C#

Pablo Orduña Fernández (aka NcTrun)

Julio 2006

DotNetGroup

This work is licensed under the Creative Commons Attribution License. To view a copy of this license, visit http://creativecommons.org/licenses/by/2.0/ or send a letter to Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA

Introducción al cursillo
Introducción a . . .
Introducción a C#
Introducción breve . . .
Algunas novedades . . .
Referencias

1. Introducción al cursillo

1.1. De qué va este cursillo

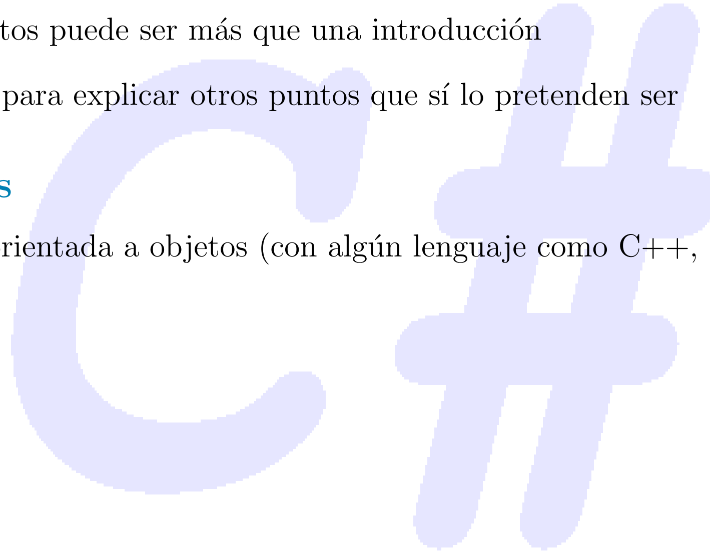
- El cursillo está entre los Cursillos de Julio de los grupos de interés de la Universidad de Deusto
  - Cursillos de Julio:
    - Desde hace varios años, alumnos y alumnas de la Facultad de Ingeniería de la Universidad de Deusto organizan de manera voluntaria una serie de cursillos que abarcan diversas áreas de conocimiento
    - Esta actividad es coordinada por la Delegación de Alumnos
    - Cuenta con el apoyo de profesores y de la Facultad de Ingeniería-ESIDE, que anima e impulsa estas actividades facilitando el uso de aulas informatizadas y demás recursos para que su realización sea lo mejor posible.
  - Filosofía de los cursillos
    - ¡Compartir conocimiento!
    - Ayudar a dar los *primeros pasos* de una tecnología, lenguaje de programación, etc
      - En consecuencia: En un cursillo se abarcan la máxima cantidad de temas en el mínimo tiempo posible. No es posible profundizar mucho en cada tema, pero sí ver lo suficiente para que el/la alumno/a pueda seguir aprendiendo por su cuenta, una vez dados los primeros pasos.
      - Cursillos introductorios, no exhaustivos
  - Más información sobre los Cursillos de Julio
- Este concretamente se da desde el grupo de .NET de la Universidad (el DotNetGroup)

1.2. Objetivos

- Pretende ser una introducción a C#
  - cubriendo algunos de los temas más divertidos del lenguaje
  - de lo que hay a día de hoy (12-14 de Julio de 2006): C# 2
- En algunos puntos puede ser más que una introducción
  - Suelen ser para explicar otros puntos que sí lo pretenden ser

1.3. Requisitos

- Programación orientada a objetos (con algún lenguaje como C++, Java...)



Introducción al cursillo
Introducción a ...
Introducción a C#
Introducción breve ...
Algunas novedades ...
Referencias

2. Introducción a Mono/.NET

2.1. ¿Qué es?

- El .NET Framework es una *plataforma de desarrollo de software*, enfocada en:
  - Desarrollo rápido y explotación de aplicaciones gestionadas (*managed*) y orientadas a objetos
  - Independencia del lenguaje
  - Independencia de la plataforma
  - Transparencia a través de la red
- Esta plataforma ofrece, entre otras cosas:
  - Nuevos y modernos lenguajes de programación (C#, VB.NET. . .)
  - Compatibilidad con otros lenguajes (Managed C++, J#. . .)
  - La posibilidad de incluir nuevos lenguajes de programación
  - Integración multilenguaje, reutilización de componentes, herencia entre componentes desarrollados en diferentes lenguajes
  - Una extensa framework de librerías de clases independiente del lenguaje
  - Un sistema de ejecución de lenguaje común (CLR)
  - Un conjunto de servidores .NET
  - Programación
    - web: ASP.NET
    - gráfica: Windows Forms
    - de Servicios Web XML independientes de la plataforma vía SOAP y WSDL
  - Conjunto de herramientas de desarrollo (Visual Studio .NET, ...)

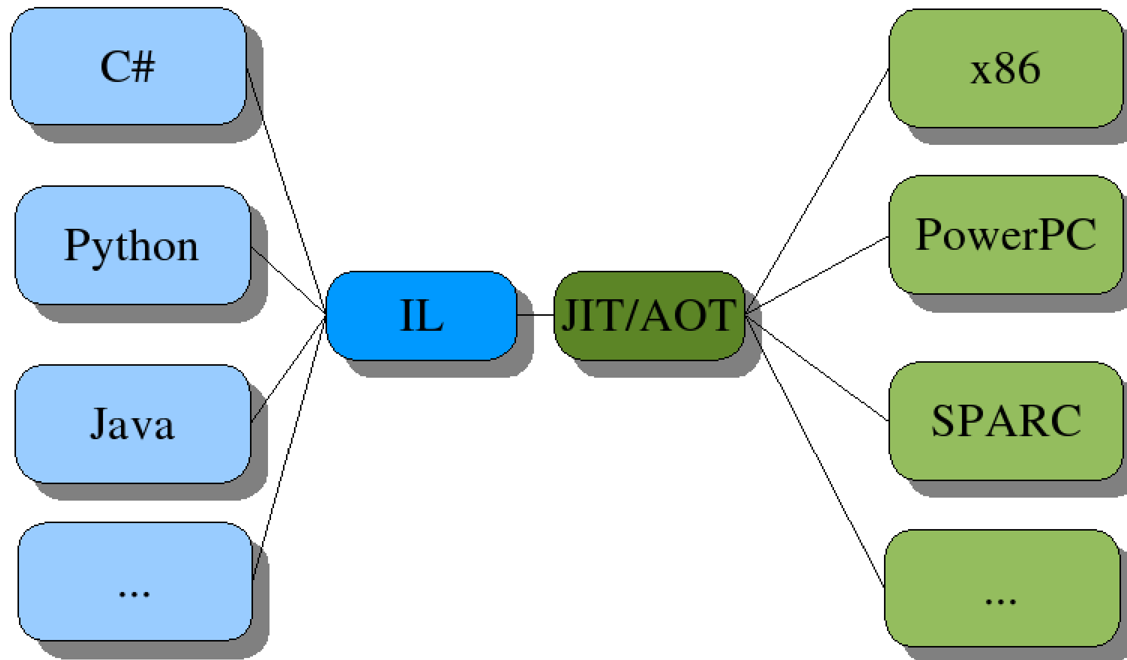
2.2. Common Language Runtime

2.2.1. Características

- La CLR trae incorporadas las características de un runtime moderno:
  - Recolección de basura
  - Gestión de hilos
  - Interoperabilidad con código nativo
  - Seguridad en el acceso a código
  - Introspección
  - ...
- El CLR es el encargado de ejecutar las aplicaciones .NET

2.2.2. Soporte multilenguaje

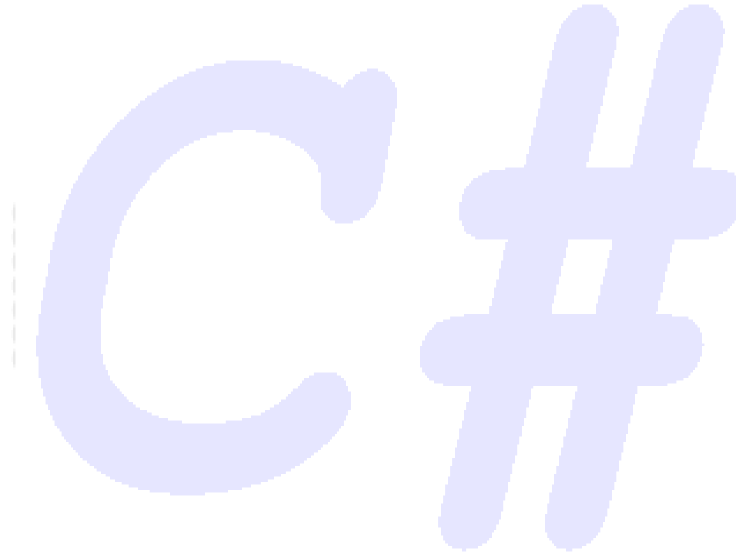
- La plataforma es independiente del lenguaje
- Cuenta con un lenguaje universal, el CIL
  - Common Intermediate Language, también llamado IL o MSIL
  - fácilmente compilable
  - cada lenguaje tiene su compilador a CIL
- Luego, del CIL se genera el código nativo de la plataforma en la que se ejecute
  - compilador JIT (Just In Time) o AOT (Ahead Of Time) o intérprete
  - las diferentes implementaciones de .NET soportan diferentes plataformas



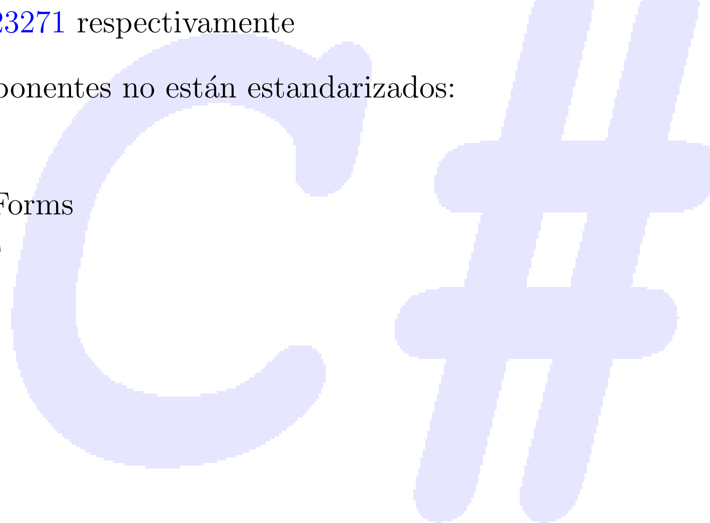
- Gracias a esto:
  - Se puede desde un lenguaje utilizar componentes escritos en otro lenguaje
  - Dada una librería, su API es accesible a todos los lenguajes
  - Lo único que es necesario es que estos lenguajes tengan su compilador a CIL
  - Incluso se pueden reutilizar compiladores hechos por terceros

2.2.3. Librerías

- Como hemos dicho, .NET trae una gran cantidad de librerías
- Todas estas librerías son accesibles a todo lenguaje que cuente con compilador para el CIL



2.3. Estandarización de .NET



- Microsoft estandarizó parte del .NET Framework en el ECMA
  - En los estándares ECMA 334 (C#) y ECMA 335 (CLI: Common Language Infrastructure)
    Estandarizó el núcleo de .NET y C#
  - C# y la CLI además está estandarizado por la ISO en los estándares ISO/IEC 23270 y ISO/IEC 23271 respectivamente
- Pero otros componentes no están estandarizados:
  - ASP.NET
  - Windows Forms
  - ADO.NET

2.4. Implementaciones

- Microsoft tiene su implementación de .NET
- Pero hay más implementaciones, algunas Open Source, como:
  - Mono
  - DotGNU
- En el aula están instaladas:
  - La implementación de Microsoft en Windows
    - .NET 1.1 (Visual Studio 2003)
      - Microsoft ha publicado este curso VS.NET 2005, con .NET 2
  - Mono en Ubuntu
    - Versión 1.1.16
- La versión 1.1.16 de Mono implementa C# 2, que daremos en clase

  Aunque todo lo que vayamos a dar sea estándar, en la versión instalada en el aula hay cosas que sólo funcionarán con Mono

Introducción al cursillo
Introducción a . . .
Introducción a C#
Introducción breve . . .
Algunas novedades . . .
Referencias

2.5. Versiones

Versiones del .NET Framework

- 1.0 $\longrightarrow$ Enero 2002
- 1.1 $\longrightarrow$ Abril 2003. Instalado en el aula en la partición de Windows. Viene con C# 1
- 2.0 $\longrightarrow$ Noviembre 2005. En español a comienzos de 2006. Viene con C# 2

En Mono:

- Algunas características de C# 2 están disponibles en el compilador `mcs`
- Para hacer uso de todas las características de C# 2, hay que utilizar `gmcs`
- En MonoDevelop, basta con activar el runtime 2.0 en las Opciones del Proyecto.

2.6. Instalación de Mono

- http://www.mono-project.com/Downloads
- En Windows hay un instalador que viene con Mono, GTK# y xsp
- Para GNU/Linux en x86 está disponible un instalador para cualquier distribución
  - Está muy bien para cacharrear con Mono:
    * Fácil de instalar (instalador `GTK`)
    * Instalación limpia (desinstalación limpia)
    * Viene con la última versión de Mono
    * Independiente de dependencias etc. (da lo mismo si usas sid, testing, suse o lo que sea)
  - Por otra parte, también tiene sus desventajas:
    * Es fácil para hacer poca cosa. Para hacer que `xsp` se ejecute como usuario www-data, tienes que configurarlo tú mismo (que es hacer lo que los mantenedores de los paquetes ya han hecho por tí)
    * Dificil de mantener: tienes que estar pendiente de actualizaciones
    * Las dependencias no lo detectan. Algunos programas exigen otras dependencias.

Introducción al cursillo
Introducción a . . .
Introducción a C#
Introducción breve . . .
Algunas novedades . . .
Referencias

2.7. Introducción al entorno

- Mono viene con las baterías puestas:
  - IDE: MonoDevelop
    - Port del SharpDevelop para GNOME, con múltiples nuevas características para desarrollo de aplicaciones para GNOME
  - Muchas otras herramientas que quedan fuera del cursillo
- Vamos a poner un poco en práctica lo que hemos comentado hasta ahora (todo esto se explica mejor en el cursillo de Mono):
  - Vamos a `ejemplos/holamundo`
  - Compilamos el "hola mundo": `mcs holamundo.cs`

```
nctrun@ord3p:~/cursillo_cs$ cd ejemplos/holamundo/
nctrun@ord3p:~/cursillo_cs/ejemplos/holamundo$ mcs holamundo.cs
holamundo.cs(26,7): warning CS0219: The variable ‘dato3’ is assigned but its
value is never used
Compilation succeeded - 1 warning(s)
nctrun@ord3p:~/cursillo_cs/ejemplos/holamundo$ ls
holamundo.cs  holamundo.exe
nctrun@ord3p:~/cursillo_cs/ejemplos/holamundo$
```

  - ¿Qué tipo de fichero es?

```
nctrun@ord3p:~/cursillo_cs/ejemplos/holamundo$ file holamundo.exe
holamundo.exe: PE executable for MS Windows (console) Intel 80386 32-bit
```

  - Vaya. ¿Cómo lo ejecuto?

Introducción al cursillo
Introducción a ...
Introducción a C#
Introducción breve ...
Algunas novedades ...
Referencias

```
nctrun@ord3p:~/cursillo_cs/ejemplos/holamundo$ mono holamundo.exe
Hola mundo 37
```

– Guay. . . pero. . . ¿Todo eso del CIL? ¿Era pipa?

```
nctrun@ord3p:~/cursillo_cs/ejemplos/holamundo$ monodis holamundo.exe |less
.assembly extern mscorlib
{
    .ver 1:0:5000:0
    .publickeytoken = (B7 7A 5C 56 19 34 E0 89 ) // .z\V.4..
}
.assembly 'holamundo'
{
    .hash algorithm 0x00008004
//...
```

– ¿Y lo de AOT?

∗ Por defecto, estamos utilizando el compilador JIT al hacer el `mono holamundo.exe`

∗ Si queremos utilizar el compilador AOT:

```
nctrun@ord3p:~/cursillo_cs/ejemplos/holamundo$ mono --aot -O=all
   holamundo.exe

Mono Ahead of Time compiler - compiling assembly
 /home/nctrun/cursillo_cs/ejemplos/holamundo/holamundo.exe
Code: 205 Info: 34 Ex Info: 17 Class Info: 31 PLT: 4 GOT: 24
Executing the native assembler: as /tmp/mono_aot_7PL9KZ -o
   /tmp/mono_aot_7PL9KZ.o
Executing the native linker: ld -shared -o
 /home/nctrun/cursillo_cs/ejemplos/holamundo/holamundo.exe.so
   /tmp/mono_aot_7PL9KZ.o
```


Introducción al cursillo
Introducción a . . .
Introducción a C#
Introducción breve . . .
Algunas novedades . . .
Referencias

```
Compiled 5 out of 5 methods (100%)
0 methods contain absolute addresses (0%)
0 methods contain wrapper references (0%)
0 methods contain lmf pointers (0%)
0 methods have other problems (0%)
AOT RESULT 0
nctrun@ord3p:~/cursillo_cs/ejemplos/holamundo$ file holamundo.exe.so
holamundo.exe.so: ELF 32-bit LSB shared object, Intel 80386, version 1
   (SYSV), not stripped
```

– ¿Y lo de integración entre lenguajes?
  * Cojamos como ejemplo Java
    · En el aula está instalado tanto Java 1.4 como Java 1.5 de Sun Microsystems
    · Para cambiar de versión: `cambiarJava 1.4` (como root)
  * Vamos a `ejemplos/libreria`

```
nctrun@ord3p:~/cursillo_cs/ejemplos/libreria$ javac MiClase.java
nctrun@ord3p:~/cursillo_cs/ejemplos/libreria$ ikvmc -out:MiClase.dll
   MiClase.class

Note: automatically adding reference to
   "/home/nctrun/mono-1.1.16/lib/ikvm/IKVM.GNU.Classpath.dll"
nctrun@ord3p:~/cursillo_cs/ejemplos/libreria$ export
   MONO_PATH=~/mono-1.1.16/lib/ikvm/:$MONO_PATH
nctrun@ord3p:~/cursillo_cs/ejemplos/libreria$ mcs DesdeCS.cs -r:MiClase.dll
nctrun@ord3p:~/cursillo_cs/ejemplos/libreria$ mono DesdeCS.exe
hola mundo
llamando al metodo
nctrun@ord3p:~/cursillo_cs/ejemplos/libreria$
```

– Más ejemplos → Cursillo de Mono

Introducción al cursillo
Introducción a . . .
Introducción a C#
Introducción breve . . .
Algunas novedades . . .
Referencias

3. Introducción a C#

3.1. Introducción

- Lenguaje sencillo
- Orientado a objetos
- El siguiente en lenguajes derivados de `C`, pasando por `C++` y `Java`
  - Recoge muchas de las cosas que `Java` no había cogido de `C++`
  - Asímismo, incorpora varias novedades no presentes en ninguno
  - La `#` viene de que es `C++++`, donde `++` encima de `++` es `#`
- Permite tanto la programación `managed` como la `unsafe` (con punteros como en `C++`)
- Más simple que `C++`, pero tan poderoso y flexible

Introducción al cursillo
Introducción a ...
Introducción a C#
Introducción breve ...
Algunas novedades ...
Referencias

3.2. Comenzando con C#

3.2.1. Ficheros de código en C#

- Al igual que en `C++`, C# tiene:
  - uno o más ficheros (con extensión `.cs`)
  - el cual tiene o no espacios de nombres (`namespaces`)
  - que generan un único ejecutable o librería (`.exe` o `.dll`)

3.2.2. Hola mundo en C#

- Primer programa en C# *(T3_E1_Holamundo)*:

```
class A{
    static void Main(){
        System.Console.WriteLine("hola mundo");
    }
}
```

- Hola mundo explicado:
  - `class`
    * Al igual que en `Java`, *todo código aparecerá en métodos de clases*
  - `Main`
    * El punto de entrada (`entrypoint`) del programa estará en un método especial llamado `Main`, que podrá ser declarado tal que:

Introducción al cursillo
Introducción a ...
Introducción a C#
Introducción breve ...
Algunas novedades ...
Referencias

```
static void Main();
static int Main();
static void Main(string [] args);
static int Main(string [] args);
```

- ∗ Sólo podrá haber un `Main` por ensamblado ejecutable.
- ∗ `C#` es *case-sensitive*.

– `System.Console`

- ∗ Al ejecutar `System.Console.WriteLine("hola mundo");` estamos:
  - · llamando al método `WriteLine`
  - · de la clase `Console`
  - · del espacio de nombres `System`
- ∗ Podríamos sustituirlo por *(T3_E2_Holamundo)*:

```
using System;

class A{
    static void Main(){
        Console.WriteLine("hola mundo");
    }
}
```

- ∗ De manera que podemos acceder a todas las clases, espacios de nombres, `delegates`, interfaces, y enumeraciones que cuelgan de `System`

3.2.3. Ejecutandolo

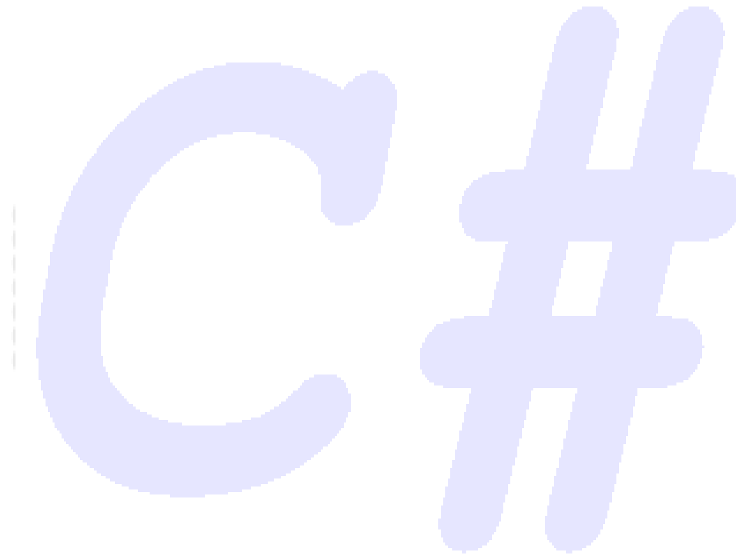
- Primero lo compilamos:

```
nctrun@ord3p:~/dev/cs$ mcs holamundo.cs
nctrun@ord3p:~/dev/cs$
```


Introducción al cursillo
Introducción a . . .
Introducción a C#
Introducción breve . . .
Algunas novedades . . .
Referencias

- Y luego lo ejecutamos:

```
nctrun@ord3p:~/dev/cs$ mono holamundo.exe
hola mundo
nctrun@ord3p:~/dev/cs$
```



Introducción al cursillo
Introducción a ...
Introducción a C#
Introducción breve ...
Algunas novedades ...
Referencias

Página www

Página de Abertura

Página 18 de 75

Regresar

Full Screen

Cerrar

Abandonar

3.3. Fundamentos de C#

3.3.1. Tipos de datos

- En C# hay dos tipos de tipos de datos:
  - por valor
    - Cada vez que son utilizados, se copia su área de memoria
    - Se alojan en la `stack`
    - Destruidos en cuanto se destruye el bloque en el que son declarados acaba.
  - por referencia
    - Cada vez que son utilizados, se copia un puntero a su posición en memoria
    - Se alojan en la `heap`
    - Destruidos cuando el `Garbage Collector` detecta que la última referencia al dato ha sido destruida
- C# cuenta con varios tipos de datos `built-in`:
  - `byte`,`char`, `bol`, `sbyte`, `short`, `ushort`, `int`, `uint`, `float`, `double`, `decimal`, `long`, `ulong`
  - Ejemplos:

```
bool bln = true;
byte byt1 = 22;
char ch1='x', ch2='\u0066';
decimal dec1 = 1.23M;
double dbl1=1.23, dbl2=1.23D;
short sh = 22;
int i = 22;
```

Introducción al cursillo
Introducción a . . .
Introducción a C#
Introducción breve . . .
Algunas novedades . . .
Referencias

```
long lng1 =22, lng2 =22L;
sbyte sb = 22;
float f=1.23F;
ushort us1=22;
uint ui1=22, ui2=22U;
ulong ul1 =22, ul2=22U, ul3=22L, ul4=2UL;
```

  – Todos estos tipos de datos son por valor

- `strings` *(T3_E3_Strings)*
  – Los `strings` son tipos de datos por referencia
  – Los caracteres escapados son los típicos de otros lenguajes de programación:
    `\',\",\\,\0,\a,\b,\f,\n,\r,\t,\v`
  – Para evitar caracteres escapados, basta con poner una `@` por delante

```
string s = "hola";
s = "hola\n";
s = @"C:\Documents and Settings\";
```

3.3.2. Variables

(T3_E4_Variables)

- Las variables se asignan como en `C++` o `Java`:

```
class Variables{
    static void Main(){
        int numero = 5;
        System.Console.WriteLine("El número es: {0}",numero);
    }
}
```


Introducción al cursillo
Introducción a ...
Introducción a C#
Introducción breve ...
Algunas novedades ...
Referencias

- Al igual que en Java, un dato debe estar definido antes de ser usado:

```
class Variables{
    static void Main(){
        int numero;
        System.Console.WriteLine("El número es: {0}",numero);
    }
}
```

el compilador suelta:

```
nctrun@ord3p:~/dev/cs$ mcs definitivo.cs
definitivo.cs(4) error CS0165: Use of unassigned local variable ‘numero’
Compilation failed: 1 error(s), 0 warnings
```

3.3.3. Constantes

- Las constantes son declaradas al estilo `C++`:

```
        const constante = 50;
        const constante2 = constante - 2;
```

3.3.4. Nullable types

- Desde C# 2, tenemos los `nullable types`
- Tipos de datos por valor clásicos como ints, por ejemplo, pueden encapsularse en un tipo nullable:

```
int? variable4 = null;
//variable4 = 5;
```


Introducción al cursillo
Introducción a . . .
Introducción a C#
Introducción breve . . .
Algunas novedades . . .
Referencias

Página www
Página de Abertura
Página 21 de 75
Regresar
Full Screen
Cerrar
Abandonar

```
Console.WriteLine(variable4);
if(variable4 != null){
    Console.WriteLine(variable4.Value);
}else
    Console.WriteLine("variable4 es null");
```

3.3.5. Enumerados

- Los enumerados de C++ se han adaptado para que sean orientados a objetos *(T3_E5_Enumerados)*

```
class Enumerados{
   enum DiaSemana{
      Lunes,
      Martes,
      Miercoles,
      Jueves,
      Viernes,
      Sabado,
      Domingo
   };
   static void Main(){
      DiaSemana ds = DiaSemana.Lunes; //yeah!
   }
}
```

3.3.6. Control de flujo

- Condicionales *(T3_E6_ControlFlujo)*
  - if...else

```
int i = 15;
```


Introducción al cursillo
Introducción a . . .
Introducción a C#
Introducción breve . . .
Algunas novedades . . .
Referencias

```
if(i == 5)
   System.Console.WriteLine("un cinco...");
else if(i == 15)
   System.Console.WriteLine("es quince");
else
   System.Console.WriteLine("mariposa");
```

– switch

```
switch(queToca){
   case ViernesToca.GhostKino:
       Console.WriteLine("ghostkino y...");
     goto case ViernesToca.Fiestas;
   case ViernesToca.Cena:
       Console.WriteLine("cena");
     break;
   case ViernesToca.Fiestas:
       Console.WriteLine("fiestas");
     break;
}
```

– En línea, al estilo C++ y Java:

```
string ropa = haceFrio?"abrigo":"camiseta";
```

- Sentencias repetitivas

  – do...while

```
int i = 6;
do{
   Console.WriteLine("i: {0}",--i);
```

```
while(i > 0);
Console.WriteLine(":-S");
```

– while

```
while(respuesta.Incorrecta())
   respuesta = Preguntar();
```

– for

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

– foreach

```
string [] nombres = new string[]{"nombre1","nombre2"};
foreach(string n in nombres)
   Console.WriteLine(n);
```

– continue y break como en el resto de lenguajes

```
int n = 10;
while(n-- > 0){
    if(n % 2 == 0)
        continue; //No hagas nada con los pares
    int n2 = n;
    Console.WriteLine("n2 es: {0}",n2);
}
```

3.3.7. Directivas de compilador

- En C#, al igual que en C++, hay directivas de compilador: *(T3_E7_DirectivasCompilador)*

```
#define DEBUG

#if DEBUG
    //código
#elif TEST
    //otro código
#else
    //otro
#endif

#undef DEBUG

#if DEBUG
    //va a ser que no
#endif

#region Mis cosillas
    //movidillas. Los IDEs permiten agrupar varios métodos,
    //por ejemplo, para no verlos
#endregion

//En C# 2:
#pragma warning disable 0168
int dato; //el warning 0168 (variable no utilizada) no salta.
```

3.4. Clases

3.4.1. Definiendo clases

- Una clase es definida tal que: *(T3_E8_Clases)*

```
public class MiClase{
    //por defecto: datos privados
    int atributo1;
    int atributo2;

    public MiClase(){ //constructor
        atributo1 = atributo2 = 3;
    }

    void Metodo(){ //por defecto: privado
        //codigo
    }
    public int Atributo1{//property
        get{
            return atributo1;
        }
    }
}
```

- C# tiene mejor trabajados los modificadores de acceso que sus lenguajes predecesores *(T3_E9_Libreria)* y *(T3_E10_UsandoLibreria)*:
  - `public`. Todo el mundo puede acceder
  - `private`. Sólo la propia clase puede acceder.

Introducción al cursillo
Introducción a . . .
Introducción a C#
Introducción breve . . .
Algunas novedades . . .
Referencias

- `protected`. Sólo la propia clase y derivadas pueden acceder.
- `internal`. Sólo accesible desde las clases que están en el mismo ensamblado.
- `protected internal`. Sólo accesible desde las clases del mismo ensamblado **o** desde las hijas.

- También contamos con un constructor estático similar al de Java

3.4.2. Construyendo objetos

- Por defecto, C# provee un constructor por defecto

  simplemente pone a cero los datos numéricos, a `false` los `booleanos`, a `'\0'` los `char`, a 0 los `enum` y a `null` las referencias

- Para crear una instancia, basta con:

```
MiClase mc = new MiClase(parametros);
```

- Al igual que en `Java`, C# tiene inicializadores:

```
private int num = 15;
```

- A diferencia de `C++`, C# no provee de ningún *constructor copy*
  - Hay que implementarlo en caso de que se desee
  - Existe el interfaz System.ICloneable

```
nctrun@ord3p:~$ /opt/mono-1.1.7/bin/monop System.ICloneable
public interface ICloneable {

   object Clone ();
}
```

- La palabra reservada `this` tiene el mismo valor que en sus predecesores
- C# introduce la palabra reservada `readonly`, que hace que un atributo pueda ser unicamente modificado en el constructor de la clase

3.4.3. Clases estáticas

- Desde C# 2, tenemos clases estáticas *(T3_E8_Clases)*
- Una vez declaradas como estáticas, el compilador nos asegura:
  - Nadie puede heredar de ella
  - Todos sus miembros son estáticos

3.4.4. Clases parciales

- Desde C# 2, tenemos clases parciales *(T3_E8_Clases)*
- Podemos definirlas a lo largo del código, incluso en diferentes ficheros

3.4.5. Métodos

- La definición de métodos en C# es *(T3_E11_MetodosYPropiedades)*:

```
modificadorAcceso (static) tipoDatoDevuelto nombreFuncion(parametros){
}
```

- Los modificadores de acceso ya hemos comentado, y el uso y definición de `static` y retorno es igual a sus predecesores
- Sin embargo, el paso de parámetros cambia bastante

Introducción al cursillo
Introducción a . . .
Introducción a C#
Introducción breve . . .
Algunas novedades . . .
Referencias

- Por defecto, el paso de parámetros es, al igual que `Java`, siempre por valor:

```
void Metodo(MiClase mc){
    mc.Modificar(); //sin problemas
    mc = null; //sin problemas, pero no afecta fuera de la función
}
//...
variable.Metodo(mc);
```

- Sin embargo, C# permite el paso por referencia, pero de manera diferente a `C++`:
  * Obliga a poner la palabra `ref` antes del parámetro en la definición del método:

    ```
    void Metodo(ref MiClase mc){
        mc = null; //afecta fuera también
    }
    ```
  * Obliga a poner la misma palabra `ref` antes del parámetro en la llamada al método:

    ```
    variable.Metodo(ref mc);
    ```
  * Como con el paso por valor, para llamarlo, la variable debe estar inicializada:

    ```
    int numero;
    Metodo(ref numero); //no compila
    ```
- Para permitir que una variable sea inicializada dentro de un método, se utiliza la palabra `out`:
  * Obliga a poner la palabra `out` antes del parámetro en la definición del método.

    ```
    void Metodo(out int dato){
        dato = 0;
    }
    ```
  * También obliga a poner la misma palabra `out` antes del parámetro en la llamada al método:

    ```
    int numero;
    Metodo(out numero);
    ```


Introducción al cursillo
Introducción a . . .
Introducción a C#
Introducción breve . . .
Algunas novedades . . .
Referencias

- * A la hora de pasar la variable por parámetro, da lo mismo si está o no inicializada
  * A la hora de salir del método, el compilador da por hecho que puede no estar inicializada y que debe ser inicializada dentro, ateniéndose a las mismas restricciones que si no estuviese inicializada desde el principio

- Además, C# permite el paso de una lista indefinida de parámetros con la palabra `params`:

```
void Metodo(params int [] numeros){
    foreach(int n in numeros)
        Console.WriteLine(n);
}
//...
Metodo();
Metodo(1);
Metodo(1,2);
Metodo(1,2,3);
```

3.4.6. Propiedades

- C# incorpora el uso de propiedades *(T3_E11_MetodosYPropiedades)*:
  - Una *property* es un método que ejecuta un código en una clase casi como otro cualquiera, pero que desde fuera es accedido como si fuese un atributo público
  - Evita el continuo uso de *getAlgo*s y *setAlgo*s, haciendo más legible el código
- Pueden ser declarados de sólo lectura:

```
public int Numero{
    get{
        Console.WriteLine("Devolviendo...{0}",numero);
```

```
        return numero;
    }
}

//...

int num = variable.Numero; //guay
variable.Numero = 5; //no compila
```

- o bien de sólo escritura:

```
public int Numero{
    set{
        Console.WriteLine("Asignando...{0}",value);
        numero = value; //value es el valor pasado por parámetro
    }
}

//...

int num = variable.Numero; //no compila
variable.Numero = 5; //guay
```

- o bien de lectura y escritura:

```
public int Numero{
    set{
        Console.WriteLine("Asignando...{0}",value);
        numero = value; //value es el valor pasado por parámetro
```

```
        }
        get{
            Console.WriteLine("Devolviendo...{0}",numero);
            return numero;
        }
    }

    int num = variable.Numero; //guay
    variable.Numero = 5; //guay
```

- En C# 2, podemos además dar diferentes permisos al get y al set:

```
public int Numero{
    get{
        return numero;
    }
    protected set{
        numero = value;
    }
}
```

3.4.7. Destructores

- En la clase `Object` .NET, de la que *todo* hereda, existe el método virtual `Finalize`, que es similar al `finalize` de `Java`, salvo que .NET te garantiza que va a ser llamado *(T3_E12_Destructores)*.
- El método que a nivel de CIL se sobrecarga es:

```
protected override void Finalize(){
    Console.WriteLine("hola");
```

```
}
```

Sin embargo, en su lugar, se debe escribir al estilo C++:

```
~MiClase(){
    Console.WriteLine("hola");
}
```

que el compilador de C# se encargará de transformar en Finalize.

- En caso de que no se deba esperar hasta que el Garbage Collector llame al destructor, se implementa el método Dispose:
  - Se implementa el interfaz System.IDisposable
  - Se puede bien controlar con bools que no se repita la llamada al método Dispose, para evitar que lo vuelva a llamar el destructor, o bien se puede evitar que el Garbage Collector llame al método Finalize con:

    ```
    System.GC.SuppressFinalize(instancia);
    ```

  - El uso de Dispose permite una sintaxis más sencilla y óptima con un nuevo uso de la palabra using:

    ```
    //MiClase implementa System.IDisposable
    using(MiClase mc = new MiClase(parametros)){
        mc.hacerAlgo();
    }//llegado aquí, se llama al Dispose y la variable deja de existir
    ```

3.5. Estructuras

3.5.1. Qué son

- Los `structs` son *(T3_E13_Estructuras)*:
  - agrupamientos lógicos de datos y métodos (al igual que las clases)
  - no permiten herencia, destructores, constructores sin parámetros, inicialización de campos...
  - sí permiten implementación de interfaces
  - Son datos por valor

3.5.2. Definición

- La definición de un `struct` es tal que:

```
public struct Punto{
    public int X;
    public int Y;
    public Punto(int x, int y){
        X = x;
        Y = y;
    }
}
```

3.5.3. Creación

- La creación de `structs` es también similar a la de instancias de clases:

```
Punto p = new Punto(0,5);
p.X = 5;
```

Introducción al cursillo
Introducción a ...
Introducción a C#
Introducción breve ...
Algunas novedades ...
Referencias

el que se use la palabra ***new*** *no significa que sea por referencia*

- Por defecto, hay un constructor sin parámetros que inicializa todo. De hecho, podría crearse un struct de esta manera:

```
Punto p;
p.X = 5;
```

No se aconseja este código (mejor crear siempre con `new`, deja el código más claro)

3.5.4. Arrays en structs

- No podemos declarar un array dentro del struct sin saber el tamaño del mismo
- Como no hay inicializadores de campos en structs, tampoco podemos definir el tamaño del array
- Solución: desde C# 2 existen arrays de tamaño fijo:

```
struct A{
  char datos [250];
}
```


Introducción al cursillo
Introducción a . . .
Introducción a C#
Introducción breve . . .
Algunas novedades . . .
Referencias

3.6. Herencia y polimorfismo

3.6.1. Heredando

- Al igual que Java, C# no soporta herencia múltiple
- La sintaxis similar a C++, pero sin modificadores de acceso en la herencia, al estilo Java: *(T3_E14_Herencia)*

```
public class Mamifero : Animal
```

- Si la clase padre tiene un constructor sin parámetros, por defecto será llamado antes de ejecutar el código del constructor de la clase hija
- Si no, será necesario llamar a base:

```
public class Mamifero : Animal{
    public Animal(string nombre) : base(nombre){
        //...
    }
}
```

- Para redefinir un método, hace falta la palabra new:

```
class A{
    public void Metodo(){
        Console.WriteLine("A");
    }
}
class B : A{
```

```
    public new void Metodo(){
        Console.WriteLine("B");
    }
}
class Test{
    public static void Main(){
        B b = new B();
        b.Metodo(); //mostrará B
        A a = b;
        a.Metodo(); //mostrará A
    }
}
```

- Para evitar que alguien pueda heredar de una clase determinada se utiliza la palabra `sealed` (similar al `final` de `Java`):

```
sealed class A{
//...
}
class B : A{ //no compila
}
```

3.6.2. Polimorfismo

- Para escribir métodos polimórficos hace falta las palabras `virtual` y `override`, al estilo `C++`:

```
class A{
    public virtual void Metodo(){
        Console.WriteLine("A");
```

```
    }
}
class B : A{
    public override void Metodo(){
        Console.WriteLine("B");
    }
}
class Test{
    public static void Main(){
        B b = new B();
        b.Metodo(); //mostrará B
        A a = b;
        a.Metodo(); //mostrará B también :)
        A [] varios = new A[2];
        varios[0] = new B();
        varios[1] = new A();
        foreach(A nueva in varios)
            nueva.Metodo();
    }
}
```

3.6.3. Clases abstractas

- Una clase abstracta:
  - no puede ser instanciada
  - puede declarar una serie de métodos que obliga a toda hija clase no abstracta a definir
- Definiendo clases abstractas *(T3_E15_ClasesAbstractas)*:

Introducción al cursillo
Introducción a ...
Introducción a C#
Introducción breve ...
Algunas novedades ...
Referencias

- Se definen con la palabra reservada `abstract`:

```
abstract public class A{
    protected int dato;
}
abstract public class B : A{
    abstract public void Metodo();
}
public class C : B{
    public override void Metodo(){
        Console.WriteLine();
    }
}
```

3.6.4. Relacionando tipos de datos por valor y por referencia

- **Todo** puede ser un `object` *(T3_E16_BoxingUnboxing)*:
  - Incluso datos por valor

    ```
    string s = 5.ToString();
    ```

- A este proceso se le llama *boxing* y al opuesto (pasar de un `object` a `int` *unboxing*.
- El proceso de *unboxing* debe ser lógicamente explícito:

```
object o = 5;
int i = (int)o;
```

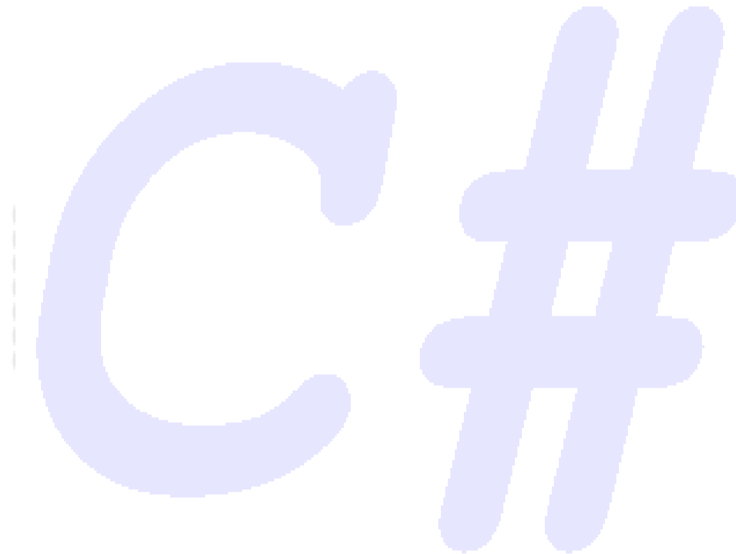
3.6.5. Sobrecarga de operadores

- Al igual que en `C++`, C# puede sobrecargar operadores *(T3_E17_SobrecargaOperadores)*

Introducción al cursillo
Introducción a ...
Introducción a C#
Introducción breve ...
Algunas novedades ...
Referencias

- Sobrecargando un operador:

```
class A : B{
    public static A operator+(A a, A b){
        return a.X + b.X;
    }
}
```



Introducción al cursillo
Introducción a . . .
Introducción a C#
Introducción breve . . .
Algunas novedades . . .
Referencias

3.7. Interfaces

3.7.1. Qué son

- Como hemos comentado antes, C#, al igual que `Java`, no tiene herencia múltiple
- Sin embargo, una clase sí puede *implementar* varios interfaces.
- El interfaz define varios métodos, que la clase que lo implemente deberá declarar como públicos
- Al igual que en `Java`, un interfaz puede heredar de otro

3.7.2. Definiendo interfaces

- Se definen tal que *(T3_E18_Interfaces)*:

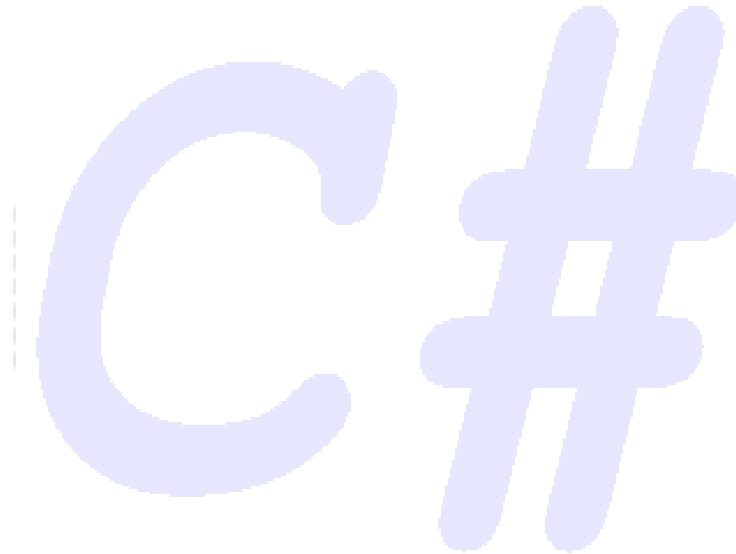
```
interface IDisposable{
    void Dispose();
}
interface IDestroyable : IDisposable{
    void Destroy();
    int Status{
        get;
    }
}
```

3.7.3. Is, as

- Para saber si una instancia implementa un interfaz o hereda de otra clase, se puede utilizar *(T3_E18_Interfaces)*:
  - `is`

    * Devuelve un `bool` informando de si es o no

- `as`
    * Devuelve `null` si **no** lo implementa, o la instancia como otra cosa en caso de que sea



Introducción al cursillo
Introducción a ...
Introducción a C#
Introducción breve ...
Algunas novedades ...
Referencias

3.8. Manejo de excepciones

3.8.1. Capturando excepciones

- Para capturar excepciones se utiliza `catch` *(T3_E19_Excepciones)*:

```
try{
    int i = 10/0;
}catch(System.DivideByZeroException){
    Console.WriteLine("capturada");
}
```

- Si queremos hacer algo con la excepción, podemos capturar la instancia de la siguiente manera:

```
try{
    int i = 10/0;
}catch(System.DivideByZeroException dbze){
    Console.WriteLine(dbze.StackTrace)
}
```

- Si en cambio queremos capturar todas las excepciones, podemos hacer:

```
try{
    int i = 10/0;
}catch{
    Console.WriteLine("capturada");
}
```

- También podemos utilizar `finally` para que, tanto si se eleva una excepción como si no, se ejecute un código:

Introducción al cursillo
Introducción a . . .
Introducción a C#
Introducción breve . . .
Algunas novedades . . .
Referencias

```
try{
    int i = 10/0;
}catch(System.DivideByZeroException dbze){
    Console.WriteLine("y aquí me he quedado :-)");
}finally{
    Console.WriteLine("Esto SIEMPRE se ejecuta");
}
```

3.8.2. Lanzando excepciones

- Para lanzar una excepción, se utiliza la palabra reservada `throw`:

```
throw new Exception("excepción");
```

- Lo lanzado debe ser una instancia de una clase hija de `System.Exception` (o de `System.Exception`)

  Se recomienda que las excepciones propias sean hijas de `System.ApplicationException` para diferenciar entre las excepciones del `Framework` y las de nuestra aplicación

3.9. Delegates

3.9.1. Qué son

- Son el equivalente a puntero a función de otros lenguajes
  - adaptados al modelo orientado a objetos
  - fáciles de usar

3.9.2. Definiendo delegates

- Un `delegate` se indica tal que *(T3_E20_Delegates)*:

```
delegate tipoDato funcion(parametros);
```

- Ejemplo:

```
delegate void Funcion(object o);

class A{
    static void Main(){}
}
```

3.9.3. Utilizando delegates

- El uso es tan simple como:

```
delegate void Funcion();

class A{
```

```
    void Hacer(Funcion f){
        f();
    }

    void HazAlgo(){
        Hacer(new Funcion(Saludar));
    }

    void Saludar(){
        System.Console.WriteLine("hola");
    }

    static void Main(){
        (new A()).HazAlgo();
    }
}
```

3.9.4. Utilizando multicasting

- Para que al llamar a un delegado, se llamen a varios métodos, sólo hay que ir añadiendo nuevos métodos con `+=` *(T3_E21_DelegateMulticast)*:

```
delegate void Funcion();
class A{
    void Hacer(Funcion f){
        f();
    }

    void HazAlgo(){
        Hacer(new Funcion(Saludar));
```

```
        Funcion f = new Funcion(Saludar);
        f += new Funcion(Saludar);
        f += new Funcion(Saludar);
        f();
    }

    void Saludar(){
        System.Console.WriteLine("hola");
    }

    static void Main(){
        (new A()).HazAlgo();
    }
}
```

3.9.5. Eventos

- El `delegate` es útil, pero hay ciertas situaciones en las que no nos sirve. . . .
  - Si quisiéramos tener un interfaz sobre el cual se pueden dar eventos. . . no se puede declarar en un interfaz
    - ¿`addListener` de Java?
  - Si quisiéramos ponerlo público para que otras clases añadiesen funciones a nuestro evento. . . se podrá llamar al evento desde otras clases (en lugar de siempre llamarlo tú)
- Para solucionar estos problemas, está la palabra reservada `event`
- Ver ejemplos: *(T3_E22_Eventos)*

Introducción al cursillo
Introducción a . . .
Introducción a C#
Introducción breve . . .
Algunas novedades . . .
Referencias

3.10. Atributos

3.10.1. Qué son

- Una forma de dar más información acerca de una clase, enumerado, atributo, método...
- Son útiles por ejemplo para informar a librerías qué partes de nuestro código son qué
- Ejemplos:
  - Informar de qué clase es el Servicio Web, y qué métodos son WebMethods
  - Informar de qué campos son Widgets en Gtk
  - Informar qué clases son pruebas unitarias y cuáles de sus métodos son tests, y qué tipo de tests en nunit

3.10.2. ¿Cómo se utilizan?

- Ver *(T3_E23_Atributos)*
- `[Atributo(parametros_constructor,Propiedad1=valorProp1,Propiedad2=valorProp2...)]`
- Ejemplo:

```
//Los utilizamos así. Puntuaciones es la property, no es obligatoria
[Mola("mazo",Puntuacion=10)]
class UnaClase{

    [Mola("lo típico")]
    int dato;
    int otroDato; //Sin Mola
```


Introducción al cursillo
Introducción a ...
Introducción a C#
Introducción breve ...
Algunas novedades ...
Referencias

```
    [Mola("no mucho",Puntuacion=6,Docu="Es un método cualquiera")]
    public int Metodo(){
        return 5;
    }
}
```

3.10.3. ¿Cómo se definen?

- Tienen que ser clases hijas de System.Attribute
- Se les puede definir más (”Este atributo sólo sirve para structs” / ”Este sirve para tanto campos como métodos”) por medio de atributos :-)
- Luego se comprueba si son atributos por medio de la API de Reflection
- Ver *(T3_E23_Atributos)*

Introducción al cursillo
Introducción a . . .
Introducción a C#
Introducción breve . . .
Algunas novedades . . .
Referencias

3.11. Generics

3.11.1. ¿Qué son?

- Similares a las `templates` de C++. En C# están desde su versión 2
- Permiten hacer clases o métodos que procesen un tipo de dato único (o heredado de él), desconocido por el autor de esta clase.
- Hasta ahora, como todo hereda de `object`, los contenedores de datos se hacían en función de `object`:

```
using System.Collections;
//...
ArrayList l = new ArrayList();
l.Add(5); //Un int cuela
l.Add("hola"); //Un string también
```

- Ahora, podemos hacer:

```
using System.Collections.Generic;
//...
List<int> l = new List<int>();
l.Add(5);
l.Add("hola"); //no compila
```

- El espacio de nombres System.Collections sigue teniendo los contenedores de C# 1, los nuevos contenedores están en System.Collections

Introducción al cursillo
Introducción a . . .
Introducción a C#
Introducción breve . . .
Algunas novedades . . .
Referencias

3.11.2. ¿Cómo se implementan?

- Ver ejemplos en *(T3_E24_Generics)*

```
//Este contenedor sólo depende de T
class MiPropioContenedor < T >{
    T [] datos;

    public MiPropioContenedor(T [] datos){
        this.datos = datos;
    }

    public T [] Datos{
        get{
            return datos;
        }
        set{
            datos = value;
        }
    }
}

//Si queremos que T implemente el interfaz IComparable:
class MiListaOrdenada < T > where T : IComparable
{
    //...
}

//Sólo para una función
class A{
```


Introducción al cursillo
Introducción a . . .
Introducción a C#
Introducción breve . . .
Algunas novedades . . .
Referencias

```
    public static T Metodo<T>(T dato){
        return dato;
    }
}
```

3.12. Iteradores

3.12.1. En C# 1

- Como en muchos otros lenguajes, en C# tenemos iteradores
- Los iteradores implementan el interfaz `System.Collections.IEnumerator`
- Los podemos utilizar en un `foreach`:

```
ArrayList al = new ArrayList();
al.Add("hola");
al.Add("adios");
foreach(string s in al)
    Console.WriteLine(s);
//Internamente, está haciendo:
IEnumerator ie = al.GetEnumerator();
while(ie.MoveNext())
    Console.WriteLine(ie.Current);
```

- El problema es que para utilizar los nuestros propios, lo tenemos que implementar a mano
- Ver ProveedorComandos en *(T3_E25_Iteradores)*

3.12.2. En C# 2

- En C# 2, con Generics, tenemos `System.Collections.Generic.IEnumerator<loquesea>`
- Además, contamos con generación automática de iteradores con la palabra reservada `yield`:

```
class ProveedorComandos2{

    readonly string [] comandos;

    public ProveedorComandos2(params string [] comandos){
        this.comandos = new string[comandos.Length];
        for(int i = 0 ; i < comandos.Length ; ++i )
            this.comandos[i] = comandos[i];
    }

    public System.Collections.Generic.IEnumerator <string> GetEnumerator(){
        foreach(string s in comandos)
            //cada vez que tenemos un comando, lo devolvemos con yield
            yield return s;
    }
}
//...
foreach(string s in new ProveedorComandos2("a","b","c")){
    Console.WriteLine(s);
}
```

- Ver ProveedorComandos2 y sobre todo ProveedorComandos3 en *(T3_E25_Iteradores)*

Introducción al cursillo
Introducción a ...
Introducción a C#
Introducción breve ...
Algunas novedades ...
Referencias

3.13. Código unsafe

3.13.1. Qué es

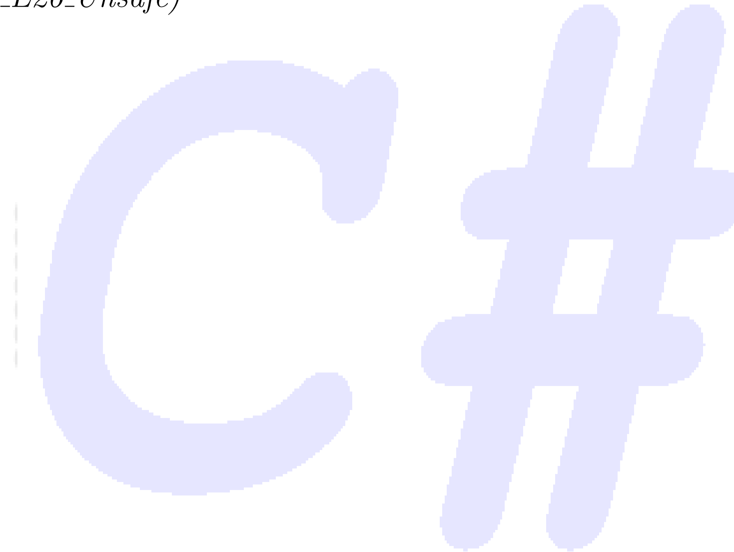
- En C#, como dijimos al comienzo del cursillo, disponemos de dos modos:
  - Código `managed`
    - El código que hemos utilizado durante todo el cursillo
    - Lo gestiona el runtime, temas de recolección de basura, acceso a memoria (no tendrás segfaults, sino excepciones de alto nivel), etc.
    - Programación de más alto nivel, facilidades para el programador
  - Código `unsafe`
    - Código no gestionado
    - Debemos indicarlo al compilador (-unsafe en mcs, activar la opción correspondiente en el proyecto de MonoDevelop)
    - Debemos indicarlo en el alcance del código (en la clase, método. . . correspondiente), con la palabra reservada `unsafe`
    - Podemos utilizar punteros, al estilo C / C++, sin recolección de basura
    - Podemos hacer accesos a memoria al estilo C / C++
    - Programación de más bajo nivel

3.13.2. ¿Y para qué me sirve?

- Interacción con código nativo
  - No necesitas `JNI` o `SWIG` como ”pegamento” entre C# y C
  - Utilizando recursos de bajo nivel en momentos concretos

Introducción al cursillo
Introducción a . . .
Introducción a C#
Introducción breve . . .
Algunas novedades . . .
Referencias

- Creación de bindings de librerías existentes (bien de terceros, GTK, SDL, OpenGL. . . , como nuestras) para poder utilizarlas desde .NET
- Optimización de partes de nuestra aplicación (desarrollando en C / C++ partes críticas y luego utilizando esto desde C#)
- Ejemplo: *(T3_E26_Unsafe)*



4. Introducción breve a la FCL

4.1. Qué es la FCL?

- La *Framework Class Library* es el conjunto de librerías que vienen con el .NET Framework
  - En Mono vienen por defecto también otra serie de librerías, algunas de las cuales se verán en el cursillo de Introducción a Mono
- La mayoría de tipos de datos cuelgan del espacio de nombres `System`
- Un punto de comienzo para buscar algo podría : ser este (este para .NET 2)
- Ejecutando `monodoc` también podemos acceder a la documentación
- En la web del DotNetGroup hay una sección `Documentación` en la que hay documentación de talleres que ha hubo el curso pasado en la que explica el uso de la FCL para comunicaciones con sockets, multihilo, etc. etc.
- Aquí sólo vamos a dar una pequeña parte y de pasada

Introducción al cursillo
Introducción a . . .
Introducción a C#
Introducción breve . . .
Algunas novedades . . .
Referencias

4.2. Clases básicas

- Echando un ojo al espacio de nombres `System`, podemos ver clases que ya conocemos realmente como:
  - `Console`: A la que ya estamos acostumbrados :-)
  - `Int32`, `Byte`, `String`: cuando en C# escribimos `int`, `byte` o `string`, internamente en CLI se están utilizando estos tipos de datos
  - `Array`: Que es lo que hay cuando utilizamos arrays
- También podemos ver la documentación de las excepciones más frecuentes:

  `ArithmeticException`, `DivideByZeroException`...
- Además podemos ver los espacios de nombres que hay definidos dentro de `System`, como por ejemplo:
  - `System.Collections`: colecciones
  - `System.IO`: manejo de entrada-salida
  - `System.Threading`: manejo de hilos
  - `System.Net`: redes
  - `System.Reflection`: reflection
  - `System.GC`: manejo del Garbage Collector
  - `System.XML`: procesamiento de XML
- y muchos muchos más

Introducción al cursillo
Introducción a ...
Introducción a C#
Introducción breve ...
Algunas novedades ...
Referencias

4.3. Colecciones

- En `System.Collections` hay una serie de clases e interfaces para usar pilas, colas, mapas...:
- Esto es, vienen definidas una serie de clases listas para ser utilizadas, como por ejemplo: *(T4_E01_Colecciones)*
  - `ArrayList`: Estilo `vector` o `Vector`. Permite el ir añadiendo, eliminando y modificando elementos a la lista
  - `Hashtable`: Array asociativo. Permite añadir variables indexadas por otras variables
  - `Stack`, `Queue`: pilas y colas
  - `SortedList`: Listas ordenadas
- Así como una serie de interfaces que implementan estas clases, sus valores, etc.

  `ICollection`, `IList`, `IDictionary`, `IComparer`...

```
using System;
using System.Collections;

class A{
    public static void Main(){
        ArrayList al = new ArrayList();
        string s;
        do{
            Console.Write("Dame un algo (salir para salir): ");
            s = Console.ReadLine();
            al.Add(s);
        }while(s.ToLower() != "salir");
        string [] todasLasFrases = (string[])al.ToArray(typeof(string));
```


Introducción al cursillo
Introducción a ...
Introducción a C#
Introducción breve ...
Algunas novedades ...
Referencias

```
            foreach(string frase in todasLasFrases)
                Console.WriteLine(frase);
        }
    }
```

- En C# 2, encontramos nuevos tipos de datos que utilizan `generics` en el espacio de nombres `System.Collections.Generic` *(T4_E02_Colecciones)*:
  - `LinkedList`: Lista doblemente enlazada
  - `List`: Versión de `ArrayList` adaptada a `generics`
  - `Dictionary`: Versión de `Hashtable` adaptada a `generics`
  - `Stack`, `Queue`, `SortedList`... adaptaciones a `generics` de sus versiones en `System.Collections`

4.4. Ficheros

- Las clases relacionadas con manejo de ficheros y directorios cuelgan de `System.IO`
- Ficheros de texto:
  - Fichero de texto lectura: `new StreamReader(nombre_fichero)`,
    o `File.OpenRead(nombre_fichero)`, que devuelve un `StreamReader`
  - Fichero de texto escritura: `new StreamWriter(nombre_fichero)`
    o `File.CreateText(nombre_fichero)`, que devuelve un `StreamWriter`
  - Console.Out y Console.Error son dos `TextWriters` (y Console.In un `TextReader`). `StreamWriter` es hija de `TextWriter`
- Ficheros y directorios:
  - Manejo básico de ficheros (copiar, mover, eliminar. . .) → `File.Move`, `File.Copy`. . .
  - Manejo básico de directorios (copiar, mover, eliminar. . .) → `Directory.Move`, `Directory.Copy`, `Directory.Delete`. . .
  - ¿Separador de directorios?
    en Windows. . . / en UNIX. . . → `Path.DirectorySeparatorChar` en todos :-)
- Ver *(T4_E03_Ficheros)*

4.5. Hilos

4.6. Introducción

- Las clases relacionadas con manejos de hilos cuelgan de `System.Threading`
- Un hilo es un `System.Threading.Thread`
  - Lanzando un hilo:

```
public void Funcion(){
   //...
}
//...
Thread t = new Thread(new ThreadStart(Funcion));
t.Start();
```

  - `System.Threading.ThreadStart` es un delegate declarado tal que:

```
public delegate void ThreadStart ();
```

  - Utilizando delegates anónimos:

```
Thread t = new Thread(delegate{
         Console.WriteLine("hola mundo desde otro hilo");
   });
t.Start();
```

  - Esperando a que terminen:

```
t.Join();
```

- Ver *(T4_E04_Hilos)*

4.7. Protección básica

- La protección más básica para gestionar hilos en C# es la palabra `lock`:

```
lock(instancia){
    //Código
}
```

- Lo que garantiza `lock` es que cuando un hilo entra en un `lock` de una instancia, cualquier otro hilo se queda bloqueado al llegar a un lock de esta instancia.
- Ver *(T4_E05_Hilos)*

4.8. Sincronización básica

- A través de monitores podemos obtener una sincronización básica:
- Protegemos el código:

```
Monitor.Enter(instancia);
    //código
Monitor.Exit(instancia);
```

- Ponemos dentro `Wait` para permitir a otros entrar en la región protegida y `Pulse` para avisar hilos que estén en un Wait

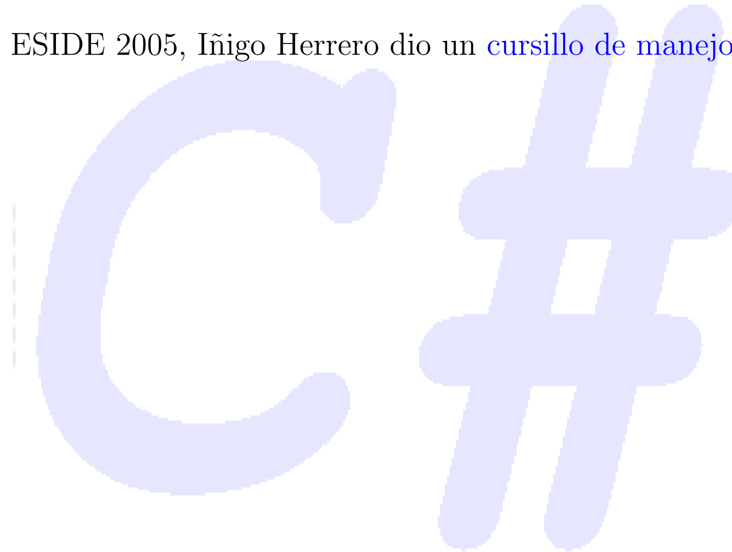
```
Monitor.Enter(instancia);
    //código
    Monitor.Wait(instancia); //Me quedo esperando a que alguien me diga algo
    //código
    Monitor.Pulse(instancia); //aviso a alguien de que ocurre algo
Monitor.Exit(instancia);
```


Introducción al cursillo
Introducción a . . .
Introducción a C#
Introducción breve . . .
Algunas novedades . . .
Referencias

- También están otras funciones, como `PulseAll` (avisar a todos) o `TryEnter` (intentar entrar, si está bloqueado hacer otro código).
- Ver *(T4_E06_Monitores)*

4.9. Otros

- En la Semana ESIDE 2005, Iñigo Herrero dio un cursillo de manejo de hilos en .net



5. Mirando al futuro: C# 3

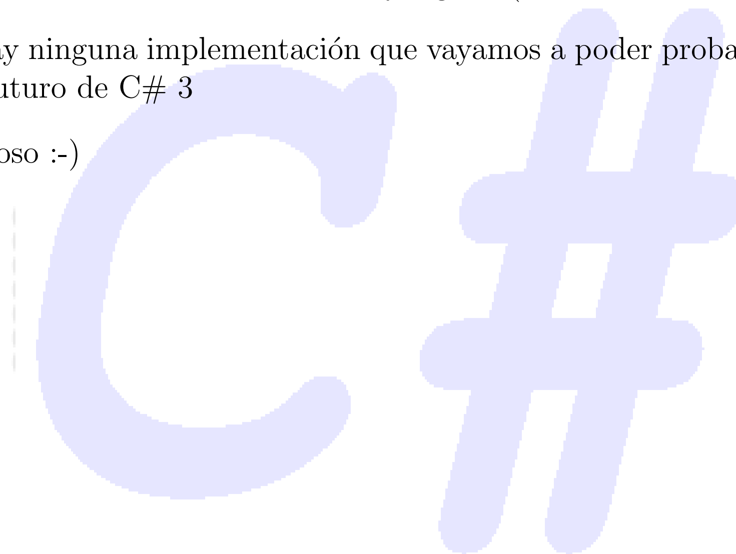
5.1. Introducción

- Ya está propuesta especificación de C# 3

  El documento es bastante interesante y legible (no es un tocho de más de 500 páginas)

- Todavía no hay ninguna implementación que vayamos a poder probar, pero podemos ver hacia dónde irá el futuro de C# 3

- Muuuuuy jugoso :-)



Introducción al cursillo
Introducción a . . .
Introducción a C#
Introducción breve . . .
Algunas novedades . . .
Referencias

5.2. var

- Cuando asignamos un valor a una variable, desde que en tiempo de compilación tenemos el retorno del valor, no es necesario definir el tipo de la variable
- En Boo (una mezcla interesante entre Python y C#), tenemos desde su comienzo:

```
a = 5 #a será 5
b = Funcion() #b será del mismo tipo que el retorno de Funcion
```

- En C# 3, se introduce la palabra reservada `var`, que realiza la misma función:

```
var a = 5; //a será int
var b = Funcion(); //b será del tipo de dato que devuelva Funcion
```

- También aplicable a arrays:

```
var a = new{ 1, 2, 3}; //array de int
```

5.3. Métodos de extensiones

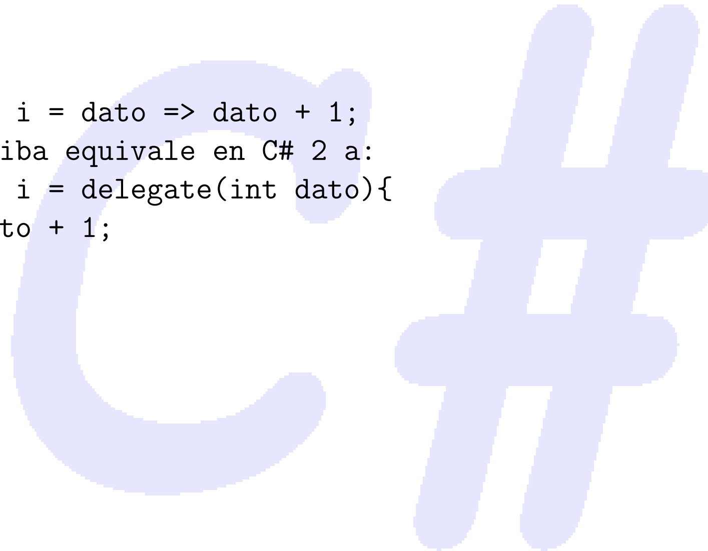
- Dada una clase, podemos extenderla añadiéndole nuevos métodos:

```
public static class MisUtilidades{
   //Con "this" por delante, indicamos el tipo de dato al que extendemos
   public static void Imprimir(this string s){
      Console.WriteLine(s);
   }
}

//...
using MisUtilidades;
//...
string s = "brutaaaal";
s.Imprimir(); //Y así con cualquier clase
```

Introducción al cursillo
Introducción a . . .
Introducción a C#
Introducción breve . . .
Algunas novedades . . .
Referencias

5.4. Expresiones lambda

- Las expresiones lambda, como en otros lenguajes, sirven para crear funciones anónimas en línea



```
delegate int Incrementador(int dato);

//...

Incrementador i = dato => dato + 1;
//esto de arriba equivale en C# 2 a:
Incrementador i = delegate(int dato){
    return dato + 1;
};
```


Introducción al cursillo
Introducción a . . .
Introducción a C#
Introducción breve . . .
Algunas novedades . . .
Referencias

5.5. Inicializadores de objetos y colecciones

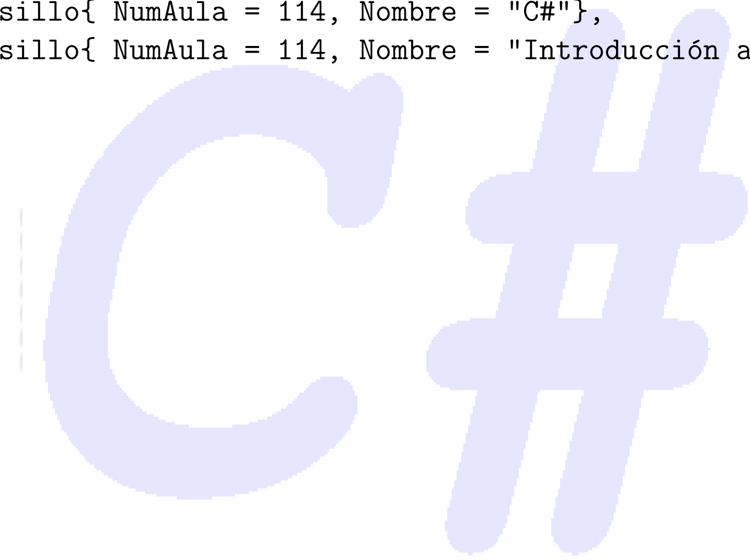
- Tenemos una nueva forma de inicializar objetos diréctamente pasándole propiedades que tenga:

```
class Cursillo{
    int numAula;
    string nombre;

    public Aula(){}

    public int NumAula{
        get{
            return numAula;
        }
        set{
            numAula = value;
        }
    }

    public string Nombre{
        get{
            return nombre;
        }
        set{
            nombre = value;
        }
    }
}

//Lo inicializamos con las propiedades en la misma línea
```

```
Cursillo cursillo = new Cursillo{NumAula = 114, Nombre = "C# rulez"};
```

- Además y junto a lo anterior podemos inicializar colecciones típicas de `System.Collections.Generic` como si fuesen arrays:

```
List<Cursillo> cursillos = new List<Cursillo>{
    new Cursillo{ NumAula = 114, Nombre = "C#"},
    new Cursillo{ NumAula = 114, Nombre = "Introducción a Mono"}
};
```



Introducción al cursillo
Introducción a . . .
Introducción a C#
Introducción breve . . .
Algunas novedades . . .
Referencias

5.6. Tipos anónimos

- Imagina que quieres una clase punto que no tienes para una función pequeña. Dos opciones:
- C# 2: definir la clase punto:

```
class Punto{
    int x, y;
    public int X { get{ return x; } set { x = value; } }
    public int Y { get{ return y; } set { y = value; } }
}
//...
Punto p = new Punto();
```

- C# 3 (también en Boo) definirlo de la misma con un tipo anónimo:

```
var miPunto = new { X = 5, Y = 6};
Console.WriteLine(miPunto.X); // :-D~~~~
```

- Muy interesante:

```
var misPuntos = new {
    new { X = 1, Y = 2},
    new { X = 3, Y = 4}
};
```

5.7. Expresiones de consulta

- Uno de los mayores cambios en el lenguaje.
- Se integra en el propio lenguaje un nuevo tipo de expresiones similar a SQL o XQuery para facilitar el procesamiento de datos
- Es bastante completo, mejor mirar la especificación
- Un ejemplo sería:

```
Cursillo [] cursillos = new Cursillo[23]; //por ejemplo

//...

var resultado = from c in cursillos
group c by c.Aula into a
select new { Aula = a.Key, NumCursillos = a.Count() };

foreach(var res in resultado)
    Console.WriteLine("En el aula {0} hay {1} cursillos",res.Aula,res.NumCursillos);
```

- La idea es que el código anterior se traduciría luego a operaciones tal que:

```
Cursillo [] cursillos = new Cursillo[23]; //por ejemplo

//...

var resultado = cursillos.GroupBy(c => c.Aula)
    .Select( a => new { Aula = a.Key, NumCursillos = a.Count() });
```

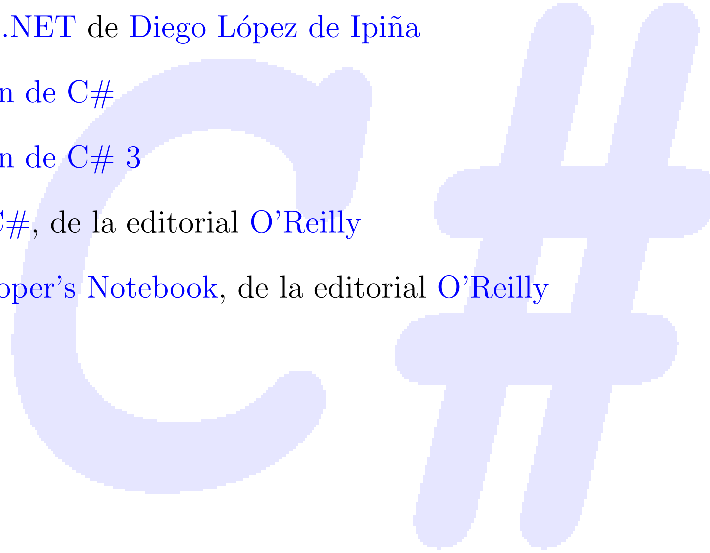
Introducción al cursillo
Introducción a . . .
Introducción a C#
Introducción breve . . .
Algunas novedades . . .
Referencias

```
foreach(var res in resultado)
    Console.WriteLine("En el aula {0} hay {1} cursillos",res.Aula,res.NumCursillos);
```

- Las expresiones de consulta luego son mucho más complejas, incluyendo `joins`, consultas de varios grupos de datos, etc., pudiendo utilizarse sobre diferentes grupos de datos (no sólo `arrays`, sino que podría aplicarse a BBDD, XML...).

Introducción al cursillo
Introducción a ...
Introducción a C#
Introducción breve ...
Algunas novedades ...
Referencias

6. Referencias



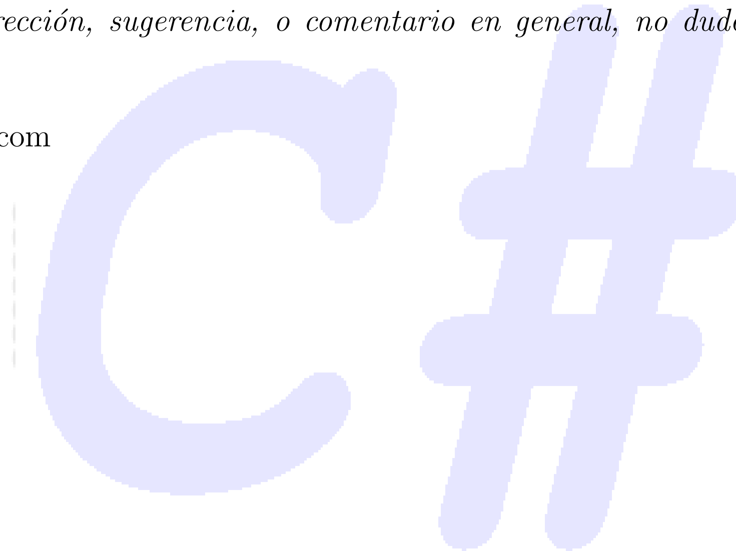
- Toda la documentación que hay en MSDN
- Más documentación que hay en el DotNetGroup de la Universidad de Deusto, incluyendo su documentación y su Biblioteca
- Los apuntes de .NET de Diego López de Ipiña
- La especificación de C#
- La especificación de C# 3
- Programming C#, de la editorial O'Reilly
- Mono: A Developer's Notebook, de la editorial O'Reilly

Introducción al cursillo
Introducción a . . .
Introducción a C#
Introducción breve . . .
Algunas novedades . . .
Referencias

Este documento está escrito por Pablo Orduña para el cursillo de Introducción a C# del grupo de .NET de la Universidad de Deusto, el DotNetGroup. Puede encontrarse junto con los ejemplos y las fuentes LaTeXen la misma web. Probablemente las actualizaciones, dado que todas las herramientas utilizadas en el cursillo son software libre, las cuelgue en mi hueco web en el grupo de software libre, el e-ghost.

Para cualquier corrección, sugerencia, o comentario en general, no dudes en ponerte en contacto conmigo en:

pablo @ ordunya . com



Introducción al cursillo
Introducción a . . .
Introducción a C#
Introducción breve . . .
Algunas novedades . . .
Referencias