ method.
    // The name specified in the DllImport attribute must be the DLL name.
    // The names of parameters are unimportant, but the types must be correct.
    [DllImport("myDLL.dll")]
    private static extern int add(int left, int right);

    static void Main(string[] args)
    {
        //The extern method can be called just as any other C# method.
        Console.WriteLine(add(1, 2));
    }
}
```

Consulte Convenciones de llamadas y el nombre de C ++ para obtener explicaciones sobre por qué son necesarias `extern "C"` y `__stdcall` .

# Encontrando la librería dinámica.

Cuando se invoca por primera vez el método externo, el programa C # buscará y cargará la DLL adecuada. Para obtener más información sobre dónde se busca el DLL, y cómo puede influir en las ubicaciones de búsqueda, consulte esta pregunta de stackoverflow .

## Código simple para exponer la clase para com

```
using System;
using System.Runtime.InteropServices;

namespace ComLibrary
{
    [ComVisible(true)]
    public interface IMainType
    {
        int GetInt();

        void StartTime();

        int StopTime();
    }

    [ComVisible(true)]
    [ClassInterface(ClassInterfaceType.None)]
    public class MainType : IMainType
    {
        private Stopwatch stopWatch;

        public int GetInt()
        {
            return 0;
        }

        public void StartTime()
        {
            stopWatch= new Stopwatch();
            stopWatch.Start();
        }

        public int StopTime()
        {
            return (int)stopWatch.ElapsedMilliseconds;
        }
    }
}
```

## Nombre en C ++

Los compiladores de C ++ codifican información adicional en los nombres de las funciones exportadas, como los tipos de argumentos, para hacer posible las sobrecargas con diferentes argumentos. Este proceso se llama mutilación de nombres . Esto causa problemas con la importación de funciones en C # (y la interoperabilidad con otros lenguajes en general), ya que el nombre de `int add(int a, int b)` ya no se `add` , puede ser `?add@@YAHHH@Z` , `_add@8` o cualquier otra cosa, dependiendo del compilador y la convención de llamada.

Hay varias formas de resolver el problema de la manipulación de nombres:

- Exportación de funciones usando `extern "C"` para cambiar a enlace externo C que usa la denominación de nombres C:

```
extern "C" __declspec(dllexport) int __stdcall add(int a, int b)
```

```
[DllImport("myDLL.dll")]
```

El nombre de la función seguirá siendo `_add@8` ( `_add@8` ), pero el compilador de C # reconoce el nombre de la función `extern "C" StdCall` + `extern "C"` .

- Especificando nombres de funciones exportadas en el archivo de definición de módulo `myDLL.def` :

```
EXPORTS
  add
```

```
int __stdcall add(int a, int b)
```

```
[DllImport("myDLL.dll")]
```

El nombre de la función será puro `add` en este caso.

- Importando nombre destrozado. Necesitará algún visor de DLL para ver el nombre mutilado, luego puede especificarlo explícitamente:

```
__declspec(dllexport) int __stdcall add(int a, int b)
```

```
[DllImport("myDLL.dll", EntryPoint = "?add@@YGHHH@Z")]
```

## Convenciones de llamadas

Existen varias convenciones sobre las funciones de llamada, que especifican quién (el llamante o el que recibe la llamada) saca los argumentos de la pila, cómo se pasan los argumentos y en qué orden. C ++ usa la convención de llamadas `Cdecl` de forma predeterminada, pero C # espera `StdCall` , que generalmente es utilizado por la API de Windows. Necesitas cambiar uno u otro:

- Cambie la convención de llamada a `StdCall` en C ++:

```
extern "C" __declspec(dllexport) int __stdcall add(int a, int b)
```

```
[DllImport("myDLL.dll")]
```

- O bien, cambie la convención de llamada a `Cdecl` en C #:

```
extern "C" __declspec(dllexport) int /*__cdecl*/ add(int a, int b)
```

```
[DllImport("myDLL.dll", CallingConvention = CallingConvention.Cdecl)]
```

Si desea usar una función con la convención de llamadas `Cdecl` y un nombre mutilado, su código se verá así:

```
__declspec(dllexport) int add(int a, int b)
```

```
[DllImport("myDLL.dll", CallingConvention = CallingConvention.Cdecl,
           EntryPoint = "?add@@YAHHH@Z")]
```

- **thiscall** ( **__thiscall** ) se usa principalmente en funciones que son miembros de una clase.
- Cuando una función usa **thiscall** ( **__thiscall** ), un puntero a la clase se pasa como primer parámetro.

## Carga y descarga dinámica de archivos DLL no administrados

Cuando use el atributo `DllImport` , debe saber el dll correcto y el nombre del método en el *momento de la compilación* . Si desea ser más flexible y decidir en *tiempo* de *ejecución* qué dll y qué métodos cargar, puede usar los métodos de la API de Windows `LoadLibrary()` , `GetProcAddress()` y `FreeLibrary()` . Esto puede ser útil si la biblioteca a usar depende de las condiciones de tiempo de ejecución.

El puntero devuelto por `GetProcAddress()` puede convertir en un delegado utilizando `Marshal.GetDelegateForFunctionPointer()` .

El siguiente ejemplo de código lo demuestra con el `myDLL.dll` de los ejemplos anteriores:

```
class Program
{
    // import necessary API as shown in other examples
    [DllImport("kernel32.dll", SetLastError = true)]
    public static extern IntPtr LoadLibrary(string lib);
    [DllImport("kernel32.dll", SetLastError = true)]
    public static extern void FreeLibrary(IntPtr module);
    [DllImport("kernel32.dll", SetLastError = true)]
    public static extern IntPtr GetProcAddress(IntPtr module, string proc);

    // declare a delegate with the required signature
    private delegate int AddDelegate(int a, int b);

    private static void Main()
    {
        // load the dll
        IntPtr module = LoadLibrary("myDLL.dll");
        if (module == IntPtr.Zero) // error handling
        {
            Console.WriteLine($"Could not load library: {Marshal.GetLastWin32Error()}");
            return;
        }

        // get a "pointer" to the method
        IntPtr method = GetProcAddress(module, "add");
```

```
        if (method == IntPtr.Zero) // error handling
        {
            Console.WriteLine($"Could not load method: {Marshal.GetLastWin32Error()}");
            FreeLibrary(module);  // unload library
            return;
        }

        // convert "pointer" to delegate
        AddDelegate add = (AddDelegate)Marshal.GetDelegateForFunctionPointer(method,
typeof(AddDelegate));

        // use function
        int result = add(750, 300);

        // unload library
        FreeLibrary(module);
    }
}
```

## Tratar con los errores de Win32

Al usar métodos de interoperabilidad, puede usar la API **GetLastError** para obtener información adicional sobre sus llamadas a la API.

**Atributo DllImport Atributo SetLastError**

*SetLastError = true*

Indica que el destinatario llamará a SetLastError (función de API de Win32).

*SetLastError = false*

Indica que la persona **que** llama **no** llamará a SetLastError (función de la API de Win32), por lo tanto, no obtendrá información de error.

- Cuando SetLastError no está establecido, se establece en falso (valor predeterminado).
- Puede obtener el código de error utilizando el método Marshal.GetLastWin32Error:

*Ejemplo:*

```
[DllImport("kernel32.dll", SetLastError=true)]
public static extern IntPtr OpenMutex(uint access, bool handle, string lpName);
```

Si intenta abrir la exclusión mutua que no existe, GetLastError devolverá **ERROR_FILE_NOT_FOUND** .

```
var lastErrorCode = Marshal.GetLastWin32Error();

if (lastErrorCode == (uint)ERROR_FILE_NOT_FOUND)
{
    //Deal with error
}
```

Los códigos de error del sistema se pueden encontrar aquí:

https://msdn.microsoft.com/en-us/library/windows/desktop/ms681382(v=vs.85).aspx

**API GetLastError**

Existe una API nativa de **GetLastError** que también puede utilizar:

```
[DllImport("coredll.dll", SetLastError=true)]
static extern Int32 GetLastError();
```

- Al llamar a la API de Win32 desde el código administrado, siempre debe usar **Marshal.GetLastWin32Error** .

Este es el por qué:

Entre su llamada a Win32 que establece el error (llamadas a SetLastError), el CLR puede llamar a otras llamadas de Win32 que también podrían llamar a **SetLastError** , este comportamiento puede anular su valor de error. En este escenario, si llama a **GetLastError** , puede obtener un error no válido.

Al establecer **SetLastError = true** , se asegura de que el CLR recupera el código de error antes de ejecutar otras llamadas de Win32.

## Objeto fijado

**GC** (recolector de basura) es responsable de limpiar nuestra basura.

Mientras **GC** limpia nuestra basura, elimina los objetos no utilizados del montón administrado que causa la fragmentación del montón. Cuando **GC** termina con la eliminación, realiza una compresión del montón (desfragmentación) que implica mover objetos en el montón.

Dado que **GC** no es determinista, al pasar la referencia / puntero del objeto administrado al código nativo, **GC** puede activarse en cualquier momento, si ocurre justo después de la llamada Inerop, existe una gran posibilidad de que el objeto (referencia que se pasa al nativo) ser movido en el montón administrado - como resultado, obtenemos una referencia no válida en el lado administrado.

En este escenario, debe **fijar** el objeto antes de pasarlo al código nativo.

**Objeto fijado**

El objeto fijado es un objeto que no está permitido mover por GC.

**Mango de Gc Pinned**

Puedes crear un objeto pin usando el método **Gc.Alloc**

```
GCHandle handle = GCHandle.Alloc(yourObject, GCHandleType.Pinned);
```

- La obtención de un **GCHandle anclado** a un objeto administrado marca un objeto específico como uno que **GC** no puede mover hasta que se libera el controlador.

Ejemplo:

```
[DllImport("kernel32.dll", SetLastError = true)]
public static extern void EnterCriticalSection(IntPtr ptr);

[DllImport("kernel32.dll", SetLastError = true)]
public static extern void LeaveCriticalSection(IntPtr ptr);

public void EnterCriticalSection(CRITICAL_SECTION section)
{
    try
    {
        GCHandle handle = GCHandle.Alloc(section, GCHandleType.Pinned);
        EnterCriticalSection(handle.AddrOfPinnedObject());
        //Do Some Critical Work
        LeaveCriticalSection(handle.AddrOfPinnedObject());
    }
    finaly
    {
        handle.Free()
    }
}
```

**Precauciones**

- Al fijar (especialmente los objetos grandes) el objeto, intente liberar el **GcHandle anclado lo** más rápido posible, ya que interrumpe la desfragmentación del montón.
- Si olvidas liberar **GcHandle** nada lo hará. Hágalo en una sección de código de seguridad (como finaly)

## Estructuras de lectura con mariscal

La clase Marshal contiene una función llamada **PtrToStructure** , esta función nos permite leer estructuras mediante un puntero no administrado.

**La** función **PtrToStructure** tiene muchas sobrecargas, pero todas tienen la misma intención.

**PtrToStructure** genérico:

```
public static T PtrToStructure<T>(IntPtr ptr);
```

*T* - tipo de estructura.

*ptr* : un puntero a un bloque de memoria no administrado.

Ejemplo:

```
NATIVE_STRUCT result = Marshal.PtrToStructure<NATIVE_STRUCT>(ptr);
```

- Si trata con objetos gestionados mientras lee estructuras nativas, no olvide fijar su objeto :)

```
T Read<T>(byte[] buffer)
   {
       T result = default(T);

       var gch = GCHandle.Alloc(buffer, GCHandleType.Pinned);

       try
       {
           result = Marshal.PtrToStructure<T>(gch.AddrOfPinnedObject());
       }
       finally
       {
           gch.Free();
       }

       return result;
   }
```

Lea Interoperabilidad en línea: https://riptutorial.com/es/csharp/topic/3278/interoperabilidad

# Capítulo 101: Interpolación de cuerdas

## Sintaxis

- $ "contenido {expresión} contenido"
- $ "contenido {expresión: formato} contenido"
- $ "contenido {expresión} {{contenido entre llaves}} contenido}"
- $ "contenido {expresión: formato} {{contenido entre llaves}} contenido}"

## Observaciones

La interpolación de cadenas es una forma abreviada del método `string.Format()` que facilita la creación de cadenas con valores de variables y expresiones dentro de ellas.

```
var name = "World";
var oldWay = string.Format("Hello, {0}!", name);  // returns "Hello, World"
var newWay = $"Hello, {name}!";                   // returns "Hello, World"
```

## Examples

### Expresiones

Las expresiones completas también se pueden utilizar en cadenas interpoladas.

```
var StrWithMathExpression = $"1 + 2 = {1 + 2}"; // -> "1 + 2 = 3"

string world = "world";
var StrWithFunctionCall = $"Hello, {world.ToUpper()}!"; // -> "Hello, WORLD!"
```

Demo en vivo en .NET Fiddle

### Formato de fechas en cadenas

```
var date = new DateTime(2015, 11, 11);
var str = $"It's {date:MMMM d, yyyy}, make a wish!";
System.Console.WriteLine(str);
```

También puede usar el método `DateTime.ToString` para formatear el objeto `DateTime` . Esto producirá la misma salida que el código anterior.

```
var date = new DateTime(2015, 11, 11);
var str = date.ToString("MMMM d, yyyy");
str = "It's " + str + ", make a wish!";
Console.WriteLine(str);
```

**Salida:**

Es el 11 de noviembre de 2015, pide un deseo!

Demo en vivo en .NET Fiddle

Demo en vivo usando DateTime.ToString

> **Nota:** `MM` significa meses y `mm` para minutos. Tenga mucho cuidado al usar estos, ya que los errores pueden introducir errores que pueden ser difíciles de descubrir.

### Uso simple

```
var name = "World";
var str = $"Hello, {name}!";
//str now contains: "Hello, World!";
```

## Entre bastidores

Internamente esto

```
$"Hello, {name}!"
```

Se compilará a algo como esto:

```
string.Format("Hello, {0}!", name);
```

### Relleno de la salida

La cadena se puede formatear para aceptar un parámetro de relleno que especificará cuántas posiciones de caracteres utilizará la cadena insertada:

```
${value, padding}
```

> **NOTA:** Los valores de relleno positivos indican el relleno izquierdo y los valores de relleno negativos indican el relleno derecho.

## Relleno izquierdo

Un relleno izquierdo de 5 (agrega 3 espacios antes del valor del número, por lo que ocupa un total de 5 posiciones de caracteres en la cadena resultante).

```
var number = 42;
var str = $"The answer to life, the universe and everything is {number, 5}.";
//str is "The answer to life, the universe and everything is    42.";
//                                                          ^^^^^
System.Console.WriteLine(str);
```

**Salida:**

```
The answer to life, the universe and everything is    42.
```

Demo en vivo en .NET Fiddle

# Relleno derecho

El relleno derecho, que utiliza un valor de relleno negativo, agregará espacios al final del valor actual.

```
var number = 42;
var str = $"The answer to life, the universe and everything is ${number, -5}.";
//str is "The answer to life, the universe and everything is 42    .";
//                                                          ^^^^^
System.Console.WriteLine(str);
```

**Salida:**

```
The answer to life, the universe and everything is 42    .
```

Demo en vivo en .NET Fiddle

# Relleno con especificadores de formato

También puede utilizar los especificadores de formato existentes junto con el relleno.

```
var number = 42;
var str = $"The answer to life, the universe and everything is ${number, 5:f1}";
//str is "The answer to life, the universe and everything is 42.1 ";
//                                                          ^^^^^
```

Demo en vivo en .NET Fiddle

## Formateo de números en cadenas

Puede usar dos puntos y la sintaxis de formato numérico estándar para controlar cómo se formatean los números.

```
var decimalValue = 120.5;

var asCurrency = $"It costs {decimalValue:C}";
// String value is "It costs $120.50" (depending on your local currency settings)

var withThreeDecimalPlaces = $"Exactly {decimalValue:F3}";
// String value is "Exactly 120.500"

var integerValue = 57;

var prefixedIfNecessary = $"{integerValue:D5}";
// String value is "00057"
```

Demo en vivo en .NET Fiddle

Lea Interpolación de cuerdas en línea: https://riptutorial.com/es/csharp/topic/22/interpolacion-de-cuerdas

# Capítulo 102: Inyección de dependencia

## Observaciones

La definición de Wikipedia de inyección de dependencia es:

En ingeniería de software, la inyección de dependencia es un patrón de diseño de software que implementa la inversión de control para resolver dependencias. Una dependencia es un objeto que se puede utilizar (un servicio). Una inyección es el paso de una dependencia a un objeto dependiente (un cliente) que lo usaría.

** **Este sitio presenta una respuesta a la pregunta Cómo explicar la inyección de dependencia a un niño de 5 años. La respuesta mejor calificada, proporcionada por John Munsch, proporciona una analogía sorprendentemente precisa dirigida al inquisidor (imaginario) de cinco años de edad: cuando va y saca cosas del refrigerador para usted mismo, puede causar problemas. Podría dejar la puerta abierta, podría obtener algo que mamá o papá no quieren que usted tenga. Puede que incluso esté buscando algo que ni siquiera tenemos o que haya caducado. Lo que debe hacer es declarar una necesidad: "Necesito algo de beber con el almuerzo", y luego nos aseguraremos de que tenga algo cuando se siente a comer. Lo que esto significa en términos de desarrollo de software orientado a objetos es esto: las clases colaboradoras (los niños de cinco años) deben confiar en la infraestructura (los padres) para proporcionar**

** Este código usa MEF para cargar dinámicamente la dll y resolver las dependencias. La dependencia de ILogger es resuelta por MEF e inyectada en la clase de usuario. La clase de usuario nunca recibe la implementación concreta de ILogger y no tiene idea de qué o qué tipo de registrador está utilizando. **

## Examples

### Inyección de dependencia mediante MEF

```
public interface ILogger
{
    void Log(string message);
}

[Export(typeof(ILogger))]
[ExportMetadata("Name", "Console")]
public class ConsoleLogger:ILogger
{
    public void Log(string message)
    {
        Console.WriteLine(message);
    }
}

[Export(typeof(ILogger))]
```

```
[ExportMetadata("Name", "File")]
public class FileLogger:ILogger
{
    public void Log(string message)
    {
        //Write the message to file
    }
}

public class User
{
    private readonly ILogger logger;
    public User(ILogger logger)
    {
        this.logger = logger;
    }
    public void LogUser(string message)
    {
        logger.Log(message)  ;
    }
}

public interface ILoggerMetaData
{
    string Name { get; }
}

internal class Program
{
    private CompositionContainer _container;

    [ImportMany]
    private IEnumerable<Lazy<ILogger, ILoggerMetaData>> _loggers;

    private static void Main()
    {
        ComposeLoggers();
        Lazy<ILogger, ILoggerMetaData> loggerNameAndLoggerMapping = _ loggers.First((n) =>
((n.Metadata.Name.ToUpper() =="Console"));
        ILogger logger= loggerNameAndLoggerMapping.Value
        var user = new User(logger);
        user.LogUser("user name");
    }

    private void ComposeLoggers()
    {
        //An aggregate catalog that combines multiple catalogs
        var catalog = new AggregateCatalog();
        string loggersDllDirectory =Path.Combine(Utilities.GetApplicationDirectory(),
"Loggers");
        if (!Directory.Exists(loggersDllDirectory ))
        {
            Directory.CreateDirectory(loggersDllDirectory );
        }
        //Adds all the parts found in the same assembly as the PluginManager class
        catalog.Catalogs.Add(new AssemblyCatalog(typeof(Program).Assembly));
        catalog.Catalogs.Add(new DirectoryCatalog(loggersDllDirectory ));

        //Create the CompositionContainer with the parts in the catalog
        _container = new CompositionContainer(catalog);
```

```
        //Fill the imports of this object
        try
        {
            this._container.ComposeParts(this);
        }
        catch (CompositionException compositionException)
        {
            throw new CompositionException(compositionException.Message);
        }
    }
}
```

## Inyección de dependencia C # y ASP.NET con Unity

Primero, ¿por qué deberíamos usar la inyección de dependencia en nuestro código? Queremos desacoplar otros componentes de otras clases en nuestro programa. Por ejemplo, tenemos la clase AnimalController que tiene un código como este:

```
public class AnimalController()
{
    private SantaAndHisReindeer _SantaAndHisReindeer = new SantaAndHisReindeer();

    public AnimalController(){
            Console.WriteLine("");
    }
}
```

Miramos este código y creemos que todo está bien, pero ahora nuestro AnimalController depende del objeto _SantaAndHisReindeer. Automáticamente mi controlador es malo para las pruebas y la reutilización de mi código será muy difícil.

Muy buena explicación de por qué deberíamos usar Depedency Injection e interfaces aquí .

Si queremos que Unity maneje DI, el camino para lograrlo es muy simple :) Con NuGet (administrador de paquetes) podemos importar fácilmente la unidad a nuestro código.

> en Visual Studio Tools -> NuGet Package Manager -> Administrar paquetes para la solución -> en la entrada de búsqueda escriba unidad -> elija nuestro proyecto -> haga clic en instalar

Ahora se crearán dos archivos con buenos comentarios.

> en la carpeta de datos de aplicación UnityConfig.cs y UnityMvcActivator.cs
>
> UnityConfig - en el método RegisterTypes, podemos ver el tipo que se inyectará en nuestros constructores.

```
namespace Vegan.WebUi.App_Start
{

public class UnityConfig
{
    #region Unity Container
```

```
    private static Lazy<IUnityContainer> container = new Lazy<IUnityContainer>(() =>
    {
        var container = new UnityContainer();
        RegisterTypes(container);
        return container;
    });

    /// <summary>
    /// Gets the configured Unity container.
    /// </summary>
    public static IUnityContainer GetConfiguredContainer()
    {
        return container.Value;
    }
    #endregion

    /// <summary>Registers the type mappings with the Unity container.</summary>
    /// <param name="container">The unity container to configure.</param>
    /// <remarks>There is no need to register concrete types such as controllers or API
controllers (unless you want to
    /// change the defaults), as Unity allows resolving a concrete type even if it was not
previously registered.</remarks>
    public static void RegisterTypes(IUnityContainer container)
    {
        // NOTE: To load from web.config uncomment the line below. Make sure to add a
Microsoft.Practices.Unity.Configuration to the using statements.
        // container.LoadConfiguration();

        // TODO: Register your types here
        // container.RegisterType<IProductRepository, ProductRepository>();

        container.RegisterType<ISanta, SantaAndHisReindeer>();

     }
 }
}
```

UnityMvcActivator -> también con buenos comentarios que dicen que esta clase integra Unity con ASP.NET MVC

```
using System.Linq;
using System.Web.Mvc;
using Microsoft.Practices.Unity.Mvc;

[assembly:
WebActivatorEx.PreApplicationStartMethod(typeof(Vegan.WebUi.App_Start.UnityWebActivator),
"Start")]
[assembly:
WebActivatorEx.ApplicationShutdownMethod(typeof(Vegan.WebUi.App_Start.UnityWebActivator),
"Shutdown")]

namespace Vegan.WebUi.App_Start
{
/// <summary>Provides the bootstrapping for integrating Unity with ASP.NET MVC.</summary>
public static class UnityWebActivator
{
    /// <summary>Integrates Unity when the application starts.</summary>
    public static void Start()
    {
        var container = UnityConfig.GetConfiguredContainer();
```

```
FilterProviders.Providers.Remove(FilterProviders.Providers.OfType<FilterAttributeFilterProvider>().Firs

        FilterProviders.Providers.Add(new UnityFilterAttributeFilterProvider(container));

        DependencyResolver.SetResolver(new UnityDependencyResolver(container));

        // TODO: Uncomment if you want to use PerRequestLifetimeManager
        //
Microsoft.Web.Infrastructure.DynamicModuleHelper.DynamicModuleUtility.RegisterModule(typeof(UnityPerRe

    }

    /// <summary>Disposes the Unity container when the application is shut down.</summary>
    public static void Shutdown()
    {
        var container = UnityConfig.GetConfiguredContainer();
        container.Dispose();
    }
}
}
```

Ahora podemos desacoplar nuestro controlador de la clase SantAndHisReindeer :)

```
 public class AnimalController()
    {
        private readonly SantaAndHisReindeer _SantaAndHisReindeer;

        public AnimalController(SantaAndHisReindeer SantaAndHisReindeer){

                _SantAndHisReindeer = SantaAndHisReindeer;
        }
    }
```

Hay una cosa final que debemos hacer antes de ejecutar nuestra aplicación.

En Global.asax.cs debemos agregar una nueva línea: UnityWebActivator.Start () que iniciará, configurará Unity y registrará nuestros tipos.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Optimization;
using System.Web.Routing;
using Vegan.WebUi.App_Start;

namespace Vegan.WebUi
{
    public class MvcApplication : System.Web.HttpApplication
    {
        protected void Application_Start()
        {
            AreaRegistration.RegisterAllAreas();
            FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
            RouteConfig.RegisterRoutes(RouteTable.Routes);
```

```
            BundleConfig.RegisterBundles(BundleTable.Bundles);
            UnityWebActivator.Start();
        }
    }
}
```

Lea Inyección de dependencia en línea: https://riptutorial.com/es/csharp/topic/5766/inyeccion-de-dependencia

# Capítulo 103: Iteradores

## Observaciones

Un iterador es un método, acceso u operador que realiza una iteración personalizada sobre una matriz o clase de colección utilizando la palabra clave yield

## Examples

### Ejemplo de iterador numérico simple

Un caso de uso común para los iteradores es realizar alguna operación sobre una colección de números. El siguiente ejemplo muestra cómo cada elemento dentro de una matriz de números puede imprimirse individualmente en la consola.

Esto es posible porque los arreglos implementan la interfaz `IEnumerable` , permitiendo a los clientes obtener un iterador para el arreglo utilizando el método `GetEnumerator()` . Este método devuelve un *enumerador* , que es un cursor de solo lectura y solo de avance sobre cada número de la matriz.

```
int[] numbers = { 1, 2, 3, 4, 5 };

IEnumerator iterator = numbers.GetEnumerator();

while (iterator.MoveNext())
{
    Console.WriteLine(iterator.Current);
}
```

**Salida**

```
1
2
3
4
5
```

También es posible lograr los mismos resultados utilizando una declaración `foreach` :

```
foreach (int number in numbers)
{
    Console.WriteLine(number);
}
```

### Creando iteradores usando el rendimiento

Los iteradores *producen* enumeradores. En C #, los enumeradores se producen mediante la definición de métodos, propiedades o indizadores que contienen declaraciones de `yield` .

La mayoría de los métodos devolverán el control a su interlocutor a través de declaraciones de `return` normales, lo que elimina todo el estado local de ese método. Por el contrario, los métodos que utilizan declaraciones de `yield` les permiten devolver múltiples valores al llamante a petición, al tiempo que *conservan* el estado local entre estos valores. Estos valores devueltos constituyen una secuencia. Hay dos tipos de declaraciones de `yield` utilizadas dentro de los iteradores:

- `yield return` , que devuelve el control a la persona que llama pero conserva el estado. La persona que llama continuará la ejecución desde esta línea cuando se le devuelva el control.
- `yield break` , que funciona de manera similar a una declaración de `return` normal, esto significa el final de la secuencia. Las declaraciones de `return` normales en sí mismas son ilegales dentro de un bloque iterador.

Este ejemplo a continuación demuestra un método de iterador que se puede usar para generar la secuencia de Fibonacci :

```
IEnumerable<int> Fibonacci(int count)
{
    int prev = 1;
    int curr = 1;

    for (int i = 0; i < count; i++)
    {
        yield return prev;
        int temp = prev + curr;
        prev = curr;
        curr = temp;
    }
}
```

Este iterador se puede usar para producir un enumerador de la secuencia de Fibonacci que puede ser consumido por un método de llamada. El siguiente código muestra cómo se pueden enumerar los primeros diez términos dentro de la secuencia de Fibonacci:

```
void Main()
{
    foreach (int term in Fibonacci(10))
    {
        Console.WriteLine(term);
    }
}
```

**Salida**

```
1
1
2
3
5
8
13
21
```

```
34
55
```

Lea Iteradores en línea: https://riptutorial.com/es/csharp/topic/4243/iteradores

# Capítulo 104: Leer y entender Stacktraces

## Introducción

Un seguimiento de pila es una gran ayuda cuando se depura un programa. Obtendrá un seguimiento de la pila cuando su programa lanza una Excepción, y algunas veces cuando el programa termina de forma anormal.

## Examples

### Rastreo de pila para una simple NullReferenceException en formularios Windows Forms

Vamos a crear un pequeño fragmento de código que lanza una excepción:

```
private void button1_Click(object sender, EventArgs e)
{
    string msg = null;
    msg.ToCharArray();
}
```

Si ejecutamos esto, obtenemos la siguiente excepción y el seguimiento de pila:

```
System.NullReferenceException: "Object reference not set to an instance of an object."
   at WindowsFormsApplication1.Form1.button1_Click(Object sender, EventArgs e) in
F:\WindowsFormsApplication1\WindowsFormsApplication1\Form1.cs:line 29
   at System.Windows.Forms.Control.OnClick(EventArgs e)
   at System.Windows.Forms.Button.OnClick(EventArgs e)
   at System.Windows.Forms.Button.OnMouseUp(MouseEventArgs mevent)
```

La traza de pila continúa así, pero esta parte será suficiente para nuestros propósitos.

En la parte superior de la traza de la pila vemos la línea:

> en WindowsFormsApplication1.Form1.button1_Click (Object sender, EventArgs e) en F: \ WindowsFormsApplication1 \ WindowsFormsApplication1 \ Form1.cs: línea 29

Esta es la parte más importante. Nos dice la línea *exacta* donde ocurrió la excepción: línea 29 en Form1.cs.
Entonces, aquí es donde empiezas tu búsqueda.

La segunda linea es

> en System.Windows.Forms.Control.OnClick (EventArgs e)

Este es el método que llamamos `button1_Click` . Así que ahora sabemos que se `button1_Click` , donde ocurrió el error, desde `System.Windows.Forms.Control.OnClick` .

Podemos continuar así; la tercera linea es

> en System.Windows.Forms.Button.OnClick (EventArgs e)

Este es, a su vez, el código que se llama `System.windows.Forms.Control.OnClick` .

El seguimiento de la pila es la lista de funciones a las que se llamó hasta que su código encontró la excepción. ¡Y al seguir esto, puede averiguar qué ruta de ejecución siguió su código hasta que se encontró con problemas!

Tenga en cuenta que el seguimiento de la pila incluye llamadas del sistema .Net; Normalmente, no es necesario seguir todos los códigos del sistema Microsofts `System.Windows.Forms` para averiguar qué salió mal, solo el código que pertenece a su propia aplicación.

Entonces, ¿por qué esto se llama un "seguimiento de pila"?
Porque, cada vez que un programa llama a un método, realiza un seguimiento de dónde estaba.
Tiene una estructura de datos llamada "pila", donde descarga su última ubicación.
Si ha terminado de ejecutar el método, busca en la pila para ver dónde estaba antes de que llamara al método, y continúa desde allí.

Así que la pila le permite a la computadora saber dónde se quedó, antes de llamar a un nuevo método.

Pero también sirve como una ayuda de depuración. Al igual que un detective que rastrea los pasos que tomó un criminal al cometer su delito, un programador puede usar la pila para rastrear los pasos que tomó un programa antes de que fallara.

Lea Leer y entender Stacktraces en línea: https://riptutorial.com/es/csharp/topic/8923/leer-y-entender-stacktraces

# Capítulo 105: Leyendo y escribiendo archivos .zip

## Sintaxis

1. ZipArchive OpenRead estático público (cadena archiveFileName)

## Parámetros

| Parámetro | Detalles |
|---|---|
| archiveFileName | La ruta al archivo para abrir, especificada como una ruta relativa o absoluta. Una ruta relativa se interpreta como relativa al directorio de trabajo actual. |

## Examples

### Escribir en un archivo zip

Para escribir un nuevo archivo .zip:

```
System.IO.Compression
System.IO.Compression.FileSystem

using (FileStream zipToOpen = new FileStream(@"C:\temp", FileMode.Open))
{
    using (ZipArchive archive = new ZipArchive(zipToOpen, ZipArchiveMode.Update))
    {
        ZipArchiveEntry readmeEntry = archive.CreateEntry("Readme.txt");
        using (StreamWriter writer = new StreamWriter(readmeEntry.Open()))
        {
            writer.WriteLine("Information about this package.");
            writer.WriteLine("========================");
        }
    }
}
```

### Escribir archivos zip en la memoria

El siguiente ejemplo devolverá los datos de `byte[]` de un archivo comprimido que contiene los archivos provistos, sin necesidad de acceder al sistema de archivos.

```
public static byte[] ZipFiles(Dictionary<string, byte[]> files)
{
    using (MemoryStream ms = new MemoryStream())
    {
```

```
        using (ZipArchive archive = new ZipArchive(ms, ZipArchiveMode.Update))
        {
            foreach (var file in files)
            {
                ZipArchiveEntry orderEntry = archive.CreateEntry(file.Key); //create a file
with this name
                using (BinaryWriter writer = new BinaryWriter(orderEntry.Open()))
                {
                    writer.Write(file.Value); //write the binary data
                }
            }
        }
        //ZipArchive must be disposed before the MemoryStream has data
        return ms.ToArray();
    }
}
```

## Obtener archivos de un archivo Zip

Este ejemplo obtiene una lista de archivos de los datos binarios del archivo zip proporcionado:

```
public static Dictionary<string, byte[]> GetFiles(byte[] zippedFile)
{
    using (MemoryStream ms = new MemoryStream(zippedFile))
    using (ZipArchive archive = new ZipArchive(ms, ZipArchiveMode.Read))
    {
        return archive.Entries.ToDictionary(x => x.FullName, x => ReadStream(x.Open()));
    }
}

private static byte[] ReadStream(Stream stream)
{
    using (var ms = new MemoryStream())
    {
        stream.CopyTo(ms);
        return ms.ToArray();
    }
}
```

## El siguiente ejemplo muestra cómo abrir un archivo zip y extraer todos los archivos .txt a una carpeta

```
using System;
using System.IO;
using System.IO.Compression;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            string zipPath = @"c:\example\start.zip";
            string extractPath = @"c:\example\extract";

            using (ZipArchive archive = ZipFile.OpenRead(zipPath))
            {
```

```
                foreach (ZipArchiveEntry entry in archive.Entries)
                {
                    if (entry.FullName.EndsWith(".txt", StringComparison.OrdinalIgnoreCase))
                    {
                        entry.ExtractToFile(Path.Combine(extractPath, entry.FullName));
                    }
                }
            }
        }
    }
}
```

Lea Leyendo y escribiendo archivos .zip en línea:
https://riptutorial.com/es/csharp/topic/6709/leyendo-y-escribiendo-archivos--zip

# Capítulo 106: Linq a los objetos

## Introducción

LINQ to Objects se refiere al uso de consultas LINQ con cualquier colección IEnumerable.

## Examples

### Cómo LINQ to Object ejecuta las consultas

Las consultas LINQ no se ejecutan inmediatamente. Cuando está creando la consulta, simplemente está almacenando la consulta para la ejecución futura. Solo cuando realmente solicita iterar la consulta se ejecuta la consulta (por ejemplo, en un bucle for, al llamar a ToList, Count, Max, Average, First, etc.)

Esto se considera *ejecución diferida* . Esto le permite crear la consulta en varios pasos, posiblemente modificándola según las declaraciones condicionales, y luego ejecutarla solo una vez que requiera el resultado.

Dado el código:

```
var query = from n in numbers
            where n % 2 != 0
            select n;
```

El ejemplo anterior solo almacena la consulta en la variable de `query` . No ejecuta la consulta en sí.

La instrucción `foreach` fuerza la ejecución de la consulta:

```
foreach(var n in query) {
    Console.WriteLine($"Number selected {n}");
}
```

Algunos métodos LINQ también activarán la ejecución de la consulta, `Count` , `First` , `Max` , `Average` . Devuelven valores únicos. `ToList` y `ToArray` recopilan resultados y los convierten en una Lista o un Array, respectivamente.

Tenga en cuenta que es posible iterar en la consulta varias veces si llama a varias funciones LINQ en la misma consulta. Esto podría darle diferentes resultados en cada llamada. Si solo desea trabajar con un conjunto de datos, asegúrese de guardarlo en una lista o matriz.

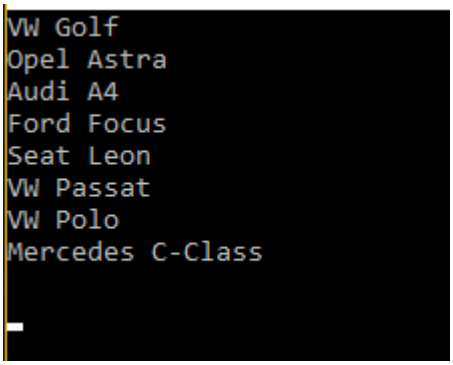
### Usando LINQ para objetos en C #

**Una simple consulta SELECCIONAR en Linq**

```
static void Main(string[] args)
{
    string[] cars = { "VW Golf",
                      "Opel Astra",
                      "Audi A4",
                      "Ford Focus",
                      "Seat Leon",
                      "VW Passat",
                      "VW Polo",
                      "Mercedes C-Class" };

    var list = from car in cars
               select car;

    StringBuilder sb = new StringBuilder();

    foreach (string entry in list)
    {
        sb.Append(entry + "\n");
    }

    Console.WriteLine(sb.ToString());
    Console.ReadLine();
}
```

En el ejemplo anterior, una matriz de cadenas (autos) se utiliza como una colección de objetos a consultar usando LINQ. En una consulta LINQ, la cláusula from viene primero para introducir la fuente de datos (autos) y la variable de rango (car). Cuando se ejecuta la consulta, la variable de rango servirá como referencia para cada elemento sucesivo en los autos. Debido a que el compilador puede inferir el tipo de auto, no tiene que especificarlo explícitamente

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

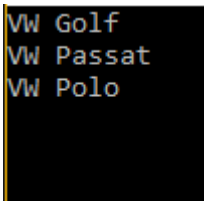



**SELECCIONE con una cláusula WHERE**

```
var list = from car in cars
           where car.Contains("VW")
           select car;
```

La cláusula WHERE se utiliza para consultar la matriz de cadenas (coches) para buscar y devolver un subconjunto de la matriz que satisfaga la cláusula WHERE.

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:




### Generando una lista ordenada

```
var list = from car in cars
           orderby car ascending
           select car;
```

A veces es útil ordenar los datos devueltos. La cláusula orderby hará que los elementos se ordenen según el comparador predeterminado para el tipo que se ordena.

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

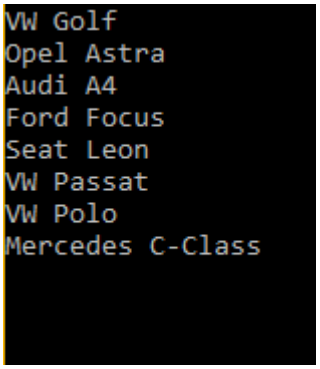

### Trabajando con un tipo personalizado

En este ejemplo, se crea una lista con tipo, se completa y luego se consulta

```
public class Car
{
    public String Name { get; private set; }
    public int UnitsSold { get; private set; }

    public Car(string name, int unitsSold)
    {
        Name = name;
        UnitsSold = unitsSold;
    }
}

class Program
{
    static void Main(string[] args)
    {

        var car1 = new Car("VW Golf", 270952);
        var car2 = new Car("Opel Astra", 56079);
        var car3 = new Car("Audi A4", 52493);
        var car4 = new Car("Ford Focus", 51677);
        var car5 = new Car("Seat Leon", 42125);
        var car6 = new Car("VW Passat", 97586);
```

```
        var car7 = new Car("VW Polo", 69867);
        var car8 = new Car("Mercedes C-Class", 67549);

        var cars = new List<Car> {
            car1, car2, car3, car4, car5, car6, car7, car8 };
        var list = from car in cars
                   select car.Name;

        foreach (var entry in list)
        {
            Console.WriteLine(entry);
        }
        Console.ReadLine();
    }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

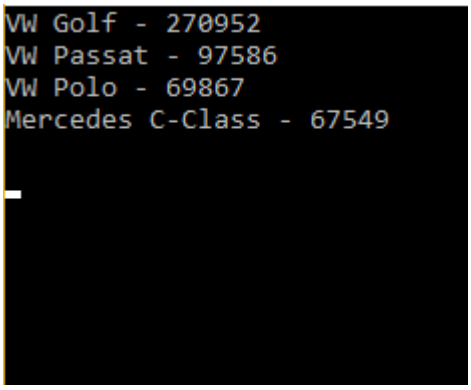



Hasta ahora, los ejemplos no parecen sorprendentes, ya que uno puede simplemente recorrer la matriz para hacer básicamente lo mismo. Sin embargo, con los pocos ejemplos a continuación, puede ver cómo crear consultas más complejas con LINQ to Objects y lograr más con mucho menos código.

En el siguiente ejemplo, podemos seleccionar los autos que se han vendido en más de 60000 unidades y clasificarlos según el número de unidades vendidas:

```
var list = from car in cars
           where car.UnitsSold > 60000
           orderby car.UnitsSold descending
           select car;

StringBuilder sb = new StringBuilder();

foreach (var entry in list)
{
    sb.AppendLine($"{entry.Name} - {entry.UnitsSold}");
}
Console.WriteLine(sb.ToString());
```

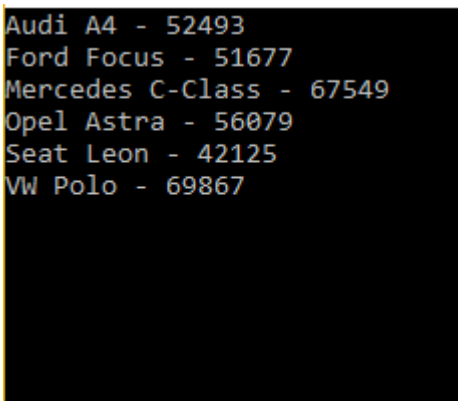
Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:




En el siguiente ejemplo, podemos seleccionar los automóviles que han vendido un número impar de unidades y ordenarlos alfabéticamente por su nombre:

```
var list = from car in cars
           where car.UnitsSold % 2 != 0
           orderby car.Name ascending
           select car;
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:




Lea Linq a los objetos en línea: https://riptutorial.com/es/csharp/topic/9405/linq-a-los-objetos

# Capítulo 107: LINQ paralelo (PLINQ)

## Sintaxis

- ParallelEnumerable.Aggregate (func)
- ParallelEnumerable.Aggregate (semilla, func)
- ParallelEnumerable.Aggregate (seed, updateAccumulatorFunc, combineAcculaulatorFunc, resultSelector)
- ParallelEnumerable.Aggregate (seedFactory, updateAccumulatorFunc, combineAccumulatorsFunc, resultSelector)
- ParallelEnumerable.All (predicado)
- ParallelEnumerable.Any ()
- ParallelEnumerable.Any (predicado)
- ParallelEnumerable.AsEnumerable ()
- ParallelEnumerable.AsOrdered ()
- ParallelEnumerable.AsParallel ()
- ParallelEnumerable.AsSequential ()
- ParallelEnumerable.AsUnordered ()
- ParallelEnumerable.Average (selector)
- ParallelEnumerable.Cast ()
- ParallelEnumerable.Concat (segundo)
- ParallelEnumerable.Contains (valor)
- ParallelEnumerable.Contains (valor, comparador)
- ParallelEnumerable.Count ()
- ParallelEnumerable.Count (predicado)
- ParallelEnumerable.DefaultIfEmpty ()
- ParallelEnumerable.DefaultIfEmpty (defaultValue)
- ParaleloEnumerable.Distinto ()
- ParaleloEnumerable.Distinto (comparador)
- ParallelEnumerable.ElementAt (index)
- ParallelEnumerable.ElementAtOrDefault (índice)
- ParallelEnumerable.Empty ()
- ParallelEnumerable.Except (segundo)
- ParallelEnumerable.Except (segundo, comparador)
- ParallelEnumerable.First ()
- ParallelEnumerable.First (predicado)
- ParallelEnumerable.FirstOrDefault ()
- ParallelEnumerable.FirstOrDefault (predicado)
- ParallelEnumerable.ForAll (action)
- ParallelEnumerable.GroupBy (keySelector)
- ParallelEnumerable.GroupBy (keySelector, comparer)
- ParallelEnumerable.GroupBy (keySelector, elementSelector)
- ParallelEnumerable.GroupBy (keySelector, elementSelector, comparer)
- ParallelEnumerable.GroupBy (keySelector, resultSelector)

- ParallelEnumerable.GroupBy (keySelector, resultSelector, comparer)
- ParallelEnumerable.GroupBy (keySelector, elementSelector, ruleSelector)
- ParallelEnumerable.GroupBy (keySelector, elementSelector, ruleSelector, comparer)
- ParallelEnumerable.GroupJoin (inner, outerKeySelector, innerKeySelector, resultSelector)
- ParallelEnumerable.GroupJoin (inner, outerKeySelector, innerKeySelector, resultSelector, comparer)
- ParallelEnumerable.Intersect (segundo)
- ParallelEnumerable.Intersect (segundo, comparador)
- ParallelEnumerable.Join (inner, outerKeySelector, innerKeySelector, resultSelector)
- ParallelEnumerable.Join (inner, outerKeySelector, innerKeySelector, resultSelector, comparer)
- ParallelEnumerable.Last ()
- ParallelEnumerable.Last (predicado)
- ParallelEnumerable.LastOrDefault ()
- ParallelEnumerable.LastOrDefault (predicado)
- ParallelEnumerable.LongCount ()
- ParallelEnumerable.LongCount (predicado)
- ParallelEnumerable.Max ()
- ParallelEnumerable.Max (selector)
- ParallelEnumerable.Min ()
- ParallelEnumerable.Min (selector)
- ParallelEnumerable.OfType ()
- ParallelEnumerable.OrderBy (keySelector)
- ParallelEnumerable.OrderBy (keySelector, comparer)
- ParallelEnumerable.OrderByDescending (keySelector)
- ParallelEnumerable.OrderByDescending (keySelector, comparer)
- ParallelEnumerable.Range (iniciar, contar)
- ParallelEnumerable.Repeat (elemento, count)
- ParallelEnumerable.Reverse ()
- ParallelEnumerable.Select (selector)
- ParallelEnumerable.SelectMany (selector)
- ParallelEnumerable.SelectMany (collectionSelector, resultSelector)
- ParallelEnumerable.SequenceEqual (segundo)
- ParallelEnumerable.SequenceEqual (segundo, comparador)
- ParallelEnumerable.Single ()
- ParallelEnumerable.Single (predicado)
- ParallelEnumerable.SingleOrDefault ()
- ParallelEnumerable.SingleOrDefault (predicado)
- ParallelEnumerable.Skip (count)
- ParallelEnumerable.SkipWhile (predicado)
- ParallelEnumerable.Sum ()
- ParallelEnumerable.Sum (selector)
- ParallelEnumerable.Take (count)
- ParallelEnumerable.TakeWhile (predicado)
- ParallelEnumerable.ThenBy (keySelector)
- ParallelEnumerable.ThenBy (keySelector, comparer)

- ParallelEnumerable.ThenByDescending (keySelector)
- ParallelEnumerable.ThenByDescending (keySelector, comparer)
- ParallelEnumerable.ToArray ()
- ParallelEnumerable.ToDictionary (keySelector)
- ParallelEnumerable.ToDictionary (keySelector, comparer)
- ParallelEnumerable.ToDictionary (elementSelector)
- ParallelEnumerable.ToDictionary (elementSelector, comparer)
- ParallelEnumerable.ToList ()
- ParallelEnumerable.ToLookup (keySelector)
- ParallelEnumerable.ToLookup (keySelector, comparer)
- ParallelEnumerable.ToLookup (keySelector, elementSelector)
- ParallelEnumerable.ToLookup (keySelector, elementSelector, comparer)
- ParallelEnumerable.Union (segundo)
- ParallelEnumerable.Union (segundo, comparador)
- ParallelEnumerable.Where (predicado)
- ParallelEnumerable.WithCancellation (cancelaciónToken)
- ParallelEnumerable.WithDegreeOfParallelism (degreeOfParallelism)
- ParallelEnumerable.WithExecutionMode (ecutionMode)
- ParallelEnumerable.WithMergeOptions (mergeOptions)
- ParallelEnumerable.Zip (segundo, resultSelector)

# Examples

## Ejemplo simple

Este ejemplo muestra cómo se puede usar PLINQ para calcular los números pares entre 1 y 10,000 usando múltiples subprocesos. Tenga en cuenta que la lista resultante no será ordenada!

```
var sequence = Enumerable.Range(1, 10000);
var evenNumbers = sequence.AsParallel()
                          .Where(x => x % 2 == 0)
                          .ToList();

// evenNumbers = { 4, 26, 28, 30, ... }
// Order will vary with different runs
```

## ConDegreeOfParalelismo

El grado de paralelismo es el número máximo de tareas que se ejecutan simultáneamente para procesar la consulta.

```
var sequence = Enumerable.Range(1, 10000);
var evenNumbers = sequence.AsParallel()
                          .WithDegreeOfParallelism(4)
                          .Where(x => x % 2 == 0);
```

## Según lo ordenado

Este ejemplo muestra cómo se puede usar PLINQ para calcular los números pares entre 1 y 10,000 usando múltiples subprocesos. El orden se mantendrá en la lista resultante, sin embargo, tenga en cuenta que `AsOrdered` puede afectar el rendimiento de una gran cantidad de elementos, por lo que es preferible el procesamiento no ordenado cuando sea posible.

```
var sequence = Enumerable.Range(1, 10000);
var evenNumbers = sequence.AsParallel()
                          .AsOrdered()
                          .Where(x => x % 2 == 0)
                          .ToList();

// evenNumbers = { 2, 4, 6, 8, ..., 10000 }
```

## As Sin orden

Las secuencias ordenadas pueden afectar el rendimiento cuando se trata de una gran cantidad de elementos. Para mitigar esto, es posible llamar a `AsUnordered` cuando el orden de la secuencia ya no es necesario.

```
var sequence = Enumerable.Range(1, 10000).Select(x => -1 * x); // -1, -2, ...
var evenNumbers = sequence.AsParallel()
                          .OrderBy(x => x)
                          .Take(5000)
                          .AsUnordered()
                          .Where(x => x % 2 == 0) // This line won't be affected by ordering
                          .ToList();
```

Lea LINQ paralelo (PLINQ) en línea: https://riptutorial.com/es/csharp/topic/3569/linq-paralelo--plinq-

# Capítulo 108: LINQ to XML

## Examples

### Leer XML usando LINQ a XML

```
<?xml version="1.0" encoding="utf-8" ?>
<Employees>
 <Employee>
  <EmpId>1</EmpId>
  <Name>Sam</Name>
  <Sex>Male</Sex>
  <Phone Type="Home">423-555-0124</Phone>
  <Phone Type="Work">424-555-0545</Phone>
  <Address>
   <Street>7A Cox Street</Street>
   <City>Acampo</City>
   <State>CA</State>
   <Zip>95220</Zip>
   <Country>USA</Country>
  </Address>
</Employee>
<Employee>
 <EmpId>2</EmpId>
 <Name>Lucy</Name>
 <Sex>Female</Sex>
 <Phone Type="Home">143-555-0763</Phone>
 <Phone Type="Work">434-555-0567</Phone>
 <Address>
  <Street>Jess Bay</Street>
  <City>Alta</City>
  <State>CA</State>
  <Zip>95701</Zip>
  <Country>USA</Country>
  </Address>
 </Employee>
</Employees>
```

Para leer ese archivo XML usando LINQ

```
XDocument xdocument = XDocument.Load("Employees.xml");
IEnumerable<XElement> employees = xdocument.Root.Elements();
foreach (var employee in employees)
{
    Console.WriteLine(employee);
}
```

Para acceder a un solo elemento

```
XElement xelement = XElement.Load("Employees.xml");
IEnumerable<XElement> employees = xelement.Root.Elements();
Console.WriteLine("List of all Employee Names :");
foreach (var employee in employees)
{
```

```
    Console.WriteLine(employee.Element("Name").Value);
}
```

Para acceder a múltiples elementos.

```
XElement xelement = XElement.Load("Employees.xml");
IEnumerable<XElement> employees = xelement.Root.Elements();
Console.WriteLine("List of all Employee Names along with their ID:");
foreach (var employee in employees)
{
    Console.WriteLine("{0} has Employee ID {1}",
    employee.Element("Name").Value,
    employee.Element("EmpId").Value);
}
```

Para acceder a todos los elementos que tienen un atributo específico

```
XElement xelement = XElement.Load("Employees.xml");
var name = from nm in xelement.Root.Elements("Employee")
       where (string)nm.Element("Sex") == "Female"
       select nm;
Console.WriteLine("Details of Female Employees:");
foreach (XElement xEle in name)
Console.WriteLine(xEle);
```

Para acceder a un elemento específico que tiene un atributo específico

```
XElement xelement = XElement.Load("..\\..\\Employees.xml");
var homePhone = from phoneno in xelement.Root.Elements("Employee")
            where (string)phoneno.Element("Phone").Attribute("Type") == "Home"
            select phoneno;
Console.WriteLine("List HomePhone Nos.");
foreach (XElement xEle in homePhone)
{
    Console.WriteLine(xEle.Element("Phone").Value);
}
```

Lea LINQ to XML en línea: https://riptutorial.com/es/csharp/topic/2773/linq-to-xml

# Capítulo 109: Literales

## Sintaxis

- **bool:** verdadero o falso
- **byte:** Ninguno, literal entero convertido implícitamente de int
- **sbyte:** Ninguno, literal entero convertido implícitamente de int
- **char:** Envuelve el valor con comillas simples
- **decimal:** m o m
- **doble:** D, d, o un número real
- **flotar:** F o f
- **int:** Ninguna, por defecto para valores integrales dentro del rango de int
- **uint:** U, u, o valores integrales dentro del rango de uint
- **long:** L, l, o valores integrales dentro del rango de long
- **ulong:** UL, ul, Ul, uL, LU, lu, Lu, lU o valores integrales dentro del rango de ulong
- **short:** Ninguno, literal entero convertido implícitamente de int
- **ushort:** Ninguno, literal entero convertido implícitamente de int
- **cadena:** ajuste el valor con comillas dobles, opcionalmente antepuesto con `@`
- **nulo** : el `null` literal

## Examples

### literales int

`int` literales `int` se definen simplemente utilizando valores integrales dentro del rango de `int` :

```
int i = 5;
```

### literales uint

`uint` literales de `uint` se definen usando el sufijo `U` o `u` , o usando valores integrales dentro del rango de `uint` :

```
uint ui = 5U;
```

### literales de cuerda

`string` literales de `string` se definen envolviendo el valor con comillas dobles `"` :

```
string s = "hello, this is a string literal";
```

Los literales de cuerdas pueden contener secuencias de escape. Ver secuencias de escape de cuerdas

Además, C # es compatible con literales de cadenas literales (consulte Cadenas verbales). Estos se definen envolviendo el valor con comillas dobles `"` , y anteponiéndolo con `@` . Las secuencias de escape se ignoran en literales de cadena textual, y se incluyen todos los caracteres de espacio en blanco:

```
string s = @"The path is:
C:\Windows\System32";
//The backslashes and newline are included in the string
```

### literales char

`char` literales de caracteres se definen envolviendo el valor con comillas simples `'` :

```
char c = 'h';
```

Los literales de caracteres pueden contener secuencias de escape. Ver secuencias de escape de cuerdas

Un literal de carácter debe tener exactamente un carácter de longitud (después de que se hayan evaluado todas las secuencias de escape). Los literales de caracteres vacíos no son válidos. El carácter predeterminado (devuelto por `default(char)` o `new char()` ) es `'\0'` , o el carácter nulo (que no debe confundirse con el `null` literal y referencias nulas).

### literales byte

`byte` tipo de `byte` no tiene sufijo literal. Los literales enteros se convierten implícitamente de `int` :

```
byte b = 127;
```

### sbyte literales

`sbyte` tipo `sbyte` no tiene sufijo literal. Los literales enteros se convierten implícitamente de `int` :

```
sbyte sb = 127;
```

### literales decimales

`decimal` se definen utilizando el sufijo M o m en un número real:

```
decimal m = 30.5M;
```

### dobles literales

`double` literales `double` se definen utilizando el sufijo D o d, o utilizando un número real:

```
double d = 30.5D;
```

## literales flotantes

`float` literales `float` se definen utilizando el sufijo F o f, o utilizando un número real:

```
float f = 30.5F;
```

## literales largos

`long` literales `long` se definen usando el sufijo `L` o `l` , o usando valores integrales dentro del rango de `long` :

```
long l = 5L;
```

## ulong literal

`ulong` literales de `ulong` se definen utilizando el sufijo `UL` , `ul` , `Ul` , `uL` , `LU` , `lu` , `Lu` o `lU` , o utilizando valores integrales dentro del rango de `ulong` :

```
ulong ul = 5UL;
```

## literal corto

`short` tipo `short` no tiene literal. Los literales enteros se convierten implícitamente de `int` :

```
short s = 127;
```

## ushort literal

`ushort` tipo `ushort` no tiene sufijo literal. Los literales enteros se convierten implícitamente de `int` :

```
ushort us = 127;
```

## literales bool

`bool` literales `bool` son `true` o `false` ;

```
bool b = true;
```

Lea Literales en línea: https://riptutorial.com/es/csharp/topic/2655/literales

# Capítulo 110: Los operadores

## Introducción

En C #, un operador es un elemento de programa que se aplica a uno o más operandos en una expresión o declaración. Los operadores que toman un operando, como el operador de incremento (++) o nuevo, se denominan operadores unarios. Los operadores que toman dos operandos, como los operadores aritméticos (+, -, *, /), se denominan operadores binarios. Un operador, el operador condicional (? :), toma tres operandos y es el único operador ternario en C #.

## Sintaxis

- Operador público OperandType operador operatorSymbol (OperandType operand1)
- Operador público OperandType operador operatorSymbol (OperandType operand1, OperandType2 operand2)

## Parámetros

| Parámetro | Detalles |
|---|---|
| símbolo de operador | El operador está sobrecargado, por ejemplo, +, -, /, * |
| OperandType | El tipo que será devuelto por el operador sobrecargado. |
| operando1 | El primer operando que se utilizará en la realización de la operación. |
| operando2 | El segundo operando que se utilizará para realizar la operación, cuando se realizan operaciones binarias. |
| declaraciones | Código opcional necesario para realizar la operación antes de devolver el resultado. |

## Observaciones

Todos los operadores se definen como `static methods` y no son `virtual` y no se heredan.

### Precedencia del operador

Todos los operadores tienen una "precedencia" particular según el grupo al que pertenezca el operador (los operadores del mismo grupo tienen la misma prioridad). Lo que significa que algunos operadores se aplicarán antes que otros. Lo que sigue es una lista de grupos (que contienen sus respectivos operadores) ordenados por precedencia (el más alto primero):

- **Operadores primarios**
  - `ab` - Acceso de miembros.
  - `a?.b` - Acceso de miembro condicional nulo.
  - `->` - Desreferenciación de punteros combinada con acceso de miembros.
  - `f(x)` - Invocación de la función.
  - `a[x]` - indexador.
  - `a?[x]` - Indizador condicional nulo.
  - `x++` - Incremento de Postfix.
  - `x--` - `x--` Postfix.
  - `new` - Tipo de instanciación.
  - `default(T)` : devuelve el valor inicializado predeterminado de tipo `T`
  - `typeof` - Devuelve el objeto `Type` del operando.
  - `checked` : habilita la comprobación de desbordamiento numérico.
  - `unchecked` marcar - Desactiva la comprobación de desbordamiento numérico.
  - `delegate` : declara y devuelve una instancia de delegado.
  - `sizeof` : devuelve el tamaño en bytes del operando de tipo.
- **Operadores Unarios**
  - `+x` - Devuelve `x` .
  - `-x` - Negación numérica.
  - `!x` - Negación lógica.
  - `~x` - Bitwise complemento / declara destructores.
  - `++x` - Incremento de prefijo.
  - `--x` - `--x` prefijo.
  - `(T)x` - Tipo de fundición.
  - `await` - espera una `Task` .
  - `&x` - Devuelve la dirección (puntero) de `x` .
  - `*x` - Desreferenciación del puntero.
- **Operadores Multiplicativos**
  - `x * y` - Multiplicación.
  - `x / y` - División.
  - `x % y` - Módulo.
- **Operadores Aditivos**
  - `x + y` - Adición.
  - `x - y` - resta.
- **Operadores de cambio bitwise**
  - `x << y` - Desplazar bits a la izquierda.
  - `x >> y` - Desplazar bits a la derecha.
- **Operadores relacionales / de prueba de tipo**

- `x < y` - Menos que.
- `x > y` - Mayor que.
- `x <= y` - menor o igual que.
- `x >= y` - Mayor o igual que.
- `is` - Compatibilidad de tipos.
- `as` - Tipo de conversión.

- **Operadores de Igualdad**
  - `x == y` - Igualdad.
  - `x != y` - No es igual.
- **Operador lógico y**
  - `x & y` - Lógica / bit a bit AND.
- **Operador XOR lógico**
  - `x ^ y` - XOR lógico / bit a bit.
- **Operador lógico o**
  - `x | y` - Lógica / bitwise OR.
- **Condicional y operador**
  - `x && y` - Cortocircuito lógico AND.
- **Operador condicional o**
  - `x || y` - Cortocircuito lógico OR.
- **Operador de fusión nula**
  - `x ?? y` - Devuelve `x` si no es nulo; De lo contrario, devuelve `y` .
- **Operador condicional**
  - `x ? y : z` - Evalúa / devuelve `y` si `x` es verdadero; De lo contrario, evalúa `z` .

---

**contenido relacionado**

- Operador de unión nula
- Operador condicional nulo
- Nombre del operador

# Examples

## Operadores sobrecargables

C # permite que los tipos definidos por el usuario sobrecarguen a los operadores definiendo funciones miembro estáticas usando la palabra clave del `operator` .
El siguiente ejemplo ilustra una implementación del operador `+` .

Si tenemos una clase `Complex` que representa un número complejo:

```
public struct Complex
{
    public double Real { get; set; }
    public double Imaginary { get; set; }
}
```

Y queremos agregar la opción de usar el operador `+` para esta clase. es decir:

```
Complex a = new Complex() { Real = 1, Imaginary = 2 };
Complex b = new Complex() { Real = 4, Imaginary = 8 };
Complex c = a + b;
```

Tendremos que sobrecargar el operador `+` para la clase. Esto se hace usando una función estática y la palabra clave del `operator` :

```
public static Complex operator +(Complex c1, Complex c2)
{
   return new Complex
   {
       Real = c1.Real + c2.Real,
       Imaginary = c1.Imaginary + c2.Imaginary
   };
}
```

Los operadores como `+` , `-` , `*` , `/` pueden estar sobrecargados. Esto también incluye a los operadores que no devuelven el mismo tipo (por ejemplo, `==` y `!=` Pueden estar sobrecargados, a pesar de los booleanos que regresan).

Los operadores de comparación deben estar sobrecargados en pares (por ejemplo, si `<` está sobrecargado, `>` también debe sobrecargarse).

Una lista completa de operadores sobrecargables (así como operadores no sobrecargables y las restricciones impuestas a algunos operadores sobrecargables) puede verse en MSDN - Operadores sobrecargables (Guía de programación de C #) .

7.0

la sobrecarga del `operator is` se introdujo con el mecanismo de coincidencia de patrones de C # 7.0. Para más detalles, ver el patrón de coincidencia

Dado un tipo `Cartesian` definido como sigue

```
public class Cartesian
```

```
{
    public int X { get; }
    public int Y { get; }
}
```

Un `operator is` sobrecargable podría, por ejemplo, definirse para coordenadas `Polar`

```
public static class Polar
{
    public static bool operator is(Cartesian c, out double R, out double Theta)
    {
        R = Math.Sqrt(c.X*c.X + c.Y*c.Y);
        Theta = Math.Atan2(c.Y, c.X);
        return c.X != 0 || c.Y != 0;
    }
}
```

que puede ser usado como este

```
var c = Cartesian(3, 4);
if (c is Polar(var R, *))
{
    Console.WriteLine(R);
}
```

(El ejemplo está tomado de la documentación de Roslyn Pattern Matching )

## Operadores relacionales

### Es igual a

Comprueba si los operandos (argumentos) suministrados son iguales

```
"a" == "b"     // Returns false.
"a" == "a"     // Returns true.
1 == 0         // Returns false.
1 == 1         // Returns true.
false == true  // Returns false.
false == false // Returns true.
```

A diferencia de Java, el operador de comparación de igualdad trabaja de forma nativa con cadenas.

El operador de comparación de igualdad trabajará con operandos de diferentes tipos si existe una conversión implícita de uno a otro. Si no existe una conversión implícita adecuada, puede llamar a una conversión explícita o usar un método para convertir a un tipo compatible.

```
1 == 1.0              // Returns true because there is an implicit cast from int to double.
new Object() == 1.0   // Will not compile.
MyStruct.AsInt() == 1 // Calls AsInt() on MyStruct and compares the resulting int with 1.
```

A diferencia de Visual Basic.NET, el operador de comparación de igualdad no es el mismo que el

operador de asignación de igualdad.

```
var x = new Object();
var y = new Object();
x == y // Returns false, the operands (objects in this case) have different references.
x == x // Returns true, both operands have the same reference.
```

*No debe confundirse con el operador de asignación ( `=` ).*

Para los tipos de valor, el operador devuelve `true` si ambos operandos tienen el mismo valor. Para los tipos de referencia, el operador devuelve `true` si ambos operandos son iguales en *referencia* (no en valor). Una excepción es que los objetos de cadena se compararán con la igualdad de valores.

**No es igual**

Comprueba si los operandos suministrados *no* son iguales.

```
"a" != "b"     // Returns true.
"a" != "a"     // Returns false.
1 != 0         // Returns true.
1 != 1         // Returns false.
false != true  // Returns true.
false != false // Returns false.

var x = new Object();
var y = new Object();
x != y // Returns true, the operands have different references.
x != x // Returns false, both operands have the same reference.
```

Este operador devuelve efectivamente el resultado opuesto al del operador igual ( `==` )

**Mas grande que**

Comprueba si el primer operando es mayor que el segundo operando.

```
3 > 5    //Returns false.
1 > 0    //Returns true.
2 > 2    //Return false.

var x = 10;
var y = 15;
x > y    //Returns false.
y > x    //Returns true.
```

**Menos que**

Comprueba si el primer operando es menor que el segundo operando.

```
2 < 4     //Returns true.
1 < -3    //Returns false.
2 < 2     //Return false.

var x = 12;
```

```
var y = 22;
x < y    //Returns true.
y < x    //Returns false.
```

### Mayor que igual a

Comprueba si el primer operando es mayor que igual al segundo operando.

```
7 >= 8    //Returns false.
0 >= 0    //Returns true.
```

### Menos que igual a

Comprueba si el primer operando es menor que igual al segundo operando.

```
2 <= 4    //Returns true.
1 <= -3    //Returns false.
1 <= 1     //Returns true.
```

## Operadores de cortocircuito

*Por definición, los operadores booleanos de cortocircuito solo evaluarán el segundo operando si el primer operando no puede determinar el resultado general de la expresión.*

Significa que, si está utilizando && operator como *firstCondition && secondCondition* , evaluará *secondCondition* solo cuando *firstCondition* sea verdadero y, por lo tanto, el resultado general será verdadero solo si tanto *firstOperand* como *secondOperand* se evalúan como verdaderos. Esto es útil en muchos escenarios, por ejemplo, imagine que desea verificar mientras que su lista tiene más de tres elementos, pero también debe verificar si la lista se ha inicializado para no ejecutarse en *NullReferenceException* . Puedes lograr esto de la siguiente manera:

```
bool hasMoreThanThreeElements = myList != null && mList.Count > 3;
```

*mList.Count> 3* no se verificará hasta que se cumpla myList! = null.

### Y lógico

`&&` es la contraparte de cortocircuito del operador booleano AND ( `&` ).

```
var x = true;
var y = false;

x && x // Returns true.
x && y // Returns false (y is evaluated).
y && x // Returns false (x is not evaluated).
y && y // Returns false (right y is not evaluated).
```

### O lógico

`||` es la contraparte de cortocircuito del operador booleano OR ( `|` ).

```
var x = true;
var y = false;

x || x // Returns true (right x is not evaluated).
x || y // Returns true (y is not evaluated).
y || x // Returns true (x and y are evaluated).
y || y // Returns false (y and y are evaluated).
```

**Ejemplo de uso**

```
if(object != null && object.Property)
// object.Property is never accessed if object is null, because of the short circuit.
    Action1();
else
    Action2();
```

## tamaño de

Devuelve un `int` contiene el tamaño de un tipo * en bytes.

```
sizeof(bool)    // Returns 1.
sizeof(byte)    // Returns 1.
sizeof(sbyte)   // Returns 1.
sizeof(char)    // Returns 2.
sizeof(short)   // Returns 2.
sizeof(ushort)  // Returns 2.
sizeof(int)     // Returns 4.
sizeof(uint)    // Returns 4.
sizeof(float)   // Returns 4.
sizeof(long)    // Returns 8.
sizeof(ulong)   // Returns 8.
sizeof(double)  // Returns 8.
sizeof(decimal) // Returns 16.
```

** Solo soporta ciertos tipos primitivos en contexto seguro.*

En un contexto inseguro, `sizeof` puede usarse para devolver el tamaño de otros tipos y estructuras primitivas.

```
public struct CustomType
{
    public int value;
}

static void Main()
{
    unsafe
    {
        Console.WriteLine(sizeof(CustomType)); // outputs: 4
    }
}
```

## Sobrecarga de operadores de igualdad.

Sobrecargar solo a los operadores de igualdad no es suficiente. Bajo diferentes circunstancias, todo lo siguiente puede ser llamado:

1. `object.Equals` and `object.GetHashCode`
2. `IEquatable<T>.Equals` (opcional, permite evitar el boxeo)
3. `operator ==` y `operator !=` (opcional, permite usar operadores)

Cuando se reemplaza `Equals` , `GetHashCode` también debe ser overriden. Cuando se implementa `Equals` , hay muchos casos especiales: comparando con objetos de un tipo diferente, comparándose con uno mismo, etc.

Cuando NO se reemplaza, el método de `Equals` y el operador `==` comportan de manera diferente para las clases y estructuras. Para las clases, solo se comparan las referencias, y para las estructuras, los valores de las propiedades se comparan a través de la reflexión, lo que puede afectar negativamente el rendimiento. `==` no se puede usar para comparar estructuras a menos que esté anulado.

Generalmente la operación de igualdad debe obedecer las siguientes reglas:

- No debe *lanzar excepciones* .
- Reflexividad: `A` siempre es igual a `A` (puede no ser cierto para valores `NULL` en algunos sistemas).
- Transitvity: si `A` es igual a `B` , y `B` es igual a `C` , entonces `A` es igual a `C`
- Si `A` es igual a `B` , entonces `A` y `B` tienen códigos hash iguales.
- Independencia del árbol de herencia: si `B` y `C` son instancias de `Class2` heredadas de `Class1` : `Class1.Equals(A,B)` siempre deben devolver el mismo valor que la llamada a `Class2.Equals(A,B)` .

```
class Student : IEquatable<Student>
{
    public string Name { get; set; } = "";

    public bool Equals(Student other)
    {
        if (ReferenceEquals(other, null)) return false;
        if (ReferenceEquals(other, this)) return true;
        return string.Equals(Name, other.Name);
    }

    public override bool Equals(object obj)
    {
        if (ReferenceEquals(null, obj)) return false;
        if (ReferenceEquals(this, obj)) return true;

        return Equals(obj as Student);
    }

    public override int GetHashCode()
    {
        return Name?.GetHashCode() ?? 0;
    }

    public static bool operator ==(Student left, Student right)
    {
```

```
        return Equals(left, right);
    }

    public static bool operator !=(Student left, Student right)
    {
        return !Equals(left, right);
    }
}
```

## Operadores de Miembros de Clase: Acceso de Miembros

```
var now = DateTime.UtcNow;
//accesses member of a class.  In this case the UtcNow property.
```

## Operadores de miembros de clase: Acceso de miembro condicional nulo

```
var zipcode = myEmployee?.Address?.ZipCode;
//returns null if the left operand is null.
//the above is the equivalent of:
var zipcode = (string)null;
if (myEmployee != null && myEmployee.Address != null)
    zipcode = myEmployee.Address.ZipCode;
```

## Operadores de Miembros de Clase: Invocación de Función

```
var age = GetAge(dateOfBirth);
//the above calls the function GetAge passing parameter dateOfBirth.
```

## Operadores de Miembros de Clase: Indización de Objetos Agregados

```
var letters = "letters".ToCharArray();
char letter = letters[1];
Console.WriteLine("Second Letter is {0}",letter);
//in the above example we take the second character from the array
//by calling letters[1]
//NB: Array Indexing starts at 0; i.e. the first letter would be given by letters[0].
```

## Operadores de Miembros de Clase: Indización Condicional Nula

```
var letters = null;
char? letter = letters?[1];
Console.WriteLine("Second Letter is {0}",letter);
//in the above example  rather than throwing an error because letters is null
//letter is assigned the value null
```

## "Exclusivo o" Operador

El operador para un "exclusivo o" (para XOR corto) es: ^

Este operador devuelve verdadero cuando uno, pero solo uno, de los valores proporcionados son

verdaderos.

```
true ^ false   // Returns true
false ^ true   // Returns true
false ^ false  // Returns false
true ^ true    // Returns false
```

## Operadores de cambio de bits

Los operadores de cambio permiten a los programadores ajustar un número entero desplazando todos sus bits hacia la izquierda o hacia la derecha. El siguiente diagrama muestra el efecto de desplazar un valor a la izquierda en un dígito.

### Shift izquierdo

```
uint value = 15;              // 00001111

uint doubled = value << 1;    // Result = 00011110 = 30
uint shiftFour = value << 4;  // Result = 11110000 = 240
```

### Giro a la derecha

```
uint value = 240;             // 11110000

uint halved = value >> 1;     // Result = 01111000 = 120
uint shiftFour = value >> 4;  // Result = 00001111 = 15
```

## Operadores de fundición implícita y explícita

C # permite que los tipos definidos por el usuario controlen la asignación y la conversión mediante el uso de palabras clave `explicit` e `implicit` . La firma del método toma la forma:

```
public static <implicit/explicit> operator <ResultingType>(<SourceType> myType)
```

El método no puede tomar más argumentos, ni puede ser un método de instancia. Sin embargo, puede acceder a cualquier miembro privado del tipo dentro del cual esté definido.

Un ejemplo de un reparto `implicit` y `explicit` :

```
public class BinaryImage
{
    private bool[] _pixels;

    public static implicit operator ColorImage(BinaryImage im)
    {
        return new ColorImage(im);
    }

    public static explicit operator bool[](BinaryImage im)
    {
        return im._pixels;
    }
```

```
}
```

Permitiendo la siguiente sintaxis de reparto:

```
var binaryImage = new BinaryImage();
ColorImage colorImage = binaryImage; // implicit cast, note the lack of type
bool[] pixels = (bool[])binaryImage; // explicit cast, defining the type
```

Los operadores de cast pueden trabajar en ambos sentidos, yendo *desde* su tipo *hasta* su tipo:

```
public class BinaryImage
{
    public static explicit operator ColorImage(BinaryImage im)
    {
        return new ColorImage(im);
    }

    public static explicit operator BinaryImage(ColorImage cm)
    {
        return new BinaryImage(cm);
    }
}
```

Finalmente, la palabra clave `as` , que puede participar en la conversión dentro de una jerarquía de tipos, **no** es válida en esta situación. Incluso después de definir una `explicit` o `implicit` , no puede hacer:

```
ColorImage cm = myBinaryImage as ColorImage;
```

Se generará un error de compilación.

## Operadores binarios con asignación

C # tiene varios operadores que se pueden combinar con un signo `=` para evaluar el resultado del operador y luego asignar el resultado a la variable original.

Ejemplo:

```
x += y
```

es lo mismo que

```
x = x + y
```

Operadores de Asignación:

- `+=`
- `-=`
- `*=`
- `/=`
- `%=`

- `&=`
- `|=`
- `^=`
- `<<=`
- `>>=`

## ? : Operador Ternario

Devuelve uno de los dos valores dependiendo del valor de una expresión booleana.

Sintaxis:

```
condition ? expression_if_true : expression_if_false;
```

Ejemplo:

```
string name = "Frank";
Console.WriteLine(name == "Frank" ? "The name is Frank" : "The name is not Frank");
```

El operador ternario es asociativo a la derecha, lo que permite utilizar expresiones ternarias compuestas. Esto se hace agregando ecuaciones ternarias adicionales en la posición verdadera o falsa de una ecuación ternaria principal. Se debe tener cuidado para garantizar la legibilidad, pero esto puede ser un atajo útil en algunas circunstancias.

En este ejemplo, una operación ternaria compuesta evalúa una función de `clamp` y devuelve el valor actual si está dentro del rango, el valor `min` si está por debajo del rango o el valor `max` si está por encima del rango.

```
light.intensity = Clamp(light.intensity, minLight, maxLight);

public static float Clamp(float val, float min, float max)
{
    return (val < min) ? min : (val > max) ? max : val;
}
```

Los operadores ternarios también pueden estar anidados, como:

```
a ? b ? "a is true, b is true" : "a is true, b is false" : "a is false"

// This is evaluated from left to right and can be more easily seen with parenthesis:

a ? (b ? x : y) : z

// Where the result is x if a && b, y if a && !b, and z if !a
```

Al escribir declaraciones ternarias compuestas, es común usar paréntesis o sangría para mejorar la legibilidad.

Los tipos de *expresión_if_true* y *expresión_if_false* deben ser idénticos o debe haber una conversión implícita de uno a otro.

```
condition ? 3 : "Not three"; // Doesn't compile because `int` and `string` lack an implicit
conversion.

condition ? 3.ToString() : "Not three"; // OK because both possible outputs are strings.

condition ? 3 : 3.5; // OK because there is an implicit conversion from `int` to `double`. The
ternary operator will return a `double`.

condition ? 3.5 : 3; // OK because there is an implicit conversion from `int` to `double`. The
ternary operator will return a `double`.
```

Los requisitos de tipo y conversión se aplican a sus propias clases también.

```
public class Car
{}

public class SportsCar : Car
{}

public class SUV : Car
{}

condition ? new SportsCar() : new Car(); // OK because there is an implicit conversion from
`SportsCar` to `Car`. The ternary operator will return a reference of type `Car`.

condition ? new Car() : new SportsCar(); // OK because there is an implicit conversion from
`SportsCar` to `Car`. The ternary operator will return a reference of type `Car`.

condition ? new SportsCar() : new SUV(); // Doesn't compile because there is no implicit
conversion from `SportsCar` to SUV or `SUV` to `SportsCar`. The compiler is not smart enough
to realize that both of them have an implicit conversion to `Car`.

condition ? new SportsCar() as Car : new SUV() as Car; // OK because both expressions evaluate
to a reference of type `Car`. The ternary operator will return a reference of type `Car`.
```

## tipo de

Obtiene el objeto `System.Type` para un tipo.

```
System.Type type = typeof(Point)        //System.Drawing.Point
System.Type type = typeof(IDisposable)  //System.IDisposable
System.Type type = typeof(Colors)       //System.Drawing.Color
System.Type type = typeof(List<>)       //System.Collections.Generic.List`1[T]
```

Para obtener el tipo de tiempo de ejecución, utilice `GetType` método para obtener el `System.Type` de la instancia actual.

Operador `typeof` toma un nombre de tipo como parámetro, que se especifica en tiempo de compilación.

```
public class Animal {}
public class Dog : Animal {}

var animal = new Dog();
```

```
Assert.IsTrue(animal.GetType() == typeof(Animal)); // fail, animal is typeof(Dog)
Assert.IsTrue(animal.GetType() == typeof(Dog));    // pass, animal is typeof(Dog)
Assert.IsTrue(animal is Animal);                   // pass, animal implements Animal
```

### operador predeterminado

## Tipo de valor (donde T: struct)

Los tipos de datos primitivos incorporados, como `char` , `int` y `float` , así como los tipos definidos por el usuario declarados con `struct` o `enum` . Su valor predeterminado es `new T()` :

```
default(int)            // 0
default(DateTime)       // 0001-01-01 12:00:00 AM
default(char)           // '\0' This is the "null character", not a zero or a line break.
default(Guid)           // 00000000-0000-0000-0000-000000000000
default(MyStruct)       // new MyStruct()

// Note: default of an enum is 0, and not the first *key* in that enum
// so it could potentially fail the Enum.IsDefined test
default(MyEnum)         // (MyEnum)0
```

## Tipo de referencia (donde T: clase)

Cualquier `class` , `interface` , matriz o tipo delegado. Su valor predeterminado es `null` :

```
default(object)         // null
default(string)         // null
default(MyClass)        // null
default(IDisposable)    // null
default(dynamic)        // null
```

### Nombre del operador

Devuelve una cadena que representa el nombre no calificado de una `variable` , `type` o `member` .

```
int counter = 10;
nameof(counter); // Returns "counter"
Client client = new Client();
nameof(client.Address.PostalCode)); // Returns "PostalCode"
```

El operador `nameof` fue introducido en C # 6.0. Se evalúa en tiempo de compilación y el compilador inserta el valor de cadena devuelto en línea, por lo que puede usarse en la mayoría de los casos donde se puede usar la cadena constante (por ejemplo, las etiquetas de `case` en una declaración de `switch` , atributos, etc.) .). Puede ser útil en casos como generar y registrar excepciones, atributos, enlaces de acción de MVC, etc.

### ?. (Operador Condicional Nulo)

6.0

Introducido en C # 6.0 , ¿el operador condicional nulo `?.` devolverá inmediatamente `null` si la expresión en su lado izquierdo se evalúa como `null` , en lugar de lanzar una `NullReferenceException` . Si su lado izquierdo se evalúa como un valor no `null` , se trata como un normal `.` operador. Tenga en cuenta que debido a que puede devolver un `null` , su tipo de retorno siempre es un tipo que puede contener `null` . Eso significa que para una estructura o tipo primitivo, se envuelve en un `Nullable<T>` .

```
var bar = Foo.GetBar()?.Value; // will return null if GetBar() returns null
var baz = Foo.GetBar()?.IntegerValue; // baz will be of type Nullable<int>, i.e. int?
```

Esto es útil cuando se disparan eventos. Normalmente, tendría que ajustar la llamada de evento en una sentencia if que compruebe si es `null` y luego elevar el evento, lo que presenta la posibilidad de una condición de carrera. Usando el operador condicional nulo, esto se puede arreglar de la siguiente manera:

```
event EventHandler<string> RaiseMe;
RaiseMe?.Invoke("Event raised");
```

## Postfix y Prefijo incremento y decremento

El incremento de Postfix `X++` agregará `1` a `x`

```
var x = 42;
x++;
Console.WriteLine(x); // 43
```

`X--` Postfix `X--` restará uno

```
var x = 42
x--;
Console.WriteLine(x); // 41
```

`++x` se denomina incremento de prefijo, incrementa el valor de x y luego devuelve x mientras que `x++` devuelve el valor de x y luego incrementa

```
var x = 42;
Console.WriteLine(++x); // 43
System.out.println(x); // 43
```

mientras

```
var x = 42;
Console.WriteLine(x++); // 42
System.out.println(x); // 43
```

Ambos se utilizan comúnmente en bucle for

```
for(int i = 0; i < 10; i++)
{
```

```
}
```

## => Operador Lambda

3.0

*El operador `=>` tiene la misma precedencia que el operador de asignación `=` y es asociativo a la derecha.*

Se utiliza para declarar expresiones lambda y también se usa ampliamente con las consultas LINQ :

```
string[] words = { "cherry", "apple", "blueberry" };

int shortestWordLength = words.Min((string w) => w.Length); //5
```

Cuando se usa en las consultas o extensiones de LINQ, el tipo de los objetos generalmente se puede omitir como lo deduce el compilador:

```
int shortestWordLength = words.Min(w => w.Length); //also compiles with the same result
```

La forma general de operador lambda es la siguiente:

```
(input parameters) => expression
```

Los parámetros de la expresión lambda se especifican antes del operador `=>` , y la expresión / declaración / bloque real que se ejecutará está a la derecha del operador:

```
// expression
(int x, string s) => s.Length > x

// expression
(int x, int y) => x + y

// statement
(string x) => Console.WriteLine(x)

// block
(string x) => {
        x += " says Hello!";
        Console.WriteLine(x);
    }
```

Este operador se puede usar para definir fácilmente delegados, sin escribir un método explícito:

```
delegate void TestDelegate(string s);

TestDelegate myDelegate = s => Console.WriteLine(s + " World");

myDelegate("Hello");
```

en lugar de

```
void MyMethod(string s)
{
    Console.WriteLine(s + " World");
}

delegate void TestDelegate(string s);

TestDelegate myDelegate = MyMethod;

myDelegate("Hello");
```

## Operador de asignación '='

El operador de asignación `=` establece el valor del operando de la mano izquierda al valor del operando de la derecha y devuelve ese valor:

```
int a = 3;     // assigns value 3 to variable a
int b = a = 5; // first assigns value 5 to variable a, then does the same for variable b
Console.WriteLine(a = 3 + 4); // prints 7
```

## ?? Operador de unión nula

¿El operador nulo-coalescente `??` devolverá el lado izquierdo cuando no sea nulo. Si es nulo, devolverá el lado derecho.

```
object foo = null;
object bar = new object();

var c = foo ?? bar;
//c will be bar since foo was null
```

El `??` El operador puede ser encadenado lo que permite la eliminación de `if` cheques.

```
//config will be the first non-null returned.
var config = RetrieveConfigOnMachine() ??
             RetrieveConfigFromService() ??
             new DefaultConfiguration();
```

Lea Los operadores en línea: https://riptutorial.com/es/csharp/topic/18/los-operadores

# Capítulo 111: Manejador de autenticación C #

## Examples

### Manejador de autentificación

```
 public class AuthenticationHandler : DelegatingHandler
    {
        /// <summary>
        /// Holds request's header name which will contains token.
        /// </summary>
        private const string securityToken = "__RequestAuthToken";

        /// <summary>
        /// Default overridden method which performs authentication.
        /// </summary>
        /// <param name="request">Http request message.</param>
        /// <param name="cancellationToken">Cancellation token.</param>
        /// <returns>Returns http response message of type <see cref="HttpResponseMessage"/>
class asynchronously.</returns>
        protected override Task<HttpResponseMessage> SendAsync(HttpRequestMessage request,
CancellationToken cancellationToken)
        {
            if (request.Headers.Contains(securityToken))
            {
                bool authorized = Authorize(request);
                if (!authorized)
                {
                    return ApiHttpUtility.FromResult(request, false,
HttpStatusCode.Unauthorized, MessageTypes.Error, Resource.UnAuthenticatedUser);
                }
            }
            else
            {
                return ApiHttpUtility.FromResult(request, false, HttpStatusCode.BadRequest,
MessageTypes.Error, Resource.UnAuthenticatedUser);
            }

            return base.SendAsync(request, cancellationToken);
        }

        /// <summary>
        /// Authorize user by validating token.
        /// </summary>
        /// <param name="requestMessage">Authorization context.</param>
        /// <returns>Returns a value indicating whether current request is authenticated or
not.</returns>
        private bool Authorize(HttpRequestMessage requestMessage)
        {
            try
            {
                HttpRequest request = HttpContext.Current.Request;
                string token = request.Headers[securityToken];
                return SecurityUtility.IsTokenValid(token, request.UserAgent,
HttpContext.Current.Server.MapPath("~/Content/"), requestMessage);
            }
            catch (Exception)
```

```
            {
                return false;
            }
        }
    }
```

Lea Manejador de autenticación C # en línea:
https://riptutorial.com/es/csharp/topic/5430/manejador-de-autenticacion-c-sharp

# Capítulo 112: Manejo de excepciones

## Examples

### Manejo básico de excepciones

```
try
{
    /* code that could throw an exception */
}
catch (Exception ex)
{
    /* handle the exception */
}
```

Tenga en cuenta que el manejo de todas las excepciones con el mismo código a menudo no es el mejor enfoque.
Esto se usa comúnmente cuando falla alguna rutina interna de manejo de excepciones, como último recurso.

### Manejo de tipos de excepción específicos

```
try
{
    /* code to open a file */
}
catch (System.IO.FileNotFoundException)
{
    /* code to handle the file being not found */
}
catch (System.IO.UnauthorizedAccessException)
{
    /* code to handle not being allowed access to the file */
}
catch (System.IO.IOException)
{
    /* code to handle IOException or it's descendant other than the previous two */
}
catch (System.Exception)
{
    /* code to handle other errors */
}
```

Tenga cuidado de que las excepciones se evalúen en orden y se aplique la herencia. Así que necesitas comenzar con los más específicos y terminar con su ancestro. En cualquier punto dado, solo se ejecutará un bloque catch.

### Usando el objeto de excepción

Se le permite crear y lanzar excepciones en su propio código. La creación de una excepción se realiza de la misma manera que cualquier otro objeto C #.

```
Exception ex = new Exception();

// constructor with an overload that takes a message string
Exception ex = new Exception("Error message");
```

A continuación, puede utilizar la palabra clave `throw` para provocar la excepción:

```
try
{
    throw new Exception("Error");
}
catch (Exception ex)
{
    Console.Write(ex.Message); // Logs 'Error' to the output window
}
```

**Nota:** si está lanzando una nueva excepción dentro de un bloque catch, asegúrese de que la excepción original se pase como "excepción interna", por ejemplo

```
void DoSomething()
{
    int b=1; int c=5;
    try
    {
        var a = 1;
        b = a - 1;
        c = a / b;
        a = a / c;
    }
    catch (DivideByZeroException dEx) when (b==0)
    {
        // we're throwing the same kind of exception
        throw new DivideByZeroException("Cannot divide by b because it is zero", dEx);
    }
    catch (DivideByZeroException dEx) when (c==0)
    {
        // we're throwing the same kind of exception
        throw new DivideByZeroException("Cannot divide by c because it is zero", dEx);
    }
}

void Main()
{
    try
    {
        DoSomething();
    }
    catch (Exception ex)
    {
        // Logs full error information (incl. inner exception)
        Console.Write(ex.ToString());
    }
}
```

En este caso, se supone que la excepción no se puede manejar, pero se agrega información útil al mensaje (y se puede acceder a la excepción original a través de `ex.InnerException` mediante un bloque de excepción externo).

Se mostrará algo como:

> System.DivideByZeroException: no se puede dividir entre b porque es cero --->
> System.DivideByZeroException: se intentó dividir entre cero.
> en UserQuery.g__DoSomething0_0 () en C: [...] \ LINQPadQuery.cs: línea 36
> --- Fin del rastro de la pila de excepción interna ---
> en UserQuery.g__DoSomething0_0 () en C: [...] \ LINQPadQuery.cs: línea 42
> en UserQuery.Main () en C: [...] \ LINQPadQuery.cs: línea 55

Si está probando este ejemplo en LinqPad, notará que los números de línea no son muy significativos (no siempre lo ayudan). Pero pasar un texto de error útil como se sugiere anteriormente a menudo reduce significativamente el tiempo para rastrear la ubicación del error, que en este ejemplo es claramente la línea

> c = a / b;

en la función `DoSomething()` .

**Pruébalo en .NET Fiddle**

## Finalmente bloque

```
try
{
    /* code that could throw an exception */
}
catch (Exception)
{
    /* handle the exception */
}
finally
{
    /* Code that will be executed, regardless if an exception was thrown / caught or not */
}
```

El bloque `try / catch / finally` puede ser muy útil al leer archivos.
Por ejemplo:

```
FileStream f = null;

try
{
    f = File.OpenRead("file.txt");
    /* process the file here */
}
finally
{
    f?.Close(); // f may be null, so use the null conditional operator.
}
```

Un bloque try debe ir seguido de un `catch` o un bloque `finally` . Sin embargo, como no hay bloque catch, la ejecución causará la terminación. Antes de la terminación, las declaraciones dentro del bloque finally serán ejecutadas.

En la lectura de archivos podríamos haber usado un bloque de `using` como `FileStream` (lo que devuelve `OpenRead` ) implementa como `IDisposable` .

Incluso si hay una declaración de `return` en el bloque `try` , el bloque `finally` generalmente se ejecutará; Hay algunos casos en los que no:

- Cuando se produce un StackOverflow .
- `Environment.FailFast`
- El proceso de solicitud es eliminado, generalmente por una fuente externa.

## Implementando IErrorHandler para los servicios WCF

La implementación de IErrorHandler para los servicios WCF es una excelente manera de centralizar el manejo de errores y el registro. La implementación que se muestra aquí debería detectar cualquier excepción no controlada que se haya generado como resultado de una llamada a uno de sus servicios WCF. También se muestra en este ejemplo cómo devolver un objeto personalizado y cómo devolver JSON en lugar del XML predeterminado.

Implementar IErrorHandler:

```
using System.ServiceModel.Channels;
using System.ServiceModel.Dispatcher;
using System.Runtime.Serialization.Json;
using System.ServiceModel;
using System.ServiceModel.Web;

namespace BehaviorsAndInspectors
{
    public class ErrorHandler : IErrorHandler
    {

        public bool HandleError(Exception ex)
        {
            // Log exceptions here

            return true;

        } // end

        public void ProvideFault(Exception ex, MessageVersion version, ref Message fault)
        {
            // Get the outgoing response portion of the current context
            var response = WebOperationContext.Current.OutgoingResponse;

            // Set the default http status code
            response.StatusCode = HttpStatusCode.InternalServerError;

            // Add ContentType header that specifies we are using JSON
            response.ContentType = new MediaTypeHeaderValue("application/json").ToString();

            // Create the fault message that is returned (note the ref parameter) with
BaseDataResponseContract
            fault = Message.CreateMessage(
                version,
                string.Empty,
                new CustomReturnType { ErrorMessage = "An unhandled exception occurred!" },
```

```
                new DataContractJsonSerializer(typeof(BaseDataResponseContract), new
List<Type> { typeof(BaseDataResponseContract) }));

            if (ex.GetType() == typeof(VariousExceptionTypes))
            {
                 // You might want to catch different types of exceptions here and process
them differently
            }

            // Tell WCF to use JSON encoding rather than default XML
            var webBodyFormatMessageProperty = new
WebBodyFormatMessageProperty(WebContentFormat.Json);
            fault.Properties.Add(WebBodyFormatMessageProperty.Name,
webBodyFormatMessageProperty);

        } // end

    } // end class

} // end namespace
```

En este ejemplo adjuntamos el controlador al comportamiento del servicio. También puede adjuntar esto a IEndpointBehavior, IContractBehavior o IOperationBehavior de una manera similar.

Adjuntar a los comportamientos de servicio:

```
using System;
using System.Collections.ObjectModel;
using System.ServiceModel;
using System.ServiceModel.Channels;
using System.ServiceModel.Configuration;
using System.ServiceModel.Description;
using System.ServiceModel.Dispatcher;

namespace BehaviorsAndInspectors
{
    public class ErrorHandlerExtension : BehaviorExtensionElement, IServiceBehavior
    {
        public override Type BehaviorType
        {
            get { return GetType(); }
        }

        protected override object CreateBehavior()
        {
            return this;
        }

        private IErrorHandler GetInstance()
        {
            return new ErrorHandler();
        }

        void IServiceBehavior.AddBindingParameters(ServiceDescription serviceDescription,
ServiceHostBase serviceHostBase, Collection<ServiceEndpoint> endpoints,
BindingParameterCollection bindingParameters) { } // end

        void IServiceBehavior.ApplyDispatchBehavior(ServiceDescription serviceDescription,
```

```
ServiceHostBase serviceHostBase)
        {
            var errorHandlerInstance = GetInstance();

            foreach (ChannelDispatcher dispatcher in serviceHostBase.ChannelDispatchers)
            {
                dispatcher.ErrorHandlers.Add(errorHandlerInstance);
            }
        }

        void IServiceBehavior.Validate(ServiceDescription serviceDescription, ServiceHostBase
serviceHostBase) { } // end

    } // end class

} // end namespace
```

Configuraciones en Web.config:

```
...
<system.serviceModel>

    <services>
      <service name="WebServices.MyService">
        <endpoint binding="webHttpBinding" contract="WebServices.IMyService" />
      </service>
    </services>

    <extensions>
      <behaviorExtensions>
        <!-- This extension if for the WCF Error Handling-->
        <add name="ErrorHandlerBehavior"
type="WebServices.BehaviorsAndInspectors.ErrorHandlerExtensionBehavior, WebServices,
Version=1.0.0.0, Culture=neutral, PublicKeyToken=null" />
      </behaviorExtensions>
    </extensions>

    <behaviors>
      <serviceBehaviors>
        <behavior>
          <serviceMetadata httpGetEnabled="true"/>
          <serviceDebug includeExceptionDetailInFaults="true"/>
          <ErrorHandlerBehavior />
        </behavior>
      </serviceBehaviors>
    </behaviors>

    ....
</system.serviceModel>
...
```

Aquí hay algunos enlaces que pueden ser útiles sobre este tema:

https://msdn.microsoft.com/en-us/library/system.servicemodel.dispatcher.ierrorhandler(v=vs.100).aspx

http://www.brainthud.com/cards/5218/25441/which-four-behavior-interfaces-exist-for-interacting-with-a-service-or-client-description-what-methods-do-they- implementar y

Otros ejemplos:

IErrorHandler devuelve el cuerpo del mensaje incorrecto cuando el código de estado HTTP es 401 No autorizado

IErrorHandler no parece estar manejando mis errores en WCF ... ¿alguna idea?

¿Cómo hacer que el controlador de errores WCF personalizado devuelva la respuesta JSON con un código http que no sea correcto?

¿Cómo configura el encabezado Content-Type para una solicitud HttpClient?

### Creación de excepciones personalizadas

Se le permite implementar excepciones personalizadas que pueden lanzarse como cualquier otra excepción. Esto tiene sentido cuando quiere hacer que sus excepciones sean distinguibles de otros errores durante el tiempo de ejecución.

En este ejemplo, crearemos una excepción personalizada para el manejo claro de los problemas que puede tener la aplicación al analizar una entrada compleja.

## Creación de una clase de excepción personalizada

Para crear una excepción personalizada, cree una subclase de `Exception` :

```
public class ParserException : Exception
{
    public ParserException() :
      base("The parsing went wrong and we have no additional information.") { }
}
```

La excepción personalizada se vuelve muy útil cuando desea proporcionar información adicional al receptor:

```
public class ParserException : Exception
{
    public ParserException(string fileName, int lineNumber) :
      base($"Parser error in {fileName}:{lineNumber}")
    {
      FileName = fileName;
      LineNumber = lineNumber;
    }
    public string FileName {get; private set;}
    public int LineNumber {get; private set;}
}
```

Ahora, cuando `catch(ParserException x)` tendrá una semántica adicional para afinar el manejo de excepciones.

Las clases personalizadas pueden implementar las siguientes funciones para admitir escenarios adicionales.

# relanzamiento

Durante el proceso de análisis, la excepción original sigue siendo de interés. En este ejemplo, es una `FormatException` porque el código intenta analizar un fragmento de cadena, que se espera que sea un número. En este caso, la excepción personalizada debe admitir la inclusión de la ' **InnerException** ':

```
//new constructor:
ParserException(string msg, Exception inner) : base(msg, inner) {
}
```

# publicación por entregas

En algunos casos, es posible que sus excepciones tengan que cruzar los límites de dominio de aplicación. Este es el caso si su analizador se ejecuta en su propio dominio de aplicación para admitir la recarga en caliente de las nuevas configuraciones del analizador. En Visual Studio, puede usar la plantilla de `Exception` para generar código como este.

```
[Serializable]
public class ParserException : Exception
{
    // Constructor without arguments allows throwing your exception without
    // providing any information, including error message. Should be included
    // if your exception is meaningful without any additional details. Should
    // set message by calling base constructor (default message is not helpful).
    public ParserException()
        : base("Parser failure.")
    {}

    // Constructor with message argument allows overriding default error message.
    // Should be included if users can provide more helpful messages than
    // generic automatically generated messages.
    public ParserException(string message)
        : base(message)
    {}

    // Constructor for serialization support. If your exception contains custom
    // properties, read their values here.
    protected ParserException(SerializationInfo info, StreamingContext context)
        : base(info, context)
    {}
}
```

# Usando la excepción ParserException

```
try
{
    Process.StartRun(fileName)
}
catch (ParserException ex)
{
    Console.WriteLine($"{ex.Message} in ${ex.FileName}:${ex.LineNumber}");
}
catch (PostProcessException x)
{
    ...
}
```

También puede usar excepciones personalizadas para atrapar y envolver excepciones. De esta manera, muchos errores diferentes se pueden convertir en un solo tipo de error que es más útil para la aplicación:

```
try
{
    int foo = int.Parse(token);
}
catch (FormatException ex)
{
    //Assuming you added this constructor
    throw new ParserException(
      $"Failed to read {token} as number.",
      FileName,
      LineNumber,
      ex);
}
```

Cuando maneje las excepciones al crear sus propias excepciones personalizadas, generalmente debe incluir una referencia a la excepción original en la propiedad `InnerException` , como se muestra arriba.

# Preocupaciones de seguridad

Si exponer el motivo de la excepción podría comprometer la seguridad al permitir que los usuarios vean el funcionamiento interno de su aplicación, puede ser una mala idea envolver la excepción interna. Esto podría aplicarse si está creando una biblioteca de clases que será utilizada por otros.

Aquí es cómo podría generar una excepción personalizada sin envolver la excepción interna:

```
try
{
  // ...
}
catch (SomeStandardException ex)
{
  // ...
  throw new MyCustomException(someMessage);
}
```

# Conclusión

Al generar una excepción personalizada (ya sea con una envoltura o con una nueva excepción no envuelta), debe presentar una excepción que sea significativa para la persona que llama. Por ejemplo, un usuario de una biblioteca de clases puede no saber mucho sobre cómo esa biblioteca hace su trabajo interno. Las excepciones que son lanzadas por las dependencias de la biblioteca de clases no son significativas. Más bien, el usuario desea una excepción que sea relevante para la forma en que la biblioteca de clases usa esas dependencias de manera errónea.

```
try
{
  // ...
}
catch (IOException ex)
{
  // ...
  throw new StorageServiceException(@"The Storage Service encountered a problem saving
your data. Please consult the inner exception for technical details.
If you are not able to resolve the problem, please call 555-555-1234 for technical
assistance.", ex);
}
```

### Excepción Anti-patrones

# Tragar excepciones

Siempre se debe volver a lanzar la excepción de la siguiente manera:

```
try
{
    ...
}
catch (Exception ex)
{
    ...
    throw;
}
```

Volver a lanzar una excepción como la que se muestra a continuación ocultará la excepción original y perderá el seguimiento de la pila original. ¡Uno nunca debe hacer esto! La traza de la pila antes de la captura y el retropropósito se perderá.

```
try
{
    ...
}
catch (Exception ex)
{
    ...
    throw ex;
```

```
}
```

# Manejo de excepciones de béisbol

Uno no debe usar excepciones como sustituto de las construcciones normales de control de flujo como las sentencias if-then y while. Este anti-patrón a veces se llama Baseball Exception Handling .

Aquí hay un ejemplo del anti-patrón:

```
try
{
    while (AccountManager.HasMoreAccounts())
    {
        account = AccountManager.GetNextAccount();
        if (account.Name == userName)
        {
            //We found it
            throw new AccountFoundException(account);
        }
    }
}
catch (AccountFoundException found)
{
    Console.Write("Here are your account details: " + found.Account.Details.ToString());
}
```

Aquí hay una mejor manera de hacerlo:

```
Account found = null;
while (AccountManager.HasMoreAccounts() && (found==null))
{
    account = AccountManager.GetNextAccount();
    if (account.Name == userName)
    {
        //We found it
        found = account;
    }
}
Console.Write("Here are your account details: " + found.Details.ToString());
```

# captura (Excepción)

Casi no hay razones (¡algunos dicen que no!) Para detectar el tipo de excepción genérico en su código. Debería capturar solo los tipos de excepción que espera que ocurran, porque de lo contrario ocultará los errores en su código.

```
try
{
     var f = File.Open(myfile);
     // do something
```

```
}
catch (Exception x)
{
     // Assume file not found
     Console.Write("Could not open file");
     // but maybe the error was a NullReferenceException because of a bug in the file handling
code?
}
```

Mejor hacer

```
try
{
     var f = File.Open(myfile);
     // do something which should normally not throw exceptions
}
catch (IOException)
{
     Console.Write("File not found");
}
// Unfortunatelly, this one does not derive from the above, so declare separatelly
catch (UnauthorizedAccessException)
{
     Console.Write("Insufficient rights");
}
```

Si ocurre alguna otra excepción, dejamos que la aplicación se bloquee a propósito, por lo que directamente entra en el depurador y podemos solucionar el problema. No debemos enviar un programa en el que ocurran otras excepciones que no sean estas, por lo que no es un problema tener un bloqueo.

El siguiente es un mal ejemplo, también, porque utiliza excepciones para evitar un error de programación. Eso no es para lo que están diseñados.

```
public void DoSomething(String s)
{
     if (s == null)
         throw new ArgumentNullException(nameof(s));
     // Implementation goes here
}

try
{
     DoSomething(myString);
}
catch(ArgumentNullException x)
{
    // if this happens, we have a programming error and we should check
    // why myString was null in the first place.
}
```

## Excepciones agregadas / excepciones múltiples de un método

Quien dice que no puedes lanzar múltiples excepciones en un método. Si no estás acostumbrado a jugar con AggregateExceptions, puedes tener la tentación de crear tu propia estructura de datos

para representar muchas cosas que van mal. Por supuesto, hay otra estructura de datos que no es una excepción sería más ideal, como los resultados de una validación. Incluso si juegas con AggregateExceptions puedes estar en el lado receptor y siempre manejarlas sin darte cuenta de que pueden serte de utilidad.

Es bastante plausible tener un método ejecutado y aunque será un error en su conjunto, querrá resaltar varias cosas que salieron mal en las excepciones que se lanzan. Como ejemplo, este comportamiento se puede ver con cómo funcionan los métodos paralelos si una tarea se divide en varios subprocesos y cualquier número de ellos podría generar excepciones y esto debe informarse. Aquí hay un ejemplo tonto de cómo podría beneficiarse de esto:

```
    public void Run()
    {
        try
        {
            this.SillyMethod(1, 2);
        }
        catch (AggregateException ex)
        {
            Console.WriteLine(ex.Message);
            foreach (Exception innerException in ex.InnerExceptions)
            {
                Console.WriteLine(innerException.Message);
            }
        }
    }

    private void SillyMethod(int input1, int input2)
    {
        var exceptions = new List<Exception>();

        if (input1 == 1)
        {
            exceptions.Add(new ArgumentException("I do not like ones"));
        }
        if (input2 == 2)
        {
            exceptions.Add(new ArgumentException("I do not like twos"));
        }
        if (exceptions.Any())
        {
            throw new AggregateException("Funny stuff happended during execution",
exceptions);
        }
    }
```

## Anidamiento de excepciones y prueba de captura de bloques.

Uno es capaz de anidar un bloque de `catch` excepción / `try` dentro del otro.

De esta manera, se pueden administrar pequeños bloques de código que pueden funcionar sin interrumpir todo su mecanismo.

```
try
{
```

```
//some code here
    try
    {
        //some thing which throws an exception. For Eg : divide by 0
    }
    catch (DivideByZeroException dzEx)
    {
        //handle here only this exception
        //throw from here will be passed on to the parent catch block
    }
    finally
    {
        //any thing to do after it is done.
    }
 //resume from here & proceed as normal;
}
catch(Exception e)
{
    //handle here
}
```

**Nota:** Evite tragar excepciones al lanzar al bloque de captura principal

### Mejores prácticas

## Hoja de trucos

| HACER | NO HACER |
|---|---|
| Flujo de control con declaraciones de control | Control de flujo con excepciones. |
| Mantenga un registro de la excepción ignorada (absorbida) mediante el registro | Ignorar la excepción |
| Repita la excepción usando el `throw` | Volver a lanzar la excepción: `throw new ArgumentNullException()` o `throw ex` |
| Lanzar excepciones del sistema predefinidas | Lanzar excepciones personalizadas similares a las excepciones del sistema predefinidas |
| Lanzar excepción personalizada / predefinida si es crucial para la lógica de la aplicación | Lanzar excepciones personalizadas / predefinidas para indicar una advertencia en el flujo |
| Captura las excepciones que quieras manejar | Atrapar todas las excepciones |

## NO maneje la lógica de negocios con excepciones.

El control de flujo NO debe hacerse por excepciones. Utilice declaraciones condicionales en su

lugar. Si se puede hacer un control con la instrucción `if-else` claramente, no use excepciones porque reduce la legibilidad y el rendimiento.

Considere el siguiente fragmento de código por el Sr. Bad Practices:

```
// This is a snippet example for DO NOT
object myObject;
void DoingSomethingWithMyObject()
{
    Console.WriteLine(myObject.ToString());
}
```

Cuando la ejecución llega a `Console.WriteLine(myObject.ToString());` La aplicación lanzará una NullReferenceException. El Sr. Bad Practices se dio cuenta de que `myObject` es nulo y editó su fragmento para capturar y manejar `NullReferenceException` :

```
// This is a snippet example for DO NOT
object myObject;
void DoingSomethingWithMyObject()
{
    try
    {
        Console.WriteLine(myObject.ToString());
    }
    catch(NullReferenceException ex)
    {
        // Hmmm, if I create a new instance of object and assign it to myObject:
        myObject = new object();
        // Nice, now I can continue to work with myObject
        DoSomethingElseWithMyObject();
    }
}
```

Dado que el fragmento de código anterior solo cubre la lógica de excepción, ¿qué debo hacer si `myObject` no es nulo en este momento? ¿Dónde debería cubrir esta parte de la lógica? Justo después de `Console.WriteLine(myObject.ToString());` ? ¿Qué tal después del `try...catch` block?

¿Qué tal el Sr. Mejores Prácticas? ¿Cómo manejaría esto?

```
// This is a snippet example for DO
object myObject;
void DoingSomethingWithMyObject()
{
    if(myObject == null)
        myObject = new object();

    // When execution reaches this point, we are sure that myObject is not null
    DoSomethingElseWithMyObject();
}
```

El Sr. Best Practices logró la misma lógica con menos código y una lógica clara y comprensible.

## NO vuelva a lanzar Excepciones

Volver a lanzar excepciones es costoso. Tiene un impacto negativo en el rendimiento. Para el código que normalmente falla, puede usar patrones de diseño para minimizar los problemas de rendimiento. Este tema describe dos patrones de diseño que son útiles cuando las excepciones pueden afectar significativamente el rendimiento.

## NO absorba excepciones sin registro

```
try
{
    //Some code that might throw an exception
}
catch(Exception ex)
{
    //empty catch block, bad practice
}
```

Nunca trague excepciones. Ignorar las excepciones ahorrará ese momento, pero creará un caos para el mantenimiento posterior. Al registrar excepciones, siempre debe registrar la instancia de excepción para que se registre el seguimiento completo de la pila y no solo el mensaje de excepción.

```
try
{
    //Some code that might throw an exception
}
catch(NullException ex)
{
    LogManager.Log(ex.ToString());
}
```

## No atrapes excepciones que no puedas manejar

Muchos recursos, como este , le recomiendan encarecidamente que considere por qué está detectando una excepción en el lugar donde la está detectando. Solo debe capturar una excepción si puede manejarla en esa ubicación. Si puede hacer algo allí para ayudar a mitigar el problema, como probar un algoritmo alternativo, conectarse a una base de datos de respaldo, probar otro nombre de archivo, esperar 30 segundos e intentarlo nuevamente o notificar a un administrador, puede detectar el error y hacerlo. Si no hay nada que pueda hacer de manera plausible y razonable, simplemente "déjelo" y deje que la excepción se maneje en un nivel superior. Si la excepción es lo suficientemente catastrófica y no existe otra opción razonable que no sea que todo el programa se bloquee debido a la gravedad del problema, entonces deje que se bloquee.

```
try
{
    //Try to save the data to the main database.
}
catch(SqlException ex)
{
    //Try to save the data to the alternative database.
```

```
}
//If anything other than a SqlException is thrown, there is nothing we can do here. Let the
exception bubble up to a level where it can be handled.
```

## Excepción no controlada y de rosca

**AppDomain.UnhandledException** Este evento proporciona una notificación de excepciones no detectadas. Permite que la aplicación registre información sobre la excepción antes de que el controlador predeterminado del sistema notifique la excepción al usuario y finalice la aplicación. Si hay suficiente información sobre el estado de la aplicación, otra se pueden emprender acciones, como guardar los datos del programa para su posterior recuperación. Se recomienda precaución, ya que los datos del programa pueden corromperse cuando no se manejan las excepciones.

```
    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    [STAThread]
    private static void Main(string[] args)
    {
        AppDomain.CurrentDomain.UnhandledException += new
UnhandledExceptionEventHandler(UnhandledException);
    }
```

**Application.ThreadException** Este evento permite que su aplicación de Windows Forms maneje las excepciones no controladas que ocurren en los hilos de Windows Forms. Adjunte sus controladores de eventos al evento ThreadException para lidiar con estas excepciones, lo que dejará su aplicación en un estado desconocido. Donde sea posible, las excepciones deben ser manejadas por un bloque estructurado de manejo de excepciones.

```
    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    [STAThread]
    private static void Main(string[] args)
    {
        AppDomain.CurrentDomain.UnhandledException += new
UnhandledExceptionEventHandler(UnhandledException);
        Application.ThreadException += new ThreadExceptionEventHandler(ThreadException);
    }
```

Y finalmente el manejo de excepciones.

```
static void UnhandledException(object sender, UnhandledExceptionEventArgs e)
    {
        Exception ex = (Exception)e.ExceptionObject;
        // your code
    }

static void ThreadException(object sender, ThreadExceptionEventArgs e)
    {
        Exception ex = e.Exception;
        // your code
    }
```

## Lanzar una excepción

Su código puede, y con frecuencia debería, lanzar una excepción cuando ha ocurrido algo inusual.

```
public void WalkInto(Destination destination)
{
    if (destination.Name == "Mordor")
    {
        throw new InvalidOperationException("One does not simply walk into Mordor.");
    }
    // ... Implement your normal walking code here.
}
```

Lea Manejo de excepciones en línea: https://riptutorial.com/es/csharp/topic/40/manejo-de-excepciones

# Capítulo 113: Manejo de FormatException al convertir cadenas a otros tipos

## Examples

### Convertir cadena a entero

Hay varios métodos disponibles para convertir explícitamente una `string` a un `integer` , como:

1. `Convert.ToInt16();`
2. `Convert.ToInt32();`
3. `Convert.ToInt64();`
4. `int.Parse();`

Pero todos estos métodos lanzarán una `FormatException` , si la cadena de entrada contiene caracteres no numéricos. Para esto, necesitamos escribir un manejo de excepciones adicional ( `try..catch` ) para tratarlas en tales casos.

---

**Explicación con ejemplos:**

Entonces, dejemos que nuestra entrada sea:

```
string inputString = "10.2";
```

**Ejemplo 1:** `Convert.ToInt32()`

```
int convertedInt = Convert.ToInt32(inputString); // Failed to Convert
// Throws an Exception "Input string was not in a correct format."
```

***Nota:*** *Lo mismo ocurre con los otros métodos mencionados, a saber:* *`Convert.ToInt16();`* *y* *`Convert.ToInt64();`*

**Ejemplo 2:** `int.Parse()`

```
int convertedInt = int.Parse(inputString); // Same result "Input string was not in a correct
format.
```

***¿Cómo sorteamos esto?***

Como se dijo anteriormente, para manejar las excepciones, usualmente necesitamos un `try..catch` como se muestra a continuación:

```
try
```

```
{
    string inputString = "10.2";
    int convertedInt = int.Parse(inputString);
}
catch (Exception Ex)
{
    //Display some message, that the conversion has failed.
}
```

Pero, usar `try..catch` todas partes no será una buena práctica, y puede haber algunos escenarios en los que quisiéramos dar `0` si la entrada es incorrecta *(si seguimos el método anterior, debemos asignar `0` a `convertedInt` desde el bloque de captura).* Para manejar tales escenarios podemos hacer uso de un método especial llamado `.TryParse()` .

El método `.TryParse()` tiene un manejo interno de Excepciones, que le dará la salida al parámetro de `out` y devuelve un valor booleano que indica el estado de la conversión *( `true` si la conversión fue exitosa; `false` si falló).* Según el valor de retorno, podemos determinar el estado de conversión. Veamos un ejemplo:

**Sintaxis 1:** almacena el valor de retorno en una variable booleana

```
 int convertedInt; // Be the required integer
 bool isSuccessConversion = int.TryParse(inputString, out convertedInt);
```

Podemos verificar la variable `isSuccessConversion` después de la ejecución para verificar el estado de la conversión. Si es falso, el valor de `convertedInt` será `0` *(no es necesario verificar el valor de retorno si desea que `0` un error de conversión).*

**Sintaxis 2:** compruebe el valor de retorno con `if`

```
if (int.TryParse(inputString, out convertedInt))
{
    // convertedInt will have the converted value
    // Proceed with that
}
else
{
 // Display an error message
}
```

**Uso 3:** Sin verificar el valor de retorno, puede usar lo siguiente, si no le importa el valor de retorno *(convertido o no, `0` estará bien)*

```
int.TryParse(inputString, out convertedInt);
// use the value of convertedInt
// But it will be 0 if not converted
```

Lea Manejo de FormatException al convertir cadenas a otros tipos en línea: https://riptutorial.com/es/csharp/topic/2886/manejo-de-formatexception-al-convertir-cadenas-a-otros-tipos

# Capítulo 114: Manipulación de cuerdas

## Examples

### Cambiando el caso de los personajes dentro de una cadena

La clase `System.String` admite varios métodos para convertir caracteres en mayúsculas y minúsculas en una cadena.

- `System.String.ToLowerInvariant` se usa para devolver un objeto String convertido a minúsculas.
- `System.String.ToUpperInvariant` se usa para devolver un objeto String convertido a mayúsculas.

**Nota:** la razón para usar las versiones *invariables* de estos métodos es evitar la producción de letras inesperadas específicas de la cultura. Esto se explica aquí en detalle .

Ejemplo:

```
string s = "My String";
s = s.ToLowerInvariant(); // "my string"
s = s.ToUpperInvariant(); // "MY STRING"
```

Tenga en cuenta que *puede* elegir especificar una **Cultura** específica al convertir a minúsculas y mayúsculas utilizando los métodos String.ToLower (CultureInfo) y String.ToUpper (CultureInfo) en consecuencia.

### Encontrar una cadena dentro de una cadena

Usando `System.String.Contains` puedes averiguar si una cadena en particular existe dentro de una cadena. El método devuelve un valor booleano, verdadero si la cadena existe, de lo contrario es falso

```
string s = "Hello World";
bool stringExists = s.Contains("ello");  //stringExists =true as the string contains the
substring
```

Usando el método `System.String.IndexOf` , puede ubicar la posición inicial de una subcadena dentro de una cadena existente.
Tenga en cuenta que la posición devuelta está basada en cero, se devuelve un valor de -1 si no se encuentra la subcadena.

```
string s = "Hello World";
int location = s.IndexOf("ello"); // location = 1
```

Para encontrar la primera ubicación desde el ***final*** de una cadena, use el método

`System.String.LastIndexOf` :

```
string s = "Hello World";
int location = s.LastIndexOf("l"); // location = 9
```

## Eliminar (recortar) espacios en blanco de una cadena

El método `System.String.Trim` se puede usar para eliminar todos los caracteres de espacio en blanco iniciales y finales de una cadena:

```
string s = "     String with spaces at both ends      ";
s = s.Trim(); // s = "String with spaces at both ends"
```

Adicionalmente:

- Para eliminar los espacios en blanco solo desde el *principio* de una cadena, utilice: `System.String.TrimStart`
- Para eliminar los espacios en blanco solo desde el *final* de una cadena, use: `System.String.TrimEnd`

**Substring para extraer parte de una cadena.**

El método `System.String.Substring` se puede utilizar para extraer una parte de la cadena.

```
string s ="A portion of word that is retained";
s=str.Substring(26);  //s="retained"

s1 = s.Substring(0,5);  //s="A por"
```

## Reemplazar una cadena dentro de una cadena

Usando el método `System.String.Replace` , puedes reemplazar parte de una cadena con otra cadena.

```
string s = "Hello World";
s = s.Replace("World", "Universe"); // s = "Hello Universe"
```

Todas las apariciones de la cadena de búsqueda se reemplazan:

```
string s = "Hello World";
s = s.Replace("l", "L"); // s = "HeLLo WorLD"
```

`String.Replace` también se puede usar para *eliminar* parte de una cadena, especificando una cadena vacía como valor de reemplazo:

```
string s = "Hello World";
s = s.Replace("ell", String.Empty); // s = "Ho World"
```

## Dividir una cadena usando un delimitador

Utilice el método `System.String.Split` para devolver una matriz de cadena que contiene subcadenas de la cadena original, dividida en función de un delimitador especificado:

```
string sentence = "One Two Three Four";
string[] stringArray = sentence.Split(' ');

foreach (string word in stringArray)
{
    Console.WriteLine(word);
}
```

Salida:

Uno
Dos
Tres
Cuatro

## Concatenar una matriz de cadenas en una sola cadena

El método `System.String.Join` permite concatenar todos los elementos en una matriz de cadenas, utilizando un separador especificado entre cada elemento:

```
string[] words = {"One", "Two", "Three", "Four"};
string singleString = String.Join(",", words); // singleString = "One,Two,Three,Four"
```

## Concatenacion de cuerdas

La concatenación de cadenas se puede hacer usando el método `System.String.Concat` , o (mucho más fácil) usando el operador `+` :

```
string first = "Hello ";
string second = "World";

string concat = first + second; // concat = "Hello World"
concat = String.Concat(first, second); // concat = "Hello World"
```

Lea Manipulación de cuerdas en línea: https://riptutorial.com/es/csharp/topic/3599/manipulacion-de-cuerdas

# Capítulo 115: Métodos

## Examples

### Declarar un método

Cada método tiene una firma única que consiste en un descriptor de acceso ( `public` , `private` , ...), un modificador opcional ( `abstract` ), un nombre y, si es necesario, parámetros del método. Tenga en cuenta que el tipo de retorno no es parte de la firma. Un prototipo de método se parece a lo siguiente:

```
AccessModifier OptionalModifier ReturnType MethodName(InputParameters)
{
    //Method body
}
```

`AccessModifier` puede ser `public` , `protected` , `pirvate` o por defecto `internal` .

`OptionalModifier` puede ser `virtual override virtual abstract static new` o `sealed` .

`ReturnType` puede `void` para no devolverse o puede ser de cualquier tipo desde los básicos, como `int` a clases complejas.

un método puede tener algunos o ningún parámetro de entrada. para establecer parámetros para un método, debe declarar cada una como declaraciones de variables normales (como `int a` ), y para más de un parámetro debe usar una coma entre ellos (como `int a, int b` ).

Los parámetros pueden tener valores por defecto. para esto, debe establecer un valor para el parámetro (como `int a = 0` ). Si un parámetro tiene un valor predeterminado, establecer el valor de entrada es opcional.

El siguiente ejemplo de método devuelve la suma de dos enteros:

```
private int Sum(int a, int b)
{
    return a + b;
}
```

### Llamando a un método

Llamando a un método estático:

```
// Single argument
System.Console.WriteLine("Hello World");

// Multiple arguments
string name = "User";
System.Console.WriteLine("Hello, {0}!", name);
```

Llamando a un método estático y almacenando su valor de retorno:

```
string input = System.Console.ReadLine();
```

Llamando a un método de instancia:

```
int x = 42;
// The instance method called here is Int32.ToString()
string xAsString = x.ToString();
```

Llamando a un método genérico

```
// Assuming a method 'T[] CreateArray<T>(int size)'
DateTime[] dates = CreateArray<DateTime>(8);
```

## Parámetros y Argumentos

Un método puede declarar cualquier número de parámetros (en este ejemplo, `i` , `s` y `o` son los parámetros):

```
static void DoSomething(int i, string s, object o) {
    Console.WriteLine(String.Format("i={0}, s={1}, o={2}", i, s, o));
}
```

Los parámetros se pueden usar para pasar valores a un método, de modo que el método pueda trabajar con ellos. Puede tratarse de todo tipo de trabajos, como imprimir los valores, realizar modificaciones en el objeto referenciado por un parámetro o almacenar los valores.

Cuando llama al método, debe pasar un valor real para cada parámetro. En este punto, los valores que realmente pasa a la llamada al método se denominan Argumentos:

```
DoSomething(x, "hello", new object());
```

## Tipos de retorno

Un método puede devolver nada ( `void` ) o un valor de un tipo especificado:

```
// If you don't want to return a value, use void as return type.
static void ReturnsNothing() {
    Console.WriteLine("Returns nothing");
}

// If you want to return a value, you need to specify its type.
static string ReturnsHelloWorld() {
    return "Hello World";
}
```

Si su método especifica un valor de retorno, el método *debe* devolver un valor. Haces esto usando la declaración de `return` . Una vez que se ha alcanzado una declaración de `return` ,

devuelve el valor especificado y cualquier código después de que ya no se ejecute (las excepciones `finally` son bloques, que aún se ejecutarán antes de que el método regrese).

Si su método no devuelve nada ( `void` ), aún puede usar la declaración de `return` sin un valor si desea regresar del método inmediatamente. Al final de tal método, una declaración de `return` sería innecesaria sin embargo.

Ejemplos de declaraciones de `return` válidas:

```
return;
return 0;
return x * 2;
return Console.ReadLine();
```

Lanzar una excepción puede finalizar la ejecución del método sin devolver un valor. Además, hay bloques de iteradores, donde los valores de retorno se generan mediante el uso de la palabra clave de rendimiento, pero esos son casos especiales que no se explicarán en este momento.

## Parámetros predeterminados

Puede usar parámetros predeterminados si desea proporcionar la opción de omitir parámetros:

```
static void SaySomething(string what = "ehh") {
    Console.WriteLine(what);
}

static void Main() {
    // prints "hello"
    SaySomething("hello");
    // prints "ehh"
    SaySomething(); // The compiler compiles this as if we had typed SaySomething("ehh")
}
```

Cuando llama a un método de este tipo y omite un parámetro para el que se proporciona un valor predeterminado, el compilador inserta ese valor predeterminado para usted.

Tenga en cuenta que los parámetros con valores predeterminados deben escribirse **después de los** parámetros sin valores predeterminados.

```
static void SaySomething(string say, string what = "ehh") {
        //Correct
        Console.WriteLine(say + what);
    }

static void SaySomethingElse(string what = "ehh", string say) {
        //Incorrect
        Console.WriteLine(say + what);
    }
```

**ADVERTENCIA** : debido a que funciona de esa manera, los valores predeterminados pueden ser problemáticos en algunos casos. Si cambia el valor predeterminado de un parámetro de método y no vuelve a compilar a todos los llamadores de ese método, esos llamadores seguirán usando el

valor predeterminado que estaba presente cuando se compilaron, lo que podría causar inconsistencias.

## Método de sobrecarga

**Definición:** Cuando se declaran múltiples métodos con el mismo nombre con diferentes parámetros, se denomina sobrecarga del método. La sobrecarga de métodos generalmente representa funciones que son idénticas en su propósito pero que están escritas para aceptar diferentes tipos de datos como sus parámetros.

**Factores que afectan**

- Número de Argumentos
- Tipo de argumentos
- Tipo de devolución **

Considere un método llamado `Area` que realizará funciones de cálculo, que aceptará varios argumentos y devolverá el resultado.

**Ejemplo**

```
public string Area(int value1)
{
    return String.Format("Area of Square is {0}", value1 * value1);
}
```

Este método aceptará un argumento y devolverá una cadena, si llamamos al método con un entero (por ejemplo, `5` ), la salida será `"Area of Square is 25"` .

```
public  double Area(double value1, double value2)
{
    return value1 * value2;
}
```

De manera similar, si pasamos dos valores dobles a este método, la salida será el producto de los dos valores y será de tipo doble. Esto puede ser usado tanto para la multiplicación como para encontrar el área de los rectángulos.

```
public double Area(double value1)
{
    return 3.14 * Math.Pow(value1,2);
}
```

Esto se puede usar especialmente para encontrar el área del círculo, que acepta un valor doble ( `radius` ) y devuelve otro valor doble como su Área.

Cada uno de estos métodos se puede llamar normalmente sin conflicto: el compilador examinará los parámetros de cada llamada de método para determinar qué versión de `Area` debe usar.

```
string squareArea = Area(2);
```

```
double rectangleArea = Area(32.0, 17.5);
double circleArea = Area(5.0); // all of these are valid and will compile.
```

---

** Tenga en cuenta que el tipo de retorno *solo* no puede diferenciar entre dos métodos. Por ejemplo, si tuviéramos dos definiciones para el Área que tenían los mismos parámetros, así:

```
public string Area(double width, double height) { ... }
public double Area(double width, double height) { ... }
// This will NOT compile.
```

Si necesitamos que nuestra clase use los mismos nombres de métodos que devuelven valores diferentes, podemos eliminar los problemas de ambigüedad implementando una interfaz y definiendo explícitamente su uso.

```
public interface IAreaCalculatorString {

    public string Area(double width, double height);

}

public class AreaCalculator : IAreaCalculatorString {

    public string IAreaCalculatorString.Area(double width, double height) { ... }
    // Note that the method call now explicitly says it will be used when called through
    // the IAreaCalculatorString interface, allowing us to resolve the ambiguity.
    public double Area(double width, double height) { ... }
```

## Método anónimo

Los métodos anónimos proporcionan una técnica para pasar un bloque de código como un parámetro delegado. Son métodos con cuerpo, pero sin nombre.

```
delegate int IntOp(int lhs, int rhs);
```

```
class Program
{
    static void Main(string[] args)
    {
        // C# 2.0 definition
        IntOp add = delegate(int lhs, int rhs)
        {
            return lhs + rhs;
        };

        // C# 3.0 definition
        IntOp mul = (lhs, rhs) =>
        {
            return lhs * rhs;
        };

        // C# 3.0 definition - shorthand
        IntOp sub = (lhs, rhs) => lhs - rhs;
```

```
        // Calling each method
        Console.WriteLine("2 + 3 = " + add(2, 3));
        Console.WriteLine("2 * 3 = " + mul(2, 3));
        Console.WriteLine("2 - 3 = " + sub(2, 3));
    }
}
```

## Derechos de acceso

```
// static: is callable on a class even when no instance of the class has been created
public static void MyMethod()

// virtual: can be called or overridden in an inherited class
public virtual  void MyMethod()

// internal: access is limited within the current assembly
internal  void MyMethod()

//private: access is limited only within the same class
private  void MyMethod()

//public: access right from every class / assembly
public void MyMethod()

//protected: access is limited to the containing class or types derived from it
protected void MyMethod()

//protected internal: access is limited to the current assembly or types derived from the
containing class.
protected internal void MyMethod()
```

Lea Métodos en línea: https://riptutorial.com/es/csharp/topic/60/metodos

# Capítulo 116: Métodos de extensión

## Sintaxis

- ReturnType estático público MyExtensionMethod (este objetivo TargetType)
- ReturnType estático público MyExtensionMethod (este objetivo TargetType, TArg1 arg1, ...)

## Parámetros

| Parámetro | Detalles |
|---|---|
| esta | El primer parámetro de un método de extensión siempre debe ir precedido por `this` palabra clave, seguido del identificador con el que se refiere a la instancia "actual" del objeto que está extendiendo. |

## Observaciones

Los métodos de extensión son azúcares sintácticos que permiten invocar métodos estáticos en instancias de objetos como si fueran miembros del tipo en sí.

Los métodos de extensión requieren un objeto objetivo explícito. Deberá usar `this` palabra clave para acceder al método desde el propio tipo extendido.

Los métodos de extensiones deben declararse estáticos y deben vivir en una clase estática.

**¿Qué espacio de nombres?**

La elección del espacio de nombres para su clase de método de extensión es una compensación entre visibilidad y capacidad de descubrimiento.

La opción más comúnmente mencionada es tener un espacio de nombres personalizado para sus métodos de extensión. Sin embargo, esto implicará un esfuerzo de comunicación para que los usuarios de su código sepan que existen los métodos de extensión y dónde encontrarlos.

Una alternativa es elegir un espacio de nombres para que los desarrolladores descubran sus métodos de extensión a través de Intellisense. Entonces, si desea extender la clase `Foo` , es lógico colocar los métodos de extensión en el mismo espacio de nombres que `Foo` .

Es importante darse cuenta de que **nada le impide usar el espacio de nombres de "otra persona"** : por lo tanto, si desea extender `IEnumerable` , puede agregar su método de extensión en el espacio de nombres `System.Linq` .

Esto no *siempre es* una buena idea. Por ejemplo, en un caso específico, es posible que desee extender un tipo común ( `bool IsApproxEqualTo(this double value, double other)` por ejemplo), pero no tener que "contaminar" todo el `System` . En este caso, es preferible elegir un espacio de

nombres local, específico.

Finalmente, también es posible poner los métodos de extensión en *ningún espacio de nombres* .

Una buena pregunta de referencia: ¿Cómo administra los espacios de nombres de sus métodos de extensión?

**Aplicabilidad**

Se debe tener cuidado al crear métodos de extensión para garantizar que sean apropiados para todas las entradas posibles y que no solo sean relevantes para situaciones específicas. Por ejemplo, es posible extender las clases del sistema como la `string` , lo que hace que su nuevo código esté disponible para **cualquier** cadena. Si su código necesita realizar una lógica específica del dominio en un formato de cadena específico del dominio, un método de extensión no sería apropiado ya que su presencia confundiría a los llamantes que trabajan con otras cadenas en el sistema.

**La siguiente lista contiene características y propiedades básicas de los métodos de extensión.**

1. Debe ser un método estático.
2. Debe estar ubicado en una clase estática.
3. Utiliza la palabra clave "this" como primer parámetro con un tipo en .NET y este método será llamado por una instancia de tipo dada en el lado del cliente.
4. También se muestra por VS intellisense. Cuando presionamos el punto . después de una instancia de tipo, entonces viene en VS intellisense.
5. Un método de extensión debe estar en el mismo espacio de nombres que se usa o debe importar el espacio de nombres de la clase mediante una instrucción using.
6. Puede dar cualquier nombre para la clase que tenga un método de extensión, pero la clase debe ser estática.
7. Si desea agregar nuevos métodos a un tipo y no tiene el código fuente para ello, entonces la solución es usar e implementar métodos de extensión de ese tipo.
8. Si crea métodos de extensión que tienen los mismos métodos de firma que el tipo que está extendiendo, los métodos de extensión nunca serán llamados.

# Examples

## Métodos de extensión - descripción general

Los métodos de extensión se introdujeron en C # 3.0. Los métodos de extensión extienden y agregan comportamiento a los tipos existentes sin crear un nuevo tipo derivado, recompilando o modificando el tipo original. *Son especialmente útiles cuando no puede modificar la fuente de un tipo que desea mejorar.* Se pueden crear métodos de extensión para los tipos de sistema, los tipos definidos por terceros y los tipos que usted mismo haya definido. El método de extensión se puede invocar como si fuera un método miembro del tipo original. Esto permite el **método de encadenamiento** utilizado para implementar una **interfaz fluida** .

Se crea un método de extensión agregando un **método estático** a una **clase estática** que es

distinta del tipo original que se extiende. La clase estática que contiene el método de extensión a menudo se crea con el único propósito de mantener métodos de extensión.

Los métodos de extensión toman un primer parámetro especial que designa el tipo original que se está extendiendo. Este primer parámetro está decorado con la palabra clave `this` (que constituye un uso especial y distinto de `this` en C #; debe entenderse como diferente del uso de `this` que permite referirse a miembros de la instancia de objeto actual).

En el siguiente ejemplo, el tipo original que se extiende es la `string` clase. `String` se ha extendido mediante un método `Shorten()` , que proporciona la funcionalidad adicional de acortamiento. La clase estática `StringExtensions` se ha creado para contener el método de extensión. El método de extensión `Shorten()` muestra que es una extensión de `string` través del primer parámetro especialmente marcado. Para mostrar que el método `Shorten()` extiende la `string` , el primer parámetro se marca con `this` . Por lo tanto, la firma completa del primer parámetro es el `this string text` , donde la `string` es el tipo original que se extiende y el `text` es el nombre del parámetro elegido.

```
static class StringExtensions
{
    public static string Shorten(this string text, int length)
    {
        return text.Substring(0, length);
    }
}

class Program
{
    static void Main()
    {
        // This calls method String.ToUpper()
        var myString = "Hello World!".ToUpper();

        // This calls the extension method StringExtensions.Shorten()
        var newString = myString.Shorten(5);

        // It is worth noting that the above call is purely syntactic sugar
        // and the assignment below is functionally equivalent
        var newString2 = StringExtensions.Shorten(myString, 5);
    }
}
```

Demo en vivo en .NET Fiddle

---

El objeto pasado como *primer argumento de un método de extensión* (que está acompañado por `this` palabra clave) es la instancia a la que se llama el método de extensión.

Por ejemplo, cuando este código se ejecuta:

```
"some string".Shorten(5);
```

Los valores de los argumentos son los siguientes:

```
text: "some string"
length: 5
```

*Tenga en cuenta que los métodos de extensión solo se pueden usar si están en el mismo espacio de nombres que su definición, si el espacio de nombres se importa explícitamente mediante el código que usa el método de extensión, o si la clase de extensión no tiene espacio de nombres.* Las pautas del marco .NET recomiendan colocar las clases de extensión en su propio espacio de nombres. Sin embargo, esto puede llevar a problemas de descubrimiento.

Esto no genera conflictos entre los métodos de extensión y las bibliotecas que se están utilizando, a menos que se incluyan explícitamente los espacios de nombres que puedan entrar en conflicto. Por ejemplo, las Extensiones LINQ :

```
using System.Linq; // Allows use of extension methods from the System.Linq namespace

class Program
{
    static void Main()
    {
        var ints = new int[] {1, 2, 3, 4};

        // Call Where() extension method from the System.Linq namespace
        var even = ints.Where(x => x % 2 == 0);
    }
}
```

Demo en vivo en .NET Fiddle

---

Desde C # 6.0, también es posible poner una directiva `using static` a la *clase que* contiene los métodos de extensión. Por ejemplo, `using static System.Linq.Enumerable;` . Esto hace que los métodos de extensión de esa clase en particular estén disponibles sin traer otros tipos del mismo espacio de nombres al alcance.

---

Cuando un método de clase con la misma firma está disponible, el compilador lo prioriza sobre la llamada al método de extensión. Por ejemplo:

```
class Test
{
   public void Hello()
   {
       Console.WriteLine("From Test");
   }
}

static class TestExtensions
{
    public static void Hello(this Test test)
    {
        Console.WriteLine("From extension method");
    }
}
```

```
class Program
{
    static void Main()
    {
        Test t = new Test();
        t.Hello(); // Prints "From Test"
    }
}
```

Demo en vivo en .NET Fiddle

---

Tenga en cuenta que si hay dos funciones de extensión con la misma firma, y una de ellas se encuentra en el mismo espacio de nombres, se asignará prioridad a esa. Por otro lado, si se accede a ambos `using` , se producirá un error de tiempo de compilación con el mensaje:

> **La llamada es ambigua entre los siguientes métodos o propiedades.**

---

Tenga en cuenta que la conveniencia sintáctica de llamar a un método de extensión a través de `originalTypeInstance.ExtensionMethod()` es una conveniencia opcional. El método también se puede llamar de la manera tradicional, de modo que el primer parámetro especial se utiliza como parámetro del método.

Es decir, ambos de los siguientes trabajos:

```
//Calling as though method belongs to string--it seamlessly extends string
String s = "Hello World";
s.Shorten(5);

//Calling as a traditional static method with two parameters
StringExtensions.Shorten(s, 5);
```

## Usando explícitamente un método de extensión

Los métodos de extensión también pueden usarse como métodos de clase estática ordinarios. Esta forma de llamar a un método de extensión es más detallada, pero es necesaria en algunos casos.

```
static class StringExtensions
{
    public static string Shorten(this string text, int length)
    {
        return text.Substring(0, length);
    }
}
```

Uso:

```
var newString = StringExtensions.Shorten("Hello World", 5);
```

# Cuándo llamar a los métodos de extensión como métodos estáticos.

Todavía hay escenarios en los que necesitarías usar un método de extensión como método estático:

- Resolución de conflictos con un método miembro. Esto puede suceder si una nueva versión de una biblioteca introduce un nuevo método de miembro con la misma firma. En este caso, el compilador preferirá el método miembro.
- Resolución de conflictos con otro método de extensión con la misma firma. Esto puede suceder si dos bibliotecas incluyen métodos de extensión similares y los espacios de nombres de ambas clases con métodos de extensión se usan en el mismo archivo.
- Pasando el método de extensión como un grupo de método en el parámetro delegado.
- Haciendo su propio enlace a través de la `Reflection` .
- Usando el método de extensión en la ventana Inmediato en Visual Studio.

# Usando estática

Si se `using static` una directiva `using static` que usa para traer miembros estáticos de una clase estática al alcance global, se omiten los métodos de extensión. Ejemplo:

```
using static OurNamespace.StringExtensions; // refers to class in previous example

// OK: extension method syntax still works.
"Hello World".Shorten(5);
// OK: static method syntax still works.
OurNamespace.StringExtensions.Shorten("Hello World", 5);
// Compile time error: extension methods can't be called as static without specifying class.
Shorten("Hello World", 5);
```

Si elimina `this` modificador del primer argumento del método `Shorten` , se compilará la última línea.

## Comprobación nula

Los métodos de extensión son métodos estáticos que se comportan como métodos de instancia. Sin embargo, a diferencia de lo que sucede cuando se llama a un método de instancia en una referencia `null` , cuando se llama a un método de extensión con una referencia `null` , no se produce una `NullReferenceException` . Esto puede ser bastante útil en algunos escenarios.

Por ejemplo, considere la siguiente clase estática:

```
public static class StringExtensions
{
    public static string EmptyIfNull(this string text)
    {
        return text ?? String.Empty;
```

```
    }

    public static string NullIfEmpty(this string text)
    {
        return String.Empty == text ? null : text;
    }
}
```

```
string nullString = null;
string emptyString = nullString.EmptyIfNull();// will return ""
string anotherNullString = emptyString.NullIfEmpty(); // will return null
```

Demo en vivo en .NET Fiddle

## Los métodos de extensión solo pueden ver miembros públicos (o internos) de la clase extendida

```
public class SomeClass
{
    public void DoStuff()
    {

    }

    protected void DoMagic()
    {

    }
}

public static class SomeClassExtensions
{
    public static void DoStuffWrapper(this SomeClass someInstance)
    {
        someInstance.DoStuff(); // ok
    }

    public static void DoMagicWrapper(this SomeClass someInstance)
    {
        someInstance.DoMagic(); // compilation error
    }
}
```

Los métodos de extensión son solo un azúcar sintáctico, y en realidad no son miembros de la clase que extienden. Esto significa que no pueden romper la encapsulación: solo tienen acceso a `public` campos, propiedades y métodos `public` (o cuando se implementan en el mismo ensamblaje, `internal` ).

## Métodos de extensión genéricos

Al igual que otros métodos, los métodos de extensión pueden usar genéricos. Por ejemplo:

```
static class Extensions
{
```

```
    public static bool HasMoreThanThreeElements<T>(this IEnumerable<T> enumerable)
    {
        return enumerable.Take(4).Count() > 3;
    }
}
```

y llamarlo sería como:

```
IEnumerable<int> numbers = new List<int> {1,2,3,4,5,6};
var hasMoreThanThreeElements = numbers.HasMoreThanThreeElements();
```

Ver demostración

Del mismo modo para múltiples argumentos de tipo:

```
public static TU GenericExt<T, TU>(this T obj)
{
     TU ret = default(TU);
     // do some stuff with obj
     return ret;
}
```

Llamarlo sería como:

```
IEnumerable<int> numbers = new List<int> {1,2,3,4,5,6};
var result = numbers.GenericExt<IEnumerable<int>,String>();
```

Ver demostración

También puede crear métodos de extensión para tipos parcialmente enlazados en varios tipos genéricos:

```
class MyType<T1, T2>
{
}

static class Extensions
{
    public static void Example<T>(this MyType<int, T> test)
    {
    }
}
```

Llamarlo sería como:

```
MyType<int, string> t = new MyType<int, string>();
t.Example();
```

Ver demostración

También puede especificar restricciones de tipo con `where` :

```
public static bool IsDefault<T>(this T obj) where T : struct, IEquatable<T>
{
     return EqualityComparer<T>.Default.Equals(obj, default(T));
}
```

Código de llamada:

```
int number = 5;
var IsDefault = number.IsDefault();
```

Ver demostración

## Métodos de extensión de despacho en función del tipo estático.

El tipo estático (tiempo de compilación) se usa en lugar del dinámico (tipo de tiempo de ejecución) para hacer coincidir los parámetros.

```
public class Base
{
    public virtual string GetName()
    {
        return "Base";
    }
}

public class Derived : Base
{
    public override string GetName()
    {
        return "Derived";
    }
}

public static class Extensions
{
    public static string GetNameByExtension(this Base item)
    {
        return "Base";
    }

    public static string GetNameByExtension(this Derived item)
    {
        return "Derived";
    }
}

public static class Program
{
    public static void Main()
    {
        Derived derived = new Derived();
        Base @base = derived;

        // Use the instance method "GetName"
        Console.WriteLine(derived.GetName()); // Prints "Derived"
        Console.WriteLine(@base.GetName()); // Prints "Derived"
```

```
        // Use the static extension method "GetNameByExtension"
        Console.WriteLine(derived.GetNameByExtension()); // Prints "Derived"
        Console.WriteLine(@base.GetNameByExtension()); // Prints "Base"
    }
}
```

Demo en vivo en .NET Fiddle

Además, el envío basado en el tipo estático no permite llamar a un método de extensión en un objeto `dynamic` :

```
public class Person
{
    public string Name { get; set; }
}

public static class ExtenionPerson
{
    public static string GetPersonName(this Person person)
    {
        return person.Name;
    }
}

dynamic person = new Person { Name = "Jon" };
var name = person.GetPersonName(); // RuntimeBinderException is thrown
```

## Los métodos de extensión no son compatibles con el código dinámico.

```
static class Program
{
    static void Main()
    {
        dynamic dynamicObject = new ExpandoObject();

        string awesomeString = "Awesome";

        // Prints True
        Console.WriteLine(awesomeString.IsThisAwesome());

        dynamicObject.StringValue = awesomeString;

        // Prints True
        Console.WriteLine(StringExtensions.IsThisAwesome(dynamicObject.StringValue));

        // No compile time error or warning, but on runtime throws RuntimeBinderException
        Console.WriteLine(dynamicObject.StringValue.IsThisAwesome());
    }
}

static class StringExtensions
{
    public static bool IsThisAwesome(this string value)
    {
        return value.Equals("Awesome");
    }
}
```

> La razón por la que [los métodos de extensión de llamada de código dinámico] no funcionan es porque en los métodos de extensión de código no dinámico regulares, se realiza una búsqueda completa de todas las clases conocidas por el compilador para una clase estática que tenga un método de extensión que coincida . La búsqueda se realiza en orden según el anidamiento del espacio de nombres y está disponible `using` directivas en cada espacio de nombres.
>
> Eso significa que para que la invocación de un método de extensión dinámica se resuelva correctamente, de alguna manera, el DLR tiene que saber *en tiempo de ejecución* qué todos los anidamientos de espacio de nombres y directivas de `using` estaban *en su código fuente* . No tenemos un mecanismo útil para codificar toda esa información en el sitio de la llamada. Consideramos la posibilidad de inventar un mecanismo de este tipo, pero decidimos que era un costo demasiado alto y producía demasiado riesgo programado para que valiera la pena.

Fuente

## Métodos de extensión como envoltorios fuertemente tipados.

Los métodos de extensión se pueden usar para escribir envoltorios fuertemente tipados para objetos similares a diccionarios. Por ejemplo, un caché, `HttpContext.Items` at cetera ...

```
public static class CacheExtensions
{
    public static void SetUserInfo(this Cache cache, UserInfo data) =>
        cache["UserInfo"] = data;

    public static UserInfo GetUserInfo(this Cache cache) =>
        cache["UserInfo"] as UserInfo;
}
```

Este enfoque elimina la necesidad de utilizar literales de cadena como claves en todo el código base, así como la necesidad de convertir el tipo requerido durante la operación de lectura. En general, crea una forma más segura y fuertemente tipada de interactuar con objetos tan vagamente escritos como los Diccionarios.

## Métodos de extensión para el encadenamiento.

Cuando un método de extensión devuelve un valor que tiene el mismo tipo que `this` argumento, se puede usar para "encadenar" una o más llamadas de método con una firma compatible. Esto puede ser útil para tipos sellados y / o primitivos, y permite la creación de las llamadas API "fluidas" si los nombres de los métodos se leen como lenguaje humano natural.

```
void Main()
{
    int result = 5.Increment().Decrement().Increment();
    // result is now 6
}

public static class IntExtensions
```

```
{
    public static int Increment(this int number) {
        return ++number;
    }

    public static int Decrement(this int number) {
        return --number;
    }
}
```

O así

```
void Main()
{
    int[] ints = new[] { 1, 2, 3, 4, 5, 6};
    int[] a = ints.WhereEven();
    //a is { 2, 4, 6 };
    int[] b = ints.WhereEven().WhereGreaterThan(2);
    //b is { 4, 6 };
}

public static class IntArrayExtensions
{
    public static int[] WhereEven(this int[] array)
    {
        //Enumerable.* extension methods use a fluent approach
        return array.Where(i => (i%2) == 0).ToArray();
    }

    public static int[] WhereGreaterThan(this int[] array, int value)
    {
        return array.Where(i => i > value).ToArray();
    }
}
```

## Métodos de extensión en combinación con interfaces.

Es muy conveniente usar métodos de extensión con interfaces, ya que la implementación se puede almacenar fuera de clase y todo lo que se necesita para agregar alguna funcionalidad a la clase es decorar la clase con la interfaz.

```
public interface IInterface
{
   string Do()
}

public static class ExtensionMethods{
    public static string DoWith(this IInterface obj){
      //does something with IInterface instance
    }
}

public class Classy : IInterface
{
   // this is a wrapper method; you could also call DoWith() on a Classy instance directly,
   // provided you import the namespace containing the extension method
   public Do(){
```

```
        return this.DoWith();
    }
}
```

usar como

```
 var classy = new Classy();
 classy.Do(); // will call the extension
 classy.DoWith(); // Classy implements IInterface so it can also be called this way
```

## IList Ejemplo de Método de Extensión: Comparando 2 Listas

Puede usar el siguiente método de extensión para comparar el contenido de dos instancias de IList <T> del mismo tipo.

De forma predeterminada, los elementos se comparan según su orden dentro de la lista y los elementos en sí mismos, al pasar falso al parámetro `isOrdered` solo se compararán los elementos sin importar su orden.

Para que este método funcione, el tipo genérico ( `T` ) debe anular los métodos `Equals` y `GetHashCode` .

**Uso:**

```
List<string> list1 = new List<string> {"a1", "a2", null, "a3"};
List<string> list2 = new List<string> {"a1", "a2", "a3", null};

list1.Compare(list2);//this gives false
list1.Compare(list2, false);//this gives true. they are equal when the order is disregarded
```

**Método:**

```
public static bool Compare<T>(this IList<T> list1, IList<T> list2, bool isOrdered = true)
{
    if (list1 == null && list2 == null)
        return true;
    if (list1 == null || list2 == null || list1.Count != list2.Count)
        return false;

    if (isOrdered)
    {
        for (int i = 0; i < list2.Count; i++)
        {
            var l1 = list1[i];
            var l2 = list2[i];
            if (
                 (l1 == null && l2 != null) ||
                 (l1 != null && l2 == null) ||
                 (!l1.Equals(l2)))
            {
                    return false;
            }
        }
        return true;
```

```
    }
    else
    {
        List<T> list2Copy = new List<T>(list2);
        //Can be done with Dictionary without O(n^2)
        for (int i = 0; i < list1.Count; i++)
        {
            if (!list2Copy.Remove(list1[i]))
                return false;
        }
        return true;
    }
}
```

## Métodos de extensión con enumeración.

Los métodos de extensión son útiles para agregar funcionalidad a las enumeraciones.

Un uso común es implementar un método de conversión.

```
public enum YesNo
{
    Yes,
    No,
}

public static class EnumExtentions
{
    public static bool ToBool(this YesNo yn)
    {
        return yn == YesNo.Yes;
    }
    public static YesNo ToYesNo(this bool yn)
    {
        return yn ? YesNo.Yes : YesNo.No;
    }
}
```

Ahora puede convertir rápidamente su valor de enumeración a un tipo diferente. En este caso un bool.

```
bool yesNoBool = YesNo.Yes.ToBool(); // yesNoBool == true
YesNo yesNoEnum = false.ToYesNo();   // yesNoEnum == YesNo.No
```

Alternativamente, los métodos de extensión se pueden usar para agregar propiedades similares a los métodos.

```
public enum Element
{
    Hydrogen,
    Helium,
    Lithium,
    Beryllium,
    Boron,
    Carbon,
```

```
    Nitrogen,
    Oxygen
    //Etc
}

public static class ElementExtensions
{
    public static double AtomicMass(this Element element)
    {
        switch(element)
        {
            case Element.Hydrogen:  return 1.00794;
            case Element.Helium:    return 4.002602;
            case Element.Lithium:   return 6.941;
            case Element.Beryllium: return 9.012182;
            case Element.Boron:     return 10.811;
            case Element.Carbon:    return 12.0107;
            case Element.Nitrogen:  return 14.0067;
            case Element.Oxygen:    return 15.9994;
            //Etc
        }
        return double.Nan;
    }
}

var massWater = 2*Element.Hydrogen.AtomicMass() + Element.Oxygen.AtomicMass();
```

## Las extensiones y las interfaces juntas permiten el código DRY y una funcionalidad similar a la mezcla

Los métodos de extensión le permiten simplificar las definiciones de su interfaz al solo incluir la funcionalidad requerida en la interfaz y le permiten definir métodos convenientes y sobrecargas como métodos de extensión. Las interfaces con menos métodos son más fáciles de implementar en las nuevas clases. Mantener las sobrecargas como extensiones en lugar de incluirlas en la interfaz directamente le evita copiar código repetitivo en cada implementación, lo que lo ayuda a mantener el código SECO. De hecho, esto es similar al patrón de mezcla que C # no admite.

Las extensiones de `System.Linq.Enumerable` a `IEnumerable<T>` son un gran ejemplo de esto. `IEnumerable<T>` solo requiere que la clase implementadora implemente dos métodos: `GetEnumerator()` genérico y no genérico. Pero `System.Linq.Enumerable` proporciona innumerables utilidades útiles como extensiones que permiten un consumo claro y conciso de `IEnumerable<T>` .

La siguiente es una interfaz muy simple con sobrecargas de conveniencia provistas como extensiones.

```
public interface ITimeFormatter
{
   string Format(TimeSpan span);
}

public static class TimeFormatter
{
    // Provide an overload to *all* implementers of ITimeFormatter.
    public static string Format(
        this ITimeFormatter formatter,
```

```
        int millisecondsSpan)
        => formatter.Format(TimeSpan.FromMilliseconds(millisecondsSpan));
}

// Implementations only need to provide one method. Very easy to
// write additional implementations.
public class SecondsTimeFormatter : ITimeFormatter
{
   public string Format(TimeSpan span)
   {
       return $"{(int)span.TotalSeconds}s";
   }
}

class Program
{
    static void Main(string[] args)
    {
        var formatter = new SecondsTimeFormatter();
        // Callers get two method overloads!
        Console.WriteLine($"4500ms is rougly {formatter.Format(4500)}");
        var span = TimeSpan.FromSeconds(5);
        Console.WriteLine($"{span} is formatted as {formatter.Format(span)}");
    }
}
```

## Métodos de extensión para el manejo de casos especiales.

Los métodos de extensión se pueden usar para "ocultar" el procesamiento de reglas comerciales poco elegantes que de otro modo requerirían saturar una función de llamada con las declaraciones if / then. Esto es similar y análogo al manejo de nulos con métodos de extensión. Por ejemplo,

```
public static class CakeExtensions
{
    public static Cake EnsureTrueCake(this Cake cake)
    {
        //If the cake is a lie, substitute a cake from grandma, whose cakes aren't as tasty
but are known never to be lies. If the cake isn't a lie, don't do anything and return it.
        return CakeVerificationService.IsCakeLie(cake) ? GrandmasKitchen.Get1950sCake() :
cake;
    }
}
```

```
Cake myCake = Bakery.GetNextCake().EnsureTrueCake();
myMouth.Eat(myCake);//Eat the cake, confident that it is not a lie.
```

## Usando métodos de extensión con métodos estáticos y devoluciones de llamada

Considere el uso de métodos de extensión como funciones que envuelven otro código, aquí hay un gran ejemplo que usa tanto un método estático como un método de extensión para envolver la construcción Try Catch. Haz tu código a prueba de balas ...

```
using System;
using System.Diagnostics;

namespace Samples
{
    /// <summary>
    /// Wraps a try catch statement as a static helper which uses
    /// Extension methods for the exception
    /// </summary>
    public static class Bullet
    {
        /// <summary>
        /// Wrapper for Try Catch Statement
        /// </summary>
        /// <param name="code">Call back for code</param>
        /// <param name="error">Already handled and logged exception</param>
        public static void Proof(Action code, Action<Exception> error)
        {
            try
            {
                code();
            }
            catch (Exception iox)
            {
                //extension method used here
                iox.Log("BP2200-ERR-Unexpected Error");
                //callback, exception already handled and logged
                error(iox);
            }
        }
        /// <summary>
        /// Example of a logging method helper, this is the extension method
        /// </summary>
        /// <param name="error">The Exception to log</param>
        /// <param name="messageID">A unique error ID header</param>
        public static void Log(this Exception error, string messageID)
        {
            Trace.WriteLine(messageID);
            Trace.WriteLine(error.Message);
            Trace.WriteLine(error.StackTrace);
            Trace.WriteLine("");
        }
    }
    /// <summary>
    /// Shows how to use both the wrapper and extension methods.
    /// </summary>
    public class UseBulletProofing
    {
        public UseBulletProofing()
        {
            var ok = false;
            var result = DoSomething();
            if (!result.Contains("ERR"))
            {
                ok = true;
                DoSomethingElse();
            }
        }

        /// <summary>
        /// How to use Bullet Proofing in your code.
```

```
        /// </summary>
        /// <returns>A string</returns>
        public string DoSomething()
        {
            string result = string.Empty;
            //Note that the Bullet.Proof method forces this construct.
            Bullet.Proof(() =>
            {
                //this is the code callback
                result = "DST5900-INF-No Exceptions in this code";
            }, error =>
            {
                //error is the already logged and handled exception
                //determine the base result
                result = "DTS6200-ERR-An exception happened look at console log";
                if (error.Message.Contains("SomeMarker"))
                {
                    //filter the result for Something within the exception message
                    result = "DST6500-ERR-Some marker was found in the exception";
                }
            });
            return result;
        }

        /// <summary>
        /// Next step in workflow
        /// </summary>
        public void DoSomethingElse()
        {
            //Only called if no exception was thrown before
        }
    }
}
```

## Métodos de extensión en interfaces

Una característica útil de los métodos de extensión es que puede crear métodos comunes para una interfaz. Normalmente, una interfaz no puede tener implementaciones compartidas, pero con los métodos de extensión pueden.

```
public interface IVehicle
{
    int MilesDriven { get; set; }
}

public static class Extensions
{
    public static int FeetDriven(this IVehicle vehicle)
    {
        return vehicle.MilesDriven * 5028;
    }
}
```

En este ejemplo, el método `FeetDriven` se puede utilizar en cualquier `IVehicle` . Esta lógica en este método se aplicaría a todos los `IVehicle` , por lo que se puede hacer de esta manera para que no tenga que haber un `FeetDriven` en la definición de `IVehicle` que se implementaría de la misma manera para todos los niños.

## Usando métodos de extensión para crear hermosas clases de mapeadores

Podemos crear mejores clases de asignadores con métodos de extensión. Supongamos que si tengo algunas clases DTO como

```
public class UserDTO
{
       public AddressDTO Address { get; set; }
}

public class AddressDTO
{
       public string Name { get; set; }
}
```

y necesito asignarme a las clases de modelos de vista correspondientes

```
public class UserViewModel
{
    public AddressViewModel Address { get; set; }
}

public class AddressViewModel
{
    public string Name { get; set; }
}
```

Entonces puedo crear mi clase de asignador como abajo

```
public static class ViewModelMapper
{
      public static UserViewModel ToViewModel(this UserDTO user)
      {
            return user == null ?
                null :
                new UserViewModel()
                {
                    Address = user.Address.ToViewModel()
                    // Job = user.Job.ToViewModel(),
                    // Contact = user.Contact.ToViewModel() .. and so on
                };
      }

      public static AddressViewModel ToViewModel(this AddressDTO userAddr)
      {
            return userAddr == null ?
                null :
                new AddressViewModel()
                {
                    Name = userAddr.Name
                };
      }
}
```

Entonces finalmente puedo invocar mi mapeador como abajo

```
    UserDTO userDTOObj = new UserDTO() {
            Address = new AddressDTO() {
                Name = "Address of the user"
            }
        };

    UserViewModel user = userDTOObj.ToViewModel(); // My DTO mapped to Viewmodel
```

La belleza aquí es que todos los métodos de mapeo tienen un nombre común (ToViewModel) y podemos reutilizarlos de varias maneras.

## Usar métodos de extensión para crear nuevos tipos de colección (por ejemplo, DictList)

Puede crear métodos de extensión para mejorar la usabilidad de colecciones anidadas como un `Dictionary` con un valor de `List<T>` .

Considere los siguientes métodos de extensión:

```
public static class DictListExtensions
{
    public static void Add<TKey, TValue, TCollection>(this Dictionary<TKey, TCollection> dict,
TKey key, TValue value)
            where TCollection : ICollection<TValue>, new()
    {
        TCollection list;
        if (!dict.TryGetValue(key, out list))
        {
            list = new TCollection();
            dict.Add(key, list);
        }

        list.Add(value);
    }

    public static bool Remove<TKey, TValue, TCollection>(this Dictionary<TKey, TCollection>
dict, TKey key, TValue value)
        where TCollection : ICollection<TValue>
    {
        TCollection list;
        if (!dict.TryGetValue(key, out list))
        {
            return false;
        }

        var ret = list.Remove(value);
        if (list.Count == 0)
        {
            dict.Remove(key);
        }
        return ret;
    }
}
```

Puede utilizar los métodos de extensión de la siguiente manera:

```
var dictList = new Dictionary<string, List<int>>();

dictList.Add("example", 5);
dictList.Add("example", 10);
dictList.Add("example", 15);

Console.WriteLine(String.Join(", ", dictList["example"])); // 5, 10, 15

dictList.Remove("example", 5);
dictList.Remove("example", 10);

Console.WriteLine(String.Join(", ", dictList["example"])); // 15

dictList.Remove("example", 15);

Console.WriteLine(dictList.ContainsKey("example")); // False
```

Ver demostración

Lea Métodos de extensión en línea: https://riptutorial.com/es/csharp/topic/20/metodos-de-extension

# Capítulo 117: Métodos de fecha y hora

## Examples

### DateTime.Add (TimeSpan)

```
// Calculate what day of the week is 36 days from this instant.
System.DateTime today = System.DateTime.Now;
System.TimeSpan duration = new System.TimeSpan(36, 0, 0, 0);
System.DateTime answer = today.Add(duration);
System.Console.WriteLine("{0:dddd}", answer);
```

### DateTime.AddDays (Doble)

Añadir días en un objeto dateTime.

```
DateTime today = DateTime.Now;
DateTime answer = today.AddDays(36);
Console.WriteLine("Today: {0:dddd}", today);
Console.WriteLine("36 days from today: {0:dddd}", answer);
```

También puedes restar días pasando un valor negativo:

```
DateTime today = DateTime.Now;
DateTime answer = today.AddDays(-3);
Console.WriteLine("Today: {0:dddd}", today);
Console.WriteLine("-3 days from today: {0:dddd}", answer);
```

### DateTime.AddHours (Doble)

```
double[] hours = {.08333, .16667, .25, .33333, .5, .66667, 1, 2,
                  29, 30, 31, 90, 365};
DateTime dateValue = new DateTime(2009, 3, 1, 12, 0, 0);

foreach (double hour in hours)
  Console.WriteLine("{0} + {1} hour(s) = {2}", dateValue, hour,
                    dateValue.AddHours(hour));
```

### DateTime.AddMilliseconds (Doble)

```
string dateFormat = "MM/dd/yyyy hh:mm:ss.fffffff";
DateTime date1 = new DateTime(2010, 9, 8, 16, 0, 0);
Console.WriteLine("Original date: {0} ({1:N0} ticks)\n",
                  date1.ToString(dateFormat), date1.Ticks);

DateTime date2 = date1.AddMilliseconds(1);
Console.WriteLine("Second date:   {0} ({1:N0} ticks)",
                  date2.ToString(dateFormat), date2.Ticks);
Console.WriteLine("Difference between dates: {0} ({1:N0} ticks)\n",
```

```
                  date2 - date1, date2.Ticks - date1.Ticks);

DateTime date3 = date1.AddMilliseconds(1.5);
Console.WriteLine("Third date:    {0} ({1:N0} ticks)",
                  date3.ToString(dateFormat), date3.Ticks);
Console.WriteLine("Difference between dates: {0} ({1:N0} ticks)",
                  date3 - date1, date3.Ticks - date1.Ticks);
```

## DateTime.Compare (DateTime t1, DateTime t2)

```
DateTime date1 = new DateTime(2009, 8, 1, 0, 0, 0);
DateTime date2 = new DateTime(2009, 8, 1, 12, 0, 0);
int result = DateTime.Compare(date1, date2);
string relationship;

if (result < 0)
    relationship = "is earlier than";
else if (result == 0)
    relationship = "is the same time as";
else relationship = "is later than";

Console.WriteLine("{0} {1} {2}", date1, relationship, date2);
```

## DateTime.DaysInMonth (Int32, Int32)

```
const int July = 7;
const int Feb = 2;

int daysInJuly = System.DateTime.DaysInMonth(2001, July);
Console.WriteLine(daysInJuly);

// daysInFeb gets 28 because the year 1998 was not a leap year.
int daysInFeb = System.DateTime.DaysInMonth(1998, Feb);
Console.WriteLine(daysInFeb);

// daysInFebLeap gets 29 because the year 1996 was a leap year.
int daysInFebLeap = System.DateTime.DaysInMonth(1996, Feb);
Console.WriteLine(daysInFebLeap);
```

## DateTime.AddYears (Int32)

Agregue años en el objeto dateTime:

```
DateTime baseDate = new DateTime(2000, 2, 29);
Console.WriteLine("Base Date: {0:d}\n", baseDate);

// Show dates of previous fifteen years.
for (int ctr = -1; ctr >= -15; ctr--)
   Console.WriteLine("{0,2} year(s) ago:{1:d}",
                     Math.Abs(ctr), baseDate.AddYears(ctr));

Console.WriteLine();

// Show dates of next fifteen years.
for (int ctr = 1; ctr <= 15; ctr++)
```

```
   Console.WriteLine("{0,2} year(s) from now: {1:d}",
                     ctr, baseDate.AddYears(ctr));
```

## Advertencia de funciones puras cuando se trata de DateTime

Wikipedia actualmente define una función pura de la siguiente manera:

1. La función siempre evalúa el mismo valor de resultado dado el mismo valor de argumento (s). El valor del resultado de la función no puede depender de ninguna información oculta o estado que pueda cambiar durante la ejecución del programa o entre diferentes ejecuciones del programa, ni puede depender de ninguna entrada externa de los dispositivos de E / S.
2. La evaluación del resultado no causa ningún efecto o efecto secundario observable semánticamente, como la mutación de objetos mutables o la salida a dispositivos de E / S

Como desarrollador, debes estar al tanto de los métodos puros y encontrarás muchos en muchos aspectos. He visto que muchos desarrolladores junior están trabajando con los métodos de clase DateTime. Muchos de estos son puros y, si no los conoces, puedes sorprenderte. Un ejemplo:

```
        DateTime sample = new DateTime(2016, 12, 25);
        sample.AddDays(1);
        Console.WriteLine(sample.ToShortDateString());
```

Dado el ejemplo anterior, se puede esperar que el resultado impreso en la consola sea '26 / 12/2016 'pero en realidad terminas con la misma fecha. Esto se debe a que AddDays es un método puro y no afecta la fecha original. Para obtener el resultado esperado, tendría que modificar la llamada AddDays a lo siguiente:

```
        sample = sample.AddDays(1);
```

## DateTime.Parse (String)

```
// Converts the string representation of a date and time to its DateTime equivalent

var dateTime = DateTime.Parse("14:23 22 Jul 2016");

Console.WriteLine(dateTime.ToString());
```

## DateTime.TryParse (String, DateTime)

```
// Converts the specified string representation of a date and time to its DateTime equivalent
and returns a value that indicates whether the conversion succeeded

string[] dateTimeStrings = new []{
    "14:23 22 Jul 2016",
    "99:23 2x Jul 2016",
    "22/7/2016 14:23:00"
};

foreach(var dateTimeString in dateTimeStrings){
```

```
    DateTime dateTime;

    bool wasParsed = DateTime.TryParse(dateTimeString, out dateTime);

    string result = dateTimeString +
        (wasParsed
            ? $"was parsed to {dateTime}"
            : "can't be parsed to DateTime");

    Console.WriteLine(result);
}
```

## Parse y TryParse con información de la cultura

Es posible que desee utilizarlo al analizar DateTimes de diferentes culturas (idiomas) ; a continuación, el ejemplo analiza la fecha en holandés.

```
DateTime dateResult;
var dutchDateString = "31 oktober 1999 04:20";
var dutchCulture = CultureInfo.CreateSpecificCulture("nl-NL");
DateTime.TryParse(dutchDateString, dutchCulture, styles, out dateResult);
// output {31/10/1999 04:20:00}
```

Ejemplo de Parse:

```
DateTime.Parse(dutchDateString, dutchCulture)
// output {31/10/1999 04:20:00}
```

## DateTime como inicializador en for-loop

```
// This iterates through a range between two DateTimes
// with the given iterator (any of the Add methods)

DateTime start = new DateTime(2016, 01, 01);
DateTime until = new DateTime(2016, 02, 01);

// NOTICE: As the add methods return a new DateTime you have
// to overwrite dt in the iterator like dt = dt.Add()
for (DateTime dt = start; dt < until; dt = dt.AddDays(1))
{
    Console.WriteLine("Added {0} days. Resulting DateTime: {1}",
                      (dt - start).Days, dt.ToString());
}
```

*Iterar en un `TimeSpan` funciona de la misma manera.*

## DateTime ToString, ToShortDateString, ToLongDateString y ToString formateados

```
using System;

public class Program
{
```

```
    public static void Main()
    {
        var date = new DateTime(2016,12,31);

        Console.WriteLine(date.ToString());        //Outputs: 12/31/2016 12:00:00 AM
        Console.WriteLine(date.ToShortDateString()); //Outputs: 12/31/2016
        Console.WriteLine(date.ToLongDateString()); //Outputs: Saturday, December 31, 2016
        Console.WriteLine(date.ToString("dd/MM/yyyy"));    //Outputs: 31/12/2016
    }
}
```

## Fecha actual

Para obtener la fecha actual, use la propiedad `DateTime.Today` . Esto devuelve un objeto `DateTime` con la fecha de hoy. Cuando esto se convierte entonces `.ToString()` se hace en la localidad de su sistema de forma predeterminada.

Por ejemplo:

```
Console.WriteLine(DateTime.Today);
```

Escribe la fecha de hoy, en tu formato local a la consola.

## Formateo de fecha y hora

### Formato de fecha y hora estándar

DateTimeFormatInfo especifica un conjunto de especificadores para el formateo de fecha y hora simples. Cada especificador corresponde a un patrón de formato DateTimeFormatInfo particular.

```
//Create datetime
DateTime dt = new DateTime(2016,08,01,18,50,23,230);

var t = String.Format("{0:t}", dt); // "6:50 PM"                             ShortTime
var d = String.Format("{0:d}", dt); // "8/1/2016"                            ShortDate
var T = String.Format("{0:T}", dt); // "6:50:23 PM"                          LongTime
var D = String.Format("{0:D}", dt); // "Monday, August 1, 2016"              LongDate
var f = String.Format("{0:f}", dt); // "Monday, August 1, 2016 6:50 PM"
LongDate+ShortTime
var F = String.Format("{0:F}", dt); // "Monday, August 1, 2016 6:50:23 PM"   FullDateTime
var g = String.Format("{0:g}", dt); // "8/1/2016 6:50 PM"
ShortDate+ShortTime
var G = String.Format("{0:G}", dt); // "8/1/2016 6:50:23 PM"
ShortDate+LongTime
var m = String.Format("{0:m}", dt); // "August 1"                            MonthDay
var y = String.Format("{0:y}", dt); // "August 2016"                         YearMonth
var r = String.Format("{0:r}", dt); // "SMon, 01 Aug 2016 18:50:23 GMT"      RFC1123
var s = String.Format("{0:s}", dt); // "2016-08-01T18:50:23"                 SortableDateTime
var u = String.Format("{0:u}", dt); // "2016-08-01 18:50:23Z"
UniversalSortableDateTime
```

### Formato de fecha y hora personalizado

Existen los siguientes especificadores de formato personalizados:

- `y` (año)
- `M` (mes)
- `d` (día)
- `h` (hora 12)
- `H` (hora 24)
- `m` (minuto)
- `s` (segundo)
- `f` (segunda fracción)
- `F` (segunda fracción, se recortan los ceros finales)
- `t` (PM o AM)
- `z` (zona horaria).

```
var year =       String.Format("{0:y yy yyy yyyy}", dt); // "16 16 2016 2016"  year
var month =      String.Format("{0:M MM MMM MMMM}", dt); // "8 08 Aug August"  month
var day =        String.Format("{0:d dd ddd dddd}", dt); // "1 01 Mon Monday"  day
var hour =       String.Format("{0:h hh H HH}",     dt); // "6 06 18 18"       hour 12/24
var minute =     String.Format("{0:m mm}",          dt); // "50 50"            minute
var secound =    String.Format("{0:s ss}",          dt); // "23 23"            second
var fraction =   String.Format("{0:f ff fff ffff}", dt); // "2 23 230 2300"    sec.fraction
var fraction2 =  String.Format("{0:F FF FFF FFFF}", dt); // "2 23 23 23"       without zeroes
var period =     String.Format("{0:t tt}",          dt); // "P PM"             A.M. or P.M.
var zone =       String.Format("{0:z zz zzz}",      dt); // "+0 +00 +00:00"    time zone
```

También puede usar el separador de fecha `/` (barra) y el separador de tiempo `:` (dos puntos).

Por ejemplo de código

Para más información MSDN .

## DateTime.ParseExact (String, String, IFormatProvider)

Convierte la representación de cadena especificada de una fecha y hora en su equivalente de DateTime utilizando el formato especificado y la información de formato específico de la cultura. El formato de la representación de cadena debe coincidir exactamente con el formato especificado.

**Convertir una cadena de formato específico a DateTime equivalente**

Digamos que tenemos una cadena DateTime específica de la cultura `08-07-2016 11:30:12 PM` como `08-07-2016 11:30:12 PM` `MM-dd-yyyy hh:mm:ss tt` y queremos que se convierta en un objeto `DateTime` equivalente

```
string str = "08-07-2016 11:30:12 PM";
DateTime date = DateTime.ParseExact(str, "MM-dd-yyyy hh:mm:ss tt",
CultureInfo.CurrentCulture);
```

**Convierta una cadena de fecha y hora en un objeto `DateTime` equivalente sin ningún formato de cultura específico**

Digamos que tenemos una cadena DateTime en formato `dd-MM-yy hh:mm:ss tt` y queremos que se

convierta en un objeto `DateTime` equivalente, sin ninguna información de cultura específica

```
string str = "17-06-16 11:30:12 PM";
DateTime date = DateTime.ParseExact(str, "dd-MM-yy hh:mm:ss tt",
CultureInfo.InvariantCulture);
```

**Convierta una cadena de fecha y hora en un objeto DateTime equivalente sin ningún formato de cultura específico con un formato diferente**

Digamos que tenemos una cadena de fecha, ejemplo como '23 -12-2016 'o '12 / 23/2016' y queremos que se convierta en un objeto `DateTime` equivalente, sin ninguna información de cultura específica

```
   string date =  '23-12-2016' or date = 12/23/2016';
   string[] formats = new string[] {"dd-MM-yyyy","MM/dd/yyyy"}; // even can add more possible
formats.
   DateTime date = DateTime.ParseExact(date,formats,
CultureInfo.InvariantCulture,DateTimeStyles.None);
```

**NOTA: Es necesario agregar `System.Globalization` para CultureInfo Class**

## DateTime.TryParseExact (String, String, IFormatProvider, DateTimeStyles, DateTime)

Convierte la representación de cadena especificada de una fecha y hora en su equivalente de DateTime utilizando el formato especificado, la información del formato específico de la cultura y el estilo. El formato de la representación de cadena debe coincidir exactamente con el formato especificado. El método devuelve un valor que indica si la conversión se realizó correctamente.

Por ejemplo

```
CultureInfo enUS = new CultureInfo("en-US");
string dateString;
System.DateTime dateValue;
```

Fecha de análisis sin banderas de estilo.

```
dateString = " 5/01/2009 8:30 AM";
if (DateTime.TryParseExact(dateString, "g", enUS, DateTimeStyles.None, out dateValue))
{
   Console.WriteLine("Converted '{0}' to {1} ({2}).", dateString, dateValue, dateValue.Kind);
}
else
{
   Console.WriteLine("'{0}' is not in an acceptable format.", dateString);
}


// Allow a leading space in the date string.
if(DateTime.TryParseExact(dateString, "g", enUS, DateTimeStyles.AllowLeadingWhite, out
dateValue))
{
```

```
   Console.WriteLine("Converted '{0}' to {1} ({2}).", dateString, dateValue, dateValue.Kind);
else
{
   Console.WriteLine("'{0}' is not in an acceptable format.", dateString);
}
```

Usa formatos personalizados con M y MM.

```
dateString = "5/01/2009 09:00";
if(DateTime.TryParseExact(dateString, "M/dd/yyyy hh:mm", enUS, DateTimeStyles.None, out
dateValue))
{
    Console.WriteLine("Converted '{0}' to {1} ({2}).", dateString, dateValue, dateValue.Kind);
}
else
{
   Console.WriteLine("'{0}' is not in an acceptable format.", dateString);
}

// Allow a leading space in the date string.
if(DateTime.TryParseExact(dateString, "MM/dd/yyyy hh:mm", enUS, DateTimeStyles.None, out
dateValue))
{
   Console.WriteLine("Converted '{0}' to {1} ({2}).", dateString, dateValue, dateValue.Kind);
}
else
{
   Console.WriteLine("'{0}' is not in an acceptable format.", dateString);
}
```

Analizar una cadena con información de zona horaria.

```
dateString = "05/01/2009 01:30:42 PM -05:00";
if (DateTime.TryParseExact(dateString, "MM/dd/yyyy hh:mm:ss tt zzz", enUS,
DateTimeStyles.None, out dateValue))
{
    Console.WriteLine("Converted '{0}' to {1} ({2}).", dateString, dateValue, dateValue.Kind);
}
else
{
   Console.WriteLine("'{0}' is not in an acceptable format.", dateString);
}


// Allow a leading space in the date string.
if (DateTime.TryParseExact(dateString, "MM/dd/yyyy hh:mm:ss tt zzz", enUS,
DateTimeStyles.AdjustToUniversal, out dateValue))
{
   Console.WriteLine("Converted '{0}' to {1} ({2}).", dateString, dateValue, dateValue.Kind);
}
else
{
   Console.WriteLine("'{0}' is not in an acceptable format.", dateString);
}
```

Analizar una cadena que representa UTC.

```
dateString = "2008-06-11T16:11:20.0904778Z";
if(DateTime.TryParseExact(dateString, "o", CultureInfo.InvariantCulture, DateTimeStyles.None,
out dateValue))
{
   Console.WriteLine("Converted '{0}' to {1} ({2}).", dateString, dateValue, dateValue.Kind);
}
else
{
  Console.WriteLine("'{0}' is not in an acceptable format.", dateString);
}

if (DateTime.TryParseExact(dateString, "o", CultureInfo.InvariantCulture,
DateTimeStyles.RoundtripKind, out dateValue))
{
   Console.WriteLine("Converted '{0}' to {1} ({2}).", dateString, dateValue, dateValue.Kind);
}
else
{
   Console.WriteLine("'{0}' is not in an acceptable format.", dateString);
}
```

## Salidas

```
' 5/01/2009 8:30 AM' is not in an acceptable format.
Converted ' 5/01/2009 8:30 AM' to 5/1/2009 8:30:00 AM (Unspecified).
Converted '5/01/2009 09:00' to 5/1/2009 9:00:00 AM (Unspecified).
'5/01/2009 09:00' is not in an acceptable format.
Converted '05/01/2009 01:30:42 PM -05:00' to 5/1/2009 11:30:42 AM (Local).
Converted '05/01/2009 01:30:42 PM -05:00' to 5/1/2009 6:30:42 PM (Utc).
Converted '2008-06-11T16:11:20.0904778Z' to 6/11/2008 9:11:20 AM (Local).
Converted '2008-06-11T16:11:20.0904778Z' to 6/11/2008 4:11:20 PM (Utc).
```

Lea Métodos de fecha y hora en línea: https://riptutorial.com/es/csharp/topic/1587/metodos-de-fecha-y-hora

# Capítulo 118: Microsoft.Exchange.WebServices

## Examples

### Recuperar la configuración de fuera de la oficina del usuario especificado

Primero vamos a crear un objeto `ExchangeManager` , donde el constructor se conectará con los servicios para nosotros. También tiene un método `GetOofSettings` , que devolverá el objeto `OofSettings` para la dirección de correo electrónico especificada:

```
using System;
using System.Web.Configuration;
using Microsoft.Exchange.WebServices.Data;

namespace SetOutOfOffice
{
    class ExchangeManager
    {
        private ExchangeService Service;

        public ExchangeManager()
        {
            var password =
WebConfigurationManager.ConnectionStrings["Password"].ConnectionString;
            Connect("exchangeadmin", password);
        }
        private void Connect(string username, string password)
        {
            var service = new ExchangeService(ExchangeVersion.Exchange2010_SP2);
            service.Credentials = new WebCredentials(username, password);
            service.AutodiscoverUrl("autodiscoveremail@domain.com" ,
RedirectionUrlValidationCallback);

            Service = service;
        }
        private static bool RedirectionUrlValidationCallback(string redirectionUrl)
        {
            return
redirectionUrl.Equals("https://mail.domain.com/autodiscover/autodiscover.xml");
        }
        public OofSettings GetOofSettings(string email)
        {
            return Service.GetUserOofSettings(email);
        }
    }
}
```

Ahora podemos llamar a esto en otro lugar así:

```
var em = new ExchangeManager();
var oofSettings = em.GetOofSettings("testemail@domain.com");
```

## Actualizar la configuración de fuera de la oficina del usuario específico

Usando la clase a continuación, podemos conectarnos a Exchange y luego establecer la configuración de fuera de la oficina de un usuario específico con `UpdateUserOof` :

```
using System;
using System.Web.Configuration;
using Microsoft.Exchange.WebServices.Data;

class ExchangeManager
{
    private ExchangeService Service;

    public ExchangeManager()
    {
        var password = WebConfigurationManager.ConnectionStrings["Password"].ConnectionString;
        Connect("exchangeadmin", password);
    }
    private void Connect(string username, string password)
    {
        var service = new ExchangeService(ExchangeVersion.Exchange2010_SP2);
        service.Credentials = new WebCredentials(username, password);
        service.AutodiscoverUrl("autodiscoveremail@domain.com" ,
RedirectionUrlValidationCallback);

        Service = service;
    }
    private static bool RedirectionUrlValidationCallback(string redirectionUrl)
    {
        return redirectionUrl.Equals("https://mail.domain.com/autodiscover/autodiscover.xml");
    }
    /// <summary>
    /// Updates the given user's Oof settings with the given details
    /// </summary>
    public void UpdateUserOof(int oofstate, DateTime starttime, DateTime endtime, int
externalaudience, string internalmsg, string externalmsg, string emailaddress)
    {
        var newSettings = new OofSettings
        {
            State = (OofState)oofstate,
            Duration = new TimeWindow(starttime, endtime),
            ExternalAudience = (OofExternalAudience)externalaudience,
            InternalReply = internalmsg,
            ExternalReply = externalmsg
        };

        Service.SetUserOofSettings(emailaddress, newSettings);
    }
}
```

Actualice la configuración del usuario con lo siguiente:

```
var oofState = 1;
var startDate = new DateTime(01,08,2016);
var endDate = new DateTime(15,08,2016);
var externalAudience = 1;
var internalMessage = "I am not in the office!";
var externalMessage = "I am not in the office <strong>and neither are you!</strong>"
```

```
var theUser = "theuser@domain.com";

var em = new ExchangeManager();
em.UpdateUserOof(oofstate, startDate, endDate, externalAudience, internalMessage,
externalMessage, theUser);
```

Tenga en cuenta que puede formatear los mensajes utilizando etiquetas `html` estándar.

Lea Microsoft.Exchange.WebServices en línea:
https://riptutorial.com/es/csharp/topic/4863/microsoft-exchange-webservices

# Capítulo 119: Modificadores de acceso

## Observaciones

Si se omite el modificador de acceso,

- las clases son `internal` por defecto
- Los métodos son por privado `private`
- Los captadores y definidores heredan el modificador de la propiedad, de forma predeterminada, esto es `private`

Los modificadores de acceso en los configuradores o en los que obtienen propiedades solo pueden restringir el acceso, no `public string someProperty {get; private set;}` : `public string someProperty {get; private set;}`

## Examples

### público

La palabra clave `public` hace que una clase (incluidas las clases anidadas), propiedad, método o campo esté disponible para cada consumidor:

```
public class Foo()
{
    public string SomeProperty { get; set; }

    public class Baz
    {
        public int Value { get; set; }
    }
}

public class Bar()
{
    public Bar()
    {
        var myInstance = new Foo();
        var someValue = foo.SomeProperty;
        var myNestedInstance = new Foo.Baz();
        var otherValue = myNestedInstance.Value;
    }
}
```

### privado

La palabra clave `private` marca propiedades, métodos, campos y clases anidadas para usar solo dentro de la clase:

```
public class Foo()
{
```

```
    private string someProperty { get; set; }

    private class Baz
    {
        public string Value { get; set; }
    }

    public void Do()
    {
        var baz = new Baz { Value = 42 };
    }
}

public class Bar()
{
    public Bar()
    {
        var myInstance = new Foo();

        // Compile Error - not accessible due to private modifier
        var someValue = foo.someProperty;
        // Compile Error - not accessible due to private modifier
        var baz = new Foo.Baz();
    }
}
```

## interno

La palabra clave interna hace que una clase (incluidas las clases anidadas), propiedad, método o campo estén disponibles para todos los consumidores en el mismo ensamblaje:

```
internal class Foo
{
    internal string SomeProperty {get; set;}
}

internal class Bar
{
    var myInstance = new Foo();
    internal string SomeField = foo.SomeProperty;

    internal class Baz
    {
        private string blah;
        public int N { get; set; }
    }
}
```

Esto se puede romper para permitir que un ensamblaje de prueba acceda al código mediante la adición de código al archivo AssemblyInfo.cs:

```
using System.Runtime.CompilerServices;

[assembly:InternalsVisibleTo("MyTests")]
```

## protegido

El campo de marcas de palabras clave `protected` , las propiedades de los métodos y las clases anidadas para su uso solo dentro de la misma clase y las clases derivadas:

```
public class Foo()
{
    protected void SomeFooMethod()
    {
        //do something
    }

    protected class Thing
    {
        private string blah;
        public int N { get; set; }
    }
}

public class Bar() : Foo
{
    private void someBarMethod()
    {
        SomeFooMethod(); // inside derived class
        var thing = new Thing(); // can use nested class
    }
}

public class Baz()
{
    private void someBazMethod()
    {
        var foo = new Foo();
        foo.SomeFooMethod(); //not accessible due to protected modifier
    }
}
```

## protegido interno

El campo, los métodos, las propiedades y las clases anidadas de las marcas de palabras clave `protected internal` para usar dentro del mismo conjunto o clases derivadas en otro conjunto:

Asamblea 1

```
public class Foo
{
    public string MyPublicProperty { get; set; }
    protected internal string MyProtectedInternalProperty  { get; set; }

    protected internal class MyProtectedInternalNestedClass
    {
        private string blah;
        public int N { get; set; }
    }
}

public class Bar
{
    void MyMethod1()
```

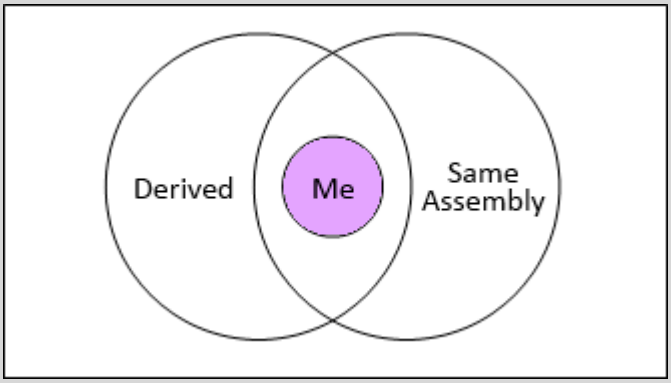
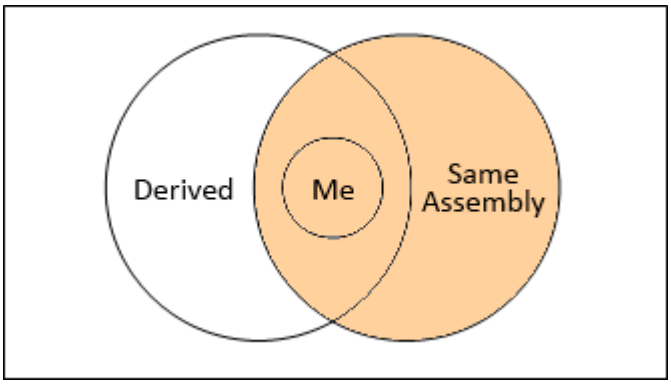
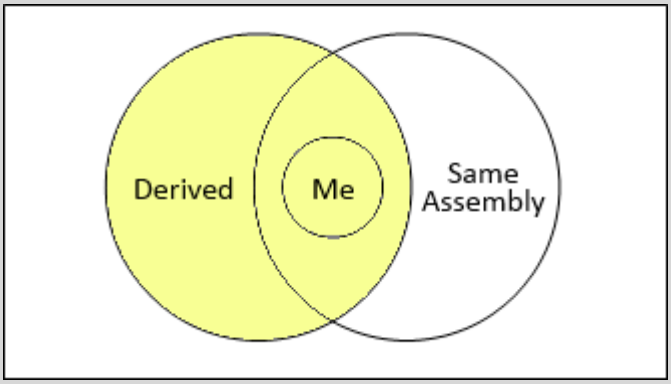
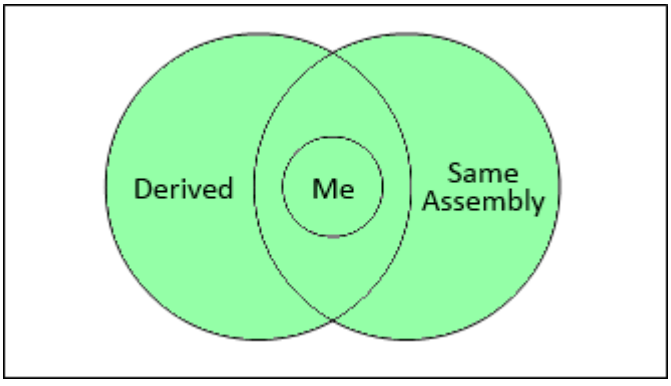
```
    {
        Foo foo = new Foo();
        var myPublicProperty = foo.MyPublicProperty;
        var myProtectedInternalProperty = foo.MyProtectedInternalProperty;
        var myProtectedInternalNestedInstance =
            new Foo.MyProtectedInternalNestedClass();
    }
}
```

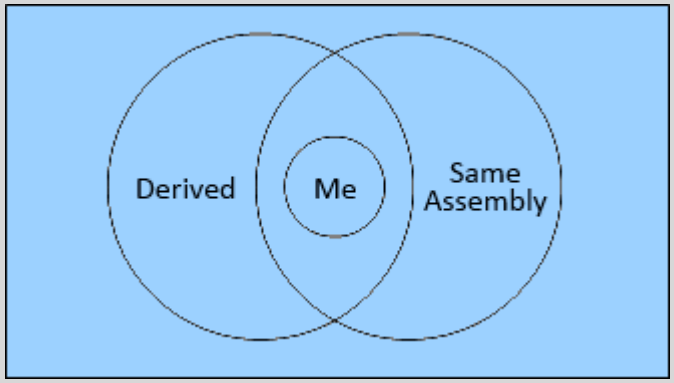
Asamblea 2

```
public class Baz : Foo
{
    void MyMethod1()
    {
        var myPublicProperty = MyPublicProperty;
        var myProtectedInternalProperty = MyProtectedInternalProperty;
        var thing = new MyProtectedInternalNestedClass();
    }

    void MyMethod2()
    {
        Foo foo = new Foo();
        var myPublicProperty = foo.MyPublicProperty;

        // Compile Error
        var myProtectedInternalProperty = foo.MyProtectedInternalProperty;
        // Compile Error
        var myProtectedInternalNestedInstance =
            new Foo.MyProtectedInternalNestedClass();
    }

}

public class Qux
{
    void MyMethod1()
    {
        Baz baz = new Baz();
        var myPublicProperty = baz.MyPublicProperty;

        // Compile Error
        var myProtectedInternalProperty = baz.MyProtectedInternalProperty;
        // Compile Error
        var myProtectedInternalNestedInstance =
            new Baz.MyProtectedInternalNestedClass();
    }

    void MyMethod2()
    {
        Foo foo = new Foo();
        var myPublicProperty = foo.MyPublicProperty;

        //Compile Error
        var myProtectedInternalProperty = foo.MyProtectedInternalProperty;
        // Compile Error
        var myProtectedInternalNestedInstance =
            new Foo.MyProtectedInternalNestedClass();
    }
}
```

## Diagramas de modificadores de acceso

Aquí están todos los modificadores de acceso en los diagramas de venn, desde más limitados hasta más accesibles:

| Modificador de acceso | Diagrama |
|---|---|
| privado |   |
| interno |   |
| protegido |   |
| protegido interno |   |

| Modificador de acceso | Diagrama |
| --- | --- |
| público |  |



A continuación puede encontrar más información.

Lea Modificadores de acceso en línea: https://riptutorial.com/es/csharp/topic/960/modificadores-de-acceso

# Capítulo 120: Nombre del operador

## Introducción

El operador `nameof` permite obtener el nombre de una **variable** , **tipo** o **miembro** en forma de cadena sin tener que codificarlo como un literal.

La operación se evalúa en tiempo de compilación, lo que significa que puede cambiar el nombre de un identificador al que se hace referencia, mediante la función de cambio de nombre de un IDE, y la cadena de nombre se actualizará con él.

## Sintaxis

- nameof (expresión)

## Examples

### Uso básico: imprimiendo un nombre de variable

El operador `nameof` permite obtener el nombre de una variable, tipo o miembro en forma de cadena sin tener que codificarlo como un literal. La operación se evalúa en tiempo de compilación, lo que significa que puede cambiar el nombre mediante la función de cambio de nombre de un IDE, un identificador al que se hace referencia y la cadena de nombre se actualizará con él.

```
var myString = "String Contents";
Console.WriteLine(nameof(myString));
```

Saldría

    myString

porque el nombre de la variable es "myString". Refactorizar el nombre de la variable cambiaría la cadena.

Si se llama en un tipo de referencia, el operador `nameof` devuelve el nombre de la referencia actual, *no* el nombre o el tipo de nombre del objeto subyacente. Por ejemplo:

```
string greeting = "Hello!";
Object mailMessageBody = greeting;

Console.WriteLine(nameof(greeting)); // Returns "greeting"
Console.WriteLine(nameof(mailMessageBody)); // Returns "mailMessageBody", NOT "greeting"!
```

### Imprimiendo un nombre de parámetro

### Retazo

```
public void DoSomething(int paramValue)
{
    Console.WriteLine(nameof(paramValue));
}

...

int myValue = 10;
DoSomething(myValue);
```

### Salida de consola

paramValue

## Aumento de evento PropertyChanged

### Retazo

```
public class Person : INotifyPropertyChanged
{
    private string _address;

    public event PropertyChangedEventHandler PropertyChanged;

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

    public string Address
    {
        get { return _address; }
        set
        {
            if (_address == value)
            {
                return;
            }

            _address = value;
            OnPropertyChanged(nameof(Address));
        }
    }
}

...

var person = new Person();
person.PropertyChanged += (s,e) => Console.WriteLine(e.PropertyName);

person.Address = "123 Fake Street";
```

### Salida de consola

Dirección

## Manejo de eventos de PropertyChanged

### Retazo

```
public class BugReport : INotifyPropertyChanged
{
    public string Title { ... }
    public BugStatus Status { ... }
}

...

private void BugReport_PropertyChanged(object sender, PropertyChangedEventArgs e)
{
    var bugReport = (BugReport)sender;

    switch (e.PropertyName)
    {
        case nameof(bugReport.Title):
            Console.WriteLine("{0} changed to {1}", e.PropertyName, bugReport.Title);
            break;

        case nameof(bugReport.Status):
            Console.WriteLine("{0} changed to {1}", e.PropertyName, bugReport.Status);
            break;
    }
}

...

var report = new BugReport();
report.PropertyChanged += BugReport_PropertyChanged;

report.Title = "Everything is on fire and broken";
report.Status = BugStatus.ShowStopper;
```

### Salida de consola

Título cambiado a Todo está en llamas y roto

Estado cambiado a ShowStopper

## Aplicado a un parámetro de tipo genérico

### Retazo

```
public class SomeClass<TItem>
{
    public void PrintTypeName()
    {
        Console.WriteLine(nameof(TItem));
    }
}

...

var myClass = new SomeClass<int>();
```

```
myClass.PrintTypeName();

Console.WriteLine(nameof(SomeClass<int>));
```

**Salida de consola**

Tiempo

Algo de clase

## Aplicado a identificadores calificados

**Retazo**

```
Console.WriteLine(nameof(CompanyNamespace.MyNamespace));
Console.WriteLine(nameof(MyClass));
Console.WriteLine(nameof(MyClass.MyNestedClass));
Console.WriteLine(nameof(MyNamespace.MyClass.MyNestedClass.MyStaticProperty));
```

**Salida de consola**

MyNamespace

Mi clase

MyNestedClass

MyStaticProperty

## Verificación de argumentos y cláusulas de guardia

Preferir

```
public class Order
{
    public OrderLine AddOrderLine(OrderLine orderLine)
    {
        if (orderLine == null) throw new ArgumentNullException(nameof(orderLine));
        ...
    }
}
```

Terminado

```
public class Order
{
    public OrderLine AddOrderLine(OrderLine orderLine)
    {
        if (orderLine == null) throw new ArgumentNullException("orderLine");
        ...
    }
}
```

El uso de la función `nameof` hace que sea más fácil refactorizar los parámetros del método.

### Enlaces de acción MVC fuertemente tipados

En lugar de lo habitual, escrito a la ligera:

```
@Html.ActionLink("Log in", "UserController", "LogIn")
```

Ahora puedes hacer enlaces de acción fuertemente escritos:

```
@Html.ActionLink("Log in", @typeof(UserController), @nameof(UserController.LogIn))
```

Ahora, si desea refactorizar su código y cambiar el nombre del método `UserController.LogIn` a `UserController.SignIn` , no debe preocuparse por buscar todas las ocurrencias de cadenas. El compilador hará el trabajo.

Lea Nombre del operador en línea: https://riptutorial.com/es/csharp/topic/80/nombre-del-operador

# Capítulo 121: O (n) Algoritmo para rotación circular de una matriz

## Introducción

En mi camino hacia el estudio de la programación ha habido problemas simples, pero interesantes para resolver como ejercicios. Uno de esos problemas era rotar una matriz (u otra colección) por un cierto valor. Aquí compartiré contigo una fórmula simple para hacerlo.

## Examples

### Ejemplo de un método genérico que rota una matriz en un turno dado

Me gustaría señalar que giramos a la izquierda cuando el valor de cambio es negativo y giramos a la derecha cuando el valor es positivo.

```
    public static void Main()
    {
        int[] array = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
        int shiftCount = 1;
        Rotate(ref array, shiftCount);
        Console.WriteLine(string.Join(", ", array));
        // Output: [10, 1, 2, 3, 4, 5, 6, 7, 8, 9]

        array = new []{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
        shiftCount = 15;
        Rotate(ref array, shiftCount);
        Console.WriteLine(string.Join(", ", array));
        // Output: [6, 7, 8, 9, 10, 1, 2, 3, 4, 5]

        array = new[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
        shiftCount = -1;
        Rotate(ref array, shiftCount);
        Console.WriteLine(string.Join(", ", array));
        // Output: [2, 3, 4, 5, 6, 7, 8, 9, 10, 1]

        array = new[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
        shiftCount = -35;
        Rotate(ref array, shiftCount);
        Console.WriteLine(string.Join(", ", array));
        // Output: [6, 7, 8, 9, 10, 1, 2, 3, 4, 5]
    }

    private static void Rotate<T>(ref T[] array, int shiftCount)
    {
        T[] backupArray= new T[array.Length];

        for (int index = 0; index < array.Length; index++)
        {
            backupArray[(index + array.Length + shiftCount % array.Length) % array.Length] =
array[index];
        }
```

```
        array = backupArray;
    }
```

Lo que es importante en este código es la fórmula con la que encontramos el nuevo valor de índice después de la rotación.

**(index + array.Length + shiftCount% array.Length)% array.Length**

Aquí hay un poco más de información al respecto:

**(shiftCount% array.Length)** -> normalizamos el valor del cambio para que esté en la longitud del array (ya que en un array con la longitud 10, el cambio de 1 u 11 es lo mismo, lo mismo ocurre con -1 y -11) .

**array.Length + (shiftCount% array.Length)** -> esto se hace debido a las rotaciones a la izquierda para asegurarnos de que no ingresamos en un índice negativo, sino que lo giramos hasta el final de la matriz. Sin él para una matriz con longitud 10 para el índice 0 y una rotación -1 entraríamos en un número negativo (-1) y no obtendríamos el valor del índice de rotación real, que es 9. (10 + (-1% 10) = 9)

**index + array.Length + (shiftCount% array.Length)** -> no hay mucho que decir aquí ya que aplicamos la rotación al índice para obtener el nuevo índice. (0 + 10 + (-1% 10) = 9)

**index + array.Length + (shiftCount% array.Length)% array.Length** -> la segunda normalización es asegurarse de que el nuevo valor de índice no salga de la matriz, sino que rote el valor al principio de la matriz. Es para rotaciones a la derecha, ya que en una matriz con longitud 10 sin ella para el índice 9 y una rotación 1 entraríamos en el índice 10, que está fuera de la matriz, y no obtendremos el valor del índice de rotación real es 0. ((9 + 10 + (1% 10))% 10 = 0)

Lea O (n) Algoritmo para rotación circular de una matriz en línea:
https://riptutorial.com/es/csharp/topic/9770/o--n--algoritmo-para-rotacion-circular-de-una-matriz

# Capítulo 122: ObservableCollection

## Examples

### Inicializar ObservableCollection

`ObservableCollection` es una colección de tipo `T` como `List<T>` que significa que contiene objetos de tipo `T`

De la documentación leemos que:

> `ObservableCollection` representa una recopilación de datos dinámica que proporciona notificaciones cuando se agregan o eliminan elementos o cuando se actualiza toda la lista.

La diferencia clave con respecto a otras colecciones es que `ObservableCollection` implementa las interfaces `INotifyCollectionChanged` e `INotifyPropertyChanged` e inmediatamente `INotifyPropertyChanged` un evento de notificación cuando se agrega o elimina un nuevo objeto y cuando se borra la colección.

Esto es especialmente útil para conectar la interfaz de usuario y el backend de una aplicación sin tener que escribir código adicional porque cuando se agrega o se elimina un objeto de una colección observable, la interfaz de usuario se actualiza automáticamente.

El primer paso para utilizarlo es incluir

```
using System.Collections.ObjectModel
```

Puede crear una instancia vacía de una colección, por ejemplo, de tipo `string`

```
ObservableCollection<string> collection = new ObservableCollection<string>();
```

o una instancia que está llena de datos

```
ObservableCollection<string> collection = new ObservableCollection<string>()
{
 "First_String", "Second_String"
};
```

Recuerde que, como en toda la colección de IList, el índice comienza desde 0 ( propiedad de IList.Item ).

Lea ObservableCollection en línea:
https://riptutorial.com/es/csharp/topic/7351/observablecollection--t-

# Capítulo 123: Operaciones de cadena comunes

## Examples

### Dividir una cadena por carácter específico

```
string helloWorld = "hello world, how is it going?";
string[] parts1 = helloWorld.Split(',');

//parts1: ["hello world", " how is it going?"]

string[] parts2 = helloWorld.Split(' ');

//parts2: ["hello", "world,", "how", "is", "it", "going?"]
```

### Obtención de subcadenas de una cadena dada

```
string helloWorld = "Hello World!";
string world = helloWorld.Substring(6); //world = "World!"
string hello = helloWorld.Substring(0,5); // hello = "Hello"
```

`Substring` devuelve la cadena hacia arriba desde un índice dado, o entre dos índices (ambos inclusive).

### Determine si una cadena comienza con una secuencia dada

```
string HelloWorld = "Hello World";
HelloWorld.StartsWith("Hello"); // true
HelloWorld.StartsWith("Foo"); // false
```

#### Encontrar una cadena dentro de una cadena

Usando `System.String.Contains` puedes averiguar si una cadena en particular existe dentro de una cadena. El método devuelve un valor booleano, verdadero si la cadena existe, de lo contrario es falso

```
string s = "Hello World";
bool stringExists = s.Contains("ello");  //stringExists =true as the string contains the
substring
```

### Recorte de caracteres no deseados fuera del inicio y / o final de las cadenas.

**`String.Trim()`**

```
string x = "   Hello World!    ";
```

```
string y = x.Trim(); // "Hello World!"

string q = "{(Hi!*";
string r = q.Trim( '(', '*', '{' ); // "Hi!"
```

#### `String.TrimStart()` y `String.TrimEnd()`

```
string q = "{(Hi*";
string r = q.TrimStart( '{' ); // "(Hi*"
string s = q.TrimEnd( '*' );   // "{(Hi"
```

## Formatear una cadena

Use el método `String.Format()` para reemplazar uno o más elementos en la cadena con la representación de la cadena de un objeto especificado:

```
String.Format("Hello {0} Foo {1}", "World", "Bar") //Hello World Foo Bar
```

## Unir una serie de cadenas en una nueva

```
var parts = new[] { "Foo", "Bar", "Fizz", "Buzz"};
var joined = string.Join(", ", parts);

//joined = "Foo, Bar, Fizz, Buzz"
```

## Relleno de una cuerda a una longitud fija

```
string s = "Foo";
string paddedLeft = s.PadLeft(5);        // paddedLeft = "  Foo" (pads with spaces by default)
string paddedRight = s.PadRight(6, '+'); // paddedRight = "Foo+++"
string noPadded = s.PadLeft(2);          // noPadded = "Foo" (original string is never
shortened)
```

## Construye una cadena de Array

El método `String.Join` nos ayudará a construir una cadena de matriz / lista de caracteres o cadena. Este método acepta dos parámetros. El primero es el delimitador o el separador que le ayudará a separar cada elemento de la matriz. Y el segundo parámetro es el propio Array.

**Cadena de la `char array` :**

```
string delimiter=",";
char[] charArray = new[] { 'a', 'b', 'c' };
string inputString = String.Join(delimiter, charArray);
```

**Salida** : `a,b,c` si cambiamos el `delimiter` como `""` entonces la salida se convertirá en `abc` .

**Cadena de la `List of char` :**

```
string delimiter = "|";
List<char> charList = new List<char>() { 'a', 'b', 'c' };
string inputString = String.Join(delimiter, charList);
```

**Salida** : `a|b|c`

**Cadena de la `List of Strings` :**

```
string delimiter = " ";
List<string> stringList = new List<string>() { "Ram", "is", "a","boy" };
string inputString = String.Join(delimiter, stringList);
```

**Salida** : `Ram is a boy`

**Cadena de la `array of strings` :**

```
string delimiter = "_";
string[] stringArray = new [] { "Ram", "is", "a","boy" };
string inputString = String.Join(delimiter, stringArray);
```

**Salida** : `Ram_is_a_boy`

## Formateo utilizando ToString

Por lo general, `String.Format` método `String.Format` para formatear, el `.ToString` se usa generalmente para convertir otros tipos en cadenas. Podemos especificar el formato junto con el método ToString mientras se lleva a cabo la conversión, por lo que podemos evitar un formato adicional. Déjame explicarte cómo funciona con diferentes tipos;

**Entero a la cadena formateada:**

```
int intValue = 10;
string zeroPaddedInteger = intValue.ToString("000"); // Output will be "010"
string customFormat = intValue.ToString("Input value is 0"); // output will be   "Input value
is 10"
```

**doble a la cadena con formato:**

```
double doubleValue = 10.456;
string roundedDouble = doubleValue.ToString("0.00"); // output 10.46
string integerPart = doubleValue.ToString("00");    // output 10
string customFormat = doubleValue.ToString("Input value is 0.0");  // Input value is 10.5
```

**Formato de DateTime utilizando ToString**

```
DateTime currentDate = DateTime.Now; //  {7/21/2016 7:23:15 PM}
string dateTimeString = currentDate.ToString("dd-MM-yyyy HH:mm:ss"); // "21-07-2016 19:23:15"
string dateOnlyString = currentDate.ToString("dd-MM-yyyy"); // "21-07-2016"
string dateWithMonthInWords = currentDate.ToString("dd-MMMM-yyyy HH:mm:ss"); // "21-July-2016
19:23:15"
```

## Obtención de x caracteres del lado derecho de una cadena

Visual Basic tiene funciones Izquierda, Derecha y Media que devuelven caracteres desde la izquierda, derecha y centro de una cadena. Estos métodos no existen en C #, pero pueden implementarse con `Substring()` . Se pueden implementar como métodos de extensión como los siguientes:

```
public static class StringExtensions
{
   /// <summary>
   /// VB Left function
   /// </summary>
   /// <param name="stringparam"></param>
   /// <param name="numchars"></param>
   /// <returns>Left-most numchars characters</returns>
   public static string Left( this string stringparam, int numchars )
   {
      // Handle possible Null or numeric stringparam being passed
      stringparam += string.Empty;

      // Handle possible negative numchars being passed
      numchars = Math.Abs( numchars );

      // Validate numchars parameter
      if (numchars > stringparam.Length)
         numchars = stringparam.Length;

      return stringparam.Substring( 0, numchars );
   }

   /// <summary>
   /// VB Right function
   /// </summary>
   /// <param name="stringparam"></param>
   /// <param name="numchars"></param>
   /// <returns>Right-most numchars characters</returns>
   public static string Right( this string stringparam, int numchars )
   {
      // Handle possible Null or numeric stringparam being passed
      stringparam += string.Empty;

      // Handle possible negative numchars being passed
      numchars = Math.Abs( numchars );

      // Validate numchars parameter
      if (numchars > stringparam.Length)
         numchars = stringparam.Length;

      return stringparam.Substring( stringparam.Length - numchars );
   }

   /// <summary>
   /// VB Mid function - to end of string
   /// </summary>
   /// <param name="stringparam"></param>
   /// <param name="startIndex">VB-Style startindex, 1st char startindex = 1</param>
   /// <returns>Balance of string beginning at startindex character</returns>
   public static string Mid( this string stringparam, int startindex )
   {
```

```
        // Handle possible Null or numeric stringparam being passed
        stringparam += string.Empty;

        // Handle possible negative startindex being passed
        startindex = Math.Abs( startindex );

        // Validate numchars parameter
        if (startindex > stringparam.Length)
           startindex = stringparam.Length;

        // C# strings are zero-based, convert passed startindex
        return stringparam.Substring( startindex - 1 );
     }

     /// <summary>
     /// VB Mid function - for number of characters
     /// </summary>
     /// <param name="stringparam"></param>
     /// <param name="startIndex">VB-Style startindex, 1st char startindex = 1</param>
     /// <param name="numchars">number of characters to return</param>
     /// <returns>Balance of string beginning at startindex character</returns>
     public static string Mid( this string stringparam, int startindex, int numchars)
     {
        // Handle possible Null or numeric stringparam being passed
        stringparam += string.Empty;

        // Handle possible negative startindex being passed
        startindex = Math.Abs( startindex );

        // Handle possible negative numchars being passed
        numchars = Math.Abs( numchars );

        // Validate numchars parameter
        if (startindex > stringparam.Length)
           startindex = stringparam.Length;

        // C# strings are zero-based, convert passed startindex
        return stringparam.Substring( startindex - 1, numchars );

      }
   }
```

Este método de extensión se puede utilizar de la siguiente manera:

```
string myLongString = "Hello World!";
string myShortString = myLongString.Right(6);  // "World!"
string myLeftString = myLongString.Left(5);    // "Hello"
string myMidString1 = myLongString.Left(4);    // "lo World"
string myMidString2 = myLongString.Left(2,3);    // "ell"
```

## Comprobación de cadenas vacías utilizando String.IsNullOrEmpty () y String.IsNullOrWhiteSpace ()

```
string nullString = null;
string emptyString = "";
string whitespaceString = "    ";
string tabString = "\t";
```

```
string newlineString = "\n";
string nonEmptyString = "abc123";

bool result;

result = String.IsNullOrEmpty(nullString);            // true
result = String.IsNullOrEmpty(emptyString);           // true
result = String.IsNullOrEmpty(whitespaceString);      // false
result = String.IsNullOrEmpty(tabString);             // false
result = String.IsNullOrEmpty(newlineString);         // false
result = String.IsNullOrEmpty(nonEmptyString);        // false

result = String.IsNullOrWhiteSpace(nullString);       // true
result = String.IsNullOrWhiteSpace(emptyString);      // true
result = String.IsNullOrWhiteSpace(tabString);        // true
result = String.IsNullOrWhiteSpace(newlineString);    // true
result = String.IsNullOrWhiteSpace(whitespaceString); // true
result = String.IsNullOrWhiteSpace(nonEmptyString);   // false
```

## Obtener un char en un índice específico y enumerar la cadena

Puede usar el método de `Substring` para obtener cualquier número de caracteres de una cadena en cualquier ubicación. Sin embargo, si solo desea un solo carácter, puede usar el indexador de cadena para obtener un solo carácter en cualquier índice dado como lo hace con una matriz:

```
string s = "hello";
char c = s[1]; //Returns 'e'
```

Observe que el tipo de retorno es `char` , a diferencia del método de `Substring` que devuelve un tipo de `string` .

También puede usar el indexador para iterar a través de los caracteres de la cadena:

```
string s = "hello";
foreach (char c in s)
    Console.WriteLine(c);
/********* This will print each character on a new line:
h
e
l
l
o
**********/
```

## Convertir número decimal a formato binario, octal y hexadecimal

1. Para convertir un número decimal a formato binario usa **base 2**

   ```
   Int32 Number = 15;
   Console.WriteLine(Convert.ToString(Number, 2));  //OUTPUT : 1111
   ```

2. Para convertir un número decimal a formato octal, use **base 8**

```
int Number = 15;
Console.WriteLine(Convert.ToString(Number, 8));  //OUTPUT : 17
```

3. Para convertir un número decimal a formato hexadecimal, use **base 16**

```
var Number = 15;
Console.WriteLine(Convert.ToString(Number, 16));  //OUTPUT : f
```

## Dividir una cadena por otra cadena

```
string str = "this--is--a--complete--sentence";
string[] tokens = str.Split(new[] { "--" }, StringSplitOptions.None);
```

Resultado:

["este", "es", "a", "completo", "oración"]

## Invertir correctamente una cadena

La mayoría de las veces cuando las personas tienen que revertir una cadena, lo hacen más o menos así:

```
char[] a = s.ToCharArray();
System.Array.Reverse(a);
string r = new string(a);
```

Sin embargo, lo que estas personas no se dan cuenta es que esto está realmente mal.
Y no me refiero a que falte el cheque NULL.

En realidad es incorrecto porque un Glyph / GraphemeCluster puede constar de varios puntos de código (también conocidos como caracteres).

Para ver por qué esto es así, primero debemos ser conscientes del hecho de lo que realmente significa el término "carácter".

Referencia:

> Carácter es un término sobrecargado que puede significar muchas cosas.
>
> Un punto de código es la unidad atómica de información. El texto es una secuencia de puntos de código. Cada punto de código es un número que tiene el significado de la norma Unicode.
>
> Un grafema es una secuencia de uno o más puntos de código que se muestran como una unidad gráfica única que un lector reconoce como un elemento único del sistema de escritura. Por ejemplo, tanto a como ä son grafemas, pero pueden constar de múltiples puntos de código (por ejemplo, ä pueden ser dos puntos de código, uno para el carácter base a seguido de uno para la diaresis; pero también hay un código alternativo, heredado, único). punto que representa este grafema). Algunos puntos de

> código nunca forman parte de ningún grafema (p. Ej., El no-ensamblador de ancho cero o las anulaciones direccionales).
>
> Un glifo es una imagen, generalmente almacenada en una fuente (que es una colección de glifos), utilizada para representar grafemas o partes de ellos. Las fuentes pueden componer múltiples glifos en una sola representación, por ejemplo, si el ä anterior es un solo punto de código, una fuente puede elegir convertir eso como dos glifos separados, superpuestos espacialmente. Para OTF, las tablas GSUB y GPOS de la fuente contienen información de sustitución y posicionamiento para que esto funcione. Una fuente también puede contener múltiples glifos alternativos para el mismo grafema.

Así que en C #, un personaje es en realidad un CodePoint.

Lo que significa que, si solo inviertes una cadena válida como `Les Misé rables` , que puede verse así

```
string s = "Les Mise\u0301rables";
```

como secuencia de caracteres, obtendrás:

> selbaŕesiM seL

Como puede ver, el acento está en el carácter R, en lugar del carácter e.
Aunque string.reverse.reverse producirá la cadena original si ambas veces invierte la matriz char, este tipo de inversión definitivamente NO es lo contrario de la cadena original.

Tendrá que revertir cada GraphemeCluster solamente.
Entonces, si se hace correctamente, invierte una cadena como esta:

```
    private static System.Collections.Generic.List<string> GraphemeClusters(string s)
    {
        System.Collections.Generic.List<string> ls = new
System.Collections.Generic.List<string>();

        System.Globalization.TextElementEnumerator enumerator =
System.Globalization.StringInfo.GetTextElementEnumerator(s);
        while (enumerator.MoveNext())
        {
            ls.Add((string)enumerator.Current);
        }

        return ls;
    }


    // this
    private static string ReverseGraphemeClusters(string s)
    {
        if(string.IsNullOrEmpty(s) || s.Length == 1)
             return s;

        System.Collections.Generic.List<string> ls = GraphemeClusters(s);
```

```
        ls.Reverse();

        return string.Join("", ls.ToArray());
    }

    public static void TestMe()
    {
        string s = "Les Mise\u0301rables";
        // s = "noël";
        string r = ReverseGraphemeClusters(s);

        // This would be wrong:
        // char[] a = s.ToCharArray();
        // System.Array.Reverse(a);
        // string r = new string(a);

        System.Console.WriteLine(r);
    }
```

Y, oh, alegría, te darás cuenta si lo haces correctamente de esta manera, también funcionará para los idiomas de Asia / Sudeste Asiático / Asia Oriental (y Francés / Sueco / Noruego, etc.) ...

## Reemplazar una cadena dentro de una cadena

Usando el método `System.String.Replace` , puedes reemplazar parte de una cadena con otra cadena.

```
string s = "Hello World";
 s = s.Replace("World", "Universe"); // s = "Hello Universe"
```

Todas las apariciones de la cadena de búsqueda se reemplazan.

Este método también se puede usar para eliminar parte de una cadena, utilizando el campo `String.Empty` :

```
string s = "Hello World";
s = s.Replace("ell", String.Empty); // s = "Ho World"
```

## Cambiando el caso de los personajes dentro de una cadena

La clase `System.String` admite varios métodos para convertir caracteres en mayúsculas y minúsculas en una cadena.

- `System.String.ToLowerInvariant` se usa para devolver un objeto String convertido a minúsculas.
- `System.String.ToUpperInvariant` se usa para devolver un objeto String convertido a mayúsculas.

**Nota:** la razón para usar las versiones *invariables* de estos métodos es evitar la producción de letras inesperadas específicas de la cultura. Esto se explica aquí en detalle .

Ejemplo:

```
string s = "My String";
s = s.ToLowerInvariant(); // "my string"
s = s.ToUpperInvariant(); // "MY STRING"
```

Tenga en cuenta que *puede* elegir especificar una **Cultura** específica al convertir a minúsculas y mayúsculas utilizando los métodos String.ToLower (CultureInfo) y String.ToUpper (CultureInfo) en consecuencia.

## Concatenar una matriz de cadenas en una sola cadena

El método `System.String.Join` permite concatenar todos los elementos en una matriz de cadenas, utilizando un separador especificado entre cada elemento:

```
string[] words = {"One", "Two", "Three", "Four"};
string singleString = String.Join(",", words); // singleString = "One,Two,Three,Four"
```

## Concatenacion de cuerdas

La concatenación de cadenas se puede hacer usando el método `System.String.Concat` , o (mucho más fácil) usando el operador `+` :

```
string first = "Hello ";
string second = "World";

string concat = first + second; // concat = "Hello World"
concat = String.Concat(first, second); // concat = "Hello World"
```

En C # 6 esto se puede hacer de la siguiente manera:

```
string concat = $"{first},{second}";
```

Lea Operaciones de cadena comunes en línea:
https://riptutorial.com/es/csharp/topic/73/operaciones-de-cadena-comunes

# Capítulo 124: Operador de Igualdad

## Examples

### Clases de igualdad en c # y operador de igualdad

En C #, hay dos tipos diferentes de igualdad: igualdad de referencia e igualdad de valores. La igualdad de valores es el significado comúnmente entendido de igualdad: significa que dos objetos contienen los mismos valores. Por ejemplo, dos enteros con el valor de 2 tienen igualdad de valores. Igualdad de referencia significa que no hay dos objetos para comparar. En su lugar, hay dos referencias de objeto, ambas de las cuales se refieren al mismo objeto.

```
object a = new object();
object b = a;
System.Object.ReferenceEquals(a, b);  //returns true
```

Para tipos de valores predefinidos, el operador de igualdad (==) devuelve verdadero si los valores de sus operandos son iguales, de lo contrario, falso. Para tipos de referencia distintos a la cadena, == devuelve verdadero si sus dos operandos se refieren al mismo objeto. Para el tipo de cadena, == compara los valores de las cadenas.

```
// Numeric equality: True
Console.WriteLine((2 + 2) == 4);

// Reference equality: different objects,
// same boxed value: False.
object s = 1;
object t = 1;
Console.WriteLine(s == t);

// Define some strings:
string a = "hello";
string b = String.Copy(a);
string c = "hello";

// Compare string values of a constant and an instance: True
Console.WriteLine(a == b);

// Compare string references;
// a is a constant but b is an instance: False.
Console.WriteLine((object)a == (object)b);

// Compare string references, both constants
// have the same value, so string interning
// points to same reference: True.
Console.WriteLine((object)a == (object)c);
```

Lea Operador de Igualdad en línea: https://riptutorial.com/es/csharp/topic/1491/operador-de-igualdad

# Capítulo 125: Operador de unión nula

## Sintaxis

- var result = possibleNullObject ?? valor por defecto;

## Parámetros

| Parámetro | Detalles |
|---|---|
| `possibleNullObject` | El valor para probar el valor nulo. Si no es nulo, se devuelve este valor. Debe ser un tipo anulable. |
| `defaultValue` | El valor devuelto si es `possibleNullObject` NullObject es nulo. Debe ser del mismo tipo que `possibleNullObject` . |

## Observaciones

El operador de unión nula es dos caracteres de signo de interrogación consecutivos: `??`

Es una abreviatura de la expresión condicional:

```
possibleNullObject != null ? possibleNullObject : defaultValue
```

El operando del lado izquierdo (objeto que se está probando) debe ser un tipo de valor anulable o un tipo de referencia, o se producirá un error de compilación.

Los ?? El operador trabaja tanto para tipos de referencia como para tipos de valor.

## Examples

### Uso básico

El uso del `null-coalescing operator (??)` permite especificar un valor predeterminado para un tipo anulable si el operando de la izquierda es `null` .

```
string testString = null;
Console.WriteLine("The specified string is - " + (testString ?? "not provided"));
```

Demo en vivo en .NET Fiddle

Esto es lógicamente equivalente a:

```
string testString = null;
```

```
if (testString == null)
{
    Console.WriteLine("The specified string is - not provided");
}
else
{
    Console.WriteLine("The specified string is - " + testString);
}
```

o utilizando el operador ternario (? :) operador:

```
string testString = null;
Console.WriteLine("The specified string is - " + (testString == null ? "not provided" :
testString));
```

## Nula caída y encadenamiento

El operando de la izquierda debe ser anulable, mientras que el operando de la derecha puede ser o no. El resultado se escribirá en consecuencia.

**No nulable**

```
int? a = null;
int b = 3;
var output = a ?? b;
var type = output.GetType();

Console.WriteLine($"Output Type :{type}");
Console.WriteLine($"Output value :{output}");
```

**Salida:**

Tipo: System.Int32
valor: 3

Ver demostración

**Anulable**

```
int? a = null;
int? b = null;
var output = a ?? b;
```

`output` será de tipo `int?` e igual a `b` , o `null` .

**Coalescente múltiple**

La coalescencia también se puede hacer en cadenas:

```
int? a = null;
int? b = null;
int c = 3;
```

```
var output = a ?? b ?? c;

var type = output.GetType();
Console.WriteLine($"Type :{type}");
Console.WriteLine($"value :{output}");
```

**Salida:**

Tipo: System.Int32
valor: 3

Ver demostración

**Encadenamiento condicional nulo**

El operador de unión nula se puede utilizar junto con el operador de propagación nula para proporcionar un acceso más seguro a las propiedades de los objetos.

```
object o = null;
var output = o?.ToString() ?? "Default Value";
```

**Salida:**

Tipo: System.String
valor: valor predeterminado

Ver demostración

## Operador coalescente nulo con llamadas a método

El operador de unión nula facilita la tarea de garantizar que un método que pueda devolver `null` recaiga en un valor predeterminado.

Sin el operador de unión nula:

```
string name = GetName();

if (name == null)
    name = "Unknown!";
```

Con el operador de unión nula:

```
string name = GetName() ?? "Unknown!";
```

## Usa existente o crea nuevo.

Un escenario de uso común con el que esta característica realmente ayuda es cuando busca un objeto en una colección y necesita crear uno nuevo si aún no existe.

```
IEnumerable<MyClass> myList = GetMyList();
```

```
var item = myList.SingleOrDefault(x => x.Id == 2) ?? new MyClass { Id = 2 };
```

### Inicialización de propiedades diferidas con operador coalescente nulo

```
private List<FooBar> _fooBars;

public List<FooBar> FooBars
{
    get { return _fooBars ?? (_fooBars = new List<FooBar>()); }
}
```

La primera vez que se accede a la propiedad `.FooBars` la variable `_fooBars` se evaluará como `null` , por lo que `_fooBars` en la instrucción de asignación asigna y evalúa el valor resultante.

## Seguridad del hilo

Esta **no** es **una** forma **segura** de implementar propiedades perezosas. Para la holgazanería segura, use la clase `Lazy<T>` integrada en .NET Framework.

## C # 6 Azúcar sintáctico utilizando cuerpos de expresión.

Tenga en cuenta que desde C # 6, esta sintaxis se puede simplificar utilizando el cuerpo de la expresión para la propiedad:

```
private List<FooBar> _fooBars;

public List<FooBar> FooBars => _fooBars ?? ( _fooBars = new List<FooBar>() );
```

Los accesos posteriores a la propiedad producirán el valor almacenado en la variable `_fooBars` .

## Ejemplo en el patrón MVVM

Esto se usa a menudo cuando se implementan comandos en el patrón MVVM. En lugar de inicializar los comandos con entusiasmo con la construcción de un modelo de vista, los comandos se inician perezosamente utilizando este patrón de la siguiente manera:

```
private ICommand _actionCommand = null;
public ICommand ActionCommand =>
   _actionCommand ?? ( _actionCommand = new DelegateCommand( DoAction ) );
```

Lea Operador de unión nula en línea: https://riptutorial.com/es/csharp/topic/37/operador-de-union-nula

# Capítulo 126: Operadores de condicionamiento nulo

## Sintaxis

- X? .Y; // nulo si X es nulo o XY
- X? .Y? .Z; // nulo si X es nulo o Y es nulo o XYZ
- X? [Índice]; // nulo si X es nulo otra cosa X [índice]
- X? .ValueMethod (); // nulo si X es nulo o el resultado de X.ValueMethod ();
- X? .VoidMethod (); // no hacer nada si X es nulo o llama a X.VoidMethod ();

## Observaciones

Tenga en cuenta que al utilizar el operador de unión nula en un tipo de valor `T` , obtendrá un `Nullable<T>` devuelto por `Nullable<T>` .

## Examples

### Operador condicional nulo

El `?.` El operador es el azúcar sintáctico para evitar verificaciones nulas verbosas. También es conocido como el operador de navegación segura .

---

Clase utilizada en el siguiente ejemplo:

```
public class Person
{
    public int Age { get; set; }
    public string Name { get; set; }
    public Person Spouse { get; set; }
}
```

Si un objeto es potencialmente nulo (como una función que devuelve un tipo de referencia), primero se debe verificar si el objeto es nulo para evitar una posible `NullReferenceException` . Sin el operador condicional nulo, esto se vería así:

```
Person person = GetPerson();

int? age = null;
if (person != null)
    age = person.Age;
```

El mismo ejemplo usando el operador condicional nulo:

```
Person person = GetPerson();

var age = person?.Age;    // 'age' will be of type 'int?', even if 'person' is not null
```

---

# Encadenamiento del operador

El operador condicional nulo se puede combinar en los miembros y sub-miembros de un objeto.

```
// Will be null if either `person` or `person.Spouse` are null
int? spouseAge = person?.Spouse?.Age;
```

---

# Combinando con el Operador Nulo-Coalescente

El operador de condición nula se puede combinar con el operador de unión nula para proporcionar un valor predeterminado:

```
// spouseDisplayName will be "N/A" if person, Spouse, or Name is null
var spouseDisplayName = person?.Spouse?.Name ?? "N/A";
```

## El índice nulo condicional

Al igual que el `?.` operador, el operador de índice condicional nulo verifica los valores nulos al indexar en una colección que puede ser nula.

```
string item = collection?[index];
```

es azúcar sintáctica para

```
string item = null;
if(collection != null)
{
    item = collection[index];
}
```

## Evitando NullReferenceExceptions

```
var person = new Person
{
    Address = null;
};

var city = person.Address.City; //throws a NullReferenceException
var nullableCity = person.Address?.City; //returns the value of null
```

Este efecto puede ser encadenado:

```
var person = new Person
{
    Address = new Address
    {
        State = new State
        {
            Country = null
        }
    }
};

// this will always return a value of at least "null" to be stored instead
// of throwing a NullReferenceException
var countryName = person?.Address?.State?.Country?.Name;
```

## El operador condicional nulo se puede utilizar con el método de extensión

El método de extensión puede funcionar en referencias nulas , pero puede usar `?.` para nulo-comprobar de todos modos.

```
public class Person
{
    public string Name {get; set;}
}

public static class PersonExtensions
{
    public static int GetNameLength(this Person person)
    {
        return person == null ? -1 : person.Name.Length;
    }
}
```

Normalmente, el método se activará para referencias `null` y devolverá -1:

```
Person person = null;
int nameLength = person.GetNameLength(); // returns -1
```

Utilizando `?.` el método no se activará para referencias `null` , y el tipo es `int?` :

```
Person person = null;
int? nameLength = person?.GetNameLength(); // nameLength is null.
```

Este comportamiento se espera realmente de la forma en que el `?.` el operador funciona: evitará realizar llamadas de método de instancia para instancias nulas, para evitar `NullReferenceExceptions` . Sin embargo, la misma lógica se aplica al método de extensión, a pesar de la diferencia en cómo se declara el método.

Para obtener más información sobre por qué se llama al método de extensión en el primer ejemplo, consulte los Métodos de extensión: documentación de comprobación nula .

Lea Operadores de condicionamiento nulo en línea:
https://riptutorial.com/es/csharp/topic/41/operadores-de-condicionamiento-nulo

# Capítulo 127: Palabra clave de rendimiento

## Introducción

Cuando utiliza la palabra clave de rendimiento en una declaración, indica que el método, el operador o el elemento de acceso en el que aparece es un iterador. El uso del rendimiento para definir un iterador elimina la necesidad de una clase adicional explícita (la clase que mantiene el estado para una enumeración) cuando implementa el patrón IEnumerable e IEnumerator para un tipo de colección personalizado.

## Sintaxis

- rendimiento [TIPO]
- pausa de rendimiento

## Observaciones

Poner la palabra clave de `yield` en un método con el tipo de retorno de `IEnumerable` , `IEnumerable<T>` , `IEnumerator` o `IEnumerator<T>` le dice al compilador que genere una implementación del tipo de retorno ( `IEnumerable` o `IEnumerator` ) que, cuando se realiza un bucle, ejecuta el Método hasta cada "rendimiento" para obtener cada resultado.

La palabra clave de `yield` es útil cuando desea devolver "el siguiente" elemento de una secuencia teóricamente ilimitada, por lo que sería imposible calcular la secuencia completa de antemano, o cuando calcular la secuencia completa de valores antes de regresar daría lugar a una pausa indeseable para el usuario .

`yield break` también se puede utilizar para terminar la secuencia en cualquier momento.

Como la palabra clave de `yield` requiere un tipo de interfaz de iterador como tipo de retorno, como `IEnumerable<T>` , no puede usar esto en un método asíncrono ya que devuelve un objeto `Task<IEnumerable<T>>` .

**Otras lecturas**

- https://msdn.microsoft.com/en-us/library/9k7k7cf0.aspx

## Examples

### Uso simple

La palabra clave de `yield` se utiliza para definir una función que devuelve un `IEnumerable` o `IEnumerator` (así como sus variantes genéricas derivadas) cuyos valores se generan perezosamente cuando un llamante recorre la colección devuelta. Lea más sobre el propósito en la sección de comentarios .

El siguiente ejemplo tiene una declaración de rendimiento que está dentro de un bucle `for` .

```
public static IEnumerable<int> Count(int start, int count)
{
    for (int i = 0; i <= count; i++)
    {
        yield return start + i;
    }
}
```

Entonces puedes llamarlo:

```
foreach (int value in Count(start: 4, count: 10))
{
    Console.WriteLine(value);
}
```

**Salida de consola**

```
4
5
6
...
14
```

Demo en vivo en .NET Fiddle

Cada iteración del cuerpo de la instrucción `foreach` crea una llamada a la función de iterador de `Count` . Cada llamada a la función de iterador pasa a la siguiente ejecución de la declaración de `yield return` , que se produce durante la siguiente iteración del bucle `for` .

## Uso más pertinente

```
public IEnumerable<User> SelectUsers()
{
    // Execute an SQL query on a database.
    using (IDataReader reader = this.Database.ExecuteReader(CommandType.Text, "SELECT Id, Name
FROM Users"))
    {
        while (reader.Read())
        {
            int id = reader.GetInt32(0);
            string name = reader.GetString(1);
            yield return new User(id, name);
        }
    }
}
```

Por supuesto, hay otras formas de obtener un `IEnumerable<User>` de una base de datos SQL. Esto solo demuestra que puede usar el `yield` para convertir cualquier cosa que tenga la semántica de "secuencia de elementos" en un `IEnumerable<T>` que alguien puede iterar. .

## Terminación anticipada

Puede ampliar la funcionalidad de los métodos de `yield` existentes pasando uno o más valores o elementos que podrían definir una condición de terminación dentro de la función llamando a un `yield break` para detener la ejecución del ciclo interno.

```
public static IEnumerable<int> CountUntilAny(int start, HashSet<int> earlyTerminationSet)
{
    int curr = start;

    while (true)
    {
        if (earlyTerminationSet.Contains(curr))
        {
            // we've hit one of the ending values
            yield break;
        }

        yield return curr;

        if (curr == Int32.MaxValue)
        {
            // don't overflow if we get all the way to the end; just stop
            yield break;
        }

        curr++;
    }
}
```

El método anterior se repetirá desde una posición de `start` dada hasta que se encuentre uno de los valores dentro del conjunto de `earlyTerminationSet` .

```
// Iterate from a starting point until you encounter any elements defined as
// terminating elements
var terminatingElements = new HashSet<int>{ 7, 9, 11 };
// This will iterate from 1 until one of the terminating elements is encountered (7)
foreach(var x in CountUntilAny(1,terminatingElements))
{
    // This will write out the results from 1 until 7 (which will trigger terminating)
    Console.WriteLine(x);
}
```

**Salida:**

1
2
3
4
5
6

Demo en vivo en .NET Fiddle

## Comprobando correctamente los argumentos

Un método iterador no se ejecuta hasta que se enumera el valor de retorno. Por lo tanto, es ventajoso afirmar condiciones previas fuera del iterador.

```
public static IEnumerable<int> Count(int start, int count)
{
    // The exception will throw when the method is called, not when the result is iterated
    if (count < 0)
        throw new ArgumentOutOfRangeException(nameof(count));

    return CountCore(start, count);
}

private static IEnumerable<int> CountCore(int start, int count)
{
    // If the exception was thrown here it would be raised during the first MoveNext()
    // call on the IEnumerator, potentially at a point in the code far away from where
    // an incorrect value was passed.
    for (int i = 0; i < count; i++)
    {
        yield return start + i;
    }
}
```

**Código lateral de llamada (uso):**

```
// Get the count
var count = Count(1,10);
// Iterate the results
foreach(var x in count)
{
    Console.WriteLine(x);
}
```

**Salida:**

1
2
3
4
5
6
7
8
9
10

Demo en vivo en .NET Fiddle

Cuando un método utiliza el `yield` para generar un enumerable, el compilador crea una máquina de estado que, cuando se itera, ejecuta el código hasta un `yield` . A continuación, devuelve el elemento cedido y guarda su estado.

Esto significa que no encontrará información sobre argumentos no válidos (pasar `null` etc.)

cuando llame por primera vez al método (porque eso crea la máquina de estado), solo cuando intente acceder al primer elemento (porque solo entonces el código dentro de la método se ejecuta por la máquina de estado). Al ajustarlo en un método normal que primero verifica los argumentos, puede verificarlos cuando se llama al método. Este es un ejemplo de fallar rápido.

Cuando se usa C # 7+, la función `CountCore` se puede ocultar convenientemente en la función `Count` como una *función local* . Vea el ejemplo aquí .

### Devuelve otro Enumerable dentro de un método que devuelve Enumerable.

```
public IEnumerable<int> F1()
{
    for (int i = 0; i < 3; i++)
        yield return i;

    //return F2(); // Compile Error!!
    foreach (var element in F2())
        yield return element;
}

public int[] F2()
{
    return new[] { 3, 4, 5 };
}
```

### Evaluación perezosa

Solo cuando la instrucción `foreach` mueve al siguiente elemento, el bloque iterador evalúa hasta la siguiente declaración de `yield` .

Considere el siguiente ejemplo:

```
private IEnumerable<int> Integers()
{
    var i = 0;
    while(true)
    {
        Console.WriteLine("Inside iterator: " + i);
        yield return i;
        i++;
    }
}

private void PrintNumbers()
{
    var numbers = Integers().Take(3);
    Console.WriteLine("Starting iteration");

    foreach(var number in numbers)
    {
        Console.WriteLine("Inside foreach: " + number);
    }
}
```

Esto dará como resultado:

Iteración inicial
Iterador interior: 0
Dentro de foreach: 0
Iterador interior: 1
Dentro de foreach: 1
Iterador interior: 2
Dentro de foreach: 2

Ver demostración

Como consecuencia:

- La "iteración de inicio" se imprime primero aunque se llamó al método del iterador antes de que la línea lo imprimiera porque la línea `Integers().Take(3);` en realidad no inicia la iteración (no se realizó ninguna llamada a `IEnumerator.MoveNext()` )
- Las líneas que se imprimen en la consola alternan entre la que está dentro del método del iterador y la que está dentro del `foreach` , en lugar de todas las que están dentro del método del iterador evaluando primero
- Este programa termina debido al método `.Take()` , aunque el método del iterador tiene un `while true` que nunca se rompe.

## Intenta ... finalmente

Si un método de iterador tiene un rendimiento dentro de un `try...finally` , el `IEnumerator` devuelto ejecutará la instrucción `finally` cuando se llame a `Dispose` , siempre que el punto actual de evaluación esté dentro del bloque `try` .

Dada la función:

```
private IEnumerable<int> Numbers()
{
    yield return 1;
    try
    {
        yield return 2;
        yield return 3;
    }
    finally
    {
        Console.WriteLine("Finally executed");
    }
}
```

Al llamar:

```
private void DisposeOutsideTry()
{
    var enumerator = Numbers().GetEnumerator();

    enumerator.MoveNext();
    Console.WriteLine(enumerator.Current);
    enumerator.Dispose();
```

```
}
```

Luego se imprime:

1

Ver demostración

Al llamar:

```
private void DisposeInsideTry()
{
    var enumerator = Numbers().GetEnumerator();

    enumerator.MoveNext();
    Console.WriteLine(enumerator.Current);
    enumerator.MoveNext();
    Console.WriteLine(enumerator.Current);
    enumerator.Dispose();
}
```

Luego se imprime:

1
2
Finalmente ejecutado

Ver demostración

## Usando el rendimiento para crear un IEnumerator al implementar IEnumerable

La `IEnumerable<T>` tiene un método único, `GetEnumerator()` , que devuelve un `IEnumerator<T>` .

Si bien la palabra clave de `yield` se puede usar para crear directamente un `IEnumerable<T>` , *también se* puede usar exactamente de la misma manera para crear un `IEnumerator<T>` . Lo único que cambia es el tipo de retorno del método.

Esto puede ser útil si queremos crear nuestra propia clase que implementa `IEnumerable<T>` :

```
public class PrintingEnumerable<T> : IEnumerable<T>
{
    private IEnumerable<T> _wrapped;

    public PrintingEnumerable(IEnumerable<T> wrapped)
    {
        _wrapped = wrapped;
    }

    // This method returns an IEnumerator<T>, rather than an IEnumerable<T>
    // But the yield syntax and usage is identical.
    public IEnumerator<T> GetEnumerator()
    {
        foreach(var item in _wrapped)
        {
```

```
            Console.WriteLine("Yielding: " + item);
            yield return item;
        }
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }
}
```

(Tenga en cuenta que este ejemplo en particular es solo ilustrativo y podría implementarse de forma más limpia con un solo método iterador que devuelva un `IEnumerable<T>` ).

## Evaluación impaciente

La palabra clave de `yield` permite la evaluación perezosa de la colección. La carga forzada de toda la colección en la memoria se llama **evaluación impaciente** .

El siguiente código muestra esto:

```
IEnumerable<int> myMethod()
{
    for(int i=0; i <= 8675309; i++)
    {
        yield return i;
    }
}
...
// define the iterator
var it = myMethod.Take(3);
// force its immediate evaluation
// list will contain 0, 1, 2
var list = it.ToList();
```

Llamar a `ToList` , `ToDictionary` o `ToArray` forzará la evaluación inmediata de la enumeración, recuperando todos los elementos en una colección.

## Ejemplo de evaluación perezosa: números de Fibonacci

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Numerics; // also add reference to System.Numberics

namespace ConsoleApplication33
{
    class Program
    {
        private static IEnumerable<BigInteger> Fibonacci()
        {
            BigInteger prev = 0;
            BigInteger current = 1;
            while (true)
            {
```

```
                yield return current;
                var next = prev + current;
                prev = current;
                current = next;
            }
        }

        static void Main()
        {
            // print Fibonacci numbers from 10001 to 10010
            var numbers = Fibonacci().Skip(10000).Take(10).ToArray();
            Console.WriteLine(string.Join(Environment.NewLine, numbers));
        }
    }
}
```

Cómo funciona bajo el capó (recomiendo descompilar el archivo .exe resultante en la herramienta IL Disaambler):

1. El compilador C # genera una clase que implementa `IEnumerable<BigInteger>` y `IEnumerator<BigInteger>` ( `<Fibonacci>d__0` en ildasm).
2. Esta clase implementa una máquina de estado. El estado consiste en la posición actual en el método y los valores de las variables locales.
3. Los códigos más interesantes están en el `bool IEnumerator.MoveNext()` . Básicamente, lo que `MoveNext()` hace:
    - Restaura el estado actual. Variables como `prev` y `current` convierten en campos en nuestra clase ( `<current>5__2` y `<prev>5__1` en ildasm). En nuestro método tenemos dos posiciones ( `<>1__state` ): primero en la llave de apertura, segundo en `yield return` .
    - Ejecuta el código hasta la próxima `yield return` o la `yield break` / `}` .
    - Para el `yield return` valor resultante se guarda, por lo que la propiedad `Current` puede devolverlo. se devuelve el `true` En este punto, el estado actual se guarda de nuevo para la siguiente invocación de `MoveNext` .
    - Para el método de `yield break` / `}` , solo se devuelve el significado `false` se realizó la iteración.

También tenga en cuenta que ese número 10001 tiene 468 bytes de longitud. La máquina de estado solo guarda `current` variables `current` y `prev` como campos. Si bien, si nos gustaría guardar todos los números en la secuencia desde el primero hasta el 10000, el tamaño de memoria consumido será de más de 4 megabytes. Por lo tanto, la evaluación perezosa, si se usa correctamente, puede reducir la huella de memoria en algunos casos.

## La diferencia entre rotura y rotura de rendimiento.

Usar la `yield break` en lugar de la `break` puede no ser tan obvio como se podría pensar. Hay muchos ejemplos malos en Internet donde el uso de los dos es intercambiable y no demuestra la diferencia.

La parte confusa es que ambas palabras clave (o frases clave) solo tienen sentido dentro de los bucles ( `foreach` , `while` ...) Entonces, ¿cuándo elegir una sobre la otra?

Es importante darse cuenta de que una vez que use la palabra clave de `yield` en un método,

convertirá el método en un iterador . El único propósito de tal método es, entonces, iterar sobre una colección finita o infinita y producir (salida) sus elementos. Una vez que se cumple el propósito, no hay razón para continuar con la ejecución del método. A veces, sucede naturalmente con el último corchete de cierre del método `}` . Pero a veces, quieres terminar el método prematuramente. En un método normal (no iterativo), usaría la palabra clave `return` . Pero no se puede usar el `return` en un iterador, se debe usar la `yield break` . En otras palabras, el `yield break` para un iterador es el mismo que el `return` para un método estándar. Mientras que, la instrucción `break` solo termina el bucle más cercano.

Veamos algunos ejemplos:

```
/// <summary>
/// Yields numbers from 0 to 9
/// </summary>
/// <returns>{0,1,2,3,4,5,6,7,8,9}</returns>
public static IEnumerable<int> YieldBreak()
{
    for (int i = 0; ; i++)
    {
        if (i < 10)
        {
            // Yields a number
            yield return i;
        }
        else
        {
            // Indicates that the iteration has ended, everything
            // from this line on will be ignored
            yield break;
        }
    }
    yield return 10; // This will never get executed
}
```

```
/// <summary>
/// Yields numbers from 0 to 10
/// </summary>
/// <returns>{0,1,2,3,4,5,6,7,8,9,10}</returns>
public static IEnumerable<int> Break()
{
    for (int i = 0; ; i++)
    {
        if (i < 10)
        {
            // Yields a number
            yield return i;
        }
        else
        {
            // Terminates just the loop
            break;
        }
    }
    // Execution continues
    yield return 10;
}
```

Lea Palabra clave de rendimiento en línea: https://riptutorial.com/es/csharp/topic/61/palabra-clave-de-rendimiento

# Capítulo 128: Palabras clave

## Introducción

Las palabras clave son identificadores predefinidos y reservados con un significado especial para el compilador. No se pueden usar como identificadores en su programa sin el prefijo `@` . Por ejemplo, `@if` es un identificador legal pero no la palabra clave `if` .

## Observaciones

C # tiene una colección predefinida de "palabras clave" (o palabras reservadas), cada una de las cuales tiene una función especial. Estas palabras no se pueden usar como identificadores (nombres para variables, métodos, clases, etc.) a menos que se prefijen con `@` .

- `abstract`
- `as`
- `base`
- `bool`
- `break`
- `byte`
- `case`
- `catch`
- `char`
- `checked`
- `class`
- `const`
- `continue`
- `decimal`
- `default`
- `delegate`
- `do`
- `double`
- `else`
- `enum`
- `event`
- `explicit`
- `extern`
- `false`
- `finally`
- `fixed`
- `float`
- `for`
- `foreach`
- `goto`
- `if`
- `implicit`
- `in`
- `int`
- `interface`
- `internal`
- `is`

- lock
- long
- namespace
- new
- null
- object
- operator
- out
- override
- params
- private
- protected
- public
- readonly
- ref
- return
- sbyte
- sealed
- short
- sizeof
- stackalloc
- static
- string
- struct
- switch
- this
- throw
- true
- try
- typeof
- uint
- ulong
- unchecked
- unsafe
- ushort
- using (directiva)
- using (declaración)
- virtual
- void
- volatile
- when
- while

Aparte de estos, C # también usa algunas palabras clave para proporcionar un significado específico en el código. Se llaman palabras clave contextuales. Las palabras clave contextuales se pueden usar como identificadores y no es necesario que tengan un prefijo con `@` cuando se usan como identificadores.

- add
- alias
- ascending
- async
- await
- descending
- dynamic

- from
- get
- global
- group
- into
- join
- let
- nameof
- orderby
- partial
- remove
- select
- set
- value
- var
- where
- yield

# Examples

## stackalloc

La palabra clave `stackalloc` crea una región de memoria en la pila y devuelve un puntero al inicio de esa memoria. La memoria asignada a la pila se elimina automáticamente cuando se sale del ámbito en el que se creó.

```
//Allocate 1024 bytes. This returns a pointer to the first byte.
byte* ptr = stackalloc byte[1024];

//Assign some values...
ptr[0] = 109;
ptr[1] = 13;
ptr[2] = 232;
...
```

*Utilizado en un contexto inseguro.*

Al igual que con todos los punteros en C #, no hay límites de verificación en las lecturas y asignaciones. La lectura más allá de los límites de la memoria asignada tendrá resultados impredecibles: puede acceder a una ubicación arbitraria dentro de la memoria o puede causar una excepción de infracción de acceso.

```
//Allocate 1 byte
byte* ptr = stackalloc byte[1];

//Unpredictable results...
ptr[10] = 1;
ptr[-1] = 2;
```

La memoria asignada a la pila se elimina automáticamente cuando se sale del ámbito en el que se creó. Esto significa que nunca debe devolver la memoria creada con stackalloc o almacenarla más allá de la vida útil del alcance.

```
unsafe IntPtr Leak() {
    //Allocate some memory on the stack
    var ptr = stackalloc byte[1024];

    //Return a pointer to that memory (this exits the scope of "Leak")
    return new IntPtr(ptr);
}

unsafe void Bad() {
     //ptr is now an invalid pointer, using it in any way will have
     //unpredictable results. This is exactly the same as accessing beyond
     //the bounds of the pointer.
     var ptr = Leak();
}
```

`stackalloc` solo se puede utilizar al declarar *e* inicializar variables. Lo siguiente *no* es válido:

```
byte* ptr;
...
ptr = stackalloc byte[1024];
```

---

**Observaciones:**

`stackalloc` solo debe usarse para optimizaciones de rendimiento (ya sea para computación o interoperabilidad). Esto se debe al hecho de que:

- El recolector de basura no es necesario ya que la memoria se asigna en la pila en lugar del montón, la memoria se libera tan pronto como la variable queda fuera del alcance
- Es más rápido asignar memoria en la pila que en el montón
- Aumente la posibilidad de que la memoria caché llegue a la CPU debido a la ubicación de los datos.

## volátil

Agregar la palabra clave `volatile` a un campo indica al compilador que el valor del campo puede ser cambiado por varios subprocesos separados. El propósito principal de la palabra clave `volatile` es evitar las optimizaciones del compilador que asumen solo el acceso de un solo hilo. El uso de `volatile` garantiza que el valor del campo sea el valor más reciente disponible, y que el valor no esté sujeto al almacenamiento en caché que tienen los valores no volátiles.

Es una buena práctica marcar *cada variable* que puede ser utilizada por múltiples subprocesos como `volatile` para evitar comportamientos inesperados debido a optimizaciones detrás de escena. Considere el siguiente bloque de código:

```
public class Example
{
    public int x;

    public void DoStuff()
    {
        x = 5;
```

```
        // the compiler will optimize this to y = 15
        var y = x + 10;

        /* the value of x will always be the current value, but y will always be "15" */
        Debug.WriteLine("x = " + x + ", y = " + y);
    }
}
```

En el bloque de código anterior, el compilador lee las declaraciones `x = 5` y `y = x + 10` y determina que el valor de `y` siempre terminará como 15. Por lo tanto, optimizará la última instrucción como `y = 15` . Sin embargo, la variable `x` es de hecho un campo `public` y el valor de `x` se puede modificar en tiempo de ejecución a través de un hilo diferente que actúa en este campo por separado. Ahora considere este código de bloque modificado. Tenga en cuenta que el campo `x` ahora se declara como `volatile` .

```
public class Example
{
    public volatile int x;

    public void DoStuff()
    {
        x = 5;

        // the compiler no longer optimizes this statement
        var y = x + 10;

        /* the value of x and y will always be the correct values */
        Debug.WriteLine("x = " + x + ", y = " + y);
    }
}
```

Ahora, el compilador busca los usos de *lectura* del campo `x` y asegura que el valor actual del campo siempre se recupera. Esto asegura que incluso si varios subprocesos están leyendo y escribiendo en este campo, el valor actual de `x` siempre se recupera.

`volatile` solo puede usarse en campos dentro de las `class` o `struct` . Lo siguiente *no* es válido :

```
public void MyMethod()
{
    volatile int x;
}
```

`volatile` solo puede aplicarse a campos de los siguientes tipos:

- tipos de referencia o parámetros de tipo genérico conocidos como tipos de referencia
- tipos primitivos como `sbyte` , `byte` , `short` , `ushort` , `int` , `uint` , `char` , `float` y `bool`
- enums tipos basados en `byte` , `sbyte` , `short` , `ushort` , `int` o `uint`
- `IntPtr` y `UIntPtr`

---

**Observaciones:**

- El modificador `volatile` se usa generalmente para un campo al que se accede mediante

varios subprocesos sin usar la instrucción de bloqueo para serializar el acceso.
- La palabra clave `volatile` se puede aplicar a campos de tipos de referencia
- La palabra clave `volatile` no funcionará con primitivos de 64 bits en una plataforma atómica de 32 bits. Las operaciones interbloqueadas, como `Interlocked.Read` y `Interlocked.Exchange` , todavía deben usarse para el acceso seguro de múltiples subprocesos en estas plataformas.

## fijo

La declaración fija corrige la memoria en una ubicación. Los objetos en la memoria generalmente se mueven alrededor, esto hace posible la recolección de basura. Pero cuando usamos punteros no seguros para las direcciones de memoria, esa memoria no debe ser movida.

- Usamos la declaración fija para garantizar que el recolector de basura no reubique los datos de cadena.

## Variables fijas

```
var myStr = "Hello world!";

fixed (char* ptr = myStr)
{
    // myStr is now fixed (won't be [re]moved by the Garbage Collector).
    // We can now do something with ptr.
}
```

*Utilizado en un contexto inseguro.*

## Tamaño del arreglo fijo

```
unsafe struct Example
{
    public fixed byte SomeField[8];
    public fixed char AnotherField[64];
}
```

`fixed` solo se puede usar en campos de una `struct` (también se debe usar en un contexto inseguro).

## defecto

Para las clases, interfaces, delegado, matriz, nullable (como int?) Y tipos de puntero, el `default(TheType)` devuelve `null` :

```
class MyClass {}
Debug.Assert(default(MyClass) == null);
Debug.Assert(default(string) == null);
```

Para estructuras y enumeraciones, el `default(TheType)` devuelve lo mismo que el `new TheType()` :

```
struct Coordinates
{
    public int X { get; set; }
    public int Y { get; set; }
}

struct MyStruct
{
    public string Name { get; set; }
    public Coordinates Location { get; set; }
    public Coordinates? SecondLocation { get; set; }
    public TimeSpan Duration { get; set; }
}

var defaultStruct = default(MyStruct);
Debug.Assert(defaultStruct.Equals(new MyStruct()));
Debug.Assert(defaultStruct.Location.Equals(new Coordinates()));
Debug.Assert(defaultStruct.Location.X == 0);
Debug.Assert(defaultStruct.Location.Y == 0);
Debug.Assert(defaultStruct.SecondLocation == null);
Debug.Assert(defaultStruct.Name == null);
Debug.Assert(defaultStruct.Duration == TimeSpan.Zero);
```

`default(T)` puede ser particularmente útil cuando `T` es un parámetro genérico para el que no hay restricciones para decidir si `T` es un tipo de referencia o un tipo de valor, por ejemplo:

```
public T GetResourceOrDefault<T>(string resourceName)
{
   if (ResourceExists(resourceName))
   {
      return (T)GetResource(resourceName);
   }
   else
   {
      return default(T);
   }
}
```

## solo lectura

La palabra clave `readonly` es un modificador de campo. Cuando una declaración de campo incluye un modificador de `readonly` , las asignaciones a ese campo solo pueden ocurrir como parte de la declaración o en un constructor en la misma clase.

La palabra clave `readonly` es diferente de la palabra clave `const` . Un campo `const` solo se puede inicializar en la declaración del campo. Un campo de `readonly` puede inicializarse en la declaración o en un constructor. Por lo tanto, los campos de `readonly` pueden tener diferentes valores dependiendo del constructor utilizado.

La palabra clave `readonly` se usa a menudo cuando se inyectan dependencias.

```
class Person
{
    readonly string _name;
    readonly string _surname = "Surname";
```

```
    Person(string name)
    {
        _name = name;
    }
    void ChangeName()
    {
        _name = "another name"; // Compile error
        _surname = "another surname"; // Compile error
    }
}
```

> Nota: la declaración de un campo de *solo lectura* no implica *inmutabilidad* . Si el campo es un *tipo de referencia,* entonces se puede cambiar el **contenido** del objeto. *Readonly* se usa normalmente para evitar que el objeto se **sobrescriba** y se asigne solo durante la **creación** de **instancias** de ese objeto.
>
> Nota: Dentro del constructor se puede reasignar un campo de solo lectura.

```
public class Car
{
    public double Speed {get; set;}
}

//In code

private readonly Car car = new Car();

private void SomeMethod()
{
    car.Speed = 100;
}
```

## como

La palabra clave `as` es un operador similar a un *reparto* . Si una conversión no es posible, usar `as` produce `null` lugar de dar como resultado una excepción `InvalidCastException` .

`expression as type` es equivalente a `expression is type ? (type)expression : (type)null` con la advertencia de que, `as` solo es válido en las conversiones de referencia, las conversiones que admiten `expression is type ? (type)expression : (type)null` y las conversiones de boxeo. Las conversiones definidas por el usuario *no* son compatibles; en su lugar se debe usar un elenco regular.

Para la expansión anterior, el compilador genera código de tal manera que la `expression` solo se evaluará una vez y utilizará la comprobación de tipo dinámico único (a diferencia de los dos en el ejemplo anterior).

`as` puede ser útil cuando se espera que un argumento facilite varios tipos. Específicamente, se concede al usuario múltiples opciones - en lugar de comprobar todas las posibilidades con `is` antes de la colada, o simplemente la fundición y la captura de excepciones. Es una buena práctica usar 'como' al lanzar / verificar un objeto, lo que causará solo una penalización de

desempaquetado. El uso `is` para verificar, luego el lanzamiento causará dos penalizaciones de desempaquetado.

Si se espera que un argumento sea una instancia de un tipo específico, se prefiere una conversión regular, ya que su propósito es más claro para el lector.

Debido a que una llamada a `as` puede producir `null` , siempre verifique el resultado para evitar una `NullReferenceException` .

### Ejemplo de uso

```
object something = "Hello";
Console.WriteLine(something as string);        //Hello
Console.Writeline(something as Nullable<int>); //null
Console.WriteLine(something as int?);          //null

//This does NOT compile:
//destination type must be a reference type (or a nullable value type)
Console.WriteLine(something as int);
```

Demo en vivo en .NET Fiddle

Ejemplo equivalente sin usar `as` :

```
Console.WriteLine(something is string ? (string)something : (string)null);
```

Esto es útil cuando se reemplaza la función de `Equals` en clases personalizadas.

```
class MyCustomClass
{

    public override bool Equals(object obj)
    {
        MyCustomClass customObject = obj as MyCustomClass;

        // if it is null it may be really null
        // or it may be of a different type
        if (Object.ReferenceEquals(null, customObject))
        {
            // If it is null then it is not equal to this instance.
            return false;
        }

        // Other equality controls specific to class
    }

}
```

## es

Comprueba si un objeto es compatible con un tipo dado, es decir, si un objeto es una instancia del tipo `BaseInterface` , o un tipo que se deriva de `BaseInterface` :

```
interface BaseInterface {}
class BaseClass : BaseInterface {}
class DerivedClass : BaseClass {}

var d = new DerivedClass();
Console.WriteLine(d is DerivedClass);  // True
Console.WriteLine(d is BaseClass);     // True
Console.WriteLine(d is BaseInterface); // True
Console.WriteLine(d is object);        // True
Console.WriteLine(d is string);        // False

var b = new BaseClass();
Console.WriteLine(b is DerivedClass);  // False
Console.WriteLine(b is BaseClass);     // True
Console.WriteLine(b is BaseInterface); // True
Console.WriteLine(b is object);        // True
Console.WriteLine(b is string);        // False
```

Si la intención de la conversión es usar el objeto, es una buena práctica usar la palabra clave `as` '

```
interface BaseInterface {}
class BaseClass : BaseInterface {}
class DerivedClass : BaseClass {}

var d = new DerivedClass();
Console.WriteLine(d is DerivedClass);  // True - valid use of 'is'
Console.WriteLine(d is BaseClass);     // True - valid use of 'is'

if(d is BaseClass){
    var castedD = (BaseClass)d;
    castedD.Method(); // valid, but not best practice
}

var asD = d as BaseClass;

if(asD!=null){
    asD.Method(); //prefered method since you incur only one unboxing penalty
}
```

Sin embargo, desde la característica de `pattern matching` C # 7 se extiende el operador is para verificar un tipo y declarar una nueva variable al mismo tiempo. Misma parte de código con C # 7:

7.0

```
if(d is BaseClass asD ){
    asD.Method();
}
```

## tipo de

Devuelve el `Type` de un objeto, sin la necesidad de instanciarlo.

```
Type type = typeof(string);
Console.WriteLine(type.FullName); //System.String
Console.WriteLine("Hello".GetType() == type); //True
Console.WriteLine("Hello".GetType() == typeof(string)); //True
```

## const

`const` se utiliza para representar valores que **nunca cambiarán a lo** largo de la vida útil del programa. Su valor es constante desde **el tiempo** de **compilación** , a diferencia de la palabra clave `readonly` , cuyo valor es constante desde el tiempo de ejecución.

Por ejemplo, dado que la velocidad de la luz nunca cambiará, podemos almacenarla en una constante.

```
const double c = 299792458;  // Speed of light

double CalculateEnergy(double mass)
{
    return mass * c * c;
}
```

Esto es esencialmente lo mismo que tener una `return mass * 299792458 * 299792458` , ya que el compilador sustituirá directamente `c` con su valor constante.

Como resultado, `c` no se puede cambiar una vez declarado. Lo siguiente producirá un error en tiempo de compilación:

```
const double c = 299792458;  // Speed of light

c = 500;  //compile-time error
```

Una constante se puede prefijar con los mismos modificadores de acceso que los métodos:

```
private const double c = 299792458;
public const double c = 299792458;
internal const double c = 299792458;
```

`const` miembros `const` son `static` por naturaleza. Sin embargo, el uso de `static` explícitamente no está permitido.

También puedes definir constantes locales de método:

```
double CalculateEnergy(double mass)
{
    const c = 299792458;
    return mass * c * c;
}
```

Estos no pueden ser prefijados con una palabra clave `private` o `public` , ya que son implícitamente locales al método en el que están definidos.

---

No todos los tipos se pueden utilizar en una declaración `const` . Los tipos de valores permitidos son los tipos predefinidos `sbyte` , `byte` , `short` , `ushort` , `int` , `uint` , `long` , `ulong` , `char` , `float` , `double` , `decimal` , `bool` y todos los tipos de `enum` . Intentar declarar miembros `const` con otros tipos de valor

(como `TimeSpan` o `Guid` ) fallará en tiempo de compilación.

Para la `string` tipo de referencia predefinida especial, las constantes se pueden declarar con cualquier valor. Para todos los demás tipos de referencia, las constantes se pueden declarar, pero siempre deben tener el valor `null` .

---

Debido a que los valores `const` son conocidos en tiempo de compilación, se permiten como etiquetas de `case` en una declaración de `switch` , como argumentos estándar para parámetros opcionales, como argumentos para atribuir especificaciones, y así sucesivamente.

---

Si se utilizan valores `const` en diferentes ensamblajes, se debe tener cuidado con el control de versiones. Por ejemplo, si el ensamblaje A define una `public const int MaxRetries = 3;` , y el conjunto B usa esa constante, entonces si el valor de `MaxRetries` se cambia más tarde a `5` en el conjunto A (que luego se vuelve a compilar), ese cambio no será efectivo en el conjunto B a *menos que el* conjunto B también se vuelva a compilar (con Una referencia a la nueva versión de A).

Por esa razón, si un valor puede cambiar en futuras revisiones del programa, y si el valor debe ser públicamente visible, no declare ese valor `const` menos que sepa que todos los conjuntos dependientes se volverán a compilar cada vez que se cambie algo. La alternativa es usar `static readonly` lugar de `const` , que se resuelve en tiempo de ejecución.

## espacio de nombres

La palabra clave del `namespace` es una estructura de organización que nos ayuda a comprender cómo se organiza una base de código. Los espacios de nombres en C # son espacios virtuales en lugar de estar en una carpeta física.

```
namespace StackOverflow
{
    namespace Documentation
    {
        namespace CSharp.Keywords
        {
            public class Program
            {
                public static void Main()
                {
                    Console.WriteLine(typeof(Program).Namespace);
                    //StackOverflow.Documentation.CSharp.Keywords
                }
            }
        }
    }
}
```

Los espacios de nombres en C # también se pueden escribir en sintaxis encadenada. Lo siguiente es equivalente a lo anterior:

```
namespace StackOverflow.Documentation.CSharp.Keywords
```

```
{
    public class Program
    {
        public static void Main()
        {
            Console.WriteLine(typeof(Program).Namespace);
            //StackOverflow.Documentation.CSharp.Keywords
        }
    }
}
```

## tratar, atrapar, finalmente, tirar

`try` , `catch` , `finally` y `throw` permite manejar excepciones en su código.

```
var processor = new InputProcessor();

// The code within the try block will be executed. If an exception occurs during execution of
// this code, execution will pass to the catch block corresponding to the exception type.
try
{
    processor.Process(input);
}
// If a FormatException is thrown during the try block, then this catch block
// will be executed.
catch (FormatException ex)
{
    // Throw is a keyword that will manually throw an exception, triggering any catch block
that is
    // waiting for that exception type.
    throw new InvalidOperationException("Invalid input", ex);
}
// catch can be used to catch all or any specific exceptions. This catch block,
// with no type specified, catches any exception that hasn't already been caught
// in a prior catch block.
catch
{
    LogUnexpectedException();
    throw; // Re-throws the original exception.
}
// The finally block is executed after all try-catch blocks have been; either after the try
has
// succeeded in running all commands or after all exceptions have been caught.
finally
{
    processor.Dispose();
}
```

**Nota:** la palabra clave `return` se puede usar en el bloque `try` , y el bloque `finally` todavía se ejecutará (justo antes de regresar). Por ejemplo:

```
try
{
    connection.Open();
    return connection.Get(query);
}
finally
```

```
{
    connection.Close();
}
```

La declaración `connection.Close()` se ejecutará antes de `connection.Get(query)` se devuelva el resultado de `connection.Get(query)` .

### continuar

Inmediatamente pase el control a la siguiente iteración de la construcción de bucle envolvente (para, foreach, do, while):

```
for (var i = 0; i < 10; i++)
{
    if (i < 5)
    {
        continue;
    }
    Console.WriteLine(i);
}
```

Salida:

5
6
7
8
9

Demo en vivo en .NET Fiddle

```
var stuff = new [] {"a", "b", null, "c", "d"};

foreach (var s in stuff)
{
    if (s == null)
    {
        continue;
    }
    Console.WriteLine(s);
}
```

Salida:

una
segundo
do
re

Demo en vivo en .NET Fiddle

### ref, fuera

Las palabras clave `ref` y `out` hacen que un argumento se pase por referencia, no por valor. Para los tipos de valor, esto significa que el llamado puede cambiar el valor de la variable.

```
int x = 5;
ChangeX(ref x);
// The value of x could be different now
```

Para los tipos de referencia, la instancia en la variable no solo se puede modificar (como es el caso sin `ref` ), sino que también se puede reemplazar por completo:

```
Address a = new Address();
ChangeFieldInAddress(a);
// a will be the same instance as before, even if it is modified
CreateANewInstance(ref a);
// a could be an entirely new instance now
```

La principal diferencia entre las palabras clave `out` y `ref` es que `ref` requiere que la persona inicialice la variable, mientras que `out` pasa esa responsabilidad a la persona que llama.

Para usar un parámetro de `out` , tanto la definición del método como el método de llamada deben usar explícitamente la palabra clave de `out` .

```
int number = 1;
Console.WriteLine("Before AddByRef: " + number); // number = 1
AddOneByRef(ref number);
Console.WriteLine("After AddByRef: " + number);  // number = 2
SetByOut(out number);
Console.WriteLine("After SetByOut: " + number);  // number = 34

void AddOneByRef(ref int value)
{
    value++;
}

void SetByOut(out int value)
{
    value = 34;
}
```

Demo en vivo en .NET Fiddle

Lo siguiente *no* compila, porque `out` parámetros de `out` deben tener un valor asignado antes de que el método regrese (se compilaría usando `ref` ):

```
void PrintByOut(out int value)
{
    Console.WriteLine("Hello!");
}
```

### usando la palabra clave out como modificador genérico

`out` palabra clave `out` también se puede utilizar en parámetros de tipo genérico al definir interfaces y delegados genéricos. En este caso, la palabra clave `out` especifica que el parámetro de tipo es

covariante.

> La covarianza le permite utilizar un tipo más derivado que el especificado por el parámetro genérico. Esto permite la conversión implícita de clases que implementan interfaces variantes y la conversión implícita de tipos de delegado. La covarianza y la contravarianza son compatibles con los tipos de referencia, pero no son compatibles con los tipos de valor. - MSDN

```
//if we have an interface like this
interface ICovariant<out R> { }

//and two variables like
ICovariant<Object> iobj = new Sample<Object>();
ICovariant<String> istr = new Sample<String>();

// then the following statement is valid
// without the out keyword this would have thrown error
iobj = istr; // implicit conversion occurs here
```

## comprobado, sin marcar

Las palabras clave `checked` y `unchecked` `checked` definen cómo las operaciones manejan el desbordamiento matemático. "Desbordamiento" en el contexto de las palabras clave `checked` y `unchecked` `checked` es cuando una operación aritmética entera da como resultado un valor que es mayor en magnitud de lo que puede representar el tipo de datos objetivo.

Cuando se produce un desbordamiento dentro de un bloque `checked` (o cuando el compilador está configurado para usar aritmética comprobada globalmente), se lanza una excepción para advertir de un comportamiento no deseado. Mientras tanto, en un bloque `unchecked` marcar, el desbordamiento es silencioso: no se lanzan excepciones, y el valor simplemente se ajustará al límite opuesto. Esto puede llevar a errores sutiles y difíciles de encontrar.

Como la mayoría de las operaciones aritméticas se realizan en valores que no son lo suficientemente grandes o pequeños como para desbordarse, la mayoría de las veces, no es necesario definir explícitamente un bloque como `checked` . Se debe tener cuidado al realizar operaciones aritméticas en entradas no limitadas que pueden causar un desbordamiento, por ejemplo, cuando se realizan operaciones aritméticas en funciones recursivas o al recibir entradas del usuario.

*Ni `checked` ni `unchecked` afecto flotante operaciones aritméticas de punto.*

Cuando un bloque o expresión se declara como `unchecked` , cualquier operación aritmética dentro de él puede desbordarse sin causar un error. Un ejemplo en el que se *desea* este comportamiento sería el cálculo de una suma de comprobación, donde se permite que el valor se "ajuste" durante el cálculo:

```
byte Checksum(byte[] data) {
    byte result = 0;
    for (int i = 0; i < data.Length; i++) {
        result = unchecked(result + data[i]); // unchecked expression
    }
```

```
    return result;
}
```

Uno de los usos más comunes para `unchecked` es la implementación de un reemplazo personalizado para `object.GetHashCode()` , un tipo de suma de comprobación. Puede ver el uso de la palabra clave en las respuestas a esta pregunta: ¿Cuál es el mejor algoritmo para un System.Object.GetHashCode anulado? .

Cuando se declara que un bloque o expresión está `checked` , cualquier operación aritmética que cause un desbordamiento da lugar a que se `OverflowException` una `OverflowException` .

```
int SafeSum(int x, int y) {
    checked { // checked block
        return x + y;
    }
}
```

Ambos marcados y sin marcar pueden estar en forma de bloque y expresión.

Los bloques marcados y no marcados no afectan los métodos llamados, solo los operadores llamados directamente en el método actual. Por ejemplo, `Enum.ToObject()` , `Convert.ToInt32()` , y los operadores definidos por el usuario no se ven afectados por los contextos personalizados marcados / no seleccionados.

***Nota** : El comportamiento predeterminado de desbordamiento predeterminado (marcado contra no seleccionado) puede cambiarse en las **Propiedades** del **proyecto** o mediante el interruptor de línea de comando **/ marcado [+ | -]** . Es común predeterminar las operaciones comprobadas para las compilaciones de depuración y no verificadas para las compilaciones de lanzamiento. Las palabras clave `checked` y `unchecked` marcar se usarían entonces solo cuando el enfoque predeterminado no se aplique y usted necesite un comportamiento explícito para garantizar la corrección.*

### ir

`goto` se puede usar para saltar a una línea específica dentro del código, especificada por una etiqueta.

## `goto` como a

## Etiqueta:

```
void InfiniteHello()
{
    sayHello:
    Console.WriteLine("Hello!");
    goto sayHello;
}
```

Demo en vivo en .NET Fiddle

## Declaración del caso:

```
enum Permissions { Read, Write };

switch (GetRequestedPermission())
{
    case Permissions.Read:
        GrantReadAccess();
        break;

    case Permissions.Write:
        GrantWriteAccess();
        goto case Permissions.Read; //People with write access also get read
}
```

Demo en vivo en .NET Fiddle

Esto es particularmente útil en la ejecución de múltiples comportamientos en una instrucción de conmutación, ya que C # no admite bloqueos de casos directos .

## Reintento de excepción

```
var exCount = 0;
retry:
try
{
    //Do work
}
catch (IOException)
{
    exCount++;
    if (exCount < 3)
    {
        Thread.Sleep(100);
        goto retry;
    }
    throw;
}
```

Demo en vivo en .NET Fiddle

Al igual que en muchos idiomas, se desaconseja el uso de la palabra clave goto, excepto en los casos siguientes.

Usos válidos de `goto` que se aplican a C #:

- Caso fallido en la declaración de cambio.
- Descanso multinivel. LINQ a menudo se puede usar en su lugar, pero generalmente tiene un peor rendimiento.

- Desasignación de recursos cuando se trabaja con objetos de bajo nivel no envueltos. En C #, los objetos de bajo nivel generalmente se deben envolver en clases separadas.
- Máquinas de estados finitos, por ejemplo, analizadores; utilizado internamente por el compilador generado async / await máquinas de estado.

## enumerar

La palabra clave `enum` le dice al compilador que esta clase hereda de la clase abstracta `Enum` , sin que el programador tenga que heredarla explícitamente. `Enum` es un descendiente de `ValueType` , que está diseñado para usarse con un conjunto distinto de constantes con nombre.

```
public enum DaysOfWeek
{
    Monday,
    Tuesday,
}
```

Opcionalmente, puede especificar un valor específico para cada uno (o algunos de ellos):

```
public enum NotableYear
{
   EndOfWwI = 1918;
   EnfOfWwII = 1945,
}
```

En este ejemplo, omití un valor para 0, esto suele ser una mala práctica. Una `enum` siempre tendrá un valor predeterminado producido por la conversión explícita `(YourEnumType) 0` , donde `YourEnumType` es su tipo de `enume` declarado. Sin un valor de 0 definido, una `enum` no tendrá un valor definido al inicio.

El tipo subyacente predeterminado de `enum` es `int` , puede cambiar el tipo subyacente a cualquier tipo integral, incluidos `byte` , `sbyte` , `short` , `ushort` , `int` , `uint` , `long` y `ulong` . A continuación se muestra una enumeración con el `byte` tipo subyacente:

```
enum Days : byte
{
    Sunday = 0,
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday
};
```

También tenga en cuenta que puede convertir a / desde el tipo subyacente simplemente con una conversión:

```
int value = (int)NotableYear.EndOfWwI;
```

Por estos motivos, es mejor que siempre compruebe si una `enum` es válida cuando expone las funciones de la biblioteca:

```
void PrintNotes(NotableYear year)
{
    if (!Enum.IsDefined(typeof(NotableYear), year))
        throw InvalidEnumArgumentException("year", (int)year, typeof(NotableYear));

    // ...
}
```

## base

La palabra clave **`base`** se utiliza para acceder a los miembros de una clase base. Se usa comúnmente para llamar implementaciones base de métodos virtuales, o para especificar a qué constructor base se debe llamar.

### Elegir un constructor

```
public class Child : SomeBaseClass {
    public Child() : base("some string for the base class")
    {
    }
}

public class SomeBaseClass {
    public SomeBaseClass()
    {
        // new Child() will not call this constructor, as it does not have a parameter
    }
    public SomeBaseClass(string message)
    {
        // new Child() will use this base constructor because of the specified parameter in
Child's constructor
        Console.WriteLine(message);
    }
}
```

### Llamando a la implementación base del método virtual.

```
public override void SomeVirtualMethod() {
    // Do something, then call base implementation
    base.SomeVirtualMethod();
}
```

Es posible utilizar la palabra clave base para llamar a una implementación base desde cualquier método. Esto vincula la llamada del método directamente a la implementación base, lo que significa que incluso si las nuevas clases secundarias anulan un método virtual, la implementación base se seguirá llamando, por lo que debe usarse con precaución.

```
public class Parent
{
    public virtual int VirtualMethod()
```

```
    {
        return 1;
    }
}

public class Child : Parent
{
    public override int VirtualMethod() {
        return 11;
    }

    public int NormalMethod()
    {
        return base.VirtualMethod();
    }

    public void CallMethods()
    {
        Assert.AreEqual(11, VirtualMethod());

        Assert.AreEqual(1, NormalMethod());
        Assert.AreEqual(1, base.VirtualMethod());
    }
}

public class GrandChild : Child
{
    public override int VirtualMethod()
    {
        return 21;
    }

    public void CallAgain()
    {
        Assert.AreEqual(21, VirtualMethod());
        Assert.AreEqual(11, base.VirtualMethod());

        // Notice that the call to NormalMethod below still returns the value
        // from the extreme base class even though the method has been overridden
        // in the child class.
        Assert.AreEqual(1, NormalMethod());
    }
}
```

## para cada

`foreach` se utiliza para iterar sobre los elementos de una matriz o los elementos dentro de una colección que implementa `IEnumerable` †.

```
var lines = new string[] {
    "Hello world!",
    "How are you doing today?",
    "Goodbye"
};

foreach (string line in lines)
{
    Console.WriteLine(line);
}
```

Esto dará salida

> "¡Hola Mundo!"
> "¿Cómo estás hoy?"
> "Adiós"

Demo en vivo en .NET Fiddle

Puede salir del bucle `foreach` en cualquier momento utilizando la palabra clave break o pasar a la siguiente iteración utilizando la palabra clave continue .

```
var numbers = new int[] {1, 2, 3, 4, 5, 6};

foreach (var number in numbers)
{
    // Skip if 2
    if (number == 2)
        continue;

    // Stop iteration if 5
    if (number == 5)
        break;

    Console.Write(number + ", ");
}

// Prints: 1, 3, 4,
```

Demo en vivo en .NET Fiddle

Tenga en cuenta que el orden de iteración está garantizado *solo* para ciertas colecciones como matrices y `List` , pero **no está** garantizado para muchas otras colecciones.

---

† Si bien `IEnumerable` se usa generalmente para indicar colecciones enumerables, `foreach` solo requiere que la colección exponga públicamente el método `object GetEnumerator()` del `object GetEnumerator()` , que debe devolver un objeto que expone el método `bool MoveNext()` y el `object Current { get; }` Propiedad.

## params

`params` permite que un parámetro de método reciba un número variable de argumentos, es decir, se permiten cero, uno o varios argumentos para ese parámetro.

```
static int AddAll(params int[] numbers)
{
    int total = 0;
    foreach (int number in numbers)
    {
        total += number;
    }

    return total;
}
```

Ahora se puede llamar a este método con una lista típica de argumentos `int` , o una matriz de ints.

```
AddAll(5, 10, 15, 20);                // 50
AddAll(new int[] { 5, 10, 15, 20 });  // 50
```

`params` deben aparecer como máximo una vez y, si se utilizan, deben estar en **último lugar** en la lista de argumentos, incluso si el tipo posterior es diferente al de la matriz.

---

Tenga cuidado al sobrecargar las funciones cuando use la palabra clave `params` . C # prefiere hacer coincidir sobrecargas más específicas antes de recurrir a tratar de usar sobrecargas con `params` . Por ejemplo si tienes dos métodos:

```
static double Add(params double[] numbers)
{
    Console.WriteLine("Add with array of doubles");
    double total = 0.0;
    foreach (double number in numbers)
    {
        total += number;
    }

    return total;
}

static int Add(int a, int b)
{
    Console.WriteLine("Add with 2 ints");
    return a + b;
}
```

Luego, la sobrecarga específica de 2 argumentos tendrá prioridad antes de intentar la sobrecarga de `params` .

```
Add(2, 3);      //prints "Add with 2 ints"
Add(2, 3.0);    //prints "Add with array of doubles" (doubles are not ints)
Add(2, 3, 4);   //prints "Add with array of doubles" (no 3 argument overload)
```

## descanso

En un bucle (for, foreach, do, while), la instrucción `break` la ejecución del bucle más interno y vuelve al código posterior. También se puede utilizar con un `yield` en el que se especifica que un iterador ha llegado a su fin.

```
for (var i = 0; i < 10; i++)
{
    if (i == 5)
    {
        break;
    }
    Console.WriteLine("This will appear only 5 times, as the break will stop the loop.");
}
```

Demo en vivo en .NET Fiddle

```
foreach (var stuff in stuffCollection)
{
    if (stuff.SomeStringProp == null)
        break;
    // If stuff.SomeStringProp for any "stuff" is null, the loop is aborted.
    Console.WriteLine(stuff.SomeStringProp);
}
```

La declaración de ruptura también se usa en construcciones de casos de conmutación para romper un caso o segmento predeterminado.

```
switch(a)
{
    case 5:
        Console.WriteLine("a was 5!");
        break;

    default:
        Console.WriteLine("a was something else!");
        break;
}
```

En las declaraciones de cambio, se requiere la palabra clave 'break' al final de cada declaración de caso. Esto es contrario a algunos idiomas que permiten "pasar" a la siguiente declaración de caso en la serie. Las soluciones para esto incluirían declaraciones 'goto' o apilar las declaraciones 'case' secuencialmente.

El siguiente código dará los números `0, 1, 2, ..., 9` y la última línea no se ejecutará. `yield break` significa el final de la función (no solo un bucle).

```
public static IEnumerable<int> GetNumbers()
{
    int i = 0;
    while (true) {
        if (i < 10) {
            yield return i++;
        } else {
            yield break;
        }
    }
    Console.WriteLine("This line will not be executed");
}
```

Demo en vivo en .NET Fiddle

Tenga en cuenta que, a diferencia de otros idiomas, no hay forma de etiquetar una ruptura particular en C #. Esto significa que en el caso de bucles anidados, solo se detendrá el bucle más interno:

```
foreach (var outerItem in outerList)
{
    foreach (var innerItem in innerList)
```

```
    {
        if (innerItem.ShoudBreakForWhateverReason)
            // This will only break out of the inner loop, the outer will continue:
            break;
    }
}
```

Si quiere salir del bucle *externo* aquí, puede usar una de varias estrategias diferentes, como:

- Una instrucción **goto** para saltar fuera de toda la estructura de bucle.
- Una variable de `shouldBreak` específica ( `shouldBreak` en el siguiente ejemplo) que se puede verificar al final de cada iteración del bucle externo.
- Refactorizando el código para usar una declaración de `return` en el cuerpo del bucle más interno, o evitar por completo la estructura del bucle anidado.

```
bool shouldBreak = false;
while(comeCondition)
{
    while(otherCondition)
    {
        if (conditionToBreak)
        {
            // Either tranfer control flow to the label below...
            goto endAllLooping;

            // OR use a flag, which can be checked in the outer loop:
            shouldBreak = true;
        }
    }

    if(shouldBreakNow)
    {
        break; // Break out of outer loop if flag was set to true
    }
}

endAllLooping: // label from where control flow will continue
```

## resumen

Una clase marcada con la palabra clave `abstract` no puede ser instanciada.

Una clase *debe* marcarse como abstracta si contiene miembros abstractos o si hereda miembros abstractos que no implementa. Una clase *puede* marcarse como abstracta incluso si no hay miembros abstractos involucrados.

Las clases abstractas se usan generalmente como clases base cuando alguna parte de la implementación necesita ser especificada por otro componente.

```
abstract class Animal
{
    string Name { get; set; }
    public abstract void MakeSound();
}
```

```
public class Cat : Animal
{
    public override void MakeSound()
    {
        Console.WriteLine("Meov meov");
    }
}

public class Dog : Animal
{
    public override void MakeSound()
    {
        Console.WriteLine("Bark bark");
    }
}

Animal cat = new Cat();       // Allowed due to Cat deriving from Animal
cat.MakeSound();              // will print out "Meov meov"

Animal dog = new Dog();       // Allowed due to Dog deriving from Animal
dog.MakeSound();              // will print out "Bark bark"

Animal animal = new Animal(); // Not allowed due to being an abstract class
```

Un método, propiedad o evento marcado con la palabra clave `abstract` indica que se espera que la implementación de ese miembro se proporcione en una subclase. Como se mencionó anteriormente, los miembros abstractos solo pueden aparecer en clases abstractas.

```
abstract class Animal
{
   public abstract string Name { get; set; }
}

public class Cat : Animal
{
    public override string Name { get; set; }
}

public class Dog : Animal
{
    public override string Name { get; set; }
}
```

### flotador, doble, decimal

# flotador

`float` es un alias para el tipo de datos .NET `System.Single` . Permite almacenar los números de punto flotante de precisión simple IEEE 754. Este tipo de datos está presente en `mscorlib.dll` que todos los proyectos de C # hacen referencia implícitamente cuando los creas.

Rango aproximado: -3.4 × $10^{38}$ a 3.4 × $10^{38}$

Precisión decimal: 6-9 dígitos significativos

**Notación**

```
float f = 0.1259;
var f1 = 0.7895f; // f is literal suffix to represent float values
```

> Cabe señalar que el tipo de `float` menudo produce errores de redondeo significativos. En aplicaciones donde la precisión es importante, se deben considerar otros tipos de datos.

---

# doble

`double` es un alias para el tipo de datos .NET `System.Double` . Representa un número de coma flotante de 64 bits de doble precisión. Este tipo de datos está presente en `mscorlib.dll` que se hace referencia implícitamente en cualquier proyecto de C #.

Rango: ± 5.0 × 10 $^{-324}$ a ± 1.7 × 10 $^{308}$

Precisión decimal: 15-16 dígitos significativos

**Notación**

```
double distance = 200.34; // a double value
double salary = 245; // an integer implicitly type-casted to double value
var marks = 123.764D; // D is literal suffix to represent double values
```

---

# decimal

`decimal` es un alias para el tipo de datos .NET `System.Decimal` . Representa una palabra clave indica un tipo de datos de 128 bits. En comparación con los tipos de punto flotante, el tipo decimal tiene más precisión y un rango más pequeño, lo que lo hace apropiado para los cálculos financieros y monetarios. Este tipo de datos está presente en `mscorlib.dll` que se hace referencia implícitamente en cualquier proyecto de C #.

Rango: -7.9 × 10 $^{28}$ a 7.9 × 10 $^{28}$

Precisión decimal: 28-29 dígitos significativos

**Notación**

```
decimal payable = 152.25m; // a decimal value
var marks = 754.24m; // m is literal suffix to represent decimal values
```

## uint

Un **entero sin signo** , o **uint** , es un tipo de datos numérico que solo puede contener enteros positivos. Como su nombre lo sugiere, representa un entero de 32 bits sin signo. La propia palabra clave **uint** es un alias para el tipo de sistema de tipo común `System.UInt32` . Este tipo de datos está presente en `mscorlib.dll` , al que todos los proyectos de C # hacen referencia implícitamente cuando los creas. Ocupa cuatro bytes de espacio de memoria.

Los enteros sin signo pueden contener cualquier valor de 0 a 4,294,967,295.

*Ejemplos de cómo y ahora no declarar enteros sin signo*

```
uint i = 425697; // Valid expression, explicitly stated to compiler
var i1 = 789247U; // Valid expression, suffix allows compiler to determine datatype
uint x = 3.0; // Error, there is no implicit conversion
```

---

**Tenga en cuenta:** Según Microsoft , se recomienda utilizar el tipo de datos **int** siempre que sea posible, ya que el tipo de datos **uint** no es compatible con CLS.

## esta

La palabra clave `this` refiere a la instancia actual de class (objeto). De esta manera, se pueden distinguir dos variables con el mismo nombre, una en el nivel de clase (un campo) y una que es un parámetro (o variable local) de un método.

```
public MyClass {
    int a;

    void set_a(int a)
    {
        //this.a refers to the variable defined outside of the method,
        //while a refers to the passed parameter.
        this.a = a;
    }
}
```

Otros usos de la palabra clave son el encadenamiento de sobrecargas de constructores no estáticos :

```
public MyClass(int arg) : this(arg, null)
{
}
```

y escritura de indexadores :

```
public string this[int idx1, string idx2]
{
    get { /* ... */ }
    set { /* ... */ }
}
```

y declarando métodos de extensión :

```
public static int Count<TItem>(this IEnumerable<TItem> source)
{
    // ...
}
```

Si no hay conflicto con una variable o parámetro local, es una cuestión de estilo si usar `this` o no, por lo que `this.MemberOfType` y `MemberOfType` serían equivalentes en ese caso. También vea la palabra clave `base` .

Tenga en cuenta que si se va a llamar a un método de extensión en la instancia actual, `this` es obligatorio. Por ejemplo, si está dentro de un método no estático de una clase que implementa `IEnumerable<>` y desea llamar al `Count` extensiones desde antes, debe usar:

```
this.Count()  // works like StaticClassForExtensionMethod.Count(this)
```

y `this` no puede ser omitido allí.

## para

Sintaxis: `for (initializer; condition; iterator)`

- El bucle `for` se usa comúnmente cuando se conoce el número de iteraciones.
- Las declaraciones en la sección de `initializer` se ejecutan solo una vez, antes de ingresar al bucle.
- La sección de `condition` contiene una expresión booleana que se evalúa al final de cada iteración de bucle para determinar si el bucle debería salir o debería ejecutarse de nuevo.
- La sección del `iterator` define lo que sucede después de cada iteración del cuerpo del bucle.

Este ejemplo muestra cómo se puede usar `for` para iterar sobre los caracteres de una cadena:

```
string str = "Hello";
for (int i = 0; i < str.Length; i++)
{
    Console.WriteLine(str[i]);
}
```

Salida:

H
mi
l
l
o

Demo en vivo en .NET Fiddle

Todas las expresiones que definen una sentencia `for` son opcionales; por ejemplo, la siguiente declaración se utiliza para crear un bucle infinito:

```
for( ; ; )
{
    // Your code here
}
```

La sección de `initializer` puede contener múltiples variables, siempre que sean del mismo tipo. La sección de `condition` puede consistir en cualquier expresión que pueda ser evaluada como un `bool` . Y la sección del `iterator` puede realizar múltiples acciones separadas por comas:

```
string hello = "hello";
for (int i = 0, j = 1, k = 9; i < 3 && k > 0; i++, hello += i) {
    Console.WriteLine(hello);
}
```

Salida:

Hola
hola1
hola12

Demo en vivo en .NET Fiddle

## mientras

El operador `while` itera sobre un bloque de código hasta que la consulta condicional es falsa o el código se interrumpe con una `goto` , `return` , `break` o `throw` .

Sintaxis por palabra clave `while` :

while ( *condición* ) { *bloque de código;* }

Ejemplo:

```
int i = 0;
while (i++ < 5)
{
    Console.WriteLine("While is on loop number {0}.", i);
}
```

Salida:

"Mientras está en el bucle número 1."
"Mientras está en el bucle número 2."
"Mientras está en el bucle número 3."
"Mientras está en el bucle número 4."
"Mientras está en el bucle número 5."

Demo en vivo en .NET Fiddle

Un bucle while está **controlado por entrada** , ya que la condición se verifica **antes de** la ejecución del bloque de código adjunto. Esto significa que el bucle while no ejecutaría sus

declaraciones si la condición es falsa.

```
bool a = false;

while (a == true)
{
    Console.WriteLine("This will never be printed.");
}
```

Darle una condición de `while` sin aprovisionarla para que se vuelva falso en algún punto resultará en un bucle infinito o infinito. En la medida de lo posible, esto debe evitarse, sin embargo, puede haber algunas circunstancias excepcionales cuando lo necesite.

Puede crear dicho bucle de la siguiente manera:

```
while (true)
{
//...
}
```

Tenga en cuenta que el compilador de C # transformará bucles como

```
while (true)
{
// ...
}
```

o

```
for(;;)
{
// ...
}
```

dentro

```
{
:label
// ...
goto label;
}
```

Tenga en cuenta que un bucle while puede tener cualquier condición, independientemente de su complejidad, siempre que se evalúe (o devuelva) un valor booleano (bool). También puede contener una función que devuelve un valor booleano (como una función de este tipo se evalúa al mismo tipo que una expresión como `a == x '). Por ejemplo,

```
while (AgriculturalService.MoreCornToPick(myFarm.GetAddress()))
{
    myFarm.PickCorn();
}
```

## regreso

> MSDN: la instrucción de retorno termina la ejecución del método en el que aparece y devuelve el control al método de llamada. También puede devolver un valor opcional. Si el método es un tipo nulo, se puede omitir la declaración de retorno.

```
public int Sum(int valueA, int valueB)
{
    return valueA + valueB;
}


public void Terminate(bool terminateEarly)
{
    if (terminateEarly) return; // method returns to caller if true was passed in
    else Console.WriteLine("Not early"); // prints only if terminateEarly was false
}
```

## en

La palabra clave `in` tiene tres usos:

a) Como parte de la sintaxis en una declaración `foreach` o como parte de la sintaxis en una consulta LINQ

```
foreach (var member in sequence)
{
    // ...
}
```

b) En el contexto de las interfaces genéricas y los tipos de delegados genéricos significa la *contravarianza* para el parámetro de tipo en cuestión:

```
public interface IComparer<in T>
{
    // ...
}
```

c) En el contexto de la consulta LINQ se refiere a la colección que se está consultando

```
var query = from x in source select new { x.Name, x.ID, };
```

## utilizando

Hay dos tipos de `using` palabras clave, `using statement` y `using directive` :

1. **utilizando declaración** :

   La palabra clave de `using` garantiza que los objetos que implementan la interfaz `IDisposable` se eliminan correctamente después del uso. Hay un tema separado para la declaración de

uso

2. **usando directiva**

   La directiva de `using` tiene tres usos, vea la página msdn para la directiva de uso . Hay un tema separado para la directiva using .

## sellado

Cuando se aplica a una clase, el modificador `sealed` evita que otras clases se hereden de ella.

```
class A { }
sealed class B : A { }
class C : B { } //error : Cannot derive from the sealed class
```

Cuando se aplica a un método `virtual` (o propiedad virtual), el modificador `sealed` evita que este método (propiedad) se *invalide* en las clases derivadas.

```
public class A
{
    public sealed override string ToString() // Virtual method inherited from class Object
    {
        return "Do not override me!";
    }
}

public class B: A
{
    public override string ToString() // Compile time error
    {
        return "An attempt to override";
    }
}
```

## tamaño de

Se utiliza para obtener el tamaño en bytes para un tipo no administrado

```
int byteSize = sizeof(byte) // 1
int sbyteSize = sizeof(sbyte) // 1
int shortSize = sizeof(short) // 2
int ushortSize = sizeof(ushort) // 2
int intSize = sizeof(int) // 4
int uintSize = sizeof(uint) // 4
int longSize = sizeof(long) // 8
int ulongSize = sizeof(ulong) // 8
int charSize = sizeof(char) // 2(Unicode)
int floatSize = sizeof(float) // 4
int doubleSize = sizeof(double) // 8
int decimalSize = sizeof(decimal) // 16
int boolSize = sizeof(bool) // 1
```

## estático

El modificador `static` se usa para declarar un miembro estático, que no necesita ser instanciado para poder acceder, sino que se accede a él simplemente a través de su nombre, es decir, `DateTime.Now` .

`static` se puede usar con clases, campos, métodos, propiedades, operadores, eventos y constructores.

Mientras que una instancia de una clase contiene una copia separada de todos los campos de instancia de la clase, solo hay una copia de cada campo estático.

```
class A
{
    static public int count = 0;

    public A()
    {
        count++;
    }
}

class Program
{
    static void Main(string[] args)
    {
        A a = new A();
        A b = new A();
        A c = new A();

        Console.WriteLine(A.count); // 3
    }
}
```

`count` es igual al número total de instancias de `A` clase.

El modificador estático también se puede usar para declarar un constructor estático para una clase, para inicializar datos estáticos o ejecutar código que solo necesita ser llamado una vez. Los constructores estáticos se llaman antes de que se haga referencia a la clase por primera vez.

```
class A
{
    static public DateTime InitializationTime;

    // Static constructor
    static A()
    {
        InitializationTime = DateTime.Now;
        // Guaranteed to only run once
        Console.WriteLine(InitializationTime.ToString());
    }
}
```

Una `static class` está marcada con la palabra clave `static` y puede usarse como un contenedor beneficioso para un conjunto de métodos que funcionan con parámetros, pero que no necesariamente requieren estar vinculados a una instancia. Debido a la naturaleza `static` de la clase, no se puede crear una instancia, pero puede contener un `static constructor` . Algunas

características de una `static class` incluyen:

- No puede ser heredado
- No se puede heredar de otra cosa que no sea `Object`
- Puede contener un constructor estático pero no un constructor de instancia
- Solo puede contener miembros estáticos
- Está sellado

El compilador también es amigable y le permitirá al desarrollador saber si existen miembros de la instancia dentro de la clase. Un ejemplo sería una clase estática que convierte entre métricas de EE. UU. Y Canadá:

```
static class ConversionHelper {
    private static double oneGallonPerLitreRate = 0.264172;

    public static double litreToGallonConversion(int litres) {
        return litres * oneGallonPerLitreRate;
    }
}
```

Cuando las clases son declaradas estáticas:

```
public static class Functions
{
  public static int Double(int value)
  {
    return value + value;
  }
}
```

Todas las funciones, propiedades o miembros dentro de la clase también deben declararse estáticas. No se puede crear ninguna instancia de la clase. En esencia, una clase estática le permite crear paquetes de funciones que se agrupan de forma lógica.

Dado que C # 6 `static` también se puede usar junto con el `using` para importar miembros y métodos estáticos. Se pueden usar luego sin nombre de clase.

Manera antigua, sin `using static` :

```
using System;

public class ConsoleApplication
{
    public static void Main()
    {
         Console.WriteLine("Hello World!"); //Writeline is method belonging to static class
Console
    }

}
```

Ejemplo con el `using static`

```
using static System.Console;

public class ConsoleApplication
{
    public static void Main()
    {
         WriteLine("Hello World!"); //Writeline is method belonging to static class Console
    }

}
```

# Inconvenientes

Si bien las clases estáticas pueden ser increíblemente útiles, vienen con sus propias advertencias:

- Una vez que se ha llamado a la clase estática, la clase se carga en la memoria y no se puede ejecutar a través del recolector de basura hasta que se descargue el AppDomain que contiene la clase estática.
- Una clase estática no puede implementar una interfaz.

## En t

`int` es un alias para `System.Int32` , que es un tipo de datos para enteros de 32 bits con signo. Este tipo de datos se puede encontrar en `mscorlib.dll` cual todos los proyectos de C # hacen referencia implícitamente cuando los creas.

Rango: -2,147,483,648 a 2,147,483,647

```
int int1 = -10007;
var int2 = 2132012521;
```

## largo

La palabra clave **larga** se utiliza para representar enteros de 64 bits con signo. Es un alias para el tipo de datos `System.Int64` presente en `mscorlib.dll` , al que todos los proyectos de C # hacen referencia implícitamente al crearlos.

*Cualquier variable **larga** se puede declarar explícita e implícitamente:*

```
long long1 = 9223372036854775806;  // explicit declaration, long keyword used
var long2 = -9223372036854775806L; // implicit declaration, 'L' suffix used
```

Una variable **larga** puede contener cualquier valor desde –9,223,372,036,854,775,808 a 9,223,372,036,854,775,807, y puede ser útil en situaciones en que una variable debe tener un valor que exceda los límites de lo que pueden contener otras variables (como la variable **int** ).

## ulong

Palabra clave utilizada para enteros de 64 bits sin signo. Representa el tipo de datos `System.UInt64` que se encuentra en `mscorlib.dll` que se hace referencia implícitamente en todos los proyectos de C # cuando los crea.

Rango: 0 a 18,446,744,073,709,551,615

```
ulong veryLargeInt = 18446744073609451315;
var anotherVeryLargeInt = 1544674406360945131 5UL;
```

## dinámica

La palabra clave `dynamic` se utiliza con objetos tipificados dinámicamente . Los objetos declarados como `dynamic` renuncian a las verificaciones estáticas en tiempo de compilación y, en cambio, se evalúan en tiempo de ejecución.

```
using System;
using System.Dynamic;

dynamic info = new ExpandoObject();
info.Id = 123;
info.Another = 456;

Console.WriteLine(info.Another);
// 456

Console.WriteLine(info.DoesntExist);
// Throws RuntimeBinderException
```

El siguiente ejemplo usa `dynamic` con la biblioteca Json.NET de Newtonsoft, para leer fácilmente los datos de un archivo JSON deserializado.

```
try
{
    string json = @"{ x : 10, y : ""ho""}";
    dynamic deserializedJson = JsonConvert.DeserializeObject(json);
    int x = deserializedJson.x;
    string y = deserializedJson.y;
    // int z = deserializedJson.z; // throws RuntimeBinderException
}
catch (RuntimeBinderException e)
{
    // This exception is thrown when a property
    // that wasn't assigned to a dynamic variable is used
}
```

Hay algunas limitaciones asociadas con la palabra clave dinámica. Uno de ellos es el uso de métodos de extensión. El siguiente ejemplo agrega un método de extensión para la cadena: `SayHello` .

```
static class StringExtensions
{
    public static string SayHello(this string s) => $"Hello {s}!";
}
```

El primer enfoque será llamarlo como de costumbre (como para una cadena):

```
var person = "Person";
Console.WriteLine(person.SayHello());

dynamic manager = "Manager";
Console.WriteLine(manager.SayHello()); // RuntimeBinderException
```

No hay error de compilación, pero en tiempo de ejecución obtiene una `RuntimeBinderException` . La solución para esto será llamar al método de extensión a través de la clase estática:

```
var helloManager = StringExtensions.SayHello(manager);
Console.WriteLine(helloManager);
```

### virtual, anular, nuevo

## virtual y anular

La palabra clave `virtual` permite que un método, una propiedad, un indexador o un evento sean anulados por clases derivadas y presente comportamiento polimórfico. (Los miembros son no virtuales por defecto en C #)

```
public class BaseClass
{
    public virtual void Foo()
    {
        Console.WriteLine("Foo from BaseClass");
    }
}
```

Para anular un miembro, la palabra clave de `override` se utiliza en las clases derivadas. (Note que la firma de los miembros debe ser idéntica)

```
public class DerivedClass: BaseClass
{
    public override void Foo()
    {
        Console.WriteLine("Foo from DerivedClass");
    }
}
```

El comportamiento polimórfico de los miembros virtuales significa que cuando se invoca, el miembro real que se está ejecutando se determina en tiempo de ejecución en lugar de en tiempo de compilación. El miembro que prevalece en la clase más derivada del cual el objeto particular es una instancia será el ejecutado.

En resumen, el objeto se puede declarar del tipo `BaseClass` en tiempo de compilación, pero si en tiempo de ejecución es una instancia de `DerivedClass` , el miembro anulado se ejecutará:

```
BaseClass obj1 = new BaseClass();
obj1.Foo(); //Outputs "Foo from BaseClass"

obj1 = new DerivedClass();
obj1.Foo(); //Outputs "Foo from DerivedClass"
```

Anular un método es opcional:

```
public class SecondDerivedClass: DerivedClass {}

var obj1 = new SecondDerivedClass();
obj1.Foo(); //Outputs "Foo from DerivedClass"
```

# nuevo

Dado que solo los miembros definidos como `virtual` son reemplazables y polimórficos, una clase derivada que redefine un miembro no virtual podría generar resultados inesperados.

```
public class BaseClass
{
    public void Foo()
    {
        Console.WriteLine("Foo from BaseClass");
    }
}

public class DerivedClass: BaseClass
{
    public void Foo()
    {
        Console.WriteLine("Foo from DerivedClass");
    }
}

BaseClass obj1 = new BaseClass();
obj1.Foo(); //Outputs "Foo from BaseClass"

obj1 = new DerivedClass();
obj1.Foo(); //Outputs "Foo from BaseClass" too!
```

Cuando esto sucede, el miembro ejecutado siempre se determina en el momento de la compilación en función del tipo de objeto.

- Si el objeto se declara de tipo `BaseClass` (incluso si el tiempo de ejecución es de una clase derivada), se ejecuta el método de `BaseClass`
- Si el objeto se declara de tipo `DerivedClass` entonces se `DerivedClass` el método de `DerivedClass` .

Esto suele ser un accidente (cuando se agrega un miembro al tipo base después de que se agregó uno idéntico al tipo derivado) y se genera una advertencia del compilador **CS0108** en esos escenarios.

Si fue intencional, entonces la `new` palabra clave se usa para suprimir la advertencia del compilador (¡e informar a otros desarrolladores de sus intenciones!). el comportamiento sigue siendo el mismo, la `new` palabra clave simplemente suprime la advertencia del compilador.

```
public class BaseClass
{
    public void Foo()
    {
        Console.WriteLine("Foo from BaseClass");
    }
}

public class DerivedClass: BaseClass
{
    public new void Foo()
    {
        Console.WriteLine("Foo from DerivedClass");
    }
}

BaseClass obj1 = new BaseClass();
obj1.Foo(); //Outputs "Foo from BaseClass"

obj1 = new DerivedClass();
obj1.Foo(); //Outputs "Foo from BaseClass" too!
```

# El uso de anulación *no* es opcional

A diferencia de C ++, el uso de la palabra clave de `override` *no* es opcional:

```
public class A
{
    public virtual void Foo()
    {
    }
}

public class B : A
{
    public void Foo() // Generates CS0108
    {
    }
}
```

El ejemplo anterior también provoca la advertencia **CS0108** , porque `B.Foo()` no reemplaza automáticamente a `A.Foo()` . Agregue la `override` cuando la intención sea anular la clase base y cause un comportamiento polimórfico, agregue una `new` cuando desee un comportamiento no polimórfico y resuelva la llamada utilizando el tipo estático. Este último debe usarse con precaución, ya que puede causar una confusión grave.

El siguiente código incluso resulta en un error:

```
public class A
```

```
{
    public void Foo()
    {
    }
}

public class B : A
{
    public override void Foo() // Error: Nothing to override
    {
    }
}
```

# Las clases derivadas pueden introducir polimorfismo.

El siguiente código es perfectamente válido (aunque raro):

```
    public class A
    {
        public void Foo()
        {
            Console.WriteLine("A");
        }
    }

    public class B : A
    {
        public new virtual void Foo()
        {
            Console.WriteLine("B");
        }
    }
```

Ahora todos los objetos con una referencia estática de B (y sus derivados) usan polimorfismo para resolver `Foo()` , mientras que las referencias de A usan `A.Foo()` .

```
A a = new A();
a.Foo(); // Prints "A";
a = new B();
a.Foo(); // Prints "A";
B b = new B();
b.Foo(); // Prints "B";
```

# Los métodos virtuales no pueden ser privados.

El compilador de C # es estricto en la prevención de construcciones sin sentido. Los métodos marcados como `virtual` no pueden ser privados. Debido a que un método privado no se puede

ver desde un tipo derivado, tampoco se puede sobrescribir. Esto no puede compilar:

```
public class A
{
    private virtual void Foo() // Error: virtual methods cannot be private
    {
    }
}
```

## asíncrono, espera

La palabra clave `await` se agregó como parte de la versión C # 5.0 que se admite desde Visual Studio 2012 en adelante. Aprovecha la biblioteca paralela de tareas (TPL) que hizo que el subprocesamiento múltiple sea relativamente más fácil. Las palabras clave `async` y `await` se utilizan en pares en la misma función que se muestra a continuación. La palabra clave `await` se utiliza para pausar la ejecución del método asíncrono actual hasta que se complete la tarea asíncrona esperada y / o se devuelvan sus resultados. Para utilizar la palabra clave `await` , el método que la usa debe estar marcado con la palabra clave `async` .

Se desaconseja fuertemente el uso de `async` con `void` . Para más información podéis consultar aquí .

Ejemplo:

```
public async Task DoSomethingAsync()
{
    Console.WriteLine("Starting a useless process...");
    Stopwatch stopwatch = Stopwatch.StartNew();
    int delay = await UselessProcessAsync(1000);
    stopwatch.Stop();
    Console.WriteLine("A useless process took {0} milliseconds to execute.",
stopwatch.ElapsedMilliseconds);
}

public async Task<int> UselessProcessAsync(int x)
{
    await Task.Delay(x);
    return x;
}
```

Salida:

> "Comenzando un proceso inútil ..."
>
> ** ... 1 segundo de retraso ... **
>
> "Un proceso inútil tomó 1000 milisegundos para ejecutarse".

Los pares de palabras clave `async` y `await` pueden omitirse si un método de devolución de `Task` o `Task<T>` solo devuelve una sola operación asíncrona.

*En vez de esto:*

```
public async Task PrintAndDelayAsync(string message, int delay)
{
    Debug.WriteLine(message);
    await Task.Delay(x);
}
```

*Se prefiere hacer esto:*

```
public Task PrintAndDelayAsync(string message, int delay)
{
    Debug.WriteLine(message);
    return Task.Delay(x);
}
```

5.0

En C # 5.0, la `await` no se puede usar en `catch` y `finally` .

6.0

Con C # 6.0 `await` se puede usar en `catch` y `finally` .

## carbonizarse

Un char es una sola letra almacenada dentro de una variable. Es un tipo de valor incorporado que ocupa dos bytes de espacio de memoria. Representa el tipo de datos `System.Char` que se encuentra en `mscorlib.dll` que todos los proyectos de C # hacen referencia implícitamente cuando los creas.

Hay varias formas de hacer esto.

1. `char c = 'c';`
2. `char c = '\u0063'; //Unicode`
3. `char c = '\x0063'; //Hex`
4. `char c = (char)99;//Integral`

Un char puede convertirse implícitamente en `ushort, int, uint, long, ulong, float, double,` o `decimal` y devolverá el valor entero de ese char.

```
ushort u = c;
```

devuelve 99 etc.

Sin embargo, no hay conversiones implícitas de otros tipos a char. En su lugar debes lanzarlos.

```
ushort u = 99;
 char c = (char)u;
```

## bloquear

`lock` proporciona seguridad para subprocesos para un bloque de código, de modo que solo un

subproceso puede acceder a él dentro del mismo proceso. Ejemplo:

```
private static object _lockObj = new object();
static void Main(string[] args)
{
    Task.Run(() => TaskWork());
    Task.Run(() => TaskWork());
    Task.Run(() => TaskWork());

    Console.ReadKey();
}

private static void TaskWork()
{
    lock(_lockObj)
    {
        Console.WriteLine("Entered");

        Task.Delay(3000);
        Console.WriteLine("Done Delaying");

        // Access shared resources safely

        Console.WriteLine("Leaving");
    }
}

Output:

Entered
Done Delaying
Leaving
Entered
Done Delaying
Leaving
Entered
Done Delaying
Leaving
```

**Casos de uso:**

Siempre que tenga un bloque de código que pueda producir efectos secundarios si se ejecuta por varios subprocesos al mismo tiempo. La palabra clave de bloqueo junto con un **objeto de sincronización compartido** ( `_objLock` en el ejemplo) se puede usar para evitar eso.

Tenga en cuenta que `_objLock` no puede ser `null` y que varios subprocesos que ejecutan el código deben usar la misma instancia de objeto (ya sea convirtiéndolo en un campo `static` o usando la misma instancia de clase para ambos subprocesos)

Desde el lado del compilador, la palabra clave de bloqueo es un azúcar sintáctico que se reemplaza por `Monitor.Enter(_lockObj);` y `Monitor.Exit(_lockObj);` . Entonces, si reemplaza el bloqueo rodeando el bloque de código con estos dos métodos, obtendría los mismos resultados. Puede ver el código real en Azúcar sintáctica en C #: ejemplo de bloqueo

## nulo

Una variable de un tipo de referencia puede contener una referencia válida a una instancia o una referencia nula. La referencia nula es el valor predeterminado de las variables de tipo de referencia, así como los tipos de valor que admiten valores nulos.

`null` es la palabra clave que representa una referencia nula.

Como expresión, se puede utilizar para asignar la referencia nula a las variables de los tipos mencionados:

```
object a = null;
string b = null;
int? c = null;
List<int> d  = null;
```

A los tipos de valores no anulables no se les puede asignar una referencia nula. Todas las siguientes asignaciones son inválidas:

```
int a = null;
float b = null;
decimal c = null;
```

La referencia nula *no* debe confundirse con instancias válidas de varios tipos, tales como:

- una lista vacía ( `new List<int>()` )
- una cadena vacía ( `""` )
- el número cero ( `0` , `0f` , `0m` )
- el carácter nulo ( `'\0'` )

A veces, es significativo verificar si algo es nulo o un objeto vacío / predeterminado. El método System.String.IsNullOrEmpty (String) puede usarse para verificar esto, o puede implementar su propio método equivalente.

```
private void GreetUser(string userName)
{
    if (String.IsNullOrEmpty(userName))
    {
        //The method that called us either sent in an empty string, or they sent us a null
reference. Either way, we need to report the problem.
        throw new InvalidOperationException("userName may not be null or empty.");
    }
    else
    {
        //userName is acceptable.
        Console.WriteLine("Hello, " + userName + "!");
    }
}
```

## interno

La palabra clave `internal` es un modificador de acceso para tipos y miembros de tipo. Los tipos internos o miembros son **accesibles solo dentro de los archivos en el mismo ensamblaje**

*uso:*

```
public class BaseClass
{
    // Only accessible within the same assembly
    internal static int x = 0;
}
```

La diferencia entre los diferentes modificadores de acceso se aclara aquí.

***Modificadores de acceso***

**público**

Se puede acceder al tipo o al miembro mediante cualquier otro código en el mismo conjunto u otro conjunto que lo haga referencia.

**privado**

Solo se puede acceder al tipo o miembro por código en la misma clase o estructura.

**protegido**

Solo se puede acceder al tipo o miembro por código en la misma clase o estructura, o en una clase derivada.

**interno**

Se puede acceder al tipo o miembro mediante cualquier código en el mismo ensamblaje, pero no desde otro ensamblaje.

**protegido interno**

Se puede acceder al tipo o miembro mediante cualquier código en el mismo ensamblaje, o mediante cualquier clase derivada en otro ensamblaje.

Cuando **no** se establece ningún modificador de acceso, se utiliza un modificador de acceso predeterminado. Por lo tanto, siempre hay algún tipo de modificador de acceso, incluso si no está configurado.

## dónde

`where` puede servir dos propósitos en C #: restringir el tipo en un argumento genérico y filtrar consultas LINQ.

En una clase genérica, consideremos

```
public class Cup<T>
{
```

```
    // ...
}
```

T se llama un parámetro de tipo. La definición de clase puede imponer restricciones en los tipos reales que se pueden suministrar para T.

Se pueden aplicar los siguientes tipos de restricciones:

- tipo de valor
- tipo de referencia
- Constructor predeterminado
- herencia e implementación

**tipo de valor**

En este caso, solo se pueden suministrar `struct` (esto incluye tipos de datos 'primitivos' como `int` , `boolean` , etc.)

```
public class Cup<T> where T : struct
{
    // ...
}
```

**tipo de referencia**

En este caso solo se pueden suministrar tipos de clase.

```
public class Cup<T> where T : class
{
    // ...
}
```

**valor híbrido / tipo de referencia**

Ocasionalmente, se desea restringir los argumentos de tipo a los disponibles en una base de datos, y estos generalmente se asignan a tipos de valor y cadenas. Como todas las restricciones de tipo deben cumplirse, no es posible especificar `where T : struct or string` (esto no es una sintaxis válida). Una solución es restringir los argumentos de tipo a `IConvertible` que ha incorporado tipos de "... Boolean, SByte, Byte, Int16, UInt16, Int32, UInt32, Int64, UInt64, Single, Double, Decimal, DateTime, Char y String. " Es posible que otros objetos implementen IConvertible, aunque esto es raro en la práctica.

```
public class Cup<T> where T : IConvertible
{
    // ...
}
```

**Constructor predeterminado**

Solo se permitirán los tipos que contengan un constructor por defecto. Esto incluye tipos de valor

y clases que contienen un constructor predeterminado (sin parámetros)

```
public class Cup<T> where T : new
{
    // ...
}
```

**herencia e implementación**

Solo se pueden suministrar los tipos que heredan de una determinada clase base o implementan una interfaz determinada.

```
public class Cup<T> where T : Beverage
{
    // ...
}


public class Cup<T> where T : IBeer
{
    // ...
}
```

La restricción puede incluso hacer referencia a otro parámetro de tipo:

```
public class Cup<T, U> where U : T
{
    // ...
}
```

Se pueden especificar múltiples restricciones para un argumento de tipo:

```
public class Cup<T> where T : class, new()
{
    // ...
}
```

## Los ejemplos anteriores muestran restricciones genéricas en una definición de clase, pero las restricciones se pueden usar en cualquier lugar donde se proporcione un argumento de tipo: clases, estructuras, interfaces, métodos, etc.

`where` también puede haber una cláusula LINQ. En este caso es análogo a `WHERE` en SQL:

```
int[] nums = { 5, 2, 1, 3, 9, 8, 6, 7, 2, 0 };

var query =
    from num in nums
    where num < 5
    select num;
```

```
    foreach (var n in query)
    {
        Console.Write(n + " ");
    }
    // prints 2 1 3 2 0
```

## externo

La palabra clave `extern` se utiliza para declarar métodos que se implementan externamente. Esto se puede usar junto con el atributo DllImport para llamar al código no administrado usando los servicios de Interop. que en este caso vendrá con modificador `static`

Por ejemplo:

```
using System.Runtime.InteropServices;
public class MyClass
{
    [DllImport("User32.dll")]
    private static extern int SetForegroundWindow(IntPtr point);

    public void ActivateProcessWindow(Process p)
    {
        SetForegroundWindow(p.MainWindowHandle);
    }
}
```

Esto utiliza el método SetForegroundWindow importado de la biblioteca User32.dll

Esto también se puede utilizar para definir un alias de ensamblaje externo. Lo que nos permite hacer referencia a diferentes versiones de los mismos componentes de un solo conjunto.

Para hacer referencia a dos ensamblajes con los mismos nombres de tipo completamente calificados, se debe especificar un alias en el símbolo del sistema, de la siguiente manera:

```
/r:GridV1=grid.dll
/r:GridV2=grid20.dll
```

Esto crea los alias externos GridV1 y GridV2. Para usar estos alias dentro de un programa, haga referencia a ellos usando la palabra clave extern. Por ejemplo:

```
extern alias GridV1;
extern alias GridV2;
```

## bool

Palabra clave para almacenar los valores booleanos `true` y `false` . bool es un alias de System.Boolean.

El valor predeterminado de un bool es falso.

```
bool b; // default value is false
b = true; // true
b = ((5 + 2) == 6); // false
```

Para que un bool permita valores nulos, debe inicializarse como un bool ?.

El valor por defecto de un bool? es nulo.

```
bool? a // default value is null
```

## cuando

El `when` es una palabra clave agregada en **C # 6** , y se usa para el filtrado de excepciones.

Antes de la introducción de la palabra clave `when` , podría haber tenido una cláusula catch para cada tipo de excepción; con la adición de la palabra clave, ahora es posible un control más preciso.

A `when` expresión se adjunta a una rama `catch` , y solo si la condición `when` es `true` , se ejecutará la cláusula `catch` . Es posible tener varias cláusulas `catch` con los mismos tipos de clase de excepción y diferentes `when` condiciones.

```
private void CatchException(Action action)
{
    try
    {
        action.Invoke();
    }

    // exception filter
    catch (Exception ex) when (ex.Message.Contains("when"))
    {
        Console.WriteLine("Caught an exception with when");
    }

    catch (Exception ex)
    {
        Console.WriteLine("Caught an exception without when");
    }
}

private void Method1() { throw new Exception("message for exception with when"); }
private void Method2() { throw new Exception("message for general exception"); }


CatchException(Method1);
CatchException(Method2);
```

## desenfrenado

La palabra clave `unchecked` evita que el compilador compruebe desbordamientos / subdesbordos.

Por ejemplo:

```
const int ConstantMax = int.MaxValue;
unchecked
{
    int1 = 2147483647 + 10;
}
int1 = unchecked(ConstantMax + 10);
```

Sin la palabra clave `unchecked` marcar, ninguna de las dos operaciones de adición se compilará.

# ¿Cuándo es esto útil?

Esto es útil ya que puede ayudar a acelerar los cálculos que definitivamente no se desbordarán ya que la verificación del desbordamiento lleva tiempo, o cuando se desea un comportamiento de desbordamiento / subdesbordamiento (por ejemplo, al generar un código hash).

### vacío

La palabra reservada `"void"` es un alias de tipo `System.Void` , y tiene dos usos:

1. Declare un método que no tiene un valor de retorno:

```
public void DoSomething()
{
    // Do some work, don't return any value to the caller.
}
```

Un método con un tipo de retorno de vacío todavía puede tener la palabra clave de `return` en su cuerpo. Esto es útil cuando desea salir de la ejecución del método y devolver el flujo a la persona que llama:

```
public void DoSomething()
{
    // Do some work...

    if (condition)
        return;

    // Do some more work if the condition evaluated to false.
}
```

2. Declare un puntero a un tipo desconocido en un contexto inseguro.

En un contexto inseguro, un tipo puede ser un tipo de puntero, un tipo de valor o un tipo de referencia. Una declaración de tipo de puntero suele ser `type* identifier` , donde el tipo es un tipo conocido, es decir, `int* myInt` , pero también puede ser `void* identifier` , donde el tipo es desconocido.

Tenga en cuenta que Microsoft no recomienda declarar un tipo de puntero nulo .

### si, si ... más, si ... más si

La sentencia `if` se usa para controlar el flujo del programa. Una sentencia `if` identifica qué sentencia ejecutar según el valor de una expresión `Boolean` .

Para una sola declaración, las `braces` {} son opcionales pero se recomiendan.

```
int a = 4;
if(a % 2 == 0)
{
     Console.WriteLine("a contains an even number");
}
// output: "a contains an even number"
```

---

El `if` también puede tener una cláusula `else` , que se ejecutará en caso de que la condición se evalúe como falsa:

```
int a = 5;
if(a % 2 == 0)
{
     Console.WriteLine("a contains an even number");
}
else
{
     Console.WriteLine("a contains an odd number");
}
// output: "a contains an odd number"
```

---

La construcción `if` ... `else if` permite especificar múltiples condiciones:

```
int a = 9;
if(a % 2 == 0)
{
     Console.WriteLine("a contains an even number");
}
else if(a % 3 == 0)
{
     Console.WriteLine("a contains an odd number that is a multiple of 3");
}
else
{
     Console.WriteLine("a contains an odd number");
}
// output: "a contains an odd number that is a multiple of 3"
```

## ***Es importante tener en cuenta*** **que si se cumple una condición en el ejemplo anterior, el control omite otras pruebas y salta al final de esa construcción particular. De lo contrario, el *orden* de las pruebas es importante si está utilizando if ... else if construct**

Las expresiones booleanas de C # utilizan la evaluación de cortocircuito . Esto es importante en los casos en que las condiciones de evaluación pueden tener efectos secundarios:

```
if (someBooleanMethodWithSideEffects() && someOtherBooleanMethodWithSideEffects()) {
  //...
}
```

No hay garantía de que se `someOtherBooleanMethodWithSideEffects` realmente algún otro `someOtherBooleanMethodWithSideEffects` con `someOtherBooleanMethodWithSideEffects` .

También es importante en los casos en que las condiciones anteriores aseguran que es "seguro" evaluar las posteriores. Por ejemplo:

```
if (someCollection != null && someCollection.Count > 0) {
   // ..
}
```

El orden es muy importante en este caso porque, si revertimos el orden:

```
if (someCollection.Count > 0 && someCollection != null) {
```

lanzará una `NullReferenceException` si `someCollection` es `null` .

## hacer

El operador do itera sobre un bloque de código hasta que una consulta condicional es igual a falso. El bucle do-while también puede ser interrumpido por una `goto` , `return` , `break` o `throw` .

La sintaxis de la palabra clave `do` es:

> hacer { *bloque de código;* } while ( *condición* );

Ejemplo:

```
int i = 0;

do
{
    Console.WriteLine("Do is on loop number {0}.", i);
} while (i++ < 5);
```

Salida:

> "Do está en el bucle número 1."
> "Do está en el bucle número 2."
> "Do está en el bucle número 3."
> "Do está en el bucle número 4."
> "Do está en el bucle número 5".

A diferencia del `while` de bucle, el bucle do-while es **la salida controlada.** Esto significa que el

bucle do-while ejecutaría sus declaraciones al menos una vez, incluso si la condición falla la primera vez.

```
bool a = false;

do
{
    Console.WriteLine("This will be printed once, even if a is false.");
} while (a == true);
```

## operador

La mayoría de los operadores integrados (incluidos los operadores de conversión) se pueden sobrecargar utilizando la palabra clave del `operator` junto con los modificadores `public` y `static` .

Los operadores se presentan en tres formas: operadores unarios, operadores binarios y operadores de conversión.

Los operadores unarios y binarios requieren al menos un parámetro del mismo tipo que el tipo que contiene, y algunos requieren un operador coincidente complementario.

Los operadores de conversión deben convertir hacia o desde el tipo adjunto.

```
public struct Vector32
{

    public Vector32(int x, int y)
    {
        X = x;
        Y = y;
    }

    public int X { get; }
    public int Y { get; }

    public static bool operator ==(Vector32 left, Vector32 right)
        => left.X == right.X && left.Y == right.Y;

    public static bool operator !=(Vector32 left, Vector32 right)
        => !(left == right);

    public static Vector32 operator +(Vector32 left, Vector32 right)
        => new Vector32(left.X + right.X, left.Y + right.Y);

    public static Vector32 operator +(Vector32 left, int right)
        => new Vector32(left.X + right, left.Y + right);

    public static Vector32 operator +(int left, Vector32 right)
        => right + left;

    public static Vector32 operator -(Vector32 left, Vector32 right)
        => new Vector32(left.X - right.X, left.Y - right.Y);

    public static Vector32 operator -(Vector32 left, int right)
        => new Vector32(left.X - right, left.Y - right);
```

```
    public static Vector32 operator -(int left, Vector32 right)
        => right - left;

    public static implicit operator Vector64(Vector32 vector)
        => new Vector64(vector.X, vector.Y);

    public override string ToString() => $"{{{X}, {Y}}}";

}

public struct Vector64
{

    public Vector64(long x, long y)
    {
        X = x;
        Y = y;
    }

    public long X { get; }
    public long Y { get; }

    public override string ToString() => $"{{{X}, {Y}}}";

}
```

**Ejemplo**

```
var vector1 = new Vector32(15, 39);
var vector2 = new Vector32(87, 64);

Console.WriteLine(vector1 == vector2); // false
Console.WriteLine(vector1 != vector2); // true
Console.WriteLine(vector1 + vector2);  // {102, 103}
Console.WriteLine(vector1 - vector2);  // {-72, -25}
```

## estructura

Un tipo de `struct` es un tipo de valor que normalmente se usa para encapsular pequeños grupos de variables relacionadas, como las coordenadas de un rectángulo o las características de un artículo en un inventario.

> Las clases son tipos de referencia, las estructuras son tipos de valor.

```
using static System.Console;

namespace ConsoleApplication1
{
    struct Point
    {
        public int X;
        public int Y;

        public override string ToString()
        {
            return $"X = {X}, Y = {Y}";
        }
```

```
        public void Display(string name)
        {
            WriteLine(name + ": " + ToString());
        }
    }

    class Program
    {
        static void Main()
        {
            var point1 = new Point {X = 10, Y = 20};
            // it's not a reference but value type
            var point2 = point1;
            point2.X = 777;
            point2.Y = 888;
            point1.Display(nameof(point1)); // point1: X = 10, Y = 20
            point2.Display(nameof(point2)); // point2: X = 777, Y = 888

            ReadKey();
        }
    }
}
```

---

Las estructuras también pueden contener constructores, constantes, campos, métodos, propiedades, indizadores, operadores, eventos y tipos anidados, aunque si se requieren varios de estos miembros, debería considerar convertir su tipo en una clase.

---

**Algunas sugerencias de MS sobre cuándo usar struct y cuándo usar class:**

*CONSIDERAR*

Definir una estructura en lugar de una clase si las instancias del tipo son pequeñas y comúnmente duran poco o están comúnmente incrustadas en otros objetos.

*EVITAR*

definiendo una estructura a menos que el tipo tenga todas las siguientes características:

- Lógicamente representa un solo valor, similar a los tipos primitivos (int, double, etc.)
- Tiene un tamaño de instancia inferior a 16 bytes.
- Es inmutable.
- No tendrá que ser boxeado con frecuencia.

## cambiar

La instrucción de `switch` es una instrucción de control que selecciona una sección de cambio para ejecutar desde una lista de candidatos. Una declaración de conmutación incluye una o más secciones de conmutación. Cada sección de cambio contiene una o más etiquetas de `case` seguidas de una o más declaraciones. Si ninguna etiqueta de caso contiene un valor coincidente, el control se transfiere a la sección `default` , si existe. El caso de fallos no se admite en C #, estrictamente hablando. Sin embargo, si 1 o más etiquetas de `case` están vacías, la ejecución

seguirá el código del siguiente bloque de `case` que contiene código. Esto permite agrupar múltiples etiquetas de `case` con la misma implementación. En el siguiente ejemplo, si `month` es igual a 12, el código en el `case 2` se ejecutará ya que las etiquetas de `case 12 1` y `2` están agrupadas. Si un bloque de `case` no está vacío, debe haber una `break` antes de la siguiente etiqueta de `case` , de lo contrario, el compilador marcará un error.

```
int month = DateTime.Now.Month; // this is expected to be 1-12 for Jan-Dec

switch (month)
{
    case 12:
    case 1:
    case 2:
        Console.WriteLine("Winter");
        break;
    case 3:
    case 4:
    case 5:
        Console.WriteLine("Spring");
        break;
    case 6:
    case 7:
    case 8:
        Console.WriteLine("Summer");
        break;
    case 9:
    case 10:
    case 11:
        Console.WriteLine("Autumn");
        break;
    default:
        Console.WriteLine("Incorrect month index");
        break;
}
```

Un `case` solo se puede etiquetar con un valor conocido en el *momento de la compilación* (por ejemplo, `1` , `"str"` , `Enum.A` ), por lo que una `variable` no es una etiqueta de `case` válida, pero un valor `const` o `Enum` es (así como cualquier valor literal).

## interfaz

Una `interface` contiene las firmas de métodos, propiedades y eventos. Las clases derivadas definen a los miembros ya que la interfaz solo contiene la declaración de los miembros.

Se declara una interfaz usando la palabra clave de la `interface` .

```
interface IProduct
{
    decimal Price { get; }
}

class Product : IProduct
{
    const decimal vat = 0.2M;
```

```
    public Product(decimal price)
    {
        _price = price;
    }

    private decimal _price;
    public decimal Price { get { return _price * (1 + vat); } }
}
```

## inseguro

La palabra clave `unsafe` se puede usar en declaraciones de tipo o método o para declarar un bloque en línea.

El propósito de esta palabra clave es habilitar el uso del *subconjunto inseguro* de C # para el bloque en cuestión. El subconjunto inseguro incluye características como punteros, asignación de pila, matrices tipo C, etc.

El código inseguro no es verificable y es por eso que se desaconseja su uso. La compilación de código inseguro requiere pasar un interruptor al compilador de C #. Además, el CLR requiere que el ensamblado en ejecución tenga plena confianza.

A pesar de estas limitaciones, el código no seguro tiene usos válidos para hacer que algunas operaciones sean más eficaces (por ejemplo, indexación de matrices) o más fáciles (por ejemplo, interoperabilidad con algunas bibliotecas no administradas).

Como un ejemplo muy simple.

```
// compile with /unsafe
class UnsafeTest
{
   unsafe static void SquarePtrParam(int* p)
   {
      *p *= *p; // the '*' dereferences the pointer.
      //Since we passed in "the address of i", this becomes "i *= i"
   }

   unsafe static void Main()
   {
      int i = 5;
      // Unsafe method: uses address-of operator (&):
      SquarePtrParam(&i); // "&i" means "the address of i". The behavior is similar to "ref i"
      Console.WriteLine(i); // Output: 25
   }
}
```

Mientras trabajamos con punteros, podemos cambiar los valores de las ubicaciones de la memoria directamente, en lugar de tener que abordarlos por nombre. Tenga en cuenta que esto a menudo requiere el uso de la palabra clave fija para evitar posibles daños en la memoria, ya que el recolector de basura mueve las cosas (de lo contrario, puede obtener el error CS0212 ). Ya que una variable que se ha "arreglado" no se puede escribir, a menudo también tenemos que tener un segundo puntero que comienza apuntando a la misma ubicación que la primera.

```
void Main()
{
    int[] intArray = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    UnsafeSquareArray(intArray);
    foreach(int i in intArray)
        Console.WriteLine(i);
}

unsafe static void UnsafeSquareArray(int[] pArr)
{
    int len = pArr.Length;

    //in C or C++, we could say
    // int* a = &(pArr[0])
    // however, C# requires you to "fix" the variable first
    fixed(int* fixedPointer = &(pArr[0]))
    {
        //Declare a new int pointer because "fixedPointer" cannot be written to.
        // "p" points to the same address space, but we can modify it
        int* p = fixedPointer;

        for (int i = 0; i < len; i++)
        {
            *p *= *p; //square the value, just like we did in SquarePtrParam, above
            p++;      //move the pointer to the next memory space.
                      // NOTE that the pointer will move 4 bytes since "p" is an
                      // int pointer and an int takes 4 bytes

            //the above 2 lines could be written as one, like this:
            // "*p *= *p++;"
        }
    }
}
```

Salida:

```
1
4
9
16
25
36
49
64
81
100
```

`unsafe` también permite el uso de stackalloc que asignará memoria en la pila como _alloca en la biblioteca en tiempo de ejecución de C. Podemos modificar el ejemplo anterior para usar `stackalloc` siguiente manera:

```
unsafe void Main()
{
    const int len=10;
    int* seedArray = stackalloc int[len];

    //We can no longer use the initializer "{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}" as before.
```

```
    // We have at least 2 options to populate the array. The end result of either
    // option will be the same (doing both will also be the same here).

    //FIRST OPTION:
    int* p = seedArray; // we don't want to lose where the array starts, so we
                        // create a shadow copy of the pointer
    for(int i=1; i<=len; i++)
        *p++ = i;
    //end of first option

    //SECOND OPTION:
    for(int i=0; i<len; i++)
        seedArray[i] = i+1;
    //end of second option

    UnsafeSquareArray(seedArray, len);
    for(int i=0; i< len; i++)
        Console.WriteLine(seedArray[i]);
}

//Now that we are dealing directly in pointers, we don't need to mess around with
// "fixed", which dramatically simplifies the code
unsafe static void UnsafeSquareArray(int* p, int len)
{
    for (int i = 0; i < len; i++)
        *p *= *p++;
}
```

(La salida es la misma que la anterior)

## implícito

La palabra clave `implicit` se utiliza para sobrecargar un operador de conversión. Por ejemplo, puede declarar una clase de `Fraction` que debería convertirse automáticamente a `double` cuando sea necesario, y que puede convertirse automáticamente desde `int` :

```
class Fraction(int numerator, int denominator)
{
    public int Numerator { get; } = numerator;
    public int Denominator { get; } = denominator;
    // ...
    public static implicit operator double(Fraction f)
    {
        return f.Numerator / (double) f.Denominator;
    }
    public static implicit operator Fraction(int i)
    {
        return new Fraction(i, 1);
    }
}
```

## verdadero Falso

Las palabras clave de `true` y `false` tienen dos usos:

1. Como valores booleanos literales

```
var myTrueBool = true;
var myFalseBool = false;
```

2. Como operadores que pueden sobrecargarse.

```
public static bool operator true(MyClass x)
{
    return x.value >= 0;
}

public static bool operator false(MyClass x)
{
    return x.value < 0;
}
```

La sobrecarga del operador falso fue útil antes de C # 2.0, antes de la introducción de los tipos de `Nullable` .
Un tipo que sobrecargue al operador `true` , también debe sobrecargar al operador `false` .

## cuerda

`string` es un alias del tipo de datos .NET `System.String` , que permite almacenar texto (secuencias de caracteres).

Notación:

```
string a = "Hello";
var b = "world";
var f = new string(new []{ 'h', 'i', '!' }); // hi!
```

Cada carácter de la cadena está codificado en UTF-16, lo que significa que cada carácter requerirá un mínimo de 2 bytes de espacio de almacenamiento.

## ushort

Un tipo numérico utilizado para almacenar enteros positivos de 16 bits. `ushort` es un alias para `System.UInt16` , y ocupa 2 bytes de memoria.

El rango válido es de `0` a `65535` .

```
ushort a = 50; // 50
ushort b = 65536; // Error, cannot be converted
ushort c = unchecked((ushort)65536); // Overflows (wraps around to 0)
```

## sbyte

Un tipo numérico utilizado para almacenar enteros con *signo de* 8 bits. `sbyte` es un alias para `System.SByte` y ocupa 1 byte de memoria. Para el equivalente sin firmar, use `byte` .

El rango válido es de `-127` a `127` (el resto se utiliza para almacenar el letrero).

```
sbyte a = 127; // 127
sbyte b = -127; // -127
sbyte c = 200; // Error, cannot be converted
sbyte d = unchecked((sbyte)129); // -127 (overflows)
```

### var

Una variable local de tipo implícito que se escribe fuertemente como si el usuario hubiera declarado el tipo. A diferencia de otras declaraciones de variables, el compilador determina el tipo de variable que esto representa en función del valor que se le asigna.

```
var i = 10; // implicitly typed, the compiler must determine what type of variable this is
int i = 10; // explicitly typed, the type of variable is explicitly stated to the compiler

// Note that these both represent the same type of variable (int) with the same value (10).
```

A diferencia de otros tipos de variables, las definiciones de variables con esta palabra clave deben inicializarse cuando se declaran. Esto se debe a que la palabra clave **var** representa una variable de tipo implícito.

```
var i;
i = 10;

// This code will not run as it is not initialized upon declaration.
```

La palabra clave **var** también se puede utilizar para crear nuevos tipos de datos sobre la marcha. Estos nuevos tipos de datos se conocen como *tipos anónimos* . Son muy útiles, ya que permiten a un usuario definir un conjunto de propiedades sin tener que declarar explícitamente ningún tipo de objeto primero.

*Tipo anónimo*

```
var a = new { number = 1, text = "hi" };
```

*Consulta LINQ que devuelve un tipo anónimo*

```
public class Dog
{
    public string Name { get; set; }
    public int Age { get; set; }
}

public class DogWithBreed
{
    public Dog Dog { get; set; }
    public string BreedName  { get; set; }
}

public void GetDogsWithBreedNames()
{
    var db = new DogDataContext(ConnectString);
    var result = from d in db.Dogs
```

```
            join b in db.Breeds on d.BreedId equals b.BreedId
            select new
                   {
                       DogName = d.Name,
                       BreedName = b.BreedName
                   };

    DoStuff(result);
}
```

Puedes usar la palabra clave var en la sentencia foreach

```
public bool hasItemInList(List<String> list, string stringToSearch)
{
    foreach(var item in list)
    {
        if( ( (string)item ).equals(stringToSearch) )
            return true;
    }

    return false;
}
```

## delegar

Los delegados son tipos que representan una referencia a un método. Se utilizan para pasar métodos como argumentos a otros métodos.

Los delegados pueden mantener métodos estáticos, métodos de instancia, métodos anónimos o expresiones lambda.

```
class DelegateExample
{
    public void Run()
    {
        //using class method
        InvokeDelegate( WriteToConsole );

        //using anonymous method
        DelegateInvoker di = delegate ( string input )
        {
            Console.WriteLine( string.Format( "di: {0} ", input ) );
            return true;
        };
        InvokeDelegate( di );

        //using lambda expression
        InvokeDelegate( input => false );
    }

    public delegate bool DelegateInvoker( string input );

    public void InvokeDelegate(DelegateInvoker func)
    {
        var ret = func( "hello world" );
        Console.WriteLine( string.Format( " > delegate returned {0}", ret ) );
    }
```

```
    public bool WriteToConsole( string input )
    {
        Console.WriteLine( string.Format( "WriteToConsole: '{0}'", input ) );
        return true;
    }
}
```

Al asignar un método a un delegado, es importante tener en cuenta que el método debe tener el mismo tipo de retorno, así como los parámetros. Esto difiere de la sobrecarga del método 'normal', donde solo los parámetros definen la firma del método.

Los eventos se construyen sobre los delegados.

## evento

Un `event` permite al desarrollador implementar un patrón de notificación.

**Ejemplo simple**

```
public class Server
{
    // defines the event
    public event EventHandler DataChangeEvent;

    void RaiseEvent()
    {
        var ev = DataChangeEvent;
        if(ev != null)
        {
            ev(this, EventArgs.Empty);
        }
    }
}

public class Client
{
    public void Client(Server server)
    {
        // client subscribes to the server's DataChangeEvent
        server.DataChangeEvent += server_DataChanged;
    }

    private void server_DataChanged(object sender, EventArgs args)
    {
        // notified when the server raises the DataChangeEvent
    }
}
```

Referencia de MSDN

## parcial

La palabra clave `partial` se puede usar durante la definición de tipo de clase, estructura o interfaz para permitir que la definición de tipo se divida en varios archivos. Esto es útil para incorporar

nuevas características en el código generado automáticamente.

**File1.cs**

```
namespace A
{
    public partial class Test
    {
        public string Var1 {get;set;}
    }
}
```

**File2.cs**

```
namespace A
{
    public partial class Test
    {
        public string Var2 {get;set;}
    }
}
```

**Nota:** Una clase se puede dividir en cualquier número de archivos. Sin embargo, todas las declaraciones deben estar bajo el mismo espacio de nombres y el mismo ensamblado.

Los métodos también se pueden declarar parciales usando la palabra clave `partial` . En este caso, un archivo contendrá solo la definición del método y otro archivo contendrá la implementación.

> Un método parcial tiene su firma definida en una parte de un tipo parcial, y su implementación definida en otra parte del tipo. Los métodos parciales permiten a los diseñadores de clase proporcionar enlaces de métodos, similares a los controladores de eventos, que los desarrolladores pueden decidir implementar o no. Si el desarrollador no proporciona una implementación, el compilador elimina la firma en el momento de la compilación. Las siguientes condiciones se aplican a los métodos parciales:
>
> - Las firmas en ambas partes del tipo parcial deben coincidir.
> - El método debe devolver vacío.
> - No se permiten modificadores de acceso. Los métodos parciales son implícitamente privados.
>
> - MSDN

**File1.cs**

```
namespace A
{
    public partial class Test
    {
        public string Var1 {get;set;}
        public partial Method1(string str);
```

```
    }
}
```

**File2.cs**

```
namespace A
{
    public partial class Test
    {
        public string Var2 {get;set;}
        public partial Method1(string str)
        {
            Console.WriteLine(str);
        }
    }
}
```

**Nota:** El tipo que contiene el método parcial también debe ser declarado parcial.

Lea Palabras clave en línea: https://riptutorial.com/es/csharp/topic/26/palabras-clave

# Capítulo 129: Patrones de diseño creacional

## Observaciones

Los patrones creacionales apuntan a separar un sistema de cómo se crean, se componen y se representan sus objetos. Aumentan la flexibilidad del sistema en términos de qué, quién, cómo y cuándo de creación de objetos. Los patrones de creación encapsulan el conocimiento sobre qué clases usa un sistema, pero ocultan los detalles de cómo se crean y se juntan las instancias de estas clases. Los programadores se han dado cuenta de que componer sistemas con herencia hace que esos sistemas sean demasiado rígidos. Los patrones de creación están diseñados para romper este acoplamiento cercano.

## Examples

### Patrón Singleton

El patrón Singleton está diseñado para restringir la creación de una clase a una sola instancia.

Este patrón se usa en un escenario donde tiene sentido tener solo uno de algo, como por ejemplo:

- una sola clase que organiza las interacciones de otros objetos, ej. Clase de gerente
- o una clase que representa un recurso único y único, ej. Componente de registro

Una de las formas más comunes de implementar el patrón Singleton es a través de un **método de fábrica** estático como `CreateInstance()` o `GetInstance()` (o una propiedad estática en C #, `Instance` ), que luego está diseñada para devolver siempre la misma instancia.

La primera llamada al método o propiedad crea y devuelve la instancia de Singleton. A partir de entonces, el método siempre devuelve la misma instancia. De esta manera, solo hay una instancia del objeto singleton.

La prevención de la creación de instancias a través de `new` puede lograrse haciendo que los constructores de la clase sean `private`.

Este es un ejemplo típico de código para implementar un patrón Singleton en C #:

```
class Singleton
{
    // Because the _instance member is made private, the only way to get the single
    // instance is via the static Instance property below. This can also be similarly
    // achieved with a GetInstance() method instead of the property.
    private static Singleton _instance = null;

    // Making the constructor private prevents other instances from being
    // created via something like Singleton s = new Singleton(), protecting
    // against unintentional misuse.
    private Singleton()
```

```
    {
    }

    public static Singleton Instance
    {
        get
        {
            // The first call will create the one and only instance.
            if (_instance == null)
            {
                _instance = new Singleton();
            }

            // Every call afterwards will return the single instance created above.
            return _instance;
        }
    }
}
```

Para ilustrar aún más este patrón, el siguiente código comprueba si se devuelve una instancia idéntica de Singleton cuando se llama a la propiedad Instancia más de una vez.

```
class Program
{
    static void Main(string[] args)
    {
        Singleton s1 = Singleton.Instance;
        Singleton s2 = Singleton.Instance;

        // Both Singleton objects above should now reference the same Singleton instance.
        if (Object.ReferenceEquals(s1, s2))
        {
            Console.WriteLine("Singleton is working");
        }
        else
        {
            // Otherwise, the Singleton Instance property is returning something
            // other than the unique, single instance when called.
            Console.WriteLine("Singleton is broken");
        }
    }
}
```

Nota: esta implementación no es segura para subprocesos.

Para ver más ejemplos, incluyendo cómo hacer que este hilo sea seguro, visite: Implementación de Singleton

Los Singletons son conceptualmente similares a un valor global y causan fallas e inquietudes de diseño similares. Debido a esto, el patrón Singleton es ampliamente considerado como un anti-patrón.

Visita "¿Qué tiene de malo Singletons?" Para más información sobre los problemas que surgen con su uso.

En C #, tiene la capacidad de hacer que una clase sea `static` , lo que hace que todos los

miembros sean estáticos, y la clase no puede ser instanciada. Dado esto, es común ver las clases estáticas utilizadas en lugar del patrón Singleton.

Para conocer las diferencias clave entre los dos, visite C # Singleton Pattern Versus Static Class .

## Patrón de método de fábrica

Método de fábrica es uno de los patrones de diseño creativo. Se utiliza para tratar el problema de crear objetos sin especificar el tipo de resultado exacto. Este documento le enseñará cómo utilizar Factory Method DP correctamente.

Déjame explicarte la idea de ello en un ejemplo simple. Imagine que está trabajando en una fábrica que produce tres tipos de dispositivos: amperímetro, voltímetro y medidor de resistencia. Está escribiendo un programa para una computadora central que creará el dispositivo seleccionado, pero no sabe la decisión final de su jefe sobre qué producir.

`IDevice` un `IDevice` interfaz con algunas funciones comunes que todos los dispositivos tienen:

```
public interface IDevice
{
    int Measure();
    void TurnOff();
    void TurnOn();
}
```

Ahora, podemos crear clases que representen nuestros dispositivos. Esas clases deben implementar la interfaz `IDevice` :

```
public class AmMeter : IDevice
{
    private Random r = null;
    public AmMeter()
    {
        r = new Random();
    }
    public int Measure() { return r.Next(-25, 60); }
    public void TurnOff() { Console.WriteLine("AmMeter flashes lights saying good bye!"); }
    public void TurnOn() { Console.WriteLine("AmMeter turns on..."); }
}
public class OhmMeter : IDevice
{
    private Random r = null;
    public OhmMeter()
    {
        r = new Random();
    }
    public int Measure() { return r.Next(0, 1000000); }
    public void TurnOff() { Console.WriteLine("OhmMeter flashes lights saying good bye!"); }
    public void TurnOn() { Console.WriteLine("OhmMeter turns on..."); }
}
public class VoltMeter : IDevice
{
    private Random r = null;
    public VoltMeter()
    {
```

```
        r = new Random();
    }
    public int Measure() { return r.Next(-230, 230); }
    public void TurnOff() { Console.WriteLine("VoltMeter flashes lights saying good bye!"); }
    public void TurnOn() { Console.WriteLine("VoltMeter turns on..."); }
}
```

Ahora tenemos que definir el método de fábrica. Vamos a crear la clase `DeviceFactory` con el método estático dentro de:

```
public enum Device
{
    AM,
    VOLT,
    OHM
}
public class DeviceFactory
{
    public static IDevice CreateDevice(Device d)
    {
        switch(d)
        {
            case Device.AM: return new AmMeter();
            case Device.VOLT: return new VoltMeter();
            case Device.OHM: return new OhmMeter();
            default: return new AmMeter();
        }
    }
}
```

¡Genial! Probemos nuestro código:

```
public class Program
{
    static void Main(string[] args)
    {
        IDevice device = DeviceFactory.CreateDevice(Device.AM);
        device.TurnOn();
        Console.WriteLine(device.Measure());
        Console.WriteLine(device.Measure());
        Console.WriteLine(device.Measure());
        Console.WriteLine(device.Measure());
        Console.WriteLine(device.Measure());
        device.TurnOff();
        Console.WriteLine();

        device = DeviceFactory.CreateDevice(Device.VOLT);
        device.TurnOn();
        Console.WriteLine(device.Measure());
        Console.WriteLine(device.Measure());
        Console.WriteLine(device.Measure());
        Console.WriteLine(device.Measure());
        Console.WriteLine(device.Measure());
        device.TurnOff();
        Console.WriteLine();

        device = DeviceFactory.CreateDevice(Device.OHM);
        device.TurnOn();
```

```
        Console.WriteLine(device.Measure());
        Console.WriteLine(device.Measure());
        Console.WriteLine(device.Measure());
        Console.WriteLine(device.Measure());
        Console.WriteLine(device.Measure());
        device.TurnOff();
        Console.WriteLine();
    }
}
```

Este es el ejemplo de salida que puede ver después de ejecutar este código:

AmMeter enciende ...

36

6

33

43

24

AmMeter parpadea luces diciendo adiós!

VoltMeter enciende ...

102

-61

85

138

36

VoltMeter parpadea luces diciendo adiós!

OhmMeter enciende ...

723828

368536

685412

800266

578595

OhmMeter parpadea luces diciendo adiós!

## Patrón de constructor

Separe la construcción de un objeto complejo de su representación para que el mismo proceso de construcción pueda crear diferentes representaciones y proporcione un alto nivel de control sobre el ensamblaje de los objetos.

En este ejemplo, se muestra el patrón de Generador en el que los diferentes vehículos se ensamblan paso a paso. La tienda utiliza VehicleBuilders para construir una variedad de vehículos en una serie de pasos secuenciales.

```
using System;
using System.Collections.Generic;

namespace GangOfFour.Builder
{
  /// <summary>
  /// MainApp startup class for Real-World
  /// Builder Design Pattern.
  /// </summary>
  public class MainApp
  {
    /// <summary>
    /// Entry point into console application.
    /// </summary>
    public static void Main()
    {
      VehicleBuilder builder;

      // Create shop with vehicle builders
      Shop shop = new Shop();

      // Construct and display vehicles
      builder = new ScooterBuilder();
      shop.Construct(builder);
      builder.Vehicle.Show();

      builder = new CarBuilder();
      shop.Construct(builder);
      builder.Vehicle.Show();

      builder = new MotorCycleBuilder();
      shop.Construct(builder);
      builder.Vehicle.Show();

      // Wait for user
      Console.ReadKey();
    }
  }

  /// <summary>
  /// The 'Director' class
  /// </summary>
  class Shop
  {
    // Builder uses a complex series of steps
    public void Construct(VehicleBuilder vehicleBuilder)
    {
      vehicleBuilder.BuildFrame();
```

```
    vehicleBuilder.BuildEngine();
    vehicleBuilder.BuildWheels();
    vehicleBuilder.BuildDoors();
  }
}

/// <summary>
/// The 'Builder' abstract class
/// </summary>
abstract class VehicleBuilder
{
  protected Vehicle vehicle;

  // Gets vehicle instance
  public Vehicle Vehicle
  {
    get { return vehicle; }
  }

  // Abstract build methods
  public abstract void BuildFrame();
  public abstract void BuildEngine();
  public abstract void BuildWheels();
  public abstract void BuildDoors();
}

/// <summary>
/// The 'ConcreteBuilder1' class
/// </summary>
class MotorCycleBuilder : VehicleBuilder
{
  public MotorCycleBuilder()
  {
    vehicle = new Vehicle("MotorCycle");
  }

  public override void BuildFrame()
  {
    vehicle["frame"] = "MotorCycle Frame";
  }

  public override void BuildEngine()
  {
    vehicle["engine"] = "500 cc";
  }

  public override void BuildWheels()
  {
    vehicle["wheels"] = "2";
  }

  public override void BuildDoors()
  {
    vehicle["doors"] = "0";
  }
}


/// <summary>
/// The 'ConcreteBuilder2' class
/// </summary>
```

```
class CarBuilder : VehicleBuilder
{
  public CarBuilder()
  {
    vehicle = new Vehicle("Car");
  }

  public override void BuildFrame()
  {
    vehicle["frame"] = "Car Frame";
  }

  public override void BuildEngine()
  {
    vehicle["engine"] = "2500 cc";
  }

  public override void BuildWheels()
  {
    vehicle["wheels"] = "4";
  }

  public override void BuildDoors()
  {
    vehicle["doors"] = "4";
  }
}

/// <summary>
/// The 'ConcreteBuilder3' class
/// </summary>
class ScooterBuilder : VehicleBuilder
{
  public ScooterBuilder()
  {
    vehicle = new Vehicle("Scooter");
  }

  public override void BuildFrame()
  {
    vehicle["frame"] = "Scooter Frame";
  }

  public override void BuildEngine()
  {
    vehicle["engine"] = "50 cc";
  }

  public override void BuildWheels()
  {
    vehicle["wheels"] = "2";
  }

  public override void BuildDoors()
  {
    vehicle["doors"] = "0";
  }
}

/// <summary>
/// The 'Product' class
```

```
  /// </summary>
  class Vehicle
  {
    private string _vehicleType;
    private Dictionary<string,string> _parts =
      new Dictionary<string,string>();

    // Constructor
    public Vehicle(string vehicleType)
    {
      this._vehicleType = vehicleType;
    }

    // Indexer
    public string this[string key]
    {
      get { return _parts[key]; }
      set { _parts[key] = value; }
    }

    public void Show()
    {
      Console.WriteLine("\n----------------------------");
      Console.WriteLine("Vehicle Type: {0}", _vehicleType);
      Console.WriteLine(" Frame : {0}", _parts["frame"]);
      Console.WriteLine(" Engine : {0}", _parts["engine"]);
      Console.WriteLine(" #Wheels: {0}", _parts["wheels"]);
      Console.WriteLine(" #Doors : {0}", _parts["doors"]);
    }
  }
}
```

Salida

---

Tipo de vehículo: Marco de scooter: Marco de scooter
Motor: ninguno
# Ruedas: 2
#Puertas: 0

---

Tipo de vehículo: Coche
Marco: Marco del coche
Motor: 2500 cc
# Ruedas: 4
#Puertas: 4

---

Tipo de vehículo: MotorCycle
Cuadro: Cuadro MotorCycle
Motor: 500 cc
# Ruedas: 2
#Puertas: 0

## Patrón prototipo

Especifique el tipo de objetos para crear utilizando una instancia prototípica y cree nuevos objetos copiando este prototipo.

En este ejemplo, se muestra el patrón de prototipo en el que se crean nuevos objetos de color al copiar colores preexistentes definidos por el usuario del mismo tipo.

```
using System;
using System.Collections.Generic;

namespace GangOfFour.Prototype
{
  /// <summary>
  /// MainApp startup class for Real-World
  /// Prototype Design Pattern.
  /// </summary>
  class MainApp
  {
    /// <summary>
    /// Entry point into console application.
    /// </summary>
    static void Main()
    {
      ColorManager colormanager = new ColorManager();

      // Initialize with standard colors
      colormanager["red"] = new Color(255, 0, 0);
      colormanager["green"] = new Color(0, 255, 0);
      colormanager["blue"] = new Color(0, 0, 255);

      // User adds personalized colors
      colormanager["angry"] = new Color(255, 54, 0);
      colormanager["peace"] = new Color(128, 211, 128);
      colormanager["flame"] = new Color(211, 34, 20);

      // User clones selected colors
      Color color1 = colormanager["red"].Clone() as Color;
      Color color2 = colormanager["peace"].Clone() as Color;
      Color color3 = colormanager["flame"].Clone() as Color;

      // Wait for user
      Console.ReadKey();
    }
  }

  /// <summary>
  /// The 'Prototype' abstract class
  /// </summary>
  abstract class ColorPrototype
  {
    public abstract ColorPrototype Clone();
  }

  /// <summary>
  /// The 'ConcretePrototype' class
  /// </summary>
  class Color : ColorPrototype
  {
    private int _red;
    private int _green;
    private int _blue;
```

```
    // Constructor
    public Color(int red, int green, int blue)
    {
      this._red = red;
      this._green = green;
      this._blue = blue;
    }

    // Create a shallow copy
    public override ColorPrototype Clone()
    {
      Console.WriteLine(
        "Cloning color RGB: {0,3},{1,3},{2,3}",
        _red, _green, _blue);

      return this.MemberwiseClone() as ColorPrototype;
    }
  }

  /// <summary>
  /// Prototype manager
  /// </summary>
  class ColorManager
  {
    private Dictionary<string, ColorPrototype> _colors =
      new Dictionary<string, ColorPrototype>();

    // Indexer
    public ColorPrototype this[string key]
    {
      get { return _colors[key]; }
      set { _colors.Add(key, value); }
    }
  }
}
```

Salida:

Color de clonación RGB: 255, 0, 0

Color de clonación RGB: 128,211,128

Color de clonación RGB: 211, 34, 20

## Patrón abstracto de la fábrica

Proporcionar una interfaz para crear familias de objetos relacionados o dependientes sin especificar sus clases concretas.

En este ejemplo, se demuestra la creación de diferentes mundos animales para un juego de computadora que utiliza diferentes fábricas. Aunque los animales creados por las fábricas del Continente son diferentes, las interacciones entre los animales siguen siendo las mismas.

```
using System;
```

```
namespace GangOfFour.AbstractFactory
{
  /// <summary>
  /// MainApp startup class for Real-World
  /// Abstract Factory Design Pattern.
  /// </summary>
  class MainApp
  {
    /// <summary>
    /// Entry point into console application.
    /// </summary>
    public static void Main()
    {
      // Create and run the African animal world
      ContinentFactory africa = new AfricaFactory();
      AnimalWorld world = new AnimalWorld(africa);
      world.RunFoodChain();

      // Create and run the American animal world
      ContinentFactory america = new AmericaFactory();
      world = new AnimalWorld(america);
      world.RunFoodChain();

      // Wait for user input
      Console.ReadKey();
    }
  }


  /// <summary>
  /// The 'AbstractFactory' abstract class
  /// </summary>
  abstract class ContinentFactory
  {
    public abstract Herbivore CreateHerbivore();
    public abstract Carnivore CreateCarnivore();
  }

  /// <summary>
  /// The 'ConcreteFactory1' class
  /// </summary>
  class AfricaFactory : ContinentFactory
  {
    public override Herbivore CreateHerbivore()
    {
      return new Wildebeest();
    }
    public override Carnivore CreateCarnivore()
    {
      return new Lion();
    }
  }

  /// <summary>
  /// The 'ConcreteFactory2' class
  /// </summary>
  class AmericaFactory : ContinentFactory
  {
    public override Herbivore CreateHerbivore()
    {
      return new Bison();
```

```
  }
  public override Carnivore CreateCarnivore()
  {
    return new Wolf();
  }
}

/// <summary>
/// The 'AbstractProductA' abstract class
/// </summary>
abstract class Herbivore
{
}

/// <summary>
/// The 'AbstractProductB' abstract class
/// </summary>
abstract class Carnivore
{
  public abstract void Eat(Herbivore h);
}

/// <summary>
/// The 'ProductA1' class
/// </summary>
class Wildebeest : Herbivore
{
}

/// <summary>
/// The 'ProductB1' class
/// </summary>
class Lion : Carnivore
{
  public override void Eat(Herbivore h)
  {
    // Eat Wildebeest
    Console.WriteLine(this.GetType().Name +
      " eats " + h.GetType().Name);
  }
}

/// <summary>
/// The 'ProductA2' class
/// </summary>
class Bison : Herbivore
{
}

/// <summary>
/// The 'ProductB2' class
/// </summary>
class Wolf : Carnivore
{
  public override void Eat(Herbivore h)
  {
    // Eat Bison
    Console.WriteLine(this.GetType().Name +
      " eats " + h.GetType().Name);
  }
}
```

```
  /// <summary>
  /// The 'Client' class
  /// </summary>
  class AnimalWorld
  {
    private Herbivore _herbivore;
    private Carnivore _carnivore;

    // Constructor
    public AnimalWorld(ContinentFactory factory)
    {
      _carnivore = factory.CreateCarnivore();
      _herbivore = factory.CreateHerbivore();
    }

    public void RunFoodChain()
    {
      _carnivore.Eat(_herbivore);
    }
  }
}
```

Salida:

León come ñu

Lobo come bisonte

Lea Patrones de diseño creacional en línea: https://riptutorial.com/es/csharp/topic/6654/patrones-de-diseno-creacional

# Capítulo 130: Patrones de diseño estructural

## Introducción

Los patrones de diseño estructural son patrones que describen cómo los objetos y las clases se pueden combinar y forman una estructura grande y que facilitan el diseño al identificar una forma sencilla de establecer relaciones entre entidades. Hay siete patrones estructurales descritos. Son los siguientes: Adaptador, Puente, Compuesto, Decorador, Fachada, Peso mosca y Proxy

## Examples

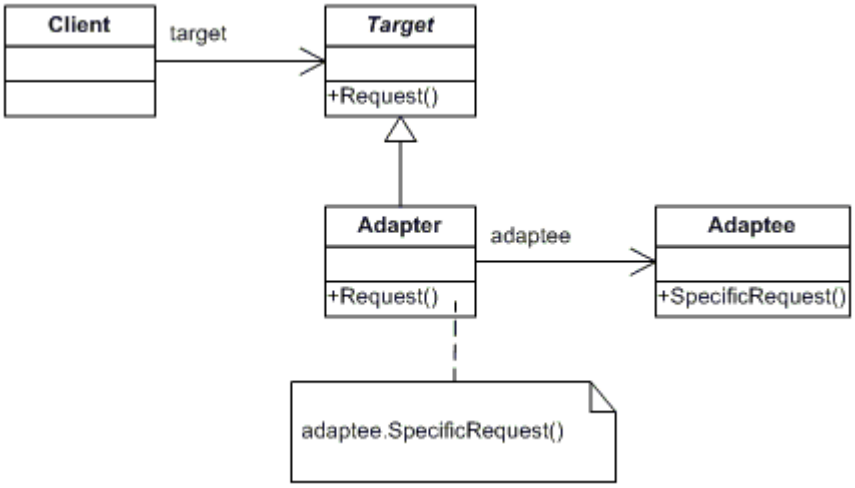
### Patrón de diseño del adaptador

> **"Adaptador"** como su nombre indica es el objeto que permite que dos interfaces incompatibles entre sí se comuniquen entre sí.
>
> **Por ejemplo:** si compra un Iphone 8 (o cualquier otro producto de Apple), necesitará muchos adaptadores. Debido a que la interfaz predeterminada no es compatible con audio jac o USB. Con estos adaptadores puede usar auriculares con cables o puede usar un cable Ethernet normal. Así que *"dos interfaces mutuamente incompatibles se comunican entre sí"* .
>
> **En términos técnicos, esto significa:** Convertir la interfaz de una clase en otra interfaz que los clientes esperan. El adaptador permite que las clases trabajen juntas y que no podrían hacerlo de otra manera debido a interfaces incompatibles. Las clases y objetos que participan en este patrón son:

**El patrón adaptador sale por 4 elementos.**

1. **ITarget:** esta es la interfaz que utiliza el cliente para lograr la funcionalidad.
2. **Adaptee:** esta es la funcionalidad que el cliente desea pero su interfaz no es compatible con el cliente.
3. **Cliente:** esta es la clase que desea lograr alguna funcionalidad mediante el uso del código adaptee.
4. **Adaptador:** esta es la clase que implementaría ITarget y llamaría al código Adaptee al que el cliente desea llamar.

**UML**




**Primer ejemplo de código (ejemplo teórico)** .

```
public interface ITarget
{
    void MethodA();
}

public class Adaptee
{
    public void MethodB()
    {
        Console.WriteLine("MethodB() is called");
    }
}

public class Client
{
    private ITarget target;

    public Client(ITarget target)
    {
        this.target = target;
    }

    public void MakeRequest()
    {
        target.MethodA();
    }
}

public class Adapter : Adaptee, ITarget
{
    public void MethodA()
    {
        MethodB();
    }
}
```

**Segundo ejemplo de código (Real World Imlementation)**

```
/// <summary>
///  Interface: This is the interface which is used by the client to achieve functionality.
/// </summary>
```

```
public interface ITarget
{
    List<string> GetEmployeeList();
}

/// <summary>
/// Adaptee: This is the functionality which the client desires but its interface is not
compatible with the client.
/// </summary>
public class CompanyEmplyees
{
    public string[][] GetEmployees()
    {
        string[][] employees = new string[4][];

        employees[0] = new string[] { "100", "Deepak", "Team Leader" };
        employees[1] = new string[] { "101", "Rohit", "Developer" };
        employees[2] = new string[] { "102", "Gautam", "Developer" };
        employees[3] = new string[] { "103", "Dev", "Tester" };

        return employees;
    }
}

/// <summary>
/// Client: This is the class which wants to achieve some functionality by using the adaptee’s
code (list of employees).
/// </summary>
public class ThirdPartyBillingSystem
{
    /*
     * This class is from a thirt party and you do'n have any control over it.
     * But it requires a Emplyee list to do its work
     */

    private ITarget employeeSource;

    public ThirdPartyBillingSystem(ITarget employeeSource)
    {
        this.employeeSource = employeeSource;
    }

    public void ShowEmployeeList()
    {
        // call the clietn list in the interface
        List<string> employee = employeeSource.GetEmployeeList();

        Console.WriteLine("######### Employee List ##########");
        foreach (var item in employee)
        {
            Console.Write(item);
        }

    }
}

/// <summary>
/// Adapter: This is the class which would implement ITarget and would call the Adaptee code
which the client wants to call.
/// </summary>
public class EmployeeAdapter : CompanyEmplyees, ITarget
```

```
{
    public List<string> GetEmployeeList()
    {
        List<string> employeeList = new List<string>();
        string[][] employees = GetEmployees();
        foreach (string[] employee in employees)
        {
            employeeList.Add(employee[0]);
            employeeList.Add(",");
            employeeList.Add(employee[1]);
            employeeList.Add(",");
            employeeList.Add(employee[2]);
            employeeList.Add("\n");
        }

        return employeeList;
    }
}

///
/// Demo
///
class Programs
{
    static void Main(string[] args)
    {
        ITarget Itarget = new EmployeeAdapter();
        ThirdPartyBillingSystem client = new ThirdPartyBillingSystem(Itarget);
        client.ShowEmployeeList();
        Console.ReadKey();
    }
}
```

### Cuándo usar

- Permitir que un sistema use clases de otro sistema que sea incompatible con él.
- Permitir la comunicación entre sistemas nuevos y ya existentes que son independientes entre sí.
- Ado.Net SqlAdapter, OracleAdapter, MySqlAdapter son el mejor ejemplo de patrón de adaptador.

Lea Patrones de diseño estructural en línea: https://riptutorial.com/es/csharp/topic/9764/patrones-de-diseno-estructural

# Capítulo 131: Plataforma de compilación .NET (Roslyn)

## Examples

### Crear espacio de trabajo desde el proyecto MSBuild

Primero obtenga el nuget `Microsoft.CodeAnalysis.CSharp.Workspaces` antes de continuar.

```
var workspace = Microsoft.CodeAnalysis.MSBuild.MSBuildWorkspace.Create();
var project = await workspace.OpenProjectAsync(projectFilePath);
var compilation = await project.GetCompilationAsync();

foreach (var diagnostic in compilation.GetDiagnostics()
    .Where(d => d.Severity == Microsoft.CodeAnalysis.DiagnosticSeverity.Error))
{
    Console.WriteLine(diagnostic);
}
```

Para cargar el código existente en el área de trabajo, compile e informe los errores. Posteriormente el código se ubicará en la memoria. A partir de aquí, tanto el lado sintáctico como el semántico estarán disponibles para trabajar.

### Árbol de sintaxis

Un **árbol de sintaxis** es una estructura de datos inmutables que representa el programa como un árbol de nombres, comandos y marcas (como se configuró previamente en el editor).

Por ejemplo, supongamos que se ha configurado una instancia de `Microsoft.CodeAnalysis.Compilation` llamada `compilation` . Hay varias formas de enumerar los nombres de cada variable declarada en el código cargado. Para hacerlo de forma ingenua, tome todas las piezas de sintaxis en cada documento (el método de Nodos `DescendantNodes` ) y use Linq para seleccionar los nodos que describen la declaración de variables:

```
foreach (var syntaxTree in compilation.SyntaxTrees)
{
    var root = await syntaxTree.GetRootAsync();
    var declaredIdentifiers = root.DescendantNodes()
        .Where(an => an is VariableDeclaratorSyntax)
        .Cast<VariableDeclaratorSyntax>()
        .Select(vd => vd.Identifier);

    foreach (var di in declaredIdentifiers)
    {
        Console.WriteLine(di);
    }
}
```

Cada tipo de construcción C # con un tipo correspondiente existirá en el árbol de sintaxis. Para

encontrar rápidamente tipos específicos, use la ventana `Syntax Visualizer` de Visual Studio. Esto interpretará el documento abierto actual como un árbol de sintaxis de Roslyn.

## Modelo semantico

Un **modelo semántico** ofrece un nivel más profundo de interpretación y conocimiento del código en comparación con un árbol de sintaxis. Donde los árboles de sintaxis pueden decir los nombres de las variables, los modelos semánticos también proporcionan el tipo y todas las referencias. Los árboles de sintaxis notan las llamadas al método, pero los modelos semánticos dan referencias a la ubicación precisa en que se declara el método (después de que se haya aplicado la resolución de sobrecarga).

```
var workspace = Microsoft.CodeAnalysis.MSBuild.MSBuildWorkspace.Create();
var sln = await workspace.OpenSolutionAsync(solutionFilePath);
var project = sln.Projects.First();
var compilation = await project.GetCompilationAsync();

foreach (var syntaxTree in compilation.SyntaxTrees)
{
    var root = await syntaxTree.GetRootAsync();

    var declaredIdentifiers = root.DescendantNodes()
        .Where(an => an is VariableDeclaratorSyntax)
        .Cast<VariableDeclaratorSyntax>();

    foreach (var di in declaredIdentifiers)
    {
        Console.WriteLine(di.Identifier);
        // => "root"

        var variableSymbol = compilation
            .GetSemanticModel(syntaxTree)
            .GetDeclaredSymbol(di) as ILocalSymbol;

        Console.WriteLine(variableSymbol.Type);
        // => "Microsoft.CodeAnalysis.SyntaxNode"

        var references = await SymbolFinder.FindReferencesAsync(variableSymbol, sln);
        foreach (var reference in references)
        {
            foreach (var loc in reference.Locations)
            {
                Console.WriteLine(loc.Location.SourceSpan);
                // => "[1375..1379)"
            }
        }
    }
}
```

Esto genera una lista de variables locales utilizando un árbol de sintaxis. Luego consulta el modelo semántico para obtener el nombre completo del tipo y encontrar todas las referencias de cada variable.

Lea Plataforma de compilación .NET (Roslyn) en línea:
https://riptutorial.com/es/csharp/topic/4886/plataforma-de-compilacion--net--roslyn-

# Capítulo 132: Polimorfismo

## Examples

### Otro ejemplo de polimorfismo

El polimorfismo es uno de los pilares de la POO. Poly deriva de un término griego que significa "formas múltiples".

A continuación se muestra un ejemplo que exhibe polimorfismo. La clase `Vehicle` toma múltiples formas como clase base.

Las clases derivadas `Ducati` y `Lamborghini` heredan del `Vehicle` y anulan el método `Display()` la clase base, para mostrar sus propias `NumberOfWheels` .

```
public class Vehicle
{
    protected int NumberOfWheels { get; set; } = 0;
    public Vehicle()
    {
    }

    public virtual void Display()
    {
        Console.WriteLine($"The number of wheels for the {nameof(Vehicle)} is
{NumberOfWheels}");
    }
}

public class Ducati : Vehicle
{
    public Ducati()
    {
        NoOfWheels = 2;
    }

    public override void Display()
    {
        Console.WriteLine($"The number of wheels for the {nameof(Ducati)} is
{NumberOfWheels}");
    }
}

public class Lamborghini : Vehicle
{
    public Lamborghini()
    {
        NoOfWheels = 4;
    }

    public override void Display()
    {
        Console.WriteLine($"The number of wheels for the {nameof(Lamborghini)} is
{NumberOfWheels}");
    }
```

```
}
```

A continuación se muestra el fragmento de código donde se exhibe el polimorfismo. El objeto se crea para el Tipo base `Vehicle` utilizando un `vehicle` variable en la Línea 1. Llama al método de clase base `Display()` en la Línea 2 y muestra la salida como se muestra.

```
static void Main(string[] args)
{
   Vehicle vehicle = new Vehicle();     //Line 1
   vehicle.Display();                   //Line 2
   vehicle = new Ducati();              //Line 3
   vehicle.Display();                   //Line 4
   vehicle = new Lamborghini();         //Line 5
   vehicle.Display();                   //Line 6
}
```

En la Línea 3, el objeto del `vehicle` apunta a la clase derivada `Ducati` y llama a su método `Display()` , que muestra la salida como se muestra. Aquí viene el comportamiento polimórfico, aunque el objeto `vehicle` es del tipo `Vehicle` , se llama al método clase derivada `Display()` como el tipo de `Ducati` anula la clase base `Display()` método, ya que el `vehicle` objeto se señaló hacia `Ducati` .

La misma explicación es aplicable cuando invoca el método `Display()` del tipo `Lamborghini` .

La salida se muestra a continuación

```
The number of wheels for the Vehicle is 0        // Line 2
The number of wheels for the Ducati is 2         // Line 4
The number of wheels for the Lamborghini is 4    // Line 6
```

### Tipos de polimorfismo

El polimorfismo significa que una operación también se puede aplicar a valores de algunos otros tipos.

Hay múltiples tipos de polimorfismo:

- **Polimorfismo ad hoc:**
  contiene la `function overloading` . El objetivo es que un método se pueda usar con diferentes tipos sin la necesidad de ser genérico.
- **Polimorfismo paramétrico:**
  Es el uso de tipos genéricos. Ver genéricos
- **Subtitulado:**
  tiene el objetivo heredado de una clase para generalizar una funcionalidad similar

---

# Polimorfismo ad hoc

El objetivo del `Ad hoc polymorphism` es crear un método, que pueda ser llamado por diferentes tipos

de datos sin necesidad de una conversión de tipo en la llamada a la función o genéricos. El siguiente método (s) `sumInt(par1, par2)` se puede llamar con tipos de datos diferentes y tiene para cada combinación de tipos una implementación propia:

```
public static int sumInt( int a, int b)
{
    return a + b;
}

public static int sumInt( string a, string b)
{
    int _a, _b;

    if(!Int32.TryParse(a, out _a))
        _a = 0;

    if(!Int32.TryParse(b, out _b))
        _b = 0;

    return _a + _b;
}

public static int sumInt(string a, int b)
{
    int _a;

    if(!Int32.TryParse(a, out _a))
        _a = 0;

    return _a + b;
}

public static int sumInt(int a, string b)
{
    return sumInt(b,a);
}
```

Aquí hay un ejemplo de llamada:

```
public static void Main()
{
    Console.WriteLine(sumInt( 1 , 2 ));  //  3
    Console.WriteLine(sumInt("3","4"));  //  7
    Console.WriteLine(sumInt("5", 6 ));  // 11
    Console.WriteLine(sumInt( 7 ,"8"));  // 15
}
```

---

# Subtitulación

Subtipo es el uso de heredar de una clase base para generalizar un comportamiento similar:

```
public interface Car{
    void refuel();
}
```

```
public class NormalCar : Car
{
    public void refuel()
    {
        Console.WriteLine("Refueling with petrol");
    }
}

public class ElectricCar : Car
{
    public void refuel()
    {
        Console.WriteLine("Charging battery");
    }
}
```

Ambas clases, `NormalCar` y `ElectricCar` ahora tienen un método para repostar, pero su propia implementación. Aquí hay un ejemplo:

```
public static void Main()
{
    List<Car> cars = new List<Car>(){
        new NormalCar(),
        new ElectricCar()
    };

    cars.ForEach(x => x.refuel());
}
```

La salida será la siguiente:

> Repostar con gasolina
> Bateria cargando

Lea Polimorfismo en línea: https://riptutorial.com/es/csharp/topic/1589/polimorfismo

# Capítulo 133: Programación Funcional

## Examples

### Func y Acción

**Func** proporciona un soporte para funciones anónimas parametrizadas. Los tipos principales son las entradas y el último tipo es siempre el valor de retorno.

```
// square a number.
Func<double, double> square = (x) => { return x * x; };

// get the square root.
// note how the signature matches the built in method.
Func<double, double> squareroot = Math.Sqrt;

// provide your workings.
Func<double, double, string> workings = (x, y) =>
    string.Format("The square of {0} is {1}.", x, square(y))
```

**Los** objetos de **acción** son como métodos de vacío, por lo que solo tienen un tipo de entrada. No se coloca ningún resultado en la pila de evaluación.

```
// right-angled triangle.
class Triangle
{
    public double a;
    public double b;
    public double h;
}

// Pythagorean theorem.
Action<Triangle> pythagoras = (x) =>
    x.h = squareroot(square(x.a) + square(x.b));

Triangle t = new Triangle { a = 3, b = 4 };
pythagoras(t);
Console.WriteLine(t.h); // 5.
```

### Inmutabilidad

La inmutabilidad es común en la programación funcional y rara en la programación orientada a objetos.

Cree, por ejemplo, un tipo de dirección con estado mutable:

```
public class Address ()
{
    public string Line1 { get; set; }
    public string Line2 { get; set; }
    public string City  { get; set; }
```

```
}
```

Cualquier pieza de código podría alterar cualquier propiedad en el objeto anterior.

Ahora crea el tipo de dirección inmutable:

```
public class Address ()
{
    public readonly string Line1;
    public readonly string Line2;
    public readonly string City;

    public Address(string line1, string line2, string city)
    {
        Line1 = line1;
        Line2 = line2;
        City  = city;
    }
}
```

Tenga en cuenta que tener colecciones de solo lectura no respeta la inmutabilidad. Por ejemplo,

```
public class Classroom
{
    public readonly List<Student> Students;

    public Classroom(List<Student> students)
    {
        Students = students;
    }
}
```

no es inmutable, ya que el usuario del objeto puede alterar la colección (agregar o quitar elementos de ella). Para hacerlo inmutable, uno tiene que usar una interfaz como IEnumerable, que no expone métodos para agregar, o hacer que sea ReadOnlyCollection.

```
public class Classroom
{
    public readonly ReadOnlyCollection<Student> Students;

    public Classroom(ReadOnlyCollection<Student> students)
    {
        Students = students;
    }
}

List<Students> list = new List<Student>();
// add students
Classroom c = new Classroom(list.AsReadOnly());
```

Con el objeto inmutable tenemos los siguientes beneficios:

- Estará en un estado conocido (otro código no puede cambiarlo).
- Es hilo seguro.
- El constructor ofrece un solo lugar para la validación.

- Saber que el objeto no puede ser alterado hace que el código sea más fácil de entender.

## Evitar referencias nulas

Los desarrolladores de C # obtienen muchas excepciones de referencia nulas con las que lidiar. Los desarrolladores de F # no lo hacen porque tienen el tipo de opción. Un tipo Opción <> (algunos prefieren Maybe <> como nombre) proporciona un tipo de retorno Algunos y Ninguno. Hace explícito que un método puede estar a punto de devolver un registro nulo.

Por ejemplo, no puede leer lo siguiente y saber si tendrá que lidiar con un valor nulo.

```
var user = _repository.GetUser(id);
```

Si conoce el posible nulo, puede introducir algún código repetitivo para resolverlo.

```
var username = user != null ? user.Name : string.Empty;
```

¿Qué pasa si tenemos una opción <> devuelta en su lugar?

```
Option<User> maybeUser = _repository.GetUser(id);
```

Ahora el código hace explícito que es posible que se nos devuelva un registro Ninguno y que se requiera el código repetitivo para verificar Algunos o Ninguno:

```
var username = maybeUser.HasValue ? maybeUser.Value.Name : string.Empty;
```

El siguiente método muestra cómo devolver una Opción <>

```
public Option<User> GetUser(int id)
{
    var users = new List<User>
    {
        new User { Id = 1, Name = "Joe Bloggs" },
        new User { Id = 2, Name = "John Smith" }
    };

    var user = users.FirstOrDefault(user => user.Id == id);

    return user != null ? new Option<User>(user) : new Option<User>();
}
```

Aquí hay una implementación mínima de la Opción <>.

```
public struct Option<T>
{
    private readonly T _value;

    public T Value
    {
        get
        {
```

```
            if (!HasValue)
                throw new InvalidOperationException();

            return _value;
        }
    }

    public bool HasValue
    {
        get { return _value != null; }
    }

    public Option(T value)
    {
        _value = value;
    }

    public static implicit operator Option<T>(T value)
    {
        return new Option<T>(value);
    }
}
```

Para demostrar lo anterior, AvoidNull.csx puede ejecutarse con C # REPL.

Como se dijo, esta es una implementación mínima. Una búsqueda de "Tal vez" paquetes NuGet mostrará una serie de buenas bibliotecas.

### Funciones de orden superior

Una función de orden superior es aquella que toma otra función como argumento o devuelve una función (o ambas).

Esto se hace comúnmente con lambdas, por ejemplo, al pasar un predicado a una cláusula LINQ Where:

```
var results = data.Where(p => p.Items == 0);
```

La cláusula Where () podría recibir muchos predicados diferentes, lo que le da una flexibilidad considerable.

Al implementar el patrón de diseño de la Estrategia, también se ve pasar un método a otro. Por ejemplo, se podrían elegir varios métodos de clasificación y pasar a un método de Clasificación en un objeto, dependiendo de los requisitos en tiempo de ejecución.

### Colecciones inmutables

El paquete `System.Collections.Immutable` NuGet proporciona clases de colección inmutables.

# Creación y adición de elementos

```
var stack = ImmutableStack.Create<int>();
var stack2 = stack.Push(1); // stack is still empty, stack2 contains 1
var stack3 = stack.Push(2); // stack2 still contains only one, stack3 has 2, 1
```

# Creando usando el constructor

Ciertas colecciones inmutables tienen una clase interna de `Builder` que se puede usar para construir grandes instancias inmutables a bajo costo:

```
var builder = ImmutableList.CreateBuilder<int>(); // returns ImmutableList.Builder
builder.Add(1);
builder.Add(2);
var list = builder.ToImmutable();
```

# Creando desde un IEnumerable existente

```
var numbers = Enumerable.Range(1, 5);
var list = ImmutableList.CreateRange<int>(numbers);
```

Lista de todos los tipos de colecciones inmutables:

- `System.Collections.Immutable.ImmutableArray<T>`
- `System.Collections.Immutable.ImmutableDictionary<TKey,TValue>`
- `System.Collections.Immutable.ImmutableHashSet<T>`
- `System.Collections.Immutable.ImmutableList<T>`
- `System.Collections.Immutable.ImmutableQueue<T>`
- `System.Collections.Immutable.ImmutableSortedDictionary<TKey,TValue>`
- `System.Collections.Immutable.ImmutableSortedSet<T>`
- `System.Collections.Immutable.ImmutableStack<T>`

Lea Programación Funcional en línea: https://riptutorial.com/es/csharp/topic/2564/programacion-funcional

# Capítulo 134: Programación Orientada a Objetos En C #

## Introducción

Este tema trata de decirnos cómo podemos escribir programas basados en el enfoque OOP. Pero no intentamos enseñar el paradigma de programación orientada a objetos. Cubriremos los siguientes temas: clases, propiedades, herencia, polimorfismo, interfaces, etc.

## Examples

### Clases:

El esqueleto de declarar clase es:

<>: Requerido

[]:Opcional

```
[private/public/protected/internal] class <Desired Class Name> [:[Inherited
class][,][[Interface Name 1],[Interface Name 2],...]
{
    //Your code
}
```

No se preocupe si no puede entender la sintaxis completa. Nos familiarizaremos con toda la parte de eso. Para el primer ejemplo, considere la siguiente clase:

```
class MyClass
{
    int i = 100;
    public void getMyValue()
    {
        Console.WriteLine(this.i);//Will print number 100 in output
    }
}
```

en esta clase creamos la variable `i` con el tipo `int` y con los modificadores de acceso privados predeterminados y el método `getMyValue()` con modificadores de acceso público.

Lea Programación Orientada a Objetos En C # en línea:
https://riptutorial.com/es/csharp/topic/9856/programacion-orientada-a-objetos-en-c-sharp

# Capítulo 135: Propiedades

## Observaciones

Las propiedades combinan el almacenamiento de datos de clase de los campos con la accesibilidad de los métodos. A veces puede ser difícil decidir si usar una propiedad, una propiedad que haga referencia a un campo o un método que haga referencia a un campo. Como una regla de oro:

- Las propiedades deben usarse sin un campo interno si solo obtienen y / o establecen valores; sin ninguna otra lógica ocurriendo. En tales casos, agregar un campo interno sería agregar código sin beneficio.
- Las propiedades deben usarse con campos internos cuando necesite manipular o validar los datos. Un ejemplo puede ser quitar los espacios iniciales y finales de las cadenas o asegurarse de que una fecha no esté en el pasado.

Con respecto a Métodos vs Propiedades, donde puede recuperar ( `get` ) y actualizar ( `set` ) un valor, una propiedad es la mejor opción. Además, .Net proporciona una gran cantidad de funciones que hacen uso de la estructura de una clase; por ejemplo, al agregar una cuadrícula a un formulario, .Net mostrará de manera predeterminada todas las propiedades de la clase en ese formulario; por lo tanto, para hacer un mejor uso de tales convenciones, planee usar propiedades cuando este comportamiento sea típicamente deseable, y métodos donde preferiría que los tipos no se agreguen automáticamente.

## Examples

### Varias propiedades en contexto

```
public class Person
{
    //Id property can be read by other classes, but only set by the Person class
    public int Id {get; private set;}
    //Name property can be retrieved or assigned
    public string Name {get; set;}

    private DateTime dob;
    //Date of Birth property is stored in a private variable, but retrieved or assigned
through the public property.
    public DateTime DOB
    {
        get { return this.dob; }
        set { this.dob = value; }
    }
    //Age property can only be retrieved; it's value is derived from the date of birth
    public int Age
    {
        get
        {
```

```
            int offset = HasHadBirthdayThisYear() ? 0 : -1;
            return DateTime.UtcNow.Year - this.dob.Year + offset;
        }
    }

    //this is not a property but a method; though it could be rewritten as a property if
desired.
    private bool HasHadBirthdayThisYear()
    {
        bool hasHadBirthdayThisYear = true;
        DateTime today = DateTime.UtcNow;
        if (today.Month > this.dob.Month)
        {
            hasHadBirthdayThisYear = true;
        }
        else
        {
            if (today.Month == this.dob.Month)
            {
                hasHadBirthdayThisYear = today.Day > this.dob.Day;
            }
            else
            {
                hasHadBirthdayThisYear = false;
            }
        }
        return hasHadBirthdayThisYear;
    }
}
```

## Obtener público

Getters se utilizan para exponer los valores de las clases.

```
string name;
public string Name
{
    get { return this.name; }
}
```

## Conjunto público

Los setters se utilizan para asignar valores a las propiedades.

```
string name;
public string Name
{
    set { this.name = value; }
}
```

## Acceso a las propiedades

```
class Program
{
    public static void Main(string[] args)
```

```
    {
        Person aPerson = new Person("Ann Xena Sample", new DateTime(1984, 10, 22));
        //example of accessing properties (Id, Name & DOB)
        Console.WriteLine("Id is:  \t{0}\nName is:\t'{1}'.\nDOB is: \t{2:yyyy-MM-dd}.\nAge is:
\t{3}", aPerson.Id, aPerson.Name, aPerson.DOB, aPerson.GetAgeInYears());
        //example of setting properties

        aPerson.Name = "   Hans Trimmer  ";
        aPerson.DOB = new DateTime(1961, 11, 11);
        //aPerson.Id = 5; //this won't compile as Id's SET method is private; so only
accessible within the Person class.
        //aPerson.DOB = DateTime.UtcNow.AddYears(1); //this would throw a runtime error as
there's validation to ensure the DOB is in past.

        //see how our changes above take effect; note that the Name has been trimmed
        Console.WriteLine("Id is:  \t{0}\nName is:\t'{1}'.\nDOB is: \t{2:yyyy-MM-dd}.\nAge is:
\t{3}", aPerson.Id, aPerson.Name, aPerson.DOB, aPerson.GetAgeInYears());

        Console.WriteLine("Press any key to continue");
        Console.Read();
    }
}

public class Person
{
    private static int nextId = 0;
    private string name;
    private DateTime dob; //dates are held in UTC; i.e. we disregard timezones
    public Person(string name, DateTime dob)
    {
        this.Id = ++Person.nextId;
        this.Name = name;
        this.DOB = dob;
    }
    public int Id
    {
        get;
        private set;
    }
    public string Name
    {
        get { return this.name; }
        set
        {
            if (string.IsNullOrWhiteSpace(value)) throw new InvalidNameException(value);
            this.name = value.Trim();
        }
    }
    public DateTime DOB
    {
        get { return this.dob; }
        set
        {
            if (value < DateTime.UtcNow.AddYears(-200) || value > DateTime.UtcNow) throw new
InvalidDobException(value);
            this.dob = value;
        }
    }
    public int GetAgeInYears()
    {
        DateTime today = DateTime.UtcNow;
```

```
        int offset = HasHadBirthdayThisYear() ? 0 : -1;
        return today.Year - this.dob.Year + offset;
    }
    private bool HasHadBirthdayThisYear()
    {
        bool hasHadBirthdayThisYear = true;
        DateTime today = DateTime.UtcNow;
        if (today.Month > this.dob.Month)
        {
            hasHadBirthdayThisYear = true;
        }
        else
        {
            if (today.Month == this.dob.Month)
            {
                hasHadBirthdayThisYear = today.Day > this.dob.Day;
            }
            else
            {
                hasHadBirthdayThisYear = false;
            }
        }
        return hasHadBirthdayThisYear;
    }
}

public class InvalidNameException : ApplicationException
{
    const string InvalidNameExceptionMessage = "'{0}' is an invalid name.";
    public InvalidNameException(string value):
base(string.Format(InvalidNameExceptionMessage,value)){}
}
public class InvalidDobException : ApplicationException
{
    const string InvalidDobExceptionMessage = "'{0:yyyy-MM-dd}' is an invalid DOB.  The date
must not be in the future, or over 200 years in the past.";
    public InvalidDobException(DateTime value):
base(string.Format(InvalidDobExceptionMessage,value)){}
}
```

## Valores predeterminados para las propiedades

Se puede establecer un valor predeterminado usando Inicializadores (C # 6)

```
public class Name
{
    public string First { get; set; } = "James";
    public string Last { get; set; } = "Smith";
}
```

Si es de solo lectura puedes devolver valores como este:

```
  public class Name
  {
      public string First => "James";
      public string Last => "Smith";
  }
```

### Propiedades auto-implementadas

Las propiedades auto-implementadas se introdujeron en C # 3.
Una propiedad implementada automáticamente se declara con un getter y setter (accessors) vacíos:

```
public bool IsValid { get; set; }
```

Cuando se escribe una propiedad implementada automáticamente en su código, el compilador crea un campo anónimo privado al que solo se puede acceder a través de los accesores de la propiedad.

La declaración de propiedad implementada automáticamente anterior es equivalente a escribir este extenso código:

```
private bool _isValid;
public bool IsValid
{
    get { return _isValid; }
    set { _isValid = value; }
}
```

---

Las propiedades implementadas automáticamente no pueden tener ninguna lógica en sus accesores, por ejemplo:

```
public bool IsValid { get; set { PropertyChanged("IsValid"); } } // Invalid code
```

Sin embargo, una propiedad implementada automáticamente *puede* tener diferentes modificadores de acceso para sus accesores:

```
public bool IsValid { get; private set; }
```

C # 6 permite que las propiedades implementadas automáticamente no tengan ningún definidor (lo que lo hace inmutable, ya que su valor solo puede establecerse dentro del constructor o codificado):

```
public bool IsValid { get; }
public bool IsValid { get; } = true;
```

Para obtener más información sobre cómo inicializar las propiedades implementadas automáticamente, lea la documentación de inicializadores de propiedades automáticas .

### Propiedades de solo lectura

# Declaración

Un malentendido común, especialmente los principiantes, es que la propiedad de solo lectura está marcada con la palabra clave `readonly` . Eso no es correcto y, de hecho, lo *siguiente es un error de tiempo de compilación* :

```
public readonly string SomeProp { get; set; }
```

Una propiedad es de solo lectura cuando solo tiene un captador.

```
public string SomeProp { get; }
```

## Usando propiedades de solo lectura para crear clases inmutables

```
public Address
{
    public string ZipCode { get; }
    public string City { get; }
    public string StreetAddress { get; }

    public Address(
        string zipCode,
        string city,
        string streetAddress)
    {
        if (zipCode == null)
            throw new ArgumentNullException(nameof(zipCode));
        if (city == null)
            throw new ArgumentNullException(nameof(city));
        if (streetAddress == null)
            throw new ArgumentNullException(nameof(streetAddress));

        ZipCode = zipCode;
        City = city;
        StreetAddress = streetAddress;
    }
}
```

Lea Propiedades en línea: https://riptutorial.com/es/csharp/topic/49/propiedades

# Capítulo 136: Punteros

## Observaciones

# Punteros e `unsafe`

Debido a su naturaleza, los punteros producen código no verificable. Por lo tanto, el uso de cualquier tipo de puntero requiere un contexto `unsafe` .

El tipo `System.IntPtr` es una envoltura segura alrededor de un `void*` . Está pensado como una alternativa más conveniente para `void*` cuando no se requiere un contexto inseguro para realizar la tarea en cuestión.

# Comportamiento indefinido

Al igual que en C y C ++, el uso incorrecto de los punteros puede invocar un comportamiento indefinido, con posibles efectos secundarios como corrupción de la memoria y ejecución de código no deseado. Debido a la naturaleza no verificable de la mayoría de las operaciones de puntero, el uso correcto de los punteros es responsabilidad exclusiva del programador.

# Tipos que soportan punteros

A diferencia de C y C ++, no todos los tipos de C # tienen los tipos de punteros correspondientes. Un tipo `T` puede tener un tipo de puntero correspondiente si se aplican los dos criterios siguientes:

- `T` es un tipo de estructura o un tipo de puntero.
- `T` contiene solo miembros que satisfacen estos dos criterios recursivamente.

## Examples

### Punteros para acceso a la matriz

Este ejemplo demuestra cómo se pueden usar los punteros para el acceso tipo C a las matrices C #.

```
unsafe
{
    var buffer = new int[1024];
    fixed (int* p = &buffer[0])
    {
        for (var i = 0; i < buffer.Length; i++)
        {
            *(p + i) = i;
```

```
        }
    }
}
```

La palabra clave `unsafe` es necesaria porque el acceso del puntero no emitirá ninguna verificación de límites que normalmente se emite al acceder a las matrices de C # de forma regular.

La palabra clave `fixed` le dice al compilador de C # que emita instrucciones para fijar el objeto de una manera segura y excepcional. La fijación es necesaria para garantizar que el recolector de basura no moverá la matriz en la memoria, ya que esto invalidaría cualquier puntero que apunte dentro de la matriz.

## Aritmética de punteros

La suma y la resta en los punteros funciona de manera diferente a los enteros. Cuando un puntero aumenta o disminuye, la dirección a la que apunta aumenta o disminuye según el tamaño del tipo de referencia.

Por ejemplo, el tipo `int` (alias para `System.Int32` ) tiene un tamaño de 4. Si se puede almacenar un `int` en la dirección 0, el `int` posterior se puede almacenar en la dirección 4, y así sucesivamente. En codigo:

```
var ptr = (int*)IntPtr.Zero;
Console.WriteLine(new IntPtr(ptr)); // prints 0
ptr++;
Console.WriteLine(new IntPtr(ptr)); // prints 4
ptr++;
Console.WriteLine(new IntPtr(ptr)); // prints 8
```

De manera similar, el tipo `long` (alias para `System.Int64` ) tiene un tamaño de 8. Si se puede almacenar un `long` en la dirección 0, el `long` posterior puede almacenarse en la dirección 8, y así sucesivamente. En codigo:

```
var ptr = (long*)IntPtr.Zero;
Console.WriteLine(new IntPtr(ptr)); // prints 0
ptr++;
Console.WriteLine(new IntPtr(ptr)); // prints 8
ptr++;
Console.WriteLine(new IntPtr(ptr)); // prints 16
```

El tipo `void` es especial y los punteros `void` también son especiales y se utilizan como punteros de captura cuando el tipo no se conoce o no importa. Debido a su naturaleza de tamaño agnóstico, los punteros de `void` no se pueden incrementar o disminuir:

```
var ptr = (void*)IntPtr.Zero;
Console.WriteLine(new IntPtr(ptr));
ptr++; // compile-time error
Console.WriteLine(new IntPtr(ptr));
ptr++; // compile-time error
Console.WriteLine(new IntPtr(ptr));
```

### El asterisco es parte del tipo.

En C y C ++, el asterisco en la declaración de una variable de puntero es *parte de la expresión que* se declara. En C #, el asterisco en la declaración es *parte del tipo* .

En C, C ++ y C #, el siguiente fragmento de código declara un puntero `int` :

```
int* a;
```

En C y C ++, el siguiente fragmento de código declara un puntero `int` y una variable `int` . En C #, declara dos punteros `int` :

```
int* a, b;
```

En C y C ++, el siguiente fragmento de código declara dos punteros `int` . En C #, no es válido:

```
int *a, *b;
```

### vacío*

C # hereda de C y C ++ el uso de `void*` como un puntero de tipo agnóstico y de tamaño.

```
void* ptr;
```

Cualquier tipo de puntero se puede asignar a `void*` mediante una conversión implícita:

```
int* p1 = (int*)IntPtr.Zero;
void* ptr = p1;
```

Lo contrario requiere una conversión explícita:

```
int* p1 = (int*)IntPtr.Zero;
void* ptr = p1;
int* p2 = (int*)ptr;
```

### Acceso de miembros usando ->

C # hereda de C y C ++ el uso del símbolo `->` como un medio para acceder a los miembros de una instancia a través de un puntero escrito.

Considere la siguiente estructura:

```
struct Vector2
{
    public int X;
    public int Y;
}
```

Este es un ejemplo del uso de `->` para acceder a sus miembros:

```
Vector2 v;
v.X = 5;
v.Y = 10;

Vector2* ptr = &v;
int x = ptr->X;
int y = ptr->Y;
string s = ptr->ToString();

Console.WriteLine(x); // prints 5
Console.WriteLine(y); // prints 10
Console.WriteLine(s); // prints Vector2
```

## Punteros genéricos

Los criterios que debe cumplir un tipo para admitir punteros (ver *Comentarios* ) no pueden expresarse en términos de restricciones genéricas. Por lo tanto, cualquier intento de declarar un puntero a un tipo proporcionado a través de un parámetro de tipo genérico fallará.

```
void P<T>(T obj)
    where T : struct
{
    T* ptr = &obj; // compile-time error
}
```

Lea Punteros en línea: https://riptutorial.com/es/csharp/topic/5524/punteros

# Capítulo 137: Punteros y código inseguro

## Examples

### Introducción al código inseguro.

C # permite usar variables de puntero en una función de bloque de código cuando está marcado por el modificador `unsafe` . El código inseguro o el código no administrado es un bloque de código que utiliza una variable de puntero.

Un puntero es una variable cuyo valor es la dirección de otra variable, es decir, la dirección directa de la ubicación de la memoria. similar a cualquier variable o constante, debe declarar un puntero antes de poder usarlo para almacenar cualquier dirección de variable.

La forma general de una declaración de puntero es:

```
type *var-name;
```

Las siguientes son declaraciones de puntero válidas:

```
int    *ip;    /* pointer to an integer */
double *dp;    /* pointer to a double */
float  *fp;    /* pointer to a float */
char   *ch     /* pointer to a character */
```

El siguiente ejemplo ilustra el uso de punteros en C #, usando el modificador no seguro:

```
using System;
namespace UnsafeCodeApplication
{
   class Program
   {
      static unsafe void Main(string[] args)
      {
         int var = 20;
         int* p = &var;
         Console.WriteLine("Data is: {0} ",  var);
         Console.WriteLine("Address is: {0}",  (int)p);
         Console.ReadKey();
      }
   }
}
```

Cuando el código anterior fue compilado y ejecutado, produce el siguiente resultado:

```
Data is: 20
Address is: 99215364
```

En lugar de declarar un método completo como inseguro, también puede declarar una parte del

código como inseguro:

```
// safe code
unsafe
{
    // you can use pointers here
}
// safe code
```

## Recuperar el valor de los datos utilizando un puntero

Puede recuperar los datos almacenados en la ubicación a la que hace referencia la variable de puntero, utilizando el método ToString (). El siguiente ejemplo demuestra esto:

```
using System;
namespace UnsafeCodeApplication
{
   class Program
   {
      public static void Main()
      {
         unsafe
         {
            int var = 20;
            int* p = &var;
            Console.WriteLine("Data is: {0} " , var);
            Console.WriteLine("Data is: {0} " , p->ToString());
            Console.WriteLine("Address is: {0} " , (int)p);
         }

         Console.ReadKey();
      }
   }
}
```

Cuando el código anterior fue compilado y ejecutado, produce el siguiente resultado:

```
Data is: 20
Data is: 20
Address is: 77128984
```

## Pasando punteros como parámetros a métodos

Puede pasar una variable de puntero a un método como parámetro. El siguiente ejemplo lo ilustra:

```
using System;
namespace UnsafeCodeApplication
{
   class TestPointer
   {
      public unsafe void swap(int* p, int *q)
      {
         int temp = *p;
```

```
         *p = *q;
         *q = temp;
      }

      public unsafe static void Main()
      {
         TestPointer p = new TestPointer();
         int var1 = 10;
         int var2 = 20;
         int* x = &var1;
         int* y = &var2;

         Console.WriteLine("Before Swap: var1:{0}, var2: {1}", var1, var2);
         p.swap(x, y);

         Console.WriteLine("After Swap: var1:{0}, var2: {1}", var1, var2);
         Console.ReadKey();
      }
   }
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Before Swap: var1: 10, var2: 20
After Swap: var1: 20, var2: 10
```

## Acceso a elementos de matriz utilizando un puntero

En C #, un nombre de matriz y un puntero a un tipo de datos igual que los datos de matriz, no son el mismo tipo de variable. Por ejemplo, `int *p` e `int[] p` , no son del mismo tipo. Puede incrementar la variable de puntero `p` porque no está fija en la memoria, pero una dirección de matriz está fija en la memoria y no puede incrementarla.

Por lo tanto, si necesita acceder a una matriz de datos utilizando una variable de puntero, como lo hacemos tradicionalmente en C, o C ++, necesita corregir el puntero usando la palabra clave fija.

El siguiente ejemplo demuestra esto:

```
using System;
namespace UnsafeCodeApplication
{
   class TestPointer
   {
      public unsafe static void Main()
      {
         int[]  list = {10, 100, 200};
         fixed(int *ptr = list)

         /* let us have array address in pointer */
         for ( int i = 0; i < 3; i++)
         {
            Console.WriteLine("Address of list[{0}]={1}",i,(int)(ptr + i));
            Console.WriteLine("Value of list[{0}]={1}", i, *(ptr + i));
         }

         Console.ReadKey();
```

```
        }
    }
}
```

Cuando el código anterior fue compilado y ejecutado, produce el siguiente resultado:

```
Address of list[0] = 31627168
Value of list[0] = 10
Address of list[1] = 31627172
Value of list[1] = 100
Address of list[2] = 31627176
Value of list[2] = 200
```

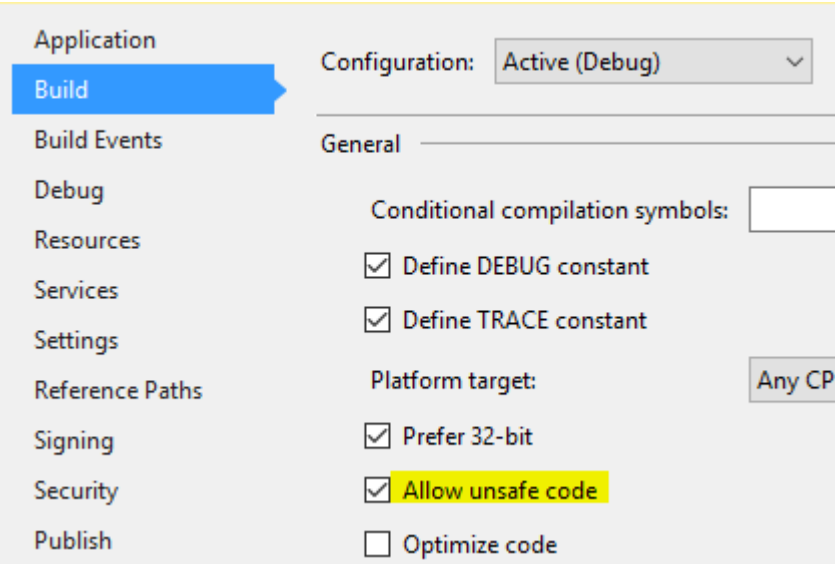
## Compilar código inseguro

Para compilar código inseguro, debe especificar el `/unsafe` de línea de comandos `/unsafe` con compilador de línea de comandos.

Por ejemplo, para compilar un programa llamado prog1.cs que contiene código no seguro, desde la línea de comandos, indique el comando:

```
csc /unsafe prog1.cs
```

Si está utilizando el IDE de Visual Studio, debe habilitar el uso de código no seguro en las propiedades del proyecto.




Para hacer esto:

- Abra las propiedades del proyecto haciendo doble clic en el nodo de propiedades en el Explorador de soluciones.
- Haga clic en la pestaña Construir.
- Seleccione la opción "Permitir código no seguro"

Lea Punteros y código inseguro en línea: https://riptutorial.com/es/csharp/topic/5514/punteros-y-codigo-inseguro

# Capítulo 138: Realizando peticiones HTTP

## Examples

### Creando y enviando una solicitud HTTP POST

```
using System.Net;
using System.IO;

...

string requestUrl = "https://www.example.com/submit.html";
HttpWebRequest request = HttpWebRequest.CreateHttp(requestUrl);
request.Method = "POST";

// Optionally, set properties of the HttpWebRequest, such as:
request.AutomaticDecompression = DecompressionMethods.Deflate | DecompressionMethods.GZip;
request.ContentType = "application/x-www-form-urlencoded";
// Could also set other HTTP headers such as Request.UserAgent, Request.Referer,
// Request.Accept, or other headers via the Request.Headers collection.

// Set the POST request body data. In this example, the POST data is in
// application/x-www-form-urlencoded format.
string postData = "myparam1=myvalue1&myparam2=myvalue2";
using (var writer = new StreamWriter(request.GetRequestStream()))
{
    writer.Write(postData);
}

// Submit the request, and get the response body from the remote server.
string responseFromRemoteServer;
using (HttpWebResponse response = (HttpWebResponse)request.GetResponse())
{
    using (StreamReader reader = new StreamReader(response.GetResponseStream()))
    {
        responseFromRemoteServer = reader.ReadToEnd();
    }
}
```

### Creando y enviando una solicitud HTTP GET

```
using System.Net;
using System.IO;

...

string requestUrl = "https://www.example.com/page.html";
HttpWebRequest request = HttpWebRequest.CreateHttp(requestUrl);

// Optionally, set properties of the HttpWebRequest, such as:
request.AutomaticDecompression = DecompressionMethods.GZip | DecompressionMethods.Deflate;
request.Timeout = 2 * 60 * 1000; // 2 minutes, in milliseconds

// Submit the request, and get the response body.
string responseBodyFromRemoteServer;
```

```
using (HttpWebResponse response = (HttpWebResponse)request.GetResponse())
{
    using (StreamReader reader = new StreamReader(response.GetResponseStream()))
    {
        responseBodyFromRemoteServer = reader.ReadToEnd();
    }
}
```

## Error al manejar códigos de respuesta HTTP específicos (como 404 No encontrado)

```
using System.Net;

...

string serverResponse;
try
{
    // Call a method that performs an HTTP request (per the above examples).
    serverResponse = PerformHttpRequest();
}
catch (WebException ex)
{
    if (ex.Status == WebExceptionStatus.ProtocolError)
    {
        HttpWebResponse response = ex.Response as HttpWebResponse;
        if (response != null)
        {
            if ((int)response.StatusCode == 404) // Not Found
            {
                // Handle the 404 Not Found error
                // ...
            }
            else
            {
                // Could handle other response.StatusCode values here.
                // ...
            }
        }
    }
    else
    {
        // Could handle other error conditions here, such as
WebExceptionStatus.ConnectFailure.
        // ...
    }
}
```

## Envío de solicitud HTTP POST asíncrona con cuerpo JSON

```
public static async Task PostAsync(this Uri uri, object value)
{
    var content = new ObjectContext(value.GetType(), value, new JsonMediaTypeFormatter());

    using (var client = new HttpClient())
    {
        return await client.PostAsync(uri, content);
```

```
    }
}

. . .

var uri = new Uri("http://stackoverflow.com/documentation/c%23/1971/performing-http-
requests");
await uri.PostAsync(new { foo = 123.45, bar = "Richard Feynman" });
```

## Enviar una solicitud HTTP GET asíncrona y leer una solicitud JSON

```
public static async Task<TResult> GetAnsync<TResult>(this Uri uri)
{
    using (var client = new HttpClient())
    {
        var message = await client.GetAsync(uri);

        if (!message.IsSuccessStatusCode)
            throw new Exception();

        return message.ReadAsAsync<TResult>();
    }
}

. . .

public class Result
{
    public double foo { get; set; }

    public string bar { get; set; }
}

var uri = new Uri("http://stackoverflow.com/documentation/c%23/1971/performing-http-
requests");
var result = await uri.GetAsync<Result>();
```

## Recuperar HTML para página web (Simple)

```
string contents = "";
string url = "http://msdn.microsoft.com";

using (System.Net.WebClient client = new System.Net.WebClient())
{
    contents = client.DownloadString(url);
}

Console.WriteLine(contents);
```

Lea Realizando peticiones HTTP en línea: https://riptutorial.com/es/csharp/topic/1971/realizando-peticiones-http

# Capítulo 139: Rebosar

## Examples

### Desbordamiento de enteros

Hay una capacidad máxima que un entero puede almacenar. Y cuando superas ese límite, volverá al lado negativo. Para `int` , es `2147483647`

```
int x = int.MaxValue;                //MaxValue is 2147483647
x = unchecked(x + 1);                //make operation explicitly unchecked so that the example
also works when the check for arithmetic overflow/underflow is enabled in the project settings

Console.WriteLine(x);                //Will print -2147483648
Console.WriteLine(int.MinValue);     //Same as Min value
```

Para los enteros fuera de este rango, use el espacio de nombres System.Numerics que tiene el tipo de datos BigInteger. Consulte el siguiente enlace para obtener más información https://msdn.microsoft.com/en-us/library/system.numerics.biginteger(v=vs.110).aspx

### Desbordamiento durante la operación

El desbordamiento también ocurre durante la operación. En el siguiente ejemplo, x es un `int` , 1 es un `int` por defecto. Por lo tanto, la adición es una adición `int` . Y el resultado será un `int` . Y se desbordará.

```
int x = int.MaxValue;               //MaxValue is 2147483647
long y = x + 1;                     //It will be overflown
Console.WriteLine(y);               //Will print -2147483648
Console.WriteLine(int.MinValue);    //Same as Min value
```

Puedes evitar eso usando 1L. Ahora 1 será un `long` y la adición será un `long` agregado

```
int x = int.MaxValue;               //MaxValue is 2147483647
long y = x + 1L;                    //It will be OK
Console.WriteLine(y);               //Will print 2147483648
```

### Asuntos de orden

Hay desbordamiento en el siguiente código.

```
int x = int.MaxValue;
Console.WriteLine(x + x + 1L);  //prints -1
```

Mientras que en el siguiente código no hay desbordamiento

```
int x = int.MaxValue;
```

```
Console.WriteLine(x + 1L + x);  //prints 4294967295
```

Esto se debe a la ordenación de izquierda a derecha de las operaciones. En el primer fragmento de código, `x + x` desborda y después de eso se vuelve `long` . Por otro lado, `x + 1L` vuelve `long` y luego se agrega `x` a este valor.

Lea Rebosar en línea: https://riptutorial.com/es/csharp/topic/3303/rebosar

# Capítulo 140: Recolector de basura en .Net

## Examples

### Compactación de objetos grandes

De forma predeterminada, el Large Object Heap no se compacta a diferencia del Object Heap clásico, que puede conducir a la fragmentación de la memoria y, además, puede llevar a `OutOfMemoryException` S

A partir de .NET 4.5.1, hay una opción para compactar explícitamente el montón de objetos grandes (junto con una recolección de basura):

```
GCSettings.LargeObjectHeapCompactionMode = GCLargeObjectHeapCompactionMode.CompactOnce;
GC.Collect();
```

Al igual que cualquier solicitud de recolección de basura explícita (se llama solicitud porque el CLR no está obligado a realizarla), utilice con cuidado y evítelo de manera predeterminada si puede, ya que puede `GC` las estadísticas de `GC` , lo que disminuye su rendimiento.

### Referencias débiles

En .NET, el GC asigna objetos cuando no hay referencias a ellos. Por lo tanto, si bien se puede acceder a un objeto desde el código (hay una fuerte referencia a él), el GC no asignará este objeto. Esto puede convertirse en un problema si hay muchos objetos grandes.

Una referencia débil es una referencia, que le permite al GC recolectar el objeto y al mismo tiempo permitir el acceso al objeto. Una referencia débil solo es válida durante un período de tiempo indeterminado hasta que el objeto se recopila cuando no existen referencias sólidas. Cuando utiliza una referencia débil, la aplicación aún puede obtener una referencia fuerte al objeto, lo que evita que se recopile. Por lo tanto, las referencias débiles pueden ser útiles para aferrar objetos grandes que son costosos de inicializar, pero deberían estar disponibles para la recolección de basura si no están en uso.

Uso simple:

```
WeakReference reference = new WeakReference(new object(), false);

GC.Collect();

object target = reference.Target;
if (target != null)
  DoSomething(target);
```

Por lo tanto, las referencias débiles podrían usarse para mantener, por ejemplo, un caché de objetos. Sin embargo, es importante recordar que siempre existe el riesgo de que el recolector de basura llegue al objeto antes de que se restablezca una referencia sólida.

Las referencias débiles también son útiles para evitar pérdidas de memoria. Un caso de uso típico es con eventos.

Supongamos que tenemos algún controlador para un evento en una fuente:

```
Source.Event += new EventHandler(Handler)
```

Este código registra un controlador de eventos y crea una fuerte referencia desde el origen del evento hasta el objeto que escucha. Si el objeto de origen tiene una vida útil más larga que la del oyente, y el oyente ya no necesita el evento cuando no hay otras referencias a él, el hecho de que los eventos .NET normales provoquen una pérdida de memoria: el objeto de origen guarda los objetos del oyente en la memoria. Se debe recoger la basura.

En este caso, puede ser una buena idea usar el Patrón de evento débil .

Algo como:

```
public static class WeakEventManager
    {
    public static void SetHandler<S, TArgs>(
    Action<EventHandler<TArgs>> add,
    Action<EventHandler<TArgs>> remove,
    S subscriber,
    Action<S, TArgs> action)
    where TArgs : EventArgs
    where S : class
        {
            var subscrWeakRef = new WeakReference(subscriber);
            EventHandler<TArgs> handler = null;

            handler = (s, e) =>
            {
                var subscrStrongRef = subscrWeakRef.Target as S;
                if (subscrStrongRef != null)
                {
                    action(subscrStrongRef, e);
                }
                else
                {
                    remove(handler);
                    handler = null;
                }
            };

            add(handler);
        }
    }
```

y usado así:

```
 EventSource s = new EventSource();
 Subscriber subscriber = new Subscriber();
 WeakEventManager.SetHandler<Subscriber, SomeEventArgs>(a => s.Event += a, r => s.Event -= r,
subscriber, (s,e) => { s.HandleEvent(e); });
```

En este caso, por supuesto, tenemos algunas restricciones: el evento debe ser un

```
public event EventHandler<SomeEventArgs> Event;
```

Como sugiere MSDN :

- Utilice referencias débiles largas solo cuando sea necesario, ya que el estado del objeto es impredecible después de la finalización.
- Evite usar referencias débiles para objetos pequeños porque el puntero en sí puede ser tan grande o más grande.
- Evite utilizar referencias débiles como una solución automática a los problemas de administración de memoria. En su lugar, desarrolle una política de almacenamiento en caché eficaz para manejar los objetos de su aplicación.

Lea Recolector de basura en .Net en línea: https://riptutorial.com/es/csharp/topic/1287/recolector-de-basura-en--net

# Capítulo 141: Recursion

## Observaciones

Tenga en cuenta que el uso de la recursión puede tener un gran impacto en su código, ya que cada llamada a la función recursiva se agregará a la pila. Si hay demasiadas llamadas, esto podría llevar a una excepción de **StackOverflow** . La mayoría de las "funciones recursivas naturales" se pueden escribir como una construcción de bucle `for` , `while` o `foreach` , y aunque no se vea tan **elegante** o **inteligente** será más eficiente.

Siempre piense dos veces y use la recursión con cuidado, sepa por qué lo usa:

- la recursión debe usarse cuando se sabe que el número de llamadas recursivas no es *excesivo*
    - medios *excesivos* , depende de la cantidad de memoria disponible
- recursion se usa porque es una versión de código más clara y limpia, es más legible que una función iterativa o basada en bucle. A menudo, este es el caso porque proporciona un código más limpio y más compacto (también conocido como menos líneas de código).
    - ¡Pero ten cuidado, puede ser menos eficiente! Por ejemplo, en la recursión de Fibonacci, para calcular el número *n* en la secuencia, ¡el tiempo de cálculo aumentará exponencialmente!

Si quieres más teoría, por favor lee:

- https://www.cs.umd.edu/class/fall2002/cmsc214/Tutorial/recursion2.html
- https://en.wikipedia.org/wiki/Recursion#In_computer_science

## Examples

### Describir recursivamente una estructura de objeto.

La recursividad es cuando un método se llama a sí mismo. Preferiblemente, lo hará hasta que se cumpla una condición específica y luego saldrá del método normalmente, volviendo al punto desde el cual se llamó el método. De lo contrario, podría producirse una excepción de desbordamiento de pila debido a demasiadas llamadas recursivas.

```
/// <summary>
/// Create an object structure the code can recursively describe
/// </summary>
public class Root
{
    public string Name { get; set; }
    public ChildOne Child { get; set; }
}
public class ChildOne
{
    public string ChildOneName { get; set; }
    public ChildTwo Child { get; set; }
```

```
}
public class ChildTwo
{
    public string ChildTwoName { get; set; }
}
/// <summary>
/// The console application with the recursive function DescribeTypeOfObject
/// </summary>
public class Program
{
    static void Main(string[] args)
    {
        // point A, we call the function with type 'Root'
        DescribeTypeOfObject(typeof(Root));
        Console.WriteLine("Press a key to exit");
        Console.ReadKey();
    }

    static void DescribeTypeOfObject(Type type)
    {
        // get all properties of this type
        Console.WriteLine($"Describing type {type.Name}");
        PropertyInfo[] propertyInfos = type.GetProperties();
        foreach (PropertyInfo pi in propertyInfos)
        {
            Console.WriteLine($"Has property {pi.Name} of type {pi.PropertyType.Name}");
            // is a custom class type? describe it too
            if (pi.PropertyType.IsClass && !pi.PropertyType.FullName.StartsWith("System."))
            {
                // point B, we call the function type this property
                DescribeTypeOfObject(pi.PropertyType);
            }
        }
        // done with all properties
        // we return to the point where we were called
        // point A for the first call
        // point B for all properties of type custom class
    }
}
```

## Recursion en ingles llano

La recursión se puede definir como:

> Un método que se llama a sí mismo hasta que se cumple una condición específica.

Un ejemplo excelente y simple de recursión es un método que obtendrá el factorial de un número dado:

```
public int Factorial(int number)
{
    return number == 0 ? 1 : n * Factorial(number - 1);
}
```

En este método, podemos ver que el método tomará un argumento, `number` .

Paso a paso:

Dado el ejemplo, ejecutando `Factorial(4)`

1. ¿Es el `number (4) == 1` ?
2. ¿No? retorno `4 * Factorial(number-1)` (3)
3. Debido a que se vuelve a llamar al método, ahora repite el primer paso utilizando `Factorial(3)` como el nuevo argumento.
4. Esto continúa hasta que se ejecuta `Factorial(1)` y el `number (1) == 1` devuelve 1.
5. En general, el cálculo "acumula" `4 * 3 * 2 * 1` y finalmente devuelve 24.

La clave para comprender la recursión es que el método llama a una *nueva instancia* de sí mismo. Después de volver, la ejecución de la instancia de llamada continúa.

## Usando la recursividad para obtener el árbol de directorios

Uno de los usos de la recursión es navegar a través de una estructura de datos jerárquica, como un árbol de directorios del sistema de archivos, sin saber cuántos niveles tiene el árbol o la cantidad de objetos en cada nivel. En este ejemplo, verá cómo usar la recursión en un árbol de directorios para encontrar todos los subdirectorios de un directorio específico e imprimir todo el árbol en la consola.

```
internal class Program
{
    internal const int RootLevel = 0;
    internal const char Tab = '\t';

    internal static void Main()
    {
        Console.WriteLine("Enter the path of the root directory:");
        var rootDirectorypath = Console.ReadLine();

        Console.WriteLine(
            $"Getting directory tree of '{rootDirectorypath}'");

        PrintDirectoryTree(rootDirectorypath);
        Console.WriteLine("Press 'Enter' to quit...");
        Console.ReadLine();
    }

    internal static void PrintDirectoryTree(string rootDirectoryPath)
    {
        try
        {
            if (!Directory.Exists(rootDirectoryPath))
            {
                throw new DirectoryNotFoundException(
                    $"Directory '{rootDirectoryPath}' not found.");
            }

            var rootDirectory = new DirectoryInfo(rootDirectoryPath);
            PrintDirectoryTree(rootDirectory, RootLevel);
        }
        catch (DirectoryNotFoundException e)
        {
            Console.WriteLine(e.Message);
        }
```

```
    }

    private static void PrintDirectoryTree(
        DirectoryInfo directory, int currentLevel)
    {
        var indentation = string.Empty;
        for (var i = RootLevel; i < currentLevel; i++)
        {
            indentation += Tab;
        }

        Console.WriteLine($"{indentation}-{directory.Name}");
        var nextLevel = currentLevel + 1;
        try
        {
            foreach (var subDirectory in directory.GetDirectories())
            {
                PrintDirectoryTree(subDirectory, nextLevel);
            }
        }
        catch (UnauthorizedAccessException e)
        {
            Console.WriteLine($"{indentation}-{e.Message}");
        }
    }
}
```

Este código es algo más complicado que el mínimo para completar esta tarea, ya que incluye la verificación de excepciones para manejar cualquier problema al obtener los directorios. A continuación encontrará un desglose del código en segmentos más pequeños con explicaciones de cada uno.

`Main` :

El método principal toma una entrada de un usuario como una cadena, que se utiliza como la ruta al directorio raíz. Luego llama al método `PrintDirectoryTree` con esta cadena como parámetro.

`PrintDirectoryTree(string)` :

Este es el primero de los dos métodos que manejan la impresión real del árbol de directorios. Este método toma una cadena que representa la ruta al directorio raíz como un parámetro. Comprueba si la ruta es un directorio real y, de no ser así, lanza una excepción `DirectoryNotFoundException` que luego se maneja en el bloque catch. Si la ruta es un directorio real, un `DirectoryInfo` objeto `rootDirectory` se crea de la trayectoria, y la segunda `PrintDirectoryTree` método se llama con el `rootDirectory` objeto y `RootLevel` , que es una constante con un valor de cero entero.

`PrintDirectoryTree(DirectoryInfo, int)` :

Este segundo método maneja la peor parte del trabajo. Toma un `DirectoryInfo` y un entero como parámetros. `DirectoryInfo` es el directorio actual y el entero es la profundidad del directorio en relación con la raíz. Para facilitar la lectura, la salida se sangra para cada nivel en que se encuentra el directorio actual, de modo que la salida se vea así:

```
-Root
    -Child 1
    -Child 2
        -Grandchild 2.1
    -Child 3
```

Una vez que se imprime el directorio actual, se recuperan sus subdirectorios, y luego se llama a este método en cada uno de ellos con un valor de nivel de profundidad de uno más que el actual. Esa parte es la recursión: el método que se llama a sí mismo. El programa se ejecutará de esta manera hasta que haya visitado todos los directorios del árbol. Cuando llegó a un directorio sin subdirectorios, el método volverá automáticamente.

Este método también detecta una `UnauthorizedAccessException` , que se produce si alguno de los subdirectorios del directorio actual está protegido por el sistema. El mensaje de error se imprime en el nivel de sangría actual para mantener la coherencia.

El método a continuación proporciona un enfoque más básico para este problema:

```
internal static void PrintDirectoryTree(string directoryName)
{
    try
    {
        if (!Directory.Exists(directoryName)) return;
        Console.WriteLine(directoryName);
        foreach (var d in Directory.GetDirectories(directoryName))
        {
            PrintDirectoryTree(d);
        }
    }
    catch (Exception e)
    {
        Console.WriteLine(e.Message);
    }
}
```

Esto no incluye la comprobación de errores específica o el formato de salida del primer enfoque, pero efectivamente hace lo mismo. Como solo usa cadenas en lugar de `DirectoryInfo` , no puede proporcionar acceso a otras propiedades de directorio como permisos.

## Secuencia Fibonacci

Puedes calcular un número en la secuencia de Fibonacci usando la recursión.

Siguiendo la teoría matemática de F (n) = F (n-2) + F (n-1), para cualquier i> 0,

```
// Returns the i'th Fibonacci number
public int fib(int i) {
    if(i <= 2) {
        // Base case of the recursive function.
        // i is either 1 or 2, whose associated Fibonacci sequence numbers are 1 and 1.
        return 1;
    }
    // Recursive case. Return the sum of the two previous Fibonacci numbers.
    // This works because the definition of the Fibonacci sequence specifies
```

```
    // that the sum of two adjacent elements equals the next element.
    return  fib(i - 2) + fib(i - 1);

}

fib(10); // Returns 55
```

## Calculo factorial

El factorial de un número (indicado con!, Como por ejemplo 9!) Es la multiplicación de ese número con el factorial de uno más bajo. Así, por ejemplo, 9! = 9 x 8! = 9 x 8 x 7! = 9 x 8 x 7 x 6 x 5 x 4 x 3 x 2 x 1.

Así en el código que se convierte, usando la recursión:

```
long Factorial(long x)
{
    if (x < 1)
    {
        throw new OutOfRangeException("Factorial can only be used with positive numbers.");
    }

    if (x == 1)
    {
        return 1;
    } else {
        return x * Factorial(x - 1);
    }
}
```

## Cálculo de PowerOf

El cálculo de la potencia de un número dado también se puede hacer recursivamente. Dado un número base `n` y un exponente `e` , debemos asegurarnos de dividir el problema en partes reduciendo el exponente `e` .

Ejemplo teórico:

- $2^2 = 2x2$
- $2^3 = 2x2x2$ o, $2^3 = 2^2 \times 2$
  Ahí radica el secreto de nuestro algoritmo recursivo (ver el código a continuación). Se trata de tomar el problema y separarlo en pequeños y más simples para resolver los trozos.
- **Notas**
  - cuando el número base es 0, debemos estar conscientes de devolver 0 como $0^3 = 0 \times 0 \times 0$
  - cuando el exponente es 0, debemos ser conscientes de devolver siempre 1, ya que esta es una regla matemática.

Ejemplo de código:

```
public int CalcPowerOf(int b, int e) {
    if (b == 0) { return 0; } // when base is 0, it doesn't matter, it will always return 0
```

```
    if (e == 0) { return 1; } // math rule, exponent 0 always returns 1
    return b * CalcPowerOf(b, e - 1); // actual recursive logic, where we split the problem,
aka: 2³ = 2 * 2² etc..
}
```

Pruebas en xUnit para verificar la lógica:
Aunque esto no es necesario, siempre es bueno escribir pruebas para verificar su lógica. Los incluyo aquí escritos en el marco de xUnit .

```
    [Theory]
    [MemberData(nameof(PowerOfTestData))]
    public void PowerOfTest(int @base, int exponent, int expected) {
        Assert.Equal(expected, CalcPowerOf(@base, exponent));
    }

    public static IEnumerable<object[]> PowerOfTestData() {
        yield return new object[] { 0, 0, 0 };
        yield return new object[] { 0, 1, 0 };
        yield return new object[] { 2, 0, 1 };
        yield return new object[] { 2, 1, 2 };
        yield return new object[] { 2, 2, 4 };
        yield return new object[] { 5, 2, 25 };
        yield return new object[] { 5, 3, 125 };
        yield return new object[] { 5, 4, 625 };
}
```

Lea Recursion en línea: https://riptutorial.com/es/csharp/topic/2470/recursion

# Capítulo 142: Redes

## Sintaxis

- TcpClient (host de cadena, puerto int);

## Observaciones

Puede obtener `NetworkStream` desde un `TcpClient` con `client.GetStream()` y pasarlo a un `StreamReader/StreamWriter` para obtener acceso a sus métodos asíncronos de lectura y escritura.

## Examples

### Cliente de comunicación TCP básico

Este ejemplo de código crea un cliente TCP, envía "Hello World" a través de la conexión de socket y luego escribe la respuesta del servidor a la consola antes de cerrar la conexión.

```
// Declare Variables
string host = "stackoverflow.com";
int port = 9999;
int timeout = 5000;

// Create TCP client and connect
using (var _client = new TcpClient(host, port))
using (var _netStream = _client.GetStream())
{
    _netStream.ReadTimeout = timeout;

    // Write a message over the socket
    string message = "Hello World!";
    byte[] dataToSend = System.Text.Encoding.ASCII.GetBytes(message);
    _netStream.Write(dataToSend, 0, dataToSend.Length);

    // Read server response
    byte[] recvData = new byte[256];
    int bytes = _netStream.Read(recvData, 0, recvData.Length);
    message = System.Text.Encoding.ASCII.GetString(recvData, 0, bytes);
    Console.WriteLine(string.Format("Server: {0}", message));
};// The client and stream will close as control exits the using block (Equivilent but safer
than calling Close();
```

### Descargar un archivo desde un servidor web

Descargar un archivo de Internet es una tarea muy común que requieren casi todas las aplicaciones que es probable que compile.

Para lograr esto, puede usar la clase " System.Net.WebClient ".

El uso más simple de esto, utilizando el patrón de "uso", se muestra a continuación:

```
using (var webClient = new WebClient())
{
    webClient.DownloadFile("http://www.server.com/file.txt", "C:\\file.txt");
}
```

Lo que hace este ejemplo es usar "usar" para asegurarse de que su cliente web se limpie correctamente cuando termine, y simplemente transfiere el recurso nombrado desde la URL en el primer parámetro, al archivo nombrado en su disco duro local en el segundo parámetro.

El primer parámetro es de tipo " System.Uri ", el segundo parámetro es de tipo " System.String "

También puede usar esta función en forma asíncrona, de modo que se active y realice la descarga en segundo plano, mientras que su aplicación se inicia con otra cosa, usar la llamada de esta manera es de gran importancia en las aplicaciones modernas, ya que ayuda para mantener su interfaz de usuario sensible.

Cuando utiliza los métodos asíncronos, puede conectar controladores de eventos que le permiten monitorear el progreso, por ejemplo, puede actualizar una barra de progreso, como la siguiente:

```
var webClient = new WebClient())
webClient.DownloadFileCompleted += new AsyncCompletedEventHandler(Completed);
webClient.DownloadProgressChanged += new DownloadProgressChangedEventHandler(ProgressChanged);
webClient.DownloadFileAsync("http://www.server.com/file.txt", "C:\\file.txt");
```

Sin embargo, debe recordar un punto importante si usa las versiones de Async, y es "Tenga mucho cuidado de usarlas en una sintaxis 'de uso'".

La razón de esto es bastante simple. Una vez que llame al método de descarga de archivo, volverá inmediatamente. Si tiene esto en un bloque de uso, regresará y luego saldrá de ese bloque, e inmediatamente desechará el objeto de clase, y así cancelará la descarga en curso.

Si utiliza la forma 'usando' de realizar una transferencia asíncrona, asegúrese de permanecer dentro del bloque adjunto hasta que finalice la transferencia.

## Async TCP Client

El uso de `async/await` en aplicaciones de C # simplifica los subprocesos múltiples. Así es como puede usar `async/await` await junto con un TcpClient.

```
// Declare Variables
string host = "stackoverflow.com";
int port = 9999;
int timeout = 5000;

// Create TCP client and connect
// Then get the netstream and pass it
// To our StreamWriter and StreamReader
using (var client = new TcpClient())
using (var netstream = client.GetStream())
using (var writer = new StreamWriter(netstream))
using (var reader = new StreamReader(netstream))
{
```

```
    // Asynchronsly attempt to connect to server
    await client.ConnectAsync(host, port);

    // AutoFlush the StreamWriter
    // so we don't go over the buffer
    writer.AutoFlush = true;

    // Optionally set a timeout
    netstream.ReadTimeout = timeout;

    // Write a message over the TCP Connection
    string message = "Hello World!";
    await writer.WriteLineAsync(message);

    // Read server response
    string response = await reader.ReadLineAsync();
    Console.WriteLine(string.Format($"Server: {response}"));
}
// The client and stream will close as control exits
// the using block (Equivilent but safer than calling Close();
```

## Cliente UDP básico

Este ejemplo de código crea un cliente UDP y luego envía "Hello World" a través de la red al destinatario deseado. Un oyente no tiene que estar activo, ya que UDP no tiene conexión y transmitirá el mensaje independientemente. Una vez que se envía el mensaje, el trabajo de los clientes está hecho.

```
byte[] data = Encoding.ASCII.GetBytes("Hello World");
string ipAddress = "192.168.1.141";
string sendPort = 55600;
try
{
     using (var client = new UdpClient())
     {
         IPEndPoint ep = new IPEndPoint(IPAddress.Parse(ipAddress), sendPort);
         client.Connect(ep);
         client.Send(data, data.Length);
     }
}
catch (Exception ex)
{
     Console.WriteLine(ex.ToString());
}
```

A continuación se muestra un ejemplo de un oyente UDP para complementar el cliente anterior. Se sentará y escuchará constantemente el tráfico en un puerto determinado y simplemente escribirá esos datos en la consola. Este ejemplo contiene un indicador de control ' `done` ' que no se establece internamente y se basa en algo para configurarlo para permitir finalizar al oyente y salir.

```
bool done = false;
int listenPort = 55600;
using(UdpClinet listener = new UdpClient(listenPort))
{
    IPEndPoint listenEndPoint = new IPEndPoint(IPAddress.Any, listenPort);
    while(!done)
```

```
    {
        byte[] receivedData = listener.Receive(ref listenPort);

        Console.WriteLine("Received broadcast message from client {0}",
listenEndPoint.ToString());

        Console.WriteLine("Decoded data is:");
        Console.WriteLine(Encoding.ASCII.GetString(receivedData)); //should be "Hello World"
sent from above client
    }
}
```

Lea Redes en línea: https://riptutorial.com/es/csharp/topic/1352/redes

# Capítulo 143: Reflexión

## Introducción

La reflexión es un mecanismo de lenguaje C # para acceder a las propiedades dinámicas de los objetos en tiempo de ejecución. Normalmente, la reflexión se utiliza para obtener información sobre el tipo de objeto dinámico y los valores de atributo de objeto. En la aplicación REST, por ejemplo, la reflexión podría usarse para iterar a través del objeto de respuesta serializado.

Observación: De acuerdo con las directrices de MS, el código crítico de rendimiento debe evitar la reflexión. Consulte https://msdn.microsoft.com/en-us/library/ff647790.aspx

## Observaciones

Reflection permite que el código acceda a información sobre los ensamblajes, módulos y tipos en tiempo de ejecución (ejecución del programa). Esto puede luego ser usado para crear, modificar o acceder dinámicamente tipos. Los tipos incluyen propiedades, métodos, campos y atributos.

Otras lecturas :

Reflexión (C #)

Reflexión en .Net Framework

## Examples

### Obtener un System.Type

Para una instancia de un tipo:

```
var theString = "hello";
var theType = theString.GetType();
```

Del propio tipo:

```
var theType = typeof(string);
```

### Obtener los miembros de un tipo

```
using System;
using System.Reflection;
using System.Linq;

public class Program
{
  public static void Main()
```

```
  {
    var members = typeof(object)
                    .GetMembers(BindingFlags.Public |
                                BindingFlags.Static |
                                BindingFlags.Instance);

    foreach (var member in members)
    {
      bool inherited = member.DeclaringType.Equals( typeof(object).Name );
      Console.WriteLine($"{member.Name} is a {member.MemberType}, " +
                        $"it has {(inherited ? "":"not")} been inherited.");
    }
  }
}
```

Salida ( *ver nota sobre el orden de salida más abajo* ):

```
GetType is a Method, it has not been inherited.
GetHashCode is a Method, it has not been inherited.
ToString is a Method, it has not been inherited.
Equals is a Method, it has not been inherited.
Equals is a Method, it has not been inherited.
ReferenceEquals is a Method, it has not been inherited.
.ctor is a Constructor, it has not been inherited.
```

También podemos usar `GetMembers()` sin pasar `BindingFlags` . Esto devolverá a *todos* los miembros públicos de ese tipo específico.

Una cosa a tener en cuenta es que `GetMembers` no devuelve a los miembros en ningún orden en particular, por lo que nunca confíe en el pedido que `GetMembers` devuelve `GetMembers` .

Ver demostración

## Consigue un método e invocalo.

**Obtener el método de instancia e invocarlo.**

```
using System;

public class Program
{
    public static void Main()
    {
        var theString = "hello";
        var method = theString
                     .GetType()
                     .GetMethod("Substring",
                                new[] {typeof(int), typeof(int)}); //The types of the method
arguments
         var result = method.Invoke(theString, new object[] {0, 4});
         Console.WriteLine(result);
    }
}
```

**Salida:**

infierno

Ver demostración

**Obtener el método estático e invocarlo.**

Por otro lado, si el método es estático, no necesita una instancia para llamarlo.

```
var method = typeof(Math).GetMethod("Exp");
var result = method.Invoke(null, new object[] {2});//Pass null as the first argument (no need
for an instance)
Console.WriteLine(result); //You'll get e^2
```

**Salida:**

7.38905609893065

Ver demostración

## Obteniendo y configurando propiedades

Uso básico:

```
PropertyInfo prop = myInstance.GetType().GetProperty("myProperty");
// get the value myInstance.myProperty
object value = prop.GetValue(myInstance);

int newValue = 1;
// set the value myInstance.myProperty to newValue
prop.setValue(myInstance, newValue);
```

La configuración de las propiedades implementadas automáticamente de solo lectura se puede hacer a través de su campo de respaldo (en .NET Framework, el nombre del campo de respaldo es "k__BackingField"):

```
// get backing field info
FieldInfo fieldInfo = myInstance.GetType()
    .GetField("<myProperty>k__BackingField", BindingFlags.Instance | BindingFlags.NonPublic);

int newValue = 1;
// set the value of myInstance.myProperty backing field to newValue
fieldInfo.SetValue(myInstance, newValue);
```

## Atributos personalizados

**Buscar propiedades con un atributo personalizado** - `MyAttribute`

```
var props = t.GetProperties(BindingFlags.NonPublic | BindingFlags.Public |
            BindingFlags.Instance).Where(
            prop => Attribute.IsDefined(prop, typeof(MyAttribute)));
```

**Encuentra todos los atributos personalizados en una propiedad dada**

```
var attributes = typeof(t).GetProperty("Name").GetCustomAttributes(false);
```

### Enumerar todas las clases con atributo personalizado - `MyAttribute`

```
static IEnumerable<Type> GetTypesWithAttribute(Assembly assembly) {
    foreach(Type type in assembly.GetTypes()) {
        if (type.GetCustomAttributes(typeof(MyAttribute), true).Length > 0) {
            yield return type;
        }
    }
}
```

### Leer el valor de un atributo personalizado en tiempo de ejecución

```
public static class AttributeExtensions
{

        /// <summary>
        /// Returns the value of a member attribute for any member in a class.
        ///     (a member is a Field, Property, Method, etc...)
        /// <remarks>
        /// If there is more than one member of the same name in the class, it will return the
first one (this applies to overloaded methods)
        /// </remarks>
        /// <example>
        /// Read System.ComponentModel Description Attribute from method 'MyMethodName' in
class 'MyClass':
        ///     var Attribute = typeof(MyClass).GetAttribute("MyMethodName",
(DescriptionAttribute d) => d.Description);
        /// </example>
        /// <param name="type">The class that contains the member as a type</param>
        /// <param name="MemberName">Name of the member in the class</param>
        /// <param name="valueSelector">Attribute type and property to get (will return first
instance if there are multiple attributes of the same type)</param>
        /// <param name="inherit">true to search this member's inheritance chain to find the
attributes; otherwise, false. This parameter is ignored for properties and events</param>
        /// </summary>
        public static TValue GetAttribute<TAttribute, TValue>(this Type type, string
MemberName, Func<TAttribute, TValue> valueSelector, bool inherit = false) where TAttribute :
Attribute
        {
            var att =
type.GetMember(MemberName).FirstOrDefault().GetCustomAttributes(typeof(TAttribute),
inherit).FirstOrDefault() as TAttribute;
            if (att != null)
            {
                return valueSelector(att);
            }
            return default(TValue);
        }
    }
```

Uso

```
//Read System.ComponentModel Description Attribute from method 'MyMethodName' in class
'MyClass'
var Attribute = typeof(MyClass).GetAttribute("MyMethodName", (DescriptionAttribute d) =>
```

```
d.Description);
```

## Recorriendo todas las propiedades de una clase.

```
Type type = obj.GetType();
//To restrict return properties. If all properties are required don't provide flag.
BindingFlags flags = BindingFlags.Public | BindingFlags.Instance;
PropertyInfo[] properties = type.GetProperties(flags);

foreach (PropertyInfo property in properties)
{
    Console.WriteLine("Name: " + property.Name + ", Value: " + property.GetValue(obj, null));
}
```

## Determinación de argumentos genéricos de instancias de tipos genéricos.

Si tiene una instancia de un tipo genérico, pero por alguna razón no conoce el tipo específico, es posible que desee determinar los argumentos genéricos que se usaron para crear esta instancia.

Digamos que alguien creó una instancia de la `List<T>` así y la pasa a un método:

```
var myList = new List<int>();
ShowGenericArguments(myList);
```

donde `ShowGenericArguments` tiene esta firma:

```
public void ShowGenericArguments(object o)
```

así que en el momento de la compilación no tiene idea de qué argumentos genéricos se han utilizado para crear `o` . La reflexión proporciona muchos métodos para inspeccionar tipos genéricos. Al principio, podemos determinar si el tipo de `o` es un tipo genérico:

```
public void ShowGenericArguments(object o)
{
    if (o == null) return;

    Type t = o.GetType();
    if (!t.IsGenericType) return;
    ...
```

`Type.IsGenericType` devuelve `true` si el tipo es un tipo genérico y `false` si no lo es.

Pero esto no es todo lo que queremos saber. `List<>` sí misma es un tipo genérico, también. Pero solo queremos examinar instancias de tipos *genéricos construidos* específicos. Un tipo genérico construido es, por ejemplo, una `List<int>` que tiene un *argumento de* tipo específico para todos sus *parámetros* genéricos.

La clase `Type` proporciona dos propiedades más, `IsConstructedGenericType` e `IsGenericTypeDefinition` , para distinguir estos tipos genéricos construidos de definiciones de tipos genéricos:

```
typeof(List<>).IsGenericType // true
typeof(List<>).IsGenericTypeDefinition // true
typeof(List<>).IsConstructedGenericType// false

typeof(List<int>).IsGenericType // true
typeof(List<int>).IsGenericTypeDefinition // false
typeof(List<int>).IsConstructedGenericType// true
```

Para enumerar los argumentos genéricos de una instancia, podemos usar el método `GetGenericArguments()` que devuelve una matriz de `Type` contiene los argumentos de tipo genérico:

```
public void ShowGenericArguments(object o)
{
    if (o == null) return;
    Type t = o.GetType();
    if (!t.IsConstructedGenericType) return;

    foreach(Type genericTypeArgument in t.GetGenericArguments())
        Console.WriteLine(genericTypeArgument.Name);
}
```

Así que la llamada desde arriba ( `ShowGenericArguments(myList)` ) da como resultado esta salida:

```
Int32
```

## Consigue un método genérico e invocalo.

Digamos que tienes clase con métodos genéricos. Y necesitas llamar sus funciones con reflexión.

```
public class Sample
{
    public void GenericMethod<T>()
    {
        // ...
    }

    public static void StaticMethod<T>()
    {
        //...
    }
}
```

Digamos que queremos llamar al GenericMethod con el tipo de cadena.

```
Sample sample = new Sample();//or you can get an instance via reflection

MethodInfo method = typeof(Sample).GetMethod("GenericMethod");
MethodInfo generic = method.MakeGenericMethod(typeof(string));
generic.Invoke(sample, null);//Since there are no arguments, we are passing null
```

Para el método estático no necesita una instancia. Por lo tanto el primer argumento también será nulo.

```
MethodInfo method = typeof(Sample).GetMethod("StaticMethod");
MethodInfo generic = method.MakeGenericMethod(typeof(string));
generic.Invoke(null, null);
```

### Crear una instancia de un tipo genérico e invocar su método.

```
var baseType = typeof(List<>);
var genericType = baseType.MakeGenericType(typeof(String));
var instance = Activator.CreateInstance(genericType);
var method = genericType.GetMethod("GetHashCode");
var result = method.Invoke(instance, new object[] { });
```

### Clases de creación de instancias que implementan una interfaz (por ejemplo, activación de plugin)

Si desea que su aplicación admita un sistema de complementos, por ejemplo, para cargar complementos de ensamblajes ubicados en la carpeta de `plugins` :

```
interface IPlugin
{
    string PluginDescription { get; }
    void DoWork();
}
```

Esta clase estaría ubicada en una dll separada.

```
class HelloPlugin : IPlugin
{
    public string PluginDescription => "A plugin that says Hello";
    public void DoWork()
    {
        Console.WriteLine("Hello");
    }
}
```

El cargador de complementos de su aplicación encontraría los archivos dll, obtendría todos los tipos en los ensamblajes que implementan `IPlugin` y creará instancias de esos.

```
    public IEnumerable<IPlugin> InstantiatePlugins(string directory)
    {
        var pluginAssemblyNames = Directory.GetFiles(directory, "*.addin.dll").Select(name =>
new FileInfo(name).FullName).ToArray();
        //load the assemblies into the current AppDomain, so we can instantiate the types
later
        foreach (var fileName in pluginAssemblyNames)
            AppDomain.CurrentDomain.Load(File.ReadAllBytes(fileName));
        var assemblies = pluginAssemblyNames.Select(System.Reflection.Assembly.LoadFile);
        var typesInAssembly = assemblies.SelectMany(asm => asm.GetTypes());
        var pluginTypes = typesInAssembly.Where(type => typeof
(IPlugin).IsAssignableFrom(type));
        return pluginTypes.Select(Activator.CreateInstance).Cast<IPlugin>();
    }
```

### Creando una instancia de un tipo

La forma más sencilla es usar la clase `Activator` .

Sin embargo, a pesar de que el rendimiento de `Activator` se ha mejorado desde .NET 3.5, el uso de `Activator.CreateInstance()` es a veces una mala opción, debido a un rendimiento (relativamente) bajo: Test 1 , Test 2 , Test 3 ...

---

## Con clase `Activator`

```
Type type = typeof(BigInteger);
object result = Activator.CreateInstance(type); //Requires parameterless constructor.
Console.WriteLine(result); //Output: 0
result = Activator.CreateInstance(type, 123); //Requires a constructor which can receive an
'int' compatible argument.
Console.WriteLine(result); //Output: 123
```

Puede pasar una matriz de objetos a `Activator.CreateInstance` si tiene más de un parámetro.

```
// With a constructor such as MyClass(int, int, string)
Activator.CreateInstance(typeof(MyClass), new object[] { 1, 2, "Hello World" });

Type type = typeof(someObject);
var instance = Activator.CreateInstance(type);
```

#### Para un tipo genérico

El método `MakeGenericType` convierte un tipo genérico abierto (como `List<>` ) en un tipo concreto (como `List<string>` ) al aplicarle argumentos de tipo.

```
// generic List with no parameters
Type openType = typeof(List<>);

// To create a List<string>
Type[] tArgs = { typeof(string) };
Type target = openType.MakeGenericType(tArgs);

// Create an instance - Activator.CreateInstance will call the default constructor.
// This is equivalent to calling new List<string>().
List<string> result = (List<string>)Activator.CreateInstance(target);
```

La sintaxis de `List<>` no está permitida fuera de una expresión `typeof` .

---

## Sin clase `Activator`

#### Usando `new` palabra clave (servirá para constructores sin parámetros)

```
T GetInstance<T>() where T : new()
{
```

```
    T instance = new T();
    return instance;
}
```

### Utilizando el método Invoke

```
// Get the instance of the desired constructor (here it takes a string as a parameter).
ConstructorInfo c = typeof(T).GetConstructor(new[] { typeof(string) });
// Don't forget to check if such constructor exists
if (c == null)
    throw new InvalidOperationException(string.Format("A constructor for type '{0}' was not
found.", typeof(T)));
T instance = (T)c.Invoke(new object[] { "test" });
```

### Usando árboles de expresión

Los árboles de expresión representan el código en una estructura de datos similar a un árbol, donde cada nodo es una expresión. Como explica MSDN :

> La expresión es una secuencia de uno o más operandos y cero o más operadores que se pueden evaluar a un solo valor, objeto, método o espacio de nombres. Las expresiones pueden consistir en un valor literal, una invocación de método, un operador y sus operandos, o un nombre simple. Los nombres simples pueden ser el nombre de una variable, miembro de tipo, parámetro de método, espacio de nombres o tipo.

```
public class GenericFactory<TKey, TType>
    {
       private readonly Dictionary<TKey, Func<object[], TType>> _registeredTypes; //
dictionary, that holds constructor functions.
       private object _locker = new object(); // object for locking dictionary, to guarantee
thread safety

        public GenericFactory()
        {
            _registeredTypes = new Dictionary<TKey, Func<object[], TType>>();
        }

        /// <summary>
        /// Find and register suitable constructor for type
        /// </summary>
        /// <typeparam name="TType"></typeparam>
        /// <param name="key">Key for this constructor</param>
        /// <param name="parameters">Parameters</param>
        public void Register(TKey key, params Type[] parameters)
        {
            ConstructorInfo ci = typeof(TType).GetConstructor(BindingFlags.Public |
BindingFlags.Instance, null, CallingConventions.HasThis, parameters, new ParameterModifier[] {
}); // Get the instance of ctor.
            if (ci == null)
                throw new InvalidOperationException(string.Format("Constructor for type '{0}'
was not found.", typeof(TType)));

            Func<object[], TType> ctor;

            lock (_locker)
```

```
            {
                if (!_registeredTypes.TryGetValue(key, out ctor)) // check if such ctor
already been registered
                {
                    var pExp = Expression.Parameter(typeof(object[]), "arguments"); // create
parameter Expression
                    var ctorParams = ci.GetParameters(); // get parameter info from
constructor

                    var argExpressions = new Expression[ctorParams.Length]; // array that will
contains parameter expessions
                    for (var i = 0; i < parameters.Length; i++)
                    {

                        var indexedAcccess = Expression.ArrayIndex(pExp,
Expression.Constant(i));

                        if (!parameters[i].IsClass && !parameters[i].IsInterface) // check if
parameter is a value type
                        {
                            var localVariable = Expression.Variable(parameters[i],
"localVariable"); // if so - we should create local variable that will store paraameter value

                            var block = Expression.Block(new[] { localVariable },
                                    Expression.IfThenElse(Expression.Equal(indexedAcccess,
Expression.Constant(null)),
                                        Expression.Assign(localVariable,
Expression.Default(parameters[i])),
                                        Expression.Assign(localVariable,
Expression.Convert(indexedAcccess, parameters[i]))
                                    ),
                                    localVariable
                                );

                            argExpressions[i] = block;

                        }
                        else
                            argExpressions[i] = Expression.Convert(indexedAcccess,
parameters[i]);
                    }
                    var newExpr = Expression.New(ci, argExpressions); // create expression
that represents call to specified ctor with the specified arguments.

                    _registeredTypes.Add(key, Expression.Lambda(newExpr, new[] { pExp
}).Compile() as Func<object[], TType>); // compile expression to create delegate, and add
fucntion to dictionary
                }
            }
        }

        /// <summary>
        /// Returns instance of registered type by key.
        /// </summary>
        /// <typeparam name="TType"></typeparam>
        /// <param name="key"></param>
        /// <param name="args"></param>
        /// <returns></returns>
        public TType Create(TKey key, params object[] args)
        {
            Func<object[], TType> foo;
```

```
            if (_registeredTypes.TryGetValue(key, out foo))
            {
                return (TType)foo(args);
            }

            throw new ArgumentException("No type registered for this key.");
        }
    }
```

Podría ser usado así:

```
 public class TestClass
 {
        public TestClass(string parameter)
        {
            Console.Write(parameter);
        }
 }


public void TestMethod()
{
       var factory = new GenericFactory<string, TestClass>();
       factory.Register("key", typeof(string));
       TestClass newInstance = factory.Create("key", "testParameter");
}
```

**Usando FormatterServices.GetUninitializedObject**

```
T instance = (T)FormatterServices.GetUninitializedObject(typeof(T));
```

En caso de utilizar `FormatterServices.GetUninitializedObject` , no se `FormatterServices.GetUninitializedObject` constructores ni los inicializadores de campo. Está destinado a ser usado en serializadores y motores remotos.

## Obtener un tipo por nombre con espacio de nombres

Para hacer esto necesita una referencia al conjunto que contiene el tipo. Si tiene otro tipo disponible que sabe que está en el mismo ensamblaje que el que desea, puede hacer esto:

```
typeof(KnownType).Assembly.GetType(typeName);
```

- donde `typeName` es el nombre del tipo que está buscando (incluido el espacio de nombres), y `KnownType` es el tipo que sabe que se encuentra en el mismo ensamblaje.

Menos eficiente pero más general es el siguiente:

```
Type t = null;
foreach (Assembly ass in AppDomain.CurrentDomain.GetAssemblies())
{
    if (ass.FullName.StartsWith("System."))
        continue;
    t = ass.GetType(typeName);
```

```
    if (t != null)
        break;
}
```

Observe la marca de verificación para excluir los conjuntos de espacios de nombres del sistema de escaneo para acelerar la búsqueda. Si su tipo puede ser realmente un tipo CLR, tendrá que eliminar estas dos líneas.

Si resulta que tiene el nombre de tipo totalmente calificado para el ensamblaje, incluido el ensamblaje, simplemente puede obtenerlo con

```
Type.GetType(fullyQualifiedName);
```

## Obtenga un delegado de tipo fuerte en un método o propiedad a través de la reflexión

Cuando el rendimiento es una preocupación, invocar un método a través de la reflexión (es decir, a través del método `MethodInfo.Invoke` ) no es ideal. Sin embargo, es relativamente sencillo obtener un delegado con un tipo de ejecución más potente utilizando la función `Delegate.CreateDelegate` . La penalización de rendimiento por usar la reflexión se incurre solo durante el proceso de creación de delegados. Una vez que se crea el delegado, hay poca penalización de rendimiento para invocarlo:

```
// Get a MethodInfo for the Math.Max(int, int) method...
var maxMethod = typeof(Math).GetMethod("Max", new Type[] { typeof(int), typeof(int) });
// Now get a strongly-typed delegate for Math.Max(int, int)...
var stronglyTypedDelegate = (Func<int, int, int>)Delegate.CreateDelegate(typeof(Func<int, int,
int>), null, maxMethod);
// Invoke the Math.Max(int, int) method using the strongly-typed delegate...
Console.WriteLine("Max of 3 and 5 is: {0}", stronglyTypedDelegate(3, 5));
```

Esta técnica también puede extenderse a las propiedades. Si tenemos una clase llamada `MyClass` con una propiedad `int` llamada `MyIntProperty` , el código para obtener un getter de tipo fuerte sería (el siguiente ejemplo asume que 'target' es una instancia válida de `MyClass` ):

```
// Get a MethodInfo for the MyClass.MyIntProperty getter...
var theProperty = typeof(MyClass).GetProperty("MyIntProperty");
var theGetter = theProperty.GetGetMethod();
// Now get a strongly-typed delegate for MyIntProperty that can be executed against any
MyClass instance...
var stronglyTypedGetter = (Func<MyClass, int>)Delegate.CreateDelegate(typeof(Func<MyClass,
int>), theGetter);
// Invoke the MyIntProperty getter against MyClass instance 'target'...
Console.WriteLine("target.MyIntProperty is: {0}", stronglyTypedGetter(target));
```

... y lo mismo se puede hacer para el colocador:

```
// Get a MethodInfo for the MyClass.MyIntProperty setter...
var theProperty = typeof(MyClass).GetProperty("MyIntProperty");
var theSetter = theProperty.GetSetMethod();
// Now get a strongly-typed delegate for MyIntProperty that can be executed against any
```

```
MyClass instance...
var stronglyTypedSetter = (Action<MyClass, int>)Delegate.CreateDelegate(typeof(Action<MyClass,
int>), theSetter);
// Set MyIntProperty to 5...
stronglyTypedSetter(target, 5);
```

Lea Reflexión en línea: https://riptutorial.com/es/csharp/topic/28/reflexion

# Capítulo 144: Regex Parsing

## Sintaxis

- `new Regex(pattern);` *// Crea una nueva instancia con un patrón definido.*
- `Regex.Match(input);` *// Inicia la búsqueda y devuelve el Match.*
- `Regex.Matches(input);` *// Inicia la búsqueda y devuelve una MatchCollection*

## Parámetros

| Nombre | Detalles |
|---|---|
| Modelo | El patrón de `string` que se debe utilizar para la búsqueda. Para más información: msdn |
| RegexOptions *[Opcional]* | Las opciones comunes aquí son `Singleline` y `Multiline` . Están cambiando el comportamiento de los elementos de patrón como el punto (.) Que no cubrirá una `NewLine` (\ n) en `Multiline-Mode` sino en el `Multiline-Mode` de Una `SingleLine-Mode` . Comportamiento por defecto: msdn |
| Tiempo de espera *[Opcional]* | Donde los patrones son cada vez más complejos, la búsqueda puede consumir más tiempo. Este es el tiempo de espera superado para la búsqueda, tal como se conoce en la programación en red. |

## Observaciones

### Necesitado usando

```
using System.Text.RegularExpressions;
```

### Agradable tener

- Puede probar sus patrones en línea sin la necesidad de compilar su solución para obtener resultados aquí: Haga clic en mí
- Regex101 Ejemplo: Click me

---

*Especialmente los principiantes tienden a exagerar sus tareas con expresiones regulares porque se siente poderoso y en el lugar correcto para búsquedas más complejas basadas en texto. Este es el punto en el que las personas intentan analizar documentos xml con expresiones regulares sin siquiera preguntarse si podría haber una clase ya terminada para esta tarea como* `XmlDocument` *.*

*Regex debería ser la última arma para elegir la complejidad. Al menos no olvide esforzarse para*

*buscar el `right way` antes de escribir 20 líneas de patrones.*

# Examples

## Partido individual

*`using System.Text.RegularExpressions;`*

```
string pattern = ":(.*?):";
string lookup = "--:text in here:--";

// Instanciate your regex object and pass a pattern to it
Regex rgxLookup = new Regex(pattern, RegexOptions.Singleline, TimeSpan.FromSeconds(1));
// Get the match from your regex-object
Match mLookup = rgxLookup.Match(lookup);

// The group-index 0 always covers the full pattern.
// Matches inside parentheses will be accessed through the index 1 and above.
string found = mLookup.Groups[1].Value;
```

**Resultado:**

```
found = "text in here"
```

## Múltiples partidos

*`using System.Text.RegularExpressions;`*

```
List<string> found = new List<string>();
string pattern = ":(.*?):";
string lookup = "--:text in here:--:another one:-:third one:---!123:fourth:";

// Instanciate your regex object and pass a pattern to it
Regex rgxLookup = new Regex(pattern, RegexOptions.Singleline, TimeSpan.FromSeconds(1));
MatchCollection mLookup = rgxLookup.Matches(lookup);

foreach(Match match in mLookup)
{
    found.Add(match.Groups[1].Value);
}
```

**Resultado:**

```
found = new List<string>() { "text in here", "another one", "third one", "fourth" }
```

Lea Regex Parsing en línea: https://riptutorial.com/es/csharp/topic/3774/regex-parsing

# Capítulo 145: Resolución de sobrecarga

## Observaciones

El proceso de resolución de sobrecarga se describe en la especificación C # , sección 7.5.3. También son relevantes las secciones 7.5.2 (inferencia de tipo) y 7.6.5 (expresiones de invocación).

La forma en que funciona la resolución de sobrecarga probablemente se modificará en C # 7. Las notas de diseño indican que Microsoft lanzará un nuevo sistema para determinar qué método es mejor (en escenarios complicados).

## Examples

### Ejemplo de sobrecarga básica

Este código contiene un método sobrecargado llamado **Hello** :

```
class Example
{
    public static void Hello(int arg)
    {
        Console.WriteLine("int");
    }

    public static void Hello(double arg)
    {
        Console.WriteLine("double");
    }

    public static void Main(string[] args)
    {
        Hello(0);
        Hello(0.0);
    }
}
```

Cuando se llama al método **principal** , se imprimirá

```
int
double
```

En tiempo de compilación, cuando el compilador encuentra el método llamado `Hello(0)` , encuentra todos los métodos con el nombre `Hello` . En este caso, encuentra dos de ellos. Luego trata de determinar cuál de los métodos es *mejor* . El algoritmo para determinar qué método es mejor es complejo, pero generalmente se reduce a "hacer la menor cantidad posible de conversiones implícitas".

Por lo tanto, en el caso de `Hello(0)` , no se necesita conversión para el método `Hello(int)` pero se

necesita una conversión numérica implícita para el método `Hello(double)` . Así, el primer método es elegido por el compilador.

En el caso de `Hello(0.0)` , no hay forma de convertir `0.0` a `int` implícita, por lo que el método `Hello(int)` ni siquiera se considera para la resolución de sobrecarga. Sólo queda el método y así lo elige el compilador.

### "params" no se expande, a menos que sea necesario.

El siguiente programa:

```
class Program
{
    static void Method(params Object[] objects)
    {
        System.Console.WriteLine(objects.Length);
    }
    static void Method(Object a, Object b)
    {
        System.Console.WriteLine("two");
    }
    static void Main(string[] args)
    {
        object[] objectArray = new object[5];

        Method(objectArray);
        Method(objectArray, objectArray);
        Method(objectArray, objectArray, objectArray);
    }
}
```

imprimirá:

```
5
two
3
```

El `Method(objectArray)` expresión de llamada `Method(objectArray)` se puede interpretar de dos maneras: un solo argumento de `Object` que resulta ser una matriz (por lo que el programa generaría `1` porque ese sería el número de argumentos, o como una matriz de argumentos, dados en la forma normal, como si el método `Method` no tuviera la palabra clave `params` . En estas situaciones, la forma normal no expandida siempre tiene prioridad. Por lo tanto, el programa produce `5` .

En la segunda expresión, `Method(objectArray, objectArray)` tanto la forma expandida del primer método como el segundo método tradicional. También en este caso, las formas no expandidas tienen prioridad, por lo que el programa imprime `two` .

En la tercera expresión, `Method(objectArray, objectArray, objectArray)` , la única opción es usar la forma expandida del primer método, y así se imprime el programa `3` .

### Pasando nulo como uno de los argumentos.

Si usted tiene

```
void F1(MyType1 x) {
    // do something
}

void F1(MyType2 x) {
    // do something else
}
```

y por alguna razón, debe llamar a la primera sobrecarga de `F1` pero con `x = null` , y luego hacerlo simplemente

```
F1(null);
```

No compilará ya que la llamada es ambigua. Para contrarrestar esto puedes hacer

```
F1(null as MyType1);
```

Lea Resolución de sobrecarga en línea: https://riptutorial.com/es/csharp/topic/77/resolucion-de-sobrecarga

# Capítulo 146: Secuencias de escape de cadena

## Sintaxis

- \ '- comilla simple (0x0027)
- \ "- comillas dobles (0x0022)
- \\ - barra invertida (0x005C)
- \ 0 - nulo (0x0000)
- \ a - alerta (0x0007)
- \ b - retroceso (0x0008)
- \ f - alimentación de formulario (0x000C)
- \ n - nueva línea (0x000A)
- \ r - retorno de carro (0x000D)
- \ t - pestaña horizontal (0x0009)
- \ v - pestaña vertical (0x000B)
- \ u0000 - \ uFFFF - Carácter Unicode
- \ x0 - \ xFFFF - Carácter Unicode (código con longitud variable)
- \ U00000000 - \ U0010FFFF - Carácter Unicode (para generar sustitutos)

## Observaciones

Las secuencias de escape de cadena se transforman en el carácter correspondiente en **tiempo de compilación** . Las cadenas normales que contienen barras inclinadas hacia atrás **no se** transforman.

Por ejemplo, las cadenas `notEscaped` y `notEscaped2` continuación no se transforman en un carácter de nueva línea, sino que permanecerán como dos caracteres diferentes ( `'\'` y `'n'` ).

```
string escaped = "\n";
string notEscaped = "\\" + "n";
string notEscaped2 = "\\n";

Console.WriteLine(escaped.Length); // 1
Console.WriteLine(notEscaped.Length); // 2
Console.WriteLine(notEscaped2.Length); // 2
```

## Examples

### Secuencias de escape de caracteres Unicode

```
string sqrt = "\u221A";      // √
string emoji = "\U0001F601"; // 
string text = "\u0022Hello World\u0022"; // "Hello World"
string variableWidth = "\x22Hello World\x22"; // "Hello World"
```

## Escapar de símbolos especiales en literales de personajes.

### Apóstrofes

```
char apostrophe = '\'';
```

### Barra invertida

```
char oneBackslash = '\\';
```

## Escapando símbolos especiales en cadenas literales.

### Barra invertida

```
// The filename will be c:\myfile.txt in both cases
string filename = "c:\\myfile.txt";
string filename = @"c:\myfile.txt";
```

El segundo ejemplo utiliza una cadena literal literal , que no trata la barra invertida como un carácter de escape.

### Citas

```
string text = "\"Hello World!\", said the quick brown fox.";
string verbatimText = @"""Hello World!"", said the quick brown fox.";
```

Ambas variables contendrán el mismo texto.

> "¡Hola mundo!", Dijo el rápido zorro marrón.

### Nuevas lineas

Los literales de cadenas verbales pueden contener nuevas líneas:

```
string text = "Hello\r\nWorld!";
string verbatimText = @"Hello
World!";
```

Ambas variables contendrán el mismo texto.

## Las secuencias de escape no reconocidas producen errores en tiempo de compilación

Los siguientes ejemplos no se compilarán:

```
string s = "\c";
char c = '\c';
```

En su lugar, producirán el error `Unrecognized escape sequence` en el momento de la compilación.

### Usando secuencias de escape en identificadores

Secuencias de escape no se restringen a `string` y `char` literales.

Supongamos que necesita anular un método de terceros:

```
protected abstract IEnumerable<Texte> ObtenirŒuvres();
```

y suponga que el carácter `Œ` no está disponible en la codificación de caracteres que utiliza para sus archivos fuente de C #. Tiene suerte, está permitido usar escapes del tipo `\u####` o `\U########` en los ***identificadores*** del código. Entonces es legal escribir:

```
protected override IEnumerable<Texte> Obtenir\u0152uvres()
{
    // ...
}
```

y el compilador de C # sabrá que `Œ` y `\u0152` son el mismo carácter.

(Sin embargo, podría ser una buena idea cambiar a UTF-8 o una codificación similar que pueda manejar todos los caracteres).

Lea Secuencias de escape de cadena en línea:
https://riptutorial.com/es/csharp/topic/39/secuencias-de-escape-de-cadena

# Capítulo 147: Serialización binaria

## Observaciones

El motor de serialización binario es parte del marco .NET, pero los ejemplos que se dan aquí son específicos de C #. En comparación con otros motores de serialización integrados en el marco .NET, el serializador binario es rápido y eficiente y, por lo general, requiere muy poco código adicional para que funcione. Sin embargo, también es menos tolerante a los cambios de código; es decir, si serializa un objeto y luego realiza un ligero cambio en la definición del objeto, es probable que no se deserialice correctamente.

## Examples

### Haciendo un objeto serializable

Agregue el atributo `[Serializable]` para marcar un objeto completo para la serialización binaria:

```
[Serializable]
public class Vector
{
    public int X;
    public int Y;
    public int Z;

    [NonSerialized]
    public decimal DontSerializeThis;

    [OptionalField]
    public string Name;
}
```

Todos los miembros se serializarán a menos que `[NonSerialized]` explícitamente con el atributo `[NonSerialized]` . En nuestro ejemplo, `X` , `Y` , `Z` y `Name` están todos serializados.

Se requiere que todos los miembros estén presentes en la deserialización a menos que estén marcados con `[NonSerialized]` o `[OptionalField] [NonSerialized] [OptionalField]` . En nuestro ejemplo, `X` , `Y` y `Z` son todos necesarios y la deserialización fallará si no están presentes en el flujo. `DontSerializeThis` siempre se establecerá en `default(decimal)` (que es 0). Si el `Name` está presente en la ruta, entonces se establecerá en ese valor, de lo contrario se establecerá en `default(string)` (que es nulo). El propósito de `[OptionalField]` es proporcionar un poco de tolerancia de versión.

### Controlando el comportamiento de serialización con atributos.

Si usa el atributo `[NonSerialized]` , entonces ese miembro siempre tendrá su valor predeterminado después de la deserialización (por ejemplo, 0 para un `int` , nulo para `string` , falso para un `bool` , etc.), independientemente de cualquier inicialización realizada en el propio objeto. (constructores, declaraciones, etc.). Para compensar, se proporcionan los atributos `[OnDeserializing]` (llamado

solo ANTES de deserializar) y `[OnDeserialized]` (llamado simplemente DESPUÉS de deserializar) junto con sus homólogos, `[OnSerializing]` y `[OnSerialized]` .

Supongamos que queremos agregar una "Clasificación" a nuestro Vector y queremos asegurarnos de que el valor siempre comience en 1. De la forma en que está escrito a continuación, será 0 después de ser deserializado:

```
[Serializable]
public class Vector
{
    public int X;
    public int Y;
    public int Z;

    [NonSerialized]
    public decimal Rating = 1M;

    public Vector()
    {
        Rating = 1M;
    }

    public Vector(decimal initialRating)
    {
        Rating = initialRating;
    }
}
```

Para solucionar este problema, simplemente podemos agregar el siguiente método dentro de la clase para establecerlo en 1:

```
[OnDeserializing]
void OnDeserializing(StreamingContext context)
{
    Rating = 1M;
}
```

O, si queremos establecerlo en un valor calculado, podemos esperar a que finalice la deserialización y luego establecerlo:

```
[OnDeserialized]
void OnDeserialized(StreamingContext context)
{
    Rating = 1 + ((X+Y+Z)/3);
}
```

De manera similar, podemos controlar cómo se escriben las cosas usando `[OnSerializing]` y `[OnSerialized]` .

## Añadiendo más control implementando ISerializable.

Eso obtendría más control sobre la serialización, cómo guardar y cargar tipos

Implementar la interfaz ISerializable y crear un constructor vacío para compilar

```
[Serializable]
public class Item : ISerializable
{
    private string _name;

    public string Name
    {
        get { return _name; }
        set { _name = value; }
    }

    public Item ()
    {

    }

    protected Item (SerializationInfo info, StreamingContext context)
    {
        _name = (string)info.GetValue("_name", typeof(string));
    }

    public void GetObjectData(SerializationInfo info, StreamingContext context)
    {
        info.AddValue("_name", _name, typeof(string));
    }
}
```

Para la serialización de datos, puede especificar el nombre deseado y el tipo deseado

```
info.AddValue("_name", _name, typeof(string));
```

Cuando se deserialicen los datos, podrá leer el tipo deseado

```
_name = (string)info.GetValue("_name", typeof(string));
```

## Sustitutos de serialización (Implementando ISerializationSurrogate)

Implementa un selector sustituto de serialización que permite que un objeto realice la serialización y deserialización de otro

También permite serializar o deserializar correctamente una clase que no es serializable en sí misma

Implementar la interfaz ISerializationSurrogate

```
public class ItemSurrogate : ISerializationSurrogate
{
    public void GetObjectData(object obj, SerializationInfo info, StreamingContext context)
    {
        var item = (Item)obj;
        info.AddValue("_name", item.Name);
    }

    public object SetObjectData(object obj, SerializationInfo info, StreamingContext context,
ISurrogateSelector selector)
```

```
    {
        var item = (Item)obj;
        item.Name = (string)info.GetValue("_name", typeof(string));
        return item;
    }
}
```

Luego, debe informar a su IFormatter sobre los sustitutos definiendo e inicializando un SurrogateSelector y asignándolo a su IFormatter.

```
var surrogateSelector = new SurrogateSelector();
surrogateSelector.AddSurrogate(typeof(Item), new StreamingContext(StreamingContextStates.All),
new ItemSurrogate());
var binaryFormatter = new BinaryFormatter
{
    SurrogateSelector = surrogateSelector
};
```

Incluso si la clase no está marcada como serializable.

```
//this class is not serializable
public class Item
{
    private string _name;

    public string Name
    {
        get { return _name; }
        set { _name = value; }
    }
}
```

La solucion completa

```
using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;

namespace BinarySerializationExample
{
    class Item
    {
        private string _name;

        public string Name
        {
            get { return _name; }
            set { _name = value; }
        }
    }

    class ItemSurrogate : ISerializationSurrogate
    {
        public void GetObjectData(object obj, SerializationInfo info, StreamingContext
context)
        {
```

```
            var item = (Item)obj;
            info.AddValue("_name", item.Name);
        }

        public object SetObjectData(object obj, SerializationInfo info, StreamingContext
context, ISurrogateSelector selector)
        {
            var item = (Item)obj;
            item.Name = (string)info.GetValue("_name", typeof(string));
            return item;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            var item = new Item
            {
                Name = "Orange"
            };

            var bytes = SerializeData(item);
            var deserializedData = (Item)DeserializeData(bytes);
        }

        private static byte[] SerializeData(object obj)
        {
            var surrogateSelector = new SurrogateSelector();
            surrogateSelector.AddSurrogate(typeof(Item), new
StreamingContext(StreamingContextStates.All), new ItemSurrogate());

            var binaryFormatter = new BinaryFormatter
            {
                SurrogateSelector = surrogateSelector
            };

            using (var memoryStream = new MemoryStream())
            {
                binaryFormatter.Serialize(memoryStream, obj);
                return memoryStream.ToArray();
            }
        }

        private static object DeserializeData(byte[] bytes)
        {
            var surrogateSelector = new SurrogateSelector();
            surrogateSelector.AddSurrogate(typeof(Item), new
StreamingContext(StreamingContextStates.All), new ItemSurrogate());

            var binaryFormatter = new BinaryFormatter
            {
                SurrogateSelector = surrogateSelector
            };

            using (var memoryStream = new MemoryStream(bytes))
                return binaryFormatter.Deserialize(memoryStream);
        }
    }
}
```

## Carpeta de serialización

La carpeta le brinda la oportunidad de inspeccionar qué tipos se están cargando en su dominio de aplicación

Crear una clase heredada de SerializationBinder

```
class MyBinder : SerializationBinder
{
    public override Type BindToType(string assemblyName, string typeName)
    {
        if (typeName.Equals("BinarySerializationExample.Item"))
            return typeof(Item);
        return null;
    }
}
```

Ahora podemos verificar qué tipos se están cargando y sobre esta base para decidir qué es lo que realmente queremos recibir

Para usar un cuaderno, debe agregarlo a BinaryFormatter.

```
object DeserializeData(byte[] bytes)
{
    var binaryFormatter = new BinaryFormatter();
    binaryFormatter.Binder = new MyBinder();

    using (var memoryStream = new MemoryStream(bytes))
        return binaryFormatter.Deserialize(memoryStream);
}
```

La solucion completa

```
using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;

namespace BinarySerializationExample
{
    class MyBinder : SerializationBinder
    {
        public override Type BindToType(string assemblyName, string typeName)
        {
            if (typeName.Equals("BinarySerializationExample.Item"))
                return typeof(Item);
            return null;
        }
    }

    [Serializable]
    public class Item
    {
        private string _name;

        public string Name
```

```
        {
            get { return _name; }
            set { _name = value; }
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            var item = new Item
            {
                Name = "Orange"
            };

            var bytes = SerializeData(item);
            var deserializedData = (Item)DeserializeData(bytes);
        }

        private static byte[] SerializeData(object obj)
        {
            var binaryFormatter = new BinaryFormatter();
            using (var memoryStream = new MemoryStream())
            {
                binaryFormatter.Serialize(memoryStream, obj);
                return memoryStream.ToArray();
            }
        }

        private static object DeserializeData(byte[] bytes)
        {
            var binaryFormatter = new BinaryFormatter
            {
                Binder = new MyBinder()
            };

            using (var memoryStream = new MemoryStream(bytes))
                return binaryFormatter.Deserialize(memoryStream);
        }
    }
}
```

## Algunos errores en la compatibilidad hacia atrás.

Este pequeño ejemplo muestra cómo puede perder la compatibilidad con versiones anteriores en sus programas si no se cuida con anticipación sobre esto. Y formas de obtener más control del proceso de serialización.

Al principio, escribiremos un ejemplo de la primera versión del programa:

Versión 1

```
[Serializable]
class Data
{
    [OptionalField]
    private int _version;
```

```
    public int Version
    {
        get { return _version; }
        set { _version = value; }
    }
}
```

Y ahora, asumamos que en la segunda versión del programa se agregó una nueva clase. Y tenemos que almacenarlo en una matriz.

Ahora el código se verá así:

Versión 2

```
[Serializable]
class NewItem
{
    [OptionalField]
    private string _name;

    public string Name
    {
        get { return _name; }
        set { _name = value; }
    }
}

[Serializable]
class Data
{
    [OptionalField]
    private int _version;

    public int Version
    {
        get { return _version; }
        set { _version = value; }
    }

    [OptionalField]
    private List<NewItem> _newItems;

    public List<NewItem> NewItems
    {
        get { return _newItems; }
        set { _newItems = value; }
    }
}
```

Y código para serializar y deserializar.

```
private static byte[] SerializeData(object obj)
{
    var binaryFormatter = new BinaryFormatter();
    using (var memoryStream = new MemoryStream())
    {
        binaryFormatter.Serialize(memoryStream, obj);
        return memoryStream.ToArray();
```

```
    }
}

private static object DeserializeData(byte[] bytes)
{
    var binaryFormatter = new BinaryFormatter();
    using (var memoryStream = new MemoryStream(bytes))
        return binaryFormatter.Deserialize(memoryStream);
}
```

Y así, ¿qué pasaría si serializas los datos en el programa de v2 y tratas de deserializarlos en el programa de v1?

Obtienes una excepción:

```
System.Runtime.Serialization.SerializationException was unhandled
Message=The ObjectManager found an invalid number of fixups. This usually indicates a problem
in the Formatter.Source=mscorlib
StackTrace:
   at System.Runtime.Serialization.ObjectManager.DoFixups()
   at System.Runtime.Serialization.Formatters.Binary.ObjectReader.Deserialize(HeaderHandler
handler, __BinaryParser serParser, Boolean fCheck, Boolean isCrossAppDomain,
IMethodCallMessage methodCallMessage)
   at System.Runtime.Serialization.Formatters.Binary.BinaryFormatter.Deserialize(Stream
serializationStream, HeaderHandler handler, Boolean fCheck, Boolean isCrossAppDomain,
IMethodCallMessage methodCallMessage)
   at System.Runtime.Serialization.Formatters.Binary.BinaryFormatter.Deserialize(Stream
serializationStream)
   at Microsoft.Samples.TestV1.Main(String[] args) in c:\Users\andrew\Documents\Visual Studio
2013\Projects\vts\CS\V1 Application\TestV1Part2\TestV1Part2.cs:line 29
   at System.AppDomain._nExecuteAssembly(Assembly assembly, String[] args)
   at Microsoft.VisualStudio.HostingProcess.HostProc.RunUsersAssembly()
   at System.Threading.ExecutionContext.Run(ExecutionContext executionContext, ContextCallback
callback, Object state)
   at System.Threading.ThreadHelper.ThreadStart()
```

¿Por qué?

El ObjectManager tiene una lógica diferente para resolver dependencias para matrices y para tipos de referencia y valor. Agregamos una serie de nuevos tipos de referencia que están ausentes en nuestro ensamblaje.

Cuando ObjectManager intenta resolver dependencias, construye el gráfico. Cuando ve la matriz, no puede corregirla inmediatamente, por lo que crea una referencia ficticia y luego corrige la matriz más tarde.

Y dado que este tipo no está en el ensamblaje y las dependencias no se pueden arreglar. Por alguna razón, no elimina la matriz de la lista de elementos para los arreglos y, al final, lanza una excepción "IncorrectNumberOfFixups".

Se trata de algunas 'trampas' en el proceso de serialización. Por alguna razón, no funciona correctamente solo para matrices de nuevos tipos de referencia.

```
A Note:
```

```
Similar code will work correctly if you do not use arrays with new classes
```

¿Y la primera forma de solucionarlo y mantener la compatibilidad?

- Use una colección de nuevas estructuras en lugar de clases o use un diccionario (clases posibles), porque un diccionario es una colección de pares de valores-clave (su estructura)
- Use ISerializable, si no puede cambiar el código anterior

Lea Serialización binaria en línea: https://riptutorial.com/es/csharp/topic/4120/serializacion-binaria

# Capítulo 148: StringBuilder

## Examples

### Qué es un StringBuilder y cuándo usar uno

Un `StringBuilder` representa una serie de caracteres que, a diferencia de una cadena normal, son mutables. Muchas veces es necesario modificar las cadenas que ya hemos creado, pero el objeto de cadena estándar no es mutable. Esto significa que cada vez que se modifica una cadena, un nuevo objeto de cadena debe crearse, copiarse y luego reasignarse.

```
string myString = "Apples";
mystring += " are my favorite fruit";
```

En el ejemplo anterior, `myString` inicialmente solo tiene el valor `"Apples"` . Sin embargo, cuando concatenamos "son mi fruta favorita", lo que la clase de cuerdas necesita hacer internamente implica:

- Creando una nueva matriz de caracteres igual a la longitud de `myString` y la nueva cadena que estamos agregando.
- Copiando todos los caracteres de `myString` en el comienzo de nuestra nueva matriz y copiando la nueva cadena en el final de la matriz.
- Cree un nuevo objeto de cadena en la memoria y `myString` a `myString` a `myString` .

Para una sola concatenación, esto es relativamente trivial. Sin embargo, ¿qué sucede si es necesario realizar muchas operaciones de adición, por ejemplo, en un bucle?

```
String myString = "";
for (int i = 0; i < 10000; i++)
    myString += " "; // puts 10,000 spaces into our string
```

Debido a la copia repetida y la creación de objetos, esto degradará significativamente el rendimiento de nuestro programa. Podemos evitar esto utilizando en su lugar un `StringBuilder` .

```
StringBuilder myStringBuilder = new StringBuilder();
for (int i = 0; i < 10000; i++)
    myStringBuilder.Append(' ');
```

Ahora, cuando se ejecuta el mismo bucle, el rendimiento y la velocidad del tiempo de ejecución del programa serán significativamente más rápidos que usar una cadena normal. Para hacer que `StringBuilder` vuelva a ser una cadena normal, simplemente podemos llamar al método `ToString()` de `StringBuilder` .

---

Sin embargo, esta no es la única optimización que tiene `StringBuilder` . Para optimizar aún más las funciones, podemos aprovechar otras propiedades que ayudan a mejorar el rendimiento.

```
StringBuilder sb = new StringBuilder(10000); // initializes the capacity to 10000
```

Si sabemos de antemano cuánto tiempo debe `StringBuilder` nuestro `StringBuilder` , podemos especificar su tamaño antes de tiempo, lo que evitará que tenga que cambiar el tamaño de la matriz de caracteres que tiene internamente.

```
sb.Append('k', 2000);
```

Aunque el uso de `StringBuilder` para añadir es mucho más rápido que una cadena, puede ejecutarse incluso más rápido si solo necesita agregar un solo carácter muchas veces.

Una vez que haya completado la construcción de su cadena, puede usar el método `ToString()` en el `StringBuilder` para convertirla en una `string` básica. Esto suele ser necesario porque la clase `StringBuilder` no hereda de la `string` .

Por ejemplo, aquí es cómo puede usar un `StringBuilder` para crear una `string` :

```
string RepeatCharacterTimes(char character, int times)
{
    StringBuilder builder = new StringBuilder("");
    for (int counter = 0; counter < times; counter++)
    {
        //Append one instance of the character to the StringBuilder.
        builder.Append(character);
    }
    //Convert the result to string and return it.
    return builder.ToString();
}
```

---

En conclusión, `StringBuilder` debe usar en lugar de una cadena cuando se deben hacer muchas modificaciones a una cadena teniendo en cuenta el rendimiento.

## Utilice StringBuilder para crear cadenas a partir de una gran cantidad de registros

```
public string GetCustomerNamesCsv()
{
    List<CustomerData> customerDataRecords = GetCustomerData(); // Returns a large number of
records, say, 10000+

    StringBuilder customerNamesCsv = new StringBuilder();
    foreach (CustomerData record in customerDataRecords)
    {
       customerNamesCsv
           .Append(record.LastName)
           .Append(',')
           .Append(record.FirstName)
           .Append(Environment.NewLine);
    }

    return customerNamesCsv.ToString();
}
```

Lea StringBuilder en línea: https://riptutorial.com/es/csharp/topic/4675/stringbuilder

# Capítulo 149: System.DirectoryServices.Protocols.LdapConnec

## Examples

### La conexión LDAP SSL autenticada, el certificado SSL no coincide con el DNS inverso

Configure algunas constantes para el servidor y la información de autenticación. Suponiendo LDAPv3, pero es bastante fácil cambiar eso.

```
// Authentication, and the name of the server.
private const string LDAPUser =
"cn=example:app:mygroup:accts,ou=Applications,dc=example,dc=com";
private readonly char[] password = { 'p', 'a', 's', 's', 'w', 'o', 'r', 'd' };
private const string TargetServer = "ldap.example.com";

// Specific to your company. Might start "cn=manager" instead of "ou=people", for example.
private const string CompanyDN = "ou=people,dc=example,dc=com";
```

En realidad, cree la conexión con tres partes: un LdapDirectoryIdentifier (el servidor) y NetworkCredentials.

```
// Configure server and port. LDAP w/ SSL, aka LDAPS, uses port 636.
// If you don't have SSL, don't give it the SSL port.
LdapDirectoryIdentifier identifier = new LdapDirectoryIdentifier(TargetServer, 636);

// Configure network credentials (userid and password)
var secureString = new SecureString();
foreach (var character in password)
        secureString.AppendChar(character);
NetworkCredential creds = new NetworkCredential(LDAPUser, secureString);

// Actually create the connection
LdapConnection connection = new LdapConnection(identifier, creds)
{
    AuthType = AuthType.Basic,
    SessionOptions =
    {
        ProtocolVersion = 3,
        SecureSocketLayer = true
    }
};

// Override SChannel reverse DNS lookup.
// This gets us past the "The LDAP server is unavailable." exception
// Could be
//    connection.SessionOptions.VerifyServerCertificate += { return true; };
// but some certificate validation is probably good.
connection.SessionOptions.VerifyServerCertificate +=
    (sender, certificate) => certificate.Subject.Contains(string.Format("CN={0},",
TargetServer));
```

Utilice el servidor LDAP, por ejemplo, busque a alguien por ID de usuario para todos los valores de clase de objeto. ObjectClass está presente para demostrar una búsqueda compuesta: el símbolo "es" y el operador booleano "y" para las dos cláusulas de consulta.

```
 SearchRequest searchRequest = new SearchRequest(
        CompanyDN,
        string.Format((&(objectClass=*)(uid={0})), uid),
        SearchScope.Subtree,
        null
);

// Look at your results
foreach (SearchResultEntry entry in searchResponse.Entries) {
    // do something
}
```

## Super simple anónimo LDAP

Suponiendo LDAPv3, pero es bastante fácil cambiar eso. Esta es una creación LDAPv3 LdapConnection anónima, sin cifrar.

```
private const string TargetServer = "ldap.example.com";
```

En realidad, cree la conexión con tres partes: un LdapDirectoryIdentifier (el servidor) y NetworkCredentials.

```
// Configure server and credentials
LdapDirectoryIdentifier identifier = new LdapDirectoryIdentifier(TargetServer);
NetworkCredential creds = new NetworkCredential();
LdapConnection connection = new LdapConnection(identifier, creds)
{
    AuthType=AuthType.Anonymous,
    SessionOptions =
    {
        ProtocolVersion = 3
    }
};
```

Para usar la conexión, algo como esto haría que las personas con el apellido Smith

```
SearchRequest searchRequest = new SearchRequest("dn=example,dn=com", "(sn=Smith)",
SearchScope.Subtree,null);
```

Lea System.DirectoryServices.Protocols.LdapConnection en línea:
https://riptutorial.com/es/csharp/topic/5177/system-directoryservices-protocols-ldapconnection

# Capítulo 150: Temporizadores

## Sintaxis

- `myTimer.Interval` : establece la frecuencia con la que se llama al evento "Tick" (en milisegundos)
- `myTimer.Enabled` : valor booleano que establece que el temporizador se habilite / deshabilite
- `myTimer.Start()` : inicia el temporizador.
- `myTimer.Stop()` : detiene el temporizador.

## Observaciones

Si usa Visual Studio, los temporizadores se pueden agregar como control directamente a su formulario desde la caja de herramientas.

## Examples

### Temporizadores multiproceso

`System.Threading.Timer` - El temporizador multihilo más simple. Contiene dos métodos y un constructor.

Ejemplo: un temporizador llama al método DataWrite, que escribe "multihilo ejecutado ..." después de que hayan transcurrido cinco segundos, y luego cada segundo después de eso hasta que el usuario presiona Intro:

```
using System;
using System.Threading;
class Program
{
  static void Main()
  {
    // First interval = 5000ms; subsequent intervals = 1000ms
    Timer timer = new Timer (DataWrite, "multithread executed...", 5000, 1000);
    Console.ReadLine();
    timer.Dispose(); // This both stops the timer and cleans up.
  }

  static void DataWrite (object data)
  {
    // This runs on a pooled thread
    Console.WriteLine (data); // Writes "multithread executed..."
  }
}
```

Nota: publicará una sección separada para disponer de temporizadores multiproceso.

`Change` : este método puede llamarse cuando desee cambiar el intervalo del temporizador.

`Timeout.Infinite` - Si quieres disparar solo una vez. Especifique esto en el último argumento del constructor.

`System.Timers` - Otra clase de temporizador proporcionada por .NET Framework. Envuelve el `System.Threading.Timer` .

## caracteristicas:

- `IComponent` : permitir que se ubique en la bandeja de componentes del Diseñador de Visual Studio
- Propiedad de `Interval` lugar de un método de `Change`
- `event Elapsed` lugar de un `delegate` devolución de llamada
- Propiedad `Enabled` para iniciar y detener el temporizador ( `default value = false` )
- Métodos de `Start` y `Stop` en caso de que se confunda con la propiedad `Enabled` (punto anterior)
- `AutoReset` : para indicar un evento recurrente ( `default value = true` )
- Propiedad `SynchronizingObject` con los métodos `Invoke` y `BeginInvoke` para llamar de forma segura a los elementos de WPF y los controles de Windows Forms

Ejemplo que representa todas las características anteriores:

```
using System;
using System.Timers; // Timers namespace rather than Threading
class SystemTimer
{
  static void Main()
  {
    Timer timer = new Timer(); // Doesn't require any args
    timer.Interval = 500;
    timer.Elapsed += timer_Elapsed; // Uses an event instead of a delegate
    timer.Start(); // Start the timer
    Console.ReadLine();
    timer.Stop(); // Stop the timer
    Console.ReadLine();
    timer.Start(); // Restart the timer
    Console.ReadLine();
    timer.Dispose(); // Permanently stop the timer
 }

 static void timer_Elapsed(object sender, EventArgs e)
 {
   Console.WriteLine ("Tick");
 }
}
```

`Multithreaded timers` : use el grupo de subprocesos para permitir que algunos subprocesos sirvan para muchos temporizadores. Significa que el método de devolución de llamada o `Elapsed` evento `Elapsed` pueden activarse en un hilo diferente cada vez que se llama.

`Elapsed` : este evento siempre se activa a tiempo, independientemente de si el evento `Elapsed` anterior terminó de ejecutarse. Debido a esto, las devoluciones de llamada o los controladores de eventos deben ser seguros para subprocesos. La precisión de los temporizadores multiproceso

depende del sistema operativo, y suele ser de 10 a 20 ms.

`interop` : cuando necesite una mayor precisión, use esto y llame al temporizador multimedia de Windows. Esto tiene una precisión de hasta 1 ms y se define en `winmm.dll` .

`timeBeginPeriod` : llame a este primero para informar al sistema operativo que necesita una alta precisión de sincronización

`timeSetEvent` : llame a esto después de `timeBeginPeriod` para iniciar un temporizador multimedia.

`timeKillEvent` : llama a esto cuando hayas terminado, esto detiene el temporizador

`timeEndPeriod` : llame a esto para informar al sistema operativo que ya no necesita una alta precisión de tiempo.

Puede encontrar ejemplos completos en Internet que utilizan el temporizador multimedia buscando las palabras clave `dllimport winmm.dll timesetevent` .

## Creando una instancia de un temporizador

Los temporizadores se utilizan para realizar tareas en intervalos de tiempo específicos (hacer X cada Y segundos). A continuación se muestra un ejemplo de cómo crear una nueva instancia de un temporizador.

**NOTA** : Esto se aplica a los temporizadores que usan WinForms. Si usa WPF, puede querer mirar `DispatcherTimer`

```
    using System.Windows.Forms; //Timers use the Windows.Forms namespace

    public partial class Form1 : Form
    {

        Timer myTimer = new Timer(); //create an instance of Timer named myTimer


        public Form1()
        {
            InitializeComponent();
        }

    }
```

## Asignación del controlador de eventos "Tick" a un temporizador

Todas las acciones realizadas en un temporizador se manejan en el evento "Tick".

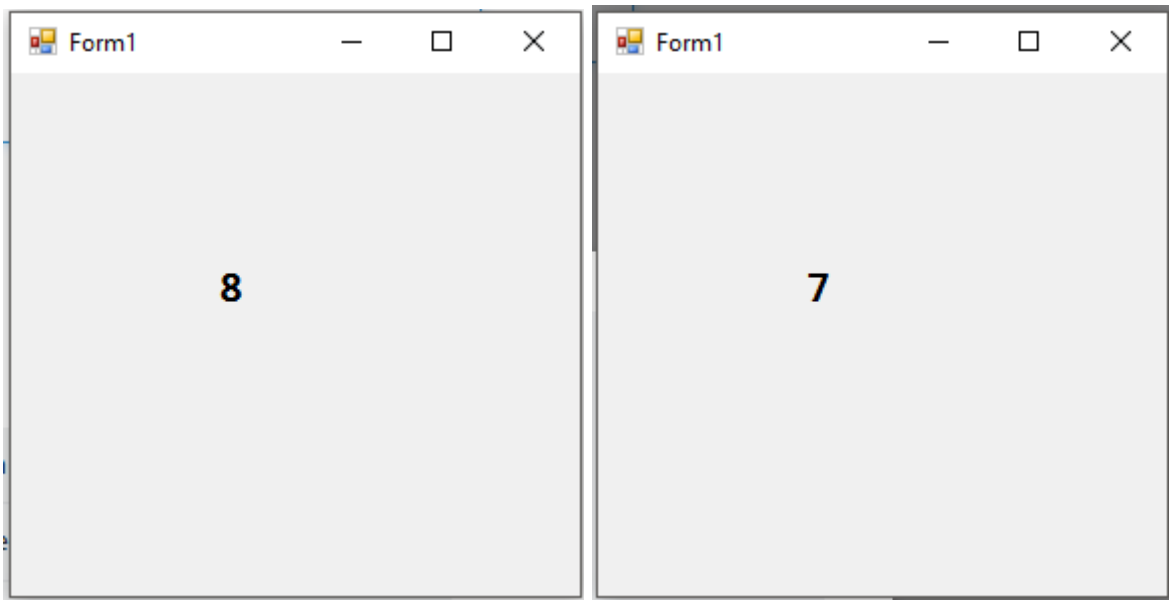
```
public partial class Form1 : Form
{

    Timer myTimer = new Timer();


    public Form1()
```

```
    {
        InitializeComponent();

        myTimer.Tick += myTimer_Tick; //assign the event handler named "myTimer_Tick"
    }

    private void myTimer_Tick(object sender, EventArgs e)
    {
        // Perform your actions here.
    }
}
```

## Ejemplo: usar un temporizador para realizar una cuenta regresiva simple.

```
    public partial class Form1 : Form
    {

    Timer myTimer = new Timer();
    int timeLeft = 10;

        public Form1()
        {
            InitializeComponent();

            //set properties for the Timer
            myTimer.Interval = 1000;
            myTimer.Enabled = true;

            //Set the event handler for the timer, named "myTimer_Tick"
            myTimer.Tick += myTimer_Tick;

            //Start the timer as soon as the form is loaded
            myTimer.Start();

            //Show the time set in the "timeLeft" variable
            lblCountDown.Text = timeLeft.ToString();

        }

        private void myTimer_Tick(object sender, EventArgs e)
        {
            //perform these actions at the interval set in the properties.
            lblCountDown.Text = timeLeft.ToString();
            timeLeft -= 1;

            if (timeLeft < 0)
            {
                myTimer.Stop();
            }
        }
    }
```

Resultados en ...




Y así...

Lea Temporizadores en línea: https://riptutorial.com/es/csharp/topic/3829/temporizadores

# Capítulo 151: Tipo de conversión

## Observaciones

La conversión de tipos es convertir un tipo de datos a otro tipo. También se le conoce como Casting de Tipo. En C #, el casting de tipos tiene dos formas:

**Conversión de tipo implícita** : C # realiza estas conversiones de una manera segura para el tipo. Por ejemplo, son conversiones de tipos integrales de menor a mayor y conversiones de clases derivadas a clases base.

**Conversión explícita de tipo** : estas conversiones las realizan explícitamente los usuarios que utilizan las funciones predefinidas. Las conversiones explícitas requieren un operador de cast.

## Examples

### Ejemplo de operador implícito de MSDN

```
class Digit
{
    public Digit(double d) { val = d; }
    public double val;

    // User-defined conversion from Digit to double
    public static implicit operator double(Digit d)
    {
        Console.WriteLine("Digit to double implict conversion called");
        return d.val;
    }
    //  User-defined conversion from double to Digit
    public static implicit operator Digit(double d)
    {
        Console.WriteLine("double to Digit implict conversion called");
        return new Digit(d);
    }
}

class Program
{
    static void Main(string[] args)
    {
        Digit dig = new Digit(7);
        //This call invokes the implicit "double" operator
        double num = dig;
        //This call invokes the implicit "Digit" operator
        Digit dig2 = 12;
        Console.WriteLine("num = {0} dig2 = {1}", num, dig2.val);
        Console.ReadLine();
    }
}
```

**Salida:**

Dígito para duplicar la conversión implícita llamada
doble a la conversión implícita de dígitos llamada
num = 7 dig2 = 12

Demo en vivo en .NET Fiddle

## Conversión explícita de tipos

```
using System;
namespace TypeConversionApplication
{
   class ExplicitConversion
   {
      static void Main(string[] args)
      {
         double d = 5673.74;
         int i;

         // cast double to int.
         i = (int)d;
         Console.WriteLine(i);
         Console.ReadKey();
      }
   }
}
```

Lea Tipo de conversión en línea: https://riptutorial.com/es/csharp/topic/3489/tipo-de-conversion

# Capítulo 152: Tipo de valor vs tipo de referencia

## Sintaxis

- Pasando por referencia: public void Double (ref int numberToDouble) {}

## Observaciones

## Introducción

### Tipos de valor

Los tipos de valor son los más simples de los dos. Los tipos de valor se utilizan a menudo para representar los datos en sí. Un entero, un booleano o un punto en el espacio 3D son ejemplos de tipos de buen valor.

Los tipos de valor (estructuras) se declaran utilizando la palabra clave struct. Consulte la sección de sintaxis para ver un ejemplo de cómo declarar una nueva estructura.

En términos generales, tenemos 2 palabras clave que se utilizan para declarar tipos de valor:

- Estructuras
- Enumeraciones

### Tipos de referencia

Los tipos de referencia son un poco más complejos. Los tipos de referencia son objetos tradicionales en el sentido de programación orientada a objetos. Por lo tanto, admiten la herencia (y los beneficios de la misma) y también admiten los finalizadores.

En C # generalmente tenemos estos tipos de referencia:

- Las clases
- Delegados
- Interfaces

Los nuevos tipos de referencia (clases) se declaran usando la palabra clave class. Para ver un ejemplo, consulte la sección de sintaxis para saber cómo declarar un nuevo tipo de referencia.

## Grandes diferencias

Las principales diferencias entre los tipos de referencia y los tipos de valor se pueden ver a continuación.

### Los tipos de valor existen en la pila, los tipos de referencia existen en el montón

Esta es la diferencia que se menciona a menudo entre los dos, pero en realidad, a lo que se reduce es que cuando se usa un tipo de valor en C #, como un int, el programa usará esa variable para referirse directamente a ese valor. Si dice que int mine = 0, entonces la variable mine se refiere directamente a 0, lo cual es eficiente. Sin embargo, los tipos de referencia realmente mantienen (como sugiere su nombre) una referencia al objeto subyacente, esto es similar a los punteros en otros lenguajes como C ++.

Puede que no note los efectos de esto inmediatamente, pero los efectos están ahí, son poderosos y sutiles. Consulte el ejemplo sobre cómo cambiar los tipos de referencia en otro lugar para ver un ejemplo.

Esta diferencia es la razón principal de las siguientes diferencias y vale la pena conocerla.

### Los tipos de valor no cambian cuando los cambia en un método, los tipos de referencia sí cambian

Cuando se pasa un tipo de valor a un método como un parámetro, si el método cambia el valor de alguna manera, el valor no se cambia. Por el contrario, pasar un tipo de referencia a ese mismo método y cambiarlo cambiará el objeto subyacente, de modo que otras cosas que usen ese mismo objeto tendrán el objeto recién cambiado en lugar de su valor original.

Vea el ejemplo de los tipos de valor frente a los tipos de referencia en los métodos para obtener más información.

¿Qué pasa si quiero cambiarlos?

Simplemente páselos a su método usando la palabra clave "ref", y luego está pasando este objeto por referencia. Es decir, es el mismo objeto en la memoria. Así que las modificaciones que hagas serán respetadas. Ver el ejemplo en pasar por referencia para un ejemplo.

### Los tipos de valor no pueden ser nulos, los tipos de referencia pueden

Casi como dice, puede asignar nulo a un tipo de referencia, lo que significa que la variable que ha asignado no puede tener ningún objeto real asignado. En el caso de los tipos de valor, sin embargo, esto no es posible. Sin embargo, puede usar Nullable para permitir que su tipo de valor sea anulable, si este es un requisito, aunque si esto es algo que está considerando, piense con firmeza si una clase podría no ser el mejor enfoque aquí, si es el suyo. tipo.

## Examples

### Cambio de valores en otra parte

```
public static void Main(string[] args)
{
```

```
    var studentList = new List<Student>();
    studentList.Add(new Student("Scott", "Nuke"));
    studentList.Add(new Student("Vincent", "King"));
    studentList.Add(new Student("Craig", "Bertt"));

    // make a separate list to print out later
    var printingList = studentList; // this is a new list object, but holding the same student
objects inside it

    // oops, we've noticed typos in the names, so we fix those
    studentList[0].LastName = "Duke";
    studentList[1].LastName = "Kong";
    studentList[2].LastName = "Brett";

    // okay, we now print the list
    PrintPrintingList(printingList);
}

private static void PrintPrintingList(List<Student> students)
{
    foreach (Student student in students)
    {
        Console.WriteLine(string.Format("{0} {1}", student.FirstName, student.LastName));
    }
}
```

Notará que aunque la lista de lista de impresión se realizó antes de las correcciones a los nombres de los estudiantes después de los errores tipográficos, el método PrintPrintingList aún imprime los nombres corregidos:

```
Scott Duke
Vincent Kong
Craig Brett
```

Esto se debe a que ambas listas contienen una lista de referencias a los mismos estudiantes. Por lo tanto, el cambio del objeto de estudiante subyacente se propaga a usos por cualquiera de las listas.

Así es como se vería la clase de los estudiantes.

```
public class Student
{
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public Student(string firstName, string lastName)
    {
        this.FirstName = firstName;
        this.LastName = lastName;
    }
}
```

## Pasando por referencia

Si desea que los tipos de valor frente a los tipos de referencia en el ejemplo de métodos

funcionen correctamente, use la palabra clave ref en su firma de método para el parámetro que desea pasar por referencia, así como cuando llame al método.

```
public static void Main(string[] args)
{
    ...
    DoubleNumber(ref number); // calling code
    Console.WriteLine(number); // outputs 8
    ...
}
```

```
public void DoubleNumber(ref int number)
{
    number += number;
}
```

Realizar estos cambios haría que el número se actualizara como se esperaba, lo que significa que la salida de la consola para el número sería 8.

## Pasando por referencia utilizando la palabra clave ref.

De la documentación :

> En C #, los argumentos se pueden pasar a los parámetros ya sea por valor o por referencia. Pasar por referencia permite que los miembros, métodos, propiedades, indizadores, operadores y constructores de funciones cambien el valor de los parámetros y hagan que ese cambio persista en el entorno de llamada. Para pasar un parámetro por referencia, use la palabra clave `ref` o `out` .

La diferencia entre `ref` y `out` es que `out` significa que el parámetro pasado debe asignarse antes de que la función termine. En contraste, los parámetros pasados con `ref` pueden cambiarse o dejarse sin cambios.

```
using System;

class Program
{
    static void Main(string[] args)
    {
        int a = 20;
        Console.WriteLine("Inside Main - Before Callee: a = {0}", a);
        Callee(a);
        Console.WriteLine("Inside Main - After Callee: a = {0}", a);

        Console.WriteLine("Inside Main - Before CalleeRef: a = {0}", a);
        CalleeRef(ref a);
        Console.WriteLine("Inside Main - After CalleeRef: a = {0}", a);

        Console.WriteLine("Inside Main - Before CalleeOut: a = {0}", a);
        CalleeOut(out a);
        Console.WriteLine("Inside Main - After CalleeOut: a = {0}", a);

        Console.ReadLine();
    }
```

```
    static void Callee(int a)
    {
        a = 5;
        Console.WriteLine("Inside Callee a : {0}", a);
    }

    static void CalleeRef(ref int a)
    {
        a = 6;
        Console.WriteLine("Inside CalleeRef a : {0}", a);
    }

    static void CalleeOut(out int a)
    {
        a = 7;
        Console.WriteLine("Inside CalleeOut a : {0}", a);
    }
}
```

**Salida** :

```
Inside Main - Before Callee: a = 20
Inside Callee a : 5
Inside Main - After Callee: a = 20
Inside Main - Before CalleeRef: a = 20
Inside CalleeRef a : 6
Inside Main - After CalleeRef: a = 6
Inside Main - Before CalleeOut: a = 6
Inside CalleeOut a : 7
Inside Main - After CalleeOut: a = 7
```

## Asignación

```
var a = new List<int>();
var b = a;
a.Add(5);
Console.WriteLine(a.Count); // prints 1
Console.WriteLine(b.Count); // prints 1 as well
```

La asignación a una variable de una `List<int>` no crea una copia de la `List<int>` . En su lugar, copia la referencia a la `List<int>` . Llamamos a los tipos que se comportan de esta manera los *tipos de referencia* .

## Diferencia con los parámetros del método ref y out.

Hay dos formas posibles de pasar un tipo de valor por referencia: `ref` y `out` . La diferencia es que al pasarlo con `ref` el valor debe inicializarse, pero no al pasarlo por `out` . Usando `out` asegura que la variable tiene un valor después de la llamada al método:

```
public void ByRef(ref int value)
{
    Console.WriteLine(nameof(ByRef) + value);
    value += 4;
```

```
    Console.WriteLine(nameof(ByRef) + value);
}

public void ByOut(out int value)
{
    value += 4 // CS0269: Use of unassigned out parameter `value'
    Console.WriteLine(nameof(ByOut) + value); // CS0269: Use of unassigned out parameter
`value'

    value = 4;
    Console.WriteLine(nameof(ByOut) + value);
}

public void TestOut()
{
    int outValue1;
    ByOut(out outValue1); // prints 4

    int outValue2 = 10;   // does not make any sense for out
    ByOut(out outValue2); // prints 4
}

public void TestRef()
{
    int refValue1;
    ByRef(ref refValue1); // S0165  Use of unassigned local variable 'refValue'

    int refValue2 = 0;
    ByRef(ref refValue2); // prints 0 and 4

    int refValue3 = 10;
    ByRef(ref refValue3); // prints 10 and 14
}
```

El problema es que mediante el uso `out` el parámetro `must` ser inicializado antes de abandonar el método, por lo tanto, el siguiente método es posible con `ref` pero no con `out` :

```
public void EmtyRef(bool condition, ref int value)
{
    if (condition)
    {
        value += 10;
    }
}

public void EmtyOut(bool condition, out int value)
{
    if (condition)
    {
        value = 10;
    }
} //CS0177: The out parameter 'value' must be assigned before control leaves the current
method
```

Esto se debe a que si la `condition` no se cumple, el `value` no se asigna.

## Parámetros ref vs out

### Código

```
class Program
{
    static void Main(string[] args)
    {
        int a = 20;
        Console.WriteLine("Inside Main - Before Callee: a = {0}", a);
        Callee(a);
        Console.WriteLine("Inside Main - After Callee: a = {0}", a);
        Console.WriteLine();

        Console.WriteLine("Inside Main - Before CalleeRef: a = {0}", a);
        CalleeRef(ref a);
        Console.WriteLine("Inside Main - After CalleeRef: a = {0}", a);
        Console.WriteLine();

        Console.WriteLine("Inside Main - Before CalleeOut: a = {0}", a);
        CalleeOut(out a);
        Console.WriteLine("Inside Main - After CalleeOut: a = {0}", a);
        Console.ReadLine();
    }

    static void Callee(int a)
    {
        a += 5;
        Console.WriteLine("Inside Callee a : {0}", a);
    }

    static void CalleeRef(ref int a)
    {
        a += 10;
        Console.WriteLine("Inside CalleeRef a : {0}", a);
    }

    static void CalleeOut(out int a)
    {
        // can't use a+=15 since for this method 'a' is not intialized only declared in the
method declaration
        a = 25; //has to be initialized
        Console.WriteLine("Inside CalleeOut a : {0}", a);
    }
}
```

### Salida

```
Inside Main - Before Callee: a = 20
Inside Callee a : 25
Inside Main - After Callee: a = 20

Inside Main - Before CalleeRef: a = 20
Inside CalleeRef a : 30
Inside Main - After CalleeRef: a = 30

Inside Main - Before CalleeOut: a = 30
Inside CalleeOut a : 25
Inside Main - After CalleeOut: a = 25
```

Lea Tipo de valor vs tipo de referencia en línea: https://riptutorial.com/es/csharp/topic/3014/tipo-

de-valor-vs-tipo-de-referencia

# Capítulo 153: Tipo dinámico

## Observaciones

La palabra clave `dynamic` declara una variable cuyo tipo no se conoce en el momento de la compilación. Una variable `dynamic` puede contener cualquier valor y el tipo de valor puede cambiar durante el tiempo de ejecución.

Como se señaló en el libro "Metaprogramación en .NET", C # no tiene un tipo de respaldo para la palabra clave `dynamic` :

> La funcionalidad habilitada por la palabra clave `dynamic` es un conjunto inteligente de acciones del compilador que emiten y usan objetos `CallSite` en el contenedor del sitio del ámbito de ejecución local. El compilador administra lo que los programadores perciben como referencias de objetos dinámicos a través de esas instancias de `CallSite` . Los parámetros, tipos de retorno, campos y propiedades que reciben un tratamiento dinámico en el momento de la compilación pueden marcarse con algunos metadatos para indicar que se generaron para uso dinámico, pero el tipo de datos subyacente para ellos siempre será `System.Object` .

## Examples

### Creando una variable dinámica

```
dynamic foo = 123;
Console.WriteLine(foo + 234);
// 357    Console.WriteLine(foo.ToUpper())
// RuntimeBinderException, since int doesn't have a ToUpper method

foo = "123";
Console.WriteLine(foo + 234);
// 123234
Console.WriteLine(foo.ToUpper()):
// NOW A STRING
```

### Dinámica de retorno

```
using System;

public static void Main()
{
    var value = GetValue();
    Console.WriteLine(value);
    // dynamics are useful!
}

private static dynamic GetValue()
{
    return "dynamics are useful!";
```

```
}
```

## Creando un objeto dinámico con propiedades.

```
using System;
using System.Dynamic;

dynamic info = new ExpandoObject();
info.Id = 123;
info.Another = 456;

Console.WriteLine(info.Another);
// 456

Console.WriteLine(info.DoesntExist);
// Throws RuntimeBinderException
```

## Manejo de tipos específicos desconocidos en tiempo de compilación

Los siguientes resultados equivalentes de salida:

```
class IfElseExample
{
    public string DebugToString(object a)
    {
        if (a is StringBuilder)
        {
            return DebugToStringInternal(a as StringBuilder);
        }
        else if (a is List<string>)
        {
            return DebugToStringInternal(a as List<string>);
        }
        else
        {
            return a.ToString();
        }
    }

    private string DebugToStringInternal(object a)
    {
        // Fall Back
        return a.ToString();
    }

    private string DebugToStringInternal(StringBuilder sb)
    {
        return $"StringBuilder - Capacity: {sb.Capacity}, MaxCapacity: {sb.MaxCapacity},
Value: {sb.ToString()}";
    }

    private string DebugToStringInternal(List<string> list)
    {
        return $"List<string> - Count: {list.Count}, Value: {Environment.NewLine + "\t" +
string.Join(Environment.NewLine + "\t", list.ToArray())}";
    }
}
```

```
class DynamicExample
{
    public string DebugToString(object a)
    {
        return DebugToStringInternal((dynamic)a);
    }

    private string DebugToStringInternal(object a)
    {
        // Fall Back
        return a.ToString();
    }

    private string DebugToStringInternal(StringBuilder sb)
    {
        return $"StringBuilder - Capacity: {sb.Capacity}, MaxCapacity: {sb.MaxCapacity},
Value: {sb.ToString()}";
    }

    private string DebugToStringInternal(List<string> list)
    {
        return $"List<string> - Count: {list.Count}, Value: {Environment.NewLine + "\t" +
string.Join(Environment.NewLine + "\t", list.ToArray())}";
    }
}
```

La ventaja de la dinámica, es que agregar un nuevo Tipo para manejar solo requiere agregar una sobrecarga de DebugToStringInternal del nuevo tipo. También elimina la necesidad de convertirlo manualmente al tipo también.

Lea Tipo dinámico en línea: https://riptutorial.com/es/csharp/topic/762/tipo-dinamico

# Capítulo 154: Tipos anónimos

## Examples

### Creando un tipo anónimo

Como los tipos anónimos no tienen nombre, las variables de esos tipos deben escribirse implícitamente ( `var` ).

```
var anon = new { Foo = 1, Bar = 2 };
// anon.Foo == 1
// anon.Bar == 2
```

Si no se especifican los nombres de los miembros, se establecen en el nombre de la propiedad / variable utilizada para inicializar el objeto.

```
int foo = 1;
int bar = 2;
var anon2 = new { foo, bar };
// anon2.foo == 1
// anon2.bar == 2
```

Tenga en cuenta que los nombres solo se pueden omitir cuando la expresión en la declaración de tipo anónimo es un simple acceso de propiedad; para llamadas a métodos o expresiones más complejas, se debe especificar un nombre de propiedad.

```
string foo = "some string";
var anon3 = new { foo.Length };
// anon3.Length == 11
var anon4 = new { foo.Length <= 10 ? "short string" : "long string" };
// compiler error - Invalid anonymous type member declarator.
var anon5 = new { Description = foo.Length <= 10 ? "short string" : "long string" };
// OK
```

### Anónimo vs dinámico

Los tipos anónimos permiten la creación de objetos sin tener que definir explícitamente sus tipos antes de tiempo, mientras se mantiene la comprobación de tipos estática.

```
var anon = new { Value = 1 };
Console.WriteLine(anon.Id); // compile time error
```

Por el contrario, la `dynamic` tiene una comprobación dinámica de tipos, optando por errores de tiempo de ejecución, en lugar de errores de compilación.

```
dynamic val = "foo";
Console.WriteLine(val.Id); // compiles, but throws runtime error
```

## Métodos genéricos con tipos anónimos.

Los métodos genéricos permiten el uso de tipos anónimos a través de la inferencia de tipos.

```
void Log<T>(T obj) {
    // ...
}
Log(new { Value = 10 });
```

Esto significa que las expresiones LINQ se pueden usar con tipos anónimos:

```
var products = new[] {
    new { Amount = 10, Id = 0 },
    new { Amount = 20, Id = 1 },
    new { Amount = 15, Id = 2 }
};
var idsByAmount = products.OrderBy(x => x.Amount).Select(x => x.Id);
// idsByAmount: 0, 2, 1
```

## Creando tipos genéricos con tipos anónimos

El uso de constructores genéricos requeriría el nombre de los tipos anónimos, lo que no es posible. Alternativamente, se pueden usar métodos genéricos para permitir que ocurra la inferencia de tipos.

```
var anon = new { Foo = 1, Bar = 2 };
var anon2 = new { Foo = 5, Bar = 10 };
List<T> CreateList<T>(params T[] items) {
    return new List<T>(items);
}

var list1 = CreateList(anon, anon2);
```

En el caso de la `List<T>` , las matrices tipificadas implícitamente se pueden convertir en una `List<T>` través del método `ToList` LINQ:

```
var list2 = new[] {anon, anon2}.ToList();
```

## Igualdad de tipo anónimo

La igualdad de tipo anónima viene dada por el método de instancia `Equals` . Dos objetos son iguales si tienen el mismo tipo y valores iguales (a través de `a.Prop.Equals(b.Prop)` ) para cada propiedad.

```
var anon = new { Foo = 1, Bar = 2 };
var anon2 = new { Foo = 1, Bar = 2 };
var anon3 = new { Foo = 5, Bar = 10 };
var anon3 = new { Foo = 5, Bar = 10 };
var anon4 = new { Bar = 2, Foo = 1 };
// anon.Equals(anon2) == true
// anon.Equals(anon3) == false
```

```
// anon.Equals(anon4) == false (anon and anon4 have different types, see below)
```

Dos tipos anónimos se consideran iguales solo si sus propiedades tienen el mismo nombre y tipo y aparecen en el mismo orden.

```
var anon = new { Foo = 1, Bar = 2 };
var anon2 = new { Foo = 7, Bar = 1 };
var anon3 = new { Bar = 1, Foo = 3 };
var anon4 = new { Fa = 1, Bar = 2 };
// anon and anon2 have the same type
// anon and anon3 have diferent types (Bar and Foo appear in different orders)
// anon and anon4 have different types (property names are different)
```

## Arrays implícitamente escritos

Las matrices de tipos anónimos se pueden crear con escritura implícita.

```
var arr = new[] {
    new { Id = 0 },
    new { Id = 1 }
};
```

Lea Tipos anónimos en línea: https://riptutorial.com/es/csharp/topic/765/tipos-anonimos

# Capítulo 155: Tipos anulables

## Sintaxis

- `Nullable<int> i = 10;`
- ¿En t? j = 11;
- ¿En t? k = nulo;
- ¿Fecha y hora? DateOfBirth = DateTime.Now;
- ¿decimal? Cantidad = 1.0m;
- bool IsAvailable = true;
- ¿carbonizarse? Letra = 'a';
- (tipo)? nombre de la variable

## Observaciones

Los tipos anulables pueden representar todos los valores de un tipo subyacente, así como `null` .

La sintaxis `T?` es taquigrafía para `Nullable<T>`

Los valores anulables son objetos `System.ValueType` realidad, por lo que se pueden encuadrar y desempaquetar. Además, el valor `null` de un objeto anulable no es lo mismo que el valor `null` de un objeto de referencia, es solo un indicador.

Cuando se utiliza un cuadro de objetos anulables, el valor nulo se convierte en referencia `null` y el valor no nulo se convierte en un tipo subyacente no anulable.

```
DateTime? dt = null;
var o = (object)dt;
var result = (o == null); // is true

DateTime? dt = new DateTime(2015, 12, 11);
var o = (object)dt;
var dt2 = (DateTime)dt; // correct cause o contains DateTime value
```

La segunda regla conduce al código correcto, pero paradójico:

```
DateTime? dt = new DateTime(2015, 12, 11);
var o = (object)dt;
var type = o.GetType(); // is DateTime, not Nullable<DateTime>
```

En forma corta:

```
DateTime? dt = new DateTime(2015, 12, 11);
var type = dt.GetType(); // is DateTime, not Nullable<DateTime>
```

## Examples

## Inicializando un nullable

Para valores `null` :

```
Nullable<int> i = null;
```

O:

```
int? i = null;
```

O:

```
var i = (int?)null;
```

Para valores no nulos:

```
Nullable<int> i = 0;
```

O:

```
int? i = 0;
```

## Compruebe si un Nullable tiene un valor

```
int? i = null;

if (i != null)
{
    Console.WriteLine("i is not null");
}
else
{
    Console.WriteLine("i is null");
}
```

Que es lo mismo que:

```
if (i.HasValue)
{
    Console.WriteLine("i is not null");
}
else
{
    Console.WriteLine("i is null");
}
```

## Obtener el valor de un tipo anulable

Dado lo siguiente nullable `int`

```
int? i = 10;
```

En caso de que se requiera un valor predeterminado, puede asignar uno usando el operador de unión nula , el método `GetValueOrDefault` o verificar si int `HasValue` antes de la asignación.

```
int j = i ?? 0;
int j = i.GetValueOrDefault(0);
int j = i.HasValue ? i.Value : 0;
```

El siguiente uso es siempre *inseguro* . Si `i` es nulo en el tiempo de ejecución, se lanzará una `System.InvalidOperationException` . En el momento del diseño, si no se establece un valor, obtendrá un error `Use of unassigned local variable 'i'` .

```
int j = i.Value;
```

## Obtener un valor predeterminado de un nullable

El método `.GetValueOrDefault()` devuelve un valor incluso si la propiedad `.HasValue` es falsa (a diferencia de la propiedad Value, que lanza una excepción).

```
class Program
{
    static void Main()
    {
        int? nullableExample = null;
        int result = nullableExample.GetValueOrDefault();
        Console.WriteLine(result); // will output the default value for int - 0
        int secondResult = nullableExample.GetValueOrDefault(1);
        Console.WriteLine(secondResult) // will output our specified default - 1
        int thirdResult = nullableExample ?? 1;
        Console.WriteLine(secondResult) // same as the GetValueOrDefault but a bit shorter
    }
}
```

---

Salida:

```
0
1
```

## Compruebe si un parámetro de tipo genérico es un tipo anulable

```
public bool IsTypeNullable<T>()
{
    return Nullable.GetUnderlyingType( typeof(T) )!=null;
}
```

## El valor predeterminado de los tipos anulables es nulo

```
public class NullableTypesExample
```

```
{
    static int? _testValue;

    public static void Main()
    {
        if(_testValue == null)
            Console.WriteLine("null");
        else
            Console.WriteLine(_testValue.ToString());
    }
}
```

Salida:

nulo

## Uso efectivo de Nullable subyacente argumento

Cualquier tipo anulable es un tipo **genérico** . Y cualquier tipo anulable es un tipo de **valor** .

Hay algunos trucos que permiten **utilizar efectivamente** el resultado del método Nullable.GetUnderlyingType al crear código relacionado con propósitos de reflexión / generación de código:

```
public static class TypesHelper {
    public static bool IsNullable(this Type type) {
        Type underlyingType;
        return IsNullable(type, out underlyingType);
    }
    public static bool IsNullable(this Type type, out Type underlyingType) {
        underlyingType = Nullable.GetUnderlyingType(type);
        return underlyingType != null;
    }
    public static Type GetNullable(Type type) {
        Type underlyingType;
        return IsNullable(type, out underlyingType) ? type : NullableTypesCache.Get(type);
    }
    public static bool IsExactOrNullable(this Type type, Func<Type, bool> predicate) {
        Type underlyingType;
        if(IsNullable(type, out underlyingType))
            return IsExactOrNullable(underlyingType, predicate);
        return predicate(type);
    }
    public static bool IsExactOrNullable<T>(this Type type)
        where T : struct {
        return IsExactOrNullable(type, t => Equals(t, typeof(T)));
    }
}
```

El uso:

```
Type type = typeof(int).GetNullable();
Console.WriteLine(type.ToString());

if(type.IsNullable())
    Console.WriteLine("Type is nullable.");
```

```
Type underlyingType;
if(type.IsNullable(out underlyingType))
    Console.WriteLine("The underlying type is " + underlyingType.Name + ".");
if(type.IsExactOrNullable<int>())
    Console.WriteLine("Type is either exact or nullable Int32.");
if(!type.IsExactOrNullable(t => t.IsEnum))
    Console.WriteLine("Type is neither exact nor nullable enum.");
```

Salida:

```
System.Nullable`1[System.Int32]
Type is nullable.
The underlying type is Int32.
Type is either exact or nullable Int32.
Type is neither exact nor nullable enum.
```

PD. El `NullableTypesCache` se define de la siguiente manera:

```
static class NullableTypesCache {
    readonly static ConcurrentDictionary<Type, Type> cache = new ConcurrentDictionary<Type,
Type>();
    static NullableTypesCache() {
        cache.TryAdd(typeof(byte), typeof(Nullable<byte>));
        cache.TryAdd(typeof(short), typeof(Nullable<short>));
        cache.TryAdd(typeof(int), typeof(Nullable<int>));
        cache.TryAdd(typeof(long), typeof(Nullable<long>));
        cache.TryAdd(typeof(float), typeof(Nullable<float>));
        cache.TryAdd(typeof(double), typeof(Nullable<double>));
        cache.TryAdd(typeof(decimal), typeof(Nullable<decimal>));
        cache.TryAdd(typeof(sbyte), typeof(Nullable<sbyte>));
        cache.TryAdd(typeof(ushort), typeof(Nullable<ushort>));
        cache.TryAdd(typeof(uint), typeof(Nullable<uint>));
        cache.TryAdd(typeof(ulong), typeof(Nullable<ulong>));
        //...
    }
    readonly static Type NullableBase = typeof(Nullable<>);
    internal static Type Get(Type type) {
        // Try to avoid the expensive MakeGenericType method call
        return cache.GetOrAdd(type, t => NullableBase.MakeGenericType(t));
    }
}
```

Lea Tipos anulables en línea: https://riptutorial.com/es/csharp/topic/1240/tipos-anulables

# Capítulo 156: Tipos incorporados

## Examples

### Tipo de referencia inmutable - cadena

```
// assign string from a string literal
string s = "hello";

// assign string from an array of characters
char[] chars = new char[] { 'h', 'e', 'l', 'l', 'o' };
string s = new string(chars, 0, chars.Length);

// assign string from a char pointer, derived from a string
string s;
unsafe
{
    fixed (char* charPointer = "hello")
    {
        s = new string(charPointer);
    }
}
```

### Tipo de valor - char

```
// single character s
char c = 's';

// character s: casted from integer value
char c = (char)115;

// unicode character: single character s
char c = '\u0073';

// unicode character: smiley face
char c = '\u263a';
```

### Tipo de valor - short, int, long (enteros con signo de 16 bits, 32 bits, 64 bits)

```
// assigning a signed short to its minimum value
short s = -32768;

// assigning a signed short to its maximum value
short s = 32767;

// assigning a signed int to its minimum value
int i = -2147483648;

// assigning a signed int to its maximum value
int i = 2147483647;

// assigning a signed long to its minimum value (note the long postfix)
long l = -9223372036854775808L;
```

```
// assigning a signed long to its maximum value (note the long postfix)
long l = 9223372036854775807L;
```

También es posible hacer que estos tipos sean anulables, lo que significa que además de los valores habituales, también se puede asignar un valor nulo. Si una variable de un tipo anulable no se inicializa, será nula en lugar de 0. Los tipos anulables se marcan agregando un signo de interrogación (?) Después del tipo.

```
int a; //This is now 0.
int? b; //This is now null.
```

## Tipo de valor - ushort, uint, ulong (enteros sin signo de 16 bits, 32 bits, 64 bits)

```
// assigning an unsigned short to its minimum value
ushort s = 0;

// assigning an unsigned short to its maximum value
ushort s = 65535;

// assigning an unsigned int to its minimum value
uint i = 0;

// assigning an unsigned int to its maximum value
uint i = 4294967295;

// assigning an unsigned long to its minimum value (note the unsigned long postfix)
ulong l = 0UL;

// assigning an unsigned long to its maximum value (note the unsigned long postfix)
ulong l = 18446744073709551615UL;
```

También es posible hacer que estos tipos sean anulables, lo que significa que además de los valores habituales, también se puede asignar un valor nulo. Si una variable de un tipo anulable no se inicializa, será nula en lugar de 0. Los tipos anulables se marcan agregando un signo de interrogación (?) Después del tipo.

```
uint a; //This is now 0.
uint? b; //This is now null.
```

## Tipo de valor - bool

```
// default value of boolean is false
bool b;
//default value of nullable boolean is null
bool? z;
b = true;
if(b) {
    Console.WriteLine("Boolean has true value");
}
```

La palabra clave bool es un alias de System.Boolean. Se utiliza para declarar variables para almacenar los valores booleanos, `true` y `false` .

## Comparaciones con tipos de valor en caja

Si los tipos de valor se asignan a las variables del tipo de `object` , están *encuadrados* : el valor se almacena en una instancia de un objeto `System.Object` . Esto puede llevar a consecuencias no intencionadas al comparar valores con `==` , por ejemplo:

```
object left = (int)1;  // int in an object box
object right = (int)1; // int in an object box

var comparison1 = left == right;      // false
```

Esto se puede evitar utilizando el método de sobrecarga de `Equals` , que dará el resultado esperado.

```
var comparison2 = left.Equals(right); // true
```

Alternativamente, se podría hacer lo mismo desempaquetando las variables `left` y `right` para que los valores `int` sean comparados:

```
var comparison3 = (int)left == (int)right; // true
```

## Conversión de tipos de valor en caja

Los tipos de valor en caja solo se pueden desempaquetar en su `Type` original, incluso si una conversión de los dos `Type` s es válida, por ejemplo:

```
object boxedInt = (int)1; // int boxed in an object

long unboxedInt1 = (long)boxedInt; // invalid cast
```

Esto se puede evitar si primero se desempaqueta en el `Type` original, por ejemplo:

```
long unboxedInt2 = (long)(int)boxedInt; // valid
```

Lea Tipos incorporados en línea: https://riptutorial.com/es/csharp/topic/42/tipos-incorporados

# Capítulo 157: Trabajador de fondo

## Sintaxis

- `bgWorker.CancellationPending //returns whether the bgWorker was cancelled during its operation`
- `bgWorker.IsBusy //returns true if the bgWorker is in the middle of an operation`
- `bgWorker.ReportProgress(int x) //Reports a change in progress. Raises the "ProgressChanged" event`
- `bgWorker.RunWorkerAsync() //Starts the BackgroundWorker by raising the "DoWork" event`
- `bgWorker.CancelAsync() //instructs the BackgroundWorker to stop after the completion of a task.`

## Observaciones

La realización de operaciones de larga ejecución dentro del subproceso de la interfaz de usuario puede hacer que su aplicación deje de responder, y le parece al usuario que ha dejado de funcionar. Es preferible que estas tareas se ejecuten en un subproceso en segundo plano. Una vez completado, la interfaz de usuario se puede actualizar.

Realizar cambios en la interfaz de usuario durante la operación de BackgroundWorker requiere invocar los cambios en el subproceso de la interfaz de usuario, generalmente mediante el método Control.Invoke en el control que está actualizando. Si no lo hace, su programa generará una excepción.

El BackgroundWorker normalmente solo se usa en aplicaciones de Windows Forms. En las aplicaciones WPF, las tareas se utilizan para descargar trabajos en subprocesos en segundo plano (posiblemente en combinación con async / await ). La actualización de las actualizaciones en el subproceso de la interfaz de usuario se realiza de forma automática, cuando la propiedad que se está actualizando implementa INotifyPropertyChanged , o manualmente mediante el Dispatcher del subproceso de la interfaz de usuario.

## Examples

### Asignar controladores de eventos a un BackgroundWorker

Una vez que se ha declarado la instancia de BackgroundWorker, se le deben otorgar propiedades y controladores de eventos para las tareas que realiza.

```
    /* This is the backgroundworker's "DoWork" event handler. This
       method is what will contain all the work you
       wish to have your program perform without blocking the UI. */

    bgWorker.DoWork += bgWorker_DoWork;
```

```
    /*This is how the DoWork event method signature looks like:*/
    private void bgWorker_DoWork(object sender, DoWorkEventArgs e)
    {
        // Work to be done here
        // ...
        // To get a reference to the current Backgroundworker:
        BackgroundWorker worker = sender as BackgroundWorker;
        // The reference to the BackgroundWorker is often used to report progress
        worker.ReportProgress(...);
    }

    /*This is the method that will be run once the BackgroundWorker has completed its tasks */

    bgWorker.RunWorkerCompleted += bgWorker_CompletedWork;

    /*This is how the RunWorkerCompletedEvent event method signature looks like:*/
    private void bgWorker_CompletedWork(object sender, RunWorkerCompletedEventArgs e)
    {
        // Things to be done after the backgroundworker has finished
    }

   /* When you wish to have something occur when a change in progress
     occurs, (like the completion of a specific task) the "ProgressChanged"
     event handler is used. Note that ProgressChanged events may be invoked
     by calls to bgWorker.ReportProgress(...) only if bgWorker.WorkerReportsProgress
     is set to true.  */

     bgWorker.ProgressChanged += bgWorker_ProgressChanged;

    /*This is how the ProgressChanged event method signature looks like:*/
    private void bgWorker_ProgressChanged(object sender, ProgressChangedEventArgs e)
    {
        // Things to be done when a progress change has been reported

        /* The ProgressChangedEventArgs gives access to a percentage,
         allowing for easy reporting of how far along a process is*/
        int progress = e.ProgressPercentage;
    }
```

## Asignación de propiedades a un BackgroundWorker

Esto permite que BackgroundWorker se cancele entre tareas

```
bgWorker.WorkerSupportsCancellation = true;
```

Esto permite al trabajador informar el progreso entre la finalización de las tareas ...

```
bgWorker.WorkerReportsProgress = true;

//this must also be used in conjunction with the ProgressChanged event
```

## Creando una nueva instancia de BackgroundWorker

Un BackgroundWorker se usa comúnmente para realizar tareas, a veces mucho tiempo, sin bloquear el subproceso de la interfaz de usuario.

```
// BackgroundWorker is part of the ComponentModel namespace.
using System.ComponentModel;

namespace BGWorkerExample
{
     public partial class ExampleForm : Form
     {

      // the following creates an instance of the BackgroundWorker named "bgWorker"
      BackgroundWorker bgWorker = new BackgroundWorker();

      public ExampleForm() { ...
```

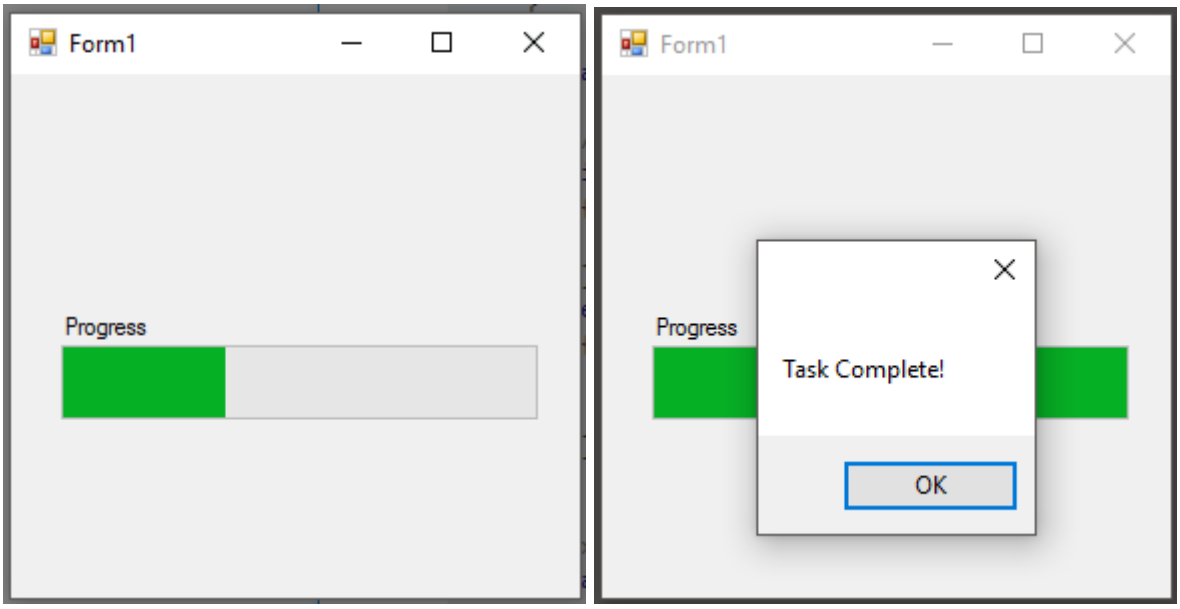
## Usando un BackgroundWorker para completar una tarea.

El siguiente ejemplo demuestra el uso de un BackgroundWorker para actualizar un WinForms ProgressBar. El backgroundWorker actualizará el valor de la barra de progreso sin bloquear el subproceso de la interfaz de usuario, mostrando así una interfaz de usuario reactiva mientras el trabajo se realiza en segundo plano.

```
namespace BgWorkerExample
{
    public partial class Form1 : Form
{

    //a new instance of a backgroundWorker is created.
    BackgroundWorker bgWorker = new BackgroundWorker();

    public Form1()
    {
        InitializeComponent();

        prgProgressBar.Step = 1;

        //this assigns event handlers for the backgroundWorker
        bgWorker.DoWork += bgWorker_DoWork;
        bgWorker.RunWorkerCompleted += bgWorker_WorkComplete;

        //tell the backgroundWorker to raise the "DoWork" event, thus starting it.
        //Check to make sure the background worker is not already running.
        if(!bgWorker.IsBusy)
            bgWorker.RunWorkerAsync();

    }

    private void bgWorker_DoWork(object sender, DoWorkEventArgs e)
    {
        //this is the method that the backgroundworker will perform on in the background
thread.
        /* One thing to note! A try catch is not necessary as any exceptions will terminate
the backgroundWorker and report
          the error to the "RunWorkerCompleted" event */
        CountToY();
    }

    private void bgWorker_WorkComplete(object sender, RunWorkerCompletedEventArgs e)
    {
        //e.Error will contain any exceptions caught by the backgroundWorker
```

```
        if (e.Error != null)
        {
            MessageBox.Show(e.Error.Message);
        }
        else
        {
            MessageBox.Show("Task Complete!");
            prgProgressBar.Value = 0;
        }
    }

    // example method to perform a "long" running task.
    private void CountToY()
    {
        int x = 0;

        int maxProgress = 100;
        prgProgressBar.Maximum = maxProgress;


        while (x < maxProgress)
        {
            System.Threading.Thread.Sleep(50);
            Invoke(new Action(() => { prgProgressBar.PerformStep(); }));
            x += 1;
        }
    }


}
```

## El resultado es el siguiente...




Lea Trabajador de fondo en línea: https://riptutorial.com/es/csharp/topic/1588/trabajador-de-fondo

# Capítulo 158: Tuplas

## Examples

### Creando tuplas

Las tuplas se crean utilizando los tipos genéricos `Tuple<T1>` - `Tuple<T1,T2,T3,T4,T5,T6,T7,T8>` . Cada uno de los tipos representa una tupla que contiene de 1 a 8 elementos. Los elementos pueden ser de diferentes tipos.

```
// tuple with 4 elements
var tuple = new Tuple<string, int, bool, MyClass>("foo", 123, true, new MyClass());
```

Las tuplas también se pueden crear utilizando métodos estáticos de `Tuple.Create` . En este caso, los tipos de los elementos son inferidos por el compilador de C #.

```
// tuple with 4 elements
var tuple = Tuple.Create("foo", 123, true, new MyClass());
```

7.0

Desde C # 7.0, las tuplas se pueden crear fácilmente usando ValueTuple .

```
var tuple = ("foo", 123, true, new MyClass());
```

Los elementos pueden ser nombrados para una descomposición más fácil.

```
(int number, bool flag, MyClass instance) tuple = (123, true, new MyClass());
```

### Accediendo a los elementos de la tupla.

Para acceder a los elementos de tupla utilizan `Item1` - `Item8` propiedades. Solo las propiedades con número de índice menor o igual al tamaño de tupla estarán disponibles (es decir, no se puede acceder a la propiedad `Item3` en `Tuple<T1,T2>` ).

```
var tuple = new Tuple<string, int, bool, MyClass>("foo", 123, true, new MyClass());
var item1 = tuple.Item1; // "foo"
var item2 = tuple.Item2; // 123
var item3 = tuple.Item3; // true
var item4 = tuple.Item4; // new My Class()
```

### Comparando y clasificando tuplas

Las tuplas se pueden comparar en función de sus elementos.

Como ejemplo, un enumerable cuyos elementos son de tipo `Tuple` puede ordenarse en función de

los operadores de comparación definidos en un elemento específico:

```
List<Tuple<int, string>> list = new List<Tuple<int, string>>();
list.Add(new Tuple<int, string>(2, "foo"));
list.Add(new Tuple<int, string>(1, "bar"));
list.Add(new Tuple<int, string>(3, "qux"));

list.Sort((a, b) => a.Item2.CompareTo(b.Item2)); //sort based on the string element

foreach (var element in list) {
    Console.WriteLine(element);
}

// Output:
// (1, bar)
// (2, foo)
// (3, qux)
```

O para revertir el uso de ordenación:

```
list.Sort((a, b) => b.Item2.CompareTo(a.Item2));
```

## Devuelve múltiples valores de un método

Las tuplas se pueden usar para devolver múltiples valores de un método sin usar parámetros. En el siguiente ejemplo, `AddMultiply` se usa para devolver dos valores (suma, producto).

```
void Write()
{
    var result = AddMultiply(25, 28);
    Console.WriteLine(result.Item1);
    Console.WriteLine(result.Item2);
}

Tuple<int, int> AddMultiply(int a, int b)
{
    return new Tuple<int, int>(a + b, a * b);
}
```

Salida:

53
700

Ahora C # 7.0 ofrece una forma alternativa de devolver múltiples valores desde métodos que utilizan tuplas de valor. Más información sobre la estructura `ValueTuple` .

Lea Tuplas en línea: https://riptutorial.com/es/csharp/topic/838/tuplas

# Capítulo 159: Una visión general de c # colecciones

## Examples

### HashSet

Esta es una colección de artículos únicos, con búsqueda O (1).

```
HashSet<int> validStoryPointValues = new HashSet<int>() { 1, 2, 3, 5, 8, 13, 21 };
bool containsEight = validStoryPointValues.Contains(8); // O(1)
```

A modo de comparación, hacer un `Contains` en una lista produce un rendimiento más bajo:

```
List<int> validStoryPointValues = new List<int>() { 1, 2, 3, 5, 8, 13, 21 };
bool containsEight = validStoryPointValues.Contains(8); // O(n)
```

`HashSet.Contains` utiliza una tabla hash, por lo que las búsquedas son extremadamente rápidas, independientemente del número de elementos en la colección.

### SortedSet

```
// create an empty set
var mySet = new SortedSet<int>();

// add something
// note that we add 2 before we add 1
mySet.Add(2);
mySet.Add(1);

// enumerate through the set
foreach(var item in mySet)
{
    Console.WriteLine(item);
}

// output:
// 1
// 2
```

### T [] (Array de T)

```
// create an array with 2 elements
var myArray = new [] { "one", "two" };

// enumerate through the array
foreach(var item in myArray)
{
    Console.WriteLine(item);
```

```
}

// output:
// one
// two

// exchange the element on the first position
// note that all collections start with the index 0
myArray[0] = "something else";


// enumerate through the array again
foreach(var item in myArray)
{
    Console.WriteLine(item);
}

// output:
// something else
// two
```

## Lista

`List<T>` es una lista de un tipo dado. Los elementos se pueden agregar, insertar, eliminar y direccionar por índice.

```
using System.Collections.Generic;

var list = new List<int>() { 1, 2, 3, 4, 5 };
list.Add(6);
Console.WriteLine(list.Count); // 6
list.RemoveAt(3);
Console.WriteLine(list.Count); // 5
Console.WriteLine(list[3]);    // 5
```

`List<T>` se puede considerar como una matriz que se puede cambiar de tamaño. La enumeración de la colección en orden es rápida, al igual que el acceso a elementos individuales a través de su índice. Para acceder a elementos basados en algún aspecto de su valor, o alguna otra clave, un `Dictionary<T>` proporcionará una búsqueda más rápida.

## Diccionario

Diccionario <TKey, TValue> es un mapa. Para una clave dada puede haber un valor en el diccionario.

```
using System.Collections.Generic;

var people = new Dictionary<string, int>
{
    { "John", 30 }, {"Mary", 35}, {"Jack", 40}
};

// Reading data
Console.WriteLine(people["John"]); // 30
Console.WriteLine(people["George"]); // throws KeyNotFoundException
```

```
int age;
if (people.TryGetValue("Mary", out age))
{
    Console.WriteLine(age); // 35
}

// Adding and changing data
people["John"] = 40;    // Overwriting values this way is ok
people.Add("John", 40); // Throws ArgumentException since "John" already exists

// Iterating through contents
foreach(KeyValuePair<string, int> person in people)
{
    Console.WriteLine("Name={0}, Age={1}", person.Key, person.Value);
}

foreach(string name in people.Keys)
{
    Console.WriteLine("Name={0}", name);
}

foreach(int age in people.Values)
{
    Console.WriteLine("Age={0}", age);
}
```

# Clave duplicada al usar la inicialización de la colección

```
var people = new Dictionary<string, int>
{
    { "John", 30 }, {"Mary", 35}, {"Jack", 40}, {"Jack", 40}
}; // throws ArgumentException since "Jack" already exists
```

## Apilar

```
// Initialize a stack object of integers
var stack = new Stack<int>();

// add some data
stack.Push(3);
stack.Push(5);
stack.Push(8);

// elements are stored with "first in, last out" order.
// stack from top to bottom is: 8, 5, 3

// We can use peek to see the top element of the stack.
Console.WriteLine(stack.Peek()); // prints 8

// Pop removes the top element of the stack and returns it.
Console.WriteLine(stack.Pop()); // prints 8
Console.WriteLine(stack.Pop()); // prints 5
```

```
Console.WriteLine(stack.Pop()); // prints 3
```

## Lista enlazada

```
// initialize a LinkedList of integers
LinkedList list = new LinkedList<int>();

// add some numbers to our list.
list.AddLast(3);
list.AddLast(5);
list.AddLast(8);

// the list currently is 3, 5, 8

list.AddFirst(2);
// the list now is 2, 3, 5, 8

list.RemoveFirst();
// the list is now 3, 5, 8

list.RemoveLast();
// the list is now 3, 5
```

Tenga en cuenta que `LinkedList<T>` representa la lista *doblemente* enlazada. Por lo tanto, es simplemente una colección de nodos y cada nodo contiene un elemento de tipo `T` Cada nodo está vinculado al nodo anterior y al siguiente nodo.

## Cola

```
// Initalize a new queue of integers
var queue = new Queue<int>();

// Add some data
queue.Enqueue(6);
queue.Enqueue(4);
queue.Enqueue(9);

// Elements in a queue are stored in "first in, first out" order.
// The queue from first to last is: 6, 4, 9

// View the next element in the queue, without removing it.
Console.WriteLine(queue.Peek()); // prints 6

// Removes the first element in the queue, and returns it.
Console.WriteLine(queue.Dequeue()); // prints 6
Console.WriteLine(queue.Dequeue()); // prints 4
Console.WriteLine(queue.Dequeue()); // prints 9
```

> Hilo seguro cabezas arriba! Use ConcurrentQueue en entornos multi-hilo.

Lea Una visión general de c # colecciones en línea:
https://riptutorial.com/es/csharp/topic/2344/una-vision-general-de-c-sharp-colecciones

# Capítulo 160: Usando json.net

## Introducción

Usando JSON.net JsonConverter clase.

## Examples

### Usando JsonConverter en valores simples

Ejemplo de uso de JsonCoverter para deserializar la propiedad de tiempo de ejecución de la respuesta de la API en un objeto de intervalo de tiempo en el modelo de películas

# JSON ( http://www.omdbapi.com/?i=tt1663662)

```
{
    Title: "Pacific Rim",
    Year: "2013",
    Rated: "PG-13",
    Released: "12 Jul 2013",
    Runtime: "131 min",
    Genre: "Action, Adventure, Sci-Fi",
    Director: "Guillermo del Toro",
    Writer: "Travis Beacham (screenplay), Guillermo del Toro (screenplay), Travis Beacham
(story)",
    Actors: "Charlie Hunnam, Diego Klattenhoff, Idris Elba, Rinko Kikuchi",
    Plot: "As a war between humankind and monstrous sea creatures wages on, a former pilot and
a trainee are paired up to drive a seemingly obsolete special weapon in a desperate effort to
save the world from the apocalypse.",
    Language: "English, Japanese, Cantonese, Mandarin",
    Country: "USA",
    Awards: "Nominated for 1 BAFTA Film Award. Another 6 wins & 46 nominations.",
    Poster: "https://images-na.ssl-images-
amazon.com/images/M/MV5BMTY3MTI5NjQ4Nl5BMl5BanBnXkFtZTcwOTU1OTU0OQ@@._V1_SX300.jpg",
    Ratings: [{
            Source: "Internet Movie Database",
            Value: "7.0/10"
        },
        {
            Source: "Rotten Tomatoes",
            Value: "71%"
        },
        {
            Source: "Metacritic",
            Value: "64/100"
        }
    ],
    Metascore: "64",
    imdbRating: "7.0",
```

```
    imdbVotes: "398,198",
    imdbID: "tt1663662",
    Type: "movie",
    DVD: "15 Oct 2013",
    BoxOffice: "$101,785,482.00",
    Production: "Warner Bros. Pictures",
    Website: "http://pacificrimmovie.com",
    Response: "True"
}
```

# Modelo de película

```
using Project.Serializers;
using Newtonsoft.Json;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Runtime.Serialization;
using System.Threading.Tasks;

namespace Project.Models
{
    [DataContract]
    public class Movie
    {
        public Movie() { }

        [DataMember]
        public int Id { get; set; }

        [DataMember]
        public string ImdbId { get; set; }

        [DataMember]
        public string Title { get; set; }

        [DataMember]
        public DateTime Released { get; set; }

        [DataMember]
        [JsonConverter(typeof(RuntimeSerializer))]
        public TimeSpan Runtime { get; set; }

    }
}
```

# RuntimeSerializer

```
using Newtonsoft.Json;
using Newtonsoft.Json.Linq;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text.RegularExpressions;
```

```
using System.Threading.Tasks;

namespace Project.Serializers
{
    public class RuntimeSerializer : JsonConverter
    {
        public override bool CanConvert(Type objectType)
        {
            return objectType == typeof(TimeSpan);
        }

        public override object ReadJson(JsonReader reader, Type objectType, object
existingValue, JsonSerializer serializer)
        {
            if (reader.TokenType == JsonToken.Null)
                return null;

            JToken jt = JToken.Load(reader);
            String value = jt.Value<String>();

            Regex rx = new Regex("(\\s*)min$");
            value = rx.Replace(value, (m) => "");

            int timespanMin;
            if(!Int32.TryParse(value, out timespanMin))
            {
                throw new NotSupportedException();
            }

            return new TimeSpan(0, timespanMin, 0);
        }

        public override void WriteJson(JsonWriter writer, object value, JsonSerializer
serializer)
        {
            serializer.Serialize(writer, value);
        }
    }
}
```

# Llamándolo

```
Movie m = JsonConvert.DeserializeObject<Movie>(apiResponse));
```

## Recoge todos los campos del objeto JSON

```
using Newtonsoft.Json.Linq;
using System.Collections.Generic;

public class JsonFieldsCollector
{
    private readonly Dictionary<string, JValue> fields;

    public JsonFieldsCollector(JToken token)
    {
        fields = new Dictionary<string, JValue>();
```

```
        CollectFields(token);
    }

    private void CollectFields(JToken jToken)
    {
        switch (jToken.Type)
        {
            case JTokenType.Object:
                foreach (var child in jToken.Children<JProperty>())
                    CollectFields(child);
                break;
            case JTokenType.Array:
                foreach (var child in jToken.Children())
                    CollectFields(child);
                break;
            case JTokenType.Property:
                CollectFields(((JProperty) jToken).Value);
                break;
            default:
                fields.Add(jToken.Path, (JValue)jToken);
                break;
        }
    }

    public IEnumerable<KeyValuePair<string, JValue>> GetAllFields() => fields;
}
```

**Uso:**

```
var json = JToken.Parse(/* JSON string */);
var fieldsCollector = new JsonFieldsCollector(json);
var fields = fieldsCollector.GetAllFields();

foreach (var field in fields)
    Console.WriteLine($"{field.Key}: '{field.Value}'");
```

**Manifestación**
Para este objeto JSON

```
{
  "User": "John",
  "Workdays": {
    "Monday": true,
    "Tuesday": true,
    "Friday": false
  },
  "Age": 42
}
```

La salida esperada será:

```
User: 'John'
Workdays.Monday: 'True'
Workdays.Tuesday: 'True'
Workdays.Friday: 'False'
Age: '42'
```

Lea Usando json.net en línea: https://riptutorial.com/es/csharp/topic/9879/usando-json-net

# Capítulo 161: Usando SQLite en C #

## Examples

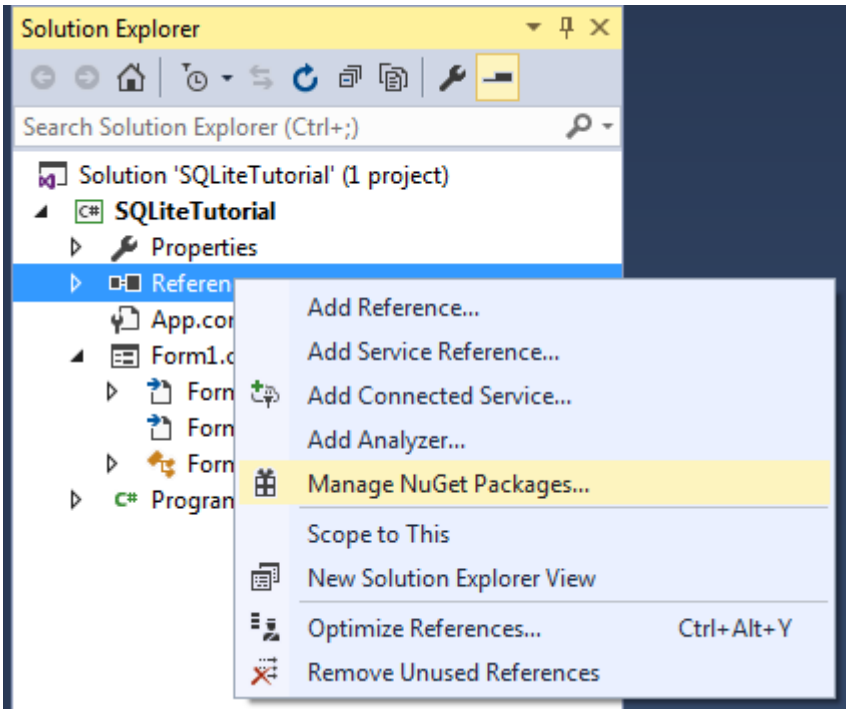
### Creando CRUD simple usando SQLite en C #

En primer lugar, necesitamos agregar soporte SQLite a nuestra aplicación. Hay dos maneras de hacer eso

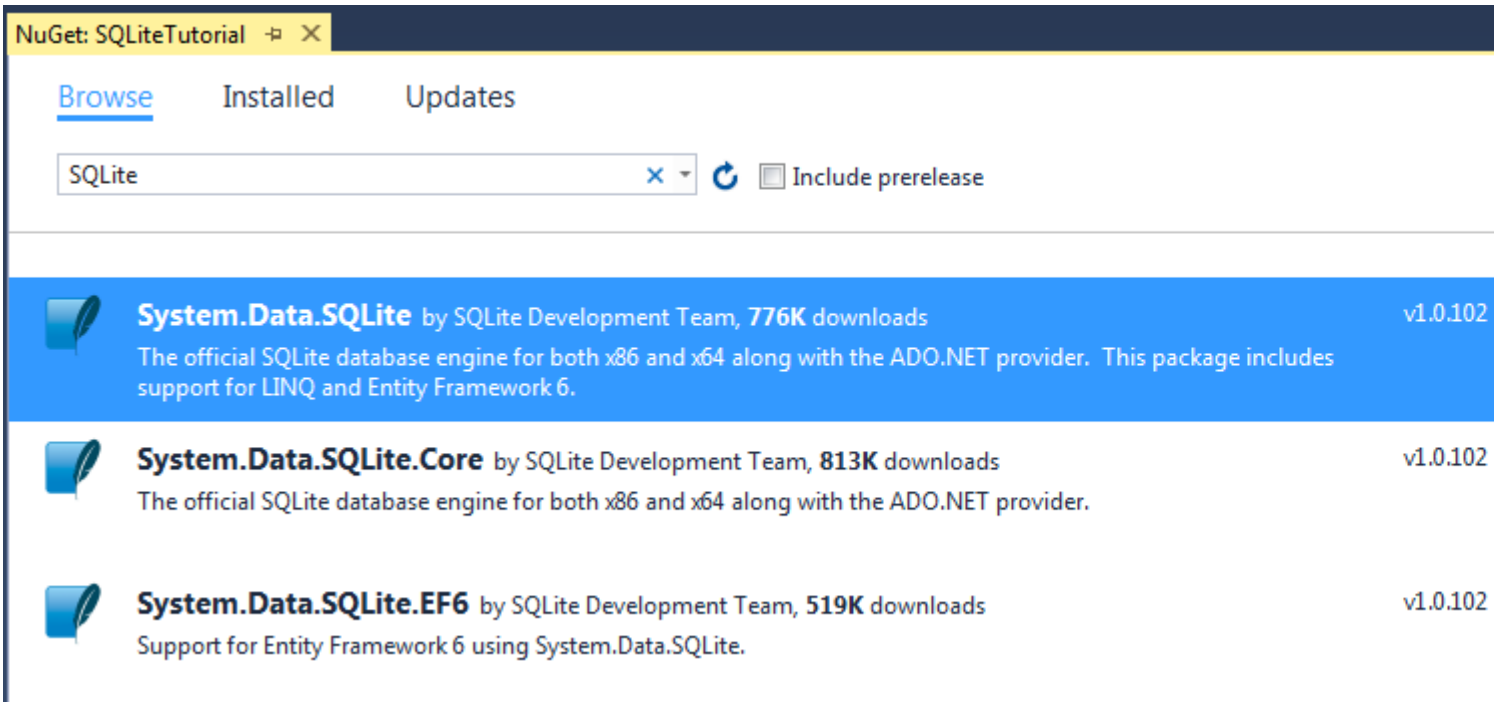
- Descargue DLL que se adapte a su sistema desde la página de descarga de SQLite y luego agregue al proyecto manualmente
- Añadir la dependencia de SQLite a través de NuGet

Lo haremos de la segunda manera.

Primero abre el menú de NuGet




y busque **System.Data.SQLite** , selecciónelo y presione **Instalar**




La instalación también se puede hacer desde la Consola de Package Manager con

```
PM> Install-Package System.Data.SQLite
```

O sólo para las características básicas

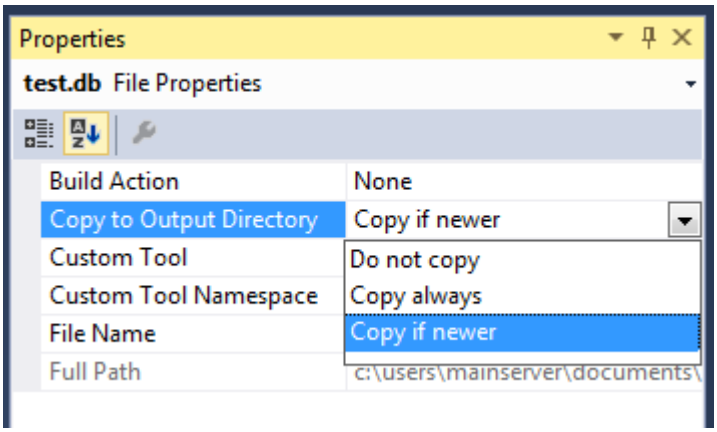
```
PM> Install-Package System.Data.SQLite.Core
```

Eso es todo para la descarga, por lo que podemos ir directamente a la codificación.

Primero cree una base de datos SQLite simple con esta tabla y agréguela como un archivo al proyecto

```
CREATE TABLE User(
  Id INTEGER PRIMARY KEY AUTOINCREMENT,
  FirstName TEXT NOT NULL,
  LastName TEXT NOT NULL
);
```

Además, no olvide configurar la propiedad **Copiar en el directorio de salida** del archivo a **Copiar si es más nuevo** en **Copiar siempre** , según sus necesidades




Cree una clase llamada Usuario, que será la entidad base para nuestra base de datos

```
private class User
{
    public string FirstName { get; set; }
    public string Lastname { get; set; }
}
```

Escribiremos dos métodos para la ejecución de consultas, el primero para insertar, actualizar o eliminar de la base de datos

```
private int ExecuteWrite(string query, Dictionary<string, object> args)
{
    int numberOfRowsAffected;

    //setup the connection to the database
    using (var con = new SQLiteConnection("Data Source=test.db"))
    {
        con.Open();

        //open a new command
        using (var cmd = new SQLiteCommand(query, con))
        {
            //set the arguments given in the query
            foreach (var pair in args)
            {
                cmd.Parameters.AddWithValue(pair.Key, pair.Value);
            }

            //execute the query and get the number of row affected
            numberOfRowsAffected = cmd.ExecuteNonQuery();
        }

        return numberOfRowsAffected;
    }
}
```

y el segundo para leer de la base de datos.

```
private DataTable Execute(string query)
{
    if (string.IsNullOrEmpty(query.Trim()))
        return null;
```

```
    using (var con = new SQLiteConnection("Data Source=test.db"))
    {
        con.Open();
        using (var cmd = new SQLiteCommand(query, con))
        {
            foreach (KeyValuePair<string, object> entry in args)
            {
                cmd.Parameters.AddWithValue(entry.Key, entry.Value);
            }

            var da = new SQLiteDataAdapter(cmd);

            var dt = new DataTable();
            da.Fill(dt);

            da.Dispose();
            return dt;
        }
    }
}
```

Ahora vamos a entrar en nuestros métodos **CRUD**

Añadiendo usuario

```
private int AddUser(User user)
{
    const string query = "INSERT INTO User(FirstName, LastName) VALUES(@firstName,
@lastName)";

    //here we are setting the parameter values that will be actually
    //replaced in the query in Execute method
    var args = new Dictionary<string, object>
    {
        {"@firstName", user.FirstName},
        {"@lastName", user.Lastname}
    };

    return ExecuteWrite(query, args);
}
```

Usuario de edición

```
private int EditUser(User user)
{
    const string query = "UPDATE User SET FirstName = @firstName, LastName = @lastName WHERE
Id = @id";

    //here we are setting the parameter values that will be actually
    //replaced in the query in Execute method
    var args = new Dictionary<string, object>
    {
        {"@id", user.Id},
        {"@firstName", user.FirstName},
        {"@lastName", user.Lastname}
    };

    return ExecuteWrite(query, args);
```

```
}
```

### Borrando usuario

```
private int DeleteUser(User user)
{
    const string query = "Delete from User WHERE Id = @id";

    //here we are setting the parameter values that will be actually
    //replaced in the query in Execute method
    var args = new Dictionary<string, object>
    {
        {"@id", user.Id}
    };

    return ExecuteWrite(query, args);
}
```

### Obtención de usuario por `Id`

```
private User GetUserById(int id)
{
    var query = "SELECT * FROM User WHERE Id = @id";

    var args = new Dictionary<string, object>
    {
        {"@id", id}
    };

    DataTable dt = ExecuteRead(query, args);

    if (dt == null || dt.Rows.Count == 0)
    {
        return null;
    }

    var user = new User
    {
        Id = Convert.ToInt32(dt.Rows[0]["Id"]),
        FirstName = Convert.ToString(dt.Rows[0]["FirstName"]),
        Lastname = Convert.ToString(dt.Rows[0]["LastName"])
    };

    return user;
}
```

## Ejecutando consulta

```
using (SQLiteConnection conn = new SQLiteConnection(@"Data
Source=data.db;Pooling=true;FailIfMissing=false"))
{
    conn.Open();
    using (SQLiteCommand cmd = new SQLiteCommand(conn))
    {
       cmd.CommandText = "query";
       using (SqlDataReader dr = cmd.ExecuteReader())
       {
```

```
            while(dr.Read())
            {
                //do stuff
            }
        }
    }
}
```

*Nota* : si se establece `FailIfMissing` en true, se crea el archivo `data.db` si falta. Sin embargo, el archivo estará vacío. Por lo tanto, cualquier tabla requerida debe ser recreada.

Lea Usando SQLite en C # en línea: https://riptutorial.com/es/csharp/topic/4960/usando-sqlite-en-c-sharp

# Capítulo 162: Utilizando la declaración

## Introducción

Proporciona una sintaxis conveniente que asegura el uso correcto de los objetos IDisponibles .

## Sintaxis

- utilizando (desechable) {}
- utilizando (IDisposable desechable = nuevo MyDisposable ()) {}

## Observaciones

El objeto en la declaración de `using` debe implementar la interfaz `IDisposable` .

```
using(var obj = new MyObject())
{
}

class MyObject : IDisposable
{
    public void Dispose()
    {
        // Cleanup
    }
}
```

Se pueden encontrar ejemplos más completos para la implementación `IDisposable` en los documentos de MSDN .

## Examples

### Uso de los fundamentos de la declaración

`using` azúcar sintáctica le permite garantizar que un recurso se limpie sin necesidad de un bloque explícito de `try-finally` . Esto significa que su código será mucho más limpio y no perderá recursos no administrados.

El patrón estándar de limpieza de `Dispose` , para los objetos que implementan la interfaz `IDisposable` (que la clase base `Stream` `FileStream` hace en .NET):

```
int Foo()
{
    var fileName = "file.txt";

    {
        FileStream disposable = null;
```

```
        try
        {
            disposable = File.Open(fileName, FileMode.Open);

            return disposable.ReadByte();
        }
        finally
        {
            // finally blocks are always run
            if (disposable != null) disposable.Dispose();
        }
    }
}
```

`using` simplifica su sintaxis al ocultar el `try-finally` explícito `try-finally` :

```
int Foo()
{
    var fileName = "file.txt";

    using (var disposable = File.Open(fileName, FileMode.Open))
    {
        return disposable.ReadByte();
    }
    // disposable.Dispose is called even if we return earlier
}
```

Al igual que, `finally` bloques siempre se ejecutan independientemente de los errores o devoluciones, `using` siempre las llamadas `Dispose()` , incluso en caso de error:

```
int Foo()
{
    var fileName = "file.txt";

    using (var disposable = File.Open(fileName, FileMode.Open))
    {
        throw new InvalidOperationException();
    }
    // disposable.Dispose is called even if we throw an exception earlier
}
```

**Nota:** dado que se garantiza que `Dispose` se llame independientemente del flujo de código, es una buena idea asegurarse de que `Dispose` nunca arroje una excepción cuando implemente `IDisposable` . De lo contrario, una excepción real quedaría anulada por la nueva excepción que resultaría en una pesadilla de depuración.

# Volviendo de usar bloque

```
using ( var disposable = new DisposableItem() )
{
    return disposable.SomeProperty;
}
```

Debido a la semántica de `try..finally` a la que se traduce el bloque de `using` , la declaración de

`return` funciona como se esperaba: el valor de retorno se evalúa antes de que `finally` se ejecute el bloque y se elimine el valor. El orden de evaluación es el siguiente:

1. Evaluar el cuerpo de `try`
2. Evaluar y almacenar en caché el valor devuelto
3. Ejecutar finalmente el bloque
4. Devuelve el valor de retorno en caché

Sin embargo, no puede devolver la variable `disposable` sí misma, ya que contendría una referencia desechada no válida; consulte el ejemplo relacionado .

## Múltiples declaraciones usando un bloque

Es posible utilizar múltiples anidadas `using` declaraciones sin agregados múltiples niveles de tirantes anidados. Por ejemplo:

```
using (var input = File.OpenRead("input.txt"))
{
    using (var output = File.OpenWrite("output.txt"))
    {
        input.CopyTo(output);
    } // output is disposed here
} // input is disposed here
```

Una alternativa es escribir:

```
using (var input = File.OpenRead("input.txt"))
using (var output = File.OpenWrite("output.txt"))
{
    input.CopyTo(output);
} // output and then input are disposed here
```

Que es exactamente equivalente al primer ejemplo.

*Nota: las* declaraciones de `using` anidadas pueden activar la regla de análisis de código de Microsoft CS2002 (consulte esta respuesta para obtener una aclaración) y generar una advertencia. Como se explica en la respuesta vinculada, generalmente es seguro anidar `using` declaraciones.

Cuando los tipos dentro de la declaración de `using` son del mismo tipo, puede delimitarlos con comas y especificar el tipo solo una vez (aunque esto no es común):

```
using (FileStream file = File.Open("MyFile.txt"), file2 = File.Open("MyFile2.txt"))
{
}
```

Esto también se puede utilizar cuando los tipos tienen una jerarquía compartida:

```
using (Stream file = File.Open("MyFile.txt"), data = new MemoryStream())
{
}
```

La palabra clave `var` *no se* puede utilizar en el ejemplo anterior. Se produciría un error de compilación. Incluso la declaración separada por comas no funcionará cuando las variables declaradas tengan tipos de diferentes jerarquías.

## Gotcha: devolviendo el recurso que estas tirando.

La siguiente es una mala idea porque eliminaría la variable `db` antes de devolverla.

```
public IDBContext GetDBContext()
{
    using (var db = new DBContext())
    {
        return db;
    }
}
```

Esto también puede crear errores más sutiles:

```
public IEnumerable<Person> GetPeople(int age)
{
    using (var db = new DBContext())
    {
        return db.Persons.Where(p => p.Age == age);
    }
}
```

Esto se ve bien, pero el problema es que la evaluación de la expresión LINQ es perezosa, y posiblemente solo se ejecutará más adelante cuando el `DBContext` subyacente ya haya sido eliminado.

En resumen, la expresión no se evalúa antes de dejar de `using` . Una posible solución a este problema, que aún utiliza el `using` , es hacer que la expresión se evalúe de inmediato llamando a un método que enumere el resultado. Por ejemplo, `ToList()` , `ToArray()` , etc. Si está utilizando la versión más reciente de Entity Framework, podría usar los equivalentes `async` como `ToListAsync()` o `ToArrayAsync()` .

A continuación encontrará el ejemplo en acción:

```
public IEnumerable<Person> GetPeople(int age)
{
    using (var db = new DBContext())
    {
        return db.Persons.Where(p => p.Age == age).ToList();
    }
}
```

Sin embargo, es importante tener en cuenta que al llamar a `ToList()` o `ToArray()` , la expresión se evaluará con entusiasmo, lo que significa que todas las personas con la edad especificada se cargarán en la memoria incluso si no las itera.

## Las declaraciones de uso son nulas seguras

No es necesario comprobar el objeto `IDisposable` para `null` . `using` no lanzará una excepción y no se llamará `Dispose()` :

```
DisposableObject TryOpenFile()
{
    return null;
}

// disposable is null here, but this does not throw an exception
using (var disposable = TryOpenFile())
{
    // this will throw a NullReferenceException because disposable is null
    disposable.DoSomething();

    if(disposable != null)
    {
        // here we are safe because disposable has been checked for null
        disposable.DoSomething();
    }
}
```

## Gotcha: Excepción en el método de Disposición que enmascara otros errores en el uso de bloques

Considere el siguiente bloque de código.

```
try
{
    using (var disposable = new MyDisposable())
    {
        throw new Exception("Couldn't perform operation.");
    }
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}

class MyDisposable : IDisposable
{
    public void Dispose()
    {
        throw new Exception("Couldn't dispose successfully.");
    }
}
```

Puede esperar ver "No se pudo realizar la operación" impresa en la Consola, pero en realidad vería "No se pudo desechar con éxito". ya que aún se llama al método Dispose incluso después de que se lanza la primera excepción.

Vale la pena ser consciente de esta sutileza, ya que puede estar ocultando el error real que evitó que el objeto se eliminara y dificultara la depuración.

## Uso de declaraciones y conexiones de base de datos

La palabra clave `using` garantiza que el recurso definido dentro de la declaración solo exista dentro del alcance de la declaración. Cualquier recurso definido dentro de la declaración debe implementar la interfaz `IDisposable` .

Estos son increíblemente importantes cuando se trata de conexiones que implementan la interfaz `IDisposable` , ya que puede garantizar que las conexiones no solo se cierren correctamente, sino que sus recursos se liberen después de que la declaración de `using` esté fuera del alcance.

## Clases de datos `IDisposable` comunes

Muchas de las siguientes son clases relacionadas con datos que implementan la interfaz `IDisposable` y son candidatos perfectos para una declaración de `using` :

- `SqlConnection` , `SqlCommand` , `SqlDataReader` , etc.
- `OleDbConnection` , `OleDbCommand` , `OleDbDataReader` , etc.
- `MySqlConnection` , `MySqlCommand` , `MySqlDbDataReader` , etc.
- `DbContext`

Todos estos se usan comúnmente para acceder a los datos a través de C # y se encontrarán comúnmente en las aplicaciones de creación de datos centradas. Se puede esperar que muchas otras clases que no se mencionan y que implementan las mismas `FooConnection` , `FooCommand` , `FooDataReader` se comporten de la misma manera.

## Patrón de acceso común para conexiones ADO.NET

Un patrón común que se puede usar al acceder a sus datos a través de una conexión ADO.NET puede tener el siguiente aspecto:

```
// This scopes the connection (your specific class may vary)
using(var connection = new SqlConnection("{your-connection-string}")
{
    // Build your query
    var query = "SELECT * FROM YourTable WHERE Property = @property");
    // Scope your command to execute
    using(var command = new SqlCommand(query, connection))
    {
         // Open your connection
         connection.Open();

         // Add your parameters here if necessary

         // Execute your query as a reader (again scoped with a using statement)
         using(var reader = command.ExecuteReader())
         {
               // Iterate through your results here
         }
    }
}
```

O si solo estuvieras realizando una actualización simple y no necesitaras un lector, se aplicaría el mismo concepto básico:

```
using(var connection = new SqlConnection("{your-connection-string}"))
{
     var query = "UPDATE YourTable SET Property = Value WHERE Foo = @foo";
     using(var command = new SqlCommand(query,connection))
     {
          connection.Open();

          // Add parameters here

          // Perform your update
          command.ExecuteNonQuery();
     }
}
```

# Uso de declaraciones con DataContexts

Muchos ORM, como Entity Framework, exponen las clases de abstracción que se utilizan para interactuar con las bases de datos subyacentes en forma de clases como `DbContext` . Estos contextos generalmente implementan también la interfaz `IDisposable` y deberían aprovechar esto mediante el `using` declaraciones cuando sea posible:

```
using(var context = new YourDbContext())
{
      // Access your context and perform your query
      var data = context.Widgets.ToList();
}
```

## Usando la sintaxis de Dispose para definir el alcance personalizado

Para algunos casos de uso, puede usar la sintaxis de `using` para ayudar a definir un ámbito personalizado. Por ejemplo, puede definir la siguiente clase para ejecutar código en una cultura específica.

```
public class CultureContext : IDisposable
{
    private readonly CultureInfo originalCulture;

    public CultureContext(string culture)
    {
        originalCulture = CultureInfo.CurrentCulture;
        Thread.CurrentThread.CurrentCulture = new CultureInfo(culture);
    }

    public void Dispose()
    {
        Thread.CurrentThread.CurrentCulture = originalCulture;
    }
}
```

Luego puede usar esta clase para definir bloques de código que se ejecutan en una cultura específica.

```
Thread.CurrentThread.CurrentCulture = new CultureInfo("en-US");
```

```
using (new CultureContext("nl-NL"))
{
    // Code in this block uses the "nl-NL" culture
    Console.WriteLine(new DateTime(2016, 12, 25)); // Output: 25-12-2016 00:00:00
}

using (new CultureContext("es-ES"))
{
    // Code in this block uses the "es-ES" culture
    Console.WriteLine(new DateTime(2016, 12, 25)); // Output: 25/12/2016 0:00:00
}

// Reverted back to the original culture
Console.WriteLine(new DateTime(2016, 12, 25)); // Output: 12/25/2016 12:00:00 AM
```

Nota: como no usamos la instancia de `CultureContext` que creamos, no le asignamos una variable.

Esta técnica es utilizada por el ayudante `BeginForm` en ASP.NET MVC.

## Ejecución de código en contexto de restricción

Si tiene un código (una *rutina* ) que desea ejecutar en un contexto específico (restricción), puede usar la inyección de dependencia.

El siguiente ejemplo muestra la restricción de ejecutar bajo una conexión SSL abierta. Esta primera parte estaría en su biblioteca o marco, que no expondrá al código del cliente.

```
public static class SSLContext
{
    // define the delegate to inject
    public delegate void TunnelRoutine(BinaryReader sslReader, BinaryWriter sslWriter);

    // this allows the routine to be executed under SSL
    public static void ClientTunnel(TcpClient tcpClient, TunnelRoutine routine)
    {
        using (SslStream sslStream = new SslStream(tcpClient.GetStream(), true, _validate))
        {
            sslStream.AuthenticateAsClient(HOSTNAME, null, SslProtocols.Tls, false);

            if (!sslStream.IsAuthenticated)
            {
                throw new SecurityException("SSL tunnel not authenticated");
            }

            if (!sslStream.IsEncrypted)
            {
                throw new SecurityException("SSL tunnel not encrypted");
            }

            using (BinaryReader sslReader = new BinaryReader(sslStream))
            using (BinaryWriter sslWriter = new BinaryWriter(sslStream))
            {
                routine(sslReader, sslWriter);
            }
        }
    }
```

```
}
```

Ahora el código de cliente que quiere hacer algo bajo SSL pero no quiere manejar todos los detalles de SSL. Ahora puede hacer lo que quiera dentro del túnel SSL, por ejemplo, intercambiar una clave simétrica:

```
public void ExchangeSymmetricKey(BinaryReader sslReader, BinaryWriter sslWriter)
{
    byte[] bytes = new byte[8];
    (new RNGCryptoServiceProvider()).GetNonZeroBytes(bytes);
    sslWriter.Write(BitConverter.ToUInt64(bytes, 0));
}
```

Ejecuta esta rutina de la siguiente manera:

```
SSLContext.ClientTunnel(tcpClient, this.ExchangeSymmetricKey);
```

Para hacer esto, necesita la cláusula `using()` porque es la única manera (aparte de un bloque `try..finally` ) puede garantizar que el código del cliente ( `ExchangeSymmetricKey` ) nunca salga sin disponer de los recursos desechables. Sin `using()` cláusula `using()` , nunca sabría si una rutina podría romper la restricción del contexto para disponer de esos recursos.

Lea Utilizando la declaración en línea: https://riptutorial.com/es/csharp/topic/38/utilizando-la-declaracion

# Capítulo 163: XDocument y el espacio de nombres System.Xml.Linq

## Examples

### Generar un documento XML

El objetivo es generar el siguiente documento XML:

```
<FruitBasket xmlns="http://www.fruitauthority.fake">
  <Fruit ID="F0001">
    <FruitName>Banana</FruitName>
    <FruitColor>Yellow</FruitColor>
  </Fruit>
  <Fruit ID="F0002">
    <FruitName>Apple</FruitName>
    <FruitColor>Red</FruitColor>
  </Fruit>
</FruitBasket>
```

Código:

```
XNamespace xns = "http://www.fruitauthority.fake";
XDeclaration xDeclaration = new XDeclaration("1.0", "utf-8", "yes");
XDocument xDoc = new XDocument(xDeclaration);
XElement xRoot = new XElement(xns + "FruitBasket");
xDoc.Add(xRoot);

XElement xelFruit1 = new XElement(xns + "Fruit");
XAttribute idAttribute1 = new XAttribute("ID", "F0001");
xelFruit1.Add(idAttribute1);
XElement xelFruitName1 = new XElement(xns + "FruitName", "Banana");
XElement xelFruitColor1 = new XElement(xns + "FruitColor", "Yellow");
xelFruit1.Add(xelFruitName1);
xelFruit1.Add(xelFruitColor1);
xRoot.Add(xelFruit1);

XElement xelFruit2 = new XElement(xns + "Fruit");
XAttribute idAttribute2 = new XAttribute("ID", "F0002");
xelFruit2.Add(idAttribute2);
XElement xelFruitName2 = new XElement(xns + "FruitName", "Apple");
XElement xelFruitColor2 = new XElement(xns + "FruitColor", "Red");
xelFruit2.Add(xelFruitName2);
xelFruit2.Add(xelFruitColor2);
xRoot.Add(xelFruit2);
```

### Modificar archivo XML

Para modificar un archivo XML con `XDocument` , cargue el archivo en una variable de tipo `XDocument` , `XDocument` en la memoria y luego guárdelo, sobrescribiendo el archivo original. Un error común es modificar el XML en la memoria y esperar que el archivo en el disco cambie.

Dado un archivo XML:

```
<?xml version="1.0" encoding="utf-8"?>
<FruitBasket xmlns="http://www.fruitauthority.fake">
  <Fruit>
    <FruitName>Banana</FruitName>
    <FruitColor>Yellow</FruitColor>
  </Fruit>
  <Fruit>
    <FruitName>Apple</FruitName>
    <FruitColor>Red</FruitColor>
  </Fruit>
</FruitBasket>
```

Quieres modificar el color del plátano a marrón:

1. Necesitamos saber la ruta al archivo en el disco.
2. Una sobrecarga de `XDocument.Load` recibe un URI (ruta del archivo).
3. Dado que el archivo xml utiliza un espacio de nombres, debemos consultar con el espacio de nombres Y el nombre del elemento.
4. Una consulta de Linq que utiliza la sintaxis de C # 6 para adaptarse a la posibilidad de valores nulos. Cada `.` utilizado en esta consulta tiene el potencial de devolver un conjunto nulo si la condición no encuentra elementos. Antes de C # 6, haría esto en varios pasos, comprobando si hay nulos en el camino. El resultado es el elemento `<Fruit>` que contiene el plátano. En realidad, un `IEnumerable<XElement>` , razón por la cual el siguiente paso usa `FirstOfDefault()` .
5. Ahora extraemos el elemento FruitColor del elemento Fruit que acabamos de encontrar. Aquí asumimos que solo hay uno, o solo nos importa el primero.
6. Si no es nulo, configuramos FruitColor en "Marrón".
7. Finalmente, guardamos el documento `XDocument` , sobrescribiendo el archivo original en el disco.

```
// 1.
string xmlFilePath = "c:\\users\\public\\fruit.xml";

// 2.
XDocument xdoc = XDocument.Load(xmlFilePath);

// 3.
XNamespace ns = "http://www.fruitauthority.fake";

//4.
var elBanana = xdoc.Descendants()?.
    Elements(ns + "FruitName")?.
    Where(x => x.Value == "Banana")?.
    Ancestors(ns + "Fruit");

// 5.
var elColor = elBanana.Elements(ns + "FruitColor").FirstOrDefault();

// 6.
if (elColor != null)
{
    elColor.Value = "Brown";
```

```
}

// 7.
xdoc.Save(xmlFilePath);
```

El archivo ahora se ve así:

```
<?xml version="1.0" encoding="utf-8"?>
<FruitBasket xmlns="http://www.fruitauthority.fake">
  <Fruit>
    <FruitName>Banana</FruitName>
    <FruitColor>Brown</FruitColor>
  </Fruit>
  <Fruit>
    <FruitName>Apple</FruitName>
    <FruitColor>Red</FruitColor>
  </Fruit>
</FruitBasket>
```

## Generar un documento XML usando sintaxis fluida

Gol:

```
<FruitBasket xmlns="http://www.fruitauthority.fake">
  <Fruit>
    <FruitName>Banana</FruitName>
    <FruitColor>Yellow</FruitColor>
  </Fruit>
  <Fruit>
    <FruitName>Apple</FruitName>
    <FruitColor>Red</FruitColor>
  </Fruit>
</FruitBasket>
```

Código:

```
XNamespace xns = "http://www.fruitauthority.fake";
XDocument xDoc =
    new XDocument(new XDeclaration("1.0", "utf-8", "yes"),
        new XElement(xns + "FruitBasket",
            new XElement(xns + "Fruit",
                new XElement(xns + "FruitName", "Banana"),
                new XElement(xns + "FruitColor", "Yellow")),
            new XElement(xns + "Fruit",
                new XElement(xns + "FruitName", "Apple"),
                new XElement(xns + "FruitColor", "Red"))
                ));
```

Lea XDocument y el espacio de nombres System.Xml.Linq en línea:
https://riptutorial.com/es/csharp/topic/1866/xdocument-y-el-espacio-de-nombres-system-xml-linq

# Capítulo 164: XmlDocument y el espacio de nombres System.Xml

## Examples

### Interacción de documentos XML básicos

```
public static void Main()
{
    var xml  = new XmlDocument();
    var root = xml.CreateElement("element");
        // Creates an attribute, so the element will now be "<element attribute='value' />"
        root.SetAttribute("attribute", "value");

    // All XML documents must have one, and only one, root element
    xml.AppendChild(root);

    // Adding data to an XML document
    foreach (var dayOfWeek in Enum.GetNames((typeof(DayOfWeek))))
    {
        var day = xml.CreateElement("dayOfWeek");
            day.SetAttribute("name", dayOfWeek);

        // Don't forget to add the new value to the current document!
        root.AppendChild(day);
    }

    // Looking for data using XPath; BEWARE, this is case-sensitive
    var monday = xml.SelectSingleNode("//dayOfWeek[@name='Monday']");
    if (monday != null)
    {
        // Once you got a reference to a particular node, you can delete it
        // by navigating through its parent node and asking for removal
        monday.ParentNode.RemoveChild(monday);
    }

    // Displays the XML document in the screen; optionally can be saved to a file
    xml.Save(Console.Out);
}
```

### Leyendo del documento XML

Un ejemplo de archivo XML

```
    <Sample>
    <Account>
        <One number="12"/>
        <Two number="14"/>
    </Account>
    <Account>
        <One number="14"/>
        <Two number="16"/>
    </Account>
```

```
    </Sample>
```

Leyendo de este archivo XML:

```
    using System.Xml;
    using System.Collections.Generic;

    public static void Main(string fullpath)
    {
        var xmldoc = new XmlDocument();
        xmldoc.Load(fullpath);

        var oneValues = new List<string>();

        // Getting all XML nodes with the tag name
        var accountNodes = xmldoc.GetElementsByTagName("Account");
        for (var i = 0; i < accountNodes.Count; i++)
        {
            // Use Xpath to find a node
            var account = accountNodes[i].SelectSingleNode("./One");
            if (account != null && account.Attributes != null)
            {
                // Read node attribute
                oneValues.Add(account.Attributes["number"].Value);
            }
        }
}
```

## XmlDocument vs XDocument (Ejemplo y comparación)

Hay varias formas de interactuar con un archivo Xml.

1. Documento xml
2. XDocumento
3. XmlReader / XmlWriter

Antes de LINQ to XML, usábamos XMLDocument para manipulaciones en XML, como agregar atributos, elementos, etc. Ahora LINQ to XML usa XDocument para el mismo tipo de cosas. Las sintaxis son mucho más fáciles que XMLDocument y requieren una cantidad mínima de código.

También XDocument es mutch más rápido como XmlDocument. XmlDoucument es una solución antigua y sucia para consultar un documento XML.

**Voy a mostrar algunos ejemplos de la clase XmlDocument y la clase XDocument :**

**Cargar archivo XML**

```
string filename = @"C:\temp\test.xml";
```

**XmlDocument**

```
XmlDocument _doc = new XmlDocument();
_doc.Load(filename);
```

### XDocumento

```
XDocument _doc = XDocument.Load(fileName);
```

## Crear XmlDocument

### XmlDocument

```
XmlDocument doc = new XmlDocument();
XmlElement root = doc.CreateElement("root");
root.SetAttribute("name", "value");
XmlElement child = doc.CreateElement("child");
child.InnerText = "text node";
root.AppendChild(child);
doc.AppendChild(root);
```

### XDocumento

```
 XDocument doc = new XDocument(
    new XElement("Root", new XAttribute("name", "value"),
    new XElement("Child", "text node"))
);

/*result*/
<root name="value">
    <child>"TextNode"</child>
</root>
```

## Cambiar InnerText del nodo en XML

### XmlDocument

```
XmlNode node = _doc.SelectSingleNode("xmlRootNode");
node.InnerText = value;
```

### XDocumento

```
 XElement rootNote = _doc.XPathSelectElement("xmlRootNode");
rootNode.Value = "New Value";
```

## Guardar archivo después de editar

Asegúrese de proteger el xml después de cualquier cambio.

```
// Safe XmlDocument and XDocument
_doc.Save(filename);
```

## Valores de recuperación de XML

### XmlDocument

```
 XmlNode node = _doc.SelectSingleNode("xmlRootNode/levelOneChildNode");
string text = node.InnerText;
```

### XDocumento

```
 XElement node = _doc.XPathSelectElement("xmlRootNode/levelOneChildNode");
 string text = node.Value;
```

## Recupere el valor de todos de todos los elementos secundarios donde atributo = algo.

### XmlDocument

```
List<string> valueList = new List<string>();
    foreach (XmlNode n in nodelist)
    {
        if(n.Attributes["type"].InnerText == "City")
        {
            valueList.Add(n.Attributes["type"].InnerText);
        }
    }
```

### XDocumento

```
var accounts = _doc.XPathSelectElements("/data/summary/account").Where(c =>
c.Attribute("type").Value == "setting").Select(c => c.Value);
```

## Anexar un nodo

### XmlDocument

```
XmlNode nodeToAppend = doc.CreateElement("SecondLevelNode");
nodeToAppend.InnerText = "This title is created by code";

/* Append node to parent */
XmlNode firstNode= _doc.SelectSingleNode("xmlRootNode/levelOneChildNode");
firstNode.AppendChild(nodeToAppend);

/*After a change make sure to safe the document*/
_doc.Save(fileName);
```

### XDocumento

```
_doc.XPathSelectElement("ServerManagerSettings/TcpSocket").Add(new
XElement("SecondLevelNode"));

 /*After a change make sure to safe the document*/
_doc.Save(fileName);
```

Lea XmlDocument y el espacio de nombres System.Xml en línea:

https://riptutorial.com/es/csharp/topic/1528/xmldocument-y-el-espacio-de-nombres-system-xml

# Capítulo 165: Zócalo asíncrono

## Introducción

Al usar sockets asíncronos, un servidor puede escuchar las conexiones entrantes y, mientras tanto, hacer alguna otra lógica, en contraste con el socket síncrono cuando están escuchando, bloquean el hilo principal y la aplicación deja de responder y se congelará hasta que el cliente se conecte.

## Observaciones

**Zócalo y red**

¿Cómo acceder a un servidor fuera de mi propia red? Esta es una pregunta común y cuando se la pregunta se marca principalmente como tema.

**Lado del servidor**

En la red de su servidor necesita reenviar su enrutador a su servidor.

Por ejemplo, PC donde el servidor se está ejecutando en:

IP local = `192.168.1.115`

El servidor está escuchando el puerto 1234.

Reenvíe las conexiones entrantes en el enrutador del `Port 1234` al `192.168.1.115`

**Lado del cliente**

Lo único que necesitas cambiar es la IP. No desea conectarse a su dirección de bucle de retorno sino a la IP pública de la red en la que se ejecuta su servidor. Esta IP la puedes obtener aquí .

```
_connectingSocket.Connect(new IPEndPoint(IPAddress.Parse("10.10.10.10"), 1234));
```

Así que ahora creas una solicitud en este punto final: `10.10.10.10:1234` si hiciste que el puerto de propiedad reenvíe tu enrutador, tu servidor y el cliente se conectarán sin ningún problema.

Si desea conectarse a una IP local, no tendrá que portforwart, simplemente cambie la dirección de loopback a `192.168.1.178` o algo así.

**Enviando datos:**

Los datos se envían en una matriz de bytes. Debe empaquetar sus datos en una matriz de bytes y descomprimirlos en el otro lado.

Si está familiarizado con el socket, también puede intentar cifrar su matriz de bytes antes de

enviar. Esto evitará que alguien robe tu paquete.

# Examples

## Ejemplo de Socket Asíncrono (Cliente / Servidor).

### Ejemplo del lado del servidor

**Crear escucha para el servidor**

Comience con la creación de un servidor que maneje los clientes que se conecten y las solicitudes que se enviarán. Así que crea una clase de oyente que manejará esto.

```
class Listener
{
    public Socket ListenerSocket; //This is the socket that will listen to any incoming
connections
    public short Port = 1234; // on this port we will listen

    public Listener()
    {
        ListenerSocket = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
ProtocolType.Tcp);
    }
 }
```

Primero necesitamos inicializar el socket de escucha donde podemos escuchar cualquier conexión. Vamos a usar un Socket Tcp, por eso usamos SocketType.Stream. También especificamos a qué puerto debe escuchar el servidor.

Entonces comenzamos a escuchar cualquier conexión entrante.

**Los métodos de árbol que usamos aquí son:**

1. ListenerSocket.Bind ();

   Este método une el socket a un IPEndPoint . Esta clase contiene el host y la información del puerto local o remoto que necesita una aplicación para conectarse a un servicio en un host.

2. ListenerSocket.Listen (10);

   El parámetro backlog especifica el número de conexiones entrantes que pueden ponerse en cola para su aceptación.

3. ListenerSocket.BeginAccept ();

   El servidor comenzará a escuchar las conexiones entrantes y continuará con otra lógica. Cuando hay una conexión, el servidor vuelve a este método y ejecutará el método AcceptCallBack

```
    public void StartListening()
```

```
    {
        try
        {
                MessageBox.Show($"Listening started port:{Port} protocol type:
{ProtocolType.Tcp}");
                ListenerSocket.Bind(new IPEndPoint(IPAddress.Any, Port));
                ListenerSocket.Listen(10);
                ListenerSocket.BeginAccept(AcceptCallback, ListenerSocket);
        }
        catch(Exception ex)
        {
            throw new Exception("listening error" + ex);
        }
    }
```

Así que cuando un cliente se conecta podemos aceptarlos por este método:

**Tres métodos que utilizamos aquí son:**

1. ListenerSocket.EndAccept ()

   Comenzamos la devolución de llamada con `Listener.BeginAccept()` fin ahora que tenemos que terminar esa llamada. `The EndAccept()` método `The EndAccept()` acepta un parámetro IAsyncResult, esto almacenará el estado del método asíncrono. Desde este estado podemos extraer el socket del que provenía la conexión entrante.

2. `ClientController.AddClient()`

   Con el socket que obtuvimos de `EndAccept()` creamos un Cliente con un método propio *(código ClientController debajo del ejemplo del servidor)* .

3. ListenerSocket.BeginAccept ()

   Necesitamos comenzar a escuchar nuevamente cuando el socket haya terminado con el manejo de la nueva conexión. Pase en el método que capturará esta devolución de llamada. Y también pase int en el socket del oyente para que podamos reutilizarlo para las próximas conexiones.

```
    public void AcceptCallback(IAsyncResult ar)
    {
        try
        {
            Console.WriteLine($"Accept CallBack port:{Port} protocol type:
{ProtocolType.Tcp}");
            Socket acceptedSocket = ListenerSocket.EndAccept(ar);
            ClientController.AddClient(acceptedSocket);

            ListenerSocket.BeginAccept(AcceptCallback, ListenerSocket);
        }
        catch (Exception ex)
        {
            throw new Exception("Base Accept error"+ ex);
        }
    }
```

Ahora tenemos un zócalo de escucha, pero ¿cómo recibimos los datos enviados por el cliente que es lo que muestra el siguiente código?

**Crear un receptor de servidor para cada cliente**

Primero, cree una clase de recepción con un constructor que tome un Socket como parámetro:

```
    public class ReceivePacket
    {
        private byte[] _buffer;
        private Socket _receiveSocket;

        public ReceivePacket(Socket receiveSocket)
        {
           _receiveSocket = receiveSocket;
        }
    }
```

En el siguiente método, primero comenzamos con darle al búfer un tamaño de 4 bytes (Int32) o el paquete contiene las partes {longitud, datos reales}. Así que los primeros 4 bytes nos reservamos para la longitud de los datos el resto para los datos reales.

A continuación utilizamos el método BeginReceive () . Este método se utiliza para comenzar a recibir de los clientes conectados y cuando reciba datos ejecutará la función `ReceiveCallback` .

```
    public void StartReceiving()
    {
        try
        {
            _buffer = new byte[4];
            _receiveSocket.BeginReceive(_buffer, 0, _buffer.Length, SocketFlags.None,
ReceiveCallback, null);
        }
        catch {}
    }

    private void ReceiveCallback(IAsyncResult AR)
    {
        try
        {
            // if bytes are less than 1 takes place when a client disconnect from the server.
            // So we run the Disconnect function on the current client
            if (_receiveSocket.EndReceive(AR) > 1)
            {
                // Convert the first 4 bytes (int 32) that we received and convert it to an
Int32 (this is the size for the coming data).
                _buffer = new byte[BitConverter.ToInt32(_buffer, 0)];
                // Next receive this data into the buffer with size that we did receive before
                _receiveSocket.Receive(_buffer, _buffer.Length, SocketFlags.None);
                // When we received everything its onto you to convert it into the data that
you've send.
                // For example string, int etc... in this example I only use the
implementation for sending and receiving a string.

                // Convert the bytes to string and output it in a message box
                string data = Encoding.Default.GetString(_buffer);
                MessageBox.Show(data);
```

```
                // Now we have to start all over again with waiting for a data to come from
the socket.
                StartReceiving();
            }
            else
            {
                Disconnect();
            }
        }
        catch
        {
            // if exeption is throw check if socket is connected because than you can
startreive again else Dissconect
            if (!_receiveSocket.Connected)
            {
                Disconnect();
            }
            else
            {
                StartReceiving();
            }
        }
    }

    private void Disconnect()
    {
        // Close connection
        _receiveSocket.Disconnect(true);
        // Next line only apply for the server side receive
        ClientController.RemoveClient(_clientId);
        // Next line only apply on the Client Side receive
        Here you want to run the method TryToConnect()
    }
```

Así que hemos configurado un servidor que puede recibir y escuchar las conexiones entrantes. Cuando los clientes se conectan, se agregará a una lista de clientes y cada cliente tendrá su propia clase de recepción. Para hacer que el servidor escuche:

```
Listener listener = new Listener();
listener.StartListening();
```

### Algunas clases que uso en este ejemplo

```
    class Client
    {
        public Socket _socket { get; set; }
        public ReceivePacket Receive { get; set; }
        public int Id { get; set; }

        public Client(Socket socket, int id)
        {
            Receive = new ReceivePacket(socket, id);
            Receive.StartReceiving();
            _socket = socket;
            Id = id;
        }
    }
```

```
static class ClientController
{
     public static List<Client> Clients = new List<Client>();

     public static void AddClient(Socket socket)
     {
         Clients.Add(new Client(socket,Clients.Count));
     }

     public static void RemoveClient(int id)
     {
         Clients.RemoveAt(Clients.FindIndex(x => x.Id == id));
     }
 }
```

### Ejemplo del lado del cliente

### Conectando al servidor

En primer lugar, queremos crear una clase que se conecte con el nombre del servidor que le damos: Connector:

```
class Connector
{
    private Socket _connectingSocket;
}
```

El siguiente método para esta clase es TryToConnect ()

Este método goth algunas cosas interesantes:

1. Crea el zócalo;
2. A continuación hago un bucle hasta que el zócalo está conectado.
3. En cada bucle, solo mantiene el subproceso durante 1 segundo, no queremos DOS del servidor XD
4. Con Connect () intentará conectarse al servidor. Si falla, lanzará una excepción, pero el programa mantendrá el programa conectado al servidor. Puedes usar un método Connect CallBack para esto, pero solo iré por llamar a un método cuando el Socket esté conectado.
5. Observe que el Cliente ahora está intentando conectarse a su PC local en el puerto 1234.

```
 public void TryToConnect()
 {
     _connectingSocket = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
ProtocolType.Tcp);

      while (!_connectingSocket.Connected)
      {
          Thread.Sleep(1000);

          try
```

```
          {
              _connectingSocket.Connect(new IPEndPoint(IPAddress.Parse("127.0.0.1"),
1234));
          }
          catch { }
      }
      SetupForReceiveing();
     }
 }

 private void SetupForReceiveing()
 {
    // View Client Class bottom of Client Example
     Client.SetClient(_connectingSocket);
     Client.StartReceiving();
 }
```

**Enviando un mensaje al servidor.**

Así que ahora tenemos una aplicación casi final o Socket. Lo único que no tenemos Jet es una clase para enviar un mensaje al servidor.

```
public class SendPacket
{
    private Socket _sendSocked;

    public SendPacket(Socket sendSocket)
    {
        _sendSocked = sendSocket;
    }

    public void Send(string data)
    {
        try
        {
            /* what hapends here:
                 1. Create a list of bytes
                 2. Add the length of the string to the list.
                    So if this message arrives at the server we can easily read the length of
the coming message.
                 3. Add the message(string) bytes
            */

            var fullPacket = new List<byte>();
            fullPacket.AddRange(BitConverter.GetBytes(data.Length));
            fullPacket.AddRange(Encoding.Default.GetBytes(data));

            /* Send the message to the server we are currently connected to.
            Or package stucture is {length of data 4 bytes (int32), actual data}*/
            _sendSocked.Send(fullPacket.ToArray());
        }
        catch (Exception ex)
        {
            throw new Exception();
        }
    }
```

Finalmente, active dos botones uno para conectar y el otro para enviar un mensaje:

```
    private void ConnectClick(object sender, EventArgs e)
    {
        Connector tpp = new Connector();
        tpp.TryToConnect();
    }

    private void SendClick(object sender, EventArgs e)
    {
        Client.SendString("Test data from client");
    }
```

**La clase cliente que utilicé en este ejemplo.**

```
    public static void SetClient(Socket socket)
    {
        Id = 1;
        Socket = socket;
        Receive = new ReceivePacket(socket, Id);
        SendPacket = new SendPacket(socket);
    }
```

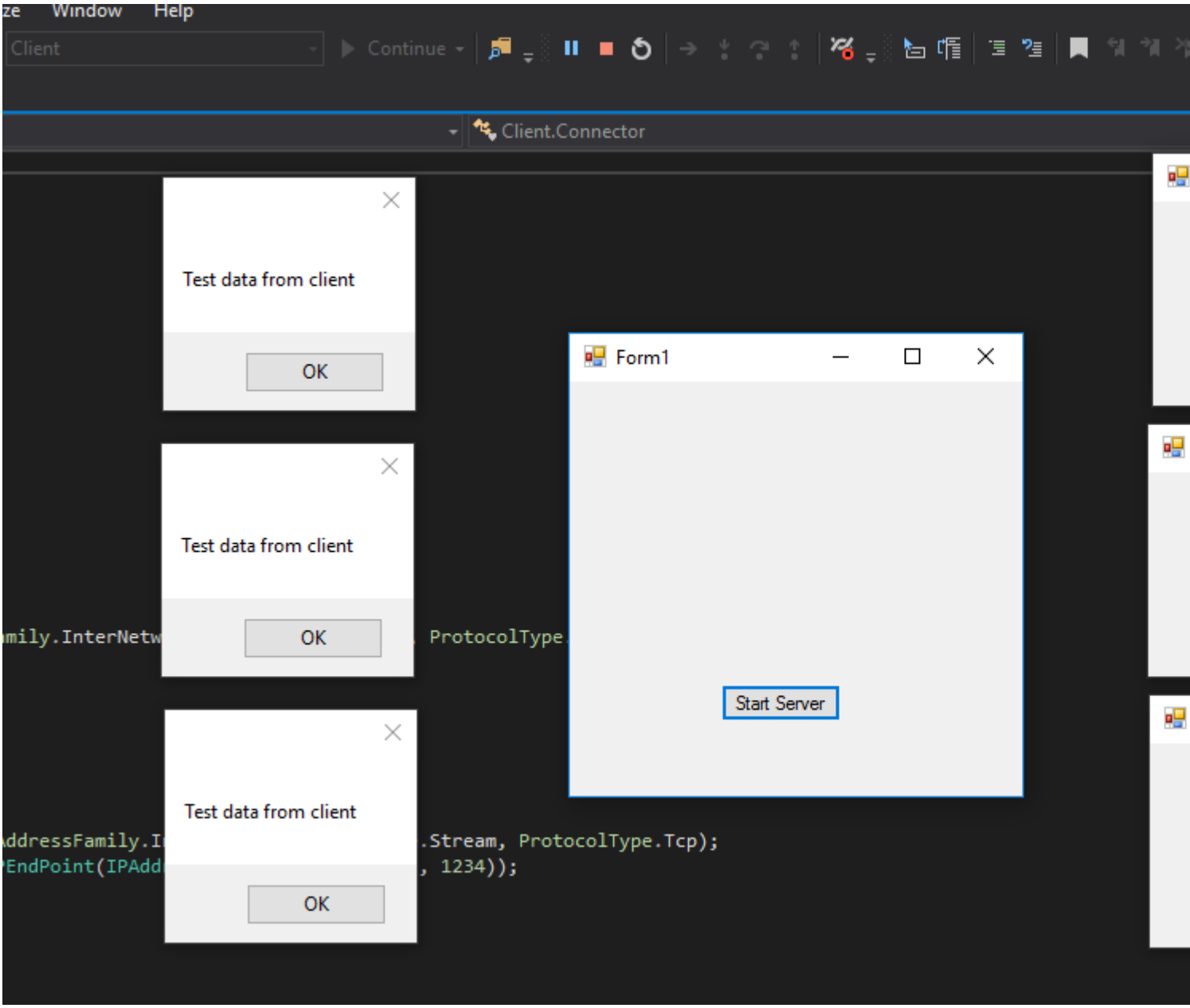
**darse cuenta**

La clase de recepción del servidor es la misma que la clase de recepción del cliente.

### Conclusión

Ahora tienes un servidor y un cliente. Puedes trabajar este ejemplo básico. Por ejemplo, haga que el servidor también pueda recibir archivos u otros datos. O enviar un mensaje al cliente. En el servidor tienes una lista de clientes, de modo que cuando recibas algo con el cliente sabrás de dónde provienen.

**Resultado final:**




Lea Zócalo asíncrono en línea: https://riptutorial.com/es/csharp/topic/9638/zocalo-asincrono

# Creditos

| S. No | Capítulos | Contributors |
|---|---|---|
| 1 | Empezando con C # Language | 4444, A. Raza, A_Arnold , aalaap, Aaron Hudon, abishekshivan, Ade Stringer, Aleksandur Murfitt, Almir Vuk, Alok Singh, Andrii Abramov, AndroidMechanic, Aravind Suresh, Artemix, Ben Aaronson, Bernard Vander Beken, Bjørn-Roger Kringsjå, Blachshma, Blorgbeard, bpoiss, Br0k3nL1m1ts, Callum Watkins, Carlos Muñoz, Chad Levy, Chris Nantau, Christopher Ronning, Community, Confiqure, crunchy, David G., David Pine, DavidG, DAXaholic , Delphi.Boy, Durgpal Singh, DWright, Ehsan Sajjad, Elie Saad, Emre Bolat, enrico.bacis, fabriciorissetto, FadedAce, Florian Greinacher, Florian Koch , Frankenstine Joe, Gennady Trubach, GingerHead, Gordon Bell, gracacs, G-Wiz, H. Pauwelyn, Happypig375, Henrik H, HodofHod, Hywel Rees, iliketocode, Iordanis , Jamie Rees, Jawa, jnovo, John Slegers, Kayathiri, ken2k , Kevin Montrose, Kritner , Krzyserious, leumas1960, M Monis |

| | | |
|---|---|---|
| | | Ahmed Khan, Mahmoud Elgindy, Malick, Marcus Höglund, Mateen Ulhaq, Matt, Matt, Matt, Matt, Michael B, Michael Brandon Morris, Miljen Mikic, Millan Sanchez, Nate Barbettini, Nick, Nick Cox, Nipun Tripathi, NotMyself, Ojen, PashaPash, pijemcolu, Prateek, Raj Rao, Rajput , Rakitić, Rion Williams, RokumDev, RomCoo, Ryan Hilbert, sebingel, SeeuD1, solidcell, Steven Ackley, sumit sharma, Tofix, Tom Bowers, Travis J, Tushar patel, User 00000, user3185569, Ven, Victor Tomaili, viggity, void, Wen Qin, Ziad Akiki , Zze |
| 2 | Acceso a la carpeta compartida de la red con nombre de usuario y contraseña. | Mohsin khan |
| 3 | Acceso a las bases de datos | ATechieThought, ravindra, Rion Williams, The_Outsider, user2321864 |
| 4 | Administración del sistema.Automation | Mikko Viitala |
| 5 | Alias de tipos incorporados | Racil Hilan, Rahul Nikate , Stephen Leppik |
| 6 | Almacenamiento en caché | Aliaksei Futryn, th1rdey3 |
| 7 | Anotación de datos | Maxime, Mikko Viitala, The_Outsider, Will Ray |
| 8 | Arboles de expresion | Benjamin Hodgson, dasblinkenlight, Dileep, George Duckett, just.another.programmer , Matas Vaitkevicius, |

| | | |
|---|---|---|
| | | matteeyah, meJustAndrew, Nathan Tuggy, NikolayKondratyev, Rob, Ruben Steins, Stephen Leppik, Рахул Маквана |
| 9 | Archivo y Stream I / O | BanksySan, Blachshma, dbmuller, DJCubed, Feelbad Soussi Wolfgun DZ, intox, Mikko Viitala, Sender, Squidward, Tolga Evcimen, Wasabi Fan |
| 10 | Argumentos con nombre | Cihan Yakar, Danny Chen, mehrandvd, Pan, Pavel Mayorov, Stephen Leppik |
| 11 | Argumentos nombrados y opcionales | RamenChef, Sibeesh Venu, Testing123, The_Outsider, Tim Yusupov |
| 12 | Arrays | A_Arnold, Aaron Hudon, Alexey Groshev, Anas Tasadduq, Andrii Abramov, Baddie, Benjamin Hodgson, bluray, coyote, D.J., das_keyboard, Fernando Matsumoto, granmirupa, Jaydip Jadhav, Jeppe Stig Nielsen, Jon Schneider, Ogoun, RamenChef, Robert Columbia, Shyju, The_Outsider, Thomas Weller, tonirush, Tormod Haugene, Wasabi Fan, Wen Qin, Xiobiq, Yotam Salmon |
| 13 | Asíncrono-espera | Aaron Hudon, AGB, aholmes, Ant P, Benjol, BrunoLM, Conrad.Dean, Craig Brett, Donald |

| | | |
|---|---|---|
| | | Webb, EJoshuaS, EvilTak, gdyrrahitis, George Duckett, Grimm The Opiner, Guanxi, guntbert, H. Pauwelyn, jdpilgrim, ken2k, Kevin Montrose, marshal craft, Michael Richardson, Moerwald, Nate Barbettini, nickguletskii, Pavel Mayorov, Pavel Voronin, pinkfloydx33, Rob, Serg Rogovtsev, Stefano d'Antonio, Stephen Leppik, SynerCoder, trashr0x, Tseng, user2321864, Vincent |
| 14 | Async / await, Backgroundworker, tareas y ejemplos de subprocesos | Dieter Meemken, Kyrylo M, nik, Pavel Mayorov, sebingel, Underscore, Xander Luciano, Yehor Hromadskyi |
| 15 | Atributos | Alexander Mandt, Andrew Diamond, Doruk, LosManos, Lukas Kolletzki, NikolayKondratyev, Pavel Sapehin, SysVoid, TKharaishvili |
| 16 | Biblioteca paralela de tareas | Benjamin Hodgson, Brandon, Collin Stevens, i3arnon, Mokhtar Ashour, Murtuza Vohra |
| 17 | BigInteger | 4444, Ed Marty, James Hughes, Rob, The_Outsider |
| 18 | Bucle | Alisson, Andrei Rînea, B Hawkins, Benjamin Hodgson, Botond Balázs, connor, Dialecticus, DJCubed, Freelex, Jon Schneider, Oluwafemi, |

| | | |
|---|---|---|
| | | Racil Hilan, Squidward, Testing123, Tolga Evcimen |
| 19 | C # Script | mehrandvd, Squidward, Stephen Leppik |
| 20 | Cadena.Formato | Aaron Hudon, Akshay Anand, Alexander Mandt , Andrius, Aseem Gautam, Benjol, BrunoLM, Dmitry Egorov , Don Vince, Dweeberly, ebattulga, ejhn5, gdoron, H. Pauwelyn, Hossein Narimani Rad, Jasmin Solanki, Marek Musielak, Mark Shevchenko, Matas Vaitkevicius, Mendhak, MGB, nikchi, Philip C, Rahul Nikate, Raidri, RamenChef, Richard, Richard, Rion Williams, ryanyuyu, teo van kot, Vincent, void, Wyck |
| 21 | Características de C # 3.0 | 0xFF, bob0the0mighty, FrenkyB, H. Pauwelyn, ken2k, Maniero, Rob |
| 22 | Características de C # 4.0 | Benjamin Hodgson, Botond Balázs, H. Pauwelyn, Proxima, Sibeesh Venu, Squidward, Theodoros Chatzigiannakis |
| 23 | Características de C # 5.0 | Abob, alex.b, H. Pauwelyn |
| 24 | Características de C # 6.0 | A_Arnold, Aaron Anodide, Aaron Hudon, Adil Mammadov, Adriano Repetti, AER, AGB, Akshay Anand, Alan McBee, Alex Logan, Amitay Stern, |

anaximander, andre_ss6, Andrea, AndroidMechanic, Ares, Arthur Rizzo, Ashwin Ramaswami, avishayp, Balagurunathan Marimuthu, Bardia, Ben Aaronson, Blubberguy22, Bobson, bpoiss, Bradley Uffner, Bret Copeland, C4u, Callum Watkins, Chad Levy, Charlie H, ChrFin, Community, Conrad.Dean, Cyprien Autexier, Dan, Daniel Minnaar, Daniel Stradowski, DarkV1, dasblinkenlight, David, David G., David Pine, Deepak gupta, DLeh, dotctor, Durgpal Singh, Ehsan Sajjad, el2iot2, Emre Bolat, enrico.bacis, Erik Schierboom, fabriciorissetto, faso, Franck Dernoncourt, FrankerZ, Gabor Kecskemeti, Gary, Gates Wong, Geoff, GingerHead, Gordon Bell, Guillaume Pascal, H. Pauwelyn, hankide, Henrik H, iliketocode, Iordanis , Irfan, Ivan Yurchenko, J. Steen, Jacob Linney, Jamie Rees, Jason Sturges, Jeppe Stig Nielsen, Jim, JNYRanger, Joe, Joel Etherton, John Slegers, Johnbot, Jojodmo, Jonas S, Juan, Kapep, ken2k, Kit, Konamiman, Krikor Ailanjian, Lafexlos, LaoR, Lasse Vågsæther Karlsen, M.kazem

Akhgary, Mafii, Magisch, Makyen, MANISH KUMAR CHOUDHARY, Marc, MarcinJuraszek, Mark Shevchenko, Matas Vaitkevicius, Mateen Ulhaq, Matt, Matt, Matt, Matt Thomas, Maximillian Laumeister, mbrdev, Mellow, Michael Mairegger, Michael Richardson, Michał Perłakowski, mike z, Minhas Kamal, Mitch Talmadge, Mohammad Mirmostafa, Mr.Mindor, mshsayem, MuiBienCarlota, Nate Barbettini, Nicholas Sizer , nik, nollidge, Nuri Tasdemir, Oliver Mellet, Orlando William, Osama AbuSitta, Panda, Parth Patel, Patrick, Pavel Voronin, PSN, qJake, QoP, Racil Hilan, Radouane ROUFID, Rahul Nikate, Raidri, Rajeev, Rakitić, ravindra, rdans, Reeven, Richa Garg, Richard, Rion Williams, Rob, Robban, Robert Columbia, Ryan Hilbert, ryanyuyu, Sam, Sam Axe, Samuel, Sender, Shekhar, Shoe, Slayther, solidcell, Squidward, Squirrel, stackptr, stark, Stilgar, Sunny R Gupta, Suren Srapyan, Sworgkh, syb0rg, takrl, Tamir Vered, Theodoros Chatzigiannakis, Timothy Shields, Tom Droste, Travis J, Trent, Trikaldarshi, Troyen,

| | | |
|---|---|---|
| | | Tushar patel, tzachs, Uri Agassi, Uriil, uTeisT, vcsjones, Ven, viggity, Vishal Madhvani, Vlad, Wai Ha Lee, Xiaoy312, Yury Kerbitskov, Zano, Ze Rubeus, Zimm1 |
| 25 | Características de C # 7.0 | Adil Mammadov, afuna, Amitay Stern, Amr Badawy, Andreas Pähler , Andrew Diamond, Avi Turner, Benjamin Hodgson, Blorgbeard, bluray, Botond Balázs, Bovaz, Cerbrus, Clueless, Conrad.Dean, Dale Chen, David Pine, Degusto, Didgeridoo, Diligent Key Presser, ECC-Dan, Emre Bolat, fallaciousreasoning, ferday, Florian Greinacher, ganchito55, Ginkgo, H. Pauwelyn, Henrik H, Icy Defiance, Igor Ševo, iliketocode, Jatin Sanghvi, Jean-Bernard Pellerin, Jesse Williams, Jon Schoning, Kimmax, Kobi, Kris Vandermotten, Kritner, leppie, Llwyd, Maakep, maf-soft, Marc Gravell, MarcinJuraszek, Mariano Desanze, Matt Rowland, Matt Thomas, MemphiZ, mnoronha, MotKohn, Name, Nate Barbettini, Nico, Niek, nietras, NikolayKondratyev, Nuri Tasdemir, PashaPash, Pavel Mayorov, PeteGO, petrzjunior, Philippe, Pratik, Priyank Gadhiya, Pyritie, qJake, Raidri, Rakitić, RamenChef, |

| | | |
|---|---|---|
| | | Ray Vega, RBT, René Vogt, Rob, samuelesque , Squidward, Stavm, Stefano, Stefano d'Antonio, Stilgar, Tim Pohlmann, Uriil, user1304444, user2321864, user3185569, uTeisT, Uwe Keim, Vlad, Vlad, Wai Ha Lee, Wasabi Fan , WerWet, wezten, Wojciech Czerniak, Zze |
| 26 | Clase parcial y metodos | Ben Jenkinson, Jonas S, Rahul Nikate, Stephen Leppik, Taras, The_Outsider |
| 27 | Clases estáticas | MCronin, The_Outsider, Xiaoy312 |
| 28 | CLSCompliantAttribute | mybirthname, Rob |
| 29 | Código inseguro en .NET | Andrew Piliser, cbale, codekaizen, Danny Varod, Isac, Jaroslav Kadlec, MSE, Nisarg Shah, Rahul Nikate, Stephen Leppik, Uwe Keim, ZenLulz |
| 30 | Comentarios y regiones | Bad, Botond Balázs, Jonathan Zúñiga, MrDKOz, Ranjit Singh, Squidward |
| 31 | Comenzando: Json con C # | Neo Vijay, Rob, VitorCioletti |
| 32 | Cómo usar C # Structs para crear un tipo de unión (similar a los sindicatos C) | DLeh, Milton Hernandez, Squidward, usr |
| 33 | Compilación de tiempo de ejecución | Artificial Stupidity, Stephen Leppik, Tommy |
| 34 | Comprobado y desactivado | Botond Balázs, Rahul Nikate, Sam Johnson, ZenLulz |

| | | |
|---|---|---|
| 35 | Concatenación de cuerdas | Abdul Rehman Sayed, Callum Watkins, ChaoticTwist, Doruk, Dweeberly, Jon Schneider, Oluwafemi, Rob, RubberDuck, Testing123, The_Outsider |
| 36 | Construcciones de flujo de datos de la biblioteca paralela de tareas (TPL) | Droritos, Stephen Leppik |
| 37 | Constructores y finalizadores | Adam Sills, Adi Lester, Adriano Repetti, Andrei Rînea, Andrew Diamond, Arjan Einbu, Avia, BackDoorNoBaby, BanksySan, Ben Fogel, Benjamin Hodgson, Benjol, Bogdan Gavril, Bovaz, Carlos Muñoz, Dan Hulme, Daryl, DLeh, Dmitry Bychenko, drusellers, Ehsan Sajjad, Fernando Matsumoto, guntbert, hatchet, Ian, Jeremy Kato, Jon Skeet, Julien Roncaglia, kamilk, Konamiman, ltiveron, Michael Richardson, Neel, Oly, Pavel Mayorov, Pavel Sapehin, Pavel Voronin, Peter Hommel, pinkfloydx33, Robert Columbia, RomCoo, Roy Dictus, Sam, Saravanan Sachi, Seph, Sklivvz, The_Cthulhu_Kid, Tim Medora, usr, Verena Haunschmid, void, Wouter, ZenLulz |
| 38 | Consultas LINQ | Adam Clifford, Ade Stringer, Adi Lester, Adil Mammadov, Akshay Anand, Aleksey L., Alexey Koptyaev, AMW, |

anaximander, Andrew Piliser, Ankit Vijay, Aphelion, bbonch, Benjamin Hodgson, bmadtiger, BOBS, BrunoLM, Budi, bumbeishvili, callisto, cbale, Chad McGrath, Chris, Chris H., coyote, Daniel Argüelles, Daniel Corzo, darcyq, David, David G., David Pine, DavidG, die maus, Diligent Key Presser, Dmitry Bychenko, Dmitry Egorov, dotctor, Ehsan Sajjad, Erick, Erik Schierboom, EvenPrime, fabriciorissetto, faso, Finickyflame, Florin M, forsvarir, fubo, gbellmann, Gene, Gert Arnold, Gilad Green, H. Pauwelyn, Hari Prasad, hellyale, HimBromBeere, hWright, iliketocode, Ioannis Karadimas, Jagadisha B S, James Ellis-Jones, jao, jiaweizhang, Jodrell, Jon Bates, Jon G, Jon Schneider, Jonas S, karaken12, KevinM, Koopakiller, leppie, LINQ , Lohitha Palagiri, ltiveron, Mafii, Martin Zikmund, Matas Vaitkevicius, Mateen Ulhaq, Matt, Maxime, mburleigh, Meloviz, Mikko Viitala, Mohammad Dehghan, mok, Nate Barbettini, Neel, Neha Jain, Néstor Sánchez A., Nico, Noctis , Pavel Mayorov, Pavel Yermalovich, Paweł

| | | |
|---|---|---|
| | | Hemperek, Pedro, Phuc Nguyen, pinkfloydx33, przno, qJake, Racil Hilan , rdans, Rémi, Rion Williams, rjdevereux, RobPethi, Ryan Abbott, S. Rangeley, S.Akbari, S.L. Barth, Salvador Rubio Martinez, Sanjay Radadiya, Satish Yadav, sebingel, Sergio Domínguez, SilentCoder, Sivanantham Padikkasu, slawekwin, Sondre, Squidward, Stephen Leppik, Steve Trout, Tamir Vered, techspider, teo van kot, th1rdey3, Theodoros Chatzigiannakis, Tim Iles , Tim S. Van Haren, Tobbe, Tom, Travis J, tungns304, Tushar patel, user1304444, user3185569, Valentin, varocarbas, VictorB, Vitaliy Fedorchenko, vivek nuna, void, wali, wertzui, WMios, Xiaoy312, Yaakov Ellis, Zev Spitz |
| 39 | Contexto de sincronización en Async-Await | codeape, Mark Shevchenko, RamenChef |
| 40 | Contratos de código | MegaTron |
| 41 | Contratos de Código y Afirmaciones | Roy Dictus |
| 42 | Convenciones de nombres | Ben Aaronson, Callum Watkins, PMF, ZenLulz |
| 43 | Corriente | Danny Bogers, jlawcordova, Jon Schneider, Nuri Tasdemir, Pushpendra |

| | | |
|---|---|---|
| 44 | Creación de una aplicación de consola con un editor de texto sin formato y el compilador de C # (csc.exe) | delete me |
| 45 | Creando un cuadro de mensaje propio en la aplicación Windows Form | Mansel Davies, Vaibhav_Welcomes_You |
| 46 | Criptografía (System.Security.Cryptography) | glaubergft, MikeS159, Ogglas, Pete |
| 47 | Cronómetros | Adam, demonplus, dotctor, Gavin Greenwalt , Jeppe Stig Nielsen, Sondre |
| 48 | Cuerdas verbatim | Alan McBee, Amitay Stern, Andrew Diamond, Aphelion, Arjan Einbu, avb, Bryan Crosby, Charlie H, David G., devuxer, DLeh, Ehsan Sajjad, Freelex, goric, Jared Hooper, Jeremy Kato, Jonas S, Kevin Montrose, Kilazur, Mateen Ulhaq, Ricardo Amores, Rion Williams, Sam Johnson, Sophie Jackson-Lee, Squirrel, th1rdey3 |
| 49 | Declaración de bloqueo | Aaron Hudon, Alexey Groshev, Andrei Rînea, Benjamin Hodgson, Botond Balázs, Christopher Currens, Cihan Yakar, David Ben Knoble, Denis Elkhov, Diligent Key Presser, George Duckett, George Polevoy, Jargon, Jasmin Solanki, Jivan, Mark Shevchenko, Matas Vaitkevicius, Mikko Viitala, Nuri Tasdemir, Oluwafemi, Pavel Mayorov, Richard, Rob, Scott Hannen, Squidward, Vahid |

| | | |
|---|---|---|
| | | Farahmandian |
| 50 | Declaraciones condicionales | Alexander Mandt, Ameya Deshpande, EJoshuaS, H. Pauwelyn, Hayden, Kroltan, RamenChef, Sklivvz |
| 51 | Delegados | Aaron Hudon, Adam, Ben Aaronson, Benjamin Hodgson, Bradley Uffner , CalmBit, Cihan Yakar, CodeWarrior, EyasSH, Huseyin Durmus, Jasmin Solanki, Jeppe Stig Nielsen, Jon G, Jonas S, Matt, NikolayKondratyev, niksofteng, Rajput, Richa Garg, Sam Farajpour Ghamari, Shog9, Stu, Thulani Chivandikwa, trashr0x |
| 52 | Delegados funcionales | Theodoros Chatzigiannakis, Valentin |
| 53 | Diagnósticos | Jasmin Solanki, Luke Ryan, TylerH |
| 54 | Directiva de uso | Fernando Matsumoto, Jesse Williams, JohnLBevan, Kit, Michael Freidgeim, Nuri Tasdemir, RamenChef, Tot Zam |
| 55 | Directivas del pre procesador | Andrei, Gilad Naaman, Matas Vaitkevicius, qJake, RamenChef, theB, volvis |
| 56 | Documentación XML Comentarios | Alexander Mandt, James , jHilscher, Jon Schneider, Nathan Tuggy, teo van kot, tsjnsn |
| 57 | Ejemplos de AssemblyInfo.cs | Adi Lester, Ameya |

| | | |
|---|---|---|
| | | Deshpande, AndreyAkinshin, Boggin, Dodzi Dzakuma, dove, Joel Martinez, pinkfloydx33, Ralf Bönning, Theodoros Chatzigiannakis, Wasabi Fan |
| 58 | Encuadernación | Bovaz, Stephen Leppik, yumaikas |
| 59 | Enhebrado | Aaron Hudon, Alexander Petrov, Austin T French, captainjamie, Eldar Dordzhiev, H. Pauwelyn, ionmike, Jacob Linney, JohnLBevan, leondepdelaw, Mamta D, Matthijs Wessels, Mellow , RamenChef, Zoba |
| 60 | Enumerable | 4444, Avia, Benjamin Hodgson, Luke Ryan, Olivier De Meulder, The_Outsider |
| 61 | Enumerar | Aaron Hudon, Abdul Rehman Sayed, Adrian Iftode, aholmes, alex, Blachshma, Chris Oldwood, Diligent Key Presser, dlatikay, Dmitry Bychenko, dove, Ghost4Man, H. Pauwelyn, ja72, Jon Schneider, Kit, konkked, Kyle Trauberman, Martin Zikmund, Matthew Whited, Maxime, mbrdev , Michael Mairegger, MuiBienCarlota, NikolayKondratyev, Osama AbuSitta, PSGuy , recursive, Richa Garg, Richard, Rob, sdgfsdh, Sergii Lischuk, Squirrel, Stefano d'Antonio, |

| | | |
|---|---|---|
| | | Tanner Swett, TarkaDaal, Theodoros Chatzigiannakis, vesi, Wasabi Fan, Yanai |
| 62 | Equals y GetHashCode | Alexey, BanksySan, hatcyl, ja72, Jeppe Stig Nielsen, meJustAndrew, Rob, scher, Timitry, viggity |
| 63 | Estructuras | abto, Alexey Groshev, Benjamin Hodgson, Botz3000, David, Elad Lachmi, ganchito55, Jon Schneider, NikolayKondratyev |
| 64 | Eventos | Aaron Hudon, Adi Lester, Benjol, CheGuevarasBeret, dcastro, matteeyah, meJustAndrew, mhoward, nik, niksofteng, NotEnoughData, OliPro007, paulius_l, PSGuy, Reza Aghaei, Roy Dictus, Squidward, Steven, vbnet3d |
| 65 | Excepcion de referencia nula | 4444, Agramer, Ashutosh, krimog, Kyle Trauberman, Mathias Müller, Philip C, RamenChef, S.L. Barth, Shelby115, Squidward, vicky, Zikato |
| 66 | Expresiones lambda | Andrei Rînea, Benjamin Hodgson, Benjol, David L, David Pine, Federico Allocati, Feelbad Soussi Wolfgun DZ, Fernando Matsumoto, H. Pauwelyn, haim770, Matas Vaitkevicius, Matt Sherman, Michael Mairegger, Michael |

| | | |
|---|---|---|
| | | Richardson, NotEnoughData, Oly, RubberDuck, S.L. Barth, Sunny R Gupta, Tagc, Thriggle |
| 67 | Extensiones reactivas (Rx) | stefankmitph |
| 68 | FileSystemWatcher | Sondre |
| 69 | Filtros de acción | Lokesh_Ram |
| 70 | Función con múltiples valores de retorno. | Adam, Alexey Mitev, Durgpal Singh, Tolga Evcimen |
| 71 | Funciones hash | Adi Lester, Callum Watkins, EvenPrime, ganchito55, Igor, jHilscher, RamenChef, ZenLulz |
| 72 | Fundación de comunicación de Windows | NtFreX |
| 73 | Fundición | Benjamin Hodgson, MSE , RamenChef, StriplingWarrior |
| 74 | Generación de Código T4 | lloyd, Pavel Mayorov |
| 75 | Generador de consultas Lambda genérico | 4444, PedroSouki |
| 76 | Generando números aleatorios en C # | A. Can Aydemir, Adi Lester, Alexander Mandt , DLeh, J3soon, Rob |
| 77 | Genéricos | AGB, andre_ss6, Ben Aaronson, Benjamin Hodgson, Benjol, Bobson, Carsten, darth_phoenixx, dymanoid, Eamon Charles, Ehsan Sajjad, Gajendra, GregC, H. Pauwelyn, ja72, Jim, Kroltan, Matas Vaitkevicius, mehmetgil, meJustAndrew, Mord Zuber, Mujassir Nasir, |

| | | |
|---|---|---|
| | | Oly, Pavel Voronin, Richa Garg, Sam, Sebi, Sjoerd222888, Theodoros Chatzigiannakis, user3185569, VictorB, void, Wallace Zhang |
| 78 | Guid | Bearington, Botond Balázs, elibyy, Jonas S, Osama AbuSitta, Sherantha, TarkaDaal, The_Outsider, Tim Ebenezer, void |
| 79 | Haciendo un hilo variable seguro | Wyck |
| 80 | Herencia | Almir Vuk, andre_ss6, Andrew Diamond, Barathon, Ben Aaronson , Ben Fogel, Benjol, David L, deloreyk, Ehsan Sajjad, harriyott, ja72, Jon Ericson, Karthik, Konamiman, MarcE, Matas Vaitkevicius, Pete Uh, Rion Williams, Robert Columbia, Steven , Suren Srapyan, VirusParadox, Yehuda Shapira |
| 81 | ICloneable | ja72, Rob |
| 82 | Identidad ASP.NET | HappyCoding, Skullomania |
| 83 | ILGenerador | Aleks Andreev, thehennyy |
| 84 | Implementación Singleton | Aaron Hudon, Adam, Adi Lester, Andrei Rînea, cbale, Disk Crasher, Ehsan Sajjad, Krzysztof Branicki, lothlarias, Mark Shevchenko, Pavel Mayorov, Sklivvz, snickro, Squidward, |

| | | |
|---|---|---|
| | | Squirrel, Stephen Leppik , Victor Tomaili, Xandrmoro |
| 85 | Implementando el patrón de diseño de peso mosca | Jan Bońkowski |
| 86 | Implementando un patrón de diseño de decorador | Jan Bońkowski |
| 87 | Importar contactos de Google | 4444, Supraj v |
| 88 | Incluyendo recursos de fuentes | Bales, Facebamm |
| 89 | Incomparables | alex |
| 90 | Indexador | A_Arnold, Ehsan Sajjad, jHilscher |
| 91 | Inicializadores de colección | Aphelion, ASh, Bart Jolling, Chronocide, CodeCaster, CyberFox, DLeh, Jacob Linney, Jeromy Irvine, Jonas S, Matas Vaitkevicius, Rob, robert demartino, rudygt, Squidward, Tamir Vered, TarkaDaal, Thulani Chivandikwa, WMios |
| 92 | Inicializadores de objetos | Andrei, Kroltan, LeopardSkinPillBoxHat, Marco, Nick DeVore, Stephen Leppik |
| 93 | Inicializando propiedades | Blorgbeard, hatchet, jaycer, Michael Sorens, Parth Patel, Stephen Leppik |
| 94 | Inmutabilidad | Boggin, Jon Schneider, Oluwafemi, Tim Ebenezer |
| 95 | Interfaces | Avia, Botond Balázs, CyberFox, harriyott, hellyale, Jeremy Kato, MarcE, MSE, PMF, Preston, Sigh, Sometowngeek, Stagg, Steven, user2441511 |

| | | |
|---|---|---|
| 96 | Interfaz IDisponible | Aaron Hudon, Adam, BatteryBackupUnit, binki, Bogdan Gavril, Bryan Crosby, ChrisWue, Dmitry Bychenko, Ehsan Sajjad, H. Pauwelyn, Jarrod Dixon, Josh Peterson, Matas Vaitkevicius, Maxime, Nicholas Sizer, OliPro007, Pavel Mayorov, pinkfloydx33, pyrocumulus, RamenChef, Rob, Thennarasan, Will Ray |
| 97 | Interfaz INotifyPropertyChanged | mbrdev, Stephen Leppik, Vlad |
| 98 | Interfaz IQueryable | lucavgobbi, Michiel van Oosterhout, RamenChef, Rob |
| 99 | Interoperabilidad | Balen Danny, Benjamin Hodgson, Bovaz, Craig Brett, Dean Van Greunen, Gajendra, Jan Bońkowski, Kimmax, Marc Wittmann, Martin, Pavel Durov, René Vogt, RomCoo, Squidward |
| 100 | Interpolación de cuerdas | Arjan Einbu, ATechieThought, avs099 , bluray, Brendan L, Dave Zych, DLeh, Ehsan Sajjad, fabriciorissetto, Guilherme de Jesus Santos, H. Pauwelyn, Jon Skeet, Nate Barbettini, RamenChef, Rion Williams, Squidward, Stephen Leppik, Tushar patel, Wasabi Fan |
| 101 | Inyección de dependencia | Buh Buh, iaminvinicble, Kyle Trauberman, Wiktor |

| | | |
|---|---|---|
| | | Dębski |
| 102 | Iteradores | Botond Balázs, Lijo, Nate Barbettini, Tagc |
| 103 | Leer y entender Stacktraces | S.L. Barth |
| 104 | Leyendo y escribiendo archivos .zip | 4444, DLeh, Naveen Gogineni, Nisarg Shah |
| 105 | Linq a los objetos | brijber, Christian Gollhardt, FortyTwo, Kevin Green, Raphael Pantaleão, Simon Halsey, Tanveer Badar |
| 106 | LINQ paralelo (PLINQ) | Adi Lester |
| 107 | LINQ to XML | Denis Elkhov, Stephen Leppik, Uali |
| 108 | Literales | jaycer, NotEnoughData, Racil Hilan |
| 109 | Los operadores | Adam Houldsworth, Adi Lester, Adil Mammadov, Akshay Anand, Alan McBee, Avi Turner, Ben Fogel, Blorgbeard, Blubberguy22, Chris Jester-Young, David Basarab, DLeh, Dmitry Bychenko, dotctor, Ehsan Sajjad, fabriciorissetto, Fernando Matsumoto, H. Pauwelyn, Henrik H, Jake Farley, Jasmin Solanki, Jephron, Jeppe Stig Nielsen, Jesse Williams, Joe, JohnLBevan, Jon Schneider, Jonas S, Kevin Montrose, Kimmax , lokusking, Matas Vaitkevicius, meJustAndrew, Mikko Viitala, mmushtaq, |

| | | |
|---|---|---|
| | | Mohamed Belal, Nate Barbettini, Nico, Oly, pascalhein, Pavel Voronin, petelids, Philip C, Racil Hilan, RhysO, Robert Columbia, Rodolfo Fadino Junior, Sachin Joseph, Sam, slawekwin, slinzerthegod , Squidward, Testing123, TyCobb, Wasabi Fan, Xiaoy312, Zaheer Ul Hassan |
| 110 | Manejador de autenticación C # | Abbas Galiyakotwala |
| 111 | Manejo de excepciones | 0x49D1, Abdul Rehman Sayed, Adam Lear, Adil Mammadov, Andrew Diamond, Aseem Gautam, Athafoud, Botond Balázs, Collin Stevens, Danny Chen, Dmitry Bychenko, dove, Eldar Dordzhiev, fabriciorissetto, faso, flq, George Duckett, Gilad Naaman, Gudradain, Jack, James Hughes, Jamie Rees, John Meyer , Jonesopolis, MadddinTribleD, Marimba, Matas Vaitkevicius, Matt, matteeyah, Mendhak, Michael Bisbjerg, Nate Barbettini, Nathaniel Ford, nik0lias, niksofteng , Oly, Pavel Pája Halbich , Pavel Voronin, PMF, Racil Hilan, raidensan, Rasa , Robert Columbia, RomCoo, Sam Hanley, Scott Koland, Squidward , Steve Dunn, Thulani Chivandikwa, vesi |

| | | |
|---|---|---|
| 112 | Manejo de FormatException al convertir cadenas a otros tipos | Rakitić, un-lucky |
| 113 | Manipulación de cuerdas | Blachshma, Jon Schneider, sferencik, The_Outsider |
| 114 | Métodos | Botz3000, F_V, fubo, H. Pauwelyn, Icy Defiance, Jasmin Solanki, Jeremy Kato, Jon Schneider, ken2k, Marco, meJustAndrew, MSL, S.Dav, Sjoerd222888, TarkaDaal, un-lucky |
| 115 | Métodos de extensión | Aaron Hudon, AbdulRahman Ansari, Adi Lester, Adil Mammadov, AGB, AldoRomo88, anaximander, Aphelion, Ashwin Ramaswami, ATechieThought, Ben Aaronson, Benjol, binki, Bjørn-Roger Kringsjå, Blachshma, Blorgbeard, Brett Veenstra, brijber, Callum Watkins, Chad McGrath, Charlie H, Chris Akridge, Chronocide, CorrectorBot, cubrr, Dan-Cook, Daniel Stradowski, David G., David Pine, Deepak gupta, diiN__________, DLeh, Dmitry Bychenko, DoNot, DWright, Đan, Ehsan Sajjad, ekolis, el2iot2, Elton, enrico.bacis, Erik Schierboom, ethorn10, extremeboredom, Ezra, fahadash, Federico Allocati, Fernando Matsumoto, FrankerZ, |

| | | |
|---|---|---|
| | | gdziadkiewicz, Gilad Naaman, GregC, Gudradain, H. Pauwelyn, HimBromBeere, Hsu Wei Cheng, Icy Defiance, Jamie Rees, Jeppe Stig Nielsen, John Peters, John Slegers, Jon Erickson, Jonas S, Jonesopolis, Kev, Kevin Avignon, Kevin DiTraglia , Kobi, Konamiman, krillgar, Kurtis Beavers, Kyle Trauberman, Lafexlos, LMK, lothlarias, Lukáš Lánský, Magisch, Marc, MarcE, Marek Musielak, Martin Zikmund, Matas Vaitkevicius, Matt, Matt Dillard, Maximilian Ast, mbrdev, MDTech.us_MAN, meJustAndrew, Michael Benford, Michael Freidgeim, Michael Richardson, Michał Perłakowski, Nate Barbettini, Nick Larsen, Nico, Nisarg Shah, Nuri Tasdemir, Parth Patel, pinkfloydx33, PMF, Prashanth Benny, QoP, Raidri, Reddy, Reeven, Ricardo Amores, Richard , Rion Williams, Rob, Robert Columbia, Ryan Hilbert, ryanyuyu, S. Tarık Çetin, Sam Axe, Shoe , Sibeesh Venu, solidcell , Sondre, Squidward, Steven, styfle, SysVoid, Tanner Swett, Timothy Rascher, TKharaishvili, T-moty, Tobbe, Tushar patel, unarist, user3185569, user40521 |

| | | |
|---|---|---|
| | | , Ven, Victor Tomaili, viggity |
| 116 | Métodos de fecha y hora | AbdulRahman Ansari, C4u, Christian Gollhardt, Felipe Oriani, Guilherme de Jesus Santos, James Hughes, Matas Vaitkevicius, midnightsyntax, Mostafiz , Oluwafemi, Pavel Yermalovich, Sondre, theinarasu, Thulani Chivandikwa |
| 117 | Microsoft.Exchange.WebServices | Bassie |
| 118 | Modificadores de acceso | Botond Balázs, H. Pauwelyn, hatcyl, John, Justin Rohr, Kobi, Robert Woods, Thaoden, ZenLulz |
| 119 | Nombre del operador | Chad, Danny Chen, heltonbiker, Kane, MotKohn, Philip C, pinkfloydx33, Racil Hilan, Rob, Robert Columbia, Sender, Sondre, Stephen Leppik, Wasabi Fan |
| 120 | O (n) Algoritmo para rotación circular de una matriz | AFT |
| 121 | ObservableCollection | demonplus, GeralexGR, Jonathan Anctil, MuiBienCarlota |
| 122 | Operaciones de cadena comunes | Austin T French, Blachshma, bluish, CharithJ, Chief Wiggum, cyberj0g, Daryl, deloreyk , jaycer, Jaydip Jadhav, Jon G, Jon Schneider, juergen d, Konamiman, Maniero, Paul Weiland, Racil Hilan, RoelF, Stefan Steiger, Steven, |

| | | |
|---|---|---|
| | | The_Outsider, tiedied61, un-lucky, WizardOfMenlo |
| 123 | Operador de Igualdad | Vadim Martynov |
| 124 | Operador de unión nula | aashishkoirala, Ankit Rana, Aristos, Bradley Uffner, David Arno, David G., David Pine, demonplus, Denis Elkhov, Diligent Key Presser, Eamon Charles, Ehsan Sajjad, eouw0o83hf, Fernando Matsumoto, H. Pauwelyn , Jodrell, Jon Schneider, Jonesopolis, Martin Zikmund, Mike C, Nate Barbettini, Nic Foster, petelids, Prateek, Rahul Nikate, Rion Williams, Rob, smead, tonirush, Wasabi Fan, Will Ray |
| 125 | Operadores de condicionamiento nulo | Alpha, dazerdude, DLeh, Draken, George Duckett, Jon Schneider, Kobi, Max, Nathan, Nicholas Sizer, Rob, Stephen Leppik, tehDorf, Timothy Shields, topolm, Wasabi Fan |
| 126 | Palabra clave de rendimiento | Aaron Hudon, Andrew Diamond, Ben Aaronson , ChrisPatrick, Damon Smithies, David G., David Pine, Dmitry Bychenko, dotctor, Ehsan Sajjad, erfanrazi, Gajendra, George Duckett, H. Pauwelyn, HimBromBeere, Jeremy Kato, João Lourenço, Joe Amenta, Julien Roncaglia, just.ru, Karthik AMR, Mark Shevchenko, Michael |

| | | |
|---|---|---|
| | | Richardson, MuiBienCarlota, Myster, Nate Barbettini, Noctis, Nuri Tasdemir, Olivier De Meulder, OP313, ravindra, Ricardo Amores, Rion Williams, rocky, Sompom, Tot Zam, un-lucky, Vlad, void, Wasabi Fan, Xiaoy312, ZenLulz |
| 127 | Palabras clave | 4444, A_Arnold, Aaron Hudon, Ade Stringer, Adi Lester, Aditya Korti, Adriano Repetti, AJ., Akshay Anand, Alex Filatov, Alexander Pacha , Amir Pourmand, Andrei Rînea, Andrew Diamond, Angela, Anna, Avia, Bart , Ben, Ben Fogel, Benjamin Hodgson, Bjørn-Roger Kringsjå, Botz3000, Brandon, brijber, BrunoLM, BunkerMentality, BurnsBA, bwegs, Callum Watkins, Chris, Chris Akridge, Chris H., Chris Skardon, ChrisPatrick, Chuu, Cihan Yakar, cl3m , Craig Brett, Daniel, Daniel J.G., Danny Chen , Darren Davies, Daryl, dasblinkenlight, David, David G., David L, David Pine, DAXaholic, deadManN, DeanoMachino, digitlworld, Dmitry Bychenko, dotctor, DPenner1, Drew Kennedy, DrewJordan, Ehsan Sajjad, EJoshuaS , Elad Lachmi, Eric Lippert, EvenPrime, F_V, |

| | | |
|---|---|---|
| | | Felix, fernacolo, Fernando Matsumoto, forsvarir, Francis Lord, Gavin Greenwalt, gdoron , George Duckett, Gilad Naaman, goric, greatwolf , H. Pauwelyn, Happypig375, Icemanind , Jack, Jacob Linney, Jake, James Hughes, Jcoffman, Jeppe Stig Nielsen, jHilscher, João Lourenço, John Slegers, JohnD, Jon Schneider, Jon Skeet, JoshuaBehrens, Kilazur, Kimmax, Kirk Woll, Kit, Kjartan, kjhf, Konamiman , Kyle Trauberman, kyurthich, levininja, lokusking, Mafii, Mamta D, Mango Wong, MarcE, MarcinJuraszek, Marco Scabbiolo, Martin, Martin Klinke, Martin Zikmund, Matas Vaitkevicius, Mateen Ulhaq, Matěj Pokorný, Mat's Mug, Matthew Whited, Max, Maximilian Ast, Medeni Baykal, Michael Mairegger, Michael Richardson, Michel Keijzers, Mihail Shishkov , mike z, Mr.Mindor, Myster, Nicholas Sizer, Nicholaus Lawson, Nick Cox, Nico, nik, niksofteng, NotEnoughData, numaroth, Nuri Tasdemir , pascalhein, Pavel Mayorov, Pavel Pája Halbich, Pavel Yermalovich, Pavieł Kraskoŭski, Paweł Mach, petelids, Peter Gordon, |

| | | |
|---|---|---|
| | | Peter L., PMF, Rakitić, RamenChef, ranieuwe, Razan, RBT, Renan Gemignani, Ringil, Rion Williams, Rob, Robert Columbia, ro binmckenzie, RobSiklos, Romain Vincent, RomCoo, ryanyuyu, Sain Pradeep, Sam, Sándor Mátyás Márton, Sanjay Radadiya, Scott, sebingel, Skipper, Sobieck, sohnryang, somebody, Sondre, Squidward, Stephen Leppik, Sujay Sarma, Suyash Kumar Singh, svick, TarkaDaal, th1rdey3, Thaoden, Theodoros Chatzigiannakis, Thorsten Dittmar, Tim Ebenezer, titol, tonirush, topolm, Tot Zam, user3185569, Valentin, vcsjones, void, Wasabi Fan, Wavum, Woodchipper, Xandrmoro, Zaheer Ul Hassan, Zalomon, Zohar Peled |
| 128 | Patrones de diseño creacional | DWright, Jan Bońkowski, Mark Shevchenko, Parth Patel, PedroSouki, Pierre Theate, Sondre, Tushar patel |
| 129 | Patrones de diseño estructural | Timon Post |
| 130 | Plataforma de compilación .NET (Roslyn) | 4444, Lukáš Lánský |
| 131 | Polimorfismo | Ade Stringer, ganchito55, H. Pauwelyn, Karthik, Maximilian Ast, void |
| 132 | Programación Funcional | Andrei Epure, Boggin, |

| | | |
|---|---|---|
| | | Botond Balázs, richard |
| 133 | Programación Orientada a Objetos En C # | Yashar Aliabasi |
| 134 | Propiedades | Botond Balázs, Callum Watkins, Jeremy Kato, John, JohnLBevan, niksofteng, Stephen Leppik, Zohar Peled |
| 135 | Punteros | Jeppe Stig Nielsen, Theodoros Chatzigiannakis |
| 136 | Punteros y código inseguro | Aaron Hudon, Botond Balázs, undefined |
| 137 | Realizando peticiones HTTP | Gordon Bell, Jon Schneider, Mark Shevchenko |
| 138 | Rebosar | Akshay Anand, Nuri Tasdemir, tonirush |
| 139 | Recolector de basura en .Net | Andrei Rînea, da_sann, Eamon Charles, J3soon, Luke Ryan, Squidward, Suren Srapyan |
| 140 | Recursion | Alexey Groshev, Botond Balázs, connor, ephtee, Florian Koch, Kroltan, Michael Brandon Morris, Mulder, Pan, qJake, Robert Columbia, Roy Dictus, SlaterCodes, Yves Schelpe |
| 141 | Redes | Adi Lester, Nicholaus Lawson, Salih Karagoz, shawty, Squirrel, Xander Luciano |
| 142 | Reflexión | Alexander Mandt, Aman Sharma, artemisart, Aseem Gautam, Axarydax, Benjamin Hodgson, Botond Balázs , Carson McManus, |

| | | |
|---|---|---|
| | | Cigano Morrison Mendez, Cihan Yakar, da_sann, DVJex, Ehsan Sajjad, H. Pauwelyn, Haim Bendanan, HimBromBeere, James Ellis-Jones, James Hughes, Jamie Rees, Jan Peldřimovský, Johny Skovdal, JSF, Kobi, Konamiman, Kristijan, Lovy, Matas Vaitkevicius, Mourndark, Nuri Tasdemir, pinkfloydx33, Rekshino, René Vogt, Sachin Chavan, Shuffler, Sjoerd222888, Sklivvz, Tamir Vered, Thriggle, Travis J, uygar.raf, Vadim Ovchinnikov, wablab, Wai Ha Lee |
| 143 | Regex Parsing | C4u |
| 144 | Resolución de sobrecarga | Dunno, Petr Hudeček, Stephen Leppik, TorbenJ |
| 145 | Secuencias de escape de cadena | Benjol, Botond Balázs, cubrr, Ed Gibbs, Jeppe Stig Nielsen, LegionMammal978, Michael Richardson, Peter Gordon, Petr Hudeček, Squidward, tonirush |
| 146 | Serialización binaria | David, Maxim, RamenChef, Stephen Leppik |
| 147 | StringBuilder | ATechieThought, brijber, Jeremy Kato, Jon Schneider, Robert Columbia, The_Outsider |
| 148 | System.DirectoryServices.Protocols.LdapConnection | Andrew Stollak |
| 149 | Temporizadores | Adam, Akshay Anand, |

| | | |
|---|---|---|
| | | Benjamin Kozuch, ephtee, RamenChef, Thennarasan |
| 150 | Tipo de conversión | Community, connor, Ehsan Sajjad, Lijo |
| 151 | Tipo de valor vs tipo de referencia | Abdul Rehman Sayed, Adam, Amir Pourmand, Blubberguy22, Chronocide, Craig Brett, docesam, GWigWam, matiaslauriti, meJustAndrew, Michael Mairegger, Michele Ceo, Moe Farag, Nate Barbettini, RamenChef, Rob, scher, Snympi, Tagc, Theodoros Chatzigiannakis |
| 152 | Tipo dinámico | Daryl, David, H. Pauwelyn, Kilazur, Mark Shevchenko, Nate Barbettini, Rob |
| 153 | Tipos anónimos | Fernando Matsumoto, goric, Stephen Leppik |
| 154 | Tipos anulables | Benjamin Hodgson, Braydie, DmitryG, Gordon Bell, Jasmin Solanki, Jon Schneider, Konstantin Vdovkin, Maximilian Ast, Mikko Viitala, Nicholas Sizer, Patrick Hofman, Pavel Mayorov, pinkfloydx33, Vitaliy Fedorchenko |
| 155 | Tipos incorporados | Alexander Mandt, David, F_V, Haseeb Asif, matteeyah, Patrick Hofman, Wai Ha Lee |
| 156 | Trabajador de fondo | Bovaz, Draken, ephtee, Jacobr365, Will |
| 157 | Tuplas | Bovaz, Chawin, EFrank, |

| | | |
|---|---|---|
| | | H. Pauwelyn, Mark Benovsky, Muhammad Albarmawi, Nathan Tuggy, Nikita, Nuri Tasdemir, petrzjunior, PMF, RaYell, slawekwin, Squidward, tire0011 |
| 158 | Una visión general de c # colecciones | Aaron Hudon, Andrew Diamond, Denuath, Jeremy Kato, Jon Schneider, Jorge, Juha Palomäki, Leon Husmann, Michael Mairegger, Michael Richardson, Nikita, rene, Rob, Sebi, TarkaDaal, wertzui, Will Ray |
| 159 | Usando json.net | Aleks Andreev, Snipzwolf |
| 160 | Usando SQLite en C # | Carmine, NikolayKondratyev, th1rdey3, Tim Yusupov |
| 161 | Utilizando la declaración | Adam Houldsworth, Ahmar, Akshay Anand, Alex Wiese, andre_ss6, Aphelion, Benjol, Boris Callens, Bradley Grainger, Bradley Uffner, bubbleking, Chris Marisic , ChrisWue, Cristian T, cubrr, Dan Ling, Danny Chen, dav_i, David Stockinger, dazerdude, Denis Elkhov, Dmitry Bychenko, Erik Schierboom, Florian Greinacher, gdoron, H. Pauwelyn, Herbstein, Jon Schneider, Jon Skeet, Jonesopolis, JT., Ken Keenan, Kev, Kobi, Kyle Trauberman, Lasse Vågsæther Karlsen, LegionMammal978, |

| | | |
|---|---|---|
| | | Lorentz Vedeler, Martin, Martin Zikmund, Maxime , Nuri Tasdemir, Peter K, Philip C, pid, René Vogt, Rion Williams, Ryan Abbott, Scott Koland, Sean, Sparrow, styfle, Sunny R Gupta, Sworgkh, Thaoden, The_Cthulhu_Kid, Tom Droste, Tot Zam, Zaheer Ul Hassan |
| 162 | XDocument y el espacio de nombres System.Xml.Linq | Crowcoder, Jon Schneider |
| 163 | XmlDocument y el espacio de nombres System.Xml | Alexander Petrov, Rokey Ge, Rubens Farias, Timon Post, Willy David Jr |
| 164 | Zócalo asíncrono | Timon Post |