

fundacionunirioja.adrformacion.com © ADR Infor SL
Héctor García González

Programación orientada a objetos © ADR Infor SL

fundacionunirioja.adrformacion.com © ADR Infor SL
Héctor García González

fundacionunirioja.adrformacion.com © ADR Infor SL
Héctor García González

Indice

Programación orientada a objetos	3
Clases y objetos	4
Sintaxis de una clase	5
La pseudovariable \$this	7
Ámbito de los métodos y propiedades de una clase	9
Propiedades estáticas	12
Métodos estáticos	14
Constructores y destructores	15
Herencia de clases	19
Ámbito de los métodos y propiedades de una clase heredada	20
Sobrescritura de métodos	22
Clases abstractas	24
Métodos abstractos	26
Métodos finales	27
Hemos aprendido	28
Ejercicios	30
Ejercicio 1: Librería de conexión a base de datos MySQL	30
Lo necesario para comenzar	30
Pasos a seguir	30
Solución	30
Ejercicio 2: Definición de clases y subclases	31
Lo necesario para comenzar	31
Pasos a seguir	31
Solución	31
Recursos	32
Preguntas Frecuentes	32
Glosario	32

Programación orientada a objetos



Al finalizar esta unidad el alumno será capaz de comprender los fundamentos de la programación orientada a objetos en PHP, definir clases y crear objetos. Del mismo modo, será capaz de crear clases que hereden de otras clases y conocerá el ámbito de las variables y funciones contenidas en las mismas.

La programación orientada a objetos es un estilo de programación que intenta acercar el código de programación a un lenguaje mucho **más humano**.

Se basa principalmente en crear unas entidades, denominadas **objetos**, que tienen sentido por si solas. Estos objetos, **generalmente representan una entidad del mundo real** que suele ser mucho más tangible para el ser humano que un conjunto variables y funciones sueltas. Sobre estos objetos, se podrán realizar una serie de **acciones** y contendrán cierta **información**.



Podemos crear un objeto que represente a un cliente de nuestra empresa.

Sobre este cliente podemos realizar diversas **acciones** como por ejemplo:

- Emitir una factura
- Abrir una incidencia
- Enviar un presupuesto

Y tendremos cierta **información** como:

- Nombre
- Dirección
- Comercial asignado
- Cantidad facturada este año

Todo ello, **se encapsulará** en un único objeto que utilizaremos para relacionarnos con el cliente.

Cuando nos enfrentamos a un desarrollo de cierta envergadura, organizar nuestra programación en objetos será de vital importancia. **Simplificará enormemente nuestro código** haciéndolo muy fácil de interpretar y mantener en el futuro. Un objeto bien definido, guiará nuestra programación y nos orientará sobre qué debemos hacer, qué no debemos hacer y cómo hacerlo en cada momento.

Algunas de las características de los objetos son:

1

Contiene una serie de **datos** y **acciones** encapsulados en su interior, que interactúan entre sí de la forma que hemos definido.

2

Solo podemos comunicarnos con los datos y acciones del objeto, por los canales de comunicación que establezcamos en él.

3

Tiene sentido por si mismo, independientemente de donde se utilice.

Como hemos dicho, un objeto **tiene sentido por si mismo**. Esto quiere decir, que es muy habitual que, una vez desarrollado, sea empleado tal cual, en otros desarrollos. Prácticamente podemos interpretar que se trata de un programa que utilizamos dentro de nuestro programa.

La gran mayoría de lenguajes de programación actuales disponen de estructuras para implementar la programación orientada a objetos. Evidentemente PHP también dispone de ellas y gracias a eso, los desarrollos web pueden crecer prácticamente sin límite, sin perder por ello el control y la legibilidad de nuestros programas.

Clases y objetos

Antes de continuar, debemos tener muy claro **qué es una clase** y **qué es un objeto**.

Clase

Una clase es la **definición del funcionamiento de una entidad**. Es un concepto abstracto como por ejemplo: cliente, documento o trabajador.

En la clase definimos qué datos tendrá el futuro objeto, qué funciones y cómo se comunicará el programa principal con el objeto.

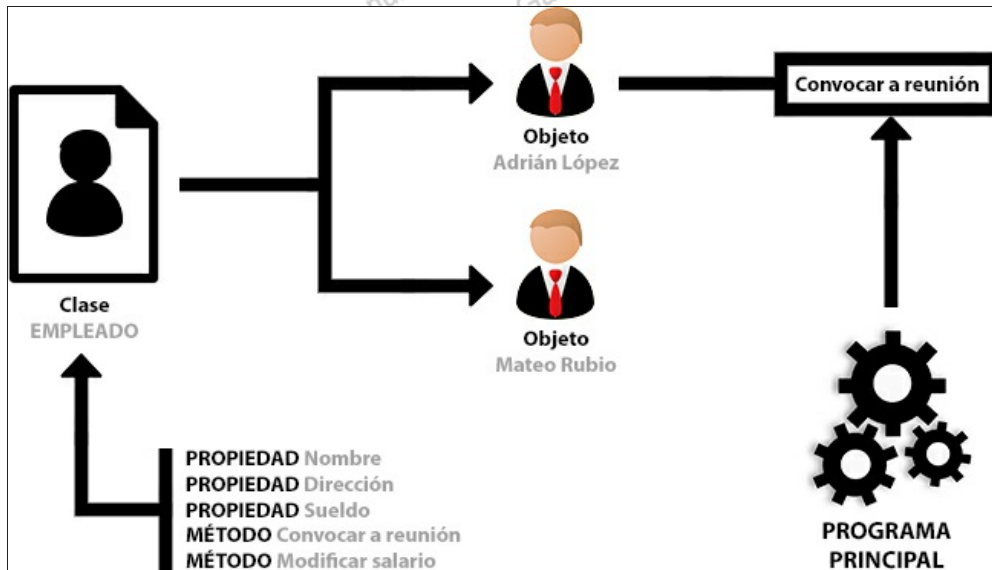
Podemos decir que es el plano con el cual se construyen los objetos.

Objeto

Un objeto es una **entidad concreta** creada a partir de una clase. Por ejemplo: el cliente Diego Martínez, el documento B4332 o el trabajador Francisco Ulecia. Si la clase es el plano, el objeto es la construcción que se realiza del plano.

Si hacemos el símil con una fábrica de automóviles, la clase sería el plano del vehículo y el objeto el vehículo terminado. Con un mismo plano, podemos hacer muchos coches, de la misma forma que con una misma clase, podemos hacer muchos objetos.

Nuestro programa principal se comunicará con los objetos que creamos, por las vías que fueron definidas en la clase.



Podemos ver en el ejemplo del gráfico cómo se **definen propiedades** (variables) y **métodos** (funciones) **en la clase empleado**. A partir de esta clase, **generamos dos objetos** (dos empleados). Cada uno de estos objetos, tendrán las propiedades definidas en la clase con su valor correspondiente y los métodos programados en la clase. Cuando el programa principal, quiere realizar la acción de convocar a reunión a Adrián López, simplemente **llama al método del objeto correspondiente**.

De esta forma, **el código del programa principal es muy simple y fácil de comprender**.



Además, cuando queramos cambiar el proceso que debemos llevar a cabo cuando se convoca a una reunión a un empleado, sólo debemos modificar la función correspondiente en la clase.

Esta modificación será mucho más simple que en el programa principal, porque dentro de la clase tendremos todas las funciones y variables que necesitamos, es decir, tenemos un entorno cerrado que tiene sentido por sí mismo.

Dentro del objeto podríamos guardar la agenda de citas del empleado para ver si está disponible o no en la hora de reunión propuesta y anotar la nueva cita en ella.

Sintaxis de una clase

Como sucede con las funciones, en PHP podemos definir nuestras clases **en cualquier lugar del código**, ya sea antes, o después de su utilización. No obstante, suele ser habitual declararlas **en un fichero .php aparte** ya que, como hemos comentado, es frecuente que reutilicemos nuestras clases en otros desarrollos. De esta forma es muy sencillo importarla o exportarla e incluirla en nuestros códigos mediante un **include**.



Proceso: Sintaxis de una clase en PHP

La sintaxis de una clase en PHP es la siguiente:

```
class nombre_clase {

    public $propiedad1;
    public $propiedad2;
    public $propiedad3;

    public function metodo1() {
        // Acciones del método 1
    }

    public function metodo2() {
        // Acciones del método 2
    }

}
```

Como podemos ver, se trata de una estructura que contiene una serie de variables y funciones. Las variables de una clase, reciben el nombre de **propiedades** y las funciones, **métodos**.

Una vez definida la clase, podemos crear tantos objetos como necesitemos de la misma. La sintaxis para crear un objeto es la siguiente:

```
$objeto = new nombre_clase();
```

Una vez definido el objeto, podemos acceder a sus propiedades y métodos de la siguiente forma:

Para acceder a una propiedad

```
$objeto -> propiedad1;
```

Para acceder a un método

```
$objeto -> metodo1();
```

Utilizamos el operador **->** (guión + mayor que) para acceder a cada una de las componentes de un objeto.

Vamos a ver un ejemplo de uso de un objeto.

Tenemos el siguiente código en el archivo `saludo.class.php` que guardamos en la carpeta "librerías" dentro de la carpeta pública:

```
<?php

class saludo {

    public function saludar($nombre) {
        echo 'Hola ' . $nombre;
    }

}

?>
```

Por otra parte, tenemos este código en el archivo `index.php` ubicado en la carpeta pública:

```
<?php
include('librerías/saludo.class.php');
$objeto = new saludo();
$objeto -> saludar('Pedro');
?>
```

Al marcar la dirección de nuestro servidor en el navegador, obtendremos el siguiente resultado:

Hola Pedro



Creando clases y objetos en PHP

La pseudovariable `$this`

Un objeto, representa una porción de código, que tiene sentido por sí mismo. Las propiedades y métodos contenidos en él, no son entes sin relación, sino que interactúan entre sí dando sentido al objeto.



Si volvemos al ejemplo de la **clase empleado**, podemos encapsular en ella:

- La propiedad **salario**
- El método **modificar_salario**

El método **modificar_salario**, accederá a la propiedad **salario** para modificarla, aplicando antes la lógica necesaria para llevar a cabo la operación:

- Validará que la modificación del salario respete el salario mínimo establecido para este tipo de trabajador.
- Creará una notificación para el departamento de contabilidad para que esté al tanto del cambio

Para poder acceder desde un método de un objeto, al resto de propiedades y métodos del mismo objeto, necesitamos utilizar la pseudovariable **\$this**.

\$this es una variable predefinida en todos los objetos de PHP que hace referencia al propio objeto. La utilizaremos para referenciar cualquier método o propiedad desde dentro de la clase.

Utilizaremos \$this, como si se tratase de un objeto normal. Por lo tanto la sintaxis para acceder a una propiedad desde un método del mismo objeto será la siguiente:

\$this -> propiedad1;



Tenemos el siguiente código PHP:

```
<?php
$empleado1 = new empleado();
$empleado1 -> set_salario(1200);
echo $empleado1 -> get_salario() . '<br>';
$empleado1 -> set_salario(800);
echo $empleado1 -> get_salario() . '<br>';

class empleado {

    public $salario;

    public function set_salario($valor) {
        if ($this -> validar_aumento($valor)) {
            $this -> salario = $valor;
        }
    }

    public function get_salario() {
        return $this -> salario;
    }

    public function validar_aumento($valor) {
        if ($valor < 900) {
            return false;
        } else {
            return true;
        }
    }

}

?>
```

Al llamarlo desde el navegador obtendremos el siguiente resultado:

```
1200
1200
```


Podemos observar en el ejemplo, cómo al llamar al método `set_salario` del objeto, éste utiliza `validar_aumento` y `$salario` mediante la pseudovariable `$this`.

Del mismo modo, el método `get_salario`, devuelve el valor de `$salario` utilizando `$this`. Es habitual utilizar métodos para establecer y recuperar el valor de algunas propiedades, en vez de acceder directamente a la propiedad, para poder añadir código a la misma. Quizá en este momento sea absurdo tener un método `get_salario` que simplemente devuelve el valor de `$salario`, pero haciéndolo de este modo, en futuras modificaciones, podremos restringir o condicionar el dato devuelto.



Importante: Obligatoriedad de `$this`

Siempre debemos utilizar `$this` para acceder a las propiedades y métodos del mismo objeto.

No es posible acceder a ellos escribiendo directamente su nombre.

Como es lógico, la pseudovariable `$this` sólo tiene sentido dentro de un objeto.

Utilizarla fuera de él provocará un error.



Uso de la pseudovariable `$this`

Ámbito de los métodos y propiedades de una clase

Habrás podido observar, que los métodos y propiedades de una clase que hemos definido hasta el momento, tienen la palabra reservada **public** delante de su definición. Esto significa, que el método o propiedad, es accesible desde fuera del objeto y el código principal puede invocarlo mediante el operador `->`.

No obstante, es muy normal que queramos tener métodos o propiedades privadas, que no puedan ser llamadas desde el código principal y que sólo existan desde la operativa interna de la clase. Para ello, utilizaremos la palabra reservada **private**.

Cuando declaramos un método o propiedad como **private**, sólo podrán acceder a él los métodos de la misma clase mediante la pseudovariable `$this`.



Ejemplo: Uso de public y private

```
$mi_objeto = new mi_clase();
echo $mi_objeto -> variable_publica; // Correcto
echo $mi_objeto -> variable_privada; // Producirá un error
$mi_objeto -> metodo_publico(); // Correcto
$mi_objeto -> metodo_privado(); // Producirá un error

class mi_clase {

    private $variable_privada;
    public $variable_publica;

    private function metodo_privado() {
        echo $this -> variable_publica; // Correcto
        echo $this -> variable_privada; // Correcto
    }

    public function metodo_publico() {
        echo $this -> variable_publica; // Correcto
        echo $this -> variable_privada; // Correcto
        $this -> metodo_privado(); // Correcto
    }

}
```

De esta forma, podemos **impedir el acceso a ciertas secciones del objeto** evitando que se utilice de forma inadecuada. Si permitimos **únicamente los canales de comunicación que nos interesan** mediante el uso de **public** y **private**, reduciremos las posibilidades de error. El programador que utilice finalmente la clase, **no podrá hacerlo de forma errónea**.

Vamos a volver al ejemplo de la clase empleado:

Un programador no familiarizado con la clase, podría acceder directamente a la propiedad \$salario, evitando todas las validaciones.

```
<?php
```

```
$empleado1 = new empleado();
$empleado1 -> salario = 800; // Estoy saltándome la validación del salario
echo $empleado1 -> get_salario();
```

```
class empleado {

    public $salario;
```

```

public function set_salario($valor) {
    if ($this -> validar_aumento($valor)) {
        $this -> salario = $valor;
    }
}

public function get_salario() {
    return $this -> salario;
}

public function validar_aumento($valor) {
    if ($valor < 900) {
        return false;
    } else {
        return true;
    }
}

}

?>

```

Si establecemos los ámbitos public y private adecuadamente solucionamos el problema.

```

<?php

$empleado1 = new empleado();
$empleado1 -> set_salario(800); // No puedo usar $salario, uso set_salario()
echo $empleado1 -> get_salario();

class empleado {

    private $salario;

    public function set_salario($valor) {
        if ($this -> validar_aumento($valor)) {
            $this -> salario = $valor;
        }
    }

    public function get_salario() {
        return $this -> salario;
    }
}

```

```
private function validar_aumento($valor) {
    if ($valor < 900) {
        return false;
    } else {
        return true;
    }
}

?>
```



En este ejemplo, **hemos establecido un canal de comunicación** en el objeto, de forma que el código principal, solo puede utilizar `set_salario` y `get_salario`. De esta forma, hacemos que el objeto cobre sentido y no pueda ser utilizado de forma inadecuada.



Ámbito de métodos y propiedades

Propiedades estáticas

Cuando tenemos varios objetos de una misma clase, cada uno tiene sus propiedades, con valores diferentes.



```
$objeto1 = new mi_clase();
$objeto2 = new mi_clase();
$objeto1 -> nombre = 'Pedro';
$objeto2 -> nombre = 'María';
```

```
class mi_clase {
    public $nombre;
}
```

La propiedad `$nombre` de `$objeto1` vale 'Pedro'.

La propiedad `$nombre` de `$objeto2` vale 'María'.

No obstante, existe la posibilidad de crear propiedades, que al cambiar su valor, lo hagan en todos los objetos creados de la clase. Estas propiedades son denominadas **estáticas** y tienen un mismo valor para todos los objetos de la clase.

Para crear una propiedad estática, utilizaremos la siguiente sintaxis:

```
public static $nombre_propiedad;
```

Y para acceder a una propiedad estática, usaremos la siguiente sintaxis:

```
nombre_clase :: $nombre_propiedad;
```



Importante: Fíjate bien en la sintaxis.

Observa que estamos utilizando el nombre de la **clase** y no uno de los objetos de la clase.

Esto es así, porque la propiedad no se almacena en un único objeto, sino en la propia clase. De hecho, **no es necesario que exista ningún objeto de la clase** para establecer y recuperar el valor de una propiedad estática.

Por este motivo, se utiliza el operador `::` (dos puntos + dos puntos) en lugar del operador habitual `->`.



Vamos a ver un ejemplo de uso. Tenemos el siguiente código PHP:

```
mi_clase :: $nombre = 'Juan';
$objeto1 = new mi_clase();
$objeto2 = new mi_clase();
$objeto1 -> mostrar_nombre();
echo '<br>';
$objeto2 -> mostrar_nombre();

class mi_clase {

    public static $nombre;

    public function mostrar_nombre() {
        echo mi_clase :: $nombre;
    }

}
```

Al llamarlo desde el navegador obtendremos el siguiente resultado:

```
Juan
Juan
```

Como puedes ver, podemos llamar a una variable estática tanto desde fuera de la clase, como desde dentro de ella con la misma sintaxis.

También podemos establecer la propiedad como **private**. En tal caso, sólo podríamos acceder a ella desde dentro de la clase.

Métodos estáticos

Del mismo modo que sucede con las propiedades, podemos crear métodos estáticos utilizando la siguiente sintaxis:

```
public static nombre_metodo() {
    // Acciones
}
```

Para acceder a un método estático, usaremos la siguiente sintaxis:

```
nombre_clase :: nombre_metodo();
```

La finalidad de crear este tipo de métodos es **no necesitar la creación de un objeto** para acceder al mismo. Suele utilizarse cuando utilizamos una clase, más como repositorio de funciones, que como un objeto funcional en sí.

Vamos a ver un ejemplo:

Tenemos el siguiente código PHP:

```
mi_clase :: saludar('Juan');

class mi_clase {

    public static function saludar($nombre) {
        echo "Hola $nombre";
    }

}
```

Al llamarlo desde el navegador obtendremos el siguiente resultado:

Hola Juan

En este ejemplo, observamos como no es necesario crear un objeto para utilizar la función **saludar**, pudiendo utilizar así la clase como repositorio de funciones.



Importante: \$this

Dentro de un método estático, **no podemos utilizar \$this**, ya que dicho método no se ejecuta dentro de un objeto.

odas las propiedades y métodos a las que acceda un método estático, deben ser también estáticos.



Llamando a métodos y propiedades estáticas desde el objeto

Aunque lo más habitual es utilizar el nombre de la clase para utilizar propiedades o métodos estáticos, también podemos hacerlo desde cualquier objeto de esta clase:

```
$objeto1 = new mi_clase();
$objeto2 = new mi_clase();
$objeto1 :: $valor = 78;
echo $objeto2 :: $valor;
```

```
class mi_clase {

    public static $valor;

}
```

El funcionamiento de una variable estática no cambia con esta sintaxis. Al asignar un valor a la variable desde un objeto, el valor de ésta variable cambiará en todos los objetos de la misma clase.

También podemos usar la pseudovariable \$this para acceder a una variable estática desde un método (no estático) del propio objeto:

```
$this :: $valor;
```



Propiedades y métodos estáticos

Constructores y destructores

Un objeto debe tener sentido en cualquier momento de su ciclo de vida. Esto quiere decir que, desde el momento que se crea, sus propiedades deben tener un valor que sea coherente.

Es muy frecuente, que necesitemos realizar una serie de cálculos **en el momento de la creación del objeto**, para asignar los valores iniciales de sus propiedades.



En nuestra clase **empleado**, no tiene sentido que un trabajador comience con su propiedad **salario** sin establecer.

En el momento de la creación del objeto empleado, nos conectaremos a la base de datos, para recuperar cuál es su sueldo y establecer el valor correspondiente de la propiedad salario.

Constructores

Para realizar estos procesos iniciales, tenemos los **constructores de clase**, que se ejecutan **en el momento de la creación del objeto**. Su sintaxis es la siguiente:

```
class nombre_clase {

    public function __construct() {
        // Acciones a realizar cuando se crea el objeto
    }

}
```



Importante: Parámetros en el constructor

Si lo necesitamos, podemos establecer parámetros en la función del constructor:

```
class nombre_clase {

    public function __construct($parametro1,$parametro2) {
        // Acciones a realizar cuando se crea el objeto
    }

}
```

En tal caso, es necesario indicarlos en el momento de la creación del objeto:

```
$nombre_objeto = new nombre_clase($valor1,$valor2);
```

Es obligatorio indicar el mismo número de parámetros que hemos establecido en el constructor de la clase a la hora de crear el objeto. No obstante, como sucede con cualquier función de PHP podemos establecer parámetros opcionales en el constructor.

Destructores

Del mismo modo que existen constructores de clase, existen **destructores de clase**, que se ejecutan cuando el objeto es destruido. Suelen utilizarse para realizar operaciones y guardados finales. Su sintaxis es la siguiente:

```
class nombre_clase {

    public function __destruct() {
        // Acciones a realizar cuando se destruye el objeto
    }

}
```



Cuándo se ejecuta un destructor de clase

Si no lo hemos hecho de manera explícita utilizando la función **unset**, el destructor se ejecutará cuando finalice el script de PHP.

Vamos a ver un ejemplo completo:

Tenemos el siguiente código PHP:

```
<?php

include('librerias/conexion_bbdd.php');

$empleado1 = new empleado(3457);

$nuevo_salario = $empleado1 -> get_salario() * 1.10;
$empleado1 -> set_salario($nuevo_salario);

$nuevo_salario = $empleado1 -> get_salario() * 1.10;
$empleado1 -> set_salario($nuevo_salario);

class empleado {

    private $salario;
    private $salario_inicial;
    private $id;

    public function __construct($numero_empleado) {
        $sql = "SELECT salario FROM empleados WHERE id = $numero_empleado";
        $datos = $GLOBALS['conexion'] -> query($sql);
        $this -> salario = $datos[0]['salario'];
        $this -> salario_inicial = $this -> salario;
        $this -> $id = $numero_empleado;
    }

}
```

```

public function __destruct() {
    if ($this -> salario_inicial != $this -> salario) {
        $sql = 'UPDATE empleados ';
        $sql .= 'SET salario = ' . $this -> salario . ' ';
        $sql .= 'WHERE id = ' . $this -> $id;
        $GLOBALS['conexion'] -> query($sql);
    }
}

public function set_salario($valor) {
    $this -> salario = $valor;
}

public function get_salario() {
    return $this -> salario;
}
}

```

?>

Al llamarlo desde el navegador se producen las siguientes acciones:

- Se crea el objeto `$empleado1` a partir de su número de empleado (**3457**) y se ejecuta el constructor que realiza las siguientes acciones:
 - Recupera el sueldo del empleado 3457 de la base de datos y lo almacena en la propiedad `$salario`.
 - También almacena el sueldo en la propiedad `$salario_inicial` para conocer cuál era el salario del empleado cuando se creó el objeto.
 - Guarda el número de empleado en la propiedad `$id`.
- Se recupera el salario de `$empleado1` con el método `get_salario`, se sube un 10% y se almacena de nuevo en el objeto con el método `set_salario`.
- Volvemos a realizar el paso anterior, incrementando otro 10% el salario del trabajador.
- El script finaliza, ejecutando el destructor del objeto `$empleado1` que realiza las siguientes acciones:
 - Comprueba si el salario del empleado se ha modificado durante la vida del objeto.
 - En caso de haberse modificado, guarda el nuevo salario en la base de datos.

Observa que al realizar el guardado del salario en la base de datos **con el destructor**, en vez de hacerlo dentro del método `set_salario`, **hemos evitado actualizar dos veces el mismo dato innecesariamente**. De esta forma aumentamos la eficiencia del script.

NOTA: Para este ejemplo hemos utilizado un objeto almacenado en `$GLOBALS['conexion']` que realiza todas las peticiones a la base de datos.



Herencia de clases

Una de las características que más definen a la programación orientada a objetos, y que la dota de mayor potencia, es la herencia de clases.

Este concepto consiste en indicar a una nueva clase, que **debe contener los métodos y propiedades de otra clase existente**.



Hemos definido una **clase empleado**, que contiene los métodos y propiedades **comunes para todos los trabajadores** de la empresa.

No obstante, tenemos diferentes tipos de empleados. Los trabajadores de mantenimiento tienen **funciones específicas** de su puesto, como puede ser, la necesidad de abrir un parte de reparación de máquina.

Crearemos una nueva clase llamada **empleado_mantenimiento**, que **heredará de la clase empleado**. De esta forma no es necesario reescribir todos los métodos y propiedades para cada tipo de empleado, y cuando deseemos realizar una modificación para todos los empleados, lo haremos en un único lugar.

La sintaxis para crear una clase heredada es la siguiente:

```
class nombre_clase extends clase_padre {
    // Definición de los métodos y propiedades exclusivos de nombre_clase
}
```

Vamos a ver un ejemplo de su funcionamiento:

Tenemos el siguiente código PHP:

```
<?php

$objeto = new clase_hija();
$objeto -> metodo_padre();
$objeto -> metodo_hija();

class clase_padre {

    public function metodo_padre() {
        echo 'Se ha ejecutado un método de la clase padre<br>';
    }

}

class clase_hija extends clase_padre {

    public function metodo_hija() {
        echo 'Se ha ejecutado un método de la clase hija<br>';
    }

}
```

```
}
?>
```

Al llamarlo desde el navegador obtendremos el siguiente resultado:

Se ha ejecutado un método de la clase padre
Se ha ejecutado un método de la clase hija



Herencia de clases

Ámbito de los métodos y propiedades de una clase heredada

Vamos a volver a analizar, cómo funcionan los ámbitos **public** y **private** en relación a la herencia de clases e introduciremos un nuevo ámbito: **protected**.

public

Un método o propiedad **public** será accesible...

- Desde un método de la **propia clase**.
- Desde un método de la **clase heredada**.
- Desde el **código principal**.



```
$objeto = new clase_hija();
echo $objeto -> propiedad; // Correcto

class clase_padre {
    public $propiedad;

    private function metodo_padre() {
        echo $this -> propiedad; // Correcto
    }
}

class clase_hija extends clase_padre {
    private function metodo_hija() {
        echo $this -> propiedad; // Correcto
    }
}
```

Podemos decir, que utilizaremos el ámbito **public**, cuando la propiedad o el método necesitan ser **consultados desde el código principal**

private

Un método o propiedad **private** será accesible...

- Desde un método de la **propia clase**.

Y no será accesible...

- Desde un método de la **clase heredada**.
- Desde el **código principal**.



```
$objeto = new clase_hija();
echo $objeto -> propiedad; // Error

class clase_padre {
    private $propiedad;

    private function metodo_padre() {
        echo $this -> propiedad; // Correcto
    }
}

class clase_hija extends clase_padre {
    private function metodo_hija() {
        echo $this -> propiedad; // Error
    }
}
```

Podemos decir, que utilizaremos el ámbito **private**, cuando la propiedad o el método **sólo debe poder ser consultado desde la propia clase** y las clases heredadas no deben acceder a él.

protected

Un método o propiedad **protected** será accesible...

- Desde un método de la **propia clase**.
- Desde un método de la **clase heredada**.

Y no será accesible...

- Desde el **código principal**.



```
$objeto = new clase_hija();
echo $objeto -> propiedad; // Error

class clase_padre {

    protected $propiedad;

    private function metodo_padre() {
        echo $this -> propiedad; // Correcto
    }
}

class clase_hija extends clase_padre {

    private function metodo_hija() {
        echo $this -> propiedad; // Correcto
    }
}
```

Podemos decir, que utilizaremos el ámbito **protected**, cuando la propiedad o el método **deben ser consultados desde el código principal, pero si desde cualquier clase que lo herede**.



Ámbitos de propiedades y métodos en clases heredadas

Sobrescritura de métodos

En PHP, tenemos la posibilidad de **sobrescribir un método heredado**, para indicar un nuevo funcionamiento adecuado para la subclase.



Tenemos nuestra clase **empleado_mantenimiento**, que hereda de clase **empleado**.

Uno de los métodos que hereda es **set_salario**. Pero los trabajadores de mantenimiento tienen unas restricciones específicas, en cuanto a la modificación de su salario, que no tienen el resto de trabajadores.

Volveremos a implementar el método **set_salario** en la clase **empleado_mantenimiento**, con la nueva funcionalidad y **reemplazará al método heredado**.

Vamos a ver un ejemplo de uso:

Tenemos el siguiente código PHP:

```
<?php

$objeto = new clase_hija();
$objeto -> metodo();

class clase_padre {

    public function metodo() {
        echo 'Método principal';
    }

}

class clase_hija extends clase_padre {

    public function metodo() {
        echo 'Método sobrescrito';
    }

}

?>
```

Al llamarlo desde el navegador obtendremos el siguiente resultado:

Método sobrescrito



Anotación: Parámetros en métodos sobrescritos

No es necesario que un método sobrescrito, tenga el mismo número de parámetros que el método heredado.

Su definición puede ser modificada y **siempre se requerirán los parámetros indicados en el método sobrescrito.**

Nos encontraremos en muchas ocasiones, con que no deseamos reemplazar el método heredado completo, sino **anexarle alguna acción adicional**. Para ello, podemos ejecutar el método heredado y realizar las acciones definidas en la clase padre mediante la siguiente sintaxis:

```
parent :: nombre_metodo();
```

Vamos a ver un ejemplo:

Tenemos el siguiente código PHP:

```

<?php

$objeto = new clase_hija();
$objeto -> metodo();

class clase_padre {

    public function metodo() {
        echo 'Método principal<br>';
    }

}

class clase_hija extends clase_padre {

    public function metodo() {
        parent :: metodo();
        echo 'Método sobrescrito<br>';
    }

}

?>

```

Al llamarlo desde el navegador obtendremos el siguiente resultado:

Método principal
Método sobrescrito



Constructores y destructores

Los constructores y destructores de clase, **funcionan como un método más** en todos los aspectos.

Es por ello que también **pueden ser sobrescritos** si es necesario.



Sobrescritura de métodos

Clases abstractas

En muchas circunstancias, nos encontraremos con que no tiene sentido que se genere un objeto de una determinada clase. Normalmente esto sucede cuando:

1

Tenemos una clase de la que heredan varias clases, pero que no tiene sentido por si sola.

2

Estamos utilizando una clase como repositorio de funciones estáticas y no tiene ninguna finalidad generar un objeto.

Ya que las clases están pensadas para ser reutilizadas y distribuidas, debemos asegurarnos de que el programador que se encuentre con nuestro conjunto de clases, no tenga dudas de cómo se pretendía que fuesen utilizadas cuando se implementaron.

Por este motivo nacen las **clases abstractas**. Una clase abstracta es una clase desde la cual **no se puede generar un objeto**.

Para indicar que una clase es abstracta, usaremos la siguiente sintaxis:

```
abstract class nombre_clase {
    // Definición de los métodos y propiedades de nombre_clase
}
```



Tenemos las siguientes clases:

```
abstract class empleado {

    public $nombre;
    public $salario;
    public $dni;

}

class empleado_mantenimiento extends empleado {

    public $partes_trabajo;
    public $maquinas_asignadas;

}

class empleado_comercial extends empleado {

    public $clientes;
    public $agenda;

}
```

Siempre que queramos generar un objeto que representa un empleado, hemos establecido que **debe generarse desde la clase heredada que representa su tipo de empleado**

Para que cualquier programador que deba implementar nuevas funcionalidades o modificar las existentes, no se equivoque y cree empleados desde la clase empleado, **restringimos esta posibilidad** estableciendo la clase como abstracta.



Clases abstractas

Métodos abstractos

Cuando tenemos una clase abstracta, cuya única finalidad es que otras clases hereden de ella, es posible que necesitemos especificar, que la subclase **debe** implementar un método concreto, sin el cual el objeto resultante podría no tener sentido.

Estas indicaciones se denominan **métodos abstractos** y tienen la siguiente sintaxis:

```
abstract class nombre_clase {

    abstract public function nombre_metodo();

}
```

Cualquier clase que herede de la anterior, **tiene la obligación** de implementar el método abstracto indicado. De lo contrario, se producirá un **Fatal error**.



Tenemos las siguientes clases:

```
abstract class empleado {

    public $nombre;
    public $salario;
    public $dni;
    abstract public function renovar_password();

}

class empleado_mantenimiento extends empleado {

    public $partes_trabajo;
    public $maquinas_asignadas;

    public function renovar_password() {
        // Implementación del método
    }

}
```

La clase empleado, es una **clase abstracta**, ya que debe definirse obligatoriamente una subclase que haga referencia al tipo de empleado.

No obstante, todos los empleados **deben tener un método para renovar su password** en la empresa. Las restricciones sobre la forma de renovar el password **varían de un tipo de puesto a otro**, por lo que indicamos, con un método abstracto, que es obligatorio definir este procedimiento cuando se implementa un nuevo tipo de empleado.



Los métodos abstractos, solo pueden ser declarados en clases abstractas.

Como sucede con las clases abstractas, los métodos abstractos ayudan al programador a utilizar el conjunto de clases de forma adecuada.

Métodos finales

Un método final, es aquel que **no puede ser sobrescrito** en una clase que hereda de su clase. Tiene la finalidad contraria que un método abstracto: evitar que se implemente cierta funcionalidad, pero con el mismo objetivo: orientar al programador sobre cómo utilizar la clase.

Definiremos un método como final cuando **debe existir tal y como es definido en la clase padre**, sin que pueda ser alterado en las clases que hereden de la misma.

La sintaxis de un método final es la siguiente:

```
class nombre_clase {  
  
    final public function nombre_metodo() {  
        // Acciones del método  
    }  
  
}
```



Tenemos las siguientes clases:

```
abstract class empleado {

    public $nombre;
    private $salario;
    public $dni;

    final public function set_salario($valor) {
        // Acciones del método
    }

}

class empleado_mantenimiento extends empleado {

    public $partes_trabajo;
    public $maquinas_asignadas;

}
```

El método `set_salario` es común a todas las clases que heredan de la clase `empleado`.

La forma de establecer el salario es **obligatoriamente la misma** para todos los empleados, por lo que impedimos que una clase heredada pueda implementar su propia forma de cambiarlo. Para ello hacemos que `set_salario` sea un **método final** y **no pueda ser sobrescrito**.



Métodos abstractos y finales

Hemos aprendido



En esta unidad hemos aprendido:

- La programación orientada a objetos, trata de **encapsular datos y acciones** en una misma entidad que **tiene sentido por sí sola**.
- Una clase es la **definición del funcionamiento de una entidad**.
- Un objeto es la **entidad en sí**. De una sola clase se pueden generar muchos objetos, que serán utilizados por nuestro código principal.
- Dentro de una clase hay funciones llamadas **métodos** y variables llamadas **propiedades**.
- Los métodos y propiedades de una clase pueden tener tres ámbitos diferentes: **public**, **protected** y **private**.
- Generaremos objetos de una clase con la instrucción **new** y accederemos a sus métodos y propiedades con el operador **->**
- Para acceder a las propiedades y métodos de una clase desde dentro de la clase, usaremos la pseudovariable **\$this**.
- Las propiedades y métodos **estáticos** son comunes a todos los objetos de una misma clase y se accede a ellos con el operador **::**
- En las clases se pueden definir **constructores**, que se ejecutan cuando se crea el objeto y **destructores** que se ejecutan cuando se destruye.
- Para que una clase herede de otra, utilizamos **extends**.
- Podemos **sobrescribir los métodos** heredados de otra clase, siempre que no sea un **método final**.
- No se puede generar un objeto a partir de una **clase abstracta**.
- Si definimos un **método abstracto**, debe ser implementado obligatoriamente en las clases que hereden de nuestra clase.

fundacionunirioja.adrformacio
Héctor García González

fundacionunirioja.adrformacion.com © ADR Infor SL
Héctor García González

ADR Infor SL

Ejercicios

Ejercicio 1: Librería de conexión a base de datos MySQL

Duración estimada del ejercicio



30
minutos

Para finalizar este ejercicio correctamente, deberás programar una clase que se encargue de gestionar las conexiones y las consultas que nuestro programa realice a una base de datos MySQL.

Lo necesario para comenzar



Para recuperar datos de una base de datos MySQL utilizamos las siguientes instrucciones:

```
$conexion = mysqli_connect($servidor,$usuario,$password,$bd);  
$conexion -> query('SET NAMES utf8');  
$resultado = $conexion -> query($sql);  
while ($fila = $resultado -> fetch_array()) {  
    // $fila será un array con los campos de la fila actual  
}
```

Pasos a seguir

1. Crea un archivo llamado **bbdd.php** y guárdalo en una carpeta llamada **lib** dentro de tu carpeta pública.
2. Dentro de él crea una clase para gestionar bases de datos MySQL
3. En el constructor de la clase, **abre la conexión con la base de datos**. Deberás indicar los parámetros de conexión al constructor.
4. Fuera de la función **crea un objeto** de la clase y almacénalo en una **variable global**.
5. Crea un método público en la clase llamado **sql**, al que se le pase un parámetro con una instrucción SQL. El método debe devolver un array bidimensional con las filas y columnas recuperadas de la consulta.
6. En tu archivo **index.php**, incluye tu librería **bbdd.php**.
7. Ejecuta una instrucción SQL desde el archivo principal utilizando tu objeto y comprueba que recupera los datos correctamente.


Solución



Solución al ejercicio

Ejercicio 2: Definición de clases y subclasses

Duración estimada del ejercicio



60
minutos

Para finalizar este ejercicio correctamente, deberás definir correctamente las clases y subclasses adecuadas para realizar una gestión de trabajadores. Tendrás que implementar los métodos de la forma más coherente posible para cumplir con los requisitos propuestos.

Lo necesario para comenzar

Descarga este archivo SQL e impórtalo en tu base de datos MySQL. Te servirá como base para desarrollar el ejercicio correctamente.



Pasos a seguir

1. Crea las clases necesarias para manejar adecuadamente las siguientes entidades, acorde con su estructura en la base de datos:
 1. Empleados de mantenimiento
 2. Empleados de contabilidad
 3. Máquinas
2. Crea un objeto para cada trabajador y máquina que hay en la base de datos.
3. Se aplicarán las siguientes restricciones:
 1. Los trabajadores no puede ganar menos de 900€
 2. Los trabajadores de contabilidad además no pueden ganar menos de 1000€
 3. A partir de un empleado de mantenimiento se pueden crear partes de trabajo indicándole una máquina.
4. Realiza las siguientes acciones:
 1. Sube el sueldo del empleado 2 a 1300€
 2. Crea un parte de trabajo

Solución



Solución al ejercicio

Recursos

Preguntas Frecuentes

1. ¿Puedo cambiar el ámbito de un método al sobrescribirlo en una clase que lo herede?

No, la sobrescritura de métodos siempre debe conservar el mismo ámbito.

2. ¿Puedo establecer un constructor como private?

No, ya que el código de fuera del objeto debe tener la capacidad de ejecutar el constructor en el momento de la creación del mismo.

Siempre debe ser **public**.

3. He observado códigos en los que utilizan una función cuyo nombre es el nombre de la clase, para indicar el constructor ¿Es correcto?

Esta forma de definir constructores está **deprecated** a partir de PHP 7, por lo que es recomendable utilizar el constructor con el nombre de función **__construct**.

4. He visto códigos de PHP que utilizan el operador:: para acceder a propiedades y métodos que no son estáticos. ¿Es correcto?

Es posible que hayas podido observar en códigos antiguos el uso de:: con elementos no estáticos, pero desde la versión 5 de PHP esta nomenclatura está deprecated y desde la versión 7 prohibida.

Siempre que accedas a propiedades o métodos no estáticos, utiliza el operador->

Glosario.

- **Clase:** Es la definición del funcionamiento de un objeto.
- **Constructor:** Método que se ejecuta en el momento de la creación de un objeto.
- **Destructor:** Método que se ejecuta en el momento de la destrucción de un objeto.
- **Herencia:** Habilidad que tienen las clases para adquirir todos los métodos y propiedades de otra clase.
- **Include:** La instrucción include, añade el código PHP de otro fichero dentro del script actual. Su sintaxis es: include('nombre_fichero.php');

Programación orientada a objetos

- **Métodos:** Funciones contenidas dentro de un objeto.
- **Objeto:** Entidad creada a partir de una clase, que aglutina datos y acciones en su interior.
- **Programación orientada a objetos:** Es un estilo de programación que intenta acercar el código de programación a un lenguaje mucho más humano. Se basa principalmente en crear unas entidades, denominadas objetos, que aglutinan datos y acciones en su interior y tienen sentido por si solas.
- **Propiedades y métodos estáticos:** Son propiedades y métodos comunes a todos los objetos de una misma clase. Son llamados haciendo referencia al propio nombre de la clase en vez de al objeto.
- **Propiedades :** Variables contenidas dentro de un objeto.
- **Unset:** La función unset, destruye una variable. Su sintaxis es: unset(\$variable);