Introducción a la arquitectura .NET y al lenguaje C#

Por Manuel Lucas Viñas Livschitz
CiTEM
Universidad Ramón Llull

Términos básicos previos

Common Language Specification (CLS)

- Todos los lenguajes .NET han de cumplir con el CLS, es por tanto un conjunto de normas
- Características fundamentales
  - Ha de permitir el acceso a todas las librerías del .NET Framework.
  - Ha de permitir interoperatividad con los demás lenguajes .NET sean cuales sean.

Código Administrado (Managed code) y Código No Administrado (Unmanaged code)

- Código Administrado es código en un “entorno controlado”, es por tanto un código que no puede causar daños
- Código No Administrado es código en un entorno no controlado, es el entorno habitual cuando programamos en lenguajes como el C, C++ o Assembler

Entorno controlado

- Las principales características de este entorno controlado son:
  - El ejecutable no contiene código de Intel x86, sino código de MSIL con llamadas a la biblioteca de clases.
  - La administración de la memoria se realiza automáticamente mediante una recolección de basura.
  - Las asignaciones se hacen con el tipo validado. Por principio, el sistema de tipos es inviolable.
  - Las operaciones que pueden poner en peligro la seguridad, como abrir un archivo, requieren permisos especiales.
- Las cualidades anteriores permiten ejecutar programas de origen dudoso, por ejemplo de Internet, sin que estos programas puedan poner en peligro la seguridad ni la integridad del sistema.

Entorno no controlado. Verificabilidad.

- Todos los programas creados por el compilador de .NET se consideran "verificables". Esto quiere decir que el compilador JIT puede (tanto en tiempo de ejecución como en tiempo de compilación) verificar y garantizar que el programa no haga ninguna operación que pueda poner en peligro la seguridad ni la integridad del sistema.

Entorno no controlado. Verificabilidad.

- Puede parecer extraño, pero existen instrucciones de MSIL capaces de crear brechas en la seguridad del sistema, por ejemplo para el manejo directo de punteros o "casts" inseguros. Estas instrucciones son necesarias en algunos casos, como por ejemplo para que la propia biblioteca llame a la API de Windows. Los programas que contienen estas instrucciones se consideran "no verificables".
- EJ: OpenGL, Mesa3D, etc.

Entorno no controlado. Verificabilidad.

- Se pueden crear programas no verificables, incluso con manipulación directa de punteros. Evidentemente, es necesario un privilegio especial de seguridad para ejecutar los programas no verificables.
- Por lógica una aplicación normal debe acceder al API de Windows, por tanto podemos suponer que las librerías .NET que gestionan ventanas (entre otras) usan código no verificable y por eso son automáticamente registradas en el sistema como de uso seguro para todos los usuarios.
- Es perfectamente posible crear programas bastante útiles sin incumplir los criterios de "verificabilidad" y, por consiguiente, de seguridad y de hecho es deseable.

Common Language Runtime (CLR)

- Es el motor de ejecución universal del código de una aplicación .NET
- Es independiente y se incluye como parte en el .NET Framework

Common Language Runtime (CLR) Características

- Mejoras de rendimiento: Se busca el conjunto de instrucciones más apropiado para cada máquina (AMD, Intel, PDAs,…)
- Uso de componentes escritos en lenguajes diferentes
- Herencia de clases escritas en distintos lenguajes

Common Language Runtime (CLR) Características

- Administración de memoria mediante un GARBAGE COLLECTOR inteligente.
- Objetos que se auto describen, eliminando la necesidad de usar el sistema IDL para usar un objeto de otra aplicación.
- Soporte multithreading
- Diseño completamente orientado a objetos (no existen funciones sueltas o variables)

Common Language Runtime (CLR) Características

- Seguridad en el uso de tipos (se comprueban antes y durante la ejecución de los programas)
- Uso de delegates para asegurar que los pasos de funciones como argumentos a otras funciones se hace con corrección de tipos
- Permiso para usar código no administrado nativo. En C# unsafe.

Common Type System (CTS)

- El Common Type System es un completo sistema de tipos, integrado en el entorno Common Language Runtime, compatible con los tipos y las operaciones que se encuentran en la mayoría de los lenguajes de programación.
  Es compatible con la implementación completa de muchos lenguajes de programación.
- Ej. : int, short, byte, double, float, string (Unicode), char (Unicode), decimal, ...

El proceso de generación de una aplicación .NET

Pasos

Pasos

1. El programador escribe el código fuente del programa usando el lenguaje .NET que prefiera (C#, Visual Basic .NET, J#, o C++ administrado, JScript .NET).
2. El código fuente es compilado usando el compilador apropiado:
   1. C# usa el CSC.EXE
   2. Visual Basic .NET usa VBC.EXE
   3. J# usa un compilador en estado Beta que no se encuentra disponible en el .NET Framework 1.0
   4. C++ Administrado usa el compilador de C++ .NET incluido en el Visual Studio .NET
   5. JScript .NET usa JSC.EXE

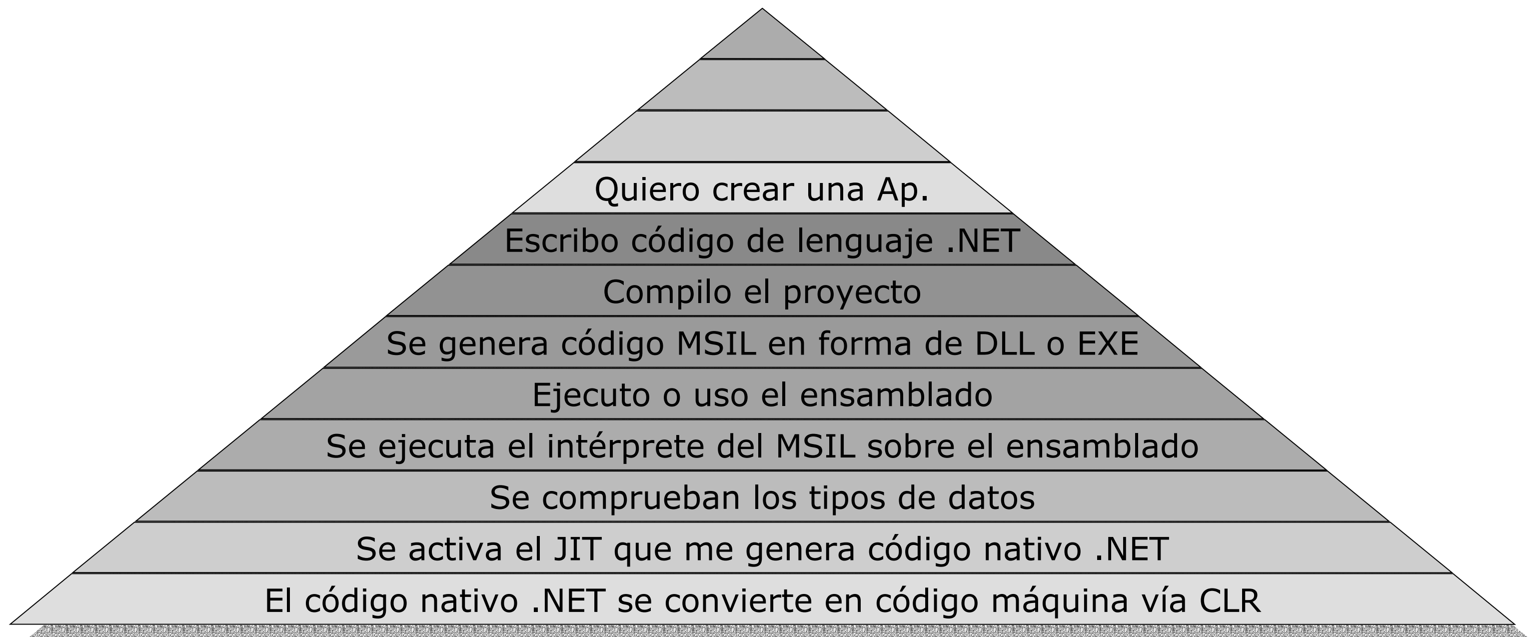
Pasos

3. El compilador convierte el código fuente en un lenguaje intermedio MSIL similar al bytecode de Java
   1. No siempre: código no administrado.
   2. El resultado se guarda bajo la forma de una DLL o un EXE denominados ensamblados y solo se puede usar / ejecutar en un sistema que disponga de la plataforma .NET

Pasos

4. El fichero compilado es “interpretado” por un intérprete de MSIL.
    1. Primero se comprueba el código para seguridad en los tipos de datos.
    2. Después se activa el Just In Time (JIT) que compila el MSIL en código nativo administrado y se combina con el Common Language Runtime (CLR) para producir el resultado final.
        1. JIT: Compilador en el momento. Se usa para convertir él código intermedio en código nativo por bloques para evitar la reinterpretación del código ya usado.
        2. CLR: Librerías base del lenguaje .NET dependiente de la máquina y optimizado para esta.

Proceso completo



El proceso de generación de una aplicación .NET

Términos en mayor detalle

Ensamblado

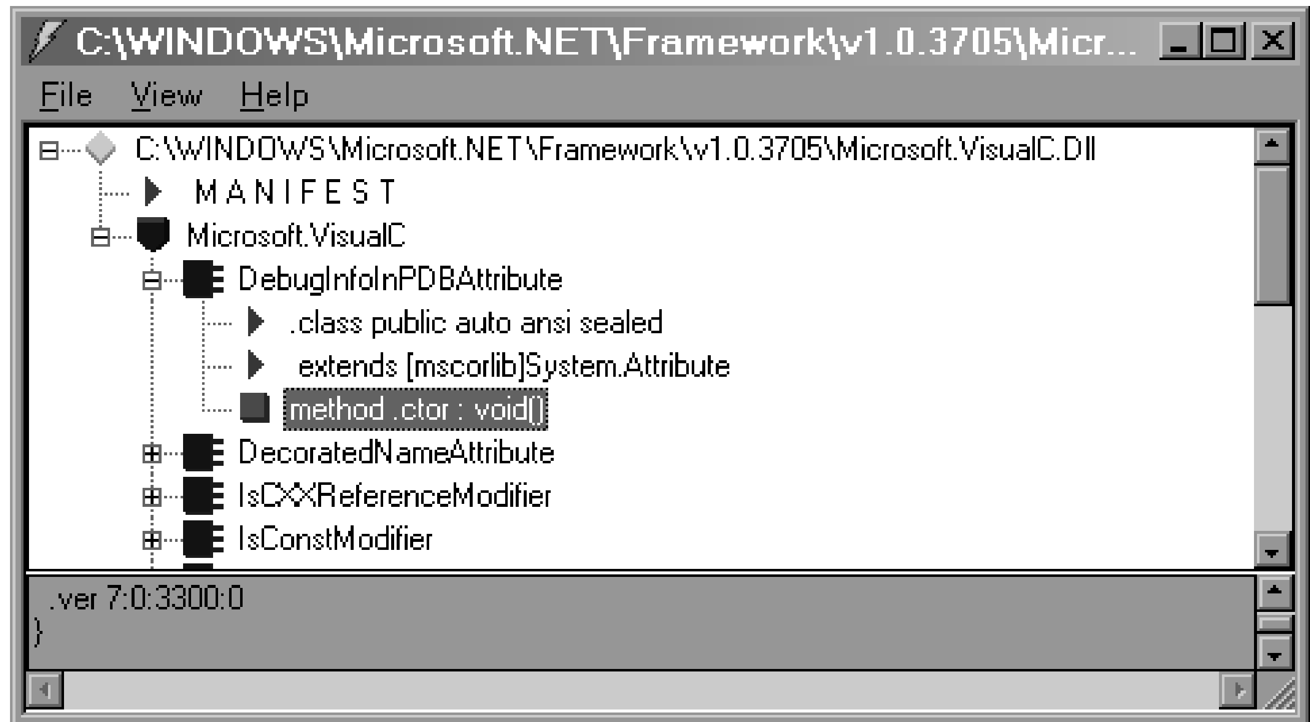
- Un ensamblado es un fichero especial con forma de DLL o EXE que contiene:
  - Información de versión detallada, tanto sobre el propio módulo como sobre los demás módulos a los que se hace referencia ("manifiesto")
  - Información detallada de todos los tipos implementados en el archivo ("meta datos")
  - Código de MSIL (Microsoft Intermediate Language). Este código se compila en el momento de la carga (o instalación) en la CPU de la computadora.
- El AL EXE permite integrar añadir el manifiesto a un módulo (ensamblado sin manifiesto) y convertirlo en una DLL o un EXE
- ***Un EXE es una DLL con un miembro estático de una clase marcado de una forma especial***

MSIL (Microsoft Intermediate Language)

- Ya que todo ensamblado tiene una secuencia de instrucciones MSIL esta se puede leer y rescribir.
- Además se incluyen las herramientas para hacerlo de manera gratuita:
  - ILDASM: Desensambla
  - ILASM: Ensambla
  - SDK: Nos dice como hacerlo y que como es el lenguaje IL al detalle para que hagamos los cambios.

MSIL

- Ejemplo del ILDASM interfaz:



MSIL

- Ejemplo de extracto de código (una función miembro):

```
.method public specialname rtspecialname
        instance void  .ctor() cil managed
{
  // Code size       13 (0xd)
  .maxstack  1
  IL_0000:  ldarg.0
  IL_0001:  call       instance void
   [mscorlib]System.Attribute::.ctor()
  IL_0006:  ldarg.0
  IL_0007:  call       instance void [mscorlib]System.Object::.ctor()
  IL_000c:  ret
} // end of method DecoratedNameAttribute::.ctor
```

MSIL

- ¿Cómo evitar el espionaje?

MSIL: Evitando el espionaje

- Usando código no administrado: Una función con código no administrado no se puede desensamblar. Ej.:

```
.class /*02000386*/ private auto ansi
  '<PrivateImplementationDetails>'
       extends [mscorlib/* 23000001
  */]System.Object/* 01000001 */
{
} // end of class '<PrivateImplementationDetails>'
```

MSIL : Evitando el espionaje

- Encriptando los ensamblados
  - Las librerías .NET tratan las DLL como si fuesen ficheros binarios, primero los carga en memoria como un array y luego los ejecuta, esto nos permite encriptar una DLL y desencriptarla en memoria tras haberla cargado.

MSIL : Evitando el espionaje

- Situar una parte de la funcionalidad de la aplicación en servicios de Web o incluso en servidores de aplicaciones (COM+/remoto). Como los usuarios no tienen acceso a los ejecutables, no tienen forma de descompilarlos. Este método está absolutamente garantizado.

MSIL : Evitando el espionaje

- Usando funciones escritas en DLLs de código no administrado escritas en C, pero existe la posibilidad que sean desensambladas.

MSIL : Evitando el espionaje

- Escribir parte del código en Assembler IL, utilizando recursos que dificultan la descompilación, por ejemplo identificadores no admitidos por los lenguajes C# o VB.NET.

MSIL : Evitando el espionaje

- Utilizar un "ofuscador" como Demeanor (http://www.wiseowl.com) o DotFuscator (http://www.preemptive.com/) para alterar el código de una forma que impida la descompilación.

JIT (Just In Time)

- Es un compilador bajo demanda en tiempo de ejecución. Puede parecer poco óptimo y puede llegar a serlo si trabajamos con ensamblados de grandes dimensiones.
- Podemos precompilarlos mediante GACUTIL y NGEN.

JIT: NGEN

- El ejecutable se sitúa en un lugar denominado "Global Assembly Cache“[C:\WINDOWS\assembly]
- Cuando un conjunto se compila con NGEN.EXE, se registran distintos datos:
  - Tipo de CPU
  - Versión del sistema operativo
  - Identificación del conjunto, una especie de "suma de comprobación + fecha y hora". **Las recompilaciones modifican esta identidad**.
  - Identificación exacta de todos los conjuntos a los que se hace referencia
  - Factores de seguridad
- Si alguno de ellos cambia cuando se ejecute la aplicación que necesita el módulo que previamente hemos puesto como NGEN, no será cargado del GAC sino del directorio de trabajo de la aplicación ignorando la precompilación.

JIT: GACUTIL

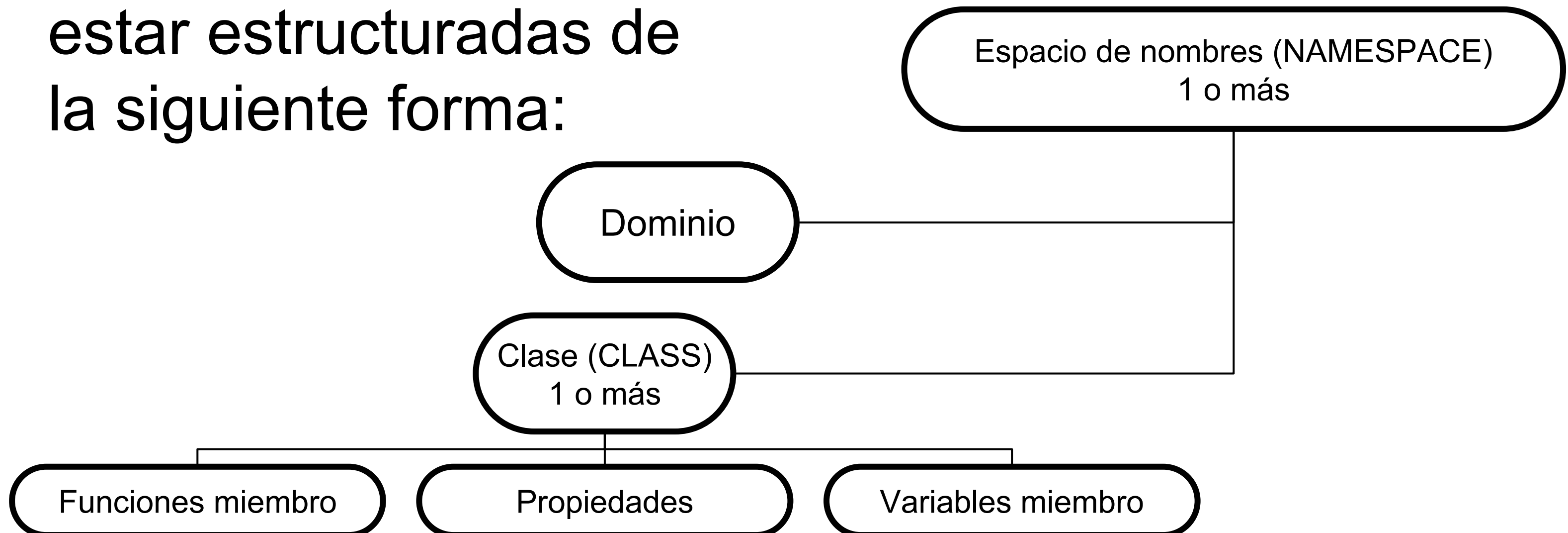
- Usa una identificación para el módulo basada en un **strongname** (*.snk)
- Si el **strongname** del solicitante conincide con el del solicitado este módulo se carga del GAC, aun siendo ficheros de tamaños y fechas diferentes
- Es el método habitual si se quiere usar el GAC

Las librerías .NET

Uso y diseño

La jerarquía de clases .NET

- Todos los programas / librerías de cualquier lenguaje .NET han de estar estructuradas de la siguiente forma:



La jerarquía de clases .NET

- La jerarquía de clases de .NET es bastante grande, pero podemos distinguir varios tipos de librerías:
  - Convencionales: Iguales que las DLLs que podemos hacer nosotros con un lenguaje .NET
  - Importaciones COM / ActiveX: Permiten el uso de clases y tipos COM como si fueran tipos convencionales.
  - Importaciones OLB: Permiten que escribamos código .NET que llame a las instrucciones VBA de una aplicación VBA como es el Office.
  - Marshals: Permiten invocar código escrito en DLLs de código nativo.
  - De ayuda para compiladores de .NET
  - Acceso Remoto

La jerarquía de clases .NET

- Si escribimos una clase en un lenguaje .NET sufriremos sus restricciones:
  - La herencia múltiple
    - Toda clase puede derivar de otra, pero solo una de ellas puede ser una clase convencional, las demás han de ser interfaces, o lo que es lo mismo con funciones miembro sin código, solo cabeceras y sin variables miembro (aunque si propiedades)

La jerarquía de clases .NET

- Las clases selladas
  - Son un tipo de clases que no se pueden heredar.
- Las clases abstractas
  - Son clases “normales”, pero no se pueden instanciar, por tanto, deberemos crear una clase derivada y usar esta.
  - No se deben confundir con las interfaces.

Librería de clase base (BCL)

- La Librería de Clase Base (BCL) es una librería incluida en el *.NET Framework* formada por cientos de tipos de datos que permiten acceder a los servicios ofrecidos por el CLR y a las funcionalidades más frecuentemente usadas a la hora de escribir programas.

Librería de clase base: Los espacios de nombres mas usados

Espacio de nombres	Utilidad de los tipos de datos que contiene
System	Tipos muy frecuentemente usados, como los los tipos básicos, excepciones, fechas, números aleatorios, entrada/salida en consola, etc.
System.Collections	Colecciones de datos de uso común como listas, diccionarios, etc.
System.Data	Manipulación de bases de datos. (**ADO.NET**.)
System.IO	Manipulación de ficheros y otros flujos de datos.
System.Net	Realización de comunicaciones en red.
System.Reflection	Acceso a los metadatos que acompañan a los módulos de código.
System.Runtime.Remoting	Acceso a objetos remotos.
System.Security	Acceso a la política de seguridad en que se basa el CLR.
System.Threading	Manipulación de hilos.
System.Web	ASP.NET
System.Windows.Forms	Creación de interfaces de usuario basadas en ventanas.
System.XML	Acceso a datos en formato XML.

El lenguaje C#

La revolución de la sencillez

Origen

- **C#** (leído en inglés “C Sharp” y en español “C Almohadilla”) es el nuevo lenguaje de propósito general diseñado por Microsoft para su plataforma .NET. Sus principales creadores son Scott Wiltamuth y **Anders Hejlsberg**, éste último también conocido por haber sido el diseñador del lenguaje **Turbo Pascal** y la herramienta RAD **Delphi**.

¿Por qué es mejor C# para .NET?

- Aunque es posible escribir código para la plataforma .NET en muchos otros lenguajes, C# es el único que ha sido diseñado específicamente para ser utilizado en ella, por lo que programarla usando C# es mucho más sencillo e intuitivo que hacerlo con cualquiera de los otros lenguajes ya que C# carece de elementos heredados innecesarios en .NET. Por esta razón, se suele decir que C# es el **lenguaje nativo de .NET**

¿Porqué no usar entonces Java?

- Un lenguaje que hubiese sido ideal utilizar para estos menesteres es Java, pero debido a problemas con la empresa creadora del mismo (Sun), Microsoft ha tenido que desarrollar un nuevo lenguaje que añadiese a las ya probadas virtudes de Java las modificaciones que Microsoft tenía pensado añadirle para mejorarlo aún más y hacerlo un lenguaje orientado al desarrollo de componentes.

Al ser nuevo tendrá muchos errores ¿no?

- En resumen, C# es un lenguaje de programación que toma las mejores características de lenguajes preexistentes como Visual Basic, Java o C++ y las combina en uno solo. El hecho de ser relativamente reciente no implica que sea inmaduro, pues Microsoft ha escrito la mayor parte de la BCL usándolo, por lo que su compilador es el más depurado y optimizado de los incluidos en el *.NET Framework SDK*

Características

- Las ofrecidas por ser lenguaje .NET (seguridad de tipos,...)
- **Sencillez:** C# elimina muchos elementos que otros lenguajes incluyen y que son innecesarios en .NET
  - El código escrito en C# es **autocontenido**, lo que significa que no necesita de ficheros adicionales al propio fuente tales como ficheros de cabecera o ficheros IDL
  - No se incluyen elementos poco útiles de lenguajes como C++ tales como macros, herencia múltiple o la necesidad de un operador diferente del punto (**.**) acceder a miembros de espacios de nombres (**::**)

Características

- **Modernidad:** C# incorpora en el propio lenguaje elementos que a lo largo de los años ha ido demostrándose son muy útiles para el desarrollo de aplicaciones y que en otros lenguajes como Java o C++ hay que simular, como:
  - Un tipo básico **decimal** que permite realizar operaciones de alta precisión con reales de 128 bits (muy útil en el mundo financiero).
  - La inclusión de una instrucción **foreach** que permite recorrer colecciones con facilidad y es ampliable a tipos definidos por el usuario.
  - La inclusión de un tipo básico **string** para representar cadenas.
  - La distinción de un tipo **bool** específico para representar valores lógicos.

Características

- **Extensibilidad de modificadores:** C# ofrece, a través del concepto de **atributos**, la posibilidad de añadir a los metadatos del módulo resultante de la compilación de cualquier fuente información adicional a la generada por el compilador que luego podrá ser consultada en tiempo ejecución a través de la librería de reflexión de .NET . Esto, que más bien es una característica propia de la plataforma .NET y no de C#, puede usarse como un mecanismo para definir nuevos modificadores.

Características

- **Eficiente:** En principio, en C# todo el código incluye numerosas restricciones para asegurar su seguridad y no permite el uso de punteros. Sin embargo, y a diferencia de Java, en C# es posible saltarse dichas restricciones manipulando objetos a través de punteros. Para ello basta marcar regiones de código como inseguras (modificador **unsafe**) y podrán usarse en ellas punteros de forma similar a cómo se hace en C++, lo que puede resultar vital para situaciones donde se necesite una eficiencia y velocidad procesamiento muy grandes.

Ejemplo: Hola mundo!

- El ejemplo típico de cualquier lenguaje:

```
using System;

namespace ConsoleApplication1
{
   class Class1
   {
        [STAThread]
        static void Main(string[] args)
        {
               Console.WriteLine("Hola mundo!");
        }
   }
}
```

Ejemplo: Hola mundo!

- Compilación:
  - Automáticamente desde Visual Studio
  - Manualmente desde la línea de comandos
    - Ejecutando

      CSC HolaMundo.cs
    - Y genera automáticamente HolaMundo.exe
- Ejecución: Como un ejecutable cualquiera.

El preprocesador

- Es como el de C/C++ pero con restricciones
- Es prácticamente inútil porque no soporta macros, INCLUDE ni IFDEF con expresiones.

```
#define PRUEBA
using System;
class A
{
      public static void Main()
      {
                #if PRUEBA
                          Console.Write (“Esto es una prueba”);
                          #if TRAZA
                                    Console.Write(“ con traza”);
                          #elif !TRAZA
                                    Console.Write(“ sin traza”);
                          #endif
                #endif
      }
}
```

El preprocesador: Marcación de regiones de código

- Es posible marcar regiones de código y asociarles un nombre usando el juego de directivas **#region** y **#endregion**. Estas ocultan código para hacer que la navegación por el código fuente sea mas eficiente, pero no conviene abusar.

```
#region <nombreRegión>
<código>
#endregion
```

Aspectos léxicos

- Comentarios: // y /* */
- Identificadores: como los de C/C++
- Palabras reservadas:
  - 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, lock, is, 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, virtual, void, while.
- Distingue mayúsculas de minúsculas
- Literales como C/C++ excepto:
  - @"e\t" que se entiende como "e\t" y no como "e   "
  - Los strings / caracteres son Unicode
- El nulo: null
- Operadores: como los de C/C++

Tipos u usos

- Nuevas instancias (de clases). Ej.:
  - a = new atype();
- Arrays:
  - Se escriben como tipo[] o tipo[,] o tipo [,,] ...
  - Ejemplo:
    - string[] lista = new string[50];
    - String[] Lista2 = new string[]{“uno”,”dos”};
    - Lista2[0] = “”;
- Typeof:
  - **typeof(**<nombreTipo>**)**
  - <expresión> **is** <nombreTipo>
- Sizeof : **sizeof(**<nombreTipo>**)**
- Object as type

Clase por ejemplo

```
class Trabajador:Persona
{
  public int Sueldo;
  public Trabajador(string nombre, int edad, string
nif, int sueldo)
                  : base(nombre, edad, nif)
  {
     Sueldo = sueldo;
  }
}
```

Métodos virtuales

- **Base**

```
virtual <tipoDevuelto> <nombreMétodo>(<parámetros>)
{
   <código>
}
```

- **Derivada**

```
override <tipoDevuelto> <nombreMétodo>(<parámetros>)
{
   <nuevoCódigo>
}
```

Clases abstractas

```
public abstract class A
{
                public abstract void F();
}
public abstract class B: A
{
                public void G() {}
}
class C: B
{
                public override void F()
        {}
}
```

Clases selladas

- Una **clase sellada** es una clase que no puede tener clases hijas. Ej. :

```
sealed class ClaseSellada
{
      }
```

Miembros de clases sellados

```
class A
{
	public abstract F();
}
class B:A
{
	public sealed override F() // F() deja de ser redefinible
	{}
}
```

Espacios de nombres

- Pueden ser simples

```
namespace EspacioEjemplo
{
  class ClaseEjemplo
  {}
}
```

- O anidados

```
namespace EspacioEjemplo.EspacioEjemplo2
{
  class ClaseEjemplo
  {}
}
```

Importación de espacios de nombres

- Se hace mediante “using”

```
using EspacioEjemplo.EspacioEjemplo2;
namespace EspacioEjemplo.EspacioEjemplo2
{
   class ClaseEjemplo
   {}
}
class Principal    // Pertenece al espacio de nombres global
{
   public static void ()
   {
   // EspacioEjemplo.EspacioEjemplo2. está implícito
   ClaseEjemplo c  = new ClaseEjemplo();
   }
}
```

Funciones miembro

- Ej. : public void f(int x)
- El paso de valores se hace por valor, si queremos que se haga por referencia hemos de añadir ref antes del valor. Si modificamos una instancia esto no es necesario ya que no cambiamos su referencia sino su contenido.
- Si el valor no está definido ref dará un error de compilación, ya que exige que el valor esté definido de antemano (aunque sea null), pero si lo que queremos es devolver una referencia hemos de poner la palabra clave out en vez de ref.

Funciones miembro externas

- Son aquellas que residen en otros ficheros como funciones del API en C. Ej.:

```
using System.Runtime.InteropServices;  // Aquí está
definido DllImport
public class Externo
{
     [DllImport("kernel32")]
     public static extern void CopyFile(string fuente,
string destino);
     public static void Main()
     {
            CopyFile("fuente.dat", "destino.dat");
     }
}
```

Propiedades

```
using System;
abstract class A{
      public abstract int PropiedadEjemplo{   set;      get;      }
}
class B:A       {
      private int valor;
      public override int PropiedadEjemplo{
                get{
                          Console.WriteLine("Leído {0} de PropiedadEjemplo",
valor);
                          return valor;
                }
                set{
                          valor = value;
                          Console.WriteLine("Escrito {0} en
PropiedadEjemplo", valor);

                }
      }
}
```

Delegados y eventos (1/3)

- Un **delegado** es un tipo especial de clase cuyos objetos pueden almacenar referencias a uno o más métodos, es decir, es un puntero a función, pero puede ser múltiple, es decir contiene un conjunto de funciones iguales en forma pero no en ubicación.
- Un **evento** es una propiedad de una clase que trabaja con tipo delegado.

Delegados y eventos (2/3)

- Ej. Como puntero a función normal

```
using System;

public delegate void D();
class A
{
  public static D obj;
  public static void Main()
  {
            B.AlmacenaPrivado();
            obj();
  }
}
class B
{
  private static void Privado()
  { Console.WriteLine("Llamado a método privado"); }
  public static void AlmacenaPrivado()
  { A.obj += new D(Privado); }
}
```

Delegados y eventos (3/3)

- Ej. Como puntero a función múltiple

```
using System;

public delegate void D();
class A     {
  public static D obj;
  public static void Main()              {
            B.AlmacenaPrivado();
            obj();
  }
}
class B     {
  private static void Privado()
  { Console.WriteLine("Llamado a método privado"); }
  private static void Privado2()
  { Console.WriteLine("Llamado a método privado"); }
  public static void AlmacenaPrivado()
  { A.obj += new D(Privado);
   A.obj += new D(Privado2); }
}
```

Structs (1/2)

- Son clases sencillas y pequeñas sin herencia que se tratan por valor.
  - Ej. :

```
struct Point{
        public int x, y;
        public Point(int x, int y) {
                this.x = x;
                this.y = y;
        }
  }
```

  - Uso

```
Punto p = new Punto(10,10);
Punto p2 = p;
p2.x = 100;
Console.WriteLine(p.x);
```

- Esto hay que tenerlo en cuenta a la hora de pasarlos como parámetros.

Structs (2/2)

- Boxing y unboxing
  - Sucede cuando el sistema necesita que el struct sea un object, como por ejemplo las colecciones.
  - El struct se convierte en un bloque en memoria dinámica y se recupera mediante un cast.
    - OJO: si el hacemos (int)((object)4.0) provoca un error.

Instrucciones del C#

Asignaciones,
Control,...

Definiciones de variables locales

- Esquema:
  - <modificadores> <tipoVariable> <nombreVariable> **=** <valor>**;**
- Ej.:
  - int a=5, b, c=-1;

Asignaciones

- Esquema:
  - <destino> = <origen>;
- Ej. :
  - J=1;
- Aviso:
  - Cuando asignamos una instancia de una clase a otra copiamos el puntero y no el valor.
  - Cuando asignamos una instancia de un struct a una variable copiamos el valor.

Llamadas a métodos

- Dentro de un objeto
  - <objeto>**.**<nombreMétodo>**(**<*valoresParámetros*> **);**
  - Ej. J.ToString();
- Estáticos
  - <*nombreTipo*>***.***<nombreMétodo>**(**<*valoresParámetros*>**);**
  - Ej. GC.Collect();

IF

- Esquema:

  if (<condición>**)**

  <instruccionesIf>

  else

  <instruccionesElse>

Instrucción switch

- Esquema:

```
switch (<expresión>)
{
  case <valor1>:  <bloque1>
                  <siguienteAcción>
  case <valor2>:  <bloque2>
                  <siguienteAcción>
  ...
  default:  <bloqueDefault>
            <siguienteAcción>
}
```

- Donde <siguienteAcción> es uno de :

```
goto case <valori>;
goto default;
break;
```

- Y donde <expresión> es un tipo enumerable o un string.

While, do y for

- While, do y for son como en C soportando break y continue.
- Ej.

```
do
{
        Console.WriteLine("Clave: ");
        leído = Console.ReadLine();
}
while (leído != "José");
```

Foreach

- Pensada para clases derivadas de IEnumerable y para simplificar la iteración
- Esquema

  foreach (<tipoElemento> <elemento> **in** <colección>**)**

  <instrucciones>

- Ej. :

```
public static void Main(String[] args){
        if (args.Length > 0)
                foreach(String arg in args)
                        Console.WriteLine(“¡Hola {0}!”, arg);
        else
                Console.WriteLine(“¡Hola mundo!”);
}
```

Excepciones 1/2

- Son objetos derivados de Exception
- Se lanzan con throw
  - Ej. throw new ArgumentNullException();
- Se capturan con try / catch / finally

  try
  <instrucciones>
  catch (*<excepción1>****)***
  <tratamiento1>
  catch (*<excepción2>****)***
  <tratamiento2>
  ...
  finally
  <instruccionesFinally>

Excepciones 2/2

- Ej. :

```
try
{
  int c = 0;
  int d = 2/c;
}
catch (DivideByZeroException)
{
  d=0;
}
```

Instrucciones checked y unchecked

- La instrucción unchecked{....} evita que se lance una excepción **System.OverflowException**
- Ej.

```
class Unchecked
{
   static short x = 32767;   // Valor maximo del tipo short
   public static void Main()
   {
        unchecked
        {
                    Console.WriteLine((short) (x+1));  // (1)
        Console.WriteLine((short) 32768); // (2)
        }
   }
}
```

Instrucción lock

- La instrucción lock es útil en aplicaciones concurrentes donde múltiples hilos pueden estar accediendo simultáneamente a un mismo recurso, ya que lo que hace es garantizar que un hilo no pueda acceder a un recurso mientras otro también lo esté haciendo. Su sintaxis es la siguiente:

  lock (<objeto>**)**

  <instrucciones>

Atributos 1/2

- Un **atributo** es información que se puede añadir a los metadatos de un módulo de código. Esta información puede ser referente tanto al propio módulo o el ensamblado al que pertenezca como a los tipos de datos en definidos en él, sus miembros, los parámetros de sus métodos, los bloques **set** y **get** de sus propiedades e indizadores o los bloques **add** y **remove** de sus eventos.
- Ej. Protected, public,....

Atributos 2/2

- Avanzados. Tienen la forma:
  [<nombreAtributo>***(****<parámetros>****)*****]**
- El siguiente ejemplo muestra cómo usar **Conditional**:

```
using System;
using System.Diagnostics;
class Condicional{
  [Conditional("DEBUG")]
  public static void F(){
Console.WriteLine("Ff()"); }
  public static void Main() {
          F();
  }
}
```

- Sólo si compilamos el este código definiendo la constante de preprocesado DEBUG se mostrará por pantalla el mensaje F() En caso contrario, nunca se hará la llamada a F()

Mas instrucciones

- Existen muchas mas instrucciones para fines diversos:
  - Unsafe
  - Fixed
  - ...

Herramientas del lenguaje C#

Lo más potente

El documentador XML

```
/// <summary>
///   Método que muestra por
/// pantalla un texto con un determinado color
/// </summary>
/// <param name=”texto”>
///Texto a mostrar </param>
/// <param  name=”color”>
///   Color con  el que mostrar
/// el <paramref name=”texto”/> indicado
/// </param>
bool MuestraTexto(string texto, Color color)
{...}
```

El uso de objetos remotos: Server

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
namespace RemotingSamples {
public class Sample {
public static int Main(string [] args) {
TcpChannel chan = new TcpChannel(8085); ChannelServices.RegisterChannel(chan);
    RemotingConfiguration.RegisterWellKnownServiceType(
    Type.GetType("RemotingSamples.HelloServer,object"), "SayHello",
    WellKnownObjectMode.SingleCall); System.Console.WriteLine("Hit to exit...");
    System.Console.ReadLine();
return 0;
} }
public class HelloServer : MarshalByRefObject {
public HelloServer() {
Console.WriteLine("HelloServer activated");
}
public String HelloMethod(String name) {
Console.WriteLine("Hello.HelloMethod : {0}", name);
return "Hi there " + name;
} }
}
```

El uso de objetos remotos: Client

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
namespace RemotingSamples {
    public class Client {
        public static int Main(string [] args) {
        TcpChannel chan = new TcpChannel();
        ChannelServices.RegisterChannel(chan);
        HelloServer obj = (HelloServer)Activator.GetObject(
          typeof(RemotingSamples.HelloServer),
          "tcp://localhost:8085/SayHello");
        if (obj == null) System.Console.WriteLine("Could not locate server");
          else Console.WriteLine(obj.HelloMethod("Caveman"));
        return 0;
        }
    }
}
```

El serializador XML

- Escribir
  - XmlSerializer serializer = new XmlSerializer(typeof(PurchaseOrder));
  - TextWriter writer = new StreamWriter("po2.xml");
  - serializer.Serialize(writer, po);
  - writer.Close();
- Leer
  - XmlSerializer serializer = new XmlSerializer(typeof(PurchaseOrder));
  - TextReader reader = new StreamReader("po.xml");
  - PurchaseOrder po = (PurchaseOrder)serializer.Deserialize(reader);
  - reader.Close();

ADO.NET

```
SqlDataReader myDataReader = null;
SqlConnection mySqlConnection = new SqlConnection(“...");
SqlCommand mySqlCommand = new SqlCommand("SELECT EmployeeID,
   LastName, FirstName, Title, ReportsTo FROM Employees",
   mySqlConnection);
...
mySqlConnection.Open();
myDataReader = mySqlCommand.ExecuteReader(...);
while (myDataReader.Read()) {
Console.Write(myDataReader.GetInt32(0) + "\t");
...
}
// Always call Close when done reading.
myDataReader.Close();
// Close the connection when done with it.
mySqlConnection.Close();
```

Compilador en tiempo de ejecución

- Primero compilamos el conjunto de ficheros desde el código de nuestra aplicación con:

```
public static CompilerError[] Compile(
        string[] sourceTexts,
        string[] sourceTextNames,
        string target,
        string[] imports,
        IDictionary options );
```

- Después cargamos el resultado dinámicamente

Criptografía

```
FileStream fs = new FileStream("EncryptedFile.txt",
FileMode.Create,FileAccess.Write);
DESCryptoServiceProvider des = new
  DESCryptoServiceProvider();
ICryptoTransform desencrypt =
  des.CreateEncryptor();
CryptoStream cryptostream = new
  CryptoStream(fs,desencrypt,CryptoStreamMode.Write
  );
cryptostream.Write(bytearrayinput,0,bytearrayinput.
  Length);
cryptostream.Close();
```

Y mas y mas y mas........

- En el Visual Studio .NET
- En el .NET framework SDK
- O en http://ultralight32.dnsalias.net sección dessarrollo

Fin