CIFP José Luis Garci
Alcobendas (Madrid)

UD 00
INTRODUCCIÓN A LA PROGRAMACIÓN

Desarrollo de Entornos Interactivos Multidispositivo

¿QUÉ ES PROGRAMAR?

La tarea de crear un programa o programar consiste en escribir detalladamente las instrucciones que debe seguir una computadora para realizar una tarea.

Al final de esta unidad, serás capaz de entender los chistes que solo un programador entendería

CREANDO ALGORITMOS

Las instrucciones deben escribirse en un lenguaje que la computadora pueda entender (lenguaje máquina), ya sea en forma directa o tras una traducción realizada por un "intérprete" también llamado compilador.

Algo importante para tener en cuenta es que la computadora carece de "sentido común" y, por tanto, no habrá en el programa nada que nosotros no hayamos previsto, por lo que al programar debemos dejar todas las situaciones posibles bien cubiertas. Los algoritmos que creemos deben ser:

- Precisos: el orden de la secuencia no puede ser alterado. Veremos que poner una línea antes que otra puede cambiar totalmente el resultado.
- Finitos: debe tener un inicio y un final. En más de una ocasión nos encontraremos con bucles infinitos en nuestro código y veremos entonces qué ocurre.
- Correcto tanto en la presentación formal (debe ajustarse a un estándar de programación) y en el resultado (la salida final debe ser la esperada). Debemos aprender la sintaxis de nuestro lenguaje de programación y adaptarnos a ella. Es como aprender un idioma nuevo.
- Eficiente: debe optimizar los resultados de almacenamiento y de procesamiento. Este es uno de los apartados más complicados de conseguir, y el buen programador no es tanto el que consiga que funcione sino el que lo hace escribiendo menos líneas de código y consumiendo menos recursos de la máquina.

Programmer *(noun.)*

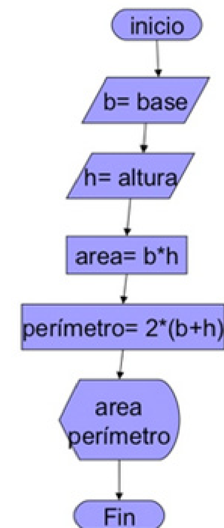
`A machine that turns coffee into code.`

Algoritmos y diagramas de flujo

El diseño de algoritmos consiste en la creación de un conjunto de reglas para desarrollar un cálculo o resolver un problema.

Programar algoritmos es lo más parecido a elaborar diagramas de flujos, en los que una serie de tareas se van desarrollando para lograr un objetivo final.

Reflejan de forma gráfica la secuencia de pasos realizados para la resolución de un determinado problema. Son independientes del lenguaje de programación empleado.



Ejemplo de un diagrama de flujo para hacer amigos (incluyendo un bucle infinito)

LENGUAJES DE PROGRAMACIÓN

Los lenguajes de programación (Javascript, C#, PHP, Java, Pascal, Visual Basic, etc.) son herramientas que nos permiten crear programas y software, creando una vía de comunicación entre el ser humano y las máquinas. Facilitan la tarea de programación ya que disponen de formas adecuadas que permiten ser leídas y escritas por personas y resultan independientes del modelo de computadora a utilizar.

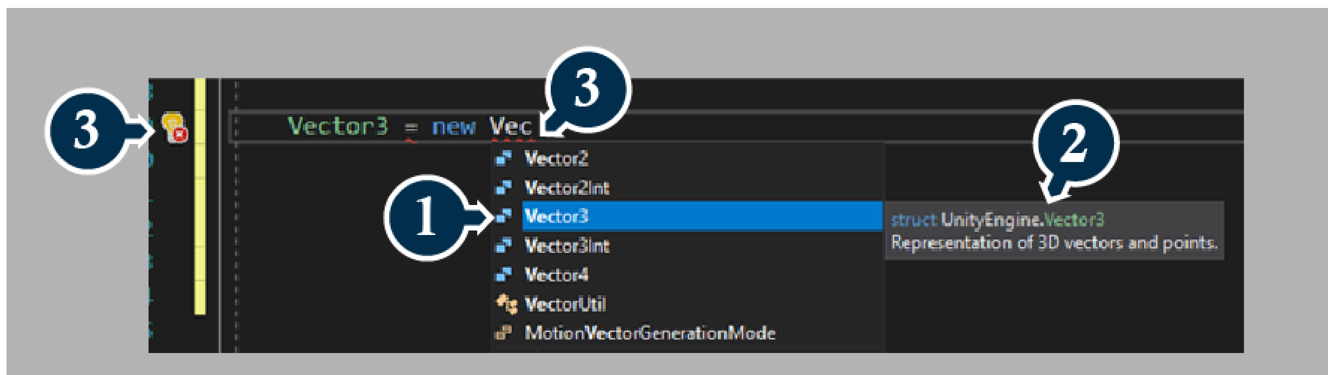
IDE'S (Entornos de desarrollo)

El código que se escribe no deja de ser texto plano, y por tanto se puede programar en el propio bloc de notas, pero no es lo más recomendable.

Lo ideal es usar lo que se llaman "entornos de desarrollo integrado", en inglés Integrated Development Environment (IDE), que nos ayudan mediante herramientas de corrección de sintaxis, autocompletado y detección de errores.

En la imagen podemos comprobar como el IDE no solo nos sugiere el código (**1**), sino que nos da información sobre lo que estamos escribiendo (**2**), e incluso nos señala dónde hay errores de sintaxis que se corregirán en tiempo real (**3**)

Existen muchos programas para editar código. En el caso de Unity, usaremos Visual Studio de Microsoft.



Lenguajes interpretados vs. compilados

Todo lenguaje de programación tiene que ser interpretado por la máquina. Al realizar este proceso de conversión, existen dos tipos:

- **Interpretados**: la conversión se realiza al mismo tiempo que se ejecuta el script (ejemplo: Javascript). Tiene la ventaja de que el tiempo entre la escritura del código y la ejecución disminuye, pero requiere un programa que interprete (como el navegador web), el cual se carga con ese procesamiento, lo que hace que sea menos eficiente, aunque más compatible.

- **Compilados**: requiere de un paso previo antes de ser ejecutado por la máquina, la denominada "compilación" ó "build". Su ejecución es más rápida y eficiente, ya que el resultado final está ya en lenguaje máquina. Pero requiere un tiempo de compilación. Además, solo funciona en el SO para el que ha sido compilado, algo que en los lenguajes interpretados no suele pasar.

Veremos que C# para Unity requiere de un proceso de compilación para ver el resultado final, algo que puede durar minutos.

```
99 little bugs in the code,
99 bugs in the code,
1 bug fixed...compile again,
100 little bugs in the code.
```

CIFP José Luis Garci
Alcobendas (Madrid)

VARIABLES EN PROGRAMACIÓN

Uno de los elementos básicos a la hora de programación son las variables

Variable: un dato referenciado por un identificador y cuyo valor puede cambiar a lo largo de la ejecución del código.

DEFINIENDO VARIABLES

Pensemos por ejemplo en las vidas de nuestro personaje en un videojuego. Al comenzar tendrá 3, pero a medida que avanza el juego irán bajando (2, 1, y finalmente 0).

Elementos de la variable

Una variable se compone de dos elementos:

• **Nombre**. No puede cambiar, y dos variables no pueden tener el mismo nombre (dentro del mismo ámbito, como luego veremos). No debe tener espacios en blanco ni se recomienda que empiecen con un número. Tampoco es recomendable usar caracteres latinos o símbolos.

Un truco para dar nombres compuestos a una variable es usar la técnica conocida como "camelCase", que separa las palabras con una mayúscula ("esteEsUnNombreDeVariable").

• **Valor**: cada variable crea un apunte en la memoria del programa y le asigna el valor que indiquemos. Ese valor podrá ir cambiando a lo largo de la ejecución.

Siguiendo el ejemplo anterior, nuestra variable se podría llamar "lifes", y su valor es el que irá cambiando (3, 2, 1, 0…):

```
var lifes = 3;
```

Tipos de variables

Por lo general, al comenzar nuestros scripts declararemos las variables, asignándoles un valor en ese momento o más adelante.

Es importante saber qué tipo de datos van a contener esas variables, es decir, la naturaleza de su contenido, lo que determinará qué podremos hacer con ellas. Aunque hay muchos tipos, vamos a ver algunas de las más utilizadas:

• **Numéricos** . Principalmente se distinguen dos tipos de números (aunque hay más)

Números enteros o "Integer" (**int**)

Decimales (**float**). En C# indicaremos que un número es decimal poniendo una "f" al final.

• **Cadenas de texto** (string): su valor se introduce delimitados por comillas

Podemos usar "\" para introducir comillas en la propia cadena

IMPORTANTE: el tipo de comillas usado para delimitar debe ser la misma en la apertura y el cierre. Ya sea comillas dobles "…", o comillas simples '…'. Y siempre usar "comillas rectas"

• **Booleanos (bool)**: Pueden adoptar solo dos valores predefinidos: true/false. Muy útiles para crear lo que se llaman "interruptores", o por ejemplo para indicar sencillamente si estamos vivos o muertos por ejemplo.

• **Arrays**: tablas de datos asociativos. Una misma variable puede contener varias variables, o incluso varios arrays asociados. Las veremos en detalle más adelante.

PARA NOTA: existen muchos más tipos de variables, tanto en texto como en número, o algunas diferentes como las que manejan fechas. Puedes consultarlas todas en este enlace: https://www.tutorialsteacher.com/csharp/csharp-data-types

DECLARANDO VARIABLES

Para poder usar una variable en nuestro script, lo primero que tenemos que hacer es declararla (lo ideal es hacerlo al comienzo del script, aunque lo podamos hacer más adelante), y una vez hecho podemos asignarle su valor mediante el símbolo "=".

Indicando el tipo de variable

En C#, como en muchos otros lenguajes de programación, al declarar una variable debemos indicar primero qué tipo de variable es, a continuación, el nombre, y finalmente darle un valor (aunque esto último lo podemos hacer más adelante).

Analicemos el siguiente pedazo de código para entenderlo mejor (ten en cuenta que por lo general el código se va ejecutando de arriba abajo):

```
(1) //Declaro la variable "lifes" en mi script
(2) int lifes; (3)

//Le asigno un valor
lifes = 3;

//Ahora me quito una vida
(4) lifes = 2;

//Ahora voy a declarar una variable al mismo tiempo que le asigno su valor
(5) bool vivo = true;

//Finalmente una cadena de texto, que por ello está entre comillas
(6) string mensaje = "Hola mundo";
```

• Cuando programamos, se pueden escribir líneas de comentarios (1), muy útiles para pasarnos notas a nosotros mismos o a otros que lean el código. En muchos lenguajes, iniciar la línea con "//"

• Lo primero que debemos hacer es "declarar" la variable (2) para que podamos usarla más adelante. En este caso, hemos declarado una variable llamada "lifes"

En C#, como en otros lenguajes, antes del nombre debemos indicar qué tipo de variable es. Es un número entero (int) porque no hay decimales en las vidas, pero si por ejemplo quisiésemos crear una variable de energía que sí admite decimales podríamos usar una variable de tipo "float"

• IMPORTANTE: en muchos lenguajes la máquina ignora los espacios en blanco y los saltos de línea. Por ello, para indicarle que hemos terminado una instrucción (3) debemos terminarla con un ";".

• Una vez declarada la variable, la podemos usar, asignándole un valor. Ese valor puede cambiar a lo largo del código. En este caso, aunque al comienzo le doy un valor de 3, al terminar este script el valor de "lifes" será 2, ya que es la última línea ejecutada (4).

Como la variable ya está declarada, no tenemos que indicar qué tipo de variable es para asignarle un valor.

• A menudo, asignamos el valor de la variable al mismo tiempo que la declaramos. En este caso, hemos creado otra variable de tipo booleana (5), llamada "vivo" y le hemos asignado un valor de "true".

• Finalmente, como muestra, hemos creado una variable de tipo "cadena de texto" (6) llamada "mensaje", y que por tanto su contenido va entre comillas.

PARA NOTA: para ahorrar código, y eso siempre es bueno, existen atajos a la hora de dar un valor numérico, o incluso booleano. Observa las siguientes expresiones:

```
//TRUCOS DE PROGRAMADOR
int num = 5;

num++; //Acabo de sumar uno a la variable, y ahora vale 6
num--; //Acabo de restarle uno, vuelve a valer 5
num += 10; //Le he sumado 10, vale 15
num -= 10; //Le he restado 10, vuelve a ser 5

//Ahora un truco útil con booleanas
bool myVar = true;

myVar = !myVar; //Le he dicho que valga lo contrario, es decir false
```

CIFP José Luis Garci

VARIABLES ESPECÍFICAS EN UNITY

Gracias a la herencia de los métodos contenidos en la clase principal de Unity "MonoBehaviour", un concepto que veremos al estudiar la POO, tendremos a nuestra disposición muchos más tipos de variables, además de las propias del lenguaje C#.

Algunos ejemplos de tipos de variables que podemos declarar en Unity:

• **Vector2 / Vector 3**: indican posiciones en el espacio del juego así como direcciones de desplazamiento, definidas por los valores de los ejes X, Y y Z (en un Vector2 solo X e Y)

• **Quaternion**: representa rotaciones en los diferentes ejes

• **Transform**: contiene los parámetros de posición, rotación y escala del GameObject

• **GameObject**: nos permite acceder a todos los componentes del GameObject

Igual que en las anteriores, al declarar una variable de un tipo ésta solo puede contener valores que coincidan con su naturaleza.

```
public class myScript : MonoBehaviour {

    private Vector2 nombreVar1;
    private Vector3 nombreVar2;
    private Quaternion nombreVar3;
    private Transform nombreVar4;
    private GameObject nombreVar5;

    // Use this for initialization
    void Start () {
        //Damos un valor a la variable de tipo Vector 3
        nombreVar2 = new Vector3(0, 0, 0);

    }

}
```

Lo que aparece antes del tipo de variable ("private") es su ámbito, que veremos más adelante.

Arrays en Unity

Un elemento muy útil son las variables de tipo **Array**, que permiten almacenar en una misma variable varios valores, identificados por su "clave" (o "key").

En Unity indicaremos que es un Array poniendo unos corchetes "[]" a continuación del tipo de variable. Por ejemplo:

```
string[] myArray;
```

Luego veremos cómo asignar valores en Unity a ese Array, pero si necesitamos que tenga un tamaño ya creado, lo haremos mediante esta sintaxis, que crea un array de cadenas de texto de un tamaño de 5:

string[] myArray = new string[5];

Ahora ya podemos añadir valores a nuestro array y recuperarlos:

```
myArray[0] = "Hola";

myArray[1] = "Adios";

print(myArray[0]);

print(myArray[1]);
```

Como en todo lenguaje de programación, es útil hacer bucles a través de un array para realizar una operación con todos sus elementos. Ejemplo: podemos obtener todos los elementos de una escena que tengan una etiqueta (por ejemplo los enemigos) e ir uno por uno diciéndoles que hagan algo (por ejemplo, atacarnos). Hay dos formas de hacer bucles:

1. Usando la longitud del array, obtenida mediante NombreDelArray.Length y usarlo como un contador de ciclo "for":

```
for(int n = 0; n < myArray.Length; n++)
{
        print(myArray[n]);
}
```

2. Mucho mejor, usar la herramienta foreach, que permite en cada ciclo asociar ese elemento del array a una variable específica:

```
foreach(string variable in myArray)
{
        print(variable);
}
```

CIFP José Luis Garci
Alcobendas (Madrid)

OPERADORES, BUCLES Y FUNCIONES

Llevemos nuestro código a otro nivel

OPERADORES MATEMÁTICOS & LÓGICOS

Los operadores nos permiten ejecutar operaciones tanto con datos numéricos como con cadenas de texto.

Muchos de estos operadores son comunes a todos los lenguajes. Veamos algunos tipos:

Aritméticos

Permiten realizar operaciones matemáticas: sumas (+), restas (-), divisiones (/), multiplicaciones (*), potencias (**).

```
//Declaro dos variables numéricas (de tipo entero)
int num1 = 5;
int num2 = 2;

//Y otra variable que contendrá el resultado
int resultado;

//Aplico un operador y lo guardo en el resultado
resultado = num1 + num2;

//Ahora la variable resultado vale "7"
```

Uno muy interesante es el que nos permite extraer el resto de una división, llamado "módulo", muy útil para saber si un número es par o impar ya que cualquier número par dividido por 2 da un "módulo" igual a cero.

```
//Declaro dos variables numéricas (ahora de tipo decimales)
float num1 = 5;
float num2 = 2;

//Y otras dos variables, una con la division y otra con el "módulo"
float resultado;
float modulo;

//Aplico un operador y lo guardo en el resultado
resultado = num1 / num2;
modulo = num1 % num2;

//Ahora la variable resultado vale "2,5"
//La variable módulo vale "1", que es el resto que queda de la división
```

IMPORTANTE: nunca pongas tildes en los nombres de las variables.

Alfanuméricos

Principalmente usados para concatenar cadenas de texto, es decir, unirlas. El símbolo usado para ello varía de un lenguaje a otro, pero en C#, igual que en Javascript, se usa el símbolo "+", por ello hay que tener cuidado de sumar cadenas de texto y no números, porque el resultado varía. Mira y analiza el siguiente código:

```
//Declaro las variables, de diferentes tipos
int num1 = 5;
int num2 = 2;

string texto1 = "Hola ";
string texto2 = "mundo";

int resultado1 = num1 + num2;
string resultado2 = texto1 + texto2;
string resultado3 = texto1 + num1;

//La variable "resultado1" valdrá 7
//La variable "resultado2" valdrá "Hola mundo"
//La variable "resultado3" valdrá "Hola 5"
```

CURIOSIDAD: si "sumamos" dos números, el resultado es una variable numérica, si "concatenamos" dos cadenas de texto, o un número y una cadena, el resultado es una variable de tipo "string", como se ve en el ejemplo anterior.

Relacionales

Permiten establecer relaciones entre unos valores y otros. Los veremos en funcionamiento en el siguiente apartado de estructuras de control:

- Para comparar si un número o cadena es igual a otro, se usa "==" (opcionalmente "===" que no solo compara los contenidos sino también el tipo de variable)
- En cambio, para comparar si son distintos se usa "!="
- Mayor que ">", Menor que "<", Mayor o igual que ">=", menor o igual que "<=".

```
int num1 = 5;
int num2 = 2;
int resultado = num1 + num2;

if(resultado >= 7)
{
    print(resultado);
}
if(resultado != 7)
{
    print(num1);
}
```

ESTRUCTURAS DE **CONTROL**

Como hemos visto anteriormente, en programación las líneas de código se leen secuencialmente, de arriba abajo. Sin embargo, hay líneas de código que solo queremos que se ejecuten si se dan determinadas circunstancias, para ello usamos las llamadas “Estructuras de control”

Pongamos un ejemplo: nuestro jugador recibe un impacto que le quita 5 puntos de daño. Es impacto nos matará si tenemos cinco o menos puntos de daño, pero en caso contrario lo que tenemos que hacer es restarlos.

Este control que regula el flujo de datos se realiza mediante las llamadas “Estructuras de Control Selectivas”. Veremos las más usadas:

• “**if**”: Comprueba si una condición se da, y si es así, ejecuta el código que se encuentra entre llaves.

• “**else if**”: siempre va después de un “if”, aunque no es imprescindible. Si la condición del if anterior no se da, comprueba otra condición que, si se da, ejecuta el código entre llaves.

•“**else**”: se pone después de un “if” o de una serie de “else if”. Básicamente dice que si no se da ninguna de las condiciones anteriores, se ejecutará lo que hay entre llaves. Muy útil, ya que evita que por un fallo en la lógica no se ejecute nada de nuestro código.

Veamos cómo funciona y cómo es la sintaxis, utilizando el ejemplo anterior (intenta descifrar para qué es cada variable y comprender por qué se usa el tipo que se usa):

```
//ESTRUCTURAS DE CONTROL
//Lo primero, siempre, declaro las variables
float energy = 10f;  (4)
float impact;

bool alive = true;

//En un momento del juego, me impacta un proyectil
impact = 11.5f;

//Compruebo si el impacto me mata, y actuo en consecuencia
//Para ello, compararé el valor del impacto y la energía que tengo
if(impact > energy)  (1)
{  (2)
    alive = false; //Me he muerto
}
else if(impact < energy)
{
    energy = energy - impact; //no me muero pero me resto la energía
}
else  (3)
{
    //No se ha dado ninguna de las circunstancias anteriores
}
```

• La sintaxis tanto de los “if” como de los “else if” es siempre la misma: expresión seguida de unos paréntesis (**1**) que contienen la condición que debe darse, y a continuación llaves (**2**), de apertura y de cierre ({ }) que contienen el código que se ejecutará si esa condición se da.

• Para este ejemplo hemos usado los operadores relacionales vistos antes: mayor qué (“>”) y menor qué (“<”)

Es importante ser preciso en las comprobaciones: ejemplo, un nº puede ser menor que 10 (<10) o menor o igual que 9 (<=9), en ambos casos el resultado es el mismo.

Si deseamos comprobar una igualdad, usaremos “==” ó “===”.

• Puede haber tantos “else if” como queramos. En este caso solo hay uno, pero podríamos añadir más.

RECUERDA: en cuanto se da una condición, la ejecución del código no comprueba el resto.

• La expresión “else”, como puedes ver, no lleva paréntesis (**3**), ya que no exige ninguna condición: se ejecuta si las anteriores no se dan.

• En C#, al asignar un valor numérico a una variable float se debe indicar con una “f” después del número (**4**), aunque no tenga decimales. Y muy importante, los decimales se indican con un “.”

PARA NOTA: demuestra tus habilidades lógico-matemáticas. En el apartado “else” no se ha dado ninguna de las circunstancias anteriores. ¿Eso qué significa, sabrías deducirlo?

OPERADORES LÓGICOS

A menudo en la condición que se pasa entre paréntesis debemos indicar no solo una sino dos o varias condiciones. Para eso se usan los operadores lógicos.

Los más usados son estos dos (aunque hay más):

• **“&&”** para la conjunción

• **“||”** para la disyunción

Para ver un ejemplo, vamos a seguir con el ejemplo anterior pero añadiendo una nueva variable booleana que representa un Power Up en nuestro juego: un escudo. Analiza y descifra el siguiente código.

Cosas a tener en cuenta:

•El “if” inicial comprueba si el impacto no me mata, pero para restar la energía se asegura de que el escudo no está “activo”. Por eso tienen que darse las dos circunstancias (1), separadas por “&&”. Al no darse ya que “escudo” es true, pasa a la siguiente ignorando lo que hay entre las llaves.

• A continuación, comprueba si el escudo está activado (2), en cuyo caso lo desactiva.

• El “else” final decide que me he muerto, ¿sabrías deducir por qué? Y más importante todavía, ¿por qué hemos usado el “mayor o igual que” en el primer if? Una pista: ¿se puede seguir jugando si la energía se ha quedado a cero? RECUERDA: escribiendo código tenemos que ser extremadamente precisos para evitar “bugs”.

```
//ESTRUCTURAS DE CONTROL AVANZADAS
//Vamos a crear un POWER UP
bool escudo = true;

//Si me impactan y tengo el escudo, no resto pero me lo quitan
if(impact <= energy && escudo == false)   (1)
{
    energy -= impact;   (3)
}
else if(escudo == true)   (2)
{
    escudo = false;
}
else
{
    alive = false;
}
```

PARA NOTA: como verás, en esta ocasión para restar el valor del impacto hemos usado uno de los atajos vistos anteriormente (3).

Además, en programación si queremos comprobar si una booleana es “true” con escribirla es suficiente: if(escudo), o bien si queremos comprobar que es false ponemos un “!” antes: if(!escudo)

IMPORTANTE: se usa un “=” para dar un valor a una variable, y dos “==” ó incluso tres “===” para comparar valores. No te confundas, porque las consecuencias pueden ser imprevisibles.

Switch

Siempre que se vayan a hacer varias comprobaciones seguidas, con una opción final por defecto, se recomienda usar la sentencia “switch” en lugar de “if”. Es una forma más “limpia” de escribir este tipo de algoritmos.

```
//SWITCH
float measurement = 10f;
switch (measurement)
{
    case 11f:
        print("vale 11");
        break;

    case 9f:
        print("vale 9");
        break;

    default:
        print("no vale ni 9 ni 11");
        break;
}
```

BUCLES

Los bucles, también conocidas como estructuras de control secuenciales, son fragmentos de código que se repiten de forma cíclica mientras se dé una condición. Veamos los dos más habituales.

PRECAUCIÓN: si no tenemos cuidado, podemos ejecutar un bucle infinito, lo que provocará que el programa -y puede que el ordenador- se cuelgue.

While

Su sintaxis es similar a un "if": expresión seguida de una condición entre paréntesis seguido de unas llaves que contendrán el código que se ejecutará de forma indefinida mientras la condición que se ha indicado siga existiendo. Veamos cómo funciona, y los peligros que tiene en el siguiente ejemplo:

```
//WHILE
//Declaramos una variable con un valor
int num = 0;
//Creamos dos bucles que se ejecutará mientras se dé la condición

//Bucle infinito
while(num < 10)
{
    print(num);
}

//Bucle que se repetirá 10 veces
while(num < 10)
{
    print(num);
    num++;
}
```

• Como vemos, la sintaxis es igual que un "if": while(condición) {   }

• El primer bucle es infinito, porque seguirá haciendo ciclos mientras la variable "num" valga menos que 10, y como "num" vale siempre cero, la "hemos liado parda".

• El segundo bucle ya no es infinito, porque en cada ciclo se suma uno a la variable "num", por lo que en el primer ciclo vale 0, en el segundo 1, en el tercero 2, etc. Cuando llegue a 10 la variable, como "10 NO es menor que 10" se para el bucle y sigue la ejecución del código.

Fíjate en el orden de las líneas, y descubrirás que no es lo mismo poner el envío a consola (print) antes de la suma, que ponerlo después.

Igualmente, no es lo mismo poner como condición que es menor que 10 (num < 10) que si es menor o igual que 10 (num <= 10). En el segundo caso el ciclo se repetiría 11 veces porque:

Los verdaderos programadores empiezan a contar desde cero.

For

Lo que hemos visto en el ejemplo anterior es un contador de ciclos (¿recuerdas el diagrama de flujo de Sheldon, y cómo evitaron el bucle infinito?). Y es algo tan habitual en programación que se creó una expresión para ello: **for**.

El funcionamiento es simple: en lugar de una condición, en el paréntesis se ponen tres elementos separados por ";":

1. El primer elemento establece una condición inicial, normalmente la declaración de una variable, o la asignación de un valor a una variable ya declarada.

2. El segundo elemento establece la condición que debe darse para que el bucle siga dando vueltas (similar al while)

3. La tercera condición indica una acción que realizar en cada ciclo, normalmente sumar uno a la variable (o restar, si queremos una cuenta hacia atrás)

Veámoslo con un ejemplo que muestra una cuenta hacia adelante y una cuenta hacia atrás:

```
//FOR
//Declaramos una variable, pero de momento sin valor
int num;

//Creamos dos bucles para crear contadores
for(num = 0; num < 10; num++)
{
    print(num);
}

for(num = 10; num >= 0; num--)
{
    print(num);
}
```

FUNCIONES

Una función, también denominada "método" en Programación Orientada a Objetos (POO), es un conjunto de instrucciones que se agrupan y sistematizan para poder "llamarlas" (el término correcto es "invocar") en cualquier momento.

Son una herramienta básica para organizar ciertos procedimientos que se repiten en un programa, optimizando enormemente los recursos y reduciendo el tiempo de programación.

Ejemplo: imaginemos que en nuestro juego cada vez que recibimos un impacto tenemos que realizar una serie de comprobaciones: dónde ha sido, por parte de quién, comparar su fuerza con nuestra energía, ver si tenemos escudo, etc. En lugar de repetir esta serie de procesos en todos los casos en los que puedo recibir un impacto, creo una función y la "invoco" cuando sea necesario.

Cuanto más compleja es la función, mejor tienen que estar definidas las variables y las operaciones, y más útil resultará al conjunto.

Las funciones se declaran con un nombre que las identifique, igual que las variables, pero se distinguen por ir acompañadas de unos paréntesis a continuación del nombre. Los parámetros básicos de cualquier función son:

- **Nombre de la función seguido de unos paréntesis**: con el que será llamada desde cualquier parte del código. Esos paréntesis pueden ir vacíos o incluir la variable o variables -separadas por coma- que necesitaré pasar a la función. Si esas variables no están declaradas, las puedo declarar en la misma función.

IMPORTANTE: como en las variables, no puede haber dos funciones con el mismo nombre.

- **Variables**: las variables que le pasaremos a la función para que ejecute sus operaciones. Se declaran dentro de los paréntesis y sólo estarán disponibles dentro de la función (variables locales).

Opcionalmente se puede dar el valor de una variable por defecto, y en ese caso no es necesario pasarla al llamar la función.

En algunos lenguajes como c#, antes del nombre debo indicar qué tipo de dato devolverá la función (string, int, float, etc). Si no devolverá nada, se indicará mediante "void".

En otros lenguajes de programación, se incluye la palabra "function" antes del nombre de la función.

Para ejecutar la función, sólo tenemos que escribir el nombre, pero no olvides los paréntesis.

```
//FUNCIONES

//Función que solo saluda
void Saludar()
{
    print("Hola mundo"); (1)
}

Saludar(); //Invoco a la función

//Función que saluda, pero con el mensaje que yo le mande
void SaludoPersonalizado(string saludo)  (3)
{
    print(saludo);
}
                 (2)
SaludoPersonalizado("Hola función con variable");

//Función similar a la anterior pero que acepta que no le pasen nada
void SaludoPorDefecto(string saludo = "¿No dices nada?")  (4)
{
    print(saludo);
}

SaludoPorDefecto();
```

- En la primera función, al invocarla aparecerá en consola "Hola Mundo" (1)

- En la segunda función, aparecerá el mensaje que yo le pase al invocar la función (2). Ya que la variable "saludo", que he declarado en la misma función (3) adoptará el valor que yo le he pasado al invocarla.

IMPORTANTE: si he declarado una variable de tipo "string" como en este caso, el valor que le pase debe ser una cadena de texto.

- Si no paso el valor de una variable, al invocarla dará error. Para evitarlo, puedo asignar un valor por defecto a la variable declarada (4), y de esta forma si no le pasan ningún valor tomará el que le haya asignado yo.

IMPORTANTE: si no se "invoca" a la función, no se ejecuta el código que contiene.

FUNCIONES (parte 2)

Retorno de un valor

En ocasiones, queremos obtener el valor que nos devuelve una función para usarlo en otra parte. En esos casos debemos usar la orden “return” para devolver el valor.

NOTA: al poner “return” en una función, deja de ejecutarse todo lo que hay a continuación, lo que a veces es útil.

Como vimos, en C#, la función se declara escribiendo primero el tipo de dato que va a devolver (“void” si no retornará nada). Veámoslo con un ejemplo:

```
//FUNCIONES (2ª parte)

//Función que admite dos variables para multiplicarlas
//También devuelve ese resultado mediante "return"
int Multiplicar(int num1, int num2)
{
    int resultado;
    resultado = num1 * num2;

    return resultado;

}

int valorRetornado = Multiplicar(5, 6);

print(valorRetornado);
```

1 2 3 4

• Como vemos, al declarar esta función indicamos qué tipo de valor va a devolver, en este caso, un número entero (1).

• Esta función, a diferencia de las anteriores, admite 2 variables, separadas por coma (2)

• Creamos una variable llamada “resultado” que tendrá el resultado de multiplicar los dos números pasados a la función.

• Mediante “return” indicamos que la función devolverá ese valor (3).

• En este caso no solo llamamos a la función, sino que indicamos que la variable “valorRetornado” adoptará el valor que nos devuelva la función (4). Por eso, este script mostrará en consola 30.

ÁMBITO DE LAS VARIABLES (SCOPE)

Observa detenidamente el siguiente código y verás dos comportamientos que te deberían extrañar, y que son producidos por lo que se llama el "ámbito" (scope) de las variables:

```
//SCOPE
string mensaje = "Hola";

void Saludar()
{
    string mensaje = "Adiós";
    string mensajeLocal = "Hasta luego";
    print(mensaje);
}

print(mensaje);
Saludar();
print(mensajeLocal);
```

Analicemos cosas que no deberían pasar en este código:

1. ¿No se supone que no debe haber variables con el mismo nombre? Y en todo caso, si ya hemos declarado la variable "mensaje", ¿por qué la volvemos a declarar dentro de la función "Saludar"?

2. El IDE nos indica que hay un error en el código, ya que la variable "mensajeLocal" no está declarada, pero, ¿acaso no la hemos declarado previamente?

Todo esto se explica por el ámbito de las variables, algo que tenemos que tener en cuenta en todo momento:

Ámbito

En ese sentido, tenemos dos tipos de variables fundamentalmente según su ámbito:

• **Globales**: se declaran al comienzo y están disponibles en todo el script, incluso dentro de las funciones.

En el ejemplo anterior, la variable "mensaje" declarada al principio es global, y podríamos usarla dentro de cualquier función

• **Locales**: se declaran dentro de una función, y por tanto solo están disponible dentro de ella.

La variable "mensajeLocal" no está disponible fuera de la función "Saludar", por eso da error.

Al "redeclarar" la variable "mensaje" dentro de la función Saludar, la convertimos en una variable local, que es independiente de la otra.

Las variables que declaramos al crear una función, dentro de sus paréntesis, son locales.

Existen otro tipo de ámbitos propios de la Programación Orientada a Objetos, que veremos más adelante al estudiar la POO.

CIFP José Luis Garci
Alcobendas (Madrid)

PROGRAMACIÓN ORIENTADA A OBJETOS

La POO es un paradigma de programación que usa los objetos en sus interacciones, para diseñar aplicaciones y programas informáticos.

Si quieres profundizar en el concepto de POO en C# puedes visitar la página de la W3Schools: https://www.w3schools.com/cs/cs_oop.php

CONCEPTOS BÁSICOS

Clases (atributos y métodos) y objetos

En la actualidad, existe una gran variedad de lenguajes de programación que soportan la orientación a objeto. Nosotros en C# para Unity la usaremos.

La POO difiere de la programación estructurada (PE) tradicional, en la que los datos y los procedimientos están separados y sin relación, ya que lo único que se busca es el procesamiento de unos datos de entrada para obtener otros de salida.

En la programación estructurada solo se escriben funciones que procesan datos. Los programadores que emplean programación orientada a objetos, en cambio, primero definen objetos para luego enviarles mensajes solicitándoles que realicen sus métodos por sí mismos.

La POO define una serie de normas para que el código desarrollado por un programador pueda ser utilizado por otros, sin apenas esfuerzo.

Una clase es una especie de contenedor que guarda en su interior atributos que la definen, y métodos para operar esos atributos. Cada clase se compone de:

- **Nombre**. Por convención, se llaman a las clases en singular y con la primera letra mayúscula, para diferenciarlo de los objetos que son instancias de esa clase.
- **Atributos** / propiedades / datos
- **Comportamientos** / operaciones / métodos (similar a las funciones en PE)

El objetivo es a partir de ella, crear objetos que funcionan como instancias de la clase: cada objeto (instancia) asigna valores a esos atributos y opera con esos métodos, siendo cada instancia independiente de las demás, gracias a una de las propiedades de la POO: el encapsulamiento.

Las clases son "instanciadas" en el código, y a través de esa instancia podemos solicitar y/o declarar todos los atributos de esa clase, así como ejecutar los métodos disponibles en esa clase.

Tomaremos un ejemplo de la vida real:

Imaginemos que trabajamos en un banco, y cada vez que viene un cliente a abrir una cuenta debemos realizar las mismas tareas:

• Asignarle un número de cuenta, un saldo, una fecha de apertura, etc. Esas son las variables -los atributos- de la cuenta.

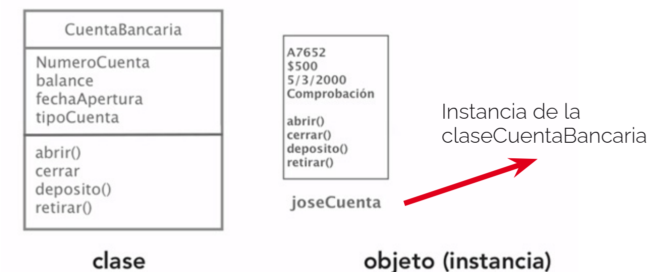
Pero, además, necesitamos realizar operaciones habituales con esa cuenta bancaria:

• Crear cuenta, cerrar cuenta, hacer un depósito, retirar dinero, etc. Esas son las funciones -los métodos- de la cuenta.

Pues bien, para sistematizar todo este proceso, crearemos una clase llamada por ejemplo CuentaBancaria, que a su vez contendrá unos atributos y unos métodos.

Cada vez que abramos una cuenta, estaremos creando una instancia de esa clase, que asignará sus propios valores a los atributos y que tendrá disponible sus métodos. Eso es un objeto de la clase. Y podremos crear tantos como queramos, todos independientes entre ellas.

En la siguiente imagen hemos traducido todo esto a POO, usando UML:



Otro ejemplo de POO en C#

Veamos otro ejemplo de creación de una clase en C#, en este caso tomando como ejemplo un coche que tiene sus atributos (color. Arca, modelo, etc.) y sus métodos (arrancar, detener, etc.)

```
//Creamos la clase
public class Coche
{
    //Atributos de la clase
    public string color;
    public string marca;

    private bool arrancado = false;

    //Métodos
    public void Arrancar()
    {
        //Arrancamos el coche
        if(!arrancado)
            arrancado = true;
    }

    public void Detener()
    {
        //Detenemos el coche si está arrancado
        if (arrancado)
            arrancado = false;
    }

    public void CambiarColor(string newColor = "rojo")
    {
        color = newColor;
    }
}
```

Primero creamos la clase (Coche) con sus atributos y sus métodos.

Como verás, los métodos funcionan como las funciones en PE, y pueden recibir variables, e incluso asignar valores por defecto para evitar errores.

En la siguiente imagen veremos cómo crear dos instancias -objetos- de una misma clase, cada

```
//Creamos 2 objetos a partir de la misma clase
Coche coche1 = new Coche();
Coche coche2 = new Coche();

//Asignamos a cada objeto una marca diferente
coche1.marca = "Ford";
coche2.marca = "Seat";

//Ejecutamos los métodos disponibles
coche1.Arrancar();
coche2.CambiarColor("verde");
```

una con sus propios valores y con los métodos disponibles.

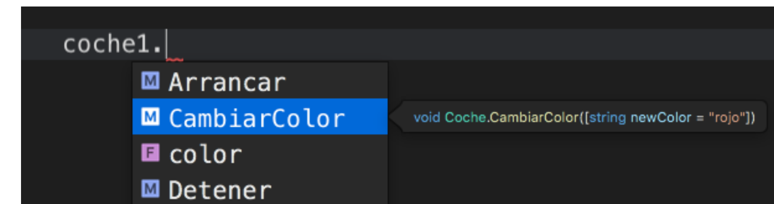
Como puedes ver, al crear los objetos, de nombre "coche1" y "coche2" al especificar el tipo de variable hemos puesto el nombre de la clase, "Coche", y en lugar de asignar un valor hemos indicado que cree una instancia mediante la sintaxis "new".

Una vez creados los objetos, al escribirlos acompañados de un punto podemos acceder a sus atributos y sus métodos que, como en las funciones, van acompañados de paréntesis.

La propiedad de "encapsulamiento" de la que hemos hablado antes permite que cada objeto tenga sus propias características, y aunque accedan a los mismos métodos de la clase, cada uno es independiente. Por ejemplo, tras ejecutar este script el objeto "coche1" estará arrancado, pero el "coche2" no.

NOTA: ¿te has fijado en la cualidad que se define antes de los atributos, private y public? Tiene que ver con el ámbito de las variables en POO, como veremos más adelante.

Como vimos anteriormente, los entornos de desarrollo nos ofrecen ayudas. Por ejemplo, aquí vemos como VisualStudio nos muestra los atributos y los métodos disponibles en la clase:



PARA NOTA: Un tipo específico de métodos en POO son los llamados constructores, que se ejeutan cada vez que se instancia una clase. Puedes investigar algo más en este enlace: https://www.w3schools.com/cs/cs_constructors.php

HERENCIA Y ÁMBITO (SCOPE)

Vemos un concepto de la POO que nos permitirá usar una serie de herramientas propias de Unity

Herencia

Otra característica en la POO es la conocida como "herencia". Un sistema que tiene la POO para reutilizar clases, sus atributos y sus métodos, sin necesidad de crear una nueva ni modificar la anterior.

Se crea una clase que "Hereda" de la anterior, y le añade nuevos elementos. Incluso si es necesario, puede "sobreescribir" sus métodos.

Veamos un ejemplo de creación de una clase que hereda de la que creamos en el ejemplo anterior:

```
public class CocheHeredado : Coche
{
    public float valorMercado;

    public void Revalorizar()
    {
        valorMercado++;
    }

}
```

Ahora, crearemos una instancia de esa nueva clase. Como veremos, no solo podemos acceder

```
//Creamos el objeto usando la nueva clase
CocheHeredado cocheNuevo = new CocheHeredado();

//Accedeos a atributos y métodos heredados y nuevos
cocheNuevo.color = "verde";
cocheNuevo.Arrancar();
cocheNuevo.valorMercado = 125.5f;
cocheNuevo.Revalorizar();
```

a los atributos y métodos heredados, sino también a los nuevos:

En muchos lenguajes se incluyen clases ya incorporadas, en librerías o Frameworks, lo que permite heredar toda una base de datos de atributos y métodos. Un ejemplo es la clase heredada " MonoBehaviour " en Unity, o las librerías que incorpora el script por defecto (Unity. Engine)

Aquí tienes una larga lista de todo lo que se hereda de la clase MonoBehaviour en Unity: https://docs.unity3d.com/ScriptReference/MonoBehaviour.html

Ámbito en POO

Ya has visto que en POO antes de indicar el tipo de variable debemos definir su ámbito dentro de la clase. Éste puede ser:

- **Públicas** (public): accesibles desde fuera de la clase. Necesario para poder cambiarlas o invocarlas desde scripts fuera de la propia clase.

- **Privadas** (private): accesibles solo desde dentro de la misma clase. Por seguridad, si no es necesario acceder a ellas fuera de la clase, es mejor hacerlas privadas. De hecho, si no indicamos nada entenderá que lo son (aunque por cortesía es bueno indicarlo)

Un tipo específico son las **protegidas** (protected) que son accesibles desde la misma clase, y desde cualquier clase que herede de ella.

Si un atributo es privado, podemos crear métodos públicos que acceden a él para tomar su valor o cambiarlo. Son los llamados "getter" y "setter".

- **Estáticas** (static) ó compartidas (shared): usado sobre todo para atributos cuyo valor es compartido por todas las instancias de las clases. Usando el ejemplo del banco, imaginemos que hay un atributo de "euribor", y cuyo valor debe ser el mismo en todos los objetos. Si lo creamos estático, al cambiar en uno cambia en todos a la vez.

ENLACES A VÍDEOS

Aquí podrás acceder a los vídeos que explican y/o profundizan sobre los conceptos vistos, por si te son de utilidad.

Introducción a la programación

1.- Tipos de lenguajes de programación y conceptos que son comunes a todos ellos:

https://www.youtube.com/watch?v=z8tYFZu8J7o

2.- Variables: tipos y su ámbito (scope):

https://www.youtube.com/watch?v=XmEjkTrr9ks

3.- Operadores matemáticos, alfanuméricos y lógicos (“if”)

https://www.youtube.com/watch?v=3h87pJjBu2o

4.- Estructuras de control (“while”, “for”...) y funciones:

https://www.youtube.com/watch?v=5UG2B7vTtqU

Programación orientada a objetos

https://www.youtube.com/watch?v=2UNPb4WS8Vw

Software colaborativo

1.- Introducción al software colaborativo y conceptos básicos en los SCV

https://www.youtube.com/watch?v=ioX-AcbnYno

2.- Funcionamiento de Git Hub

https://www.youtube.com/watch?v=SKs8xdI1uPY

3.- Git Hub como software colaborativo

https://www.youtube.com/watch?v=zc8x2eedEOU