Herencia

1. PRESENTACIÓN

La programación orientada a objetos introduce muchos conceptos que no están presentes en los lenguajes de programación tradicionales. Uno de estos conceptos es la **herencia** o jerarquía de clases.

La **herencia** es, sin lugar a dudas, una de las características más potentes de la orientación a objetos y una de las que permiten distinguir claramente entre lo que es y no es un lenguaje orientado a objetos.

He preparado un ejemplo que nos ayudará a entender el concepto de herencia.

Imagina que te han propuesto preparar una aplicación para controlar el pago de las nóminas de los empleados de su empresa.

Has analizado el problema y has decidido escribir una clase **Empleado** donde se reflejen las características (propiedades y métodos) que estas personas (objetos) tienen.

Archivo **empleado.inc.php**:

```
class Empleado
{
	private $nombre;
	private $salario;
	private $fechaAlta;
	private $fechaBaja;

	public function __construct($nombre, $salario,
				$fecha)
	{
		$this->nombre = $nombre;
		$this->salario = $salario;
		$this->fechaAlta = $fecha;
	}

	public function baja($fecha)
	{
		$this->fechaBaja = $fecha;
		$this->salario = 0;
	}
	public function calcularSalario()
	{
		return $this->salario;
	}
}
```

Has decidido incluir las propiedades **nombre**, **salario**, **fecha alta** y **fecha baja** y, siguiendo las indicaciones de la lección anterior, las has declarado como **privadas** para la clase.

También has escrito un método **constructor** en el que se inicializan las propiedades necesarias al crear un objeto **Empleado**.

Finalmente, has decidido escribir dos **métodos**: uno que permita indicar que el empleado ha causado **baja** en la empresa y otro en el que se **calcula el salario** que hay que pagarle. Todos estos métodos los has calificado como **públicos**.

Estás muy contento con la clase que has escrito, pero ahora llega el jefe de contabilidad y te indica que tu aplicación no le sirve para los **vendedores** que tiene la empresa.

¿Por qué? Porque no hay forma de reflejar las ventas que han realizado esos vendedores y se da la casualidad de que su salario depende, en parte, de dichas ventas.

Tu primera reacción es la de crear una nueva clase que represente la existencia de los vendedores en su empresa.

Empiezas a pensar y decides que estos vendedores tendrán su nombre, su salario fijo, su fecha de alta, su fecha de baja...

Además, podrán darse de baja, deberá calcular su salario a la hora de pagar las nóminas...

En definitiva, te das cuenta de que tienen todas las características que ya habías identificado para la clase **Empleado**. Por otra parte, tendrán otros detalles que otros empleados no tienen: el número de ventas realizadas y la comisión que cobran.

¿Existe alguna forma de representar esta situación en la programación orientada a objetos?

Sí, a través de la **herencia**.

2. CREAR SUBCLASES

Desde luego, podríamos escribir de nuevo todas las propiedades y los métodos comunes en la clase **Vendedor**. Pero, ¿qué ocurre si después tenemos que introducir más clases de empleados: administrativo, ingeniero, etc.?

La programación orientada a objetos permite utilizar el concepto de **subclase**.

Cuando encuentras una clase que especializa otra clase existente, entonces puedes hacer que la primera sea subclase de la segunda. La ventaja que tiene esto es que la subclase hereda las características de la clase más sencilla y, además, añade nuevas características.

En PHP, para indicar que una clase es subclase de otra, se utiliza la palabra clave **extends**.

```
include_once("empleado.inc.php");
class Vendedor extends Empleado
```

De esta forma estamos indicando que la clase **Vendedor** es una subclase de la clase **Empleado**. Esto quiere decir que, automáticamente, los objetos de la clase **Vendedor** tienen todas las propiedades y métodos de la clase **Empleado**.

Adicionalmente, podemos crear más propiedades y métodos en la subclase. Por ejemplo, en el caso de un vendedor, será interesante conocer qué **ventas** ha realizado (propiedad), qué **comisión** se lleva (propiedad) y representar que se ha realizado una **venta** (método). Fíjate cómo lo haríamos.

Archivo **vendedor.inc.php**:

```
class Vendedor extends Empleado
{
     private $ventas;
     private $comision;
     public function vender($cantidad)
     {
           $this->ventas = $this->ventas + $cantidad;
     }
}
```

Si puedes decir que X es Y, entonces seguramente X es subclase de Y.

Como puedes ver, el vendedor es un empleado especial que tiene características que otros no tienen.

En toda relación de herencia encontrarás que puedes utilizar la expresión *"es un"*. Observa que esto ocurre en nuestro caso: un vendedor es un empleado que tiene una parte variable de su sueldo en comisiones.

Sin embargo, al revés no es cierto: un empleado no es un vendedor... ya que no todos los empleados tienen por qué ser vendedores.

A la clase más sencilla se la llama clase base o **superclase**, mientras que aquella clase que la especializa o que le añade nuevas características se llama clase derivada o **subclase**.

3. CREAR OBJETOS DE LAS SUBCLASES

Un asunto delicado ocurre cuando deseas crear objetos de una subclase: ¿qué constructor se utiliza?

Si hemos escrito un constructor en la superclase, entonces este se ejecuta al crear un objeto de la subclase, suponiendo que no hemos escrito ninguno en ella.

Sin embargo, si en la subclase escribimos un constructor, entonces es este el que se ejecutará al crear objetos de la subclase.

En este último caso, es posible que se necesite realizar lo mismo que hacía el constructor de la superclase, más otras tareas adicionales.

PHP introduce la expresión **parent::** para poder acceder a las propiedades y métodos de la superclase.

Vamos ahora a añadir el constructor de la clase **Vendedor**. Lo primero que debes hacer es construir el objeto como un empleado y después añadir las nuevas características de los vendedores.

```
public function __construct($nombre, $salario, $fecha,
                            $comision)
{
     parent::__construct($nombre, $salario, $fecha);
     $this->ventas = 0;
     $this->comision = $comision;
}
```

Fíjate que en la primera línea estamos llamando al constructor de la superclase y le pasamos los parámetros que necesita.

El resto del código completa las características únicas de los vendedores. En el ejemplo, se pone a 0 el número de unidades vendidas y establecemos la comisión particular para dicho vendedor.

Recuerda que no es obligatorio escribir el constructor de la subclase, ya que, en este caso, PHP llamará al de la superclase por ti.

Sin embargo, es fácil que en el constructor de la subclase tengamos que hacer algo más, como hemos visto en nuestra clase **Vendedor**. Por lo tanto, escribiremos el nuevo constructor, llamando explícitamente al de la superclase.

4. SOBRESCRIBIR MÉTODOS

Hemos visto que una subclase puede incluir nuevas propiedades e incluso nuevos métodos de forma que amplíen las características de su superclase.

Sin embargo, en ocasiones esto no es suficiente.

Es posible que, en lugar de añadir nuevos métodos, se tenga que modificar el funcionamiento de alguno de los heredados.

Esto ocurre en nuestro ejemplo. Recuerda que, a la hora de calcular el salario de un vendedor, no es válido utilizar el método heredado de la clase **Empleado**, ya que este método no tiene en cuenta las comisiones del vendedor.

Es posible modificar el funcionamiento de un método heredado. Esto se conoce con el nombre de **sobrescribir** o **reemplazar** los métodos.

```
public function calcularSalario()
{
     return ($this->salario + ($this->ventas *
             $this->comision / 100));
}
```

Como puedes ver, escribimos el nuevo método **calcularSalario** que ya teníamos en la clase **Empleado**. Como se llama exactamente igual, entonces realmente estaremos reemplazándolo.

En el caso del empleado, simplemente se devuelve el salario que tiene establecido, mientras que el cálculo para un vendedor es un poco más complejo, ya que se tiene en cuenta no solo el salario fijo del vendedor, sino también las ventas que ha realizado, ya que se lleva cierta comisión mensual.

Lo que tienes que entender es que, cuando crees un objeto **Vendedor** y utilices su método **calcularSalario**, estarás utilizando el método sobrescrito y no el heredado.

Lógicamente, esto no será siempre necesario; es decir, en la mayoría de los casos los métodos heredados serán válidos en las subclases y solo necesitarás sobrescribir aquellos que no deban funcionar del mismo modo.

Vamos a probar si todo esto funciona como esperamos. Para ello, crearemos una nueva página y utilizaremos ambos archivos.

Archivo **nominas.php**:

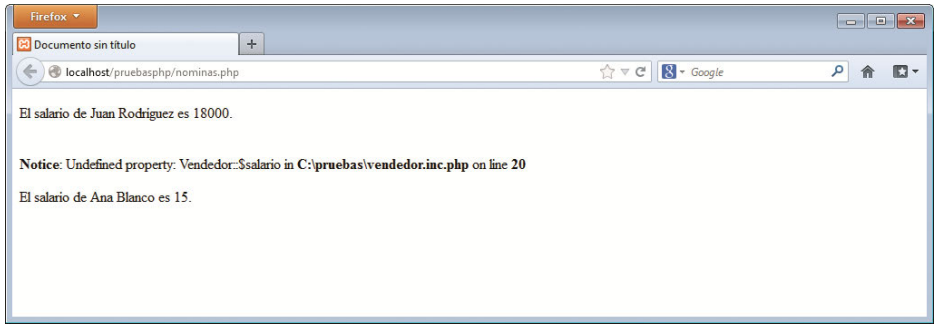
```
<body>
<?php
   include_once("empleado.inc.php");
   include_once("vendedor.inc.php");
   $juan = new Empleado("Juan Rodríguez", 18000,
                        "21/07/2005");
   $ana = new Vendedor("Ana Blanco", 18000,
                        "21/07/2005", 15);
   $sueldo = $juan->calcularSalario();
   echo "<p>El salario de Juan Rodríguez es
        $sueldo.</p>";
   $ana->vender(100);
   $sueldo = $ana->calcularSalario();
   echo "<p>El salario de Ana Blanco es $sueldo.</p>";
?>
</body>
```

Repasando este código vemos que:

- Se incluyen los archivos de las clases **Empleado** y **Vendedor**.
- Se crea un objeto de la clase **Empleado** y otro de la clase **Vendedor** utilizando sus correspondientes constructores, a los que les pasamos los parámetros que necesitan.

  Fíjate que ambos empleados tienen el mismo salario: **18000**.

- Se utiliza el método **vender** de la vendedora **Ana** para representar esa venta.
- Se utiliza el método **calcularSalario** del empleado y del vendedor y se imprime el resultado.



El resultado es un poco desconcertante porque el cálculo del salario ha funcionado bien para el objeto de la clase **Empleado** pero no para el de la clase **Vendedor**. Ahora veremos por qué.

5. EL ACCESO PROTECTED

Parece que el problema está en el método **calcularSalario** de la clase **Vendedor**, ya que no funciona demasiado bien.

```
public function calcularSalario()
{
     return ($this->salario + ($this->ventas *
             $this->comision / 100));
}
```

¿Qué sucede? Muy sencillo, fíjate que se utiliza la expresión **`$this->salario`** para acceder al salario del vendedor.

Hay un problema en esto, ya que dicha propiedad aparece en la clase **Empleado** y allí se definió como **privada** para la clase.

Es decir, que se da la paradoja de que aunque el vendedor es un empleado, no puede acceder a una de las propiedades privadas de la clase **Empleado**.

Existen dos formas de solucionar este problema:

1.- Puedes crear un método **public** o **protected** en la clase **Empleado** de forma que se acceda a la propiedad privada (el típico procedimiento **get**) y utilizar este método en la clase **Vendedor**.

2.- Puedes declarar la propiedad como **protected** en lugar de private.

Archivo **empleado.inc.php**:

```
class Empleado
{
     .....
     protected $salario;
     .....
}
```

El modificador **protected** permite que la propiedad o método sea accesible tanto en la clase en la que se define como en sus subclases.

Con este cambio sí que funciona bien, ejecutándose la versión correcta del método **calcularSalario** en función de si se trata de un objeto **Empleado** o un objeto **Vendedor**.

En este último caso, además de su salario, recibe el 15% de las ventas, que ascendían a 100 unidades en este mes. Por lo tanto, su salario es de **18015** unidades monetarias.

De todas formas, es una buena costumbre ocultar las propiedades con el modificador **private** y escribir los métodos **get / set** si necesitamos que se utilicen fuera de la clase.