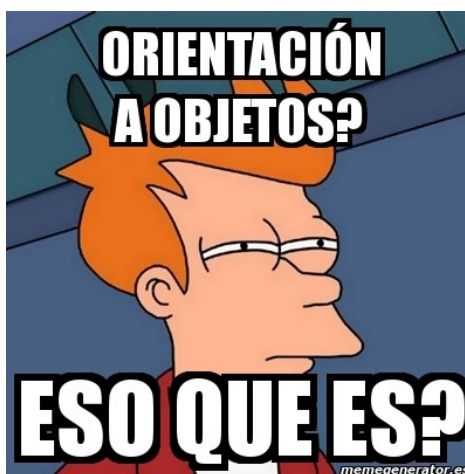


COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Tema 2 – Orientación a objetos

En este tema vamos a dar por supuesto todos los conceptos de Orientación a Objetos vistos el año pasado y simplemente nos vamos a centrar en cómo se realizan las distintas declaraciones y programaciones en C# diferenciándolo de lo que ya sabemos de Java.



Creación de una clase, instanciación de un objeto y otros elementos básicos.


Se realiza de la misma forma que vimos en Java. Se usa la palabra reservada *class* para declarar la clase y el *new* con la llamada al constructor para instanciar el objeto. Lo vemos en el siguiente ejemplo:

```
using System;

class Perro
{
    public string raza;
    public string nombre;

    private int edad;

    // Nota: aunque lo veas así en este ejemplo,
    // en C# NO haremos el set y el get de esta forma.
    // La correcta se explica en un apartado posterior.
    public int getEdad()
    {
        return edad;
    }
}
```

	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

```

public void setEdad(int edad)
{
    this.edad = edad;
}

public Perro()
{
    this.setEdad(0);
    this.raza = "";
    this.nombre = "";
}
}

class Program
{
    static void Main()
    {
        Perro objPerro=new Perro();

        objPerro.raza = "Mastín";
        objPerro.nombre = "Laika";
        objPerro.setEdad(5);
        Console.WriteLine(objPerro.getEdad());
        Console.ReadLine();
    }
}

```

El concepto de **constructor** es el mismo y se definen de la misma forma que en Java: es un método con el mismo nombre que la clase y no devuelve nada, ni siquiera *void*. Admite sobrecargas.

Encontramos también la palabra **this** con el mismo objetivo que en Java.

Los elementos estáticos (como el *Main*) se establecen con la palabra **static** igual que en Java.


Si quisiéramos establecer un destructor para liberar recursos lo escribiríamos con el mismo nombre de la clase pero anteponiendo la virgulilla (~).

```

~Perro()
{
    Console.WriteLine("Finalizado el perro");
}

```

Este método es llamado cuando se libera la memoria. En este caso al final del


	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

programa. Si queremos liberar la memoria de un objeto de forma manual podemos hacerlos asignando el objeto a **null** y forzando el paso del recolector de basura (si no la memoria no es liberada hasta que el recolector pase):

```
objPerro = null;
GC.Collect();
```

Coloca estas dos líneas antes del ReadLine final y verás como el destructor es llamado automáticamente. Incluso si metes un ReadKey() antes de la llamada al recolector, puedes ver en el gráfico de diagnóstico (diagnostic tool) como se lanza dicho recolector mediante una marca amarilla.

Existe un interface (IDisposable) que mejora la liberación de memoria al destruir objetos pero por el momento no lo veremos.

	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Propiedades set/get

En Java cuando queríamos controlar el acceso a una propiedad, esta la declarábamos como privada y creábamos dos funciones que permitían modificar y leer dicha propiedad de la misma forma que está *edad* en el ejemplo anterior.

En C# existe una forma metódica de realizar esto y es con funciones set/get. Tenemos una estructura que nos permite definir una propiedad de la siguiente forma:

```
private tipo_dato nombre_dato_no_accesible;
public tipo_dato nombre_propiedad_accesible
{
    set
    {
        <código ejecutado al modificar la propiedad>
        [nombre_dato=value;]
    }

    get
    {
        <código ejecutado al leer la propiedad>
        return [nombre_dato]
    }
}
```


Es decir, una variable privada que enlazamos mediante una propiedad con métodos **get** y **set**. El valor que se le asigna a la propiedad se guarda en la palabra reservada **value**. Y el *get* siempre debe devolver algo.

Un ejemplo de uso con la clase Perro sería la redefinición de edad de la siguiente forma:

```
private int edad;
public int Edad
{
    set
    {
        edad = value;
    }

    get
    {
        return edad;
    }
}
```

Por supuesto las funciones setEdad y getEdad las borramos.

	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

O se puede hacer con una versión mejorada en el set:

```
private int edad;
public int Edad
{
    set
    {
        if (value < 0)
        {
            edad = 0;
        }
        else
        {
            edad = value;
        }
    }

    get
    {
        return edad;
    }
}
```

Realmente **edad** es la variable en la que se guarda el valor, mientras **Edad** es una "función especial doble" que se usa para acceder a **edad**. De esta manera si en el programa se hace una lectura de **Edad**, por ejemplo:

```
Console.WriteLine(objPerro.Edad);
```

Realmente se obtiene el contenido de **edad** a través del **return** del **get**.


Análogamente, si se realiza una asignación sobre **Edad**, realmente se está ejecutando el **set** y tomando como **value** el dato asignado (es como si fuera el parámetro en el antiguo getEdad) y guardándolo en **edad**.

```
objPerro.Edad = 5;
```

Al ejecutar el set, **value** toma el valor 5 y por tanto se asigna a **edad** dónde queda almacenado.

El programa principal quedaría así:

```
Perro objPerro = new Perro();
objPerro.raza = "Mastín";
objPerro.nombre = "Laika";
objPerro.Edad=5;
Console.WriteLine(objPerro.Edad);
Console.ReadLine();
```

	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Ejecútalo paso a paso para entenderlo bien.

Propiedades autoimplementadas

Es posible en el caso de querer crear propiedades y que no se vaya a usar ninguna comprobación ni lógica adicional en los set y get crearlas automáticamente de la siguiente forma:

```
public int Edad { set; get;}
```

También en versiones actuales de C# se permite inicializarla:

```
public int Edad { set; get;} = 5;
```

Más sobre declaración y uso de propiedades:

<https://docs.microsoft.com/es-es/dotnet/csharp/programming-guide/classes-and-structs/properties>

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Herencia, sobrecarga y sobreescritura

La **herencia** se realiza con el operador dos puntos (:) en lugar de la palabra *extends* de Java.

La **sobrecarga** debe cumplir los mismos parámetros que veíamos en Java (distinta lista de parámetros. Se incluye que un parámetro sea **valor o referencia** como elemento **diferenciador**).

Se debe resaltar que en C# además de sobrecargar funciones se permite **sobrecarga de operadores** tal y como se explica en el [Apéndice I](#).

La **sobreescritura** recomienda que la función que vaya a ser sobreescrita lleve antepuesta el modificador **virtual** y el método que sobreescibe el modificador **override**.

También se puede usar el modificador **new** en vez de **override** pero esto fuerza a que al acceder desde una referencia del tipo base, el método derivado queda oculto, pudiendo acceder solo al de la clase base. Puedes ver un ejemplo en el [Apéndice II](#).

Normalmente usaremos el **override** pues es el uso habitual (tal y como lo usamos en Java).


Vemos un ejemplo:

```
class Animal
{
    private int edad;
    public string nombreCientifico;

    public virtual void MuestraEdad()
    {
        Console.WriteLine(edad.ToString());
    }
}

class Perro : Animal
{
    public string raza;
    public string nombreHumano;

    public Perro()
    {
        this.nombreCientifico = "Canis Familiaris";
    }
}
```

	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

```

public void Ladrar()
{
    Console.WriteLine("GUAU!!!");
}

public void Ladrar(int n)
{
    for (int i = 0; i < n; i++)
        Console.WriteLine("GUAU!!!");
}
}

class Mosca : Animal
{
    public Mosca()
    {
        nombreCientifico = "Drosophila Melanogaster";
    }

    public override void MuestraEdad()
    {
        Console.WriteLine("Las moscas no tienen edad");
    }
}

```

Perro y Mosca heredan de Animal.

En Perro hay dos sobrecargas de ladrar, una sin parámetros y otra con un parámetro entero.

En Mosca, MuestraEdad sobrescribe a la misma función en Animal.

Si quisiéramos hacer referencia en un método que sobrescribe al método sobrescrito podemos usar la palabra reservada **base** de forma análoga al uso de **super** en Java. Por ejemplo:

```

public override void MuestraEdad()
{
    Console.WriteLine("Las moscas no tienen edad, pero tenemos guardado como dato");
    base.MuestraEdad();
}

```


COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Control de acceso y otros modificadores

Al igual que en Java, estos modificadores permiten que ciertos métodos y propiedades estén o no visibles en otras partes del código. Son los siguientes:

- *public*: Igual que la *public* de Java. Puede ser accedido desde cualquier código.
- *protected*: No tiene equivalente en Java. Desde una clase sólo puede accederse a miembros *protected* de objetos de esa misma clase o de subclases suyas.
- *private*: Igual que la *private* de Java. Sólo puede ser accedido desde el código de la clase a la que pertenece. Es lo considerado por defecto en C# si no se pone nada (ojo, en Java era default) para miembros de la clase.
- *internal*: Similar al *default* de Java (o no poner nada en Java). Sólo puede ser accedido desde código perteneciente al ensamblado en que se ha definido. Es el considerado por defecto si no se pone nada para clases e interfaces.
- *protected internal*: Similar al *protected* de Java. Sólo puede ser accedido desde código perteneciente al ensamblado en que se ha definido o desde clases que deriven de la clase donde se ha definido.
- *private protected*: Solo desde C# 7.2. Accesible desde la propia clase y derivados pero sólo dentro del ensamblado.

Un resumen: <https://msdn.microsoft.com/es-es/library/ba0a1yw2.aspx>

Además de lo anterior, una clase se puede marcar en C# de dos maneras:

- **sealed**: Una clase sellada es aquella de la cual no queremos permitir la herencia. En Java se puede hacer esto con el modificador *final*.
- **abstract**: Una clase abstracta es aquella que no se puede instanciar, es por tanto obligatorio crear una clase heredada para instanciar objetos (Es lógico instanciar un objeto perro pero quizá no lo sea instanciar un animal que es demasiado genérico). De esta forma una clase *abstract* se puede usar para polimorfismo y para herencia, pero no para crear objetos. En Java existe el mismo modificador. Puede verse como una mezcla entre clase e Interface.

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Llamadas entre constructores

En ocasiones es muy cómodo dentro de una clase llamar de un constructor a otro o llamar a un constructor dentro de la clase padre. Esto en Java lo hacíamos con *this()* y *super()* respectivamente con sus parámetros correspondientes cumpliendo la condición de que era lo primero que se hacía dentro del constructor llamante.

En C# lógicamente se usará *this()* y *base()*, pero en lugar de llamarlo dentro del código, la llamada se hace a continuación de la cabecera del constructor separado por dos puntos. Veámoslo con un ejemplo:

Supongamos que en Animal tenemos los constructores:

```
public Animal(int edad)
{
    this.edad = edad;
}


public Animal()
: this(0)
{
}
```

En Perro podríamos tener estos:

```
public Perro(int edad, string raza, string nombre_humano)
: base(edad)
{
    this.raza = raza;
    this.nombreHumano = nombre_humano;
    this.nombreCientifico = "Canis Familiaris";
}

public Perro()
: this(0, "", "")
{
}
```

El funcionamiento es igual que en Java, simplemente que la llamada en lugar de hacerla entre las llaves con una línea más de código se pone antes de las mismas.

	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Polimorfismo

Funciona de la misma forma que en Java, lo vemos con un ejemplo de un posible *Main* para las clases anteriores dónde además se nos muestra cómo detectar que un objeto es de **determinado tipo o heredado** con el comando **is** (similar al instanceof de Java).

```
static void Main() {
    Perro objPerro = new Perro(2, "Mastin", "Neo");
    Animal objAnimal;

    objAnimal = objPerro;
    Console.WriteLine(objAnimal.nombreCientifico);
    if (objAnimal is Perro)
    {
        Console.WriteLine(((Perro)objAnimal).nombreHumano);
    }

    Console.ReadKey();
}
```

En el caso de querer comprobar el **tipo exacto** se debe usar el método **GetType()** definido en la clase **object** y el comando **typeof()** de la siguiente forma:


```
if (objAnimal.GetType()==typeof(Perro))
{
    Console.WriteLine(((Perro)objAnimal).nombreHumano);
}
```

Este sería equivalente el uso de getClass de Java.

También puede usarse un **switch para resolver el polimorfismo** y hacer el casting al mismo tiempo (funciona como **is**), simplemente es colocar como cada una de las opciones una subclase (o la clase padre) asociada a una referencia. Prueba a sustituir el if por el siguiente switch:

```
switch (objAnimal)
{
    case Perro p:
        Console.WriteLine(p.nombreHumano);
        break;
    case Mosca m:
        Console.WriteLine("Soy la Mosca!");
        m.MuestraEdad();
        break;
}
```

Si no fueran necesarias, se podría no usar las referencias p o m, dejando solo la clase.

	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Variables de sólo lectura

Las variables de sólo lectura son similares a las constantes con la diferencia que se les puede dar un valor en un constructor por lo que pueden inicializarse de distintas maneras aunque luego su valor no cambie. Se usa la palabra reservada *readonly*.

En Animal podría definirse:

```
public readonly static string definicion = "ser vivo eucariota pluricelular";
```

Interfaces

Para crear una interfaz se usa la palabra reservada *interface*, y para implementarlo se usan también los dos puntos como si se tratara de una herencia. Si tenemos que heredar de una clase y meter una o varias interfaces usamos la coma como separador.

Veámoslo con un ejemplo:

```
interface ICorredor
{
    void correr();
}
```

Como en Java, sólo se plantea la definición. La implementación del código se hará luego en la clase que implemente la interfaz.

Podríamos definir Perro así:


```
class Perro : Animal, ICorredor
```

Si no heredara de ninguna clase, se pondría el *interface* inmediatamente después de los dos puntos.

Nomenclatura.

En C# usaremos la definida por Microsoft en:

[https://msdn.microsoft.com/es-es/library/x2dbw72\(v=vs.71\).aspx](https://msdn.microsoft.com/es-es/library/x2dbw72(v=vs.71).aspx)

	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Ejercicio 1

Crear una clase abstracta **Persona** con propiedades con set y get: Nombre, Apellidos, Edad y Dni teniendo en cuenta que:

- Edad será una propiedad int cuyo set comprobará que el dato es mayor que 0 y si no es así lo igualará a 0.
- Dni es una propiedad string. Se guarda solo el número (no es necesario hacer comprobaciones) y de forma que el get devuelve además del *número* la letra del mismo.

Debe incluir un método virtual que muestre los campos y otro que permita la introducción de los mismos.


Debe haber dos constructores: Uno que inicialice los cuatro datos y el otro sin parámetros que llame al primero con valores cadena vacía o ceros en los parámetros.

Habrà un método abstracto denominado *hacienda()* que devuelve double y no tiene parámetros.

Ejercicio 2

En el mismo proyecto que el ejercicio previo, crear una clase nueva denominada **Empleado**.

- Hereda de Persona sus propiedades y se le añade Salario, IRPF y nº de teléfono teniendo en cuenta que:
 - salario debe ser una propiedad double (con set y get) de forma que si es modificada, debe contemplar la posibilidad de cambiar el IRPF. Concretamente si el salario es menos de 600 euros, el IRPF será del 7%, si está entre 600 y 3000 será del 15% y si es mayor que 3000 euros, el IRPF será del 20%.
 - IRPF será privada y es *establecida* exclusivamente a través de salario. Debe disponer de get para que sea de solo lectura.
 - El número de teléfono es una propiedad string de forma que en el get devuelve el número con el "+34" concatenado al principio.


	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

- Debe añadirse un método que muestre los elementos tanto de la clase base como los de la propia mediante sobreescritura.
- Otro método sobrecarga al anterior mediante un parámetro entero para mostrar un único dato. Dicho parámetro indica qué dato se mostrará (0:Nombre, 1: Apellidos, etc...).
- La sobreescritura de hacienda() devuelve $IRPF * Salario / 100$.
- Tendrá también dos constructores. Uno tendrá todos los datos como parámetros y llamará al constructor de la clase base para no repetir código. Otro sin parámetros llama al primero con todos los valores nulos.

Ejercicio 3

En el mismo proyecto que el ejercicio previo, realiza otra clase denominada **Directivo**

- Hereda de Persona y añade un campo que indica el nombre del departamento que dirige, otro de porcentaje de beneficios y número de personas a su cargo.
- Número de personas a cargo será una propiedad con set y get de forma que si tiene menos de 10 personas a su cargo, los beneficios son del 2%, si tiene entre 11 y 50 será del 3.5% y si tiene más de 50 los beneficios serán del 4%.
- Sobrecarga el operador -- (mira el apéndice I y deduce como hacerlo) de forma que si se aplica a un Directivo, decremente en una unidad el porcentaje de beneficios, pero nunca bajará de 0.
- Debe añadirse un método que muestre los elementos tanto de la clase base como los de la propia y otro que pida los datos. Ambos mediante sobreescritura.
- Se creará una interfaz denominada **IPastaGansa** el cual tiene un único método denominado **ganarPasta** el cual acepta un parámetro *double* que indicará los beneficios totales en euros de una empresa y calcula y devuelve el dinero resultante de beneficios del objeto que implemente dicho interfaz.

	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

- La clase Directivo implementa dicha interfaz IPastaGansa de forma que el método ganarPasta calculará a partir del porcentaje de beneficios (propiedad del Directivo) y el dinero que gana la empresa (parámetro de ganarPasta), lo que se lleva el directivo y lo devuelve.

Si la empresa tiene pérdidas (valor negativo) el directivo se lleva 0 euros y además se decrementa sus beneficios en una unidad (debe usarse el operador -- sobrecargado).

Además de devolverlo, deja también guardado el dato en una variable privada de directivo denominada PastaGanada.

Nota: como no se puede aplicar el operador sobre this (this--), asigna this a otro objeto tipo Directivo y decrementa dicho objeto.

- La sobreescritura de hacienda() devuelve el 30% de la propiedad privada PastaGanada.
- Se crea una clase denominada **EmpleadoEspecial** que hereda de Empleado e implementa el interfaz IPastaGansa de forma que los beneficios son siempre 0.5% de la ganancia de la empresa. Además sobreescribe hacienda() para que devuelva un 0.5% más de lo que devuelve la función sobreescrita.

Ejercicio 4


El programa principal manejará tres objetos. Uno para directivo, otro para empleado y otro empleado especial. Rellena mediante código todos los datos a través de sus constructores. Realiza luego un menú con 4 opciones:

- 1.- Visualizar los datos del Directivo
- 2.- Visualizar datos Empleado
- 3.- Visualizar datos EmpleadoEspecial.
- 4.- Salir

Por supuesto al mostrar datos tienes que incluir las ganancias de Directivo y EmpleadoEspecial. Para ello haz una función estática (en la misma clase que el Main) a la que le pasas un objeto IPastaGansa y que pregunte lo que gana la empresa y que luego llame a la función ganarPasta del parámetro y muestre el

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

resultado en pantalla. Lógicamente dicha función la usarás tanto para el Directivo como para el EmpleadoEspecial. También mostrarás lo que devuelven las funciones hacienda().

	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Apéndice I: Sobrecarga de operadores

En C# encontramos una característica añadida a la sobrecarga y es que no solo se pueden sobrecargar métodos, si no que también se puede realizar la sobrecarga de un operador lo cual puede ser muy práctico a la hora de crear ciertas clases.

Veamos un ejemplo en el que se crea la clase punto y se sobrecarga el operador suma para sumar las coordenadas de dos puntos con el objetivo de que den un punto final en un sistema bidimensional:

```
using System;


class Punto
{
    public int x, y;
    public Punto(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
    public Punto()
        : this(0, 0)
    {
    }
    public static Punto operator +(Punto p1, Punto p2)
    {
        return new Punto(p1.x + p2.x, p1.y + p2.y);
    }
}

class Program
{
    public static void Main()
    {
        Punto punto1 = new Punto(2, 3);
        Punto punto2 = new Punto(5, 1);
        Punto final;
        final = punto1 + punto2;
        Console.WriteLine("Final: ({0}, {1})", final.x, final.y);
        Console.ReadKey();
    }
}
```

La línea

```
public static Punto operator +(Punto p1, Punto p2)
```

resume la sobrecarga: tiene que ser una función estática y usar la palabra reservada *operator*. Los operadores que son binarios tienen que tener los dos

	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

parámetros que serían los operandos implicados en la operación.

Si la operación es unaria pues simplemente tendrá un parámetro y no es necesario crear un punto nuevo pues se actúa (si se desea) sobre el del parámetro). Por ejemplo:

```
public static Punto operator ++ (Punto p1){
    p1.x++;
    p1.y++;
    return p1.x;
}
```

Se debe tener en cuenta que no todos los operadores se pueden sobrecargar.

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Apéndice II: Uso de new y override en la sobreescritura.

En el siguiente ejemplo sacado de

<https://docs.microsoft.com/es-es/dotnet/csharp/programming-guide/classes-and-structs/known-when-to-use-override-and-new-keywords>

se observa la diferencia del uso de new y override al aplicar polimorfismo:

```
using System;
using System.Text;

namespace OverrideAndNew
{
    class Program
    {
        static void Main(string[] args)
        {
            BaseClass bc = new BaseClass();
            DerivedClass dc = new DerivedClass();
            BaseClass bcdc = new DerivedClass();

            // The following two calls do what you would expect. They call
            // the methods that are defined in BaseClass.
            bc.Method1();
            bc.Method2();
            // Output:
            // Base - Method1
            // Base - Method2

            // The following two calls do what you would expect. They call
            // the methods that are defined in DerivedClass.
            dc.Method1();
            dc.Method2();
            // Output:
            // Derived - Method1
            // Derived - Method2

            // The following two calls produce different results, depending
            // on whether override (Method1) or new (Method2) is used.
            bcdc.Method1();
            bcdc.Method2();
            // Output:
            // Derived - Method1
            // Base - Method2
        }
    }
}
```

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

```

class BaseClass
{
    public virtual void Method1()
    {
        Console.WriteLine("Base - Method1");
    }

    public virtual void Method2()
    {
        Console.WriteLine("Base - Method2");
    }
}

class DerivedClass : BaseClass
{
    public override void Method1()
    {
        Console.WriteLine("Derived - Method1");
    }

    public new void Method2()
    {
        Console.WriteLine("Derived - Method2");
    }
}

```