

# Escritura, compilación y ejecución de una aplicación

En este capítulo, vamos a detallar el ciclo de vida de una aplicación desde la redacción del código hasta la ejecución de la aplicación, estudiando en detalle los mecanismos puestos en marcha.

## 1. Escritura del código

La inmensa mayoría de las aplicaciones se desarrollan gracias a un entorno integrado que agrupa las principales herramientas necesarias, a saber:

- un editor de texto;
- un compilador;
- un depurador.

Este enfoque es, de lejos, el más cómodo. Sin embargo necesita una pequeña fase de aprendizaje para familiarizarse con la herramienta. Para nuestra primera aplicación, vamos a utilizar una manera de hacer un poco diferente, ya que vamos a utilizar herramientas individuales: el bloc de notas de Windows para la escritura del código y el compilador en línea de comandos para Visual C#.

Nuestra primera aplicación será muy sencilla, ya que visualizará simplemente el mensaje «Hola» en una ventana de comando. A continuación se presenta el código de nuestra primera aplicación, que luego explicaremos línea por línea. Se debe introducir usando el bloc de notas de Windows o cualquier otro editor de texto siempre y cuando éste no añada ningún código de formato en el interior del documento, como sí hacen por ejemplo programas de tratamiento de texto.

### Ejemplo

```
using System;
class Program
{
    static String mensaje = "Hola";
    static void Main(String[] args)
    {
        Console.WriteLine(mensaje);
    }
}
```

Se debe guardar este código en un archivo con la extensión .cs. Esta extensión no es obligatoria, pero permite respetar las convenciones utilizadas por Visual Studio. Detallamos ahora algunas líneas de nuestra primera aplicación.

```
using System
```

Esta línea permite dejar directamente accesibles los elementos presentes en el namespace System. Sin ella, habría que utilizar los nombres plenamente cualificados para todos los elementos contenidos en el namespace. En nuestro caso, deberíamos utilizar entonces:  
`System.Console.WriteLine("Hola");`

```
class Program
```

En Visual C#, cualquier porción de código debe estar contenida en una clase.

```
static String mensaje= "Hola";
```

Esta línea declara una variable. Se debe declarar todas las variables antes de poder utilizarlas. La

declaración permite especificar el tipo de información que la variable va a contener: aquí, una cadena de caracteres y eventualmente un valor inicial, «hola» en nuestro caso.

```
static void Main (String[]args)
```

Todas las instrucciones, aparte de las declaraciones, deben estar ubicadas en un procedimiento o una función. La mayor parte del código se sitúa entonces entre los caracteres { y } , delimitando cada procedimiento o función. Entre todos los procedimientos y funciones, se designa a uno de ellos como el punto de entrada en la aplicación. A través de la ejecución de este procedimiento arranca la aplicación. Este procedimiento se debe llamar Main y debe ser estático. Se debe declarar en el interior de una clase o estructura. El tipo de retorno puede ser void o int. Los parámetros son optativos y, si se utilizan, representan los argumentos pasados en la línea de comando.

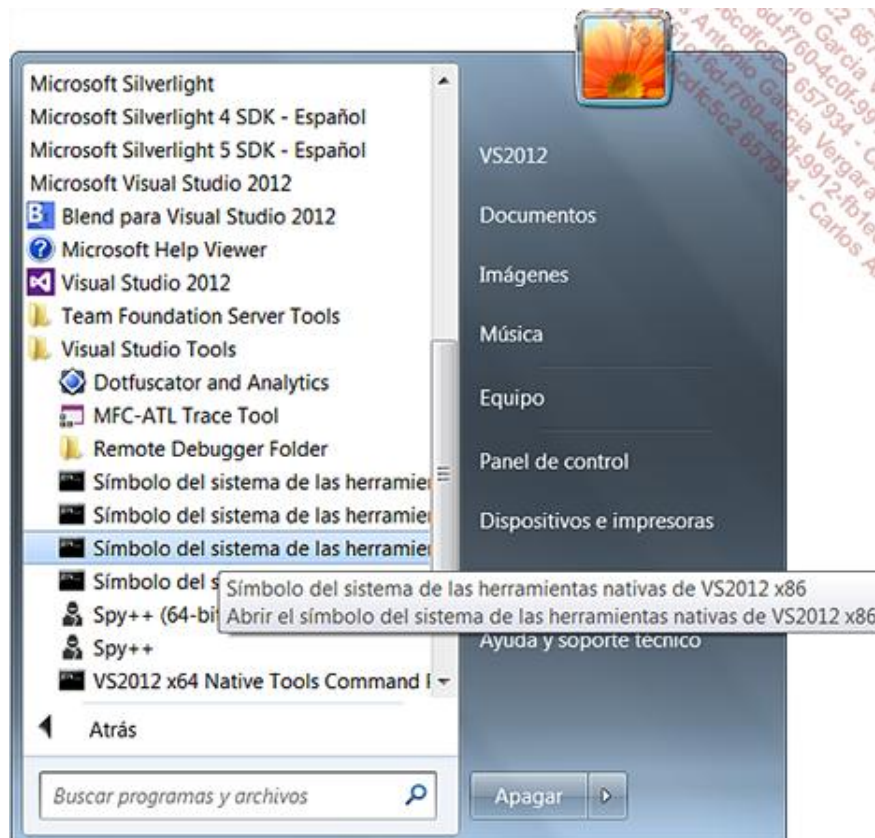
```
Console.WriteLine("Hola");
```

La clase Console definida en el espacio de nombres System provee un conjunto de métodos que permite la visualización de datos en la consola o la lectura de datos desde la consola. El procedimiento WriteLine permite la visualización de una cadena de caracteres en la consola.

Cabe destacar también que Visual C# distingue entre las minúsculas y las mayúsculas en las intrucciones. Si usted utiliza el editor de Visual Studio para redactar su código, éste le guiará para evitar errores (IntelliSense).

## 2. Compilación del código

El Framework .NET incluye un compilador en línea de comando para Visual C#. Para compilar el código fuente de nuestro ejemplo, debemos abrir una ventana de comando DOS para poder lanzar el compilador. Para ello la instalación creó un atajo en el menú Inicio. Este atajo lanza la ejecución de un archivo .bat que posiciona algunas variables de entorno necesarias para el correcto funcionamiento de las herramientas Visual Studio en línea de comando.



Desde la ventana de comandos abierta, conviene situarse en el directorio en el cual se encuentra el archivo

fuelle. Se lanza la compilación con el comando `csc Hola.cs`.

Después de un breve instante, el compilador nos devuelve el control. Podemos comprobar la presencia del archivo ejecutable y comprobar su correcto funcionamiento.



```
Administrador: Símbolo del sistema de las herramientas nativas de VS2012 x86

C:\Users\US2012\Documents\Visual Studio 2012\Projects>csc Hola.cs
Compilador de Microsoft (R) Visual C#, versión 4.0.30319.17929
para Microsoft (R) .NET Framework 4.5
(C) Microsoft Corporation. Reservados todos los derechos.

C:\Users\US2012\Documents\Visual Studio 2012\Projects>Hola
Hola

C:\Users\US2012\Documents\Visual Studio 2012\Projects>
```

Nuestra primera aplicación es realmente muy sencilla. Para aplicaciones más complejas, será útil a veces especificar algunas opciones para el funcionamiento del compilador. El conjunto de las opciones disponibles se puede obtener con el comando `csc / ?`.

Las principales opciones son:

`/out:archivo.exe`

Esta opción permite especificar el nombre del archivo resultado de la compilación. Por defecto, es el nombre del archivo fuente en curso de compilación que se utiliza.

`/target:exe`

Esta opción pide al compilador la generación de un archivo ejecutable para una aplicación en modo consola.

`/target:winexe`

Esta opción pide al compilador la generación de un archivo ejecutable de aplicación de Windows.

`/target:library`

Esta opción pide al compilador la generación de un archivo librería dll.

`/referencia:lista de archivos`

Esta opción indica al compilador la lista de los archivos referenciados en el código y necesarios para la compilación. Los nombres de los archivos se deben separar con una coma.

### 3. Análisis de un archivo compilado

Ahora que se ha creado nuestro archivo ejecutable, intentemos ver lo que contiene.

**Primera solución: abrirlo con el bloc de notas de Windows**

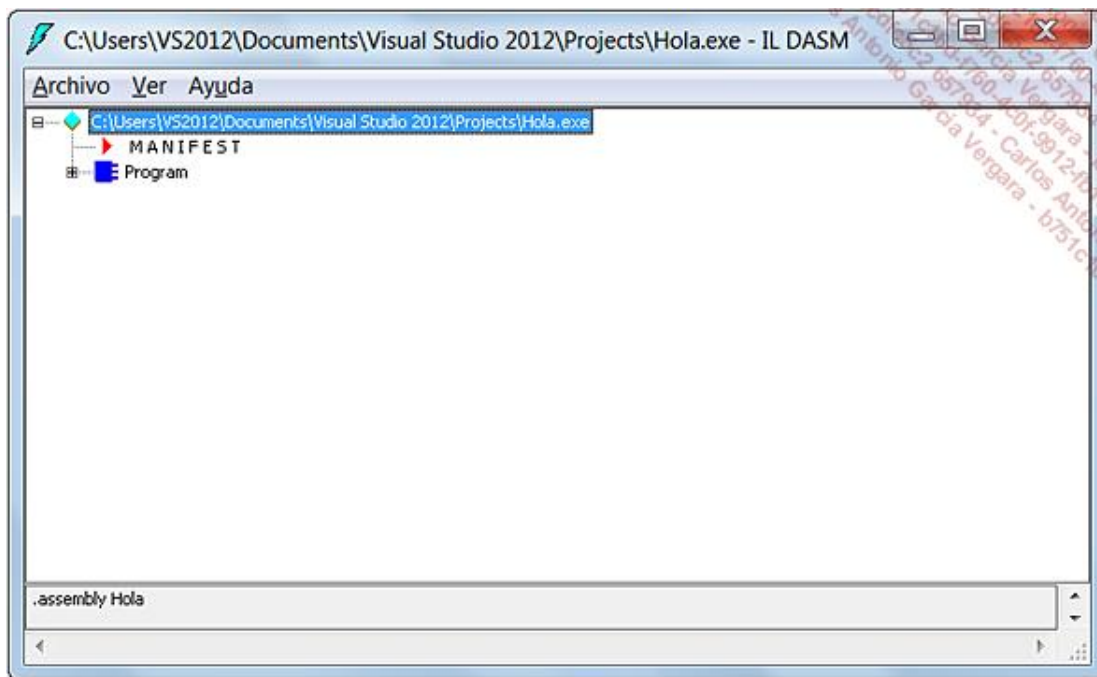
El resultado no es muy elocuente, ies lo menos que puede decirse!

Hemos dicho que el compilador genera código MSIL. Por lo tanto es este código lo que visualizamos en el bloc de notas. Para visualizar el contenido de un archivo MSIL, el Framework .NET propone una herramienta mejor adaptada.

### Segunda solución: utilizar un desensamblador

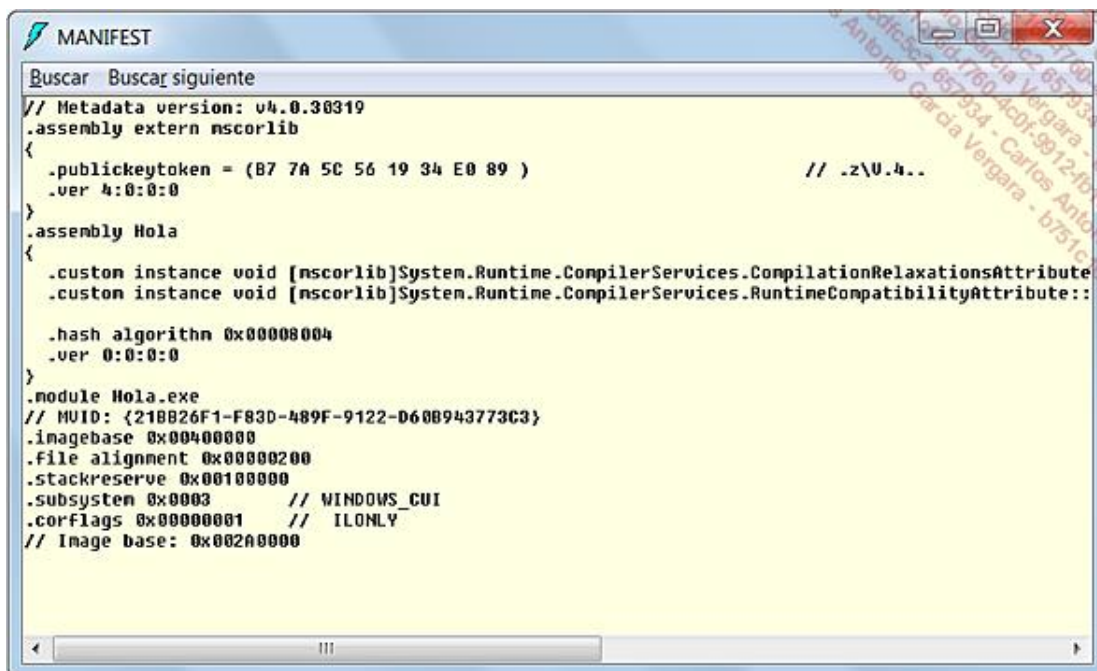
Esta herramienta se ejecuta a partir de la línea de comando con la instrucción `ildasm`.

Permite visualizar un archivo generado por el compilador, más claramente que con el bloc de notas. Conviene indicar el archivo que se desea examinar por el menú **Archivo - Abrir**. El desensamblador visualiza entonces su contenido.



La información presente en el archivo se puede separar en dos categorías: el manifiesto y el código MSIL. El manifiesto contiene los metadatos que permiten describir el contenido del archivo y los recursos que necesita. Hablamos en este caso de archivo autodescriptivo. Esta técnica es muy interesante, ya que en cuanto el Common Language Runtime lee el archivo, dispone de toda la información necesaria para su ejecución.














Ya no es necesario utilizar una grabación en el registro de la máquina. Se puede visualizar el manifiesto con un doble clic en su nombre.



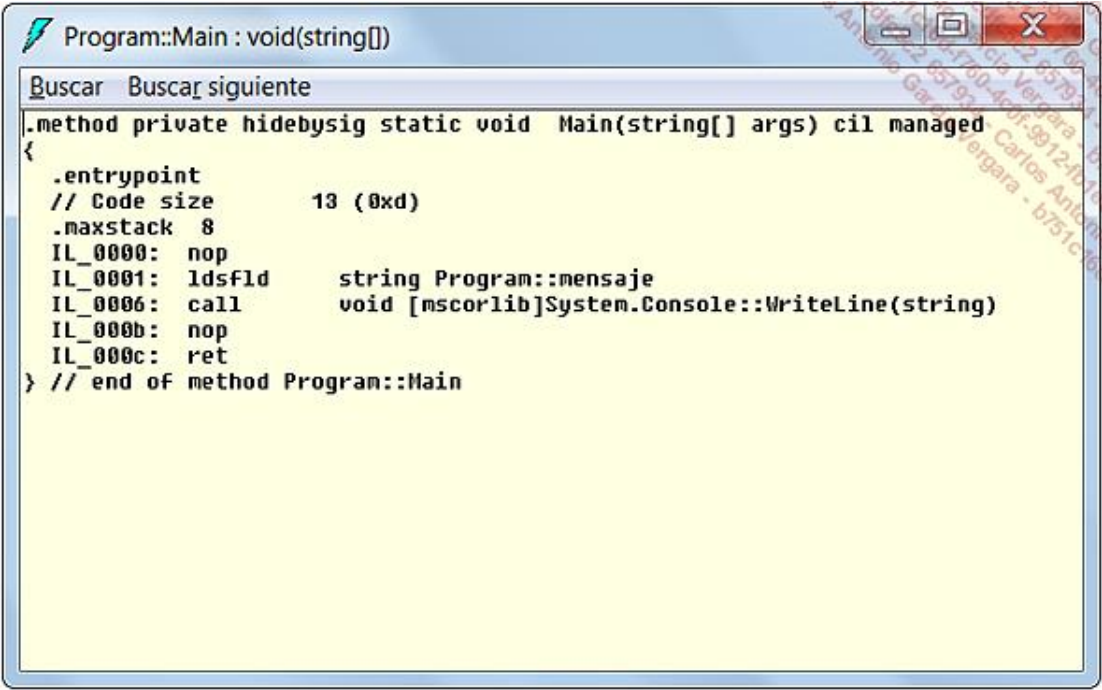
Encontramos en este manifiesto datos que indican que, para poder funcionar, la aplicación necesita el ensamblado externo mscorlib.

La segunda parte corresponde realmente al código MSIL. Un conjunto de iconos se utiliza para facilitar la visualización de los datos.



Símbolo	Significado
	Más información
	Espacio de nombres
	Clase
	Interfaz
	Clase de valores
	Enumeración
	Método
	Método estático
	Campo
	Campo estático
	Evento
	Propiedad
	Elemento de manifiesto o de información de clase

Como en el caso del manifiesto, un doble clic en un elemento permite obtener más detalles. Así podemos, por ejemplo, visualizar la traducción de nuestro procedimiento Main.



```

Program::Main : void(string[])
{
    .entrypoint
    // Code size      13 (0xd)
    .maxstack 8
    IL_0000: nop
    IL_0001: ldsfld      string Program::mensaje
    IL_0006: call         void [mscorlib]System.Console::WriteLine(string)
    IL_000b: nop
    IL_000c: ret
} // end of method Program::Main

```

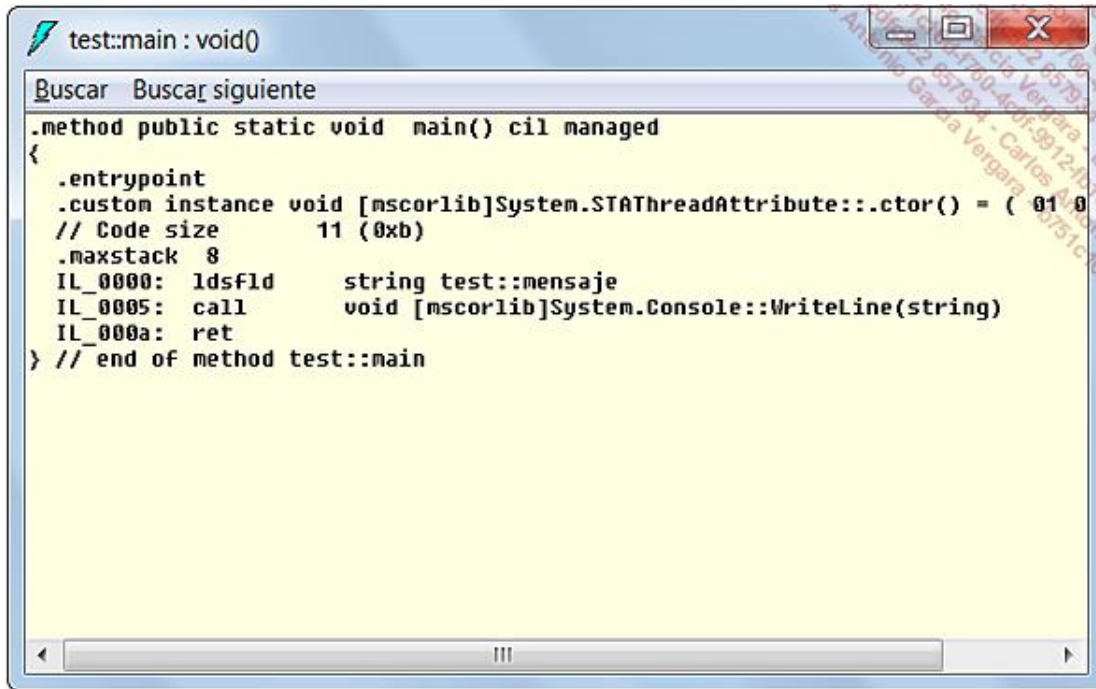
En un ejemplo de código tan sencillo, es fácil relacionar el código Visual C# y su traducción en código MSIL. Para las personas entusiasmadas por el código MSIL, existe un ensamblador MSIL: *ilasm*. Esta herramienta acepta como parámetro un archivo de texto que contiene código MSIL y lo transforma en formato binario.

Ya que somos capaces visualizar el código MSIL, podemos verificar que es realmente independiente del lenguaje

fuelle utilizado para desarrollar la aplicacii3n. A continuaci3n veamos el c3digo Visual Basic que realiza lo mismo que nuestro c3digo Visual C#.

```
using System
Imports System
Public Module test
    Dim mensaje As String = "Hola"
    Public Sub main()
        console.writeline(mensaje)
    End Sub
End Module
```

Tras la compilaci3n y desensblaje por ildasm, veamos lo que nos presenta para el m3todo Main.



No hay ninguna diferencia con respecto a la versi3n Visual C# del m3todo Main.

Tambi3n es posible dar los pasos inversos al transformar un archivo texto que contiene c3digo MSIL en archivo binario correspondiente. Esta transformaci3n se hace gracias al ensamblador ilasm. La 3nica dificultad consiste en crear un archivo texto que contiene el c3digo MSIL, ya que incluso si la sintaxis es comprensible, no es intuitiva. Una soluci3n puede consistir en pedir a la herramienta ildasm (el desensblador) que genere este archivo de texto. Para ello, despu3s de haber abierto el archivo ejecutable o la libreria DLL con ildasm, usted debe utilizar la opci3n **Volcar** del men3 **Archivo**. Se le invita entonces a elegir el nombre del archivo que hay que generar (extension .il).

Este archivo se puede modificar con un simple editor de texto. Sustituya, por ejemplo, el contenido de la variable mensaje con la cadena «Hello».

```
.method private hidebysig specialname rtspecialname static
    void .cctor() cil managed
{
    // Code size          11 (0xb)
    .maxstack 8
    IL_0000: ldstr      "Hello"
    IL_0005: stsfld      string Program::mensaje
    IL_000a: ret
} // end of method Program::.cctor
```

Guarde luego el archivo. Ahora sólo queda volver a generar el archivo ejecutable gracias al ensamblador ilasm. Para ello, introduzca la línea de comando siguiente:

```
ilasm Hola.il /output=Hello.exe
```

La opción `/output=Hello` permite indicar el nombre del archivo generado. Si no se especifica esta opción, se utilizará el nombre del archivo fuente. Usted puede ahora lanzar el nuevo ejecutable y verificar el mensaje visualizado. Todas estas operaciones se pueden hacer en cualquier archivo ejecutable o librería dll. La única dificultad reside en el volumen de información facilitado por la descompilación. Sin embargo, esto crea un problema: cualquier persona que dispone de los archivos ejecutables o librerías dll de una aplicación puede modificar la aplicación.

Por supuesto las modificaciones pueden resultar peligrosas, pero se puede considerar la modificación de un valor que representa una información importante para la aplicación (contraseña, clave de licencia...) Un remedio posible a este tipo de operación consiste en hacer lo más incomprensible posible el código generado por el descompilador. Para ello, hay que actuar a nivel del archivo ejecutable o de la librería dll con la modificación de los datos que contienen sin, por supuesto, perturbar el funcionamiento. Hay herramientas llamadas ofuscadores que son capaces de realizar esta operación. Visual Studio se suministra con una herramienta de la empresa PreEmptive Solutions llamada DotFuscator Community Edition. Esta versión permite realizar las operaciones básicas para «embrollar» un archivo. El principal tratamiento efectuado en el archivo consiste en renombrar los identificadores contenidos en él (nombre de las variables, nombre de los procedimientos y funciones...) con valores muy poco explícitos, en general a carácter único. Ahí tenemos un extracto de la descompilación del archivo `Hola.exe` tras su tratamiento por DotFuscator Community Edition.

```
.class public auto ansi sealed beforefieldinit DotfuscatorAttribute
    extends [mscorlib]System.Attribute
{
    .custom instance void [mscorlib]System.AttributeUsageAttribute::.ctor(value-
type [mscorlib]System.AttributeTargets) = ( 01 00 01 00 00 00 00 00 )
    .field private string a
    .method public hidebysig specialname rtspecialname
        instance void .ctor(string a) cil managed
    {
        // Code size          14 (0xe)
        .maxstack 2
        IL_0000: ldarg.0
        IL_0001: dup
        IL_0002: call        instance void [mscorlib]System.Attribute::.ctor()
        IL_0007: ldarg.1
        IL_0008: stfld        string DotfuscatorAttribute::a
        IL_000d: ret
    } // end of method DotfuscatorAttribute::.ctor
    .method public hidebysig string
        a() cil managed
    {
        // Code size          7 (0x7)
        .maxstack 1
        IL_0000: ldarg.0
        IL_0001: ldfld        string DotfuscatorAttribute::a
        IL_0006: ret
    } // end of method DotfuscatorAttribute::a
    .property instance string A()
    {
        .get instance string DotfuscatorAttribute::a()
    } // end of property DotfuscatorAttribute::A
} // end of class DotfuscatorAttribute

.class private auto ansi beforefieldinit a
    [mscorlib]System.Object
{
    .field private static string a
    .method private hidebysig static void a(string[] A_0) cil managed
    {
        .entrypoint
    }
```



```

// Code size      13 (0xd)
.maxstack 8
IL_0000: nop
IL_0001: ldsfld      string a::a
IL_0006: call        void [mscorlib]System.Console::WriteLine(string)
IL_000b: nop
IL_000c: ret
} // end of method a::a
.method public hidebysig specialname rtspecialname
    instance void .cil managed
{
    // Code size      7 (0x7)
    .maxstack 8
    IL_0000: ldarg.0
    IL_0001: call        instance void [mscorlib]System.Object::.ctor()
    IL_0006: ret
} // end of method a::.ctor
.method private hidebysig specialname rtspecialname static
    void .cctor() cil managed
{
    // Code size      (0xb)
    .maxstack 8
    IL_0000: ldstr      "Hola"
    IL_0005: stsfld      string a::a
    IL_000a: ret
} // end of method a::.cctor
} // end of class a

```

En este archivo, no queda rastro de los nombres utilizados en el código. La clase se llama a, el procedimiento Main se llama ahora «a», la variable mensaje se llama también ahora «a». ¡Imagínese el resultado de tal tratamiento en un archivo que contiene varias decenas de variables y procedimientos!

La versión Professional Edition permite también la encriptación de las cadenas de caracteres, la modificación y el añadido de código inútil para complicar las estructuras de controles (bucles, condiciones...).

A continuación presentamos un ejemplo de transformación de la documentación de Dotfuscator.

El código original:

```

public int CompareTo(Object o)
{
    int n = occurrences - ((WordOccurrence)o).occurrences;
    if (n == 0)
    {
        n = String.Compare(word, ((WordOccurrence)o).word);
    }
    return(n);
}

```

El código generado:

```

public virtual int _a(Object A_0) {
    int local0;
    int local1;
    local0 = this.a - (c) A_0.a;
    if (local0 != 0) goto i0;
    goto il;
    while (true) {
        return local1;
    i0: local1 = local0;
    }
    il: local0 = System.String.Compare(this.b, (c) A_0.b);
    goto i0;
}

```

¡El análisis de miles de líneas de código de este tipo puede provocar algunas migrañas! Por lo tanto, es preferible conservar el código original para las modificaciones posteriores. Dispone de más información en el sitio <http://www.preemptive.com/>

## 4. Ejecución del código

Cuando un usuario ejecuta una aplicación gestionada, el cargador de código del sistema operativo carga el Common Language Runtime que luego lanza la ejecución del código gestionado. Como el procesador de la máquina en la cual se ejecuta la aplicación no puede encargarse directamente del código MSIL, el Common Language Runtime debe convertirlo a código nativo.

Esta conversión no incluye la totalidad del código de la aplicación. Convierte el código según las necesidades. Los pasos adoptados son los siguientes:

- Al cargar una clase, el Common Language Runtime sustituye cada método de la clase con un trozo de código que requiere al compilador JIT que lo compile en lenguaje nativo.
- Luego, cuando se utiliza el método en el código, la porción de código generado en la carga entra en acción y compila el método en código nativo.
- El fragmento de código que requiere la compilación del método es sustituido luego por el código nativo generado.
- Las futuras llamadas de este método se harán directamente en el código nativo generado.