UNIVERSIDAD DE CASTILLA-LA MANCHA

ESCUELA SUPERIOR DE INFORMÁTICA

GRADO EN INGENIERÍA INFORMÁTICA

Realidad Virtual y Aumentada

Programación e interacción. Lenguaje C# en Unity 3D, control de elementos multimedia y 3D.

Autor: Javier Córdoba Romero y Juan José Corroto Martin

Profesor: Javier Alonso Albusac Jiménez

Diciembre, 2019

Índice

1. C#

La sintaxis de C# está influenciada por la de C, C++ y Java

1.1. Tipos de Variable

En C# existen 4 categorías de variables:

- Valor
- Referencia
- Parámetros genéricos
- Punteros

Tipos de variables por valor Los `structs` y la mayoría de tipos incluidos por defecto pertenecen a esta categoría, exceptuando al tipo `string`, lo mismo ocurre en Java.

Un ejemplo de tipo pasado por valor está en el Listado 1, donde se ve que aunque el valor de `x` sea asignado a `y` y posteriormente el valor de `x` cambie, el valor de `y` no lo hace.

```
1  static void Main()
2  {
3          int x = 4;
4          int y = x;            // Se copia el valor
5
6          Console.WriteLine(x); // 4
7          Console.WriteLine(y); // 4
8
9          y = 7;
10
11         Console.WriteLine(x); // 4
12         Console.WriteLine(y); // 7
13 }
```

Listado 1: Ejemplo de tipo de variable por valor

Tipos de variables por referencia Mientras que los tipos de variables por valor están compuestos por sólo una parte, el valor, los tipos por referencia están compuestos por dos partes: un objeto y la referencia a ese objeto. El valor que se almacena en la variable es la referencia a ese objeto, el objeto se creará y almacenará independientemente de la referencia cuando se utilice el operator `new`.

Un ejemplo de tipo de variable por referencia es la clase `Point2D`, que almacena el valor de

un punto en el espacio 2D, el Listado 2 ilustra un ejemplo de el comportamiento de un tipo de variable por referencia.

```
1  public class Point2D
2  {
3          public int x;
4          public int y;
5  }
6
7  static void Main()
8  {
9          Point2D p1 = new Point2D();
10         p1.x = 4;
11         Point2D p2 = p1;          // Se copia la referencia
12
13         Console.WriteLine(p1.x); // 4
14         Console.WriteLine(p2.y); // 4
15
16         p1.x = 7;
17
18         Console.WriteLine(x);     // 7
19         Console.WriteLine(y);     -// 7
20 }
```

Listado 2: Ejemplo de tipo de variable por referencia

1.1.1. Arrays

En C# los arrays están completamente soportados y gestionados por el lenguaje, sin que el programador tenga que preocuparse por la gestión de la memoria o de cómo se guardan en memoria.

Los arrays siempre se guardan en un bloque contiguo de memoria haciendo que los accesos a sucesivos elementos de un array sean muy eficientes.

Los elementos de un array siempre se inicializarán con un valor por defecto, este valor depende del tipo de variable del array.

Si el tipo de variable del array es por valor, el valor por defecto es cero en el caso de los tipos numéricos, `false` para el tipo booleano, en el caso de los structs, se inicializan con el valor predeterminado de cada elemento, como se puede ver en el Listado 3.

Sin embargo, si el tipo de variable del array es por referencia, el valor por defecto es `null` ya que , como se ha comentado anteriormente, lo que se almacena en la variable es la referencia al

```
static void Main()
{
        int[] x = new int[5];    // Se crea un array de 5 enteros
        x[1] = 2;                                   // Se asigna el valor 2
            al segundo elemento del array

        Console.WriteLine(x[0]);    // 0
        Console.WriteLine(x[1]);    // 2
        Console.WriteLine(x[2]);    // 0
        Console.WriteLine(x[3]);    // 0
        Console.WriteLine(x[4]);    // 0
}
```

Listado 3: Ejemplo de array con tipo de variable por valor

objeto.

La consecuencia de esto es que un array de tipo de variable por referencia necesita ser inicializado después de declarar el array, como se puede observar en el Listado 4.

```
public class Point2D
{
        public int x;
        public int y;
}

static void Main()
{
        Point2D[] x = new Point2D[5];    // Se crea un array de 5 Point2D
        x[1] = new Point2D();                       // Se asigna el valor 2
            al segundo elemento del array

        Console.WriteLine(x[0] == null);    // true
        Console.WriteLine(x[1] == null);    // false
        Console.WriteLine(x[2] == null);    // true
        Console.WriteLine(x[3] == null);    // true
        Console.WriteLine(x[4] == null);    // true
}
```

Listado 4: Ejemplo de array con tipo de variable por referencia

1.2. Clases

En C#, una clase es un tipo de variable por referencia, para declarar una clase basta con escribir la palabra clave `class` seguido por el nombre de la clase y finalizando por los corchetes de apertura y cierre.

Una clase puede contener uno o varios campos, estos campos son variables que pertenecen a la clase, estos campos pueden ser inicializados incluso antes de que el constructor sea ejecutado, un ejemplo de clase con un campo inicializado está en el Listado 5.

```
1 public class Point2D
2 {
3         public int x = 10;  // El valor inicial es 10
4         public int y;       // El valor inicial es el de por defecto, 0
5 }
```

Listado 5: Ejemplo de clase con un campo inicializado

Una clase también puede contener métodos, un método es una función que recibe cero o más parámetros y que puede devolver un parámetro. Un método se define de la siguiente manera:

```
1 tipo-de-retorno nombre-del-metodo(lista-de-parametros)
2 {
3         lista-de-sentencias
4 }
```

Listado 6: Estructura de un método

La definición de un método en concreto se llama *firma*, pueden cooexistir dos métodos con el mismo nombre mientras no tengan la misma firma, a esto se le llama *sobrecarga* de métodos.

Dentro de cada clase hay un método especial, el constructor, este método es llamado justo después de crear un objeto de una clase con el operator `new` y una vez los campos de la clase se han inicializado, el constructor no tiene tipo de retorno y acepta tantos parámetros como sea necesario.

Un ejemplo de clase con campos, métodos sobrecargados y un constructor está en el Listado 7

1.2.1. Herencia

Una clase puede heredar de otra clase para ampliar o personalizar la clase base. La ventaja de la herencia es la capacidad de reutilizar la funcionalidad de la clase base en lugar de construirla desde cero. Una clase sólo puede heredar de una clase, pero puede ser heredada por muchas clases, lo que forma una jerarquía de clases.

Las referencias a una clase de una jerarquía es polimórfica. Esto significa que una variable de tipo X puede referirse a un objeto que heredó de X, como se puede ver en el Listado 8, la variable `Animal` en el método `main` es polimórfica porque puede almacenar tanto una referencia a la clase `Gato` como a la clase `Pajaro`.

```
1  public class Point2D
2  {
3          public int x = 10;       // El valor inicial es 10
4          public int y;            // El valor inicial es el de por defecto, 0
5
6          public Point2D(int x, int y)    // Constructor de la clase
7          {
8                  this.x = x;                          // Asignar el
                       parametro x al campo x de la clase
9                  this.y = y;                          // Asignar el
                       parametro y al campo y de la clase
10         }
11
12         // Metodo que devuelve un Point2D y que acepta un Point2D
13         public Point2D calcularVector(Point2D p)
14         {
15                 int vx = p.x - this.x;
16                 int vy = p.y - this.y;
17
18                 return new Point2D(vx, vy); // Devolver un nuevo Point2D
19         }
20
21         // Metodo sobrecargado que devuelve un Point2D y que acepta dos
               enteros
22         public Point2D calcularVector(int px, int py)
23         {
24                 int vx = px - this.x;
25                 int vy = py - this.y;
26
27                 return new Point2D(vx, vy); // Devolver un nuevo Point2D
28         }
29 }
```

Listado 7: Ejemplo de clase con campos, sobrecarga de métodos y constructor

```
1  public class Animal
2  {
3          public int numeroDePatas;
4          public bool vegetariano;
5
6          public Animal(int nPatas, bool veget)
7          {
8                  this.numeroDePatas = nPatas;
9                  this.vegetariano = veget;
10         }
11 }
12
13 public class Gato : Animal
14 {
15         public string color;
16
17         public Gato(int nPatas, bool veget, string color) :
18                 base(nPatas, veget) // Llamar al constructor de la clase
                       base
19         {
20                 this.color = color;
21         }
22 }
23
24 public class Pajaro : Animal
25 {
26         public string color;
27         public int numeroDeAlas;
28
29         public Pajaro(int nPatas, bool veget, string color, int nAlas) :
30                 base(nPatas, veget) // Llamar al constructor de la clase
                       base
31         {
32                 this.color = color;
33                 this.numeroDeAlas = nAlas;
34         }
35 }
36
37 static void Main()
38 {
39         Animal gato = new Gato(4, false, "blanco");        // Ejemplo de
               polimorfismo
40         Animal pajaro = new Pajaro(2, false, "negro", 2);  // Ejemplo de
               polimorfismo
41
42         Console.WriteLine("El gato tiene " + gato.numeroDePatas + "
               patas");
43         Console.WriteLine("El pajaro tiene " + ((Pajaro)
               pajaro).numeroDeAlas + " alas");
44         Console.WriteLine("El gato es de color " + ((Gato) gato).color);
45         Console.WriteLine("El pajaro es de color " + ((Pajaro)
               pajaro).color);
46 }
```

Listado 8: Ejemplo de clase con campos, sobrecarga de métodos y constructor

1.3. Modificadores

Los modificadores son palabras clave que acompañan a variables, clases y parámetros que modifican la forma de acceder a ellos u otras características como el tipo

Modificadores de variables

Modificadores de clases

Modificadores de parámetros

1.4. Genericos

1.5. Unity – Programación basada en eventos

En algún momento he dicho que los scripts son componentes

En algún momento deberías decir que las variables públicas se pueden ver y modificar desde el editor.

En algún momento deberías decir que es posible que los componentes tengan componentes hijos.

En algún momento deberías hablar de los eventos más importantes de los comportamientos.

2. Uso de C# en Unity

La primera versión del motor de juegos Unity fue lanzada en el 2005, desde entonces mucho ha cambiado en el motor, pasando por el motor de scripting, nuevas tecnologías gráficas y el modelo de licencias.

El lenguaje de scripting de Unity pasó por varias etapas, la primera de ellas fue *Boo*, un lenguaje orientado a objetos usando el *Common Language Runtime* (CLR) de .NET Framework, tecnología similar a la usada por *Java* en su máquina virtual. La siguiente etapa pasó por usar una variante de *Javascript* como lenguaje de scripting y, por último, se pasó a usar C# como lenguaje de scripting con *Visual Studio* como uno de los IDEs soportados.

C# es un lenguaje de programación orientado a objetos y ejecutado sobre el *Common Language Runtime*, usando principalmente con tipos estáticos aunque con soporte para tipos dinámicos, también se puede enfocar a una programación basada en eventos gracias a sus *delegados*.

Ha conseguido obtener la condición de estándar ISO, su última revisión fue en 2018, con referencia ISO 23270:2018[1]

Actualmente Unity soporta más de 20 plataformas[2], entre las que más destacan podemos encontrar: Windows, Linux, Mac, Playstation 4, Xbox One, Nintendo 3Ds, Oculus Rift, Android, iOS, WebGL, ARKit y ARCore.

C# en Unity también permite interaccionar con los diferentes componentes del motor, como por ejemplo *Mecanim*, su sistema de animación, su sistema de físicas o su sistema de componentes, lo que proporciona un control total sobre el motor y no sólo tareas de scripting in-game.

2.1. Estructura de clases de Unity

2.2. Manejo de objetos de Unity desde C#

En esta sección se va a tratar cómo manipular objetos del mundo virtual desde un *script* en *Unity*. Como ya se ha dicho, los *scripts* son componentes de los objetos. Desde cualquier *script* podemos manipular cualquier otro objeto del mundo, o el mismo objeto. Esto último es especialmente fácil si queremos aplicar una transformación, pues tenemos acceso al componente *Transform* como variable de clase. En 9 podemos ver como se puede acceder al componente *transform* del objeto e invocar sus funciones para aplicarle una traslación. La variable *transform* de ese listado no es necesario que se defina en ningún sitio del *script*, sino que se tiene acceso a ella directamente. Esto es porque el componente *Transform* es inherente a todos los objetos que están en la escena, puesto que todos los objetos deben de tener una transformación para saber dónde hay que dibujarlos. El resto de componentes (que deben ser añadidos manualmente), pueden ser accedidos simplemente con la función *GetComponent<Tipo>()*. De esta forma podemos fácilmente aplicar transformaciones o fuerzas (en el caso de rigidBody) sobre el mismo objeto.

```
1 Vector3 direction = new Vector3(mainCamera.transform.forward.x, 0,
      mainCamera.transform.forward.z).normalized * speed * Time.deltaTime;
2 Quaternion rotation = Quaternion.Euler(new Vector3(0,
```

[1] https://www.iso.org/standard/75178.html
[2] https://unity3d.com/es/unity/features/multiplatform

```
    -transform.rotation.eulerAngles.y, 0));
transform.Translate(rotation * direction);
```

Listado 9: Acceso al componente transform para modificar el propio objeto

En el caso de querer modificar otro objeto, por ejemplo, en el caso de querer crear un enemigo a cierta distancia de otro, simplemente tenemos que encontrar la referencia al objeto. Esto se puede hacer muy fácilmente haciendo una variable pública y asignándola desde el editor. Sin embargo, si queremos acceder al objeto de forma dinámica o únicamente usando *C#*, se puede hacer de varias formas, todas usando funciones estáticas de la clase *GameObject*:

1. Por nombre: usando la función *GameObject.Find()*. Se puede buscar de forma directa un objeto en la escena virtual mediante su nombre, simplemente es necesario conocerlo de antemano. Este último matiz puede que sea la mayor inconveniencia, pues es probable que no sepamos de antemano el nombre del objeto, sobre todo si se ha generado dinámicamente. Esta función tratará los caracteres ¨ïo como parte del nombre, sino como parte de una jerarquía de objetos. La documentación oficial de *Unity* [3] desaconseja usar esta función cada frame.

2. Por tipo: usando la función *GameObject.FindObjectOfType(Type type)*. En este caso, buscamos por tipo de objeto, teniendo un ejemplo en el listado 10. Esta función retorna el **primer** objeto cargado de el tipo especificado, o *null* si no existe ninguno de dicho tipo. Para conseguir un iterador de todos los objetos de ese tipo, también existe la función *GameObject.FindObjectsOfType(Type type)*. La documentación oficial desaconseja utilizar estas funciones por ser más lentas que el resto.

3. Por etiqueta o *tag*: usando la función *GameObject.FindGameObjectWithTag(String tag)*. Esta función, al igual que la anterior, devuelve el **primer** objeto cargado con dicha tag, o *null* si no encuentra ninguno. Además, lanzará una excepción si la tag no existe. Esta función es muy fácil de usar, pues *Unity* tiene un sistema nativo de tags para los objetos muy simple de usar, a parte de tener tags predefinidas. El hecho de que sean simples cadenas de caracteres también lo hace más sencillo de utilizar.

```
using UnityEngine;
using System.Collections;
```

[3] https://docs.unity3d.com/ScriptReference/GameObject.html

```
3
4  // Search for any object of Type GUITexture,
5  // if found print its name, else print a message
6  // that says that it was not found.
7  public class ExampleClass : MonoBehaviour
8  {
9      void Start()
10     {
11         GUITexture texture =
               (GUITexture)FindObjectOfType(typeof(GUITexture));
12         if (texture)
13             Debug.Log("GUITexture object found: " + texture.name);
14         else
15             Debug.Log("No GUITexture object could be found");
16     }
17 }
```

Listado 10: Búsqueda de objetos por tipo

3. Ejemplos

4. Conclusion

5. Referencias