UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA
CAMPUS D'ALCOI

C#

Jordi Linares Pellicer

unity

Introducción al C#

- **C#** es un lenguaje creado en el año 2000 junto a la plataforma **.NET** de Microsoft
- Hereda lo mejor de C++, Java y Visual Basic
- Es un estándar ECMA e ISO, lo que ha permitido su implementación fuera de Windows y por terceros, como el proyecto **Mono** (ahora de Xamarin), que es la implementación utilizada en sus scripts en **Unity3D**
- Orientado a Objetos, y al trabajo con componentes. Moderno, flexible, seguro, productivo y sometido a una intensa evolución

Introducción al C#

- **.NET** es una plataforma para el desarrollo de aplicaciones "**gestionadas**" o "**administradas**" (managed) a través de un CLR (Common Language Runtime), equivalente a la máquina virtual de Java
- En el caso de **Unity3D**, se hace uso de Mono, la implementación libre de **C#** y .NET, y que permite distribuir las aplicaciones en las diferentes plataformas que permite **Unity3D** (Windows, Mac, Linux, iOS, Android, PS3, Wii, Xbox)
- **Unity** está empezando a utilizar tecnología propia que le permite pasar los scripts a C++ (IL2CPP) y dejar que el compilador nativo finalice la compilación en la plataforma de destino.

Introducción al C#

- No todas las características que permite **C#** están disponibles en **Unity3D**, pero sí las más importantes.
- Las últimas novedades de **C#** y **.NET** no están en **Unity3D**
- Podemos hacer uso de muchas de las clases disponibles en el entorno .NET
- **Unity3D** proporciona un amplio conjunto de clases y posibilidades que podemos utilizar en nuestras aplicaciones.
- En Windows se puede hacer uso de Visual Studio, pero el entorno de desarrollo **MonoDevelop** es una elección interesante (por ejemplo en Mac).

Introducción al C#

- En **C#** TODO es un objeto, incluidos los tipos primitivos, como veremos posteriormente
- Los comentarios con **/* ....*/** , **//** ó **///** cuando queremos hacer uso de generación de documentación
- Podemos definir bloques de código con **#region** nombre ..... **#endregion**

Variables y tipos de datos en C#

- **C#** dispone de un **Sistema Común de Tipos (CTS)**, con dos familias de tipos, valor y de referencia

- Todos los tipos derivan de **System.Object**
- Cualquier tipo puede ser tratado como un objeto

Tipos de datos básicos

Tipo	Descripción	Bits	Rango de valores	Alias
SByte	*Bytes* con signo	8	[-128, 127]	sbyte
Byte	*Bytes* sin signo	8	[0 , 255]	byte
Int16	Enteros cortos con signo	16	[-32.768, 32.767]	short
UInt16	Enteros cortos sin signo	16	[0, 65.535]	ushort
Int32	Enteros normales	32	[-2.147.483.648, 2.147.483.647]	int
UInt32	Enteros normales sin signo	32	[0, 4.294.967.295]	uint
Int64	Enteros largos	64	[-9.223.372.036.854.775.808, 9.223.372.036.854.775.807]	long
UInt64	Enteros largos sin signo	64	[0-18.446.744.073.709.551.615]	ulong
Single	Reales con 7 dígitos de precisión	32	$[1{,}5\times10^{-45} - 3{,}4\times10^{38}]$	float
Double	Reales de 15-16 dígitos de precisión	64	$[5{,}0\times10^{-324} - 1{,}7\times10^{308}]$	double
Decimal	Reales de 28-29 dígitos de precisión	128	$[1{,}0\times10^{-28} - 7{,}9\times10^{28}]$	decimal
Boolean	Valores lógicos	32	true, false	bool
Char	Caracteres Unicode	16	['\u0000', '\uFFFF']	char
String	Cadenas de caracteres	Variable	El permitido por la memoria	string
Object	Cualquier objeto	Variable	Cualquier objeto	object

Variables y tipos de datos en C#

- Comparación entre variables del **tipo valor** y del **tipo referencia**:

Variables de tipo valor:

- Contienen sus datos directamente
- Cada una tiene su propia copia de datos
- Las operaciones sobre una no afectan a otra

Variables de tipo referencia:

- Almacenan referencias a sus datos (conocidos como objetos)
- Dos variables de referencia pueden apuntar al mismo objeto
- Las operaciones sobre una pueden afectar a otra

Variables y tipos de datos en C#

- Los valores de referencia son creados con la palabra reservada new:

```
object o = new System.Object();
```

- Una variable string se puede inicializar directamente:

```
string s = "Hola";
```

- C# soporta secuencias de escape como en C:

```
string s1 = "Hola\n"; // salto de línea
string s2 = "Hola\tque\ttal"; // tabulador
```

- Como las sentencias de escape comienzan con '\', para escribir este carácter hay que doblarlo, o usar '@':

```
string s3 = "c:\\WINNT";
string s4 = @"C:\WINNT";
```

Clases en C#

- Guardan gran similitud con **C++** y **Java**, con algunas diferencias
- Las clases pueden ser abstractas
- La herencia y la implementación de interfaces se lleva a cabo con el operador **:** tras el nombre de la clase
- Los métodos **NO** son por defecto polimórficos, como sí lo son en **Java**

Variables y tipos de datos en C#

- El modificador **static** puede aplicarse a atributos, métodos y clases
- Una clase estática sólo puede contener atributos y métodos estáticos
- Atributos y métodos estáticos lo son de 'clase' y no de 'objeto', es decir, son únicos y globales a la clase, accesibles sin necesidad de crear objetos
- El modificador **const** permite definir variables constantes (son implícitamente static)
- El modificador **readonly** permitiría crear constantes de 'objeto' (cada objeto podría tener un valor distinto para esa constante)

Conversión entre tipos de datos

Conversión implícita
- Ocurre de forma automática.
- Siempre tiene éxito.
- Nunca se pierde información en la conversión.

Conversión explícita
- Requiere la realización del **cast o** el comando **Convert.Toxxx(var)**
- Puede no tener éxito.
- Puede perderse información en la conversión.

```
int intValor = 123;
long longValor = intValor;  // implícita de int a long
int valor = (int) longValor; // explícita de long a int con cast
int x = 123456;
long y = x;                              // implícita
short z = (short)x;                      // explícita
double d = 1.2345678901234;
float f = (float)d;                      // explícita
long l = (long)d;                        // explícita

//uso de la classe Convert:
int c=154;print(Convert.ToString(c));
```

Conversión entre tipos de datos

Tipo	Se convierte de forma segura a:
Byte	short, ushort, int, uint, long, ulong, float, double, decimal
Sbyte	short, int, long, float, double, decimal
Short	int, long, float, double, decimal
Ushort	int, uint, long, ulong, float, double, decimal
Int	long, float, double, decimal
Uint	long, ulong, float, double, decimal
Long	float, double, decimal
Ulong	float, double, decimal
Float	double
Char	ushort, int, uint, long, ulong, float, double, decimal

Enumeraciones

- Las enumeraciones se crean cuando se necesita que una variable tome valores de una lista limitada no predefinida
- Tienen por tipo base números enteros (la numeración empieza por defecto por 0)
- Son muy interesantes para implementar cambios de estado (enemigos, fases del juego etc.)
- Definición de una enumeración:

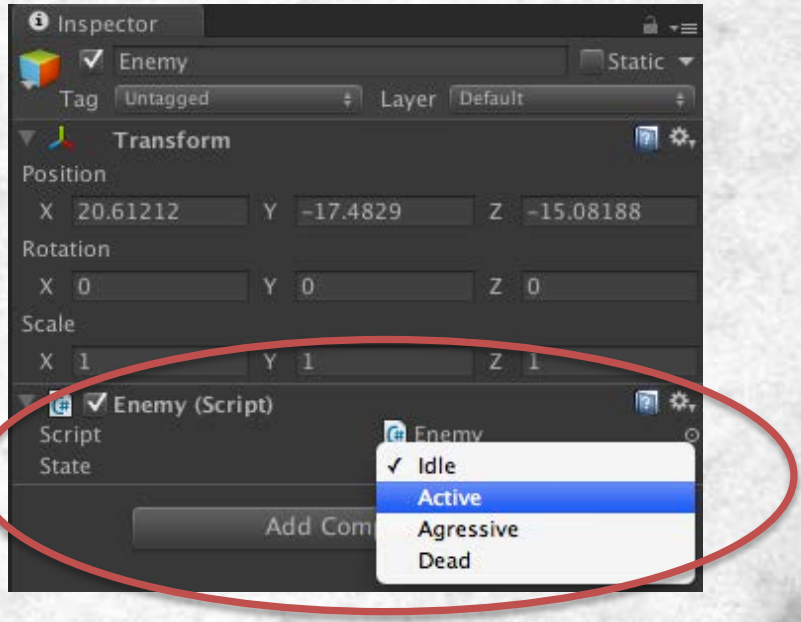
  enum Estado { Idle, Running, Dead }
- Uso de una enumeración:

  Estado estadoPlayer = Estado.Running;
- Aplicar **estadoPlayer.ToString()** devolvería "**Running**"
- Usando enumeraciones el código queda más claro y fácil de mantener.

Enumeraciones

- Las enumeraciones son muy bien entendidas por el editor de **Unity3D**:

```
using UnityEngine;
using System.Collections;

public class Enemy : MonoBehaviour {

    public enum EnemyState { Idle, Active, Agressive, Dead };

    public EnemyState state = EnemyState.Idle;

    // Use this for initialization
    void Start () {

    }

    // Update is called once per frame
    void Update () {

    }
}
```



Estructuras

- **C#** dispone de '**struct**'
- Es muy similar a una clase pero es del tipo '**valor**', lo que implica que su creación no se lleva a cabo en el '**heap**' (montículo o variable global) sino en la '**pila**'
- Puede contar también con métodos
- **Unity3D** los utiliza por ejemplo en la definición de vectores
- *Definición:*

```
struct Vector3 {      // Aproximadamente tal y como está Vector3 en Unity3d
   float x, y, z;
}
....
Vector3 v;
v.x = 3.0f; v.y = 1.0f; v.z = 0.0f;

// Esto equivale también a lo siguiente:
v = new Vector3(3.0f, 1.0f, 0.0f);
```

Arrays

- Los arrays son tipos de referencia derivados de **System.Array**
- Sus índices comienzan en 0
- Ejemplo: **string[] a**; // array de cadenas
  - »El tipo de datos array viene dado por string[]
  - »El nombre del array es una referencia al array
- Para crear espacio para los elementos usar:

  string[] a = new string[100];
- Los arrays se pueden inicializar directamente:

  string[] animales = {"gato", "perro", "caballo"};

  int[] a2 = {1, 2, 3};
- Arrays multidimensionales:

  string[,] ar = {{"perro","conejo"}, {"gato","caballo"}};

  double[,] tempAnual = new double[12,31];

Arrays

- El rango de los elementos del array no es dinámico: si se crea un nuevo array sobre el mismo se libera la memoria que ocupaba y se pierde toda la información que contenía.
- Información sobre un array:
  - Dimensiones: **.Rank**
  - Número total de elementos del array: **.Length**
  - Número de elementos de la dimensión d: **.GetLength(d)**
  - Índice superior e inferior: **.GetLowerBound(d);**
    .GetUpperBound(d);
  - (d=dimensión, desde 0 hasta Rank-1)
  - Saber si es un array: **if (a is Array) ….**
- Recorrido de los elementos de un array sin conocer sus índices
- **foreach (string a in animales)**
  print(a);
- Operaciones de la clase Array: **Copy( ); Sort( );** …

Manipulación de cadenas

- Los tipos **char** y **string** permiten manipular caracteres.
- Una variable **char** puede contener cualquier carácter Unicode
- Manipulacion de caracteres: **char.IsDigit(...); char.IsLetter(...); char.IsPunctuation(...); char.ToUpper(...); char.ToLower(...), char.ToString(...);**..., etc.
- Una variable tipo **string** es una referencia al lugar donde se guarda la cadena.
- Cada vez que se modifica el valor de la cadena se asigna un nuevo bloque de memoria y se libera el anterior (cuidado con el garbage collector)
- Concatenación: operador **+** o se puede usar también el método: **string.Concat()**
- Los operadores **==** y **!=** están sobrecargados a **string.Equals()**

Operadores y expresiones

- **Operadores aritméticos:**
  - `+` Suma unaria: `+a`
  - `-` Resta unaria: `-a`
  - `++` Incremento `++a` o `a++`
  - `--` Decremento: `--a` o `a--`
  - `+` Suma `a+b`
  - `-` Resta: `a-b`
  - `*` Multiplicación: `a*b`
  - `/` División: `a/b`
  - `%` Resto: `a%b`

- **Operadores de manipulación de bits:**

```
int i1=32;
int i2=i1<<2; // i2==128
int i3=i1>>3; // i3==4
```

- **Operadores relacionales**:
  - `==` Igualdad: `a==b`
  - `!=` Desigualdad: `a!=b`
  - `<` Menor que: `a<b`
  - `<=` Menor o igual: `a<=b`
  - `>` Mayor que: `a>b`
  - `>=` Mayor que o Igual a: `a>=b`
  - `!` Negación: `!a`
  - `&` And binario: `a&b`
  - `|` Or binario: `a|b`
  - `^` Or exclusivo: `a^b`
  - `~` Negación binaria: `~a`
  - `&&` And lógico: `a&&b`
  - `||` Or lógico: `a||b`

Operadores y expresiones

- **min=a<b ? a : b;** // equivale a: **if a<b min=a else min=b;**
- **.** accede a miembros, e.j. **args.Length**
- **(<tipo>)** cast (conversión de tipos)
- **[ ]** índice de arrays, punteros, propiedades y atributos
- **new** : crea nuevos objetos
- **.GetType()** : obtiene el tipo de un objeto
- **is** : compara el tipo de un objeto
- **sizeof** : obtiene el tamaño de un tipo en bytes

Control de flujo

- **Instrucciones**
  - Pueden ocupar más de una línea y deben terminarse con un **;**

```
int i, j;
i=1;
```

  - Grupos de instrucciones se pueden agrupar en bloques con **{** y **}**

```
{
    j=2;
    i=i+j;
}
```

  - Un bloque y su bloque padre o pueden tener una variable con el mismo nombre

```
{
      int i;
      ...
      {
        int i;
...
      }
}
```

  - Bloques hermanos pueden tener variables con el mismo nombre

```
{
      int i;
      ...
}
...
{
      int i;
      ...
}
```

Condicionales

- **Condicionales**

```
if (<condición>)
    <sentenciasCondTrue>

if (<condición>)
{
    <sentenciasCondTrue>
}
[else
{
    <sentenciasCondFalse>
} ]
```

Ejemplo:

```
if (a>b) Mayor=a;

if (a>b)
{
   Mayor=a;
   Menor=b;
}
else
{
   Mayor=b;
   Menor=a;
}
```

Condicionales

```
enum Palo { Treboles, Corazones, Diamantes, Picas}
Palo cartas = Palo.Corazones;
string color;

if (cartas == Palo.Treboles)
    color = “Negro”;
else if (cartas == Palo.Corazones)
    color = “Rojo";
else if (cartas == Palo.Diamantes)
    color = "Rojo";
else
    color = “Negro";
```

Condicionales

- Condicionales múltiples

```
switch (<expressión>)
{
  case Opc1 :
            [<sentencias-1>]
            break;
  [case Opc2 :
            [<sentencias-2>]
            break;
            …
   default:
          <sentencias-def>
          break;]
}
```

Ejemplo:

```
switch(B)
{
    case 5:
            print("B es óptimo);
            A=10;
            break;
    case 1:
    case 2:
    case 3: //1,2,3 sin break pq están vacíos
    case 4:
            print("B está por
            debajo del valor óptimo);
            A=3;
            break;
    default:
             print("B no es válido);
            break;
}
```

Condicionales

- Las opciones tienen que ser expresiones constantes, nunca variables.
- El orden de las opciones no importa y no pueden estar repetidas.
- En **C#** se permite usar en el **switch** el tipo **string**

Ejemplo:

```
string country;
const string england = "uk";
const string britain = "uk";
const string spain = "sp";
const string german = "ge";
switch(country) {
     case england: …; break;
     case britain: …; break;
     case spain: …; break;
    case german: …; break;
 }
```

Bucles

- **for**

```
for (int i=0; i < 5; i++) {   // i declarada dentro del bucle
   print(i);
}

for (;;) {
   ... // bucle infinito
}

for (int i=1, j=2; i<=5; i++, j+=2) {  //múltiples expresiones
            print("i=" + i + ", j=" + j);
}
```

Bucles

- **while**
  - Ejecuta instrucciones en función de un valor booleano.
  - Evalúa la expresión booleana al principio del bucle.

```
while (true) {
    ...
}
```

```
int i = 10;
while (i > 5) {
    ...
    i--;
}
```

```
int i = 0;
while (i < 10) {
    print(i);
    i++;
}
```

Bucles

- **do ... while**
  - Ejecuta instrucciones en función de un valor booleano.
  - Evalúa la expresión booleana al final del bucle.

```
do {
   ...
} while (true);
```

```
int i = 10;
do {
   ...
   i--;
}
while (i > 5);
```

```
int i = 0;
do {
    print(i);
    i++;
} while (i <= 10);
```

Bucles

- **foreach**
  - Este bucle es **nuevo** en **C#**.
  - Itera sobre una colección de elementos
  - No permite cambiar el valor de los ítems

```
public void func(string[] args)
{
    foreach (string s in args)
            print(s);
}
```

Métodos

- Los métodos por defecto son privados (**private**)
- **Variables locales** -> Definidas dentro del método, privadas al método, se destruyen a la salida del método (automáticas o de pila)
- **Variables compartidas** -> Variables de objeto o atributos, definidas dentro de la clase
- Los métodos que no sean '**void**' tienen que devolver siempre un valor con '**return**'
- Los parámetros en **C#** se pasan por valor por defecto, pero existe la posibilidad de pasar por referencia y una lista de parámetros de longitud variable

Métodos

- Paso por **valor**
  - Se copia el valor del parámetro.
  - Se puede cambiar el valor de la variable dentro del método.
  - No afecta al valor fuera del método.
  - El parámetro debe ser de un tipo igual o compatible.

```
Void imprime(int x)
{
        print(x);
        x++; // Incrementar x no afecta fuera
}
Void func( )
{
        int k = 6;
        imprime(k); // Muestra el valor 6, no 7
}
```

Métodos

- Paso por **referencia**
  - Se pasa una referencia a una posición de memoria.
  - Se usa la palabra clave **ref** en la declaración y las llamadas al método.
  - Los tipos y valores de variables deben coincidir.
  - Los cambios hechos en el método afectan al llamador.
  - Hay que asignar un valor al parámetro antes de la llamada al método.

```
void imprime(ref int x)
{
        print(x);
        x++; // ahora sí modificamos el valor fuera
}
void func( ) {
        int k= 6; //siempre inicializar antes de utilizarla!!
        imprime(ref k); //imprime 6 pero modifica k a 7
        print(k); //imprime 7
}
```

Métodos

- Parámetros de **salida**
  - No sirven para pasar valores al método.
  - Sólo devuelven resultados desde el método.
  - Se usa la palabra clave **out** en la declaración y las llamadas al método.
  - La variable no necesita estar inicializada.
  - Al parámetro de salida hay que asignarle siempre un valor.

```
void outDemo(out int p)
{
        p = 255;
        // ...
}
void func( ) {
        int n; // sin inicializar
        outDemo(out n); // ahora n = 255

}
```

Métodos

- Lista de **parámetros de longitud variable**
  - Se usa la palabra reservada **params.**
  - Se declara como un array al final de la lista de parámetros.
  - Siempre se pasan por valor.

```
static long SumaLista(params long[ ] v) {
    long total, i;
    for (i = 0, total = 0; i < v.Length; i++)
        total += v[i];
    return total;
}
static void Func( ) {
    long x = SumaLista(63,21,84);
}
```

- **Guía** para el paso de parámetros:
  - El paso por valor es el más habitual y suele ser el más eficaz.
  - El valor de retorno del método es útil para un solo valor.
  - **ref** y/o **out** son útiles para más de un valor de retorno.
  - **ref** sólo se usa si los datos se pasan en ambos sentidos.

Métodos

```
class OverloadingExample
{
    static int Suma(int a, int b)
    {
        return a + b;
    }
    static int Suma(int a, int b, int c)
    {
        return a + b + c;
    }
    static void Main( )
    {
        print(Suma(1,2) + Suma(1,2,3));
    }
}
```

- Podemos sobrecargar métodos (como en **Java**)
- También podemos sobrecargar operadores (como en **C++** pero con una sintaxis mejorada), un ejemplo lo tenemos en **Vector3**, donde podemos sumar, restar o realizar el producto escalar de dos vectores con los operadores **+**, **-** y ***
- El igual también puede sobrecargarse (lo tenemos para **string** y **Vector3** por ejemplo)
- El indizador (el uso de **[]** para acceder a componentes) e incluso el **casting** a un tipo concreto también puede definirse

Excepciones

- Tratamiento de excepciones orientado a objetos :

```
try
{
        ...   // bloque normal de código
}
catch
{
        ...   // bloque que controla la excepción
}
finally {
        ...   // bloque de finalización que siempre se ejecuta
              // Permite limpiar recursos utilizados por try
}
```

Ejemplo:

```
try
{
  int i = int.Parse("aaa");
}
catch (Exception capturada)
{
  print(capturada);
}
```

Las excepciones en **Unity3D** no capturadas se verán en el editor como errores, pero no provocan la interrupción de la aplicación !

Clases en C#

- Nos centraremos en las principales diferencias con otros lenguajes como **C++** y **Java**
- En **C#** se utilizan **namespaces** en lugar de paquetes

  namespace Space { class Class1 {...} .... }
- Se pueden anidar
- Acceso a una clase: **Space.Class1**
- o podemos hacer uso de **using**:

  using Space;

  // Ya podemos acceder a Class1 sin su ruta completa
- Las clases nos permiten crear nuevos tipos de datos
- Contienen atributos, métodos y propiedades
- Pueden implementar interfaces (siempre se usa **:** seguido de las clases de las que deriva y las interfaces en cualquier orden separados por coma)
- También pueden incluir delegados y eventos

Clases en C#

public	La clase o miembro es accesible en cualquier ámbito.
protected	Se aplica sólo a miembros de la clase. Indica que sólo es accesible desde la propia clase y desde las clases derivadas.
private	Se aplica a miembros de la clase. Un miembro privado sólo puede utilizarse en el interior de la clase donde se define, por lo que no es visible tampoco en clases derivadas.
internal	La clase o miembro sólo es visible en el proyecto (ensamblado) actual.
internal protected	Visible en el proyecto (ensamblado) actual y también visible en las clases derivadas.

Clases en C#

- **Constructores y Destructores**
  - En **C#** existen unos métodos específicos para controlar la creación (**constructores**) y la eliminación de objetos (**destructores**).
  - En **C#** si no se definen se heredan por defecto de **System.Object**.
  - Los **constructores** tienen el mismo nombre de la clase y **no devuelven** ningún valor.
  - Pueden tomar o no argumentos (constructor por defecto).

```
class MaClasse {
  public MaClasse()
  {
  // Codi del constructor per defecte
  }

  public MaClasse(int maEnt)
  {
   // Constructor con parámetro maEnt
  }
}
```

  - Puede haber **múltiples constructores** en una clase (si cuenta con diferentes argumentos), pero cada clase sólo puede contar como máximo con **un destructor**.

Ojo ! En los scripts de **Unity3D** los constructores están totalmente desaconsejados (usaremos **Start()** o **Awake()** o lo inicializado en el editor)

Clases en C#

- El **destructor** no retorna ningún valor y no tiene argumentos.

```
~MaClasse() {
  // Códi del destructor.
}
```

- El constructor de la clase se ejecuta en el momento que se utiliza **new** para crear un nuevo objeto de esa clase.
- El destructor de la clase se ejecuta para cierto objeto cuando ya no hay ninguna referencia a ese objeto. De ello se ocupa el **recolector de basura** (garbage collector) de la plataforma .NET. (ya no existe el **delete** en **C#)**
- Para hacer referencia explícita dentro de la clase al objeto que llama al método usaremos **this.**

Clases en C#

- **Herencia**
  - Es el mecanismo para definir una nueva clase (**clase derivada**) partiendo de una clase existente (**clase base**)
  - La herencia permite la **reutilización de código**.
  - La clase derivada hereda **todos** los miembros de su clase base, **excepto** los que son **privados**.
  - Todas las clases derivan implícitamente de Object.
  - **C# sólo permite heredar de una sola clase**.
  - Si una clase deriva de otra clase base se indica tras el nombre de la clase con dos puntos y el nombre de la clase de la cual deriva:

```
<NomDer>:<NomBase>
```

  - Para llamar al constructor de la clase base:

```
public Quadrat(string nom) : base(nom){ ... }
```

Clases en C#

- Cuando no interesa que se derive de una clase (porque se trata de una clase final) se usa el modificador **sealed** (sellada) delante de class:

  public sealed class perro : animal{...}

- En cambio, si la clase sólo debe actuar como **clase base** de otras clases se denomina **clase abstracta** y se usa el modificador **abstract** en la declaración:

  public abstract class animal {...}

  → No es posible instanciar objetos directamente de una clase abstracta, es necesario **derivar.**

Clases en C#

- En una clase se definen campos, propiedades y métodos.
- Cada miembro tiene su propio nivel de accesibilidad: `private`, `public`, `protected`, `internal`, `protected internal`.
- Además, los miembros definidos **estáticos (static)** son compartidos por todos los objetos de la clase y se acceden directamente:

```
<nombreClase>.<nombreMiembroEstatico>
```

Clases en C#

- Métodos **virtuales**.
  - Permiten que las clases sean **polimórficas**.
  - Se usa la palabra clave **virtual**.
  - Un método **virtual** especifica una implementación de un método que puede ser **sustituida** por polimorfismo en una clase derivada.
  - Un método **no virtual** especifica la única implementación de un método. No es posible sustituir por polimorfismo un método no virtual en una clase derivada.
  - Los métodos virtuales no son ni estáticos ni privados.

```
class ObjGeometrico
{
        protected string nombre;
        ...
        public virtual void Dibuja(int x, int y) {
                // dibuja el objeto en la  pos x,y
        }
       public string getNombre() { return nombre;}
    }
```

Clases en C#

- **Sustitución** de métodos (**override**):
  - Un método **override** especifica otra implementación de un método virtual definido en una clase base.
  - El método override debe coincidir **exactamente** con su método virtual heredado asociado.
  - Un método override se puede sustituir también en las clases derivadas.

```
class Cuadrado :  ObjGeometrico
{
    public override void Dibuja(int x, int y)
    {
      // nueva implementación del método
    }

    public override string getNombre() {
                return "Cuadrado" + nom;
    } ←ERROR getNombre no es virtual!!
}
```

Clases en C#

- **Propiedades**
  - Aparecen en la clase como campos de datos.
  - Las propiedades tienen un tipo, pero la asignación y lectura de las mismas se realiza a través de métodos de lectura y escritura.
  - Con las propiedades se puede limitar el acceso a un campo permitiendo sólo la lectura y también se puede realizar cualquier comprobación cuando se asignan valores a la propiedad.

  - **Ejemplo:**

```
private string nombre; // camp privat
public string Nombre   // propietat pública
{
   get {
          return nombre;
   }
   set {
          nombre = value;
   }
}
```

Interfaces en C#

```
public interface IFigura
{
void Dibujar();
float Area();
float Perimetro();
}
```

```
public class Cuadrado : IFigura
{
    protected float lado;
    public Cuadrado(float l) {
            this.lado=l;
    }
    public float Area() {
    return lado*lado;
    }
public float Perimetro() {
return 4*lado;
}
public void Dibujar() {
    print("[]");
}
}
```

Clases en C#

- **C#** permite la sobrecarga de operadores para hacer algunas operaciones más intuitivas: +, -, * , /, <,>…etc.
- Hay que usarlo **sólo** cuando su aplicación resulta más clara que con la llamada a un método.
- Ejemplos:
  - Sumar, restar, etc. dos arrays.
  - Operaciones con números complejos.
  - Comparar cadenas usando >,<, etc.
- Todos los operadores sobrecargados se definen public y static seguido del tipo devuelto, la palabra clave **operator** i el operador que se quiere sobrecargar.

Clases en C#

- El operador **`is`** devuelve **`true`** si es posible realizar una conversión

```
Enemy b;
if (a is Enemy)
b = (Enemy) a;
else
print("It's not an enemy");
```

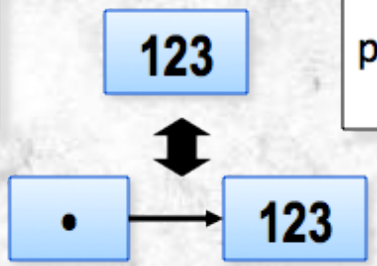
- El operador **`as`** en caso de error, devuelve **`null`** y no una excepción como en caso anterior:

```
Enemy b = a as Enemy; // Conversion
if (b == null)
   print("It's not an enemy");
```

Clases en C#

- Sistema de tipos unificado
- Boxing
- Unboxing
- Llamadas a métodos de object en tipos valor

```
int p = 123;
object box;
box = p;
```




```
p = (int)box;
```

Otras características de C#

- Retorno parcial con '**yield**'

```
using UnityEngine;
using System.Collections;
public class Yield : MonoBehaviour {
      // Use this for initialization
      void Start () {
                  foreach (int i in GiveMeNumbers())
                  print(i);
      }
      IEnumerable GiveMeNumbers() {
                  // We're returning numbers from 0 to 99
                  // but something much complex could be returned
                  for (int i = 0; i < 100; i++)
                              yield return i;
      }
}
```

- En **Unity3D** lo utilizaremos en las 'corutinas', una forma alternativa de llevar a cabo 'multithreading', devolviendo en ese caso **IEnumerator** (la otra posibilidad)

Otras características de C#

- Uso de 'var' (no deja de ser tipado estático)

```
var i = 100;  // i es del tipo int
var menu = new []{"Open", "Close", "Windows"}; // menu es un vector de strings
```

- Tipos nullables

```
int? i = null;  // i puede tomar valores enteros pero también null
```

- Parámetros opcionales

```
void optMethod(int first, double second = 0.0, string third = "hello") { ... }
....
// Posibilidades de llamadas
optMethod(99, 123,45, "World);
optMethod(99);
optMethod(first:99, third:"World");
```

- Clases anónimas

```
var myObject = new {Name = "John", Age = 44};
```

- Colecciones y colecciones genéricas

```
using System.Collections;     ó     using System.Collections.Generics;
```

Otras características de C#

- Métodos de extensión (muy cómodos en **Unity3D**, para por ejemplo extender lo que podemos hacer con los componentes del tipo Transform)
- Expresiones **Lambda** (delegados anónimos)
- **LinQ**
- Tipado dinámico
- Genéricos (listas, diccionarios etc.)
- Un largo etcétera