

<http://elvex.ugr.es/decsai/csharp/language>

Introducción

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.

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**

Características de C#

- **Sencillez.** C# elimina muchos elementos que otros lenguajes incluyen y que son innecesarios en .NET. Por ejemplo: el código escrito en C# es autocontenido (no necesita de ficheros adicionales al propio fuente tales como ficheros de cabecera) y el tamaño de los tipos de datos básicos es fijo e independiente del compilador, sistema operativo o máquina para quienes se compile (no como en C++), lo que facilita la portabilidad del código.
- **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: tipo básico `decimal` (128 bits), instrucción `foreach`, tipo básico `string`, etc.
- **Orientación a objetos.** C# no admiten ni funciones ni variables globales sino que todo el código y datos han de definirse dentro de definiciones de tipos de datos, lo que reduce problemas por conflictos de nombres y facilita la legibilidad del código.

C# soporta todas las características propias del paradigma de programación orientada a objetos: *encapsulación*, *herencia* y *polimorfismo*.

- Encapsulación: además de `public`, `private` y `protected` C# añade el modificador `internal`.
- C# sólo admite herencia simple.
- Todos los métodos son por defecto sellados y que los redefinibles hayan de marcarse con el modificador `virtual`.
- **Orientación a componentes.** La sintaxis de C# permite definir cómodamente **propiedades** (similares a campos de acceso controlado), **eventos** (asociación controlada de funciones de respuesta a notificaciones) o **atributos** (información sobre un tipo o sus miembros).

- **Gestión automática de memoria.** Todo lenguaje de .NET tiene a su disposición el recolector de basura del CLR.
- **Seguridad de tipos.** # incluye mecanismos que permiten asegurar que los accesos a tipos de datos siempre se realicen correctamente:
 - Sólo se admiten conversiones entre tipos compatibles
 - No se pueden usar variables no inicializadas.
 - Se comprueba que los accesos a los elementos de una tabla se realice con índices que se encuentren dentro del rango de la misma.
 - C# incluye **delegados**, que son similares a los punteros a funciones de C++ pero siguen un enfoque orientado a objetos: pueden almacenar referencias a varios métodos simultáneamente, y se comprueba que los métodos a los que apunten tengan parámetros y valor de retorno del tipo indicado al definirlos.
 - Siempre se comprueba que los valores que se pasan en cada llamada a métodos que admitan un número indefinido de parámetros de un cierto tipo sean de los tipos apropiados.
- **Instrucciones seguras.** En C# se han impuesto una serie de restricciones en el uso de las instrucciones de control más comunes (toda condición está controlada por una expresión condicional, todo caso de un `switch` ha de terminar en un `break` o `goto`, etc.
- **Sistema de tipos unificado.** Todos los tipos de datos que se definan siempre derivarán, aunque sea de manera implícita, de una clase base común llamada `System.Object`, por lo que dispondrán de todos los miembros definidos en ésta clase (es decir, serán "objetos"). Esto también es aplicable a los tipos de datos básicos.
- **Extensibilidad de operadores.** C# permite redefinir el significado de la mayoría de los operadores -incluidos los de conversión, tanto para conversiones implícitas como explícitas- cuando se apliquen a diferentes tipos de objetos.
- **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 biblioteca 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.
- **Versionable.** C# incluye una política de versionado que permite crear nuevas versiones de tipos sin temor a que la introducción de nuevos miembros provoquen errores difíciles de detectar en tipos hijos previamente desarrollados y ya extendidos con miembros de igual nombre a los recién introducidos.
- **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.

- **Compatible.** C# **mantiene** una sintaxis muy similar a C, C++ o Java que permite incluir directamente en código escrito en C# fragmentos de código escrito en estos lenguajes.

En resumen ...

- Lenguaje orientado al desarrollo de componentes (módulos independientes de granularidad mayor que los objetos): propiedades, métodos y eventos; atributos (metadatos); documentación integrada con XML; "one-stop programming" (sin ficheros de cabecera, ni IDL, puede integrarse en páginas ASP... ¡al más puro estilo de Java!)
- Todo es un objeto: desaparece la distinción entre tipos primitivos y objetos de lenguajes como Java o C++ (sin penalizar la eficiencia como en LISP o Smalltalk).
- Software **robusto** y duradero: recolección de basura, excepciones, comprobación de tipos, ausencia de variables sin inicializar y de castings no seguros, gestión de versiones, eliminación de errores comunes (p.ej. if (x=y) ...)...
- ... sin perder de vista el aspecto económico (\$\$\$): posibilidad de utilizar C++ puro (no gestionado), interoperabilidad (XML, SOAP, COM, DLLs...) y curva de aprendizaje suave (para los que ya conocen otros lenguajes de programación usuales, obviamente).

¡Hola, mundo!

```
using System;

class Hola
{
    static void Main( )
    {
        Console.WriteLine("¡Hola, mundo!");
        Console.ReadLine(); // Enter para terminar.
    }
}
```

Aspectos Léxicos

Comentarios

Un **comentario** es texto que incluido en el código fuente de un programa con la idea de facilitar su legibilidad a los programadores y cuyo contenido es, por defecto, completamente ignorado por el compilador.

```
// En una línea, al estilo de C++

/*
    En múltiples líneas, como en C
```

* /

Identificadores

Un **identificador** no es más que un nombre con el que identificaremos algún elemento de nuestro código, ya sea una clase, una variable, un método, etc.

- Deben comenzar por letra o subrayado (_).
- No pueden contener espacios en blanco.
- Pueden contener caracteres Unicode.
- Son sensibles a mayúsculas/minúsculas.
- No pueden coincidir con palabras reservadas (a no ser que tengan el prefijo @).

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

Literales

Un **literal** es la representación explícita de los valores que pueden tomar los tipos básicos del lenguaje.

Literales	
123	// Int
0x7B	// Hexadecimal
123U	// Unsigned
123ul	// Unsigned long
123L	// Long
123.0	// Double
123f	// Float
123D	// Double
123.456m	// Decimal
1.23e2f	// Float
12.3E1M	// Decimal
true	
false	
'A'	// Simple character
'\u0041'	// Unicode
'\x0041'	// Unsigned short hexadecimal
'\n'	// CR+LF
'\''	// Single quote
'\"'	// Double quote
'\\'	// Backslash
'\0'	// Null

Literales
'\t' // Tabulador

Operadores

Un **operador** es un símbolo formado por uno o más caracteres que permite realizar una determinada operación entre uno o más datos y produce un resultado.

C# proporciona un conjunto fijo de operadores, cuyo significado está definido para los tipos predefinidos, si bien algunos de ellos pueden sobrecargarse.

Tipos de operadores: aritméticos, lógicos, relacionales, manipulación de bits, asignación, acceso a tablas, acceso a objetos, creación de objetos, información sobre tipos, etc. La siguiente tabla recoge los operadores de C# de mayor a menor precedencia:

Categoría	Operadores
Operadores primarios	Grouping: (x) Member access: x.y Method call: f(x) Indexing: a[x] Post-increment: x++ Post-decrement: x-- Constructor call: new Type retrieval: typeof Arithmetic check on: checked Arithmetic check off: unchecked
Operadores unarios	Positive value of: + Negative value of: - Not: ! Bitwise complement: ~ Pre-increment: ++x Post-decrement: --x Type cast: (T)x
Operadores multiplicativos	Multiply: * Divide: / Division remainder: %
Operadores aditivos	Add: + Subtract: -
Operadores de desplazamiento	Shift bits left: << Shift bits right: >>
Operadores relacionales	Less than: < Greater than: >

Categoría	Operadores
	Less than or equal to: <= Greater than or equal to: >= Type equality/compatibility: is Type conversion: as
Igualdad	==
Desigualdad	!=
Bitwise AND	&
Bitwise XOR	^
Bitwise OR	
Logical AND	&&
Logical OR	
Condiciona ternario	?:
Asignación	=, *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, =

Asociatividad

Como siempre, es mejor utilizar paréntesis para controlar el orden de evaluación

```
x = y = z   se evalúa como   x = (y = z)
x + y + z   se evalúa como   (x + y) + z
```

Órdenes

- Delimitadas por punto y coma (;) como en C, C++ y Java.
- Los bloques { ... } no necesitan punto y coma al final.

Variables y constantes

```
static void Main()
{
    const float pi = 3.14f;
    const int r = 123;
    Console.WriteLine(pi * r * r);

    int a;
    int b = 2, c = 3;
    a = 1;
    Console.WriteLine(a + b + c);
}
```

El ámbito de una variable abarca desde su declaración hasta que termina el bloque en el que fue declarada. Dentro del ámbito de una variable, es un error declarar otra variable con el mismo identificador (aunque sea en un bloque interno):

```
{
    int x;
    {
        int x;        // Error!!!
    }
}
```

Siempre hay que asignarle un valor a una variable antes de utilizarla (de hecho, lo comprueba el compilador, como en Java).

```
void Foo()
{
    string s;
    Console.WriteLine(s);    // Error
}
```

E/S básica

Aplicaciones en modo consola

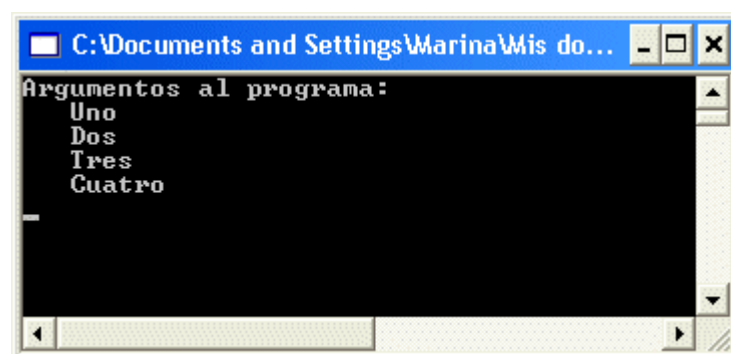
```
System.Console.WriteLine();
System.Console.ReadLine();

using System;

class ArgsApp
{
    static void Main( )
    {
        Console.WriteLine ("Argumentos al programa: ");

        foreach (string s in args)
            Console.WriteLine ("    " + s);

        Console.ReadLine();
    }
}
```



Aplicaciones Windows

```
System.Windows.MessageBox.Show();
```

El sistema unificado de tipos

- Un programa en C# es una colección de tipos: classes, structs, enums, interfaces, delegates.
- Como cualquier lenguaje de programación, C# proporciona una serie de tipos predefinidos (int, byte, char, string, object...) y mecanismos para que el usuario cree sus propios tipos.
- Como en otros lenguajes orientados a objetos, desaparecen las variables y funciones globales. Todo el código y todos los datos de una aplicación forman parte de objetos que encapsulan datos y código.
- En C#, un tipo puede incluir datos (campos, constantes, arrays y eventos), funciones (métodos, operadores, constructores, destructores, propiedades e indexadores) y otros tipos.
- Los tipos se organizan en ficheros, assemblies y espacios de nombres (de forma jerárquica).
- En el sistema unificado de tipos de la plataforma .NET, los tipos pueden ser **valores** (contienen datos, no pueden ser null) o **referencias** (contienen referencias a otros objetos, pueden ser null).

Valor (p.ej. struct)	Referencia (p.ej. string)
La variable contiene un valor	La variable contiene una referencia
El dato se almacena en la pila	El dato se almacena en el heap
El dato siempre tiene valor	El dato puede no tener valor (null)
Valor por defecto: 0	Valor por defecto: null
Una asignación copia el valor	Una asignación copia la referencia

El sistema unificado permite hacer un uso eficiente de la memoria (esto es, no cargar en exceso al recolector de basura) de forma transparente al programador.

```
int i = 123;  
string s = "Hello world";
```



El operador new

Es diferente al de C++:

- En C++ `new` indica que se pide memoria en el heap.
- En C# `new` indica que se llama al constructor de una clase.

El efecto, no obstante, es similar ya que si la variable es de un tipo referencia, al llamar al constructor se aloja memoria en el heap de manera implícita.

Conversiones de tipos

Implícitas	Explícitas
Ocurren automáticamente	Requieren un casting
Siempre tienen éxito	Pueden fallar
No se pierde información	Se puede perder información

```

int x = 123456;
long y = x;           // implicit
short z = (short)x;   // explicit

double d = 1.2345678901234;
float f = (float)d;    // explicit
long l = (long)d;      // explicit

```

Las conversiones de tipo (tanto implícitas como explícitas) puede definir las el usuario.

Polimorfismo

Capacidad de realizar una operación sobre un objeto sin conocer su tipo concreto.

```

void Poly(object o)
{
    Console.WriteLine(o.ToString());
}

Poly(42);
Poly("abcd");
Poly(12.345678901234m);
Poly(new Point(23, 45));

```

Boxing

¿Cómo se tratan polimórficamente valores y referencias? Por ejemplo, en ocasiones deseamos tratar un tipo valor como si fuera un tipo referencia. En otras palabras, ¿cómo se puede convertir un valor entero, por ejemplo, (`int`) en un objeto (`object`)?: **Boxing**.

```

int i = 123;

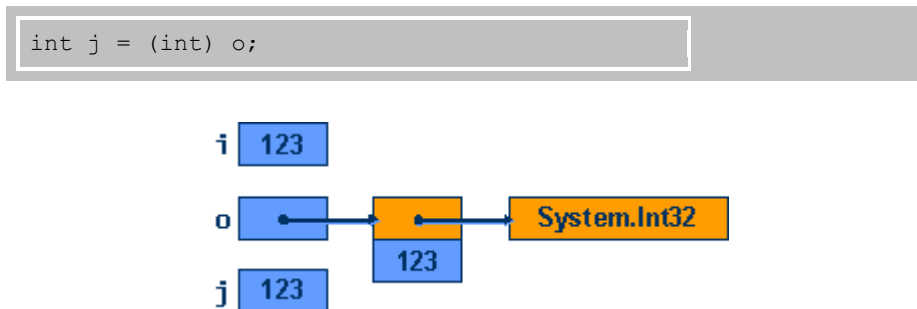
object boxedI = (object) i;

```

El efecto del boxing es el de cualquier otro tipo de *casting*, aunque debe tenerse en cuenta que el contenido de la variable se copia al heap y se crea una referencia a ésta copia.

El motivo de esta conversión es pasar un valor a un método que espera una referencia.

Una vez que se ha hecho boxing puede deshacerse la conversión (**unboxing**) haciendo casting al tipo de dato inicial.



Conclusiones

Ventajas del sistema unificado de tipos: las colecciones funcionan sobre cualquier tipo, el polimorfismo siempre se puede utilizar, se reemplaza el tipo Variant de OLE, elimina la necesidad de wrappers (p.ej. `int` e `Integer`, `float` y `Float`...).

```
Hashtable t = new Hashtable();

t.Add(0, "zero");
t.Add(1, "one");
t.Add(2, "two");

string s = string.Format("Your total was {0} on {1}", total,
date);
```

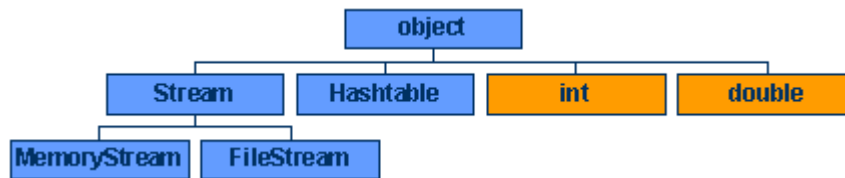
Desventajas del sistema unificado de tipos: Eficiencia.

La necesidad de utilizar *boxing* disminuirá cuando el CLR permita genéricos (algo similar a los *templates* en C++).

Objetos

En C#, todo es un objeto. Cualquier tipo hereda finalmente de `object`. Una referencia a un objeto ocupa 4 bytes, mientras que su existencia supone una carga adicional de 0 u 8 bytes (según sea un valor o una referencia)

```
object (System.Object)
```



Métodos de object

```

public bool Equals(object)
protected void Finalize()
public int GetHashCode()
public System.Type GetType()
protected object MemberwiseClone()
public void Object()
public string ToString()
  
```

this

La palabra reservada `this` es una variable predefinida disponible dentro de las funciones no estáticas de un tipo. Esta variable nos permite acceder a los miembros de un objeto de forma no ambigua, como muestra el siguiente ejemplo:

```

class Persona
{
    string nombre;

    public Persona (string nombre)
    {
        this.nombre = nombre;
    }

    public void Presenta (Persona p)
    {
        if (p != this)
            Console.WriteLine("Hola, soy " + nombre);
    }
}
  
```

Tipos básicos

Los tipos de datos básicos son ciertos tipos de datos tan comúnmente utilizados en la escritura de aplicaciones que en C# se ha incluido una sintaxis especial para tratarlos. Por ejemplo, para representar números enteros de 32 bits con signo se utiliza el tipo de dato `System.Int32` definido en la BCL, aunque a la hora de crear un objeto a de este tipo que represente el valor 2 se usa la siguiente sintaxis:

```

System.Int32 a = 2;
  
```

Como se ve, no se utiliza el operador `new` para crear objetos `System.Int32`, sino que directamente se indica el literal que representa el valor a crear, con lo que la sintaxis necesaria para crear entero de este tipo se reduce considerablemente. Es más, dado lo frecuente que es el uso de este tipo también se ha predefinido en C# el alias `int` para el mismo, por lo que la definición de variable anterior queda así de compacta:

```
int a = 2;
```

`System.Int32` no es el único tipo de dato básico incluido en C#. En el espacio de nombres `System` se han incluido todos estos:

C#	UTS	Características
sbyte	System.Sbyte	1 byte con signo
byte	System.Byte	1 byte sin signo
short	System.Short	2 bytes con signo
ushort	System.UShort	2 bytes sin signo
int	System.Int32	4 bytes con signo
uint	System.UInt32	4 bytes sin signo
long	System.Int64	8 bytes con signo
ulong	System.ULong64	8 bytes sin signo
float	System.Single	IEEE 754, 32 bits
double	System.Double	IEEE 754, 64 bits
decimal	System.Decimal	128 bits, 96 bits valor exacto + escala
bool	System.Boolean	1 byte, 2 en arrays
char	System.Char	Unicode, 2 bytes
string	System.String	Cadenas de caracteres
object	System.Object	Cualquier objeto

Enumeraciones

Una **enumeración** o **tipo enumerado** es un tipo especial de estructura en la que los literales de los valores que pueden tomar sus objetos se indican explícitamente al definirla.

Pueden convertirse explícitamente en su valor entero (como en C). Admiten determinados operadores aritméticos (+,-,++,--) y lógicos a nivel de bits (&|,^,~). Todos los tipos enumerados derivan de `System.Enum`

```
enum State { Off, On }

enum Tamaño {
    Pequeño = 1,
    Mediano = 3,
    Grande = 5
    Inmenso
```

```

}

enum Color: byte {
    Red    = 1,
    Green  = 2,
    Blue   = 4,
    Black  = 0,
    White  = Red | Green | Blue
}

Color c = Color.Black;
Console.WriteLine(c);           // 0
Console.WriteLine(c.ToString()); // Black

```

```

enum ModArchivo
{
    Lectura = 1,
    Escritura = 2,
    Oculto = 4,
    Sistema = 8
}

...
ModArchivo st = ModArchivo.Lectura | ModArchivo.Escritura;
...
Console.WriteLine (st.ToString("F")); // Lectura, Escritura
Console.WriteLine (Enum.Format(typeof(ModArchivo), st, "G")); // 3
Console.WriteLine (Enum.Format(typeof(ModArchivo), st, "X")); // 00000003
Console.WriteLine (Enum.Format(typeof(ModArchivo), st, "D")); // 3

```

Cadenas de caracteres

Una **cadena de caracteres** no es más que una *secuencia* de caracteres Unicode. En C# se representan mediante objetos del tipo `string`, que no es más que un alias del tipo `System.String` incluido en la BCL.

```

string s = "Esto es una cadena... como en C!!!"

string s1= "Esto es una cadena... ";
string s2 = "como en C!!!";
string s3 = s1 + s2;

```

En la clase `System.String` se definen propiedades (`Length`) y muchos métodos aplicables a cualquier cadena y que permiten manipularla: `IndexOf`, `Insert`, `Replace`, etc.

Vectores y matrices

En C# las tablas pueden ser multidimensionales, su índice inicial es 0 y siempre se comprueba que se esté accediendo dentro de los límites.

Todas las tablas que definamos, sea cual sea el tipo de elementos que contengan, son objetos que derivan de `System.Array`. En esta clase también está disponible la propiedad `Length`.

La sintaxis es ligeramente distinta a la del C++ porque las tablas son objetos de tipo referencia:

```
double [] array;           // Declara un a referencia
                           // (no se instancia ningún objeto)
array = new double[10];    // Instancia un objeto de la clase
                           // System.Array y le asigna 10 casillas.
```

que combinadas resulta en (lo habitual):

```
double [] array = new double[10];
```

- El tamaño del vector se determina cuando se instancia.
- La declaración emplea los paréntesis vacíos [] entre el tipo y el nombre para determinar el número de dimensiones (Rank).
- En C# el rango es parte del tipo. El número de elementos no lo es.

Lo más "parecido" en C++ es:

```
double *pArray = new double[10]; // vector (en heap) de C++
```

aunque esta declaración no es válida en C#:

```
double array [10]; // vector (en pila) de C++
```

Otros ejemplos de declaración de vectores:

```
string[] a = new string[10];
// "a" es un vector de 10 cadenas
int[] primes = new int[9];
// "primes" es un vector de 9 enteros
```

Un vector puede inicializarse a la misma vez que se declara. Las tres definiciones siguientes son equivalentes:

```
int[] prime1 = new int[10] {1,2,3,5,7,11,13,17,19,23};
int[] prime2 = new int[] {1,2,3,5,7,11,13,17,19,23};
int[] prime3 = {1,2,3,5,7,11,13,17,19,23};
```

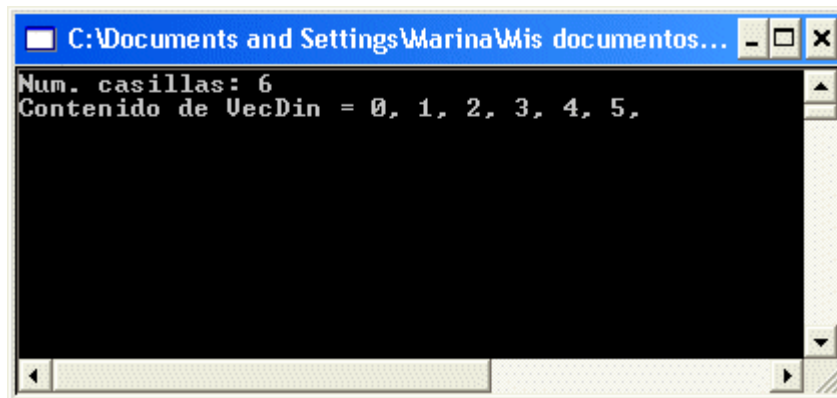
Los vectores pueden dimensionarse dinámicamente:

```
Console.Write ("Num. casillas: ");
string strTam = Console.ReadLine();
int tam = Convert.ToInt16(strTam);

int[] VecDin = new int[tam];
for (int i=0; i<tam; i++) VecDin[i]=i;

Console.Write ("Contenido de VecDin = ");
foreach (int i in VecDin)
```

```
Console.Write(i + ", ");  
  
Console.ReadLine();
```



Matrices

Las diferencias son importantes respecto a C++ ya que C# permite tanto *matrices rectangulares* (todas las filas tienen el mismo número de columnas) como *matrices dentadas* o *a jirones*.

```
int [,] array2D;           // Declara un a referencia  
                           // (no se instancia ningún objeto)  
array2D = new int [2,3];   // Instancia un objeto de la clase  
                           // System.Array y le asigna 6 casillas  
                           // (2 filas con 3 columnas cada una)
```

que combinadas (y con inicialización) podría quedar:

```
int [,] array2D = new int [2,3] {{1,0,4} {3,2,5}};
```

El número de dimensiones puede ser, lógicamente, mayor que dos:

```
int [,,] array3D = new int [2,3,2];
```

El acceso se realiza con el operador habitual [], aunque el rtecorrido se simplifica y clarifica en C# con el ciclo `foreach`:

```
for (int i= 0; i< vector1.Length; i++)  
    vector1[i] = vector2[i];  
...  
foreach (float valor in vector2)  
    Console.Wtite (valor);
```

Una **matriz dentada** no es más que una tabla cuyos elementos son a su vez tablas, pudiéndose así anidar cualquier número de tablas. Cada tabla puede tener un número propio de casillas.

```

Console.WriteLine ("Introduzca las dimensiones: ");

// Peticion de numero de filas (num. vectores)
Console.Write ("Num. Filas: ");
string strFils = Console.ReadLine();
int Fils = Convert.ToInt16(strFils);

// Declaracion de la tabla dentada: el numero de
//          casillas de cada fila es desconocido.
int[][] TablaDentada = new int[Fils][];

// Peticion del numero de columnas de cada vector
for (int f=0; f<Fils; f++)
{
    Console.Write ("Num. Cols. de fila {0}: ", f);
    string strCols = Console.ReadLine();
    int Cols = Convert.ToInt16(strCols);

    // Peticion de memoria para cada fila
    TablaDentada[f] = new int[Cols];
}

// Rellenar todas las casillas de la tabla dentada
for (int f=0; f<TablaDentada.Length; f++)
    for (int c=0; c<TablaDentada[f].Length; c++)
        TablaDentada[f][c]=((f+1)*10)+(c+1);

// Mostrar resultado
Console.WriteLine ("Contenido de la matriz: ");
for (int f=0; f<TablaDentada.Length; f++)
{
    for (int c=0; c<TablaDentada[f].Length; c++)
        Console.Write (TablaDentada[f][c] + " ");
    Console.WriteLine();
}

Console.ReadLine();

```

```

C:\Documents and Settings\Marina\Mis docum...
Introduzca las dimensiones:
Num. Filas: 4
Num. Cols. de fila 0: 2
Num. Cols. de fila 1: 5
Num. Cols. de fila 2: 3
Num. Cols. de fila 3: 7
Contenido de la matriz:
11 12
21 22 23 24 25
31 32 33
41 42 43 44 45 46 47

```

Estructuras

Una **estructura** es un tipo especial de clase pensada para representar objetos *ligeros* (que ocupen poca memoria y deban ser manipulados con velocidad) como objetos que representen puntos, fechas, etc.

Structs en C#	Structs en C++
Tipo (valor) definido por el usuario	Como una clase en C++ con todos sus miembros públicos
Se almacenan en la pila o como un miembros de otros objetos en el heap	Puede almacenarse en la pila o en el heap (de forma independiente o como miembro de otro objeto)
Sus miembros pueden ser public, internal o private	Sus miembros siempre son public
Puede tener un constructor sin parámetros	No puede tener un constructor sin parámetros

```
public struct Point
{
    public int x, y;

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

```
Point p = new Point(2,5);
Point p2;

p.x += 100;

int px = p.x;    // px = 102

p2 = p;
p2.x += 100;

Console.WriteLine (px);    // 102
Console.WriteLine (p2.x);  // 202
```

Clases

Un **objeto** es un agregado de datos y de métodos que permiten manipular dichos datos, y un programa en C# no es más que un conjunto de objetos que interaccionan unos con otros a través de sus métodos.

Una **clase** es la definición de las características concretas de un determinado tipo de objetos: cuáles son los **datos** y los **métodos** de los que van a disponer todos los objetos de ese tipo.

```
using System;

class Coche
{
    // Campos
    protected double velocidad=0;
    public string Marca;
```

```

public string Modelo;
public string Color;
public string NumBastidor;

// Método constructor
public Coche(string marca, string modelo,
             string color, string numbastidor)
{
    this.Marca=marca;
    this.Modelo=modelo;
    this.Color=color;
    this.NumBastidor=numbastidor;
}

// Propiedad (solo lectura)
public double Velocidad
{
    get { return this.velocidad; }
}

// Método
public void Acelerar(double cantidad)
{
    Console.WriteLine("--> Incrementando veloc. en {0} km/h", cantidad);
    this.velocidad += cantidad;
}

// Método
public void Girar(double cantidad)
{
    Console.WriteLine("--> Girando {0} grados", cantidad);
}

// Método
public void Frenar(double cantidad)
{
    Console.WriteLine("--> Reduciendo veloc. en {0} km/h", cantidad);
    this.velocidad -= cantidad;
}

// Método
public void Aparcar()
{
    Console.WriteLine("-->Aparcando coche");
    this.velocidad = 0;
}
} // class Coche

class EjemploCocheApp
{
    static void Main(string[] args)
    {
        Coche MiCoche = new Coche("Citroën", "Xsara Picasso",
                                   "Rojo","1546876");

        Console.WriteLine("Los datos de mi coche son:");
        Console.WriteLine("  Marca: {0}", MiCoche.Marca);
        Console.WriteLine("  Modelo: {0}", MiCoche.Modelo);
        Console.WriteLine("  Color: {0}", MiCoche.Color);
        Console.WriteLine("  Número de bastidor: {0}",
                           MiCoche.NumBastidor);
        Console.WriteLine();
    }
}

```

```

        MiCoche.Acelerar(100);
        Console.WriteLine("La velocidad actual es de {0} km/h",
            MiCoche.Velocidad);

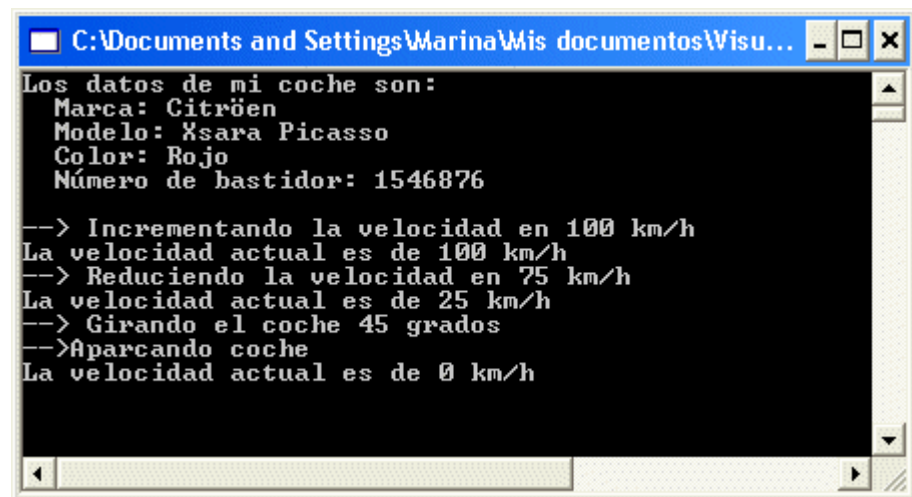
        MiCoche.Frenar(75);
        Console.WriteLine("La velocidad actual es de {0} km/h",
            MiCoche.Velocidad);

        MiCoche.Girar(45);

        MiCoche.Aparcar();
        Console.WriteLine("La velocidad actual es de {0} km/h",
            MiCoche.Velocidad);

        Console.ReadLine();
    }
} // class EjemploCocheApp

```



Clases vs. structs

Tanto las clases como los structs permiten al usuario definir sus propios tipos, pueden implementar múltiples interfaces y pueden contener datos (campos, constantes, eventos...), funciones (métodos, propiedades, indexadores, operadores, constructores, destructores y eventos) y otros tipos internos (clases, structs, enums, interfaces y delegados).

```

using System;

struct SPoint
{
    int x, y;
    public SPoint(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
    public int X {
        get { return x; }
        set { x = value; }
    }
}

```

```

    public int Y {
        get { return y; }
        set { y = value; }
    }
}

class CPoint
{
    int x, y;
    public CPoint(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
    public int X {
        get { return x; }
        set { x = value; }
    }
    public int Y {
        get { return y; }
        set { y = value; }
    }
}

class Class2App
{
    static void Main(string[] args)
    {
        SPoint sp = new SPoint(2,5);
        sp.X += 100;
        int spx = sp.X;      // spx = 102

        CPoint cp = new CPoint(2,5);
        cp.X += 100;
        int cpx = cp.X;      // cpx = 102

        Console.WriteLine ("spx es: {0} ", spx);
        Console.WriteLine ("cpx es: {0} ", cpx);
        Console.ReadLine ();
    }
}

```

Aunque las coincidencias son muchas, existen, no obstante, existen algunas diferencias entre ellos:

Clase	Struct
Tipo referencia	tipo valor
Puede heredar de otro tipo (que no esté "sellado")	No permite herencia (hereda únicamente de <code>System.ValueType</code>)
Puede tener un destructor	No puede tener destructor

Supongamos la clase `MiClase` y el struct de tipo `MiStruct`. Según sabemos de C#, las instancias de `MiClase` se almacenan en el heap mientras las instancias de `MiStruct` se almacenan en la pila:

<code>MiClase cl;</code>	Declara una referencia. Igual a la declaración de un puntero no inicializado en C++
<code>cl = new MiClase();</code>	Crea una instancia de <code>MiClase</code> . LLama al constructor sin parámetros de la clase. Además, reserva memoria en el heap.
<code>MiStruct st;</code>	Crea una instancia de <code>MiStruct</code> pero no llama a ningún constructor. Los campos de <code>st</code> quedan no inicializados.
<code>st = new MiStruct();</code>	Llama al constructor: se inicializan los campos. No se reserva memoria porque <code>st</code> ya existe en la pila.

Variables de instancia vs. miembros estáticos

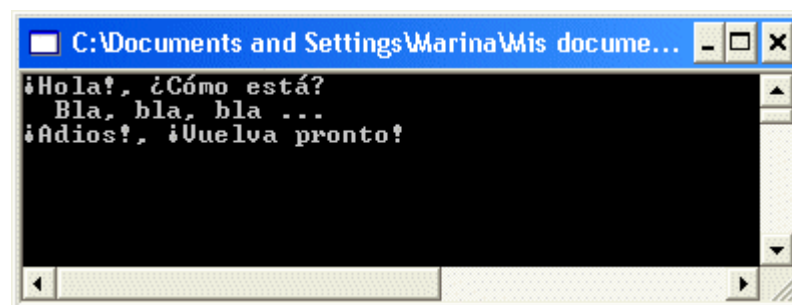
Por defecto, los miembros de una clase son variables de instancia (existe una copia de los datos por cada instancia de la clase y los métodos se aplican sobre los datos de una instancia concreta).

Se pueden definir miembros estáticos que son comunes a todas las instancias de la clase. Lógicamente, los métodos estáticos no pueden acceder a variables de instancia, ni a la variable `this` que hace referencia al objeto actual.

```
using System;

class Mensaje {
    public static string Bienvenida = "¡Hola!, ¿Cómo está?";
    public static string Despedida = "¡Adios!, ¡Vuelva pronto!";
}

class ConstructoresStaticApp {
    static void Main()
    {
        Console.WriteLine(Mensaje.Bienvenida);
        Console.WriteLine("  Bla, bla, bla ... ");
        Console.WriteLine(Mensaje.Despedida);
        Console.ReadLine();
    }
}
```



De cualquier forma, no conviene abusar de los miembros estáticos, ya que son básicamente datos y funciones globales en entornos orientados a objetos.

Miembros de una clase

Constantes (const)

Una constante es un dato cuyo valor se evalúa en tiempo de compilación y, por tanto, es implícitamente estático (p.ej. `Math.PI`).

```
public class MiClase
{
    public const string version = "1.0.0";
    public const string s1 = "abc" + "def";
    public const int i3 = 1 + 2;
    public const double PI_I3 = i3 * Math.PI;
    public const double s = Math.Sin(Math.PI); //ERROR
    ...
}
```

Campos (fields)

Un campo es una variable que almacena datos en una clase o struct (referencias a instancias de clases, structs o arrays).

Campos de sólo lectura (readonly): Similares a las constantes, si bien su valor se inicializa *en tiempo de ejecución* (en su declaración o en el constructor). A diferencia de las constantes, si cambiamos su valor no hay que recompilar los clientes de la clase. Además, los campos de sólo lectura pueden ser variables de instancia o variables estáticas.

```
public class MiClase
{
    public static readonly double d1 = Math.Sin(Math.PI);
    public readonly string s1;

    public MiClase(string s)
    {
        s1 = s;
    }
}
.....
MiClase mio = new MiClase ("Prueba");
Console.WriteLine(mio.s1);
Console.WriteLine(MiClase.d1);
.....
```

Produce como resultado:

```
1,22460635382238E-16
Prueba
```

Propiedades

Las propiedades son *campos virtuales*, al estilo de Delphi o C++Builder. Su aspecto es el de un campo (desde el exterior de la clase no se diferencian) pero están implementadas con código como los métodos. Pueden ser de sólo lectura, de sólo escritura o de lectura y escritura.

```
public class Button: Control
{
    private string caption;

    public string Caption
    {
        get { return caption; }
        set { caption = value; Repaint(); }
    }
}

Button b = new Button();

b.Caption = "OK";          // Copia "OK" al campo -private-
                           // "caption"
String s = b.Caption;      // Copia "OK" (valor del campo -
                           // private-
                           // "caption") al string "s"
```

Indexadores (indexers)

C# no permite, *hablado* con rigor, la sobrecarga del operador de acceso a tablas []. Si permite, no obstante, definir lo que llama un *indexador* para una clase que permite la misma funcionalidad.

Los indexadores permiten definir código a ejecutar cada vez que se acceda a un objeto del tipo del que son miembros usando la *sintaxis* propia de las tablas, ya sea para leer o escribir. Esto es especialmente útil para hacer más clara la sintaxis de acceso a elementos de objetos que puedan contener *colecciones* de elementos, pues permite tratarlos como si fuesen tablas normales.

A diferencia de las tablas, los índices que se les pase entre corchetes no tiene porqué ser enteros, pudiéndose definir varios indexadores en un mismo tipo siempre y cuando cada uno tome un número o tipo de índices diferente.

La sintaxis empleada para la definición de un indexador es similar a la de la definición de una propiedad.

```
public class MiClase
{
    ...
    public string this[int x]
    {
        get {
            // Obtener un elemento
        }
        set {
            // Fijar un elemento
        }
    }
}
```

```

    ...
}
...
MiClase MiObjeto = new MiClase();

```

El código que aparece en el bloque `get` se ejecuta cuando la expresión `MiObjeto[x]` aparece en la parte derecha de una expresión mientras que el código que aparece en el bloque `set` se ejecuta cuando `MiObjeto[x]` aparece en la parte izquierda de una expresión.

Algunas consideraciones finales:

- Igual que las propiedades, pueden ser de sólo lectura, de sólo escritura o de lectura y escritura.
- El nombre dado a un indexador siempre ha de ser `this`.
- Lo que diferenciará a unos indexadores de otros será el número y tipo de sus índices.

Ejemplo

```

Using System;

namespace IndexadorCoches
{
    public class Coche
    {
        private int VelocMax;
        private string Marca;
        private string Modelo;

        public Coche (string marca, string modelo, int velocMax)
        {
            this.VelocMax = velocMax;
            this.Marca = marca;
            this.Modelo = modelo;
        }

        public void MuestraCoche ()
        {
            Console.WriteLine (this.Marca + " " + this.Modelo +
                               " (" + this.VelocMax + " Km/h)");
        }
    } // class Coche

    public struct DataColeccionCoches
    {
        public int numCoches;
        public int maxCoches;
        public Coche[] vectorCoches;

        public DataColeccionCoches (int max)
        {
            this.numCoches = 0;
            this.maxCoches = max;
            this.vectorCoches = new Coche[max];
        }
    } // struct DataColeccionCoches

```



```

public class Coches
{
    // Los campos se encapsulan en un struct
    DataColeccionCoches data;

    // Constructor sin parámetros
    public Coches()
    {
        this.data = new DataColeccionCoches(10);
        // Si no se pone un valor se llamará al
        // constructor sin parámetros (por defecto) del
        // struct y tendremos problemas: no se puede
        // explicitar el constructor sin parámetros
        // para un struct.
    }

    // Constructor con parámetro
    public Coches(int max)
    {
        this.data = new DataColeccionCoches(max);
    }

    public int MaxCoches
    {
        set { data.maxCoches = value; }
        get { return (data.maxCoches); }
    }
    public int NumCoches
    {
        set { data.numCoches = value; }
        get { return (data.numCoches); }
    }
    public void MuestraDatos ()
    {
        Console.WriteLine (" --> Maximo= {0}",
            this.MaxCoches);
        Console.WriteLine (" --> Real  = {0}",
            this.NumCoches);
    }

    // El indexador devuelve un objeto Coche de acuerdo
    // a un índice numérico
    public Coche this[int pos]
    {
        // Devuelve un objeto del vector de coches
        get
        {
            if(pos < 0 || pos >= MaxCoches)
                throw new
                    IndexOutOfRangeException("Fuera de rango");
            else
                return (data.vectorCoches[pos]);
        }
        // Escribe en el vector
        set { this.data.vectorCoches[pos] = value;}
    }
} // class Coches

class IndexadorCochesApp
{
    static void Main(string[] args)
    {
        // Crear una colección de coches
        Coches MisCoches = new Coches (); // Por defecto (10)
    }
}

```

```

        Console.WriteLine ("***** Mis Coches *****");
        Console.WriteLine ("Inicialmente: ");
        MisCoches.MuestraDatos();

        // Añadir coches. Observar el acceso con [] ("set")
        MisCoches[0] = new Coche ("Opel", "Zafira", 200);
        MisCoches[1] = new Coche ("Citroën", "Xsara", 220);
        MisCoches[2] = new Coche ("Ford", "Focus", 190);

        MisCoches.NumCoches = 3;

        Console.WriteLine ("Despues de insertar 3 coches: ");
        MisCoches.MuestraDatos();

        // Mostrar la colección
        Console.WriteLine ();
        for (int i=0; i<MisCoches.NumCoches; i++)
        {
            Console.Write ("Coche Num.{0}: ", i+1);
            MisCoches[i].MuestraCoche(); // Acceso ("get")
        }
        Console.ReadLine ();

        // *****

        // Crear una colección de coches
        Coches TusCoches = new Coches (4);

        Console.WriteLine ("***** Tus Coches *****");
        Console.WriteLine ("Inicialmente: ");
        TusCoches.MuestraDatos();

        // Añadir coches. Observar el acceso con []
        TusCoches[TusCoches.NumCoches++] =
            new Coche ("Opel", "Corsa", 130);
        TusCoches[TusCoches.NumCoches++] =
            new Coche ("Citroën", "C3", 140);

        Console.WriteLine ("Despues de insertar 2 coches: ");
        TusCoches.MuestraDatos();

        // Mostrar la colección
        Console.WriteLine ();
        for (int i=0; i<TusCoches.NumCoches; i++)
        {
            Console.Write ("Coche Num.{0}: ", i+1);
            TusCoches[i].MuestraCoche();
        }

        Console.ReadLine ();

    } // Main

} // class IndexadorCochesApp

} // namespace IndexadorCoches

```

Métodos

Implementan las operaciones que se pueden realizar con los objetos de un tipo concreto. Constructores, destructores y operadores son casos particulares de métodos. Las propiedades y los indexadores se implementan con métodos (`get` y `set`).

Como en cualquier lenguaje de programación, los métodos pueden tener parámetros, contener órdenes y devolver un valor (con `return`).

Por defecto, los parámetros se pasan por valor (por lo que los tipos "valor" no podrían modificarse en la llamada a un método). El modificador `ref` permite que pasemos parámetros por referencia. Para evitar problemas de mantenimiento, el modificador `ref` hay que especificarlo tanto en la definición del método como en el código que realiza la llamada. Además, **la variable que se pase por referencia ha de estar inicializada previamente.**

```
void RefFunction (ref int p)
{
    p++;
}

int x = 10;

RefFunction(ref x);

// x vale ahora 11
```

El modificador `out` permite devolver valores a través de los argumentos de un método. De esta forma, se permite que el método inicialice el valor de una variable. En cualquier caso, la variable ha de tener un valor antes de terminar la ejecución del método. Igual que antes, Para evitar problemas de mantenimiento, el modificador `out`

hay que especificarlo tanto en la definición del método como en el código que realiza la llamada.

```
void OutFunction(out int p)
{
    p = 22;
}

int x;

// x aún no está inicializada

OutFunction(out x);

Console.WriteLine(x); // x vale ahora 22
```

Sobrecarga de métodos: Como en otros lenguajes, el identificador de un método puede estar sobrecargado siempre y cuando las firmas de las distintas implementaciones del método sean únicas (la firma tiene en cuenta los *argumentos*, no el tipo de valor que devuelven).

```
void Print(int i);
void Print(string s);
void Print(char c);
void Print(float f);

int Print(float f); // Error: Signatura duplicada
```

Arrays de parámetros: Como en C, un método puede tener un número variable de argumentos.

```
int Sum(params int[] intArr)
{
    int sum = 0;

    foreach (int i in intArr)
        sum += i;

    return sum;
}

int sum1 = Sum(13,87,34); // sum1 vale 134
Console.WriteLine(sum1);
int sum2 = Sum(13,87,34,6); // sum2 vale 140
Console.WriteLine(sum2);
```

Constructores: Métodos especiales que son invocados cuando se instancia una clase (o un struct).

- Se emplean habitualmente para inicializar correctamente un objeto.
- Como cualquier otro método, pueden sobrecargarse.
- Si una clase no define ningún constructor se crea un constructor sin parámetros (implícito).
- No se permite un constructor sin parámetros para los struct.

- Si una clase no define ningún constructor se crea un constructor sin parámetros (implícito).
- Un constructor puede llamar a otro constructor mediante un "inicializador".
- El inicializador por defecto es `base()`. La palabra clave `base` se utiliza para obtener acceso a los miembros de la clase base desde una clase derivada.
- Pueden llamar a `this()` o `base()`

```
public class B
{
    private int h;

    public B() {}
    public B(int h)
    {
        this.h = h;
    }
    public int H
    {
        get { return h; }
        set { h = value; }
    }
}

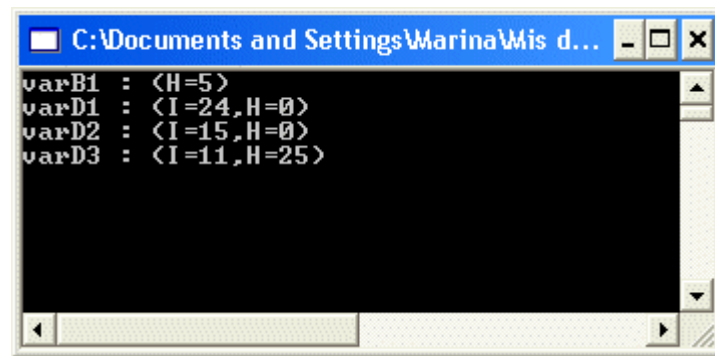
public class D : B // "D" hereda de "B"
{
    private int i;

    public D() : this(24) { }
    public D(int i) { this.i = i; }
    public D(int h, int i) : base(h) { this.i = i; }
    public int I
    {
        get { return i; }
        set { i = value; }
    }
}

...
B varB1 = new B();
varB1.H = 5;
Console.WriteLine("varB1 : (H={0})", varB1.H);

D varD1 = new D();
D varD2 = new D(15);
D varD3 = new D(25, 11);

Console.WriteLine("varD1 : (I={0},H={1})",
    varD1.I, varD1.H);
Console.WriteLine("varD2 : (I={0},H={1})",
    varD2.I, varD2.H);
Console.WriteLine("varD3 : (I={0},H={1})",
    varD3.I, varD3.H);
Console.ReadLine();
```



C# permite especificar código para inicializar una clase mediante un **constructor estático**. El constructor estático se invoca una única vez, antes de llamar al constructor de una instancia particular de la clase o a cualquier método estático de la clase. Sólo puede haber un constructor estático por tipo y éste no puede tener parámetros.

Destructores (como en C++): Se utilizan para liberar los recursos reservados por una instancia (justo antes de que el recolector de basura libere la memoria que ocupe la instancia).

A diferencia de C++, *la llamada al destructor no está garantizada* por lo que tendremos que utilizar una orden `using` e implementar el interfaz `IDisposable` para asegurarnos de que se liberan los recursos asociados a un objeto). Sólo las clases pueden tener destructores (no los `struct`).

```
class Foo
{
    ~Foo()
    {
        Console.WriteLine("Destruído {0}", this);
    }
}
```

Sobrecarga de operadores: Como en C++, se pueden sobrecargar (*siempre con un método static*) algunos operadores unarios (+, -, !, ~, ++, --, true, false) y binarios (+, -, *, /, %, &, |, ^, ==, !=, <, >, <=, >=, <, >).

No se puede sobrecargar el acceso a miembros, la invocación de métodos, el operador de asignación ni los operadores `sizeof`, `new`, `is`, `as`, `typeof`, `checked`, `unchecked`, `&`, `||` y `?:`.

Los operadores `&` y `||` se evalúan directamente a partir de los operadores `&` y `|`.

La sobrecarga de un operador binario (v.g. `*`) sobrecarga implícitamente el operador de asignación correspondiente (v.g. `*=`).

```
using System;

class OverLoadApp
{
    public class Point
```

```

{
    int x, y;

    public Point()
    {
        this.x = 0;
        this.y = 0;
    }
    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
    public int X
    {
        get { return x; }
        set { x = value; }
    }
    public int Y
    {
        get { return y; }
        set { y = value; }
    }

    // Operadores de igualdad

    public static bool operator == (Point p1, Point p2)
    {
        return ((p1.x == p2.x) && (p1.y == p2.y));
    }
    public static bool operator != (Point p1, Point p2)
    {
        return !(p1==p2);
    }

    // Operadores aritméticos

    public static Point operator + (Point p1, Point p2)
    {
        return new Point(p1.x+p2.x, p1.y+p2.y);
    }
    public static Point operator - (Point p1, Point p2)
    {
        return new Point(p1.x-p2.x, p1.y-p2.y);
    }
}

static void Main(string[] args)
{
    Point p1 = new Point(10,20);
    Point p2 = new Point();
    p2.X = p1.X;
    p2.Y = p1.Y;

    Point p3 = new Point(22,33);

    Console.WriteLine ("p1 es: ({0},{1})", p1.X, p1.Y);
    Console.WriteLine ("p2 es: ({0},{1})", p2.X, p2.Y);
    Console.WriteLine ("p3 es: ({0},{1})", p3.X, p3.Y);

    if (p1 == p2) Console.WriteLine ("p1 y p2 son iguales");
    else Console.WriteLine ("p1 y p2 son diferentes");

    if (p1 == p3) Console.WriteLine ("p1 y p3 son iguales");
    else Console.WriteLine ("p1 y p3 son diferentes");
}

```

```

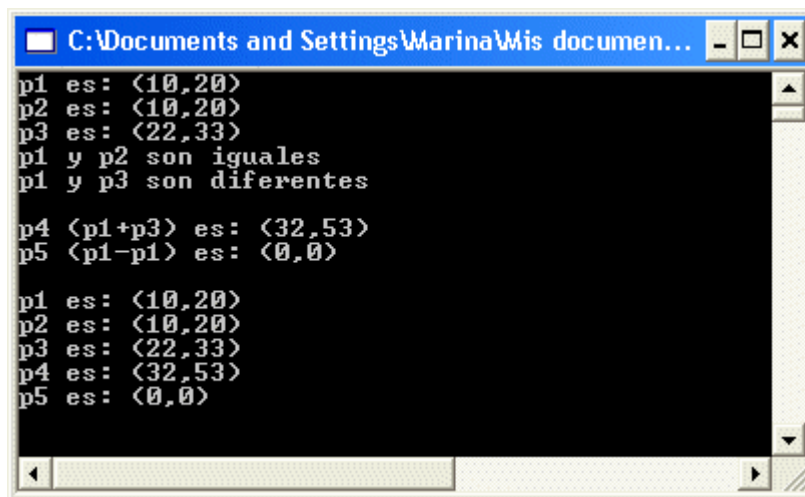
        Console.WriteLine ();

        Point p4 = p1 + p3;
        Console.WriteLine ("p4 (p1+p3) es: ({0},{1})", p4.X,
p4.Y);
        Point p5 = p1 - p1;
        Console.WriteLine ("p5 (p1-p1) es: ({0},{1})", p5.X,
p5.Y);

        Console.WriteLine ();
        Console.WriteLine ("p1 es: ({0},{1})", p1.X, p1.Y);
        Console.WriteLine ("p2 es: ({0},{1})", p2.X, p2.Y);
        Console.WriteLine ("p3 es: ({0},{1})", p3.X, p3.Y);
        Console.WriteLine ("p4 es: ({0},{1})", p4.X, p4.Y);
        Console.WriteLine ("p5 es: ({0},{1})", p5.X, p5.Y);

        Console.ReadLine ();
    }
}

```



Para asegurar la compatibilidad con otros lenguajes de .NET:

```

// Operadores aritméticos

public static Point operator + (Point p1, Point p2) {
    return SumaPoints (p1, p2);
}
public static Point operator - (Point p1, Point p2) {
    return RestaPoints (p1, p2);
}
public static Point RestaPoints (Point p1, Point p2) {
    return new Point(p1.x-p2.x, p1.y-p2.y);
}
public static Point SumaPoints (Point p1, Point p2) {
    return new Point(p1.x+p2.x, p1.y+p2.y);
}

```

Y respecto a los operadores de comparación:

```

// Operadores de igualdad

public static bool operator == (Point p1, Point p2) {

```



```

        return (p1.Equals(p2));
    }
    public static bool operator != (Point p1, Point p2) {
        return (!p1.Equals(p2));
    }
    public override bool Equals (object o) { // Por valor
        if ( (((Point) o).x == this.x) &&
            (((Point) o).y == this.y) )
            return true;
        else return false;
    }
    public override int GetHashCode() {
        return (this.ToString().GetHashCode());
    }
}

```

La sobrecarga de los operadores relacionales obliga a implementar la interface **IComparable**, concretamente el método **CompareTo**:

```

public class Point : IComparable {

    .....

    // Operadores relacionales

    public int CompareTo(object o) {
        Point tmp = (Point) o;
        if (this.x > tmp.x) return 1;
        else
            if (this.x < tmp.x) return -1;
            else return 0;
    }

    public static bool operator < (Point p1, Point p2) {
        IComparable ic1 = (IComparable) p1;
        return (ic1.CompareTo(p2) < 0);
    }
    public static bool operator > (Point p1, Point p2) {
        IComparable ic1 = (IComparable) p1;
        return (ic1.CompareTo(p2) > 0);
    }
    public static bool operator <= (Point p1, Point p2) {
        IComparable ic1 = (IComparable) p1;
        return (ic1.CompareTo(p2) <= 0);
    }
    public static bool operator >= (Point p1, Point p2) {
        IComparable ic1 = (IComparable) p1;
        return (ic1.CompareTo(p2) >= 0);
    }

    .....
} // class Point

static void Main(string[] args)
{
    .....
    if (p1 > p2) Console.WriteLine ("p1 > p2");
    if (p1 >= p2) Console.WriteLine ("p1 >= p2");
    if (p1 < p2) Console.WriteLine ("p1 < p2");
    if (p1 <= p2) Console.WriteLine ("p1 <= p2");
    if (p1 == p2) Console.WriteLine ("p1 == p2");
    Console.WriteLine ();

    if (p1 > p3) Console.WriteLine ("p1 > p3");
}

```

```

        if (p1 >= p3) Console.WriteLine ("p1 >= p3");
        if (p1 < p3) Console.WriteLine ("p1 < p3");
        if (p1 <= p3) Console.WriteLine ("p1 <= p3");
        if (p1 == p3) Console.WriteLine ("p1 == p3");
        Console.WriteLine ();
    }
}

```

El operador de asignación (=) cuando se aplica a clases copia la referencia, no el contenido. Si se desea copiar instancias de clases lo habitual en C# es redefinir (override) el método `MemberwiseCopy()` que heredan, por defecto, todas las clases en C# de `System.Object`.

También pueden programarse las conversiones de tipo (tanto explícitas como implícitas):

```

Using System;

class OverLoadApp
{
    public class Euro
    {
        private int cantidad;

        public Euro(int value)
        {
            cantidad = value;
        }
        public int valor
        {
            get { return cantidad; }
        }

        public static implicit operator double (Euro x)
        {
            return ((double) x.cantidad);
        }

        public static explicit operator Euro(double x)
        {
            double arriba = Math.Ceiling(x);
            double abajo = Math.Floor(x);
            int valor = ((x+0.5 >= arriba) ? (int)arriba :
(int)abajo);
            return new Euro(valor);
        }
    }

    static void Main(string[] args)
    {
        double d1 = 442.578;
        double d2 = 123.22;
        Euro e1 = (Euro) d1;
        Euro e2 = (Euro) d2;

        Console.WriteLine ("d1 es {0} y e1 es {1}",
            d1, e1.valor);
        Console.WriteLine ("d2 es {0} y e2 es {1}",
            d2, e2.valor);

        double n1 = e1;
        double n2 = e2;
    }
}

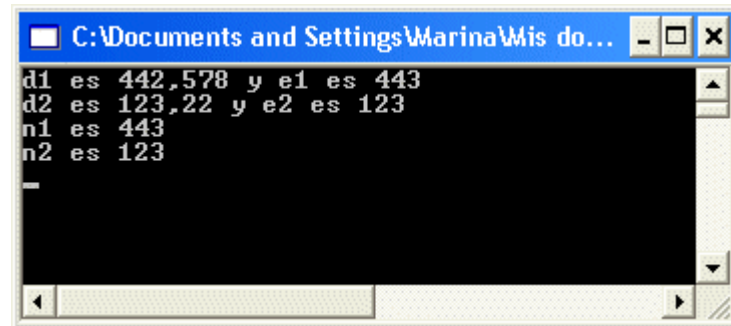
```

```

        Console.WriteLine ("n1 es {0}", n1);
        Console.WriteLine ("n2 es {0}", n2);

        Console.ReadLine ();
    }
}

```



C# no permite definir conversiones entre clases que se relacionan mediante herencia. Dichas conversiones están ya disponibles: de manera implícita desde una clase derivada a una antecesora y de manera explícita a la inversa.

Modificadores de acceso

Los modificadores de acceso nos permiten especificar quién puede usar un tipo o un miembro del tipo, de forma que nos permiten gestionar la encapsulación de los objetos en nuestras aplicaciones:

- Los tipos de nivel superior (aquéllos que se encuentran directamente en un namespace) pueden ser `public` o `internal`
- Los miembros de una clase pueden ser `public`, `private`, `protected`, `internal` o `protected internal`
- Los miembros de un struct pueden ser `public`, `private` o `internal`

Modificador de acceso	Un miembro del tipo T definido en el assembly A es accesible...
<code>public</code>	desde cualquier sitio
<code>private</code> (por defecto)	sólo desde dentro de T (por defecto)
<code>protected</code>	desde T y los tipos derivados de T
<code>internal</code>	desde los tipos incluidos en A
<code>protected internal</code>	desde T, los tipos derivados de T y los tipos incluidos en A

Herencia

C# sólo permite herencia simple.

Siempre que se redefine un método que aparecía en la clase base, hay que utilizar explícitamente la palabra reservada `override` y, de esta forma, se evitan redefiniciones accidentales (una fuente de errores en lenguajes como Java o C++).

Si en la clase `Point` añadimos el método `ToString`:

```
public class Point
{
    ...
    public override string ToString()
    {
        return "["+this.X+", "+this.Y+"]";
    }
    ...
}
```

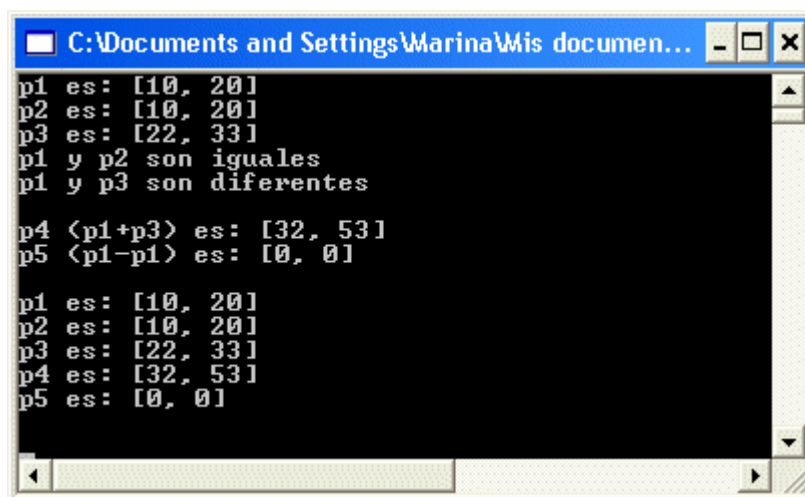
y sustituimos las instrucciones del tipo:

```
Console.WriteLine ("p1 es: ({0},{1})", p1.X, p1.Y);
```

por:

```
Console.WriteLine ("p1 es: " + p1.ToString());
```

el resultado de la ejecución es:



```
p1 es: [10, 20]
p2 es: [10, 20]
p3 es: [22, 33]
p1 y p2 son iguales
p1 y p3 son diferentes

p4 <p1+p3> es: [32, 53]
p5 <p1-p1> es: [0, 0]

p1 es: [10, 20]
p2 es: [10, 20]
p3 es: [22, 33]
p4 es: [32, 53]
p5 es: [0, 0]
```

La palabra reservada `base` sirve para hacer referencia a los miembros de una clase que quedan ocultos por otros miembros de la clase actual:

```
class Shape
{
    int x, y;

    public override string ToString()
```

```

    {
        return ("x=" + x + ",y=" + y);
    }
}

class Circle : Shape
{
    int r;

    public override string ToString()
    {
        return (base.ToString() + ",r=" + r);
    }
}

```

Métodos virtuales

Los métodos pueden ser *virtuales* o no virtuales (por defecto).

- Los métodos no virtuales no son polimórficos (no pueden reemplazarse) ni pueden ser abstractos.
- Los métodos virtuales se definen en una clase base (empleando la palabra reservada `virtual`) y pueden ser reemplazados (empleando la palabra reservada `override`) en las subclases (éstas proporcionan su propia -específica- implementación).

Generalmente, contendrán una implementación por defecto del método (si no, se deberían utilizar métodos *abstractos*).

```

class Shape // Clase base
{
    // "Draw" es un método virtual
    public virtual void Draw() { ... }
}

class Box : Shape
{
    // Reemplaza al método Draw de la clase base
    public override void Draw() { ... }
}

class Sphere : Shape
{
    // Reemplaza al método Draw de la clase base
    public override void Draw() { ... }
}

```

```

void HandleShape(Shape s)
{
    ...
    s.Draw(); // Polimorfismo
    ...
}

```

```

HandleShape(new Box());

HandleShape(new Sphere());

HandleShape(new Shape());

```

NOTA: Propiedades, indexadores y eventos también pueden ser virtuales.

Clases abstractas

Las clases abstractas (`abstract`) son *clases que no pueden ser* instanciadas. Permiten incluir métodos abstractos y métodos no abstractos cuya implementación hace que sirvan de clases base (herencia de implementación). Como es lógico, no pueden estar "selladas".

Métodos abstractos

Un método abstracto es un método virtual sin implementación y debe pertenecer a una clase abstracta. Su implementación se realizará en una clase derivada.

```
abstract class Shape // Clase base abstracta
{
    public abstract void Draw(); // Método abstracto
}

class Box : Shape
{
    public override void Draw() { ... }
}

class Sphere : Shape
{
    public override void Draw() { ... }
}

void HandleShape(Shape s)
{
    ...
    s.Draw();
    ...
}

HandleShape(new Box());

HandleShape(new Sphere());

HandleShape(new Shape()); // Error !!!
```

Clases selladas

Una clase sellada (`sealed`), es una clase de la que no pueden derivarse otras clases (esto es, no puede utilizarse como clase base). Obviamente, no puede ser una clase abstracta.

Los structs en C# son implícitamente clases selladas

¿Para qué sirve sellar clases? Para evitar que se puedan crear subclases y optimizar el código (ya que las llamadas a las funciones de una clase sellada pueden resolverse en tiempo de compilación).

Tipos anidados

C# permite declarar tipos anidados, esto es, tipos definidos en el ámbito de otro tipo. El anidamiento nos permite que el tipo anidado pueda acceder a todos los miembros del tipo que lo engloba (independientemente de los modificadores de acceso) y que el tipo esté oculto de cara al exterior (salvo que queramos que sea visible, en cuyo caso habrá que especificar el nombre del tipo que lo engloba para poder acceder a él).

Operadores especiales

is

Se utiliza para comprobar dinámicamente si el tipo de un objeto es compatible con un tipo especificado (`instanceof` en Java).

No conviene abusar de este operador (es preferible diseñar correctamente una jerarquía de tipos).

```
static void DoSomething(object o)
{
    if (o is Car)
        ((Car)o).Drive();
}
```

as

Intenta convertir de tipo una variable (al estilo de los casts dinámicos de C++). Si la conversión de tipo no es posible, el resultado es `null`. Es más eficiente que el operador `is`, si bien tampoco es conveniente abusar del operador `as`.

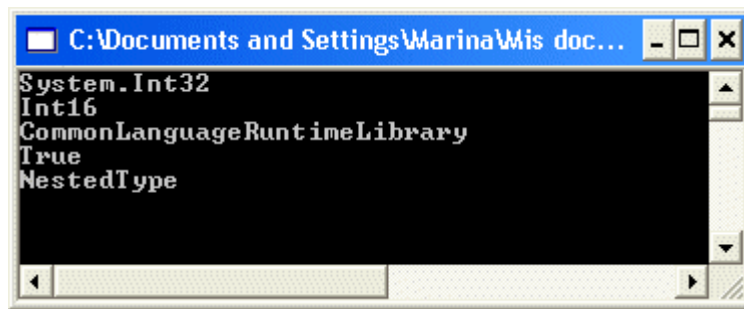
```
static void DoSomething(object o)
{
    Car c = o as Car;

    if (c != null)
        c.Drive();
}
```

typeof

El operador `typeof` devuelve el objeto derivado de `System.Type` correspondiente al tipo especificado. De esta forma se puede hacer *reflexión* para obtener dinámicamente información sobre los tipos (como en Java).

```
Console.WriteLine(typeof(int).FullName);
Console.WriteLine(typeof(System.Int).Name);
Console.WriteLine(typeof(float).Module);
Console.WriteLine(typeof(double).IsPublic);
Console.WriteLine(typeof(Point).MemberType);
```



Interfaces

Un interfaz define un **contrato** semántico que ha de respetar cualquier clase (o struct) que implemente el interfaz.

- La especificación del interfaz puede incluir métodos, propiedades, indexadores y eventos.
- El interfaz no contiene implementación alguna. La clase o struct que implementa el interfaz es la que contiene la implementación de la funcionalidad especificada por el interfaz.
- Los interfaces (como algo separado de la implementación) permiten la existencia del polimorfismo, al poder existir muchas clases o structs que implementen el interfaz.
- Como sucede en Java, se permite **herencia simple de clases** y **herencia múltiple de interfaces**.

El principal uso de las interfaces es indicar que una clase implementa ciertas características. Por ejemplo, el ciclo foreach trabaja internamente comprobando que la clase sobre la que se aplica implementa el interfaz `IEnumerable` y llamando a los métodos definidos en esa interfaz.

```
using System;

namespace Interfacel
{
    class InterfacelApp
    {
        // Definición de una interface

        public interface IDemo
        {
            void MetodoDeInterface ();
        }

        // "Clase1" y "Clase2" implementan la interface

        public class Clase1 : IDemo
        {
            public void MetodoDeInterface()
            {
                Console.WriteLine ("Método de Clase1");
            }
        }

        public class Clase2 : IDemo
        {
            public void MetodoDeInterface()
            {
            }
        }
    }
}
```



```

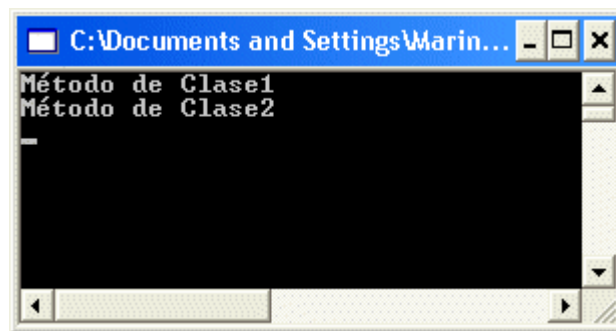
    {
        Console.WriteLine ("Método de Clase2");
    }
}
static void Main(string[] args)
{
    Clase1 c1 = new Clase1();
    Clase2 c2 = new Clase2();
    IDemo demo; // objeto de una interface

    // Ejemplo de polimorfismo
    demo = c1;
    demo.MetodoDeInterface();

    demo = c2;
    demo.MetodoDeInterface();

    Console.ReadLine();
}
}
}

```



Otro ejemplo:

```

using System;

class InterfaceApp
{
    interface IPresentable
    {
        void Presentar();
    }

    class Triangulo : IPresentable
    {
        private double b, a;
        public Triangulo(double Base, double altura)
        {
            this.b=Base;
            this.a=altura;
        }
        public double Base
        {
            get { return b; }
        }
        public double Altura
        {
            get { return a; }
        }
    }
}

```

```

public double Area
{
    get { return Base*Altura/2; }
}
public void Presentar()
{
    Console.WriteLine("Base del triángulo: {0}", Base);
    Console.WriteLine("Altura del triángulo: {0}", Altura);
    Console.WriteLine("Área del triángulo: {0}", Area);
}
}
class Persona : IPresentable
{
    private string nbre, apell, dir;
    public Persona (string nombre, string apellidos,
                    string direccion)
    {
        this.nbre = nombre;
        this.apell = apellidos;
        this.dir = direccion;
    }
    public string Nombre
    {
        get { return nbre; }
    }
    public string Apellidos
    {
        get { return apell; }
    }
    public string Direccion
    {
        get { return dir; }
    }
    public void Presentar()
    {
        Console.WriteLine("Nombre: {0}", Nombre);
        Console.WriteLine("Apellidos: {0}", Apellidos);
        Console.WriteLine("Dirección: {0}", Direccion);
    }
}

static void VerDatos(IPresentable IP)
{
    IP.Presentar();
}

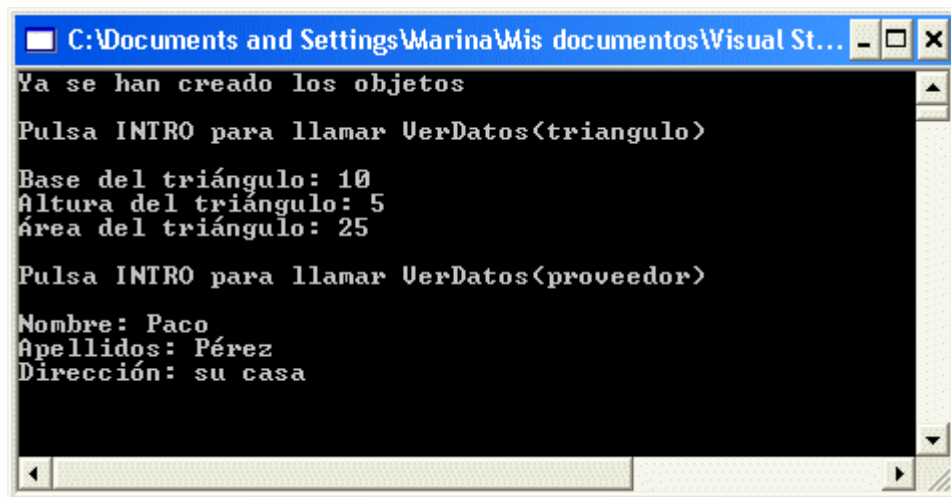
static void Main(string[] args)
{
    Triangulo t=new Triangulo(10,5);
    Persona p=new Persona ("Paco", "Pérez", "su casa");

    Console.WriteLine("Ya se han creado los objetos");
    Console.WriteLine("\nINTRO para VerDatos(triangulo)");
    Console.ReadLine();
    VerDatos(t);

    Console.WriteLine("\nINTRO para VerDatos(proveedor)");
    Console.ReadLine();
    VerDatos(p);

    Console.ReadLine();
}
}

```



Herencia múltiple

La plataforma .NET no permite herencia múltiple de implementación, aunque sí se puede conseguir **herencia múltiple de interfaz**. Clases, structs e interfaces pueden heredar de múltiples interfaces (como en Java).

```
using System;

namespace Interface2
{
    class Interface2App
    {
        // Definición de interfaces

        public interface IDemo1
        {
            void Metodo1DeInterfacel ();
            string Metodo2DeInterfacel ();
        }

        public interface IDemo2
        {
            void Metodo1DeInterface2 ();
        }

        public interface IDemo3 : IDemo1
        {
            void Metodo1DeInterface3 (string mensaje);
        }

        // "Clase1" implementan la interface "IDemo1"

        public class Clase1 : IDemo1
        {
            public void Metodo1DeInterfacel()
            { Console.WriteLine ("Mét1 de Int1 en Clase1"); }

            public string Metodo2DeInterfacel()
            { return ("En Mét2 de Int1 en Clase1"); }
        }

        // "Clase1" implementan las interfaces
        // "IDemo1" e "IDemo2"
    }
}
```

```

public class Clase2 : IDemo1, IDemo2
{
    public void Metodo1DeInterfacel()
    { Console.WriteLine ("Mét1 de Int1 en Clase2"); }

    public string Metodo2DeInterfacel()
    { return ("En Mét2 de Int1 en Clase2"); }

    public void Metodo1DeInterface2()
    { Console.WriteLine ("Mét1 de Int2 en Clase2"); }
}

// "Clase3" implementan la interface "IDemo3", la
//    cual ha heredado de "IDemo1"

public class Clase3 : IDemo3
{
    public void Metodo1DeInterfacel()
    { Console.WriteLine ("Mét1 de Int1 en Clase3"); }

    public string Metodo2DeInterfacel()
    { return ("En Mét2 de Int1 en Clase3"); }

    public void Metodo1DeInterface3 (string m)
    { Console.WriteLine (m + "Mét1 de Int3 en Clase3"); }
}

static void Main(string[] args)
{
    Clase1 c1 = new Clase1();
    Clase2 c2 = new Clase2();
    Clase3 c3 = new Clase3();

    IDemo1 i1;
    IDemo2 i2;
    IDemo3 i3;

    i1 = c1;
    Console.WriteLine("Cuando i1 = c1 ");
    i1.Metodo1DeInterfacel();
    Console.WriteLine(i1.Metodo2DeInterfacel());
    Console.WriteLine();

    i1 = c3;
    Console.WriteLine("Cuando i1 = c3 ");
    i1.Metodo1DeInterfacel();
    Console.WriteLine(i1.Metodo2DeInterfacel());
    Console.WriteLine();

    i3 = c3;
    Console.WriteLine("Cuando i3 = c3 ");
    i3.Metodo1DeInterfacel();
    Console.WriteLine(i3.Metodo2DeInterfacel());
    i3.Metodo1DeInterface3("Aplicado a i3: ");
    Console.WriteLine();

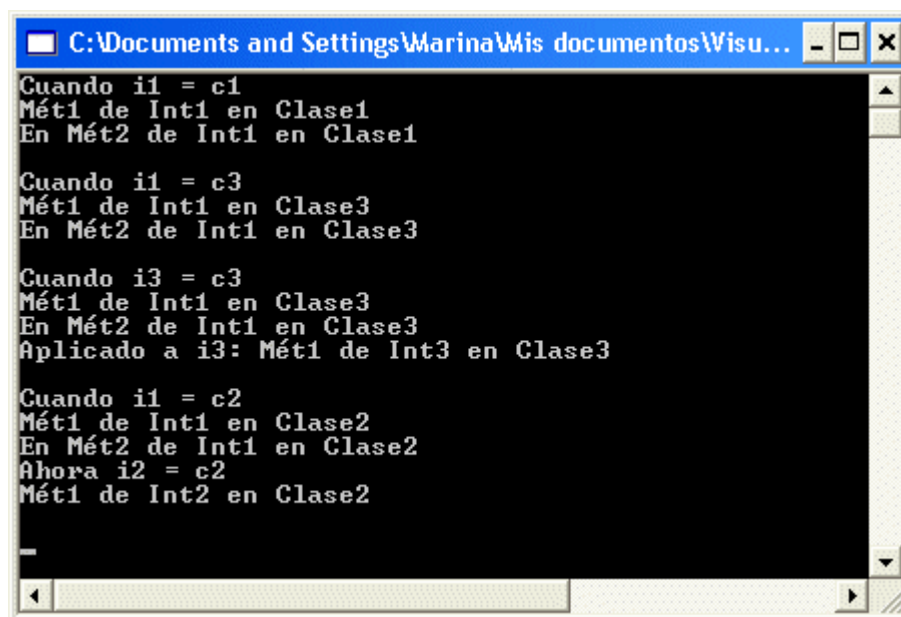
    i1 = c2;
    Console.WriteLine("Cuando i1 = c2 ");
    i1.Metodo1DeInterfacel();
    Console.WriteLine(i1.Metodo2DeInterfacel());
    i2 = c2;
    Console.WriteLine("Ahora i2 = c2 ");
    i2.Metodo1DeInterface2();
    Console.WriteLine();
}

```

```

        Console.ReadLine();
    }
}

```



Resolución de conflictos de nombres

Si dos interfaces tienen un método con el mismo nombre, se **especifica** explícitamente el interfaz al que corresponde la llamada al método para eliminar ambigüedades

```

interface IControl
{
    void Delete();
}

interface IListBox: IControl
{
    void Delete();
}

interface IComboBox: ITextBox, IListBox
{
    void IControl.Delete();
    void IListBox.Delete();
}

```

Delegados

Un **delegado** es un tipo especial de clase que define la signatura de un método. Su función es similar a la de los punteros a funciones en lenguajes como C y C++ (C# no soporta punteros a funciones).

Los delegados pueden pasarse a métodos y pueden usarse para llamar a los métodos de los que contienen referencias.

Los delegados proporcionan polimorfismo para las llamadas a **funciones**:

```
using System;

class DelegatelApp
{
    // Declaración del "tipo delegado" llamado
    // "Del": funciones que devuelven un double
    // y reciben un double.
    delegate double Del (double x);

    static double incremento (double x)
    { return (++x); }

    static void Main(string[] args)
    {
        // Instanciación
        del1 = new Del (Math.Sin);
        del2 = new Del (Math.Cos);
        del3 = new Del (incremento);

        // Llamadas
        Console.WriteLine (del1(0)); // 0
        Console.WriteLine (del2(0)); // 1
        Console.WriteLine (del3(10)); // 11

        Console.ReadLine();
    }
}
```

Para hacer más evidente el polimorfismo el método `Main` podría escribirse como:

```
static void Main(string[] args)
{
    Del f1 = new Del (Math.Sin);
    Del f2 = new Del (Math.Cos);
    Del f3 = new Del (incremento);

    Del f;
    f = f1; Console.WriteLine (f(0)); // 0
    f = f2; Console.WriteLine (f(0)); // 1
    f = f3; Console.WriteLine (f(10)); // 11
}
```

o bien así:

```
static void Main(string[] args)
{
    Del[] d = new Del[3];

    d[0] = new Del (Math.Sin);
    d[1] = new Del (Math.Cos);
    d[2] = new Del (incremento);

    for (int i=0; i<2; i++)
        Console.WriteLine (d[i](0)); // 0, 1
    Console.WriteLine (d[2](10)); // 11
}
```

Los delegados son muy útiles ya que permiten disponer de objetos cuyos métodos puedan ser modificados dinámicamente durante la ejecución de un programa.

En general, son útiles en todos aquellos casos en que interese **pasar** métodos como parámetros de otros métodos.

Los **delegados** son la base sobre la que se monta la gestión de eventos en la plataforma .NET.

Multicasting

Un delegado es un tipo especial de clase cuyos objetos pueden almacenar referencias a uno o más métodos, de tal manera que a través del objeto sea posible solicitar la ejecución en cadena de todos ellos.

Un delegado puede contener e invocar múltiples métodos. De esta forma se puede hacer multicasting de una forma sencilla y elegante. Para que un delegado pueda contener varios métodos, éstos no pueden devolver ningún valor (si lo intentasen devolver, se generaría una excepción en tiempo de ejecución).

```
using System;

class Delegate2App
{
    delegate void SomeEvent (int x, int y);

    static void Func1(int x, int y)
    { Console.WriteLine(" Desde Func1"); }

    static void Func2(int x, int y)
    { Console.WriteLine(" Desde Func2"); }

    static void Main(string[] args)
    {
        SomeEvent func = new SomeEvent(Func1);

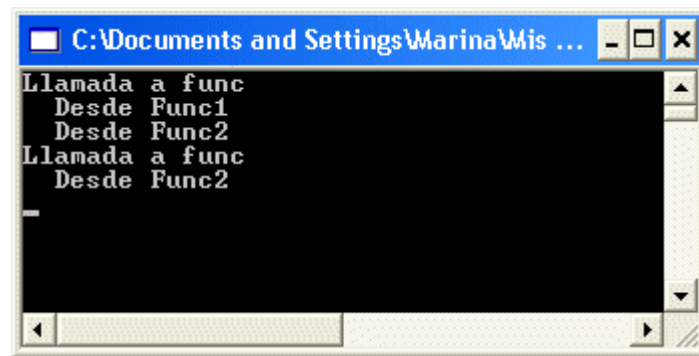
        func += new SomeEvent(Func2);

        Console.WriteLine("Llamada a func");
        func(1,2); // Se llama tanto a Func1 como a Func2

        func -= new SomeEvent(Func1);

        Console.WriteLine("Llamada a func");
        func(2,3); // Sólo se llama a Func2

        Console.ReadLine();
    }
}
```



Cada delegado tiene una lista ordenada de métodos que se invocan secuencialmente (en el mismo orden en el que fueron añadidos al delegado). Los operadores += y -= se utilizan para añadir y eliminar métodos de la lista asociada a un delegado.

Delegados vs. interfaces

Siempre se pueden utilizar interfaces en vez de delegados. Los interfaces son más versátiles, ya que pueden encapsular varios métodos y permiten herencia, si bien los delegados resultan más adecuados para implementar manejadores de eventos. Con los delegados se escribe menos código y se pueden implementar fácilmente múltiples manejadores de eventos en una única clase.

Eventos

Muchas aplicaciones actuales se programan en función de eventos. Cuando se produce algún hecho de interés para nuestra aplicación, éste se notifica mediante la generación de un evento, el cual será procesado por el manejador de eventos correspondiente (modelo "publish-subscribe"). Los eventos nos permiten enlazar código personalizado a componentes creados previamente (mecanismo de "callback").

El *callback* consiste en que un cliente notifica a un servidor que desea ser informado cuando alguna acción tenga lugar. C# usa los eventos de la misma manera que Visual Basic usa los mensajes.

Las aplicaciones en Windows se programan utilizando eventos, pues los eventos resultan especialmente indicados para la implementación de interfaces interactivos. Cuando el usuario hace algo (pulsar una tecla, hacer click con el ratón, seleccionar un dato de una lista...), el programa reacciona en función de la acción del usuario.

El uso de eventos, no obstante, no está limitado a la implementación de interfaces. También son útiles en el desarrollo de aplicaciones que deban realizar operaciones periódicamente o realizar operaciones de forma asíncrona (p.ej. llegada de un correo electrónico, terminación de una operación larga...).

El lenguaje C# da soporte a los eventos mediante el uso de delegados. Al escribir nuestra aplicación, un evento no será más que un campo que almacena un delegado. Los usuarios de la clase podrán registrar delegados (mediante los operadores += y -=), pero no podrán invocar directamente al delegado.

Eventos en C#

```
public delegate void EventHandler ( object sender, EventArgs e);

public class Button
{
    public event EventHandler Click;

    protected void OnClick (EventArgs e)
    {
        // This is called when button is clicked
        if (Click != null) Click(this, e);
    }
}

public class MyForm: Form
{
    Button okButton;

    static void OkClicked(object sender, EventArgs e)
    {
        ShowMessage("You pressed the OK button");
    }

    public MyForm()
    {
        okButton = new Button(...);
        okButton.Caption = "OK";
        okButton.Click += new EventHandler(OkClicked);
    }
}
```

Similares a las de C++, si bien:

- `if`, `while` y `do` requieren expresiones booleanas.
- `goto` no puede saltar dentro de un bloque (da igual, de todas formas no lo usaremos NUNCA).
- `switch` funciona como en Pascal (no como en C).
- Se añade una nueva estructura de control iterativa: `foreach`.

Estructuras condicionales

if

```
int Test(int a, int b)
{
    if (a > b)
        return 1;
    else
        if (a < b)
            return -1;
        else
            return 0;
}
```

switch

Funcionan sobre cualquier tipo predefinido (incluyendo `string`) o enumerado (`enum`) y debe indicar explícitamente cómo terminar cada caso (generalmente, con `break`, `goto case`, `throw` o `return`):

```
using System;

class HolaMundoSwitch
{
    public static void Main(String[] args)
    {
        if (args.Length > 0)
            switch(args[0])
            {
                case "José":
                    Console.WriteLine("Hola José. Buenos días");
                    break;
                case "Paco":
                    Console.WriteLine("Hola Paco. Me alegro de verte");
                    break;
                default: Console.WriteLine("Hola {0}", args[0]);
                    break;
            }
        else
            Console.WriteLine("Hola Mundo");
    }
}

static int Test(string label)
{
    int result;

    switch(label) {
        case null:
            goto case "runner-up";
        case "fastest":
        case "winner":
            result = 1; break;
        case "runner-up":
            result = 2; break;
        default:
            result = 0;
    }
    return result;
}

.....
Console.WriteLine (Test(""));           // 0
Console.WriteLine (Test("runner-up")); // 2
Console.WriteLine (Test("winner"));    // 1
Console.WriteLine (Test("fastest"));   // 1
Console.WriteLine (Test("unknown"));   // 0
```

Estructuras repetitivas

while

```
int i = 0;

while (i < 5) {
    ...
    i++;
}
```

do...while

```
int i = 0;

do {
    ...
    i++;
} while (i < 5);
```

for

```
int i;

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

foreach

```
public static void Main(string[] args)
{
    foreach (string s in args)
        Console.WriteLine(s);
}
```

También se aplica sobre colecciones que implementen el interfaz `IEnumerable`:

```
foreach (Customer c in customers.OrderBy("name")) {
    if (c.Orders.Count != 0) {
        ...
    }
}
```

Saltos

goto

Aunque el lenguaje lo permita, nunca escribiremos algo como lo que aparece a continuación:

```
static void Find(int value, int[,] values,
                out int row, out int col)
{
    int i, j;
    for (i = 0; i < values.GetLength(0); i++)
        for (j = 0; j < values.GetLength(1); j++)
            if (values[i, j] == value) goto found;
    throw new InvalidOperationException("Not found");
found:
    row = i; col = j;
}
```

break

Lo usaremos únicamente en sentencias `switch`.

continue

Mejor no lo usamos.

return

Procuraremos emplearlo sólo al final de un método para facilitar la legibilidad del código.

Excepciones

El mecanismo de control de excepciones es muy parecido al de C++.

Las excepciones son un mecanismo de los lenguajes de programación para tratar situaciones anómalas (generalmente errores inesperados) de una forma sencilla y elegante, si bien no conviene abusar de ellas.

Las excepciones pueden generarse en un proceso o hebra de nuestra aplicación (con la sentencia `throw`) o pueden provenir del entorno de ejecución de la plataforma .NET.

Son mejores que las sentencias `return` porque no pueden ser ignoradas y no tienen por qué tratarse en el punto en que se producen.

Exactamente igual que en C++ y en Java:

- La sentencia `throw` lanza una excepción (una instancia de una clase derivada de `System.Exception`, que contiene información sobre la excepción: `Message`, `StackTrace`, `HelpLink`, `InnerException`...).
- El bloque `try` delimita código que podría generar una excepción.
- El bloque `catch` indica cómo se manejan las excepciones. Se puede relanzar la excepción capturada o crear una nueva si fuese necesario. Se pueden especificar distintos bloques `catch` para capturar distintos tipos de excepciones. En ese caso, es recomendable poner primero los más específicos (para asegurarnos de que capturamos la excepción concreta).
- El bloque `finally` incluye código que siempre se ejecutará (se produzca o no una excepción).

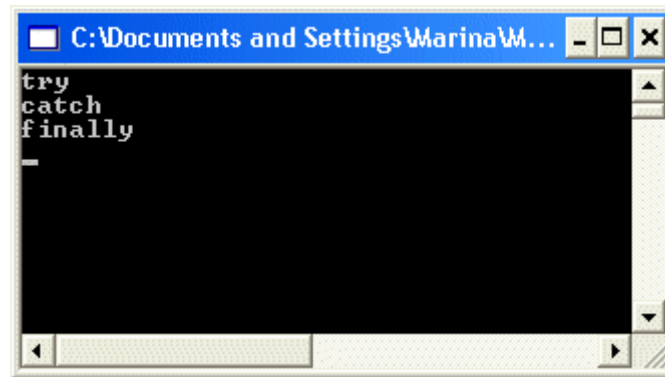
Ejemplo 1

```
try {  
    Console.WriteLine("try");  
    throw new Exception("message");  
}  
catch (ArgumentNullException e) {
```

```

    Console.WriteLine("Null argument");
}
catch {
    Console.WriteLine("catch");
}
finally {
    Console.WriteLine("finally");
}

```



Ejemplo 2

```

static void Main(string[] args)
{
    int numerador = 10;
    Console.WriteLine ("Numerador es = {0}", numerador);
    Console.Write ("Denominador = ");
    string strDen = Console.ReadLine();

    int denominador, cociente;

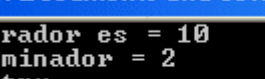
    try
    {
        Console.WriteLine("--> try");
        denominador = Convert.ToInt16(strDen);
        cociente = numerador / denominador;
        Console.WriteLine ("Cociente = {0}", cociente);
    }

    catch (ArithmeticException e)
    {
        Console.WriteLine("--> catch");
        Console.WriteLine("Excep. aritmética");
        Console.WriteLine("ArithmeticException Handler: {0}", e.ToString());
    }
    catch (ArgumentNullException e)
    {
        Console.WriteLine("--> catch");
        Console.WriteLine("Excep. de argumento nulo");
        Console.WriteLine("ArgumentNullException Handler: {0}", e.ToString());
    }
    catch (Exception e)
    {
        Console.WriteLine("--> catch");
        Console.WriteLine("generic Handler: {0}", e.ToString());
    }
    finally
    {
        Console.WriteLine("--> finally");
    }
}

```

```
        Console.ReadLine();
    }
}
```

Cuando toda funciona sin problemas:

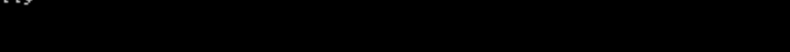


A screenshot of a Windows command prompt window. The title bar at the top reads "C:\Documents and Settings\Marina\Mis ...". The window contains the following text:

```
Numerador es = 10
Denominador = 2
--> try
Cociente = 5
--> finally
-
```

The text is displayed in a monospaced font on a black background. The window has standard Windows XP-style window controls (minimize, maximize, close) in the title bar.

Cuando se intenta dividir por cero:



The screenshot shows a Visual Studio console window with a blue title bar that reads "C:\Documents and Settings\Administrador\Mis documentos\Visual...". The console output is as follows:

```
Numerador es = 10
Denominador = 0
--> try
--> catch
Excep. aritmética
ArithmeticException Handler: System.DivideByZeroException: Intento de dividir por
cero.
   at Ex1App.Main(String[] args) in c:\documents and settings\administrador\mis
documentos\visual studio projects\holamundo\holamundo\ex1.cs:line 17
--> finally
```

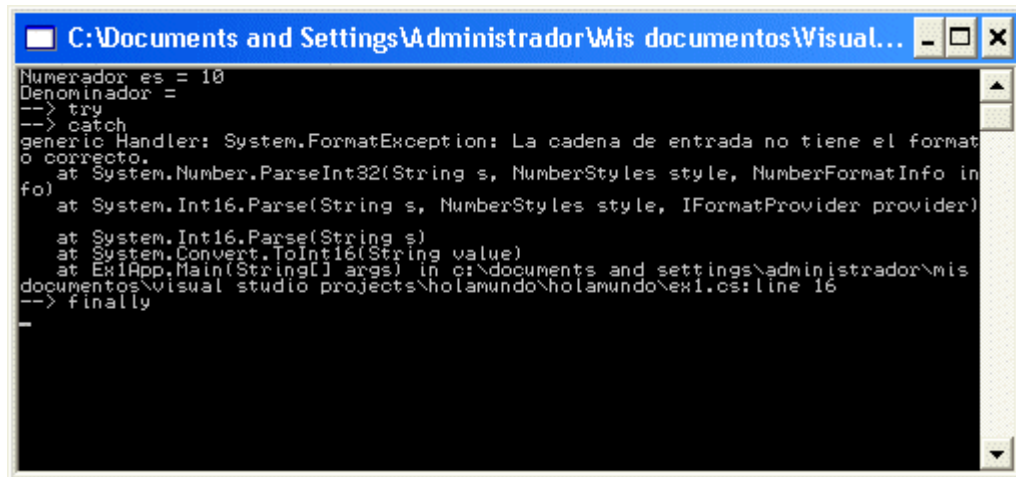
Cuando se produce desbordamiento:

```

C:\Documents and Settings\Administrador\Mis documentos\Visual...
Numerador es = 10
Denominador = 9999999999999999999999999999999
--> try
--> catch
Excep. aritmética
ArithmeticException Handler: System.OverflowException: Valor demasiado grande o
demasiado pequeño para Int32.
   at System.Number.ParseInt32(String s, NumberStyles style, NumberFormatInfo in
fo)
       at System.Int16.Parse(String s, NumberStyles style, IFormatProvider provider)
           at System.Int16.Parse(String s)
               at System.Convert.ToInt16(String value)
                   at ExiApp.Main(String[] args) in c:\documents and settings\admini
strador\mis
documentos\visual studio projects\holamundo\holamundo\ex1.cs:line 16
--> finally
-

```

Cuando se produce otro problema (cadena vacía, por ejemplo):



Ejemplo 3

Hemos visto que pueden conocerse los detalles de la excepción que se haya producido. Podemos conocer más detalles mostrando cierta información de la pila:

```

using System;
using System.Diagnostics;

class ExcepApp
{
    static void Main(string[] args)
    {
        int numerador = 10;
        Console.WriteLine ("Numerador es = {0}", numerador);

        Console.Write ("Denominador = ");
        string strDen = Console.ReadLine();

        int denominador, cociente;

        try
        {
            Console.WriteLine("--> try");
            denominador = Convert.ToInt16(strDen);
            cociente = numerador / denominador;
            Console.WriteLine ("Cociente = {0}", cociente);
        }

        catch (Exception e)
        {
            Console.WriteLine("--> catch");
            Console.WriteLine("Generic Handler: {0}", e.ToString());
            Console.WriteLine();

            StackTrace st = new StackTrace(e, true);

            Console.WriteLine("Traza de la pila:");
            for (int i = 0; i < st.FrameCount; i++) {
                StackFrame sf = st.GetFrame(i);
                Console.WriteLine("  Pila de llamadas, Método: {0}",
                    sf.GetMethod() );
                Console.WriteLine("  Pila de llamadas, Línea : {0}",
                    sf.GetFileLineNumber());
            }
        }
    }
}

```

```

        Console.WriteLine();
    }

    finally
    {
        Console.WriteLine("--> finally");
    }

    Console.ReadLine();
}
}

```

```

C:\Documents and Settings\Administrador\Mis documentos\Visual...
Numerador es = 10
Denominador = 0
--> try
--> catch
Generic Handler: System.DivideByZeroException: Intento de dividir por cero.
   at Ex1App.Main(String[] args) in c:\documents and settings\administrador\mis documentos\visual studio projects\holamundo\holamundo\ex1.cs:line 17
Traza de la pila:
  Pila de llamadas, Método: Void Main(System.String[])
  Pila de llamadas, Línea : 17
--> finally
-

```

```

C:\Documents and Settings\Administrador\Mis documentos\Visual...
Numerador es = 10
Denominador =
--> try
--> catch
Generic Handler: System.FormatException: La cadena de entrada no tiene el formato correcto.
   at System.Number.ParseInt32(String s, NumberStyles style, NumberFormatInfo info)
   at System.Int16.Parse(String s, NumberStyles style, IFormatProvider provider)
   at System.Int16.Parse(String s)
   at System.Convert.ToInt16(String value)
   at Ex1App.Main(String[] args) in c:\documents and settings\administrador\mis documentos\visual studio projects\holamundo\holamundo\ex1.cs:line 18
Traza de la pila:
  Pila de llamadas, Método: Int32 ParseInt32(System.String, System.Globalization.NumberStyles, System.Globalization.NumberFormatInfo)
  Pila de llamadas, Línea : 0
  Pila de llamadas, Método: Int16 Parse(System.String, System.Globalization.NumberStyles, System.IFormatProvider)
  Pila de llamadas, Línea : 0
  Pila de llamadas, Método: Int16 Parse(System.String)
  Pila de llamadas, Línea : 0
  Pila de llamadas, Método: Int16 ToInt16(System.String)
  Pila de llamadas, Línea : 0
  Pila de llamadas, Método: Void Main(System.String[])
  Pila de llamadas, Línea : 18
--> finally
-

```

Un programa en C# no es más que un conjunto de tipos...

Organización lógica de los tipos

Del mismo modo que los ficheros se organizan en directorios, los tipos de datos se organizan en **espacios de nombres** (del inglés, *namespaces*).

Por un lado estos espacios permiten tener más organizados los tipos de datos, lo que facilita su localización. Así es como está organizada la BCL: todas las clases más comúnmente usadas en cualquier aplicación pertenecen al espacio de nombres llamado `System`, las de acceso a bases de datos en `System.Data`, las de realización de operaciones de entrada/salida en `System.IO`, etc.

Por otro lado, los espacios de nombres también permiten poder usar en un mismo programa varias clases con igual nombre si pertenecen a espacios diferentes.

En definitiva: **Los espacios de nombres proporcionan una forma unívoca de identificar un tipo.** Eliminan cualquier tipo de ambigüedad en los nombres de los símbolos empleados en un programa.

Los espacios de nombres se pueden anidar.

No existe relación entre espacios de nombres y ficheros (a diferencia de Java).

```
namespace N1 {           // N1
    class C1 {           // N1.C1
        class C2 {       // N1.C1.C2
        }
    }
    namespace N2 {       // N1.N2
        class C2 {       // N1.N2.C2
        }
    }
}
```

Sentencia using

La sentencia `using` permite utilizar tipos sin teclear su nombre completo:

```
using N1;

C1 a;           // N1 es implícito
N1.C1 b;        // Tipo totalmente cualificado

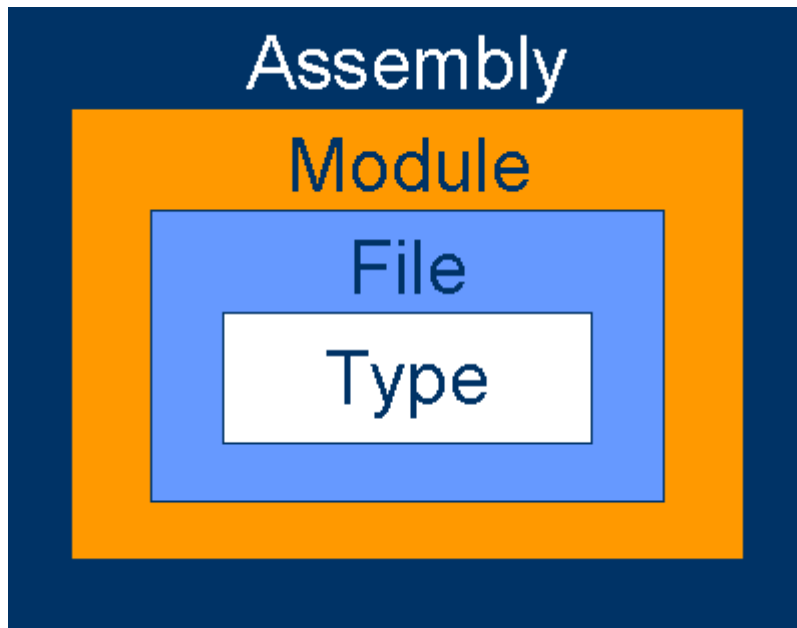
C2 c;           // ¡Error! C2 no está definida en N1
C1.C2 e;        // Así sí es correcto (N1.C1.C2)
N1.N2.C2 d;     // La otra clase llamada C2
```

También permite crear alias (aunque eso no sea demasiado recomendable):

```
using C1 = N1.N2.C1;
using N2 = N1.N2;

C1 a;           // O sea, N1.N2.C1
N2.C1 b;        // O sea, N1.N2.C1
```

Organización física de los tipos



Los tipos se definen en ficheros

- Un fichero puede contener múltiples tipos.
- Cada tipo está definido en un único fichero (declaración y definición coinciden).
- No existen dependencias de orden entre los tipos.

Los ficheros se compilan en módulos

- Un módulo es un fichero DLL o EXE
- Un módulo puede contener múltiples ficheros fuente

Los módulos se agrupan en assemblies

- Un assembly puede contener múltiples módulos

Referencias

En Visual Studio se utilizan referencias para identificar assemblies particulares, p.ej. compilador de C#

```
csc HelloWorld.cs /reference:System.Windows.dll
```

Los espacios de nombres son una construcción del lenguaje para abreviar nombres, mientras que las referencias son las que especifican qué assembly utilizar.

Ejecución de aplicaciones

El método Main

La ejecución comienza en el método `static Main`. En un assembly, sólo puede existir un método con una de las siguientes signaturas, todas ellas `static`:

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

Gestión de memoria

C# utiliza un recolector de basura para gestionar automáticamente la memoria, lo que elimina quebraderos de cabeza y una de las fuentes más comunes de error pero conlleva una finalización no determinística (no se ofrece ninguna garantía respecto a cuándo se llama a un destructor, ni siquiera podemos afirmar que llegue a llamarse el destructor).

Los objetos que deban eliminarse tras ser utilizados deberían implementar la interfaz `System.IDisposable` (escribiendo en el método `Dispose` todo aquello que haya de realizarse para liberar un objeto). El método `Dispose` siempre se invoca al terminar una sentencia `using`:

```
public class MyResource : IDisposable
{
    public void MyResource()
    {
        // Acquire valuable resource
    }

    public void Dispose()
    {
        // Release valuable resource
    }

    public void DoSomething()
    {
        ...
    }
}

using (MyResource r = new MyResource())
{
    r.DoSomething();
} // se llama a r.Dispose()
```

Código no seguro

En ocasiones necesitamos tener un control total sobre la ejecución de nuestro código (cuestiones de rendimiento, compatibilidad con código existente, uso de DLLs...), por lo que C# nos da la posibilidad de marcar fragmentos de código como código no seguro (`unsafe`) y así poder emplear C/C++ de forma nativa: punteros, aritmética de punteros, operadores `->` y `*`, ... sin recolección de basura. La instrucción `stackalloc` reserva memoria en la pila de manera similar a como `malloc` lo hace en C o `new` en C++.

```
public unsafe void MiMetodo () // Método no seguro
```

```

{ ... }

unsafe class MiClase // Clase (struct) no segura
{ ... } // Todos los miembros son no seguros

struct MiStruct
{
    private unsafe int * pX; // Campo de tipo puntero no seguro
    ...
}

unsafe
{
    // Instrucciones que usan punteros
}

```

En caso de que la compilación se vaya a realizar a través de Visual Studio .NET, la forma de indicar que se desea compilar código inseguro es activando la casilla **Proyecto | Propiedades de (proyecto) | Propiedades de Configuración | Generar | Permitir bloques de código no seguro | True**.

```

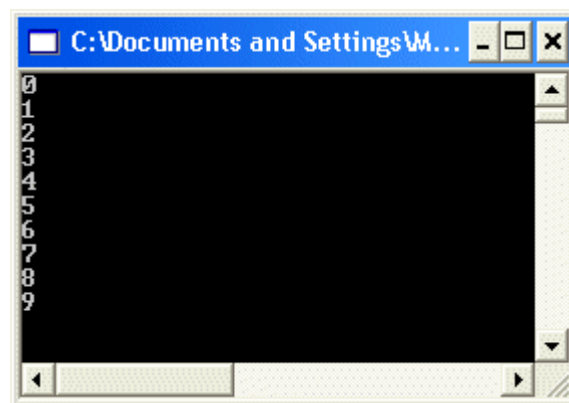
class StackallocApp
{
    public unsafe static void Main()
    {
        const int TAM = 10;
        int * pt = stackalloc int[TAM];

        for (int i=0; i<TAM; i++) pt[i] = i;

        for(int i=0; i<TAM; i++)
            System.Console.WriteLine(pt[i]);

        Console.ReadLine ();
    }
}

```



Para asegurarnos de que el recolector de basura no mueva nuestros datos tendremos que utilizar la sentencia `fixed`. El recolector puede mover los datos de tipo referencia, por lo que si un puntero contiene la dirección de un dato de tipo referencia podría apuntar a una dirección incorrecta después de que el recolector de basura trabajara. Si un conjunto de instrucciones se encierra en un bloque `fixed` se previene al recolector de basura para que no mueva el objeto al que se referencia mientras dura el bloque `fixed`.

Esta capacidad tiene su coste: al emplear punteros, el código resultante es inseguro ya que éste no se puede verificar. De modo que tendremos que extremar las precauciones si alguna vez tenemos que usar esta capacidad del lenguaje C#.

Atributos

Un **atributo** es **información** que se puede añadir a los metadatos de un módulo de código. Los atributos nos permiten "decorar" un elemento de nuestro código con información adicional.

C# es un lenguaje imperativo, pero, como todos los lenguajes de esta categoría, contiene algunos elementos declarativos. Por ejemplo, la accesibilidad de un método de una clase se especifica mediante su declaración como `public`, `protected`, `private` o `internal`. C# generaliza esta capacidad permitiendo a los programadores inventar nuevas formas de información declarativa, anexas a distintas entidades del programa y recuperarlas en tiempo de ejecución. Los programas especifican esta información declarativa adicional mediante la definición y el uso de atributos.

Esta información puede ser referente tanto al propio módulo o el ensamblado al que pertenece, como a los tipos de datos definidos en él, sus miembros, los parámetros de sus métodos, los bloques `set` y `get` de sus propiedades e indexadores o los bloques `add` y `remove` de sus eventos. Se pueden emplear en ensamblados, módulos, tipos, miembros, valores de retorno y parámetros.

Declarar una clase atributo

Declarar un atributo en C# es simple: se utiliza la forma de una declaración de clase que hereda de `System.Attribute` y que se ha marcado con el atributo `AttributeUsage`, como se indica a continuación:

```
// La clase HelpAttribute posee un parámetro posicional (url)
// de tipo string y un parámetro con nombre -opcional- (Topic)
// de tipo string.

[AttributeUsage(AttributeTargets.All)]
public class HelpAttribute: Attribute
{
    public string Topic = null;
    private string url;

    public HelpAttribute(string url)
    {
        this.url = url;
    }
    public string Url {
        get { return url; }
    }
    public override string ToString() {
        string s1 = "    Url = " + this.Url;
        string dev = (this.Topic != null) ?
            (s1 + " - Topic: " + this.Topic) : s1;
        return (dev);
    }
} // class HelpAttribute
```

- El atributo `AttributeUsage` especifica los elementos del lenguaje a los que se puede aplicar el atributo.
- Las clases de atributos son clases públicas derivadas de `System.Attribute` que disponen al menos de un constructor público.
- Las clases de atributos tienen dos tipos de parámetros:
 - *Parámetros posicionales*, que se deben especificar cada vez que se utiliza el atributo. Los parámetros posicionales se especifican como argumentos de constructor para la clase de atributo. En el ejemplo anterior, `url` es un parámetro posicional.
 - *Parámetros con nombre*, los cuales son opcionales. Si se especifican al usar el atributo, debe utilizarse el nombre del parámetro. Los parámetros con nombre se definen mediante un campo o una propiedad no estáticos. En el ejemplo anterior, `Topic` es un parámetro con nombre.
- Los parámetros de un atributo sólo pueden ser valores constantes de los siguientes tipos:
 - Tipos simples (*bool*, *byte*, *char*, *short*, *int*, *long*, *float* y *double*)
 - *string*
 - *System.Type*
 - *enumeraciones*
 - *object* (El argumento para un parámetro de atributo del tipo *object* debe ser un valor constante de uno de los tipos anteriores.)
 - Matrices *unidimensionales* de cualquiera de los tipos anteriores

Parámetros para el atributo `AttributeUsage`

El atributo `AttributeUsage` proporciona el mecanismo subyacente mediante el cual los atributos se declaran.

`AttributeUsage` tiene un parámetro posicional:

- `AllowOn`, que especifica los elementos de programa a los que se puede asignar el atributo (clase, método, propiedad, parámetro, etc.). Los valores aceptados para este parámetro se pueden encontrar en la enumeración `System.Attributes.AttributeTargets` de .NET Framework. **El valor predeterminado para este parámetro es el de todos los elementos del programa** (`AttributeElements.All`).

`AttributeUsage` tiene un parámetro con nombre:

- `AllowMultiple`, valor booleano que indica si se pueden especificar varios atributos para un elemento de programa. El valor predeterminado para este parámetro es `False`.

Utilizar una clase atributo

A continuación, se muestra un breve ejemplo de uso del atributo declarado en la sección anterior:

```
[Help("http://decsai.ugr.es/Clase1.htm")]
class Clase1
{
    /* Bla, bla, bla... */
}

[Help("http://decsai.ugr.es/Clase2.htm", Topic="Atributos")]
class Clase2
{
    /* Bla, bla, bla... */
}
```

En este ejemplo, el atributo `HelpAttribute` está asociado con las clases `Clase1` y `Clase2`.

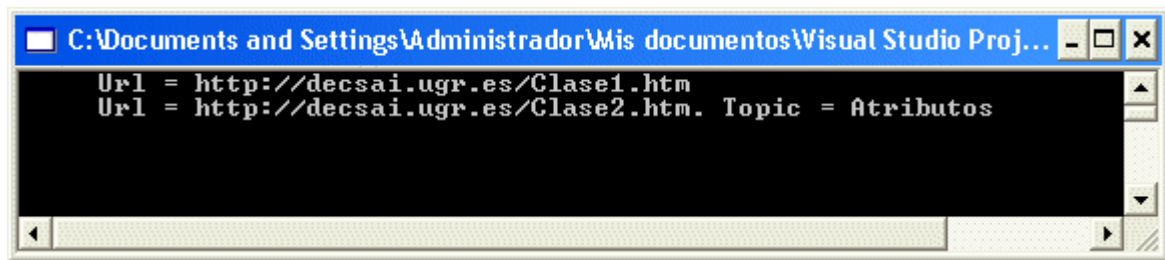
Nota: Por convención, todos los nombres de atributo finalizan con la palabra "Attribute" para distinguirlos de otros elementos de .NET Framework. No obstante, no tiene que especificar el sufijo de atributo cuando utiliza atributos en el código (véase el ejemplo).

Acceder a los atributos por reflexión

Los atributos de un tipo o de un miembro de un tipo pueden ser examinados en tiempo de ejecución (**reflexión**), heredan de la clase `System.Attribute` y sus argumentos se comprueban en tiempo de compilación.

Los principales métodos de reflexión para consultar atributos se encuentran en la clase `System.Reflection.MemberInfo`. El método clave es `GetCustomAttributes`, que devuelve un vector de objetos que son equivalentes, en tiempo de ejecución, a los atributos del código fuente. El siguiente ejemplo muestra la manera básica de utilizar la reflexión para obtener acceso a los atributos:

```
class AtributosSimpleApp
{
    static void Main(string[] args)
    {
        MemberInfo info1 = typeof(Clase1);
        object[] attributes1 = info1.GetCustomAttributes(true);
        for (int i = 0; i < attributes1.Length; i++) {
            System.Console.WriteLine(attributes1[i]);
        }
        MemberInfo info2 = typeof(Clase2);
        object[] attributes2 = info2.GetCustomAttributes(true);
        for (int i = 0; i < attributes2.Length; i++) {
            System.Console.WriteLine(attributes2[i]);
        }
        Console.ReadLine();
    } // Main ()
} // class AtributosSimpleApp
```



Atributos predefinidos

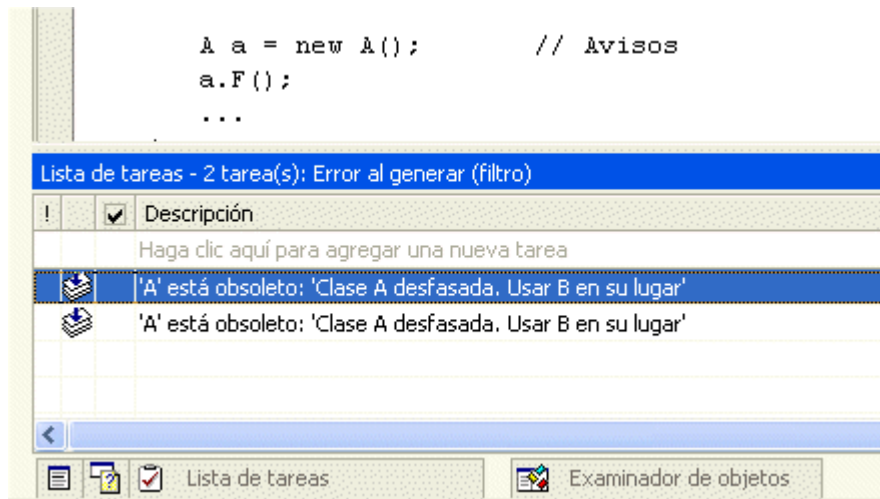
Si bien el programador puede definir cuantos atributos considere necesarios, algunos atributos ya están predefinidos en la plataforma .NET.

Atributo	Descripción
Browsable	Propiedades y eventos que deben mostrarse en el inspector de objetos.
Serializable	Clases y estructuras que pueden "serializarse" (esto es, volcarse en algún dispositivo de salida, p.ej. disco), como en Java.
Obsolete	El compilador se quejará si alguien los utiliza (deprecated en Java).
ProgId	COM Prog ID
Transaction	Características transaccionales de una clase.

Observar como al marcar como obsoleta la clase A se genera un error al compilar el módulo ya que se emplea en la línea comentada.

```
[Obsolete("Clase A desfasada. Usar B en su lugar")]
class A {
    public void F() {}
}
class B {
    public void F() {}
}

class SimpleAtrPredefApp
{
    static void Main(string[] args)
    {
        A a = new A();          // Avisos
        a.F();
        ...
    }
}
```

Preprocesador

C# proporciona una serie de directivas de preprocesamiento con distintas funciones. Aunque se le sigue llamando preprocesador (como en C o C++), el preprocesador no es independiente del compilador y se han eliminado algunas directivas como `#include` (para mejorar los tiempos de compilación, se utiliza el esquema de lenguajes como Java o Delphi en lugar de los ficheros de cabecera típicos de C/C++) o las macros de `#define` (para mejorar la claridad del código).

Directiva	Descripción
<code>#define, #undef</code>	Definición de símbolos para la compilación condicional.
<code>#if, #elif, #else, #endif</code>	Compilación condicional.
<code>#error, #warning</code>	Emisión de errores y avisos.
<code>#region, #end</code>	Delimitación de regiones.
<code>#line</code>	Especificación de números de línea.

Aserciones

Las aserciones nos permiten mejorar la calidad de nuestro código. Esencialmente, las aserciones no son más que pruebas de unidad que están incluidas en el propio código fuente. Las aserciones nos permiten comprobar precondiciones, postcondiciones e invariantes. Las aserciones sólo se habilitan cuando se compila el código para depurarlo (debug builds), de forma que su correcto funcionamiento se compruebe continuamente. Cuando distribuyamos nuestras aplicaciones, las aserciones se eliminan para no empeorar la eficiencia de nuestro código (release builds).

Compilación condicional: Aserciones

```
#define Debug
```

```

public class Debug
{
    [Conditional("Debug")]
    public static void Assert(bool cond, String s)
    {
        if (!cond) {
            throw new AssertionError(s);
        }
    }

    void DoSomething()
    {
        ...
        // If Debug is not defined,
        // the next line is not even called
        Assert((x == y), "X should equal Y");
        ...
    }
}

```

Documentación

A los programadores no les suele gustar documentar código, por lo que resulta conveniente suministrar un mecanismo sencillo que les permita mantener su documentación actualizada. Al estilo de doxygen o Javadoc, el compilador de C# es capaz de generarla automáticamente a partir de los comentarios que el programador escriba en los ficheros de código fuente. Los comentarios a partir de los cuales se genera la documentación se escriben en **XML**.

El hecho de que la documentación se genere a partir de los fuentes permite evitar que se tenga que trabajar con dos tipos de documentos por separado (fuentes y documentación) que deban actualizarse simultáneamente para evitar inconsistencias entre ellos derivadas de que evolucionen de manera separada ya sea por pereza o por error.

El compilador genera la documentación en XML con la idea de que sea fácilmente legible para cualquier aplicación. Para facilitar su legibilidad a humanos bastaría añadirle una hoja de estilo XSL o usar alguna aplicación específica encargada de leerla y mostrarla de una forma más cómoda para humanos.

Los comentarios XML se denotan con una barra triple (///) y nos permiten generar la documentación del código cuando compilamos con la opción `/doc`.

```
csc programa.cs /doc:docum_programa.xml
```

El formato de los comentarios viene definido en un esquema XML, si bien podemos añadir nuestras propias etiquetas para personalizar la documentación de nuestras aplicaciones. Algunas de las etiquetas predefinidas se verifican cuando generamos la documentación (parámetros, excepciones, tipos...).

Estos comentarios han preceder las definiciones de los elementos a documentar. Estos elementos sólo pueden ser definiciones de miembros, ya sean tipos de datos (que son

miembros de espacios de nombres) o miembros de tipos datos, y han de colocarse incluso incluso antes que sus atributos.

Etiqueta XML	Descripción
<summary>	Descripción breve de tipos y miembros.
<remarks>	Descripción detallada de tipos y miembros.
<para>	Delimita párrafos.
<example>	Ejemplo de uso.
<see> <seealso>	Referencias cruzadas. Usa el atributo <code>cref</code>
<c> <code>	Código de ejemplo (verbatim).
<param>	Parámetros de métodos. Usa el atributo <code>name</code> .
<paramref>	Referencia a parámetros de metodos. Usa el atributo <code>name</code> .
<returns>	Valor devuelto por el método.
<exception>	Descripción de Excepciones.
<value>	Descripción de propiedades.
<list>	Generar listas. Usa el atributo (opcional) <code>type</code> .
<item>	Generar listas. Usa el atributo (opcional) <code>type</code> (puede ser: <code>bullet</code> , <code>number</code> o <code>table</code>).
<permission>	Permisos.

Veamos un ejemplo detallado:

```
using System;

namespace Geometria {

    /// <summary>
    /// Clase Punto.
    /// </summary>
    /// <remarks>
    /// Caracteriza a los puntos de un espacio bidimensional.
    /// Tiene múltiples aplicaciones....
    /// </remarks>

    class Punto {

        /// <summary>
        /// Campo que contiene la coordenada X de un punto
        /// </summary>
        /// <remarks>
        /// Es de solo lectura
        /// </remarks>

        public readonly uint X;
```

```

    /// <summary>
    /// Campo que contiene la coordenada Y de un punto
    /// </summary>
    /// <remarks>
    /// Es de solo lectura
    /// </remarks>

    public readonly uint Y;

    /// <summary>
    /// Constructor de la clase
    /// </summary>
    /// <param name="x">Coordenada x</param>
    /// <param name="y">Coordenada y</param>

    public Punto(uint x, uint y) {
        this.X=x;
        this.Y=y;
    }
} // fin de class Punto

/// <summary>
/// Clase Cuadrado. Los objetos de esta clase son polígonos
/// cerrados de cuatro lados de igual longitud y que
/// forman ángulos rectos.
/// </summary>
/// <remarks>
/// Los cuatro vértices pueden numerarse de manera que
/// el vértice 1 es el que tiene los menores valores de
/// las coordenadas X e Y.
///
/// Los demás vértices se numeran a partir de éste recorriendo
/// el cuadrado en sentido antihorario.
/// </remarks>

class Cuadrado {

    /// <summary>
    /// Campo que contiene las coordenadas del vértice 1.
    /// </summary>
    ///
    protected Punto verticel;

    /// <summary>
    /// Campo que contiene la longitud del lado.
    /// </summary>
    protected uint lado;

    /// <summary>
    /// Constructor de la clase.
    /// Construye un cuadrado a partir del vértice 1 y de
    /// la longitud del lado.
    /// </summary>
    /// <param name="vert1">Coordenada del vértice 1</param>
    /// <param name="lado">Longitud del lado</param>
    ///
    public Cuadrado(Punto vert1, uint lado) {
        this.verticel=vert1;
        this.lado=lado;
    }

    /// <summary>
    /// Constructor de la clase.
    /// Construye un cuadrado a partir de los vértices 1 y 3.
    /// </summary>

```

```

    /// <param name="vert1">Coordenada del vértice 1</param>
    /// <param name="vert3">Coordenada del vértice 3</param>
    /// <remarks>
    /// Habría que comprobar si las componentes del vértice 3
    /// son mayores o menores que las del vértice 1.
    /// Vamos a presuponer que las componentes del vértice 3 son
    /// siempre mayores que las del uno.
    /// </remarks>
    public Cuadrado(Punto vert1, Punto vert3) {
        this.verticel=vert1;
        this.lado=(uint) Math.Abs(vert3.X-vert1.X);
    }

    /// <summary>
    /// Propiedad que devuelve el punto que representa a
    /// las coordenadas del vértice 1.
    /// </summary>
    public Punto Verticel {
        get {
            return this.verticel;
        }
    }

    /// <summary>
    /// Propiedad que devuelve el punto que representa a
    /// las coordenadas del vértice 2.
    /// </summary>
    public Punto Vertice2 {
        get {
            Punto p=new Punto(this.verticel.X +
this.lado,this.verticel.Y);
            return p;
        }
    }

    .....

} // Fin de class Cuadrado

} // Fin de namespace Geometria

namespace PruebaGeometria {
    using Geometria;
    class GeometriaApp {
        ....
    }
}

```

Para generar la documentación en Visual Studio .NET seleccionaremos el proyecto en el explorador de soluciones y daremos el nombre del fichero XML que contendrá la documentación: **Ver | Páginas de propiedades | Propiedades de configuración | Generar | Archivo de documentación XML** y darle el nombre: DocumentacionGeometia, por ejemplo.

Para ver el resultado: **Herramientas | Generar páginas Web de comentarios**.
Unos ejemplos:

Documentacion1 - Microsoft Visual C# .NET [diseñar] - Documentacion1

Archivo Edición Ver Proyecto Generar Depurar Herramientas Ventana Ayuda

Debug C:\D

Documentacion1 | Geometria.cs

Code Comment Web Report

Solución | Proyecto

- Geometria
 - Punto
 - Cuadrado
- PruebaGeometria

Geometria.Punto Clase

Clase Punto.

Acceso: Proyecto

Clases base: Object

Miembros	Descripción
X	Campo que contiene la coordenada X de un punto
Y	Campo que contiene la coordenada Y de un punto
Punto	Constructor de la clase

Comentarios:

Caracteriza a los puntos de un espacio bidimensional.
Tiene múltiples aplicaciones....

Code Comment Web Report

Solución | Proyecto

- Geometria
 - Punto
 - Cuadrado
- PruebaGeometria

Geometria.Cuadrado Clase

Clase Cuadrado. Los objetos de esta clase son polígonos cerrados de cuatro lados de igual longitud y que forman ángulos rectos.

Acceso: Proyecto

Clases base: Object

Miembros	Descripción
vertice1	Campo que contiene las coordenadas del vértice 1.
lado	Campo que contiene la longitud del lado.
Cuadrado	Constructor de la clase. Construye un cuadrado a partir del vértice 1 y de la longitud del lado.
Cuadrado	Constructor de la clase. Construye un cuadrado a partir de los vértices 1 y 3.
Vertice1	Propiedad que devuelve el punto que representa a las coordenadas del vértice 1.
Vertice2	Propiedad que devuelve el punto que representa a las coordenadas del vértice 2.
Vertice3	Propiedad que devuelve el punto que representa a las coordenadas del vértice 3.
Vertice4	Propiedad que devuelve el punto que representa a las coordenadas del vértice 4.
Lado	
Perimetro	
Area	

Comentarios:

Los cuatro vértices pueden numerarse de manera que el vértice 1 es el que tiene los menores valores de las coordenadas X e Y. Los demás vértices se numeran a partir de éste recorriendo el cuadrado en sentido antihorario.

Conceptos básicos

- **Objetos, instancias y clases:** Un objeto es una estructura de datos en tiempo de ejecución, formada por uno o más valores (campos) y que sirve como representación de un objeto abstracto. Todo `objeto` es instancia de una clase. Una clase es un tipo abstracto de datos implementado total o parcialmente, que encapsula datos y operaciones. Las clases sirven de módulos y de tipos (o patrones de tipos si son genéricas).
- **Módulo:** Unidad lógica que permite descomponer el software. En programación orientada a objetos, las clases proporcionan la forma básica de módulo. Para facilitar el desarrollo de software y su posible reutilización, las dependencias entre módulos deberían reducirse al máximo para conseguir sistemas débilmente acoplados.
- **Tipo:** Cada objeto tiene un tipo, que describe un conjunto de operaciones con las que están equipados todos los objetos de una misma clase.
- **Interfaz:** Contrato perfectamente definido que especifica `completamente` las condiciones precisas que gobiernan las relaciones entre una clase proveedora y sus clientes (rutinas exportadas). Además, es deseable conocer las precondiciones, postcondiciones e invariantes que sean aplicables.
- **Identidad:** Cada `objeto` (instancia de una clase) tiene una identidad única, independientemente de su contenido actual (los datos almacenados en sus campos).
- **Encapsulación (u ocultación de información):** Capacidad de evitar que ciertos aspectos sean visibles desde el exterior. De esta forma, se ocultan detalles de implementación y el usuario puede emplear `objetos` sin tener que conocer su estructura interna.
- **Herencia:** Los tipos se organizan de forma jerárquica (clases base y derivadas, superclases y subclases). La herencia proporciona un mecanismo `simple` mediante el cual se pueden definir unos tipos en función de otros, a los que añade sus características propias. Hay que distinguir entre herencia de interfaz y herencia de implementación (que aumenta el acoplamiento entre los módulos de un programa).
- **Polimorfismo:** Capacidad de usar un objeto sin saber su tipo exacto. Formalmente, el polimorfismo es la capacidad de que un elemento de código pueda denotar, en tiempo de ejecución, objetos de dos o más tipos distintos.