C#

Jordi Linares. UPV. Campus d'Alcoi. Spain

Scripting

- Introduction to C#
- Developing Unity3D C# scripts

C#

- **C#** was created in 2000 with the **.NET** platform of Microsoft
- Inherits the best from C++, Java and Delphi
- It's an ECMA and ISO, which allows third party implementations such as **MONO project** (now managed by Xamarin)

C#

- Not all C# current features are available in Unity3D

- Unity3D uses always not current Mono versions

- We can anyway use many libraries and classes available in the .NET environment, besides those included and specific of Unity3D

- We use MonoDevelop as IDE (although Visual Studio can also be used in Windows)

C# - Some interesting features

- Reference and output parameters
- Objects on the stack (structs)
- Rectangular arrays
- Enumerations
- Unified type system
- goto
- Versioning

- Component-based programming
  - Properties
  - Events
- Delegates
- Indexers
- Operator overloading
- foreach statements
- Boxing/unboxing
- Attributes
- ...

Introducción al C#

```
using UnityEngine;
using System.Collections;

public class Example : MonoBehaviour
{

    // Use this for initialization
    void Start ()
    {

    }

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

    }
}
```

- `MonoBehaviour`, our parent class form our scripts

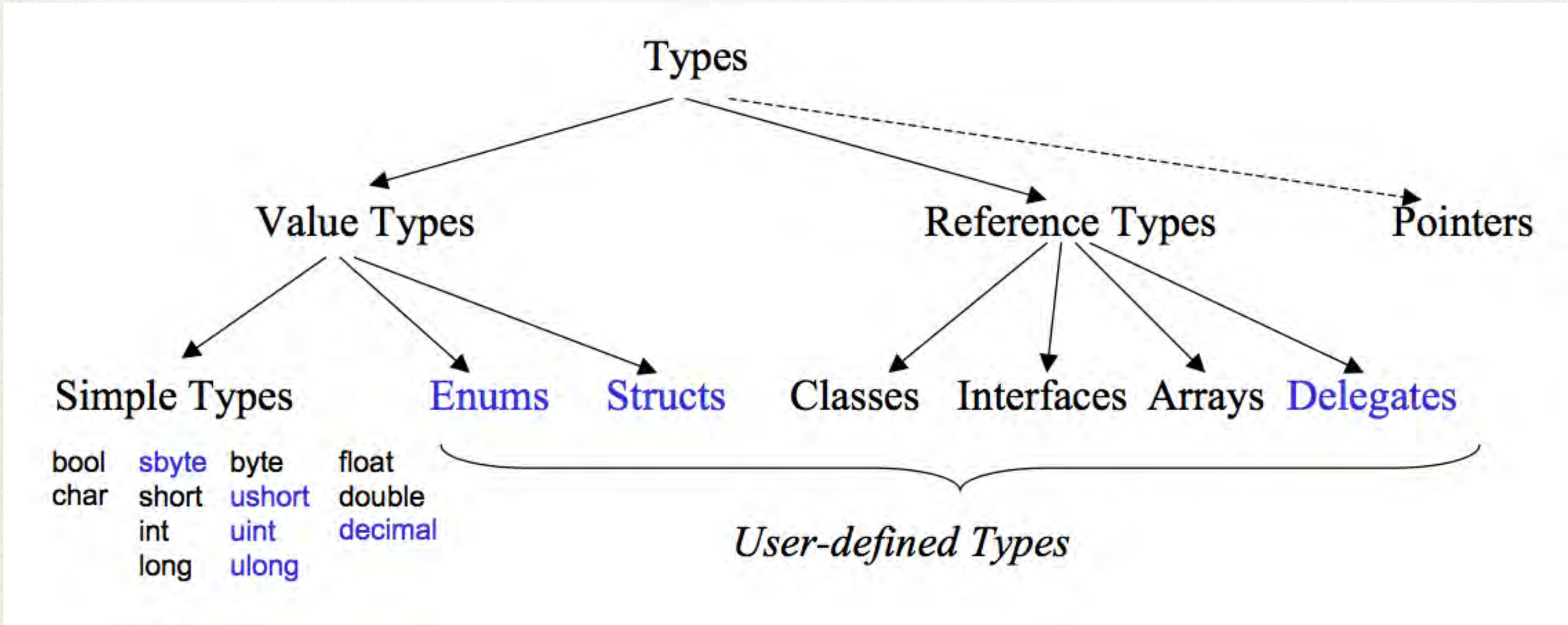
- `System.Collections` belongs to .NET

C#

- In **C#** EVERYTHING is an object (primitive data types included)
- Comments with /*…*/, // or /// (when we want to use doc generation)
- We can generate blocks of code with a name with #**region** name …. #**endregion**

C# - Unified type system

- C# has a **Common Type System** (CTS)



- All types derive from **System.Object** and
- Any type can be treated as an object

C# - Value basic types

C# Type Alias	CLS Type	Size (bits)	Suffix	Description	Range
sbyte	SByte	8		signed byte	–128 to 127
byte	Byte	8		unsigned byte	0 to 255
short	Int16	16		short integer	-32,768 to 32,767
ushort	UInt16	16		unsigned short integer	0 to 65535
int	Int32	32		integer	-2,147,483,648 to 2,147,483,647
uint	UInt32	32	u	unsigned integer	0 to 4,294,967,295
long	Int64	64	L	long integer	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
ulong	UInt64	64		unsigned long integer	0 to 18,446,744,073,709,551,615
char	Char	16		Unicode character	any valid character (e.g., 'a' , '*', '\x0058' [hexadecimal], or '\u0058' [Unicode])
float	Single	32	F	floating-point number	$\pm 1.5 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$
double	Double	64	d	double floating-point number	Range $\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$
bool	Boolean	1		logical true/false value	true/false
decimal	Decimal	128	m	used for financial and monetary calculations	from approximately 1.0×10^{-28} to 7.9×10^{28} with 28 to 29 significant digits
bool	Boolean	true or false		used to represent true or false values	

C#

- Some differences between value and reference data types:
  - **Value types:** they directly store their information, each one has each own copy of the data, operations over one doesn't affect to others
  - **Reference type:** they store references to their actual data (objects), two reference variables can point at the same object, so operations over one could affect to others

C# - Classes

- Classes are very similar to C++ and Java, with some differences
- Classes can be **abstract**
- Inheritance and interface implementation can be carried out with the **:** operator after the name of the class
- Methods are not **polymorphic** by default (as they are in Java)

C# - More details

- The **static** modifier can be applied to attributes, methods and classes
- A **static** class can only have **static** members (variables and methods)
- **Static** attributes and methods are 'class members' and not 'object members', i.e., they are unique and global to the class, and we can access to them without creating objects
- The modifier **const** allows to define constant variables (they are static implicitly)
- The modifier **readonly** allow to create 'object' constants (each object could have a different value for this constant)

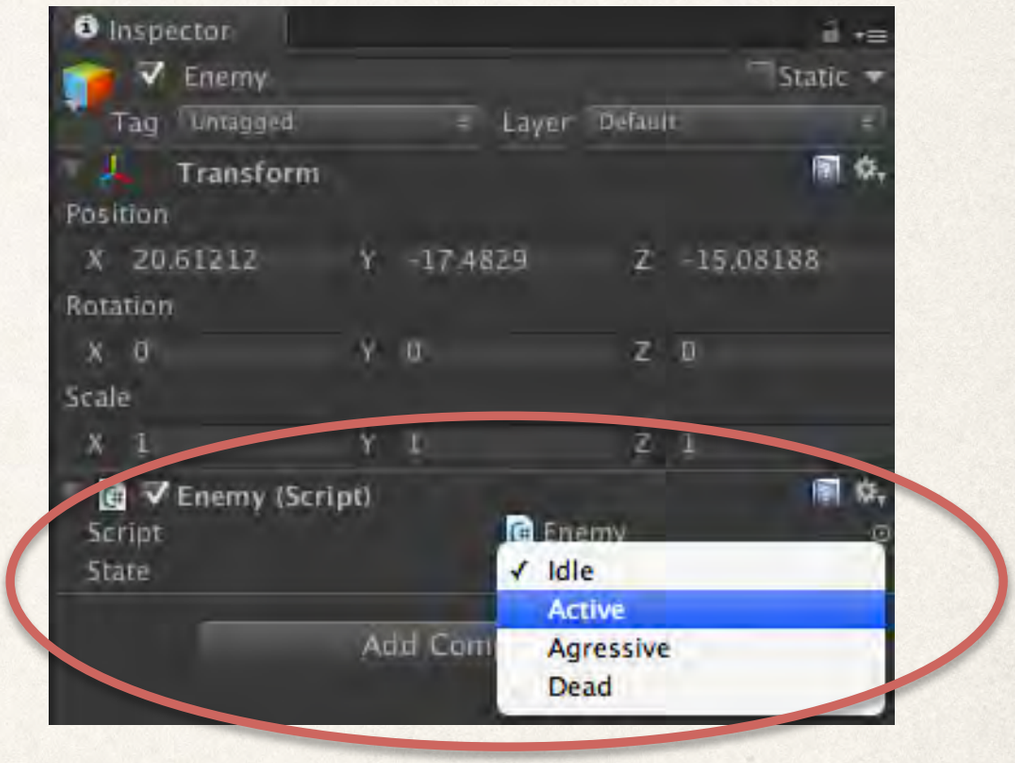
C# - Enumerations

- Enumerations are useful when a variable needs to take values from a limited list
- They are based on integer numbers (enumeration starts by default with 0)
- Very interesting to implement state changes (enemies, stages of the game, etc)
- Defining a enumeration: **enum State{Idle, Running,Dead}**
- Using a enumeration: **State playerState = State.Running;**
- Using enums the code is cleaner

C# - Enumerations

- Enums are well understood by 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 () {

    }
}
```



C# - Structs

- **Structs** are similar to classes but they are ‘value’ data types, which means their memory goes to the stack and not the heap
- It can have methods
- Unity3D uses them for vectors …

```
struct Vector3 {
float x, y, z;
}
....
Vector3 v;
v.x = 3.0f; v.y = 1.0f; v.z = 0.0f;
```

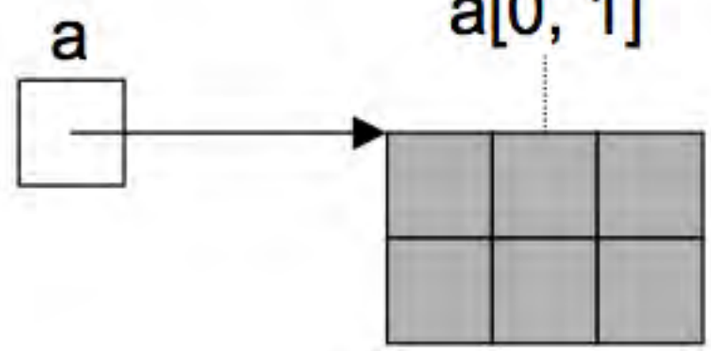
C# - Structs

Classes	Structs
Reference Types (objects stored on the heap)	Value Types (objects stored on the stack)
support inheritance (all classes are derived from *object*)	no inheritance (but compatible with *object*)
can implement interfaces	can implement interfaces
may have a destructor	no destructors allowed

C# - Arrays

Rectangular (more compact, more efficient access)

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

int x = a[0, 1];
int len = a.Length;       // 6
len = a.GetLength(0); // 2
len = a.GetLength(1); // 3
```



C# - strings

- A char variable can store any Unicode character
- char has many methods: **char.IsDigit(…); char.IsLetter(…); char.IsPunctuation(…); char.ToUpper(…); char.ToLower(…), char.ToString(…);**..., etc.
- A **string** variable is a collection of chars; it's a reference to where the **string** is
- Immutable
- Concatenation with + or **string.Concat()**
- Operators == and **!=** are overloaded to **string.Equals()**

C# - Parameters

Value Parameters (input values)

```
void Inc(int x) {x = x + 1;}
void f() {
    int val = 3;
    Inc(val); // val == 3
}
```

- "call by value"
- formal parameter is a copy of the actual parameter
- actual parameter is an expression

ref Parameters (transition values)

```
void Inc(ref int x) { x = x + 1; }
void f() {
    int val = 3;
    Inc(ref val); // val == 4
}
```

- "call by reference"
- formal parameter is an alias for the actual parameter
  (address of actual parameter is passed)
- actual parameter must be a variable

out Parameters (output values)

```
void Read (out int first, out int next) {
    first = Console.Read(); next = Console.Read();
}
void f() {
    int first, next;
    Read(out first, out next);
}
```

- similar to ref parameters
  but no value is passed by the caller.
- must not be used in the method before it got a value.

C# - Boxing/Unboxing

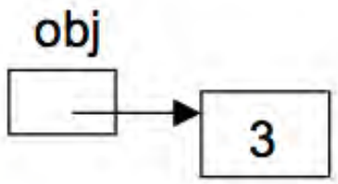
Value types (int, struct, enum) are also compatible with *object*!

Boxing

The assignment

```
object obj = 3;
```

wraps up the value 3 into a heap object



Unboxing

The assignment

```
int x = (int) obj;
```

unwraps the value again

C# - more things ...

- Polymorphism is not by default (**virtual**, **override**, **new**)
- Properties -> What a good idea, forget getter/setters

```
private string name; // private
public string Name   // public
{
   get {
      return name;
   }
   set {
      name = value;
   }
}
```

C# - more things ...

- **Operator overloading**: +, -, *, / etc. (great for Vectors !)

- **Delegates and Events** -> we can define events, subscribe as many methods as we want to them (matching the delegate signature), and these methods will be invoked whenever the event is thrown

C# - more things ...

```
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;
	}
}
```

C# - more things ...

- var

```
var i = 100;  // i es del tipo int
var menu = new []{“Open”, “Close”, “Windows”};
```

- Nullables

```
int? i = null;
```

- Optional parameters

```
void optMethod(int first, double second = 0.0, string third = “hello”) { ... }
....
// Possibilities
optMethod(99, 123,45, “World);
optMethod(99);
optMethod(first:99, third:”World”);
```

- Anonymous classes (and also delegates)

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

- Collections and generic collections

```
using System.Collections;     or    using System.Collections.Generics;
```

C# - more things ...

- Extension methods
- Lambda expressions
- LinQ
- Dynamic typing
- etc.

Developing scripts in C#

- In game development with Unity3D, not all the **C#** features are convenient:
  - Performance problems
  - Integration problems with the IDE
  - Some specific issue when deriving from **MonoBehaviour**
  - Some features are not totally implemented in Unity3D
  - etc
- However, most of C# important features are available and some of the most advanced ones can be really useful in many contexts (delegates and events, lambda expressions, linq, extension methods etc.)
- Anyway, we have to pay attention to everything executed inside methods such as **Update(), FixedUpdate(), LateUpdate(), OnGUI()** etc. since they are executed very frequently

Developing scripts in C#

- **Scripts** in **Unity3D** inherit from **MonoBehaviour**
- A script is a component of a GameObject and, so, **this** in this context makes reference to the **script** and not to the **GameObject**
- A **GameObject** can have many scripts
- Important methods to override:
  - **Awake()**, **Start()**
  - **Update()**, **FixedUpdate()**, **LateUpdate()**
  - **OnGUI()**
  - **OnCollisionEnter()**, **OnCollisionStay()**, **OnCollisionExit()** and their equivalents in non-kinematic objects **OnTrigger**
  - **OnBecameVisible()**, **OnBecameInvisible()**
  - Others: **OnMouseDown()**, **OnMouseOver()**, **OnEnable()**, **OnDisable()** **...**