PROGRAMACIÓN ORIENTADA A OBJETOS (POO)

CLASES, OBJETOS, ATRIBUTOS, MÉTODOS Y COMPARTIRLOS ENTRE GAME OBJECTS

Álvaro Holguera – CIFP José Luis Garci (Madrid)

ÍNDICE

PROGRAMACIÓN ORIENTADA A OBJETOS

- En la programación orientada a objetos (POO), es un paradigma de programación que usa los objetos en sus interacciones, para diseñar aplicaciones y programas informáticos.
  - En la actualidad, existe una gran variedad de lenguajes de programación que soportan la orientación a objeto.
- 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.
- Clases y objetos
  - La POO se basa en la existencia de objetos, con una serie de propiedades o "atributos" y que son declarados en "clases".
  - Las clases son "instanciadas" en el código, y a través de esa instancia podemos solicitar y/o delcarar todos los atributos de esa clase, así como ejecutar las funciones (también llamados "métodos") disponibles en esa clase.

Para entender qué son las clases y qué son los objetos, 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.

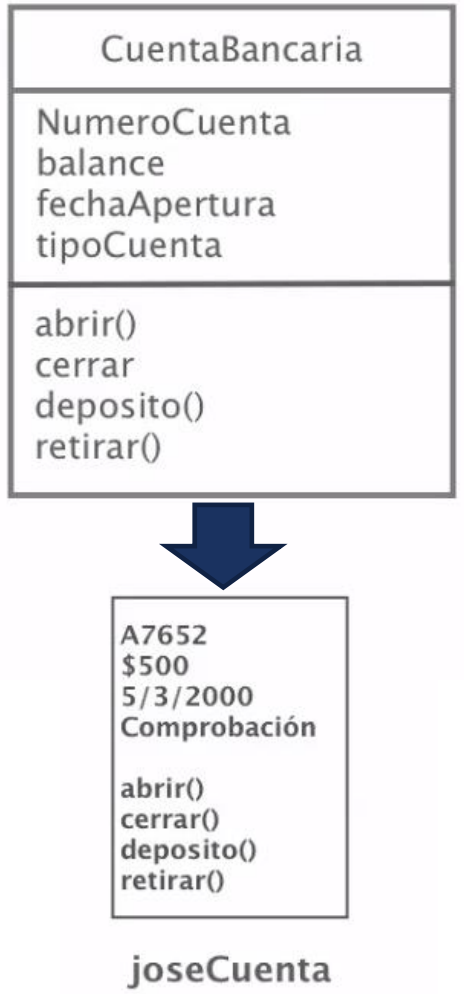
Pero además, necesitamos realizar operaciones habituales con esa cuenta:

- 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 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 tantas como queramos, todas independientes entre ellas (en el ejemplo de la derecha, hemos creado un objeto llamado joseCuenta, que es una instancia de CuentaBancaria)



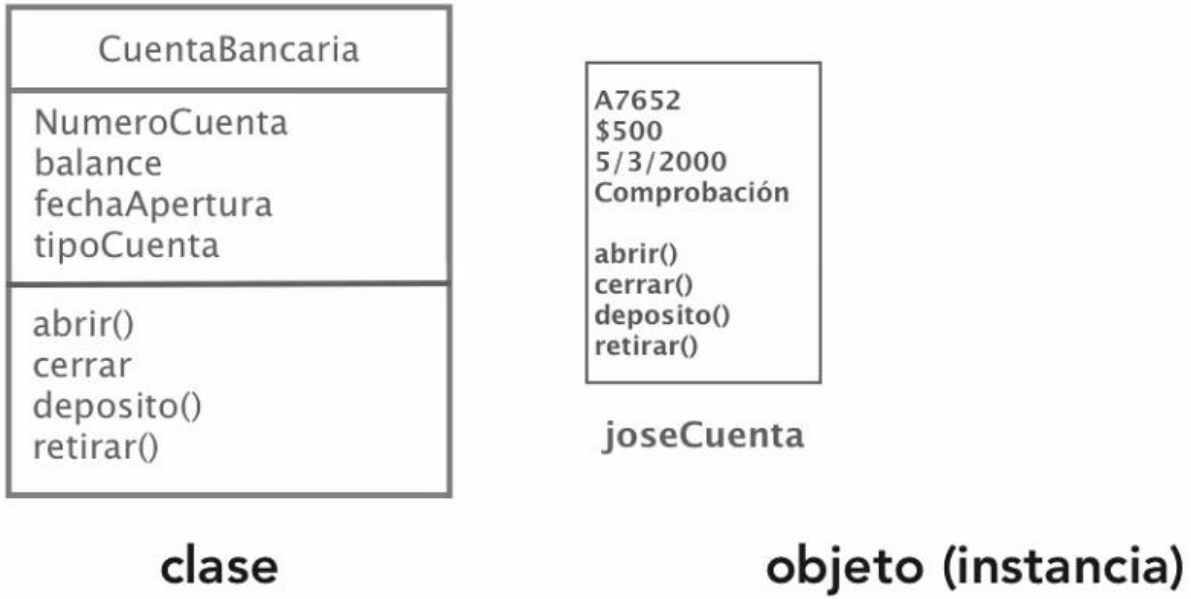
NOTA: esta forma de representar las clases sigue el sistema de modelado UML

¿QUÉ ES UNA CLASE?

- Una clase es una especie de contenedor que guarda en su interior atributos y métodos para operar esos atributos. Se compone de:
  - Nombre
  - Atributos / propiedades / datos
  - Comportamientos / operaciones / métodos (similar a las funciones en PE) Pueden requerir y retornar variables, como las funciones
- 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
  - Por convención, se llaman a las clases en singular y con la primera letra mayúscula
- En muchos lenguajes se incluyen clases ya incorporadas, en librerías o Frameworks
  - Un ejemplo es la clase heredada "Monobehaviour" en Unity, o las librerías que incorpora el script por defecto (Unity.Engine)

Un ejemplo de clase es la que creamos para definir los usuarios de un banco, contará con sus atributos (n° de cuenta, balance, etc.) y con sus métodos que nos permitirán realizar operaciones con esos atributos.

Una vez creada la clase, creamos objetos (instancias) basadas en la clase. Por ejemplo, cada cuenta abierta en el banco por un usuario sería un objeto, una instancia de esa clase, con sus propios valores asignados a cada atributo



EJEMPLO DE CREACIÓN DE UNA CLASE EN POO

- Pensar en términos de objetos es muy parecido a cómo lo haríamos en la vida real.
- Por ejemplo vamos a pensar en un coche para tratar de modelizarlo en un esquema de POO.
  - Diríamos que el coche es el elemento principal que tiene una serie de características, como podrían ser el color, el modelo o la marca.
  - Además tiene una serie de funcionalidades asociadas, como pueden ser ponerse en marcha, parar o aparcar.

```
//Declaramos nuestra clase, que como queremos que esté disponible desde todo el código, la declararemos pública
public class Coche{
            //Atributos de la clase (sus variables)
            var color = "gris";
            var modelo = "deLorean";
            var encendido = false;

            //Métodos, funcionalidades
            public function ponerseEnMarcha(){
                  //Realizamos una acción de encender el coche
                  //Indicamos que llamamos a un atributo de su propia clase con "this"
                        this.encendido = true;
                        }
}
```

```
//Creamos una instancia de nuestra clase Cocje
var miCoche = new Coche();

//Ahora tenemos disponible todos los atributos y funciones de la clase
//Parra llamarlos, se utiliza el nombre de la instancia seguido de un punto
miCoche.color = "rojo";
miCoche.ponerseEnMarcha(); //Ahora el atributo encendido es true;
```

- Una vez declarada la clase, podremos crear una instancia de ella siempre que queramos, y acceder a sus atributos, así como modificarlos y ejecutar sus funciones

- Fuente: http://www.desarrolloweb.com/articulos/499.php

MÉTODOS

- En POO, los métodos son similares a las funciones en PE: un conjunto de instrucciones que se realizan al ser llamados.
  - Tienen un nombre, seguido de unos paréntesis donde se indicarán las variables necesarias separadas por comas, y unas llaves que indican las operaciones a realizar
- Como en PE, los métodos pueden devolver un valor mediante el comando "return"
- En C# deberemos indicar qué tipo de valor se devolverá, y en caso de no devolver ninguno se indicará "void"
  - También deberemos indicar qué tipo de variables se le pasa
- Unity incorpora unos métodos por defecto que veremos más adelante (Start, Update, etc.)

```
 //Método en C#
int Multiplicar(int num1, int num2)
{
    int resultado = num1 * num2;
    return resultado;
}

//Método incorporado por Unity
void Start()
 {
     int res = multiplicar(5,10);
     print(res);
 }
```

PROPIEDADES BÁSICAS DE LAS CLASES EN POO

ABSTRACTION

POLYMORPHISM

INHERITANCE

ENCAPSULATION

ABSTRACCIÓN

- La abstracción consiste en aislar un elemento de su contexto o del resto de los elementos que lo acompañan. En programación, el término se refiere al énfasis en el "¿qué hace?" más que en el "¿cómo lo hace?"

- Nos centramos en las propiedades generales del objeto, más que en el objeto concreto
  - Como cuando decimos "mesa", que todos sabemos a qué nos referimos.
  - Descartamos las propiedades que no son relevantes

- Esta propiedad es el núcleo en POO cuando elaboramos clases.
  - Las clases deben tener los atributos esenciales que permitan incluir todos los objetos que se crearán bajo ella, y que permitan la herencia y el polimorfismo

ENCAPSULACIÓN

- Mantiene el código agrupado y protegido

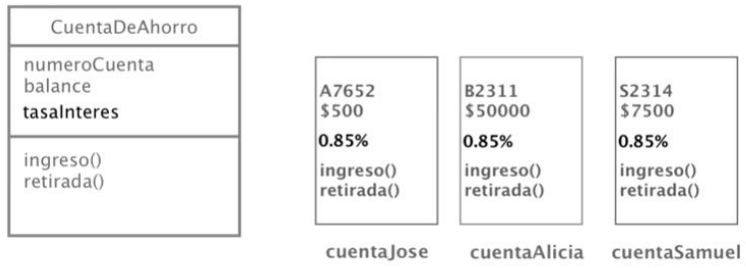
- Restringe el acceso a los mecanismos internos de la clase:
  - El objeto no debe revelar más de lo que sea necesario
  - Podemos ocultar parte de los datos para que sólo sea accesible dentro del objeto en sí (a través de los métodos)
- Este concepto se conoce como "caja negra": sabemos lo que entra y lo que sale, pero no los mecanismos que operan dentro.
- El objetivo es limitar las dependencias entre clases, para evitar que un cambio produzca una alteración en cascada
  - Lo ideal es ocultar todo lo posible y usar el concepto de herencia
  - Para cambiar atributos declarados como "private", se construyen los métodos conocidos como "getters, setters, etc."

Un atributo especial son los denominados "compartidos"

En estos casos creamos una variable estática "de nivel de clase".

No significa que no pueda variar. Pero si se cambia en una instancia, se cambia en todas a la vez.

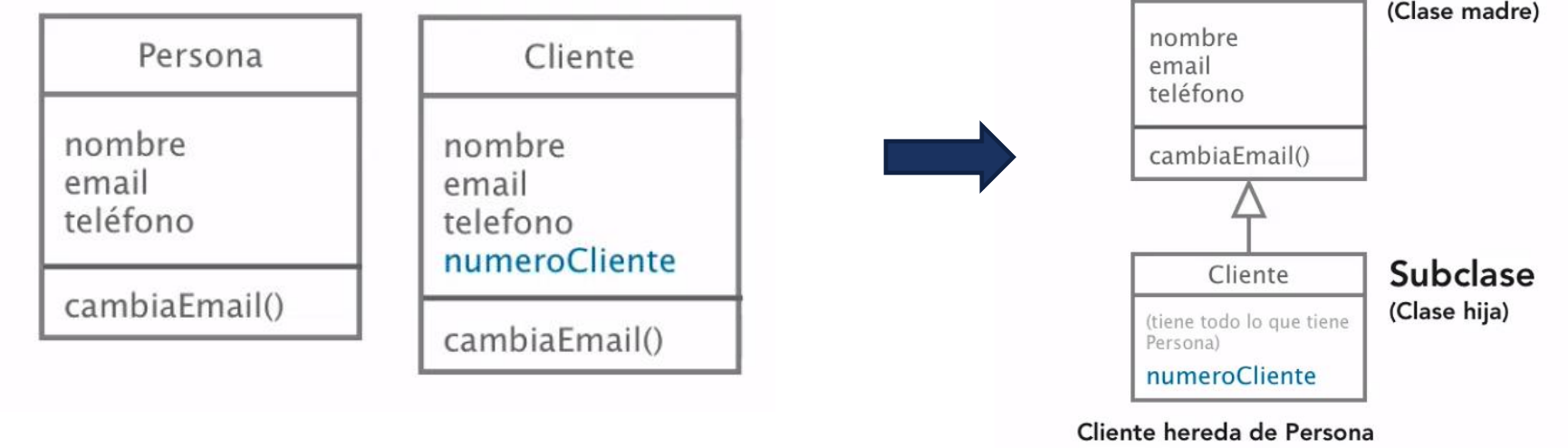
Lo declararemos con el tipo de variable "static"



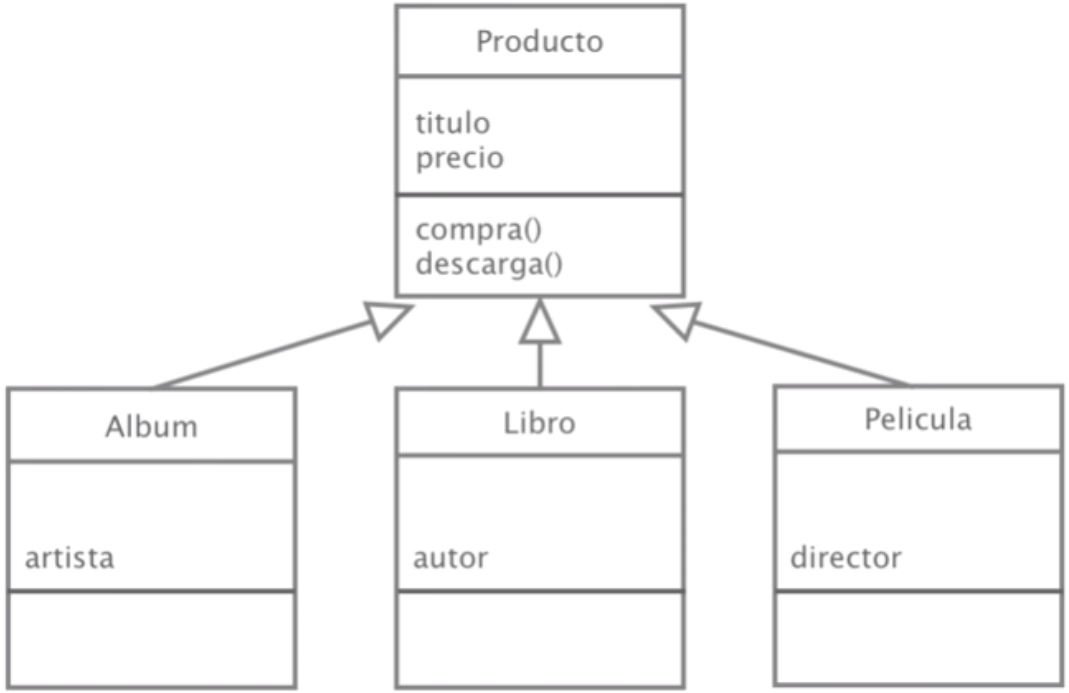

```
static float tasaIntereses = 0,85f;
```

HERENCIA

- Un método para reutilizar clases
- Permite crear una nueva clase que reutilice los atributos de otra anterior cuando estos son compartidos.
  - En ese caso, la nueva clase HEREDA de la clase anterior
  - Automáticamente la nueva clase (SUBLCASE) posee los mismos atributos y métodos
  - La clase hija puede "sobreescribir" métodos de la clase madre
- Es una propiedad muy habitual en el uso de Frameworks



Es importante detectar cuándo varias clases pueden ser agrupadas en una superclase que nos ahorre tener que repetir atributos

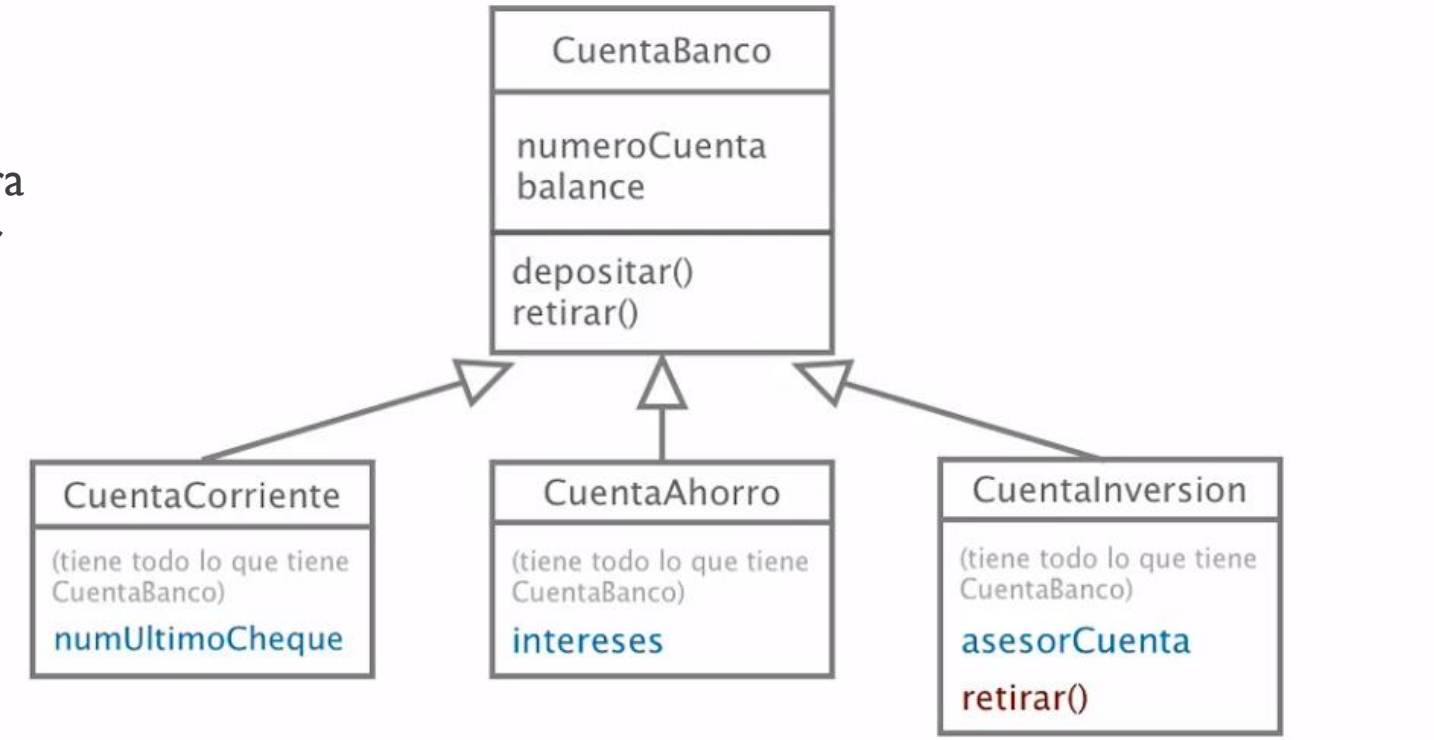


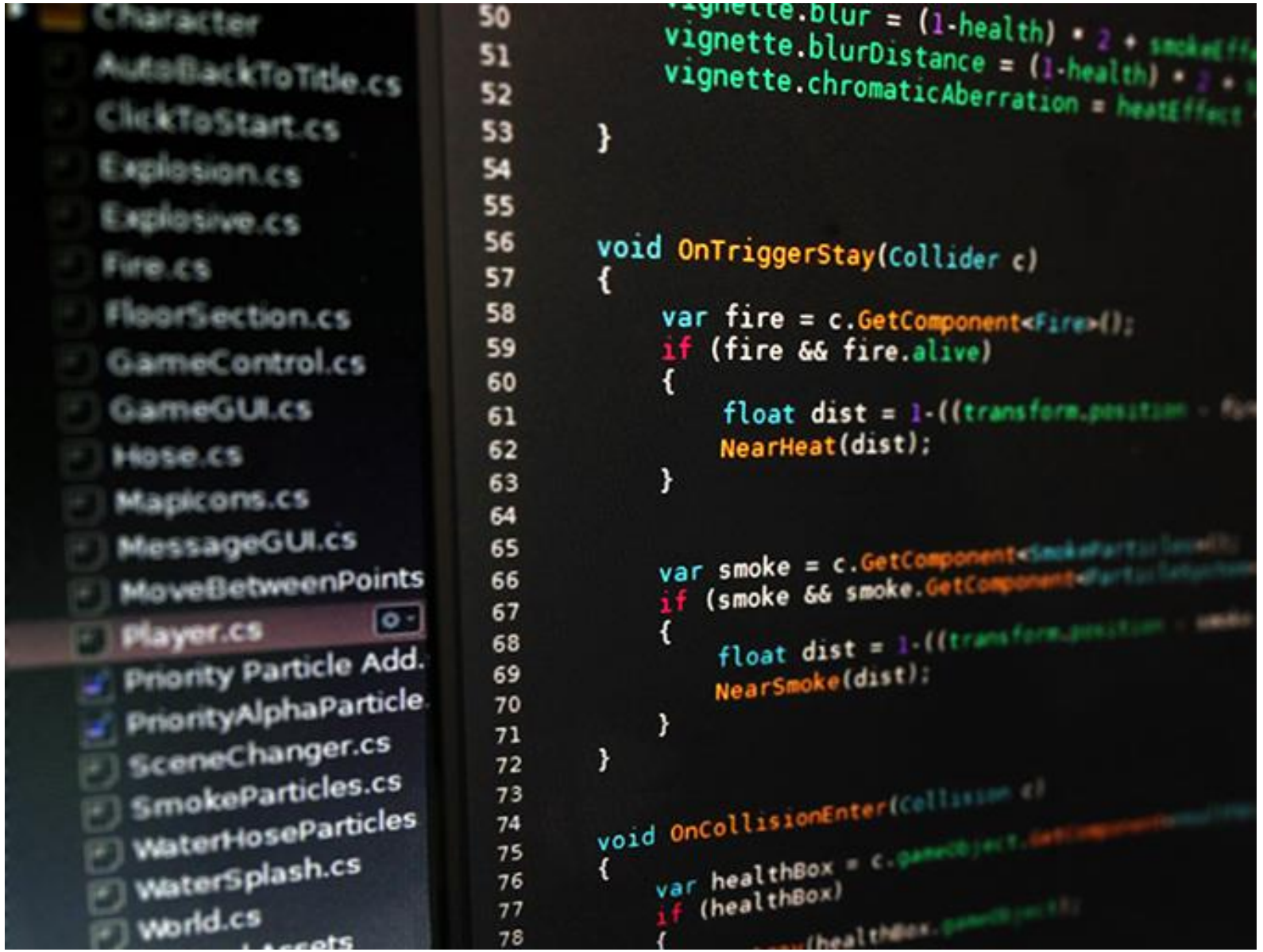

```
Java        public class Album extends Producto { ...
C#          public class Album : Producto { ...
VB.NET      Public Class Album
              Inherits Producto ...
Ruby        class Album < Producto ...
C++         class Album : public Producto { ...
Objective-C @interface Album : Producto { ...
```

Un ejemplo en Unity lo encontramos con la clase heredada de MonoBehaviour.

POLIMORFISMO

- Es el concepto más abstracto de todos
- Cuando creamos una clase heredada podemos "sobreescribir" un método para que se aplique de forma especial, a pesar de ser heredada.
- De esta forma, un mismo método se comportará de una forma y otra dependiendo del objeto que hayamos instanciado.



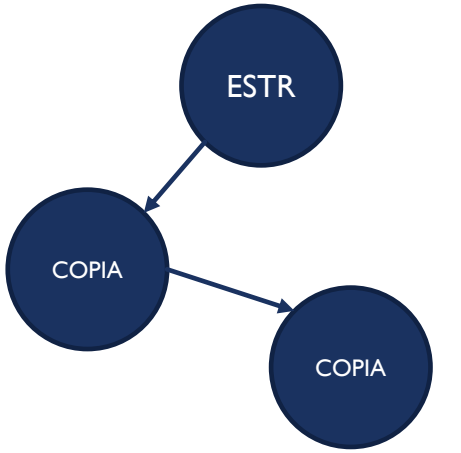


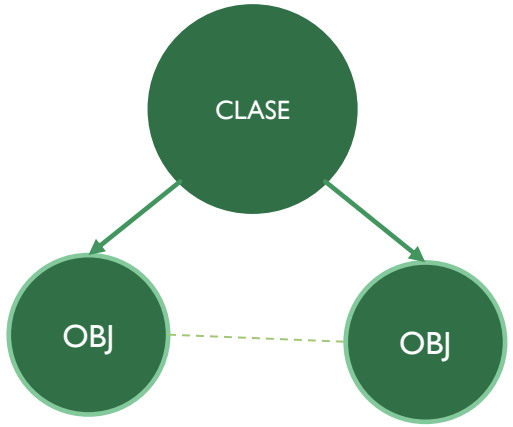
SCRIPTING

CÓMO APLICAR LA POO A C# EN UNITY

PROGRAMACIÓN ORIENTADA A OBJETOS EN UNITY

- Cada vez que creamos un script, se crea una clase con el nombre del archivo
  - Se sigue el nombre de la clase de ":" y "MonoBehaviour", que básicamente es una serie de métodos que incorpora Unity a la clase de forma heredada
- Es buen momento para recordar cómo funciona la POO:
  - Las clases son un conjunto de variables y métodos (funciones), encapsulados
  - Cada vez que instanciamos la clase, estamos creando un objeto que a su vez tiene disponible esas variables y métodos
- Cuando creamos una variable de tipo heredado de Monobehaviour, por ejemplo de tipo Transform, creamos un objeto dentro de esa clase, a su vez con sus métodos y variables disponibles.
  - Así por ejemplo, al crear un objeto Transform, tenemos disponible las propiedades de posición, escala, rotación, etc. que a su vez pueden asumir variables complejas, como las de tipo vector, que son a su vez objetos con variables y métodos propios





En el caso de las estructuras, una copia crea un duplicado, una instancia independiente

Sin embargo, si copiamos una instancia de una clase (llamada objeto), lo que habremos es generado otra instancia, cuyo contenido depende de la clase original, pero es independiente

```
public class EjemploEstructura : MonoBehaviour
{

    //Vamos a crear una estructura de ejemplo
    public struct Animal
    {
        public int id;
        public string nombre;
        public string genero;

        public Animal(int id, string nombre, string genero)
        {
            this.id = id;
            this.nombre = nombre;
            this.genero = genero;
        }

        public void MostrarInfo()
        {
            Debug.Log("EJEMPLO DE ESTRUCTURA:\nNombre: " + nombre + ". Género: " + genero);
        }
    }

    // Start is called before the first frame update
    void Start()
    {

        //Creamos una instancia de la estructura
        Animal miAnimal = new Animal(1, "Gato", "mamífero");
        //Mostramos su información usando el método mostrarInfo
        miAnimal.MostrarInfo();

        //Ahora creamos una nueva instancia de esa estructura, y mostramos la información
        Animal otroAnimal = miAnimal;
        //Mostramos la información del nuevo animal
        otroAnimal.MostrarInfo();

        //Si cambiamos los datos del primer animal, el del segundo se mantiene intacto
        miAnimal.nombre = "Canario";
        miAnimal.genero = "Ave";

        miAnimal.MostrarInfo();
        otroAnimal.MostrarInfo();
    }

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

    }
}
```

```
public class EjemploClase : MonoBehaviour
{

    //Crearemos una clase con datos de animales
    class claseAnimal
    {
        public int id;
        public string nombre;
        public string genero;

        public claseAnimal(int id, string nombre, string genero)
        {
            this.id = id;
            this.nombre = nombre;
            this.genero = genero;
        }

        public void MostrarInfo()
        {
            Debug.Log("EJEMPLO DE CLASES\nNombre: " + nombre + ". Género: " + genero);
        }
    }


    // Start is called before the first frame update
    void Start()
    {
        //Creamos una instancia de la clase y mostramos la información
        claseAnimal primerAnimal = new claseAnimal(1, "Perro", "mamífero");
        //Mostramos la información de esta instancia
        primerAnimal.MostrarInfo();

        //Creamos otra instancia
        claseAnimal segundoAnimal = primerAnimal;
        segundoAnimal.MostrarInfo();

        //Cambiamos los datos del primer objeto, y veremos como cambia el segundo también
        primerAnimal.nombre = "Koala";
        primerAnimal.genero = "Indefinido";

        primerAnimal.MostrarInfo();
        segundoAnimal.MostrarInfo();

    }

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

    }
}
```

ÁMBITO (SCOPE) DE LAS VARIABLES EN POO

- Tanto las variables como las clases pueden ser:
  - Públicas (public): accesibles desde fuera de la clase
  - Privadas (private): accesibles solo desde dentro de la misma clase
    - En una instancia de una clase, podemos hacer alusión a sí misma mediante this.
  - Protegidas (protected): accesibles desde la misma clase y desde cualquier subclase (herencia)
- Si una variable no es pública, podemos establecerla dentro de métodos y devolverla para que puedan acceder a ella o cambiarla, son los llamados "getter" y "setters“ de una clase
  - Para el setter se usará el parámetro predefinido "value"
  - Dentro de las funciones get ó set podemos incorporar todas las comprobaciones que queramos para evitar errores
- IMPORTANTE: si en Unity queremos establecer el valor de una variable desde el editor pero no es necesario que sea pública, mejor usar [SerializeField]

Ejemplo de getter y setters:

```
class MiClase {
  private string myVar;

  //Creamos el método con el nombre de la variable empezando en
mayúscula

  public string MyVar {
      get { return myVar; }
      set { myVar = value; }
  }
}
```

Utilización:

```
MiClase miObjeto = new MiClase();
//Usamos el setter llamando al método (la primera en mayúscula)
miObjeto.MyVar = "valor de la variable";
Debug.Log(miObjeto.MyVar);
```

VARIABLES MONOBEHAVIOUR

- La lista de variables y métodos disponibles gracias a la clase heredada MonoBehaviour es muy grande:
- https://www.youtube.com/watch?v=3Vr9OqoTpdo&list=PLgduEaYImLPNFd7cXnu0zm5qoXOX4wmIJ&index=20
- Métodos como Start y Update, y muchos más
- Variables más utilizadas:
  - gameObject: permite acceder a todos los componentes del GameObject a través de la sintaxis: gameObject.GetComponent<NombreDelComponente>();
  - transform: accede al componente transform del GameObject y a su estructura (trnasform.scale.y, transform.position.x, etc)
  - enabled (bool): si se asigna false, desactivaría todos los comportamientos del script.
  - tag (string) / name (string): podemos cambiar el nombre y la etiqueta. Importante: la variable "tag" debe estar entre las listas de etiquetas predefinidas.
- Recuerda, las variables heredadas no es necesaria declararlas

Inherited Members

Properties

enabled	Enabled Behaviours are Updated, disabled Behaviours are not.
isActiveAndEnabled	Has the Behaviour had active and enabled called?
gameObject	The game object this component is attached to. A component is always attached to a game object.
tag	The tag of this game object.
transform	The Transform attached to this GameObject.
hideFlags	Should the object be hidden, saved with the Scene or modifiable by the user?
name	The name of the object.

```
rigidbody = gameObject.GetComponent<Rigidbody>();
boxCollider = gameObject.GetComponent<BoxCollider>();
audioSource = gameObject.GetComponent<Rigidbody>();
```

OTROS MÉTODOS MONOBEHAVIOUR

- Además de Start y Update, la clase heredada MonoBehaviour proporciona una serie de métodos que nos serán tremendamente útiles:
  - OnCollisionEnter(): detecta cuándo el GameObject colisiona con otro objeto
  - OnTriggerEnter(): cuando el GameObject entra en el área definida por un trigger (por ejemplo cuando entra en una zona)
  - OnDestroy(): al ser destruido
  - OnMouseOver(): cuando pasamos el ratón por encima
  - OnGUI(): permitirá especificar el comportamiento que tendrá el objeto sobre la interfaz gráfica de usuario

```
void OnCollisionEnter ( )
{

     print ( "Impacto" );

}
```

COMUNICAR ENTRE SCRIPTS

ACCEDER A LAS VARIABLES DE UN GAME OBJECT DESDE UN SCRIPT ASOCIADO A OTRO

COMUNICAR ENTRE GAME OBJECTS

```
public class ScriptA : MonoBehaviour
{

    public string myVar = "Hola mundo";

    public void MostrarVar()
    {
        print(myVar);
    }


}
```

- Desde un script asociado a un Game Object podemos acceder a las variables declaradas en otro script asociado a otro Game Object
- En general las clases declaradas en cada script son públicas, por lo que pueden ser "re-declaradas" en otro script.
  - Un ejemplo que muestra este hecho es que podemos crear variables de clase con el nombre de la clase pública
  - Una vez hecho esto podemos acceder a sus variables públicas o a sus métodos, pero esto no es recomendado
- Sin embargo, el método correcto es usando el tipo genérico "GameObject", lo que nos permite asignarle cualquier GameObject de la escena. Lo declaramos y lo asignamos en la interfaz de Unity, para poder acceder a sus variables y a sus métodos

```
public class ScriptB : MonoBehaviour
{

    //creamos una clase a partir de a clase publi
    public ScriptA scriptA;

}
```

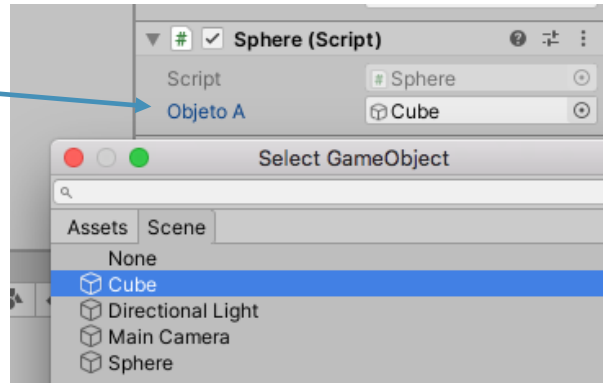
```
public class ScriptB : MonoBehaviour
{

    //creamos una clase a partir de a clase publica
    public ScriptA scriptA;

    void Start()
    {
        scriptA.myVar = "Saludos desde el más allá";
        scriptA.MostrarVar();
    }


}
```

public GameObject ObjetoA;



- A partir de ese momento, tenemos dos opciones:
  - Para acceder a sus variables públicas, deberemos usar la herramienta "GetComponent"
  - Para ejecutar sus métodos deberemos usar la herramienta " SendMessage ", que nos permite también pasar variables a esos métodos

En las siguientes diapositivas se muestra un ejemplo de cada opción

Acceder a variables de otro Game Object mediante GetComponent

Declaramos en un Objeto A, con un ScriptA asociado, en el que creamos una variable de tipo pública para que podamos acceder a ella desde el otro script.

Script A

```
public class ScriptA : MonoBehaviour
{

    public int myVar = 0;


    //En cada fotograma, sumamos uno
    public void setVar(string texto)
    {
        myVar++;
    }

}
```

Declarando una variable de tipo GameObject (por ejemplo llamado ObjetoA), podemos vincular otro elemento del escenario.

Además, debemos crear una variable de clase que contendrá el ScriptA:

```
ambito nombreDeLaClase nombreDeLaVariable;
```

y ya dentro del método Start, vinculamos el Game Object, con todas sus variables, mediante GetComponent, con la siguiente sintaxis

```
nombreDeLaVariable = ObjetoA.GetComponent< nombreDeLaClase >();
```

Script B

```
public class ScriptB : MonoBehaviour
{
    //Creamos una variable de tipo GameObject, y asingamos un objeto en Unity
      public GameObject ObjetoA;
    //Ahora declaramos una variable de la clase del otro script
    private ScriptA scriptA;

    void Start()
    {
        //Asignamos a la variable de clase el otro objeto
        scriptA = ObjetoA.GetComponent<Cube>();
        //Ahora tenemos acceso a las variables
        print(scriptA.myVar);
        //E incluso podemos modificarla
        scriptA.myVar = 1;
        print(scriptA.myVar);

    }
}
```

Ejecutar métodos de otro Game Object mediante SendMessage

Declaramos en un Objeto A, con un ScriptA asociado, en el que creamos varios métodos: uno para asignar un valor a una variable, otro para mostrarlo

Declarando una variable de tipo GameObject, podemos vincular otro elemento del escenario, y una vez hecho esto podemos ejecutar sus métodos usando la sintaxis:

```
VariableObjeto.SendMessage("nombre del metodo", "valor");
```

Script A

```
public class ScriptA : MonoBehaviour
{

    private string myVar = "Hola mundo";

    public void MostrarVar()
    {
        print(myVar);
    }

    //Creamos un método para poder asignar un
     nuevo valor externamente
    public void setVar(string texto)
    {
        myVar = texto;
    }

}
```

Script B

```
public class ScriptB : MonoBehaviour
{

    //Creamos nuestra referencia mediante GameObject
    //Después deberemos vincularla en Unity al Objeto de la
escena
    public GameObject objetoA;

    void Start()
    {
        //Mediante SendMessage, establecemos un nuevo valor
        objetoA.SendMessage("setVar", "Hola desde otro mundo");

        //ejecutamos el método del script externo
        objetoA.SendMessage("MostrarVar");
    }

}
```

BUSCAR GAME OBJECTS

- A menudo tenemos que acceder a un Game Object pero no podemos hacerlo a través de la interfaz de Unity, por múltiples motivos:
  - Debemos seleccionarlo mediante código
  - A veces tenemos que dirigirnos a más de uno
  - Cuando el script está en un prefab, solo podemos acceder desde la instancia de ese prefab, no desde el prefab mismo
- Para ello, Unity nos permite "capturar" Game Objects mediante métodos alternativos:
  - GameObject.Find("nombreDelObjeto"): debemos pasarle el nombre que le hemos dado en la escena, siempre que no tenga "padre" en la jerarquía (si lo tiene, debe llamarse a su ruta, ej: "padre/nombreDelObjeto")
  - GameObject.FindWithTag("etiqueta"): devuelve el Game Object que contenga esa etiqueta (null si no hay ninguno). Para poder usarlo, debemos crear la etiqueta solicitada en el gestor de etiquetas.
  - GameObject.FindGameObjectsWithTag("etiqueta"): devuelve un array con todos los GameObjects que tengan esa etiqueta
- Para evitar problemas de rendimiento, estos métodos no deben usarse en los métodos Update, sino en los "Start" o "Awake"

Método find para obtener el GameObject llamado "hand" que es hijo de "Arm" que a su vez es hijo de "Monster", y una vez capturado rotarlo

```
public class ExampleClass : MonoBehaviour
{
    private GameObject hand;

    void Start()
    {
        hand = GameObject.Find("/Monster/Arm/Hand");
    }

    void Update()
    {
        hand.transform.Rotate(0, 100 * Time.deltaTime, 0);
    }
}
```

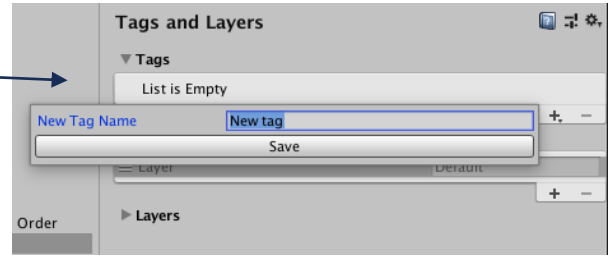
Método FindWithTag para buscar un objeto a través de su etiqueta (Tag).

Para ello, comprobaremos antes si se ha asignado ya,

NOTA: el comprobador "if" puede ir sin llaves, afectando entonces a la línea siguiente nada más

```
public class ExampleClass : MonoBehaviour
{
    public GameObject respawnPrefab;
    public GameObject respawn;
    void Start()
    {
        //Si el GameObject no se ha asignado, lo buscamos
        if (respawn == null)
            respawn = GameObject.FindWithTag("Respawn");
    }
}
```

Podemos usar las etiquetas que incorpora Unity o crear nuevas



Método que busca TODOS los GameObjects que poseen una etiqueta y lo incluyen en un array.

Una vez que tenemos un array con toodos los objetos, podemos hacer un bucle que recorre uno a uno todos esos GameObjects. En el ejemplo siguiente, usaremos cada elemento para usarlo de referencia para instanciar un prefab

```
public class ExampleClass : MonoBehaviour
{
    public GameObject respawnPrefab;
    public GameObject[] respawns;
    void Start()
    {
        if (respawns == null)
            respawns = GameObject.FindGameObjectsWithTag("Respawn");

        //Bucle que pasará por cada elemento del array (que es de tipo GameObject)
        foreach (GameObject respawn in respawns)
        {
            Instantiate(respawnPrefab, respawn.transform.position, respawn.transform.rotation);
        }
    }
}
```