

Módulo 2

Principios de OOP

Tecnara – Cursos de formación



Índice de contenidos

1. Encapsulación
2. Herencia
3. Abstracción
4. Polimorfismo

01

Encapsulación

Principios de OOP

En la siguiente sección vamos a ver los cuatro principios que definen la programación orientada a objetos:

- Encapsulación
- Herencia
- Abstracción
- Polimorfismo

Principios de OOP: Encapsulación

El principio de encapsulación plantea:

- restringir el acceso desde el exterior de una clase a toda la información sensible o no relevante para el exterior que los objetos de esa clase puedan contener
- En otras palabras, evitar que, tanto propiedades como métodos que no se desee que sean accesibles desde el exterior, lo sean. Para ello, existen los modificadores `private` y `public` (de momento).
- El modificador `private` aplicado a una propiedad o método evita su acceso externo
- El modificador `public` aplicado a una propiedad o método permite su acceso externo

Principios de OOP: Encapsulación

En C# se cumple este principio:

- Todas las propiedades de una clase son privadas por defecto
 - En C#, si declaras una propiedad sin modificador, el modificador que se aplica por defecto es `private`. Si quieres que la propiedad sea pública tienes que escribir `public` explícitamente
- Todos los métodos de una clase son privados por defecto
 - En C#, si declaras un método sin modificador, el modificador que se aplica por defecto es `private`. Si quieres que el método sea público tienes que escribir `public` explícitamente

Principios de OOP: Encapsulación

- Llegados a este punto, el profesor reconoce que os ha mentado... Os ha mentado descaradamente... 😏 ¡pero lo ha hecho por vuestro bien, para no romperos la cabeza desde el minuto cero!
- REALMENTE, los datos de un objeto de una clase determinada NO están guardados en las propiedades (en inglés, properties) de la clase, sino en los CAMPOS (en inglés, fields) de la clase.
- Las propiedades de una clase realmente solo definen métodos que permiten leer (get) o escribir (set) sobre un campo existente de la clase:

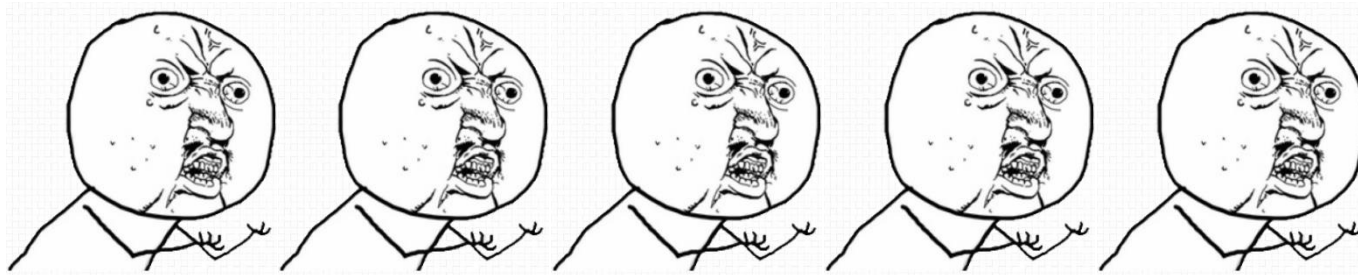
```
public class Ejercicio1
{
    private int _numero1;

    public int num1
    {
        get { return _numero1; }
        set { _numero1 = value; }
    }
}
```

Principios de OOP: Encapsulación

- Sabiendo esto, igual os estáis preguntando (y con razón...):

¿¿¿Entonces, sobre qué puñeteros campos existentes hemos estado leyendo y escribiendo hasta ahora, si no hemos declarado ninguno???



Principios de OOP: Encapsulación

C# implementa la encapsulación de campos de una manera especial, con una sintaxis conocida como **declaración de propiedades**

En ella, primero se declara un campo privado, y más adelante una propiedad pública seguida de un bloque dentro del cual se declaran e implementan los métodos get y set que permiten, respectivamente, obtener el valor del campo y modificar el valor del campo.

```
class Person
{
    private string name; // field

    public string Name // property
    {
        get { return name; } // get method
        set { name = value; } // set method
    }
}
```

Principios de OOP: Encapsulación

C# implementa la encapsulación de campos de una manera especial, con una sintaxis conocida como **declaración de propiedades**

El código mostrado es la implementación por defecto, pero dentro del método get() o del método set() se puede escribir cualquier código deseado (siempre que el método get() devuelva un valor del tipo indicado en la definición de la propiedad)

```
class Person
{
    private string name; // field

    public string Name // property
    {
        get { return name; } // get method
        set { name = value; } // set method
    }
}
```

A veces, en la literatura, al método get de una propiedad se le llama getter, y al método set se le llama setter

Principios de OOP: Encapsulación

Para facilitar la escritura de campo privado y propiedad pública asociada con el getter y el setter por defecto, que es lo habitualmente deseado, C# define una sintaxis más corta, llamada propiedad autoimplementada (auto-implemented property, en inglés), que sirve para conseguir lo mismo escribiendo menos. Es la sintaxis que ya conocemos:

```
class Person
{
    private string name; // field

    public string Name    // property
    {
        get { return name; }    // get method
        set { name = value; }    // set method
    }
}
```

=

```
class Person
{
    public string Name    // property
    { get; set; }
}
```

Principios de OOP: Encapsulación

Normalmente nos bastará con la funcionalidad que ofrecen los getters y setters por defecto (básicamente no hacen nada más que dar acceso libre al campo privado), pero si en algún caso hay que controlar alguna condición bajo la cual el acceso al campo privado esté prohibido desde fuera de la clase, entonces tendrá sentido personalizar el getter o el setter correspondiente con el código deseado

```
public class Ejercicio1
{
    public bool UsuarioValido { get; set; }
    private int dni;

    public int Dni
    {
        get {
            if (UsuarioValido) { return dni; }
            else { return 0; }
        }
        set {
            if (UsuarioValido) { dni = value; }
        }
    }
}
```


Principios de OOP: Encapsulación

Podemos querer que una propiedad sea pública (permita acceso al campo desde fuera de la clase) solo para lectura, pero sea posible emplear el setter (modificar el campo) desde cualquier método de la clase. Para ello se añade el modificador `private` solo en el setter:

```
public string Address { get; private set; }
```

Si queremos que una propiedad no permita modificaciones en el campo, más allá de la inicialización en el constructor, entonces se elimina el setter de la declaración de la propiedad:

```
public string Name { get; }
```

Principios de OOP: Encapsulación

Otro uso común para las propiedades es la de servir como **campos calculados**. Son propiedades cuyo getter devuelve como resultado una operación aplicada a otro u otros campos (o propiedades) de la clase. Habitualmente tienen getter, pero no setter:

```
public string Nombre { get; set; }
public string Apellidos { get; set; }
public string NombreYApellidos
{
    get { return $"{Nombre} {Apellidos}"; }
}

public string alturaStr { get; set; }
public decimal altura {
    get {
        bool isAlturaStrDecimal = decimal.TryParse(alturaStr, out decimal alturaTemp);
        if (isAlturaStrDecimal) { return alturaTemp; }
        else { return -1M; }
    }
}
```

Principios de OOP: Encapsulación

Beneficios:

- Los mencionados sobre los getters y los setters
- Permite que desde fuera de la clase no tenga importancia la implementación interna de la clase: ni los tipos de los campos ni el código de los métodos. Podríamos guardar un número como un string, y los métodos que quieran acceder a ese campo invocarían a un getter que les pediría como parámetro un int o a un setter que les devolvería como resultado un int (aunque internamente se guarde como string)
- De ahora en adelante, aunque sabemos que campos y propiedades son cosas diferentes, vamos a hablar solo de propiedades, que será lo que más habitualmente vayamos a declarar y usar

02

Herencia

Principios de OOP: Herencia

El principio de herencia plantea:

- crear clases que posean todas las propiedades y métodos de otra clase (llamada clase padre), sin necesidad de volver a definirlos en la clase hija
- poder añadir propiedades o métodos nuevos en la clase hija
- Poder ‘sobreescribir’ métodos de la clase padre en la clase hija, de forma que cuando se invoque un método definido en la clase padre y sobrescrito en la clase hija:
 - Si se invoca sobre un objeto de la clase padre, se ejecutará el código definido en el método de la clase padre
 - Si se invoca sobre un objeto de la clase hija, se ejecutará el código definido en el método de la clase hija

Principios de OOP: Herencia

- Concepto de extensión de funcionalidad asociado a los frameworks. Recordad:
 - El framework incluye bibliotecas con clases que se pueden utilizar pero no borrar o modificar. Lo que sí se pueden es extender.
- La extensión se realiza creando una clase nueva heredada de una clase de alguna biblioteca del framework, y sobrescribiendo los métodos deseados.
- La sintaxis para indicar que una clase hija hereda de una clase padre es la siguiente:

```
public class NombreClaseHija : NombreClasePadre  
{ (definición de nuevos, o sobrescritura, de propiedades y métodos) }
```

Principios de OOP: Herencia

- La herencia en OOP motiva la aparición de un nuevo nivel de accesibilidad además de public y private: protected
- Una propiedad con nivel de accesibilidad protected solo es accesible desde métodos de la propia clase o métodos de clases hijas de la clase en la que está definida
- Un método con nivel de accesibilidad protected solo es invocable desde métodos de la propia clase o métodos de clases hijas de la clase en la que está definido

Principios de OOP: Herencia

- Tabla de referencia de los distintos niveles de accesibilidad:

Acceso	Descripción
public	Se puede llamar desde cualquier clase
internal	Se puede llamar desde cualquier clase del proyecto en el que se encuentra declarado. Nivel por defecto
private	Solo se puede llamar desde otro método de la misma clase
protected	Se puede llamar desde otro método de la misma clase o de clases heredadas de ésta

Principios de OOP: Herencia

- Nivel de accesibilidad: ¿desde dónde está permitido llamar al método?
- Se aplican los anteriores niveles de accesibilidad para las clases, propiedades y métodos
- `public class` -> se pueden crear instancias de esa clase desde cualquier clase (la habitual en el curso)
- `class` -> se pueden crear instancias desde cualquier clase definida en el mismo proyecto (internal)
- `public int tamagno { get; set; }` -> accesible a través de un objeto de la clase
- `private int tamagno { get; set; }` -> no accesible a través de un objeto de la clase
- `protected int tamagno { get; set; }` -> accesible a través de un objeto de la clase o una clase heredera
- `public int sumar(int i1, int i2)` -> accesible a través de un objeto de la clase
- `private int sumar(int i1, int i2)` -> no accesible a través de un objeto de la clase
- `protected int sumar(int i1, int i2)` -> accesible a través de un objeto de la clase o una clase heredera

Principios de OOP: Herencia

- **Constructores de clases hijas** pueden invocar al **constructor de su clase padre**
- La sintaxis es la siguiente:

```
public NombreClaseHija(ParamsConstructor) : base(ParamsConstructorClasePadre)
```

- Los castings explícitos e implícitos aumentan: una variable de clase hija se puede cambiar a una variable de clase padre, y viceversa. Esto será útil para el cuarto principio: el polimorfismo

Herencia: modificador new

¿Y si una clase hija tiene un método con la misma signature que un método de su clase padre (o abuelo, etc), pero distinta implementación?


Al explicar el 3º principio, abstracción, se verá lo que es un método abstracto o un método virtual, pero de momento pensemos en un método normal, como los que hemos visto hasta ahora. Si lo implementa el padre de una manera, y el hijo de otra, cuando se cree un objeto de la clase hija, y se invoque a ese método sobre el objeto de clase hija... ¿qué código se ejecutará, el del padre o el del hijo?

- Pues se ejecutará el del hijo, pero al compilador de Visual Studio no le gustará la situación y añadirá un mensaje de aviso (Warning), para hacer saber al desarrollador, por si no se había dado cuenta, que está evitando que se ejecute el código de un método ya existente, sustituyéndolo por otro código nuevo.

Herencia: modificador new

```
public class ClasePadre
{
    public string Method1(int param1)
    { return param1.ToString(); }
}

public class ClaseHija : ClasePadre
{
    public string Method1(int param1)
    { return param1.ToString().ToLower(); }
}
```

▷  CS0108 'ClaseHija.Method1(int)' oculta el miembro heredado 'ClasePadre.Method1(int)'. Use la palabra clave new si su intención era ocultarlo.

Herencia: modificador new

Existe la posibilidad de que el desarrollador cree una clase que hereda de otra cuya especificación no se ha revisado, y en su ignorancia trate de reinventar la rueda, creando un método con una signatura que ya existe en la clase padre (o abuelo, etc). Para prevenir esta situación es por lo que Visual Studio genera el mensaje de warning anterior.

Si el desarrollador no se ha equivocado, si es consciente de que efectivamente quiere sustituir el código heredado por uno nuevo para objetos de esa clase, es libre de hacerlo, pero el mensaje anterior le pide que deje claro en el código que sabe lo que está haciendo, añadiendo para ello el modificador **new** antes de la signatura del método, si quiere evitar que siga saliendo ese mensaje de warning:

```
public new string Method1(int param1)
{ return param1.ToString().ToLower(); }
```

Principios de OOP: Herencia

Beneficios:

- Reutilización de propiedades y métodos de la clase padre (y de propiedades y métodos de la clase padre de esa clase padre, si la tuviera... Y así hasta el último nivel de parentesco)
- Posibilidad de extender la funcionalidad de una clase ya existente
- La herencia da sentido a una de las formas que toma el tercer principio: el de abstracción, que unida a la otra forma, la interfaz, y haciendo uso del casting implícito entre los tipos clase padre y clase hijo, o interfaz y clase que implementa la interfaz, dan lugar al cuarto principio: el polimorfismo

Ejercicio práctico

Programa 16: herencia entre clase Persona y clase Trabajador

- Crear una clase principal llamada Persona con los atributos: DNI, Nombre, Apellidos.
- Crear una segunda clase llamada Trabajador con los atributos Salario y fecha de contratación, QUE HEREDE de la anterior.
- La aplicación mostrará 3 opciones de menú: 1- Introducción de datos. 2- Mostrar datos introducidos, 3- Salir. Al finalizar opc 1 u opc 2 nos preguntará si queremos seleccionar otra opción.
- OPC -1. Nos pedirá por pantalla los datos de DNI, Nombre, Apellidos, Salario y fecha de contratación y los almacenaremos en una lista.
- OPC -2 . Mostraremos los datos introducidos en la opción 1.

03

Abstracción

Principios de OOP: Abstracción general

El principio de abstracción general en el mundo de la programación plantea:

- Identificar las variables y operaciones mínimas involucradas en la resolución de un problema informático planteado a través de un enunciado
 - variables de entrada: definir rango de valores permitidos y una referencia única (nombre)
 - variables de salida/resultados a obtener: definir rango de valores permitidos y ref. única (nombre)
 - operaciones a realizar: definir su signatura: rango par. entrada y par. salida, y ref. única (nombre)
- Definir una secuencia de pasos que incluya a todos los elementos anteriores, y que permita resolver satisfactoriamente el problema planteado en el enunciado. Esta secuencia de pasos normalmente se denomina estrategia de resolución del problema

Principios de OOP: Abstracción general

- Dicho de otro modo:
 - identificar operaciones genéricas, globales, a realizar, pero sin llegar a pensar en los detalles de su ejecución
 - identificar variables, junto con sus rangos de valores válidos, pero sin llegar a pensar en el tipo de dato que se les asociará
- Es decir, centrarse en plantear un diseño formal (algebraico*) y olvidarse de detalles de implementación
 - *Álgebra: Estudio de los símbolos matemáticos y sus reglas de manipulación

Principios de OOP: Abstracción. Ejemplo

Enunciado del problema: **programa que calcule la suma de dos números**

Análisis del enunciado para abstraer los elementos involucrados y generar una estrategia de resolución del problema:

- Datos de entrada: dos números \rightarrow num1, num2
No se dice nada sobre rangos en el enunciado, así que para esta fase de análisis todo número (natural, entero, fraccionario, real o imaginario) será válido como valor asociado, tanto a num1 como a num2.
- Datos de salida: resultado de la suma \rightarrow res
- Operaciones (escritas con algún tipo de notación algebraica):
 - Obtención del valor de num1 \rightarrow num1 = ObtenerValor1()
 - Obtención del valor de num2 \rightarrow num2 = ObtenerValor2()
 - suma de num1 y num2 para obtener valor de res \rightarrow res = suma(num1, num2)

Principios de OOP: Abstracción. Ejemplo

Enunciado del problema: **programa que calcule la suma de dos números**

Planteamiento de estrategias de resolución del problema (secuencia de pasos a seguir):

```
num1 = ObtenerValor1()
```

```
num2 = ObtenerValor2()
```

```
res = suma(num1, num2)
```

```
num2 = ObtenerValor2()
```

```
num1 = ObtenerValor1()
```

```
res = suma(num1, num2)
```

Incluso, al plantear las operaciones, se podía haber elegido:

Obtención de los valores de num1 y num2 → (num1, num2) = ObtenerValores()

... y entonces otra estrategia de resolución podría haber sido:

```
(num1, num2) = ObtenerValores()
```

```
res = suma(num1, num2)
```

Principios de OOP: Abstracción. Ejemplo

Enunciado del problema: **programa que calcule la suma de dos números**

Elección de una estrategia de resolución del problema (secuencia de pasos a seguir):

```
num1 = ObtenerValor1()  
num2 = ObtenerValor2()  
res = suma(num1, num2)
```

```
num2 = ObtenerValor2()  
num1 = ObtenerValor1()  
res = suma(num1, num2)
```

Incluso, al plantear las operaciones, se podía haber elegido:

Obtención de los valores de num1 y num2 → (num1, num2) = ObtenerValores()

... y entonces otra estrategia de resolución podría haber sido:

```
(num1, num2) = ObtenerValores()  
res = suma(num1, num2)
```

Principios de OOP: Abstracción. Ejemplo

Enunciado del problema: **programa que calcule la suma de dos números**

Con esto hemos diseñado una receta que resulta válida para sumar dos números, CUALESQUIERA que sean, usando CUALQUIER lenguaje de programación

→ 2 niveles de abstracción: del lenguaje de programación, y de los tipos de las variables

El siguiente paso consiste en bajar a la Tierra, eligiendo en qué lenguaje de programación se va a escribir el programa. El lenguaje elegido definirá:

- la sintaxis a utilizar para escribir las operaciones a realizar (en C# serán métodos de una clase, u operadores incluidos en una instrucción dentro de un método de una clase)
- Los tipos disponibles (o, si el lenguaje lo permite, la forma de crear tipos nuevos) para asociar a las variables que aparecen en la estrategia planteada

Principios de OOP: Abstracción. Ejemplo

Enunciado del problema: **programa que calcule la suma de dos números**

En un programa de **C#** que sume números enteros, se puede usar el tipo **int**

```
num1 = ObtenerValor1()  
num2 = ObtenerValor2()  
res = suma(num1, num2)
```



```
public int CalcularSuma()  
{  
    int num1 = ObtenerValor1();  
    int num2 = ObtenerValor2();  
    int res = suma(num1, num2);  
    return res;  
}  
public int ObtenerValor1()...  
public int ObtenerValor2()...  
public int suma(int n1, int n2)...
```

Principios de OOP: Abstracción. Ejemplo

Enunciado del problema: **programa que calcule la suma de dos números**

En un programa de **C#** que sume números fraccionarios, se puede usar el tipo **decimal**

```
num1 = ObtenerValor1()  
num2 = ObtenerValor2()  
res = suma(num1, num2)
```



```
public decimal CalcularSuma()  
{  
    decimal num1 = ObtenerValor1();  
    decimal num2 = ObtenerValor2();  
    decimal res = suma(num1, num2);  
    return res;  
}  
public decimal ObtenerValor1()...  
public decimal ObtenerValor2()...  
public decimal suma(decimal n1, decimal n2)...
```


Principios de OOP: Abstracción. Ejemplo

Enunciado del problema: programa que calcule la suma de dos números

Si queremos sumar números imaginarios, se puede crear una clase **NumImag**

```
public class NumImag
{
    public decimal ParteReal { get; set; }
    public decimal ParteImaginaria { get; set; }
}
```

```
num1 = ObtenerValor1()
num2 = ObtenerValor2()
res = suma(num1, num2)
```



```
public NumImag CalcularSuma()
{
    NumImag num1 = ObtenerValor1();
    NumImag num2 = ObtenerValor2();
    NumImag res = suma(num1, num2);
    return res;
}

public NumImag ObtenerValor1()...
public NumImag ObtenerValor2()...
public NumImag suma(NumImag n1, NumImag n2)...
```

Principios de OOP: Abstracción. Ejemplo

Como vemos, la abstracción nos aporta un esqueleto válido para escribir muchos programas diferentes, dejando como tarea posterior definir los tipos de datos elegidos, y también el código elegido para implementar las operaciones planteadas

Así pues, siguiendo el ejemplo solo con el esqueleto del programa para sumar números enteros, aún tenemos la obligación de elegir (libremente) cómo queremos implementar la obtención de los dos números y su suma

```
public int ObtenerValor1()
{ return 1; }

public int ObtenerValor1()
{
    Console.WriteLine("Escriba un número: ");
    return int.TryParse(Console.ReadLine(), out int n1) ? n1 : 0;
}
```

```
public int suma(int n1, int n2)
{ return n1+n2; }

public int suma(int n1, int n2)
{ return n1-n2*-1; }

public int suma(int n1, int n2)
{
    for(int i = 0; i <= n2; i++) { n1++; }
    return n1;
}
```

Principios de OOP: Abstracción. Ejemplo

Finalmente, una vez definidos el lenguaje de programación, los tipos de las variables, las signatures de los métodos y la implementación de los métodos, ya tenemos construido nuestro programa, y si queremos hacer cambios, la estructura modular planteada permite hacerlos sin que el método principal (CalcularSuma) se entere

```
public int ObtenerValor1()
{
    Console.Write("Escriba un número: ");
    return int.TryParse(Console.ReadLine(), out int n1) ? n1 : 0;
}
public int ObtenerValor2()
{
    Console.Write("Escriba otro número: ");
    return int.TryParse(Console.ReadLine(), out int n2) ? n2 : 0;
}
public int suma(int n1, int n2)
{ return n1+n2; }
```

```
public int CalcularSuma()
{
    int num1 = ObtenerValor1();
    int num2 = ObtenerValor2();
    int res = suma(num1, num2);
    return res;
}
```

Principios de OOP: Abstracción general

Entonces... hemos visto que puede haber distintos programas que comparten unas mismas operaciones, y además ejecutadas en un mismo orden, es decir, distintos programas que comparten un mismo esqueleto de ejecución.

¿Podríamos guardarnos ese esqueleto (y solo el esqueleto, sin implementaciones) en algún sitio para poder reutilizarlo?

¿o al menos podríamos guardarnos la lista de operaciones comunes (también sin implementaciones)?

En C#, sí a lo segundo... y más o menos sí a lo primero. Lo veremos tras la explicación de la abstracción enfocada desde la programación orientada a objetos

Principios de OOP: Abstracción

El principio de abstracción enfocado a la programación orientada a objetos plantea:

- Localizar y definir funcionalidades que van a ser comunes a objetos de distintas clases, y definir para ellas una misma signature de los métodos que ejecuten dichas funcionalidades comunes, sin llegar a implementar el código que las ejecute
- Existen dos variantes para la abstracción de métodos en C#: por **clase abstracta** y por **interfaz**
- La primera obliga a que exista una herencia entre una clase padre declarada como abstracta, que declarará la signature del método sin implementarlo, y todas las clases hijas que compartirán una misma funcionalidad invocable mediante la signature definida en la clase padre, cada hija ejecutando un código propio para ella
- La segunda es más versátil, ya que no depende de ninguna herencia entre clases

Clases abstractas: modificador abstract

Cuando en una clase se declara al menos un método abstracto, TODA la clase 'se convierte' en abstracta (hay que declararla como 'public **abstract** class {NombreClase}').

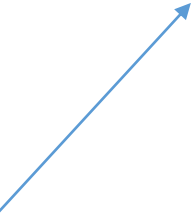
Además, para cada método abstracto declarado en una clase padre, todas sus clases hijas están OBLIGADAS a dar una implementación del mismo (o pasar la pelota a alguna clase nieta... Veremos cómo...), declarando un método con la misma signatura precedido del modificador **override**

Así, crear una clase abstracta es una forma de obligar a que todas las clases que se desee que hereden de ella tengan un método con el mismo nombre y los mismos tipos de parámetros de entrada, y que además devuelvan un resultado del mismo tipo

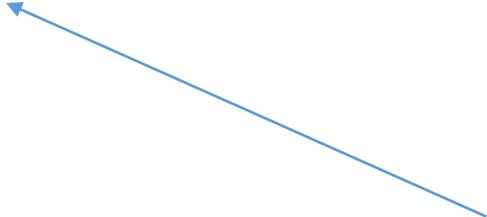
Clases abstractas: modificador abstract

```
public abstract class ClasePadre {  
    public abstract bool MostrarInfo(); // solo la signatura, sin implementación  
}
```

```
public class ClaseHija1: ClasePadre {  
    public override bool MostrarInfo() {  
        // implementación versión 1  
    }  
}
```



```
public class ClaseHija2: ClasePadre {  
    public override bool MostrarInfo() {  
        // implementación versión 2  
    }  
}
```



Clases abstractas: modificador abstract

Usar esas clases abstractas es como 'asegurar' un contrato que cumplirán todas las clases hijas.

Ese contrato lo podemos utilizar para que no nos importe cómo está implementada cada clase hija. Simplemente:

- invocamos al método abstracto del padre
- le pasamos los parámetros del tipo que tenga definidos el método abstracto
- recibimos el resultado en una variable del tipo que devuelve el método abstracto.

Clases abstractas: modificador abstract

Esta abstracción (este 'nos da igual el código que ejecuta internamente el método, es más, nos da igual hasta cuál de las clases hijas lo va a ejecutar') aplicada en nuestros programas aporta varias ventajas:

- Mayor reutilización del código (tener cosas hechas, aunque sean solo declaraciones de métodos, sin el código)
- Mejor mantenimiento del código (facilidad para cambiar cosas)
- Mejor escalabilidad del código (facilidad para añadir cosas)
- Facilidad para realizar pruebas unitarias (lo veremos)
- Permite Inversión de Dependencia (también lo veremos)

Clases abstractas: modificador virtual

¿Y si queremos que solo alguna de las clases hijas implemente un código distinto, porque las demás van a implementar un código idéntico? ¿Es necesario repetir ese código igual en varias clases hijas, perdiendo el principio de reutilización? NO

Para ese caso, la clase padre declara un método con una implementación por defecto, y su signatura irá precedida por el modificador **virtual**

- Cada clase hija que quiera utilizar una implementación distinta a la de por defecto, definirá un método con la misma signatura que sobre-escribirá el método de la clase padre
- Cada clase hija que quiera utilizar la implementación por defecto, simplemente no definirá el método. El compilador verá que no hay método definido en esa clase, y buscará (y encontrará) su implementación en la clase padre (o abuelo, o más allá...)

Clases abstractas: modificador virtual

¿Y si queremos que solo alguna de las clases hijas implemente un código distinto, porque las demás van a implementar un código idéntico? ¿Es necesario repetir ese código igual en varias clases hijas, perdiendo el principio de reutilización? NO

```
public class ClasePadre {  
    public virtual bool MostrarInfo() {  
        // implementación por defecto  
    }  
}
```

```
public class ClaseHija1: ClasePadre {  
    public override bool MostrarInfo() {  
        // reimplementación, versión 1  
    }  
}
```

```
public class ClaseHija2: ClasePadre {  
    // la clase no reimplementa MostrarInfo(),  
    // así que usa la versión de ClasePadre  
}
```

Clases abstractas: modificador virtual

- Recordemos que una clase se declara como abstracta solo si alguno de sus métodos no tiene implementación. En el caso del modificador virtual, el método sí tiene implementación, por lo que no es necesario que una clase sea abstracta para poder declarar en ella métodos virtuales
- Las propiedades auto-implementadas, al final son métodos (getter y setter), por lo que también se pueden declarar como virtuales si queremos que el getter o el setter tengan una implementación diferente en alguna clase hija:

```
public class MyBaseClass
{
    public virtual string Name { get; set; }
}
```

```
public class MyDerivedClass : MyBaseClass
{
    private string name;
    public override string Name
    {
        get { return name; }
        set { if (!string.IsNullOrEmpty(value)) { name = value; }
              else { name = "Unknown"; } }
    }
}
```


Clases abstractas: override abstract

Si queremos que una clase hija de una clase abstracta NO se vea obligada a implementar un método abstracto, sino que deje que sean las clases nietas quienes decidan si lo implementan o le pasan la pelota a las bisnietas, etc, existen dos formas de conseguirlo, las dos haciendo que la clase hija también sea abstracta:

- declarar el método que se supone que está obligada a implementar, precedido de **override abstract** (override para que el compilador no se queje de que faltan métodos por sobrescribir, y abstract para que sea posible que no haya implementación de ese método en esa clase). Es raro, pero funciona, y deja constancia visible en el código de que existen métodos abstractos en la clase padre que la clase hija deja a otras clases herederas la responsabilidad de implementar
- directamente no declarar ese método en esa clase. Es más cómodo, pero en cierto modo oculta información que, en caso de necesitar, hay que ir a buscarla a la clase padre (o abuelo, etc...)

Clases abstractas: override abstract

Con declaración override abstract:

```
public abstract class ClasePadre
{ public abstract string Method1(int param1); }

public abstract class ClaseHija : ClasePadre
{ public override abstract string Method1(int param1); }

public class ClaseNieta : ClaseHija
{
    public override string Method1(int param1)
    { return param1.ToString(); }
}
```

Sin declaración:

```
public abstract class ClasePadre
{ public abstract string Method1(int param1); }

public abstract class ClaseHija : ClasePadre { }

public class ClaseNieta : ClaseHija
{
    public override string Method1(int param1)
    { return param1.ToString(); }
}
```

Principios de OOP: Clases abstractas

Con las clases abstractas ya estamos en disposición de responder a las preguntas que hemos dejado pendientes al acabar la explicación de la abstracción general:

¿Podríamos guardarnos una lista de operaciones comunes, sin implementaciones, para poder reutilizarlas en distintos programas?

- SI, creando métodos abstractos

¿Podríamos guardarnos un esqueleto de ejecución, sin implementaciones, en algún sitio para poder reutilizarlo?

- SI, creando un método no abstracto que invoque a los métodos abstractos correspondientes en el orden indicado por el esqueleto de ejecución

Principios de OOP: Clases abstractas

```
public abstract class CalculosConEnteros
{
    public int CalcularSuma()
    {
        int num1 = ObtenerValor1();
        int num2 = ObtenerValor2();
        int res = suma(num1, num2);
        return res;
    }

    public abstract int ObtenerValor1();
    public abstract int ObtenerValor2();
    public abstract int suma(int n1, int n2);
}
```

Principios de OOP: Clases abstractas

El principal problema de las clases abstractas es que tiene que emplearse la herencia: una clase hija debe heredar de (tener como clase padre) una clase abstracta

En C#, una clase SOLO puede tener una clase padre. En otros lenguajes se permiten varios padres (lo que se conoce como herencia múltiple, y genera ciertos problemas), pero en C# no.

¿y si queremos una clase que implemente los métodos abstractos definidos en dos o mas clases abstractas? → NO PODEMOS. Para eso se crearon las INTERFACES

Principios de OOP: Interfaces

Una **interfaz** se comporta EXACTAMENTE IGUAL que una clase abstracta que solo declara métodos abstractos (recuerdo que las propiedades son métodos get y set)

Y tiene la ventaja de que una clase puede hacer uso de tantas interfaces distintas como desee (aquí no se llama 'heredar de la interfaz', sino 'implementar la interfaz')

La sintaxis para que una clase implemente una interfaz es la misma que la usada cuando una clase hereda de una clase padre, pero añadiendo que, si implementa varias interfaces, van separadas por comas:

```
public class Clase1: Interfaz1, Interfaz2, Interfaz3 ...
```


Principios de OOP: Interfaces

Ejemplo de implementación de varias interfaces por parte de la clase String:

```
...public interface IComparable
{
    ...int CompareTo(object obj);
}

...public interface IEnumerable
{
    ...IEnumerator GetEnumerator();
}

...public interface ICloneable
{
    ...object Clone();
}
```

```
...public interface IConvertible
{
    ...TypeCode GetTypeCode();
    ...bool ToBoolean(IFORMATPROVIDER provider);
    ...byte ToByte(IFORMATPROVIDER provider);
    ...char ToChar(IFORMATPROVIDER provider);
    ...DateTime ToDateTime(IFORMATPROVIDER provider);
    ...decimal ToDecimal(IFORMATPROVIDER provider);
    ...double ToDouble(IFORMATPROVIDER provider);
    ...short ToInt16(IFORMATPROVIDER provider);
    ...int ToInt32(IFORMATPROVIDER provider);
    ...long ToInt64(IFORMATPROVIDER provider);
    ...sbyte ToSByte(IFORMATPROVIDER provider);
    ...float ToSingle(IFORMATPROVIDER provider);
    ...string ToString(IFORMATPROVIDER provider);
    ...object ToType(Type conversionType, IFORMATPROVIDER provider);
    ...ushort ToUInt16(IFORMATPROVIDER provider);
    ...uint ToUInt32(IFORMATPROVIDER provider);
    ...ulong ToUInt64(IFORMATPROVIDER provider);
}
```

```
public sealed class String : IComparable, ICloneable, IConvertible, IEnumerable
```

Principios de OOP: Interfaces

- Las propiedades declaradas en una interfaz no se convierten en propiedades auto-implementadas, como ocurre en las clases, porque por definición no pueden tener implementación. Eso quiere decir que las clases que implementen esa interfaz deberán declarar las propiedades igualmente, ya sean auto-implementadas con los get y set por defecto, o indicando un código personalizado para uno u otro

```
interface IEmployee
{
    string Name { get; set; }
    int Counter { get; }
}

public class Employee : IEmployee
{
    public string Name { get; set; }
    public int Counter { get; }
}
```

Principios de OOP: Interfaces

Uno de los problemas de la herencia múltiple (no el único) es la llamada colisión de nombres, que se refiere al hecho de que distintas clases padre pudieran tener un método con la misma signatura. Pues bien, las interfaces también tienen ese problema. ¿Qué hacer si hay dos interfaces que obligan a implementar un mismo método con una misma signatura?

En principio no parece haber ningún problema. Simplemente, si la clase lo implementa para una interfaz, también lo implementa para la otra, y en paz:

```
interface Interfaz1 { void Metodo1(); }

interface Interfaz2 { void Metodo1(); }

public class Clase : Interfaz1, Interfaz2
{
    public void Metodo1()
    {
        Console.WriteLine("hola");
    }
}
```

Principios de OOP: Interfaces

Pero ¿qué pasa si yo, como desarrollador, quiero que el código que implemente ese método sea distinto si invoco al método desde un objeto de la clase convertido mediante casting explícito a una de las interfaces, que si lo invoco desde el mismo objeto convertido a la otra interfaz?

```
interface Interfaz1 { void Metodo1(); }
```

```
interface Interfaz2 { void Metodo1(); }
```

```
public class Clase : Interfaz1, Interfaz2
{
    public void Metodo1()
    {
        Console.WriteLine("hola");
    }
}
```

```
public class Trabajo
```

```
{
```

```
    public void Ejecutar()
```

```
{
```

```
        ((Interfaz1) new Clase()).Metodo1(); // quiero escribir hola
```

```
        ((Interfaz2) new Clase()).Metodo1(); // quiero escribir chao
```

```
}
```

```
}
```

Principios de OOP: Interfaces

¡Pues se puede!

Simplemente, en la signature del método, su nombre será la interfaz deseada, seguida de '.', seguida del nombre del método:

```
public interface Interfaz1 { void Metodo1(); }

interface Interfaz2 { void Metodo1(); }

public class Clase : Interfaz1, Interfaz2
{
    void Interfaz1.Metodo1()
    { Console.WriteLine("hola"); }
    void Interfaz2.Metodo1()
    { Console.WriteLine("chao"); }
    public void Metodo1()
    { Console.WriteLine("hello"); }
}
```

```
public class Trabajo
{
    public void Ejecutar()
    {
        ((Interfaz1) new Clase()).Metodo1(); // escribe hola
        ((Interfaz2) new Clase()).Metodo1(); // escribe chao
        new Clase().Metodo1();                // escribe hello
    }
}
```

Principios de OOP: Interfaces

Con las interfaces ya estamos en disposición de responder a las preguntas que hemos dejado pendientes al acabar la explicación de la abstracción general:

¿Podríamos guardarnos una lista de operaciones comunes, sin implementaciones, para poder reutilizarlas en distintos programas?

- SI, creando métodos sin implementar en una interfaz

¿Podríamos guardarnos un esqueleto de ejecución, sin implementaciones, en algún sitio para poder reutilizarlo?

- NO, porque las interfaces no permiten implementación de los métodos que definen*

*Al menos, .NET Framework no lo permite. Con C# 8.0, desarrollado para .NET Core, sí se puede... 😞

04

Polimorfismo

Principios de OOP: Polimorfismo

El principio de polimorfismo plantea:

- Poder invocar a un mismo método o usar un mismo operador para que ejecute códigos diferentes en invocaciones/usos diferentes
- Es el concepto de 'reutilizar' nombres de métodos para que se comporten de 'varias formas' (significado LITERAL del término polimorfismo) distintas
- Existen dos tipos de polimorfismo: polimorfismo estático y polimorfismo dinámico

Principios de OOP: Polimorfismo

polimorfismo estático: la ejecución de un código u otro de los posibles se conoce antes de la ejecución, porque depende del nº y tipo de los parámetros del método

- Polimorfismo estático por sobrecarga de métodos: cuando existen varios métodos con el mismo nombre, pero el nº de parámetros o los tipos de los parámetros son diferentes.
 - Ya hemos visto ejemplos de ambos casos:
 - Constructor vacío / Constructor con parámetros
 - `Console.Write(string valor)` / `Console.Write (int valor)` / `Console.Write(decimal valor)`

Principios de OOP: Polimorfismo

polimorfismo estático: la ejecución de un código u otro de los posibles se conoce antes de la ejecución, porque depende del nº y tipo de los parámetros del método

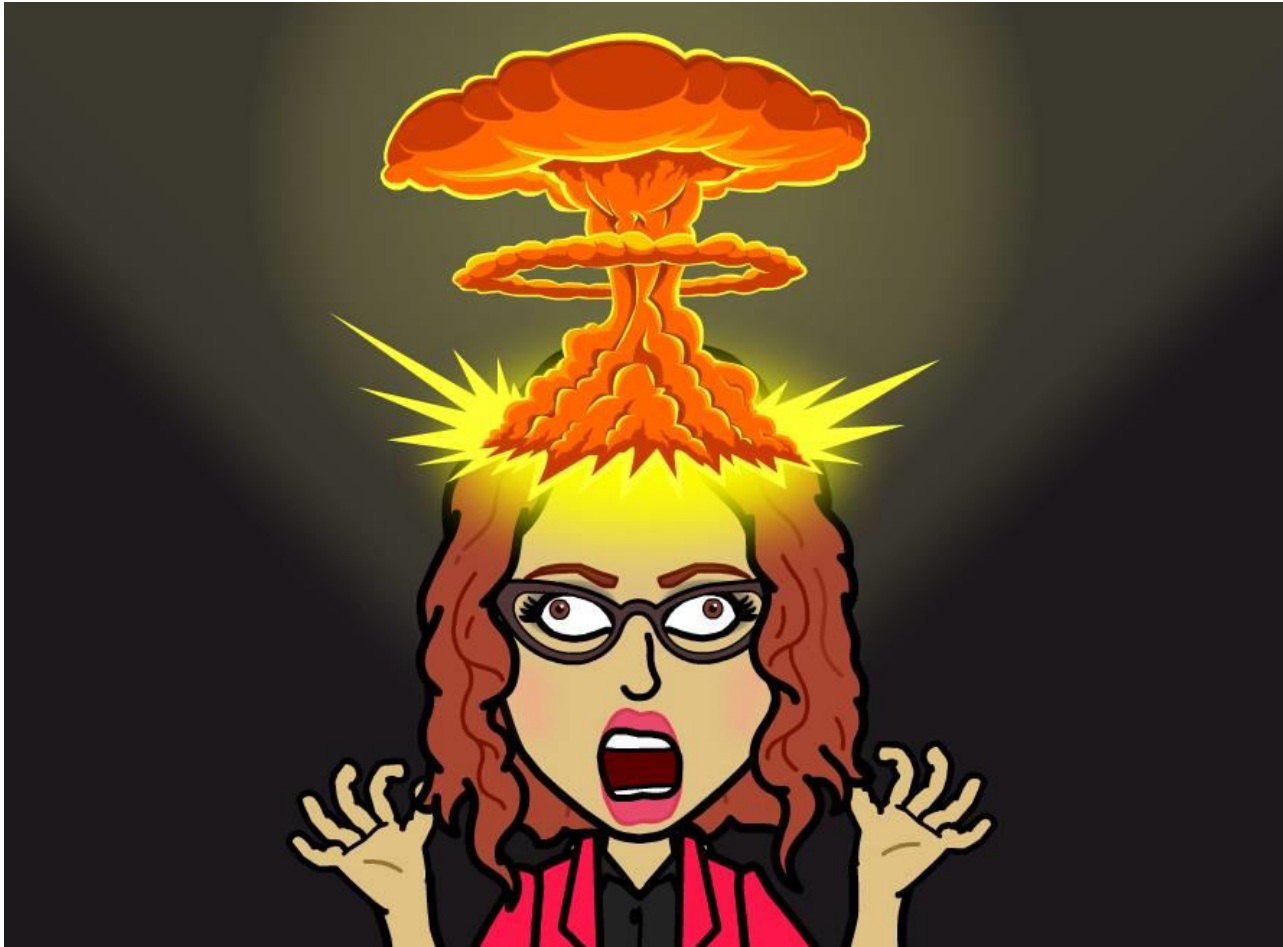
- Polimorfismo estático por sobrecarga de operadores: cuando existen varios operadores con el mismo símbolo, pero según los tipos de datos que tienen a derecha e izquierda, hacen cosas diferentes.
 - También hemos visto ejemplos sin mencionarlo:
 - operador '+': para int → suma entera; para decimal → suma decimal; para string → ¡concatenación!

Principios de OOP: Polimorfismo

polimorfismo dinámico: la ejecución de un código u otro de los posibles no se conoce hasta la ejecución, porque para el compilador todos los objetos sobre los que se va a aplicar el método son de la misma clase (una clase padre), pero durante la ejecución no se va a ejecutar el código del método definido en el padre, sino la sobreescritura que haya implementado la clase hija correspondiente

- Crear clase padre con un método abstracto o virtual, y crear varias clases hijas con el método del padre implementado en cada una con código diferente. Luego, si al recorrer un listado de objetos de la clase padre, donde se hayan añadido objetos de las diferentes clases hijas, se ejecuta ese método, el código que se ejecute corresponderá al de la clase hija correspondiente

Principios de OOP: Polimorfismo



Principios de OOP: Polimorfismo

Pasito a pasito, sin ponernos nerviosos...

- Crear clase padre con un método abstracto o virtual...

```
namespace PrimerProyectoDeConsola.Personas
{
    8 referencias
    abstract class Persona
    {
        9 referencias
        public string nombre { get; set; }
        9 referencias
        public string apellidos { get; set; }
        9 referencias
        public int edad { get; set; }
        1 referencia
        public string DarInformacionNormal()
        {
            return $"Soy una persona NORMAL (guiño, guiño), me llamo {nombre} {apellidos} y tengo {edad} años";
        }

        4 referencias
        public abstract string DarInformacionAbs();

        2 referencias
        public virtual string DarInformacionVir() {
            return $"Soy una persona, me llamo {nombre} {apellidos} y tengo {edad} años";
        }
    }
}
```

Principios de OOP: Polimorfismo

Pasito a pasito, sin ponernos nerviosos...

- ...crear varias clases hijas con el método del padre implementado en cada una con código diferente...

```
namespace PrimerProyectoDeConsola.Personas
{
    2referencias
    class PersonaMenorDeEdad : Persona
    {
        2referencias
        public string juegoFavorito { get; set; }

        4referencias
        public override string DarInformacionAbs()
        {
            return $"Aún estoy aprendiendo a vivir, me llamo {nombre} {apellidos}, tengo {edad} años y disfruto mucho jugando al {juegoFavorito} con mis amigos";
        }
    }
}
```

Principios de OOP: Polimorfismo

Pasito a pasito, sin ponernos nerviosos...

- ...crear varias clases hijas con el método del padre implementado en cada una con código diferente...

```
namespace PrimerProyectoDeConsola.Personas
{
    2 referencias
    class PersonaAdulta : Persona
    {
        3 referencias
        public string marcaCoche { get; set; }
        3 referencias
        public string modeloCoche { get; set; }

        4 referencias
        public override string DarInformacionAbs()
        {
            return $"Soy adulto, me llamo {nombre} {apellidos}, tengo {edad} años y conduzco un {marcaCoche} {modeloCoche}";
        }
        2 referencias
        public override string DarInformacionVir() {
            return $"Soy adulto, me llamo {nombre} {apellidos}, tengo {edad} años y conduzco un {marcaCoche} {modeloCoche}";
        }
    }
}
```

Principios de OOP: Polimorfismo

Pasito a pasito, sin ponernos nerviosos...

- ...crear varias clases hijas con el método del padre implementado en cada una con código diferente...

```
namespace PrimerProyectoDeConsola.Personas
{
    2referencias
    class PersonaJubilada: Persona
    {
        2referencias
        public string hobby { get; set; }

        4referencias
        public override string DarInformacionAbs()
        {
            return $"Estoy jubilado, me llamo {nombre} {apellidos}, tengo {edad} años y en mis ratos libres hago {hobby}";
        }
    }
}
```

Principios de OOP: Polimorfismo

Pasito a pasito, sin ponernos nerviosos...

- ... crear un listado de objetos de la clase padre, y añadir a él objetos de las diferentes clases hijas...

```
static void Main(string[] args)
{
    List<Persona> personas = new List<Persona>();

    PersonaMenorDeEdad primeraedad = new PersonaMenorDeEdad()
    {
        nombre= "Juan",
        apellidos= "Martín Pastor",
        edad= 9,
        juegoFavorito= "baloncesto",
    };

    PersonaAdulta segundaedad = new PersonaAdulta()
    {
        nombre = "Manuel",
        apellidos = "Huerta Penedés",
        edad = 37,
        marcaCoche = "BMW",
        modeloCoche = "M3",
    };

    PersonaJubilada terceraedad = new PersonaJubilada()
    {
        nombre = "Federico",
        apellidos = "Grasa Lapuerta",
        edad = 72,
        hobby = "largos paseos por el parque",
    };

    personas.Add(primeriedad);
    personas.Add(segundaedad);
    personas.Add(terceraedad);
}
```

Principios de OOP: Polimorfismo

Pasito a pasito, sin ponernos nerviosos...

- ...recorrer el listado anterior, ejecutando el método abstracto o virtual del padre para cada elemento
- aquí se va a recorrer tres veces:
 - la primera, ejecutando el método normal creado solo en el padre
 - la segunda ejecutando el método abstracto del padre, implementado solo en las clases hijas
 - la tercera ejecutando el método virtual del padre, que solo se ha sobrescrito en la clase hija PersonaAdulta

```
Console.WriteLine();
Console.WriteLine("Información obtenida del método normal:");
foreach (Persona aQuienLeToque in personas)
{
    Console.WriteLine(aQuienLeToque.DarInformacionNormal());
}

Console.WriteLine();
Console.WriteLine("Información obtenida del método abstracto:");
foreach (Persona aQuienLeToque in personas)
{
    Console.WriteLine(aQuienLeToque.DarInformacionAbs());
}

Console.WriteLine();
Console.WriteLine("Información obtenida del método virtual:");
foreach (Persona aQuienLeToque in personas)
{
    Console.WriteLine(aQuienLeToque.DarInformacionVir());
}

//Console.WriteLine("Hello, World!");
Console.ReadKey();
}
```


Principios de OOP: Polimorfismo

¿Qué pasará? (¿Qué misterio habrá?...): la ejecución da el siguiente resultado:

Información obtenida del método normal:

```
Soy una persona NORMAL (guiño, guiño), me llamo Juan Martín Pastor y tengo 9 años  
Soy una persona NORMAL (guiño, guiño), me llamo Manuel Huerta Penedés y tengo 37 años  
Soy una persona NORMAL (guiño, guiño), me llamo Federico Grasa Lapuerta y tengo 72 años
```

Información obtenida del método abstracto:

```
Aún estoy aprendiendo a vivir, me llamo Juan Martín Pastor, tengo 9 años y disfruto mucho jugando al baloncesto con mis amigos  
Soy adulto, me llamo Manuel Huerta Penedés, tengo 37 años y conduzco un BMW M3  
Estoy jubilado, me llamo Federico Grasa Lapuerta, tengo 72 años y en mis ratos libres hago largos paseos por el parque
```

Información obtenida del método virtual:

```
Soy una persona, me llamo Juan Martín Pastor y tengo 9 años  
Soy adulto, me llamo Manuel Huerta Penedés, tengo 37 años y conduzco un BMW M3  
Soy una persona, me llamo Federico Grasa Lapuerta y tengo 72 años
```

Ejercicio práctico

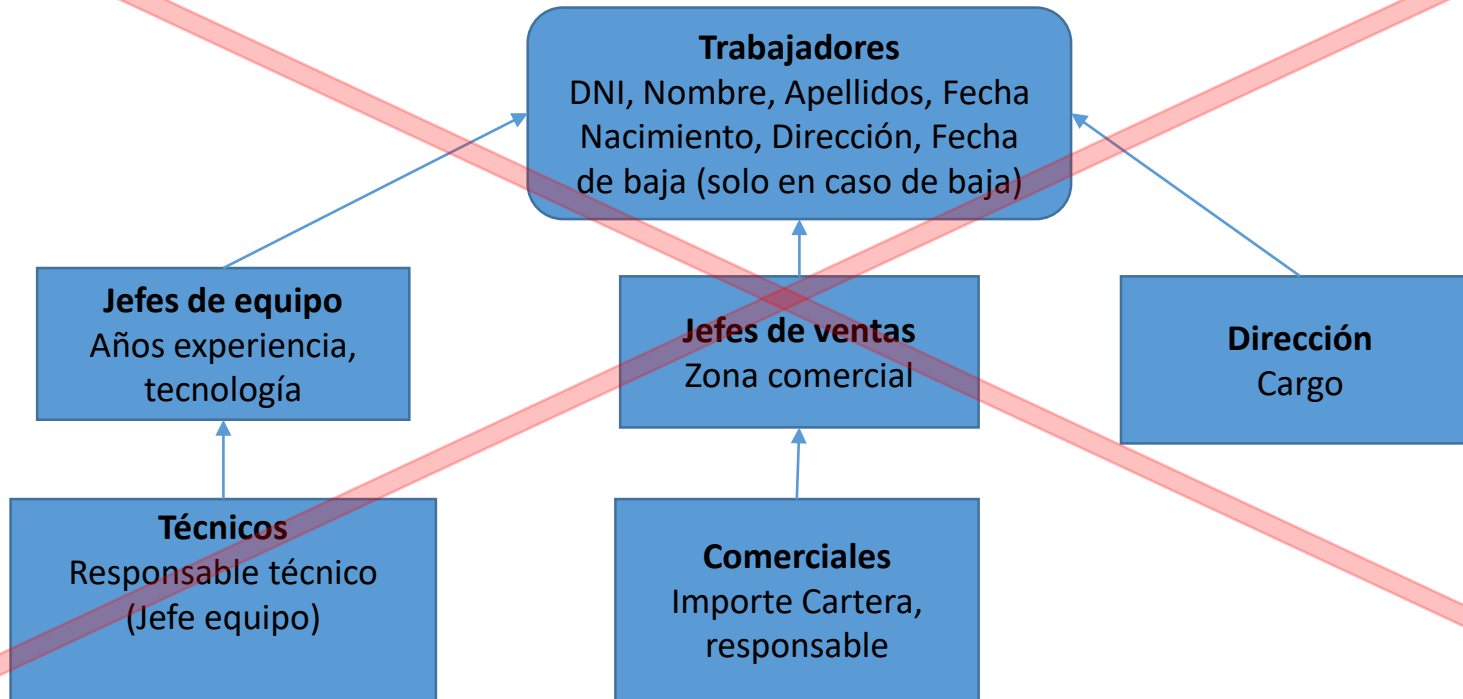
Programa 17: Gestión de trabajadores

Al cargar la aplicación, mostrará un menú con las siguientes opciones:

- 1-Alta de jefes de equipo.
- 2-Alta de técnicos (comprobar que el responsable existe)
- 3-Alta de jefes de ventas.
- 4-Alta de Comerciales (comprobar que el responsable existe).
- 5-Alta de personas de dpto. de dirección.
- 6-Mostrar todos los técnicos.
- 7-Mostrar los técnicos según responsable.
- 8-Mostrar los comerciales según responsable.
- 9-Mostrar jefes de equipo según tecnología.
- 10-Mostrar Dpto. Dirección.
- 11- Baja de personal
- 12- Salir

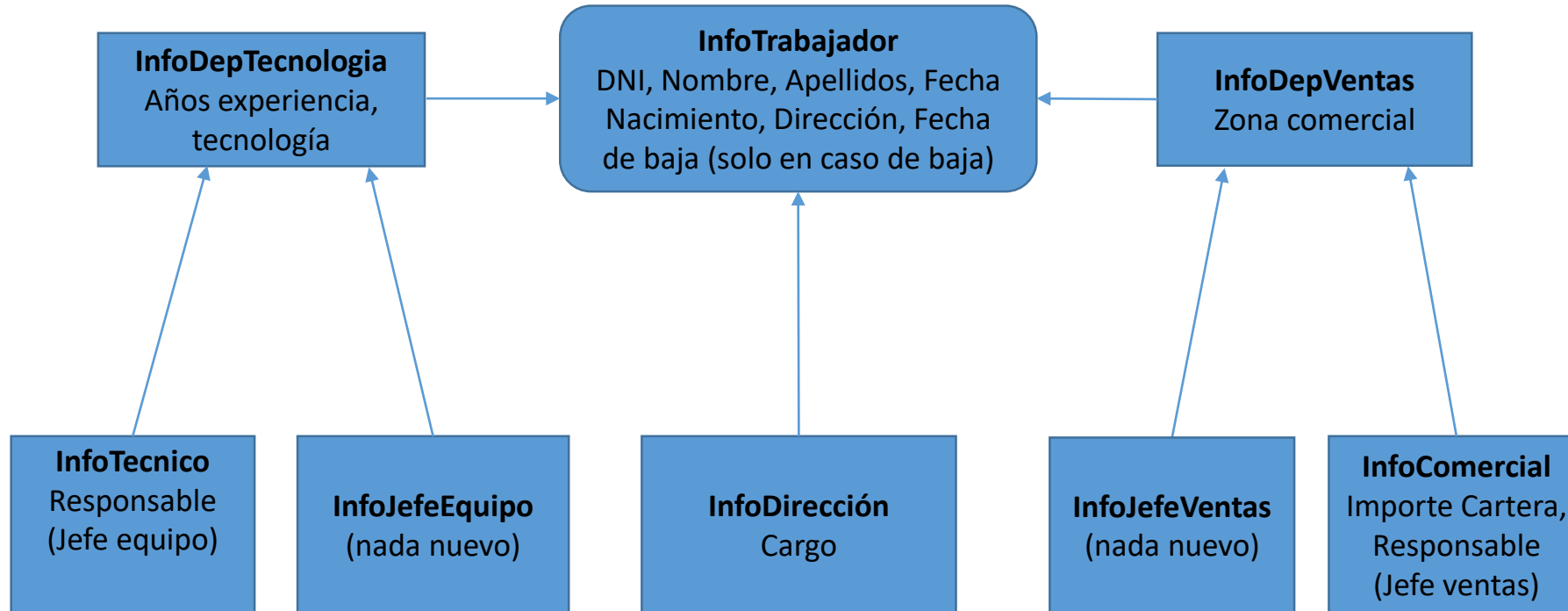
Ejercicio práctico

Programa 17: Gestión de trabajadores: estructura de información compartida



Ejercicio práctico

Programa 17: Gestión de trabajadores: diagrama de clases con herencia más adecuada



Ejercicio práctico evaluable

Ejercicio 18. Algo parecido al ejercicio 17:

- Clase base con id autoincrementado
- Al menos dos (mejor 3) niveles de herencia que comiencen desde la clase base
- Una única lista de elementos de la clase base
- CRUD sobre esa lista:
 - añadir elemento, ver, editar, borrar
- Opciones de búsqueda en lista:
 - un elemento
 - todos los elementos
 - alguna búsqueda con filtro

Ejercicio práctico evaluable

Ideas:

- Sala de cine: películas, óperas en directo, vídeos corporativos, ...
- Concesionario: coche, furgoneta, todoterreno, ...
- Hipermercado: artículo alimentación, artículo perfumería, artículo moda...
- Biblioteca, Taller de carpintería, Aeropuerto, Colegio, Hospital, Agencia de viajes...
- Seres de Star Wars, magos de Hogwarts, Pokemons, Personajes de Naruto, GoT, etc...

Estos son ejemplos, pero es mejor que cada uno piense en lo que le guste...

- Se pueden plantear como clases:
 - Los elementos que se crean, se usan o se venden (materiales o servicios, gratis o de pago)
 - Las diferentes personas que se relacionan con esos elementos: trabajadores, usuarios, ...

MUCHAS GRACIAS



Pasión por la innovación

www.integratecnologia.es