Tema 2: C# avanzado

Asignatura: Desarrollo web en entorno servidor

CS Desarrollo de Aplicaciones Web

Introducción

- En este capítulo veremos aspectos como:
  - Objetos
  - Clases
  - Herencia
  - Polimorfismo
  - Interfaces

Concepto de POO

- POO es un conjunto de reglas a seguir para hacernos la tarea de programar más fácil.
- Estas reglas son independientes del lenguaje en que trabajemos, que es un conjunto de instrucciones entendibles directamente o traducibles al lenguaje del ordenador con el que trabajemos

Concepto de POO

- Para comprender mejor el concepto de POO, pensemos en el proceso de conducción de un vehículo.

- En un coche podemos identificar **entidades** como puede ser el chasis, el motor, las ruedas, etc.

Concepto de POO

- Un vehículo puede verse como un objeto que tiene unos **atributos**:

  fabricante, modelo, color, potencia, velocidad, número de la marcha, etc.;

- Tiene un **conjunto de acciones** que puede realizar como arrancar, acelerar, frenar, parar el motor, cambiar de marcha, girar, etc.

Concepto de POO

- Por otro lado, pensemos en el conductor del vehículo.
- Desde el punto de vista de la conducción, el conductor tiene unos **atributos**: nombre, edad, antigüedad del carnet, etc.
- Las **acciones o métodos** que puede realizar un conductor serían pisar el freno, pisar el acelerador, pisar el embrague, encender las luces, etc.

Concepto de POO

- Si con el vehículo ya en marcha queremos disminuir la velocidad, el **objeto conductor** enviaría el **mensaje frenar** al **objeto vehículo**.

- Este mensaje lo realiza mediante la **acción pisar el freno**. La respuesta a este mensaje sería la ejecución por parte del vehículo de la **acción o método frenar.**

Concepto de POO

- Es el vehículo el que realiza las acciones necesarias usando el sistema de frenado para disminuir la velocidad.

- En este proceso, **al conductor** no le interesa cómo está construido y cómo actúan los frenos, **sólo quiere que el objeto vehículo responda** de forma adecuada al mensaje, **ejecutando el método frenar**.

Concepto de POO

- Como vimos en este ejemplo, todos los elementos son objetos que se relacionan entre sí o se componen de otros objetos.
- **Este conjunto de objetos dialogan entre sí a través de ‘mensajes’ para realizar tareas.**

Clases y Objetos

- Como hemos visto, en la POO debemos definir objetos y sus relaciones. El mecanismo básico de este esquema se basa en el concepto de **clase**.

Clases y Objetos. Clases

- Una **clase define una plantilla o molde para aquellos objetos que comparten características comunes**.
- Como una clase es un concepto abstracto, hay que definirlo abstrayendo las cualidades comunes de una serie de objetos.
- Por ejemplo, en la clase vehículo ¿**qué es lo común en los vehículos**?

Clases y Objetos. Clases

- Un objeto es una **instancia de la clase** con sus cualidades particulares.
- Por ejemplo: Citroen C4 rojo 115 CV, Renault Megane blanco 100 CV etc., serían objetos de la clase vehículo.
- Cada uno de ellos es diferente a otro; sus atributos son distintos, pero todos ellos realizan las mismas acciones: frenar, acelerar, girar, etc.

Clases y Objetos. Clases

- En este ejemplo, definiríamos las clases **vehículo y conductor**.
- Luego crearíamos objetos concretos, de dichas clases:
  - (Jaguar 300, blanco, 180 CV),
  - (Martín, 30, 12).

Clases y Objetos. Clases

- Con la POO, si queremos construir un objeto que comparte ciertas cualidades con otro que ya tenemos creado, no tenemos que volver a crearlo desde el principio; simplemente, decimos qué queremos usar del antiguo en el nuevo y qué nuevas características debe tener nuestro nuevo objeto.

- A definir objetos a partir de otros existentes, **se conoce como herencia**

FORMACIÓN PROFESIONAL MONTECASTELO

Actividad 1

Crea la clase Persona.

- *Define sus atributos:*
  - ***Nombre**: de tipo string, que representa el nombre de la persona.*
  - ***Apellidos**: de tipo string, que representa los apellidos de la persona.*
  - ***Edad**: de tipo int, que representa la edad de la persona.*
  - ***Genero**: de tipo string, que representa el género de la persona.*
  - ***Direccion**: de tipo string, que representa la dirección de la persona.*
  - ***Email**: de tipo string, que representa la dirección de correo electrónico de la persona.*

- *Define sus métodos::*
  - ***Presentarse**(): que devuelve un mensaje de presentación de la persona.*
  - ***EnviarCorreo(string** mensaje): que envía un correo electrónico a la dirección de correo electrónico de la persona con un mensaje específico.*
  - ***EsMayorDeEdad**(): que devuelve true si la persona tiene 18 años o más, y false en caso contrario.*

- *Crea tres objetos de tipo "Persona" llamados “profe1", “profe2" y “profe3".*
- *Oblígalos a utilizar los métodos después de crearlos.*

10 MINUTOS

Diseño de clases

- Las clases en C# se organizan en Namespaces (paquetes en Java)

```
namespace Space { class Class1 {...} .... }
```

- Para acceder a una clase podemos usar:

```
Space.Class1
```

```
using Space;
```

Diseño de clases

- A continuación, vemos un ejemplo de clase:

```
class Cuenta
{
    private string nombre; // Nombre del titular
    private string cuenta; // Número de cuenta
    private double saldo; // Saldo actual de la cuenta
    private double tipoDeInterés; // Tipo de interés en tanto por cien.
    // . . .
}
```

Diseño de clases

- Normalmente los atributos de un objeto de una clase se ocultan a los usuarios del mismo.
- Por ejemplo, un usuario que utilice la clase Cuenta **no podrá escribir código que manipule directamente estos atributos**.
- Tendrá que acceder a ellos a través de los métodos.

Diseño de clases

- **Esta protección se consigue usando el modificador private** (cuando se omite el modificador se supone private).

- **Un miembro que se declare privado es accesible solamente por los métodos de su propia clase.** Esto implica que no se puede acceder al miembro a través de métodos de cualquier otra clase.

Diseño de clases

- **Los modificadores de acceso son:**

public	La clase o miembro es accesible en cualquier ámbito.
protected	Se aplica sólo a miembros de la clase. Indica que sólo es accesible desde la propia clase y desde las clases derivadas.
private	Se aplica a miembros de la clase. Un miembro privado sólo puede utilizarse en el interior de la clase donde se define, por lo que no es visible tampoco en clases derivadas.
internal	La clase o miembro sólo es visible en el proyecto (ensamblado) actual.
internal protected	Visible en el proyecto (ensamblado) actual y también visible en las clases derivadas.

Diseño de clases

- Algunas acciones que el objeto de clase Cuenta puede realizar serían:
  - asignar el nombre de un cliente a una cuenta
  - obtener el nombre del cliente de una cuenta
  - asignar el número de cuenta
  - obtener el número de cuenta
  - realizar un ingreso
  - realizar un reintegro, etc.

Diseño de clases. Ejemplo completo

- El típico ejemplo para instanciar una clase es:

```
Cuenta c1 = new Cuenta();
```

- **Sin embargo, ¿qué significa esta sentencia? Es una forma simple de:**

```
Cuenta c1; // Se declara c1 como referencia de tipo Cuenta
c1 = new Cuenta(); // c1 referencia al objeto tipo Cuenta recién creado
```

Constructores

- Un constructor es un método especial de una clase que es llamado automáticamente siempre que se crea un objeto de dicha clase.

- **Su función es inicializar el objeto**

Constructores

- El constructor se identifica con un nombre que coincide con el nombre de la clase a la que pertenece y no tiene valor de retorno, ni siquiera void.
- Cuando en una clase no se define ningún constructor, C# asume uno por omisión.

Constructores

- Por ejemplo, en nuestra clase Cuenta, no hemos definido ningún constructor, por lo tanto se asume uno cuya definición sería:

```
public Cuenta()
{ }
```

Constructores

- **El constructor por omisión no tiene parámetros y no hace nada.**
- Sin embargo es necesario que esté definido ya que será invocado cada vez que se construya un objeto sin especificar ningún argumento.
- En este caso el objeto será iniciado con los valores predeterminados: los atributos de tipo numérico a cero, y las referencias a null.

Constructores

- Por ejemplo, en esta sentencia:

```
Cuenta c1 = new Cuenta();
```

- El operador new es quien ‘crea’ un nuevo objeto (reserva la memoria y retorna su referencia); a continuación, se invoca al constructor que ‘inicia’ los atributos. En este caso, con los valores por defecto.

Constructores

- En general, se define un constructor con los atributos de la clase:

```
public Cuenta(string nom, string cue, double sal, double tipo)
{
    nombre = nom;
    cuenta = cue;
    saldo = sal;
    tipoDeInteres = tipo;
}
```

Constructores

- A continuación, usamos el operador *new* para llamar al constructor:

```
Cuenta c2 = new Cuenta("Martin", "12345678Z", 100, 7.5);
```

Constructores

- Los constructores normalmente se definen públicos para que se puedan invocar desde cualquier parte.
- Cuando se define un constructor, el constructor por omisión es reemplazado por éste.

Constructores

- ¿Podemos tener dos constructores?
- **La respuesta es SÍ**. Esto se conoce como **sobrecarga de métodos**.
- A continuación veremos una sobrecarga del constructor de la clase Cuenta.

Constructores

```
public Cuenta(string nom, string cue, double sal, double tipo)
{
    nombre = nom;
    cuenta = cue;
    saldo = sal;
    tipoDeInteres = tipo;
}

public Cuenta(string nom, string cue, double sal)
{
    nombre = nom;
    cuenta = cue;
    saldo = sal;
    tipoDeInteres = 0;
}
```

Constructores

- Utilizando estos constructores, crearemos los objetos:

```
Cuenta c2 = new Cuenta("Martin", "12345678Z", 14, 2);

Cuenta c3 = new Cuenta("Marcos", "12345785A", 7);
```

- En cada una de las sentencias utilizamos un constructor diferente, según el número de parámetros que le pasemos.

FORMACIÓN PROFESIONAL MONTECASTELO

Actividad 2

Define un constructor para la clase Persona pasándole todos los atributos

¿Puedes definir otro constructor? Crea objetos con los dos constructores.

¿Qué pasa con el constructor por omisión?

5 MINUTOS

This

- ¿Cómo sabe un método de una clase sobre qué objeto está trabajando si en el cuerpo de dicho método no se indica nada de forma explícita?
- Para hacer referencia explícita dentro de la clase al objeto que llama al método usaremos **this.**

This

- Por ejemplo, ¿qué diferencia hay entre estos métodos de la clase Cuenta?

```
public string obtenerNombre()
{
    return nombre;
}
```

```
public string obtenerNombre()
{
    return this.nombre;
}
```

This

- Por ejemplo, ¿qué diferencia hay entre estos métodos de la clase Cuenta?

```
public string obtenerNombre()
{
    return nombre;
}
```

```
public string obtenerNombre()
{
    return this.nombre;
}
```

- **NO HAY DIFERENCIA,** pues C# utiliza **this** de forma implícita

Propiedades

- **Aparecen en la clase como campos de datos.**
- Las propiedades tienen un tipo, pero la asignación y lectura de las mismas se realiza a través de métodos de lectura y escritura: **get y set**
- Con las propiedades, **se puede limitar el acceso a un campo permitiendo sólo la lectura o sólo la escritura**

Propiedades

- Para definir una propiedad, se usa la siguiente sintaxis:

```
<tipoPropiedad> <nombrePropiedad>
{
    set
    {
        <códigoEscritura>
    }
    get
    {
        <códigoLectura>
    }
}
```

Propiedades

- Por ejemplo:

```
public int edad
{
    set
    {
        this.edad = edad;
    }
    get
    {
        return this.edad;
    }
}
```

Propiedades. Ejemplo completo

- Se va a redefinir la clase Cuenta pero usando propiedades que sustituyan a los métodos de acceso a los campos:

Propiedades. Ejemplo completo

```
class Cuenta
{
    // Atributos
    private string nombre;
    private string cuenta;
    private double saldo;
    private double tipoDeInterés;

    // Propiedades
    public double Saldo
    {
        get {return saldo;} // Propiedad de sólo lectura
    }

    public string Nombre
    {
        get {return nombre;}

        set
        {
            if (value == null || value.Length == 0)
            {
            System.Console.WriteLine("Error: cadena vacía");
            return;
            }
            nombre = value;
        }
    }
    public string Cuenta
    {
        get {return cuenta;}
        set
        {
            if (value == null || value.Length == 0)
            {
            System.Console.WriteLine("Error: cuenta no válida");
            return;
            }
            cuenta = value;
        }
    }
}
```

Propiedades.

- El **acceso get** contiene el código de la lectura de la propiedad y por tanto del atributo que se maneja. **Debe terminar siempre con la sentencia return o throw.**

- El **acceso set** contiene el código para escribir un valor en la propiedad y por tanto en el atributo que se maneja. **El parámetro value es el valor que se le da a la propiedad.**

Propiedades.

- Un ejemplo de uso de la clase Cuenta usando propiedades podría ser:

```
class PruebaPropiedades
{
    public static void Main(string[] args)
    {
        Cuenta c6 = new Cuenta();
        c05.Nombre = "Martin";
        c05.Cuenta = "12345";
        c05.TipoDeInterés = 2.5;
        c05.ingreso(250);
        c05.reintegro(50);
        System.Console.WriteLine(c05.Nombre);
        System.Console.WriteLine(c05.Cuenta);
        System.Console.WriteLine(c05.Saldo);
        System.Console.WriteLine(c05.TipoDeInterés);
    }
}
```

Herencia

- **Es el mecanismo para definir una nueva clase** (clase hija) **partiendo de una clase existente** (clase padre)
- La herencia permite la reutilización de código.
- La clase hija hereda todos los miembros de su clase padre, excepto los que son privados.

Herencia

- Todas las clases derivan implícitamente de Object.
- C# sólo permite heredar de una sola clase.
- Si una clase hereda de otra clase se indica tras el nombre de la clase con dos puntos y el nombre de la clase de la cual hereda:

```
<Nombre_Clase_Hija>:<Nombre_Clase_Padre>
```

Herencia

- Para llamar al constructor de la clase base:

```
public Suma(int numero) : base(numero){ … }
```

- **Cuando no interesa que se derive de una clase** (porque se trata de una clase final) **se usa el modificador sealed** (sellada) **delante de class**:

```
public sealed class perro: animal{...}
```

Herencia

- Si la clase sólo debe actuar como clase base de otras clases se denomina **clase abstracta** y se usa el modificador ***abstract*** en la declaración:

```
public abstract class animal {...}
```

- No es posible instanciar objetos directamente de una clase abstracta, es necesario heredar.

FORMACIÓN PROFESIONAL MONTECASTELO

Actividad 5

Crea las siguientes clases:

- *Libro* (Titulo, Precio, Autor)
- *DVD* (Titulo, Precio, Duracion)

Crea 2 libros y 2 DVDs y **muestra sus datos por pantalla**

¿Cómo harías para evitar que se repita el mismo código en las dos clases?

10 MINUTOS

Herencia. Ejemplo completo

- Vamos a diseñar una jerarquía de clases usando nuestra famosa clase Cuenta como clase base de otras dos: **CuentaCorriente** y **CuentaAhorro**.
- Estas dos clases derivan directamente de la clase Cuenta y como cualquier clase que definamos, siempre derivarán indirectamente de la clase Object.
- Todos los atributos y métodos de la clase base Cuenta se heredan, excepto los constructores y el destructor.

Herencia. Ejemplo completo

- Por lo tanto, las clases serían:

```
class CuentaAhorro : Cuenta
{ }


class CuentaCorriente : Cuenta
{ }
```

Herencia. Ejemplo completo

- Por ejemplo, vamos a añadir a la clase CuentaAhorro:
  - Dos constructores, uno con parámetros y otro sin ellos.
  - Un nuevo atributo cuotaMantenimiento
  - Los métodos asignarCuotaMantenimiento y obtenerCuotaMantenimiento para poder manipularlo

Herencia. Ejemplo completo

```
class CuentaAhorro : Cuenta
{
    private double cuotaMantenimiento; // Se añade un nuevo atributo
    public CuentaAhorro() {}
    public CuentaAhorro(string nom, string cue, double sal, double tipo,
    double mant) : base(nom, cue, sal, tipo)
    {
        asignarCuotaMantenimiento(mant);
    }
    public void asignarCuotaMantenimiento (double cantidad)
    {
    if (cantidad < 0)
        System.Console.WriteLine("Error: cantidad negativa");
    else
        cuotaMantenimiento = cantidad;
    }
    public double obtenerCuotaMantenimiento()
    {
        return cuotaMantenimiento;
    }
}
```

Herencia. Ejemplo completo

- Como se ve en el código del ejemplo, para llamar al constructor de la clase base se usa la notación con los dos puntos seguidos de la palabra reservada base con los parámetros que necesite encerrados entre paréntesis.

- Podemos construir objetos así:

```
CuentaAhorro c_a1 = new CuentaAhorro();
CuentaAhorro c_a2 = new CuentaAhorro("Martin","1234567891",600, 2.5, 0.75);
```

Redefinir métodos en la subclase

- Una clase derivada puede redefinir un método heredado si se necesita modificar su funcionalidad.
- En la redefinición sólo se puede modificar el cuerpo, no la firma del método.

Redefinir métodos en la subclase

- Por ejemplo, si en las cuentas de ahorro se necesita disponer de un saldo superior a 1.500€ con un interés mayor o igual a 3,5 para poder hacer un reintegro, debemos redefinir el método **reintegro**():

Redefinir métodos en la subclase

```
class CuentaAhorro : Cuenta
{

    public void reintegro(double cantidad) // Redefinimos el método heredado
    {
    double sal = estado();
    double tipo = obtenerTipoInteres();

    if (tipo >= 3.5)
    {
        if(sal - cantidad >= 1500)
            base.reintegro(cantidad); // Método reintegro de la clase base
        else
            System.Console.WriteLine(("Saldo insuficiente");
    }
    else
        System.Console.WriteLine(("Tipo de interés insuficiente");
    }
}
```

Protección de miembros

- Pregunta: **desde una clase hija, ¿podemos acceder libremente a todos los miembros heredados?.**
- Respuesta: **NO**. **Sólo se puede acceder a los miembros públicos**, los privados siguen siendo inaccesibles.

Protección de miembros

```
class CuentaAhorro : Cuenta
{
    public void reintegro(double cantidad) // Redefinimos el método heredado
    {
    if (tipoDeInteres >= 3.5)
    {
        if(saldo - cantidad >= 1500)
            base.reintegro(cantidad);
        else
            System.Console.WriteLine("Saldo insuficiente");
    }
    else
        System.Console.WriteLine("Tipo de interés insuficiente");
    }
}
```

FORMACIÓN PROFESIONAL MONTECASTELO

Actividad 3

Revisa el código anterior.

¿Es correcto? ¿Daría algún error de compilación?

¿Si diera error, cómo lo modificarías?

5 MINUTOS

Protección de miembros

- **Los miembros privados** ya sean campos, métodos, propiedades, etc. **no son accesibles desde fuera de la clase**, ni desde una clase derivada.
- ¿Hay alguna solución intermedia para poder tener miembros de la clase base que sólo fueran accesibles desde la clase derivada, pero que mantuvieran su privacidad para el resto de las clases?
- **SÍ. Son los miembros protegidos (protected).**

Protección de miembros

- **Un miembro protected, es accesible desde las clases hijas, pero es inaccesible desde fuera de la jerarquía.**
- Es decir, se comporta como público para la clase derivada y privado para el resto de las clases o usuarios de la clase.
- De esta forma, si los atributos saldo y tipoDeInteres se hubieran declarados en la clase base con el nivel de protección protected, el código anterior sería perfectamente válido y compilaría sin problemas.

Métodos virtuales

- Una clase derivada hereda de su clase base todos sus miembros (excepto los constructores y el destructor), y los métodos heredados podemos ocultarlos mediante redefiniciones de éstos en la clase derivada.

Métodos virtuales. Ejemplo

```
class Persona
{
    public string nombre;
    public void escribir()
    {
        System.Console.WriteLine(nombre);
    }
}
class Profesor : Persona
{
    public string depar; // Nuevo atributo
    new public void escribir() // Redefinición del método
    {
        System.Console.Writeline("Profesor " + nombre + " del Departamento de " + depar);
    }
}
class Prueba1
{
    public static void Main(string[] args)
    {
        Persona per = new Persona();
        Profesor pro = new Profesor();
        per.nombre = "Ramón";
        per.escribir();
        pro.nombre = "Martín";
        pro.depar = "Informática";
        pro.escribir();
    }
}
```

Métodos virtuales. Ejemplo

- La salida es:

```
Ramón
Profesor Martín del Departamento de Informática
```

- Podemos ver que, según el objeto que llame al método ***escribir()***, la salida va a ser la de la clase Persona o Profesor.

Métodos virtuales. Ejemplo

- Los métodos de las clases base no siempre quedan perfectamente ocultos al redefinirse en las clases derivadas.
- La redefinición de métodos, y por tanto la ocultación del método de la clase base con new funciona bien si el objeto derivado es referenciado a través de una referencia de su propia clase, pero si es referenciado a través de una referencia de la clase base el mecanismo deja de funcionar.
- Si no queremos que esto ocurra, deberemos usar los **métodos virtuales.**

Métodos virtuales. Ejemplo

- Usando **métodos virtuales** conseguimos que **los métodos invocados se correspondan con los definidos para los objetos referenciados, y no con los del tipo de referencia.**

- Para implementar métodos virtuales debemos indicar al compilador con la palabra reservada ***virtual*** el método de la clase base que queremos reemplazar, y usar la palabra reservada ***override*** en el método de la clase derivada que redefine al de la clase base.

Métodos virtuales. Ejemplo

```
class Persona
{
    public string nombre;
    virtual public void escribir() // Método virtual. Puede ser redefinido
    {
    System.Console.WriteLine(nombre);
    }
}
class Profesor : Persona
{
    public string depar;
    override public void escribir() // Método de reemplazo
    {
    System.Console.Writeline("Profesor " +nombre+ " del Departamento de " + depar);
    }
}
class Prueba1
{
    static void Main(string[] args)
    {
        Persona per = new Persona();
        Profesor pro = new Profesor();
        per.nombre = "Ramón";
        per.escribir();
        pro.nombre = "Martín";
        pro.depar = "Informática";
        pro.escribir();
        per = pro; // Una referencia base señala a un objeto derivado
        per.escribir(); // ahora, el método que se ejecuta es el del objeto
        // derivado ya que el método base es virtual
    }
}
```

Métodos virtuales. Ejemplo

- En este caso se producirá la siguiente salida:

```
Ramón
Profesor Martín del Departamento de Informática
Profesor Martín del Departamento de Informática
```

Polimorfismo

- Polimorfismo es la propiedad que indica, literalmente, la posibilidad de que una entidad tome muchas formas.
- En términos de la POO, **el polimorfismo permite hacer referencia a objetos de clases diferentes mediante el mismo elemento de programa y realizar la misma operación pero de diferentes formas, según sea el tipo del objeto que sea referenciado en ese momento**.

Polimorfismo

- Por ejemplo, cuando se describe la clase vehículo:
  - Las operaciones desplazar o frenar son fundamentales en todos los vehículos, de modo que cada tipo de vehículo debe poder realizar las operaciones de desplazar y frenar.
  - Por otra parte, las clases avión, automóvil, bicicleta, que son subclases de vehículo deben tener sus propias formas de realizar las operaciones de desplazar o frenar.
- El polimorfismo implica que, si tenemos una referencia de tipo vehículo, y le enviamos el mensaje desplazar, la acción que se ejecutará será diferente según el tipo del objeto referenciado

Polimorfismo

```
class Persona // Clase base
{
    public string nombre;
    public virtual void escribir() // Método virtual
    {
    System.Console.WriteLine(nombre);
    }
}
class Profesor : Persona
{
    public string depar;
    public override void escribir() // Método de reemplazo
    {
    System.Console.Writeline("Profesor " +nombre+ " del Departamento de " + depar);
    }
}
class Alumno : Persona
{
    public string curso;
    public override void escribir() // Método de reemplazo
    {
    System.Console.WriteLine("Alumno " + nombre + " del curso " + curso);
    }
}
class Prueba3
{
    static void Main(string[] args)
    {
        Profesor pro = new Profesor();
        Alumno alu = new Alumno();
        Persona per; // per es una referencia, ¡no un objeto!
        pro.nombre = "Ramón";
        pro.depar = "Informática";
        alu.nombre = "Martín";
        alu.curso = "2°Des.Interfaces";
        // Se activa en cada caso el método correspondiente al objeto referenciado
        per = pro; // per referencia a un profesor(clase derivada)
        per.escribir(); // actúa el método de reemplazo de profesor
        per = alu; // ahora per referencia a un alumno
        per.escribir(); // actúa el método de reemplazo de alumno
        Console.ReadLine();
    }
}
```

Polimorfismo. Ejemplo

- En este caso se producirá la siguiente salida:

```
Profesor Ramón del Departamento de Informática
Alumno Martín del curso 2ºDes.Interfaces
```

Polimorfismo. Ejemplo

- **El resultado es justo lo que queríamos.**
- Si per, que es una referencia de tipo Persona, referencia un objeto de la clase derivada Profesor el método que se activa al ejecutar per.escribir() es el redefinido para Profesor.
- Si referencia a un objeto de la clase Alumno, el método que se activa al invocar per.escribir() es el redefinido para la clase Alumno.

Clases abstractas

- Como vimos anteriormente, **la función de una clase abstracta es la de agrupar miembros comunes de otras clases que se derivarán de ellas.**
- Por ejemplo, se puede definir la clase vehículo como abstracta para después derivar de ella las clases avión, bicicleta, patinete, etc., pero todos los objetos que se declaren pertenecerán a alguna de estas últimas clases; no habrá vehículos que sean sólo vehículos.

Clases abstractas

- Las clases que derivemos de la clase abstracta vehículo, están obligadas a definir los métodos que sólo están declarados en la clase vehículo ya que de las clases derivadas si podemos instanciar objetos, y por lo tanto debe existir una definición concreta de los métodos.

Clases abstractas

- Los métodos abstractos son también por definición métodos virtuales y debe por lo tanto usarse la palabra clave ***override*** para reemplazarlos en las clases derivadas.

Clases abstractas. Ejemplo

- Vamos a modificar el ejemplo anterior de forma que la clase Persona la vamos a declarar abstracta:
  - El método escribir() lo vamos a marcar como abstracto. Esto implicará que no podamos declarar objetos de la clase Persona, aunque si podremos declarar referencias de Persona.
  - Por otro lado deberemos obligatoriamente implementar el método escribir() en las clases derivadas ya que en la clase Persona no se implementa.

Clases abstractas. Ejemplo

```
abstract class Persona // Clase base abstracta
{
    public string nombre;
    public abstract void escribir(); // Método abstracto ¡sin implementación!
    // No olvidar escribir el ;
}
class Profesor : Persona
{
    public string depar;
    public override void escribir() // Método de reemplazo
    {
    System.Console.Writeline("Profesor " + nombre + " del departamento de " + depar);
    }
}
class Alumno : Persona
{
    public string curso;
    public override void escribir() // Método de reemplazo
    {
    System.Console.WriteLine("Alumno " + nombre + " del curso " + curso);
    }
}
class Prueba4
{
    static void Main(string[] args)
    {
        Profesor pro = new Profesor();
        Alumno alu = new Alumno();
        Persona per; // per es una referencia, ¡no un objeto!
        pro.nombre = "Carlos";
        pro.depar = "Informática";
        alu.nombre = "Luis";
        alu.curso = "2º Des.Interfaces";
        // Se activará en cada caso el método correspondiente al objeto referenciado
        per = pro; // per referencia a un profesor(clase derivada)
        per.escribir(); // actúa el método de reemplazo de profesor
        per = alu; // ahora per referencia a un alumno (clase derivada)
        per.escribir(); // actúa el método de reemplazo de alumno
        // Persona Pepe = new Persona(); ¡ERROR!. No se pueden crear una Persona
    }
}
```

FORMACIÓN PROFESIONAL MONTECASTELO

Actividad 4

Crea las siguientes clases:

- *Libro* (Titulo, Precio, Autor)
- *DVD* (Titulo, Precio, Duracion)
- Utiliza una clase abstracta *Publicacion* para facilitar el proceso
- Esta clase tiene un método abstracto “**MostrarTablaInfoHTML()”**

Crea 2 libros y 3 DVDs. ¿Puedes crear objetos *Publicacion*? ¿Por qué?

5 MINUTOS

Interfaces

- Una interfaz es una **declaración de un conjunto de miembros para los que no se da implementación**, sino que se declaran de manera similar a como se declaran los métodos abstractos.

- Podría verse como una **forma especial de definir clases que sólo contarán con miembros abstractos.**

Interfaces

- Cuando una clase implementa una interfaz, se garantiza que soportará los métodos, propiedades, eventos e indexadores declarados en la misma.
- Para que una clase implemente una interfaz debe incluir en ella el código de los métodos, propiedades, eventos e indexadores declarados en la interfaz.

Interfaces

- **No es igual heredar de una clase abstracta que implementar una interfaz**
- Un coche, que es un vehículo, hereda las características y comportamiento de un vehículo, pero puede tener la capacidad de AjustarTemperatura (como una casa, o una incubadora) que no está definida en la clase vehículo.

Interfaces

- La estructura de una interfaz es la siguiente:

```
[<modificadores>] interface <nombre> [:<interfacesBase>]
{
<miembros>
}
```

Interfaces

- Los **<miembros>** de las interfaces pueden ser declaraciones de métodos, propiedades, indizadores o eventos, pero no campos, operadores, constructores o destructores.

- La sintaxis que se sigue para definir cada tipo de miembro es la misma que para definirlos como abstractos pero sin incluir abstract:

Interfaces

- **Métodos**: <tipoRetorno> <nombreMétodo>(<parámetros>);
- **Propiedades:** <tipo> <nombrePropiedad> {set; get;}
- **Indizadores:** <tipo> this[<índices>] {set; get;}
- **Eventos:** event <delegado> <nombreEvento>;

Interfaces. Ejemplo 1

- Por ejemplo, queremos crear una interfaz que defina la capacidad de ‘ser imprimible’ que se llame IImprimible y que tenga un solo método que se llame imprimir().

```
interface IImprimible
{
    void imprimir();
}
```

Interfaces. Ejemplo 2

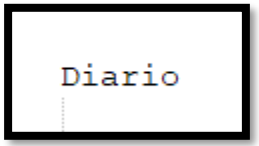
- Creemos una clase Documento que permita almacenar un texto.
- Para indicar que el tipo Documento ‘se puede imprimir’ bastaría con que implemente la interfaz **IImprimible**.

Interfaces. Ejemplo 2

```
using System;
namespace Pruebas
{
    interface IImprimible // Definición de interfaz
    {
        void imprimir();
    }
    class Documento : IImprimible // Otorgamos las capacidades de IImprimible
    {
        string contenido;
        public Documento(string frase)
        {
        contenido = frase;
        }
        public void imprimir() //Obligados a implementar el método de interfaz
        {
        Console.WriteLine(contenido);
        }
    }
    class PruebaInterfaz
    {
        static void Main(string[] args)
        {
        Documento d = new Documento("Diario");
        d.imprimir();
        }
    }
}
```

Interfaces. Ejemplo 2

- La salida de este programa es:



- Recordad que todos los miembros de una interfaz son públicos por defecto, para que puedan ser implementados por otras clases.

Interfaces. Ejemplo 3

- En este ejemplo vamos a **definir una propiedad de interfaz.**
- Recordemos que una propiedad está relacionada con funciones de acceso get y set que definen el código que debe ejecutarse cuando se lee o escribe el valor de la propiedad.

Interfaces. Ejemplo 3

- Supongamos que añadimos a la interfaz IImprimible la **propiedad de interfaz de nombre Tiempo.**

- Es de tipo entero y habrá que dar una definición para la clase que la implemente.

```
interface IImprimible // Definición de interfaz
{
    void imprimir(); // Método de interfaz
    int Tiempo // Propiedad de interfaz
    {
        get;
        set;
    }
}
```

Interfaces. Ejemplo 4

- La clase Documento, deberá añadir código para implementar la propiedad de interfaz Tiempo.

```
class Documento : IImprimible // Otorgamos las capacidades de IImprimible
{
    private string contenido;
    private int duracion; // nº de minutos empleados en escribir el documento
    public Documento(string frase)
    {
        contenido = frase;
    }
    public void imprimir() // Implementamos el método de interfaz
    {
        Console.WriteLine(contenido);
    }
    public int Tiempo // Implementamos la propiedad de interfaz
    {
        get { return duracion; }
        set { duracion = value; }
    }
}
```

Interfaces. Ejemplo 5

- Por último, si al código anterior le añadimos la siguiente clase:

```
class PruebaInterfaz
{
    static void Main(string[] args)
    {
        Documento d = new Documento("Ensayo Nocturno");
        d.Tiempo = 37;
        d.imprimir();
        Console.WriteLine("Se ha redactado en " + d.Tiempo + " minutos");
        Console.ReadLine();
    }
}
```

Interfaces. Ejemplo 5

- La salida de este programa es:

```
Ensayo Nocturno
se ha redactado en 37 minutos
```

Operadores is y as con interfaces

- El **operador is** se utiliza para comprobar en tiempo de ejecución si el tipo de un objeto es compatible con un tipo dado.

- Se usa de la forma:

```
<expresion> is tipo
```

Operadores is y as con interfaces

- <expresion> debe ser un tipo referencia.
- La expresión is se evalúa como true si <expresion> es del tipo tipo, o puede convertirse a tipo.

Operadores is y as con interfaces

- Podemos usar este operador para comprobar si un objeto soporta una interfaz. Por ejemplo:

```
if(documento is IImprimible)
{
    Console.WriteLine("Soporta la interface IImprimible");
}
```

- Comprobamos si el objeto *documento* soporta la interfaz.

Operadores is y as con interfaces

- El operador as combina el operador is y una conversión de tipos (casting).

```
<expresion> as tipo
```

- Primero se comprueba si la conversión es válida y si es así, se realiza la conversión retornándose la <expresion> convertida al tipo. En caso contrario se devuelve null.

KEEP CALM IT'S KAHOOT TIME

Tema 2: C# avanzado

Asignatura: Desarrollo web en entorno servidor

CS Desarrollo de Aplicaciones Web