PROGRAMACIÓN ORIENTADA A OBJETOS

Antonio Boronat Pérez
CFGS DAW
Des. Aplic. en Entorno Servidor
IES Joan Coromines (Benicarló)

Índice

Introducción a la Programación Orientada a Objetos (POO)

La *Programación Orientada a Objetos* (POO ó en inglés OOP) es una metodología de programación que aparece en los años 80 como una evolución de la *programación estructurada* y *modular,* a las que incluye. La POO supone una nueva vía para el análisis y desarrollo de programas.

En la *Programación Estructurada*, cuando creamos un programa, por un lado definimos las estructuras de información que contendrán los datos y resultados, y por otro definimos las instrucciones que operarán sobre las estructuras de información. En cambio, en la POO se definen objetos, que intentan modelar objetos reales que encontramos en el problema a solucionar, como por ejemplo el objeto cliente, el objeto factura, el objeto producto, etc. Al definir un objeto, se definen juntas tanto las informaciones asociadas al objeto como las instrucciones que operan con ellas, de manera que el objeto está formado por los datos y las instrucciones.

Por ejemplo si creamos un Objeto Cliente, podríamos definir como informaciones asociadas:

- NIF/CIF
- Nombre y apellidos
- Dirección
- Teléfono
- Nº de cuenta en que cargar las facturas
- Total facturado durante el año.

Y además las operaciones siguientes:

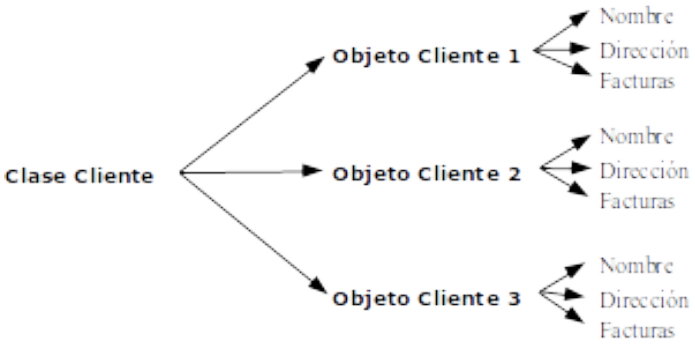
- Ver o cambiar el nombre y apellidos
- Ver o cambiar NIF/CIF
- Ver o cambiar la dirección
- Ver o cambiar el teléfono
- Ver o cambiar nº de cuenta bancaria
- Ver o cambiar el total facturado

La ventaja principal de la POO es la que se deriva de la ***reutilización*** del código escrito, esto es, que cuando tenemos un objeto, éste puede usarse en diferentes programas. Por ejemplo, antes hemos definido el objeto Cliente, que podremos usar en otros programas, bien como cliente, o bien con pequeñas modificaciones crear el objeto proveedor o empleado. Así ahorramos tiempo en el desarrollo de programas, pudiendo llegar al extremo de crear nuevos programas simplemente uniendo objetos que ya teníamos desarrollados. Esto es especialmente importante porque los entornos gráficos que usan los sistemas operativos actuales están escritos con POO, los botones, cajas de texto, ventanas, etc. son objetos. Así para crear una aplicación en un entorno gráfico, en primer lugar se creará la parte gráfica que usan los usuarios, añadiendo en ella los elementos necesarios disponibles en el entorno de desarrollo y después escribiremos el código que soluciona el problema planteado.

Conceptos de Programación Orientada a Objetos

El primer concepto a tener en consideración será el de ***Objeto***, que podemos definir como una entidad con identidad propia diferenciable de los demás objetos. Un objeto puede ser un avión, un niño, una página web. Cada objeto tiene asociada una información y unas acciones, pero además tiene una identidad que lo diferencia de los demás objetos que son iguales a él. Cada avión, aunque igual a los demás, es un objeto distinto. Así un objeto dentro de un programa también tiene una identidad diferenciada. Por ejemplo dos objetos que representan a proveedores, aunque tengan informaciones similares, el sistema los tratará como elementos diferentes.

En la POO el concepto principal es el de ***Clase***. Como hemos comentado, los objetos aunque iguales tienen identidades diferentes, pero es el concepto de Clase el que se encarga de definir las características de los objetos (sus datos y acciones). Por tanto, en la POO se estudian las clases de objetos y después de cada clase se crean los diferentes objetos usados en los programas. Por ejemplo, un programador deberá definir una clase cliente y en ella se pondrán las informaciones y acciones necesarias, y después se crearán los objetos que representarán cada uno de los diferentes clientes que intervienen en el problema:



Se podría decir que el proceso de definición de una clase y sus objetos es similar al que se aplica en Lenguaje C en la definición de las *struct* y de las variables de este tipo struct. Primero se crea la plantilla del tipo struct, en POO la clase, y posteriormente se definen las variables de este tipo, en POO los objetos. Por tanto, si usamos POO, la tarea principal consistirá en el estudio y definición de las clases necesarias para la resolución del problema planteado.

El uso de la POO favorece el concepto de ***Encapsulación***, que consiste en que vemos los objetos como elementos cerrados a los que podemos acceder mediante un mecanismo de entrada y uno de salida, sin que sepamos cómo funciona por dentro, ni las informaciones que maneja. Siendo los mecanismos de entrada y salida ofrecidos por los objetos propiedades ***Públicas*** del objeto (datos y acciones disponibles desde otros objetos de la aplicación). Y las propiedades que solamente son usadas internamente en el objeto se consideran propiedades ***Privadas***, y no se tendrá acceso a ellas más que desde dentro del propio objeto. El concepto de encapsulación sería equivalente al obtenido en un programa en C con el uso de módulos, de manera que cuando los llamamos a ejecución les pasamos unos parámetros y ellos nos devolverán unos resultados o acciones realizadas, pero no tenemos acceso a sus variables internas ni conocemos las instrucciones que ejecutan.

La encapsulación, al igual que en los lenguajes de programación estructurados permite crear librerías, en este caso de Objetos, que tienen como misión facilitar la reutilización de código, ayudando en el desarrollo de aplicaciones.
Otra ventaja que deriva de la encapsulación es la ***mantenibilidad****,* por que podemos modificar el interior de una clase sin modificar sus propiedades que enlazan con el exterior, y por tanto no se ve afectado el uso de la clase. Por ejemplo, definimos una clase que trabaja con un vector que puede ser accedido por otros objetos, una de las acciones que realiza la clase sobre el vector es su ordenación, y para ello aplicamos el método de la burbuja, pero pasado el tiempo comprobamos que este proceso es demasiado lento, así que sustituimos este método de ordenación por el *quicksort*. Externamente los objetos de esta clase funcionan igual, se ofrece el vector ordenado aunque internamente el proceso de ordenación ha cambiado.
El ***polimorfismo*** es un concepto que se introduce en la POO y consiste en crear varias operaciones con el mismo nombre, pero que se aplican sobre datos de diferente tipo. El sistema en función del tipo de los parámetros que se le pasan a la operación usará una u otra. Por ejemplo, tenemos la operación de cálculo del área, pero no es lo mismo el área de un rectángulo, que la de un círculo o un triángulo, de manera que crearemos una clase rectángulo, una clase círculo y otra triángulo. En cada una de ellas definiremos la operación de cálculo del área, siendo después cuando se desee calcular el área de un objeto cuando se usará la operación adecuada dependiendo del tipo de objeto: rectángulo, círculo o triángulo.
Para terminar con los conceptos básicos relacionados con la POO, introduciremos el concepto de ***herencia***, que consiste en que a partir de una o más clases puedan derivarse otras clases, recibiendo de la / las clases padre sus propiedades. Pero pudiendo las clases derivadas modificar, añadir y eliminar las propiedades que sea necesarias para ajustarlas a las funcionalidades deseadas. La herencia es un mecanismo que permite crear clases con un carácter genérico y después adaptarlas a necesidades concretas. Por ejemplo, podemos definir una clase persona, y de ella derivar las clase empleado, cliente y proveedor. Como vemos las subclases compartirán la mayoría de propiedades definidas en la clase padre (persona), pero en cada subclase se realizarán los modificaciones pertinentes para conseguir una funcionalidad concreta.

Las Clases

Cuando vayamos a escribir una aplicación usando POO, fundamentalmente deberemos definir las Clases de Objetos que formarán parte de nuestra aplicación. Para identificar estas clases deberemos observar las estructuras de información que vamos a tratar, intentando modelar los objetos reales relacionados con el entorno para el que desarrollamos la aplicación. Además, de clases extraídas de la realidad, también pueden formar parte de nuestra aplicación otras clases necesarias para facilitar su desarrollo, como por ejemplo, una conexión a una página web, un mecanismo de comunicación entre aplicaciones, un botón o una ventana.
Una técnica que permite identificar las propiedades de una clase (datos y operaciones) consiste en ponernos en su lugar, convertirnos en un objeto de esa clase, y así poder enumerar sus características y ver qué podemos hacer. Por ejemplo, soy una factura y tengo: un número de factura, una fecha de factura, unas líneas de detalle, unos subtotales,

un IVA, un total, un descuento, etc. Y puedo: crearme, modificarme, imprimirme, copiarme y eliminarme.
Cuando definimos una clase, a sus operaciones les llamamos ***métodos*** y a sus datos ***atributos***. Por regla general, las clases como mínimo tendrán dos métodos, el ***constructor***, que tiene como misión crear e inicializar los diferentes objetos que se creen de la clase y el ***destructor*** que se encarga de eliminar los objetos cuando ya no son necesarios. Estos métodos, si no se escriben explícitamente, el entorno de programación los creará de forma automática para poder crear o eliminar los objetos de cada clase.
Al estudiar los métodos que formarán parte de una clase debemos tomar sólo los que estén presentes en todos los objetos de la clase, mientras que si hay métodos que afectan a grupos de objetos, éstos debemos añadirlos o modificarlos mediante la herencia creando subclases.
Como en el caso de lo métodos, al elegir los atributos que formarán parte de la clase, tomaremos aquellos que sean relevantes para la aplicación. Por ejemplo, si desarrollamos una aplicación para controlar el estado físico de personas, serán atributos de interés: su peso, altura, edad, pero no su religión o si le gusta jugar al ajedrez.
Cuando resolviendo un problema en el que se aplica la orientación a objetos preveemos que va a surgir una jerarquía de herencia, necesitaremos crear clases genéricas de partida en dicha jerarquía, que recogerán muchas similitudes, pero de las cuales, probablemente, no haya un solo ejemplar, por ejemplo la clase persona, sirve de base, pero en la aplicación sólo crearemos objetos cliente, proveedor o empleado, pero no persona. La abstracción, en POO se encuentra aplicada en el concepto de clase, con las denominadas clases abstractas.
Clases abstractas: Un tipo especial de clases que se caracterizan porque no pueden ser instanciadas.
Además, los lenguajes orientados a objetos cuentan con *Métodos abstractos*: Un tipo especial de método que no tiene cuerpo. Es decir, no tiene implementación. Los métodos abstractos sirven para definir la idea o la forma de invocación de un servicio, que será definido en clases derivadas (subclases).

Las Interfaces

Podemos definir una interfaz como una clase abstracta pura, pues no es más que una colección de cabeceras de métodos que son, implícitamente, públicos y abstractos. Son las clases las que, posteriormente, permiten dar soporte a una interfaz. Si una clase implementa una interfaz, se compromete a especificar el comportamiento de todos los métodos incluidos en la misma, salvo que la clase sea virtual o abstracta.
Las interfaces pueden contener atributos pero serán de forma también implícita static y final, es decir, constantes de clase.
Las interfaces pueden tener su propia jerarquía de herencia, por lo que se permite la herencia entre interfaces.
Una clase puede implementar más de una interfaz. Al ser tan sólo una colección de cabeceras de función presentan menos problemas de colisiones que la herencia múltiple. Las interfaces nos permiten definir comportamientos comunes entre clases dispares que no se relacionan evitando crear relaciones forzadas.

POO con PHP

Definición de Clases

class nom_clase {
/* Definición de atributos, será como las variables pero con un modificador de visibilidad : public, protected o private Por ejemplo*/
private $atr1; // Sólo accesible dentro del objeto
protected $atr2; // Accesible dentro del objeto y en objetos de clases derivadas
public $atr3= valor; // Accesible desde cualquier punto de su ámbito del objeto.
// Además tiene un valor inicia
|
/* Definición de métodos, será como las funciones per con un modificador de visibilidad: public, protected o private Por ejemplo*/
private function fun1(...){... }
protected function fun2(...){...}
public function fun3(...){...}
} // class nom_clase
Por defecto los métodos si no se indica son públicos, mientras que para los atributos debemos declararlo explícitamente.
Las clases disponen de dos métodos especiales, el constructor que se ejecuta al crear objetos y el destructor que se ejecuta cuando se debe eliminar un objeto que ya no es referenciado en la aplicación.
El constructor: *__construct*(...){...} en este método se ejecutan las operaciones de inicialización necesarias para crear un objeto.
El destructor: *__destruct*(){...} este método se ejecuta cuando se elimina un objeto que ya no tiene referencias en la aplicación y antes de eliminarlo se debe liberar recursos que tenga asignados este objeto, como por ejemplo: cerrar ficheros o conexiones o bien eliminar algún tipo de señal de bloqueo en el acceso algún recurso, etc.
En muchos casos es interesante definir un método llamado __toString() que permita convertir en una cadena de caracteres el objeto, pondremos aquello que nos interese mostrar del objeto. Este método se usa automáticamente cuando nuestro objeto está en un contexto en el que se espera una cadena de caracteres, como por ejemplo en una sentencia *echo $ojb;*
Con frecuencia necesitamos dentro de un objeto hacer referencia a él mismo y se consigue mediante la variable *$this.* De forma que podemos acceder a métodos o atributos del propio objeto mediante el uso de esta variable, por ejemplo:
$res = $this->fun1(...);
$this->atr2 = 1234;
Clase Persona:

```
<?php
	//Creamos un array constante como enumerado de los valores de Sexo
	define('SEXO', array('HOMBRE', 'MUJER'));
	//Creamos un array como enumerado de los valores de Estado Civil
	define('ESTADO_CIVIL', array('SOLTERO','CASADO'));

	class Persona {
```

```
// Campos de la clase se declaran privados para limitar el acceso
private $nombre;
private $apellidos;
private $direccion;
private $poblacion;
private $cp;
private $nacionalidad;
private $sexo;
private $dni;
private $estado_civil;
private $telefono;
private $fecha_nacimiento;
// Metodos son publicos para dar acceso
// Constructor toma los parametros para crear los objetos persona
public function __construct($n, $a, $d, $p, $c, $na, $s, $dn, $es, $t, $f){
       $this->nombre = $n;
       $this->apellidos = $a;
       $this->direccion = $d;
       $this->poblacion = $p;
       $this->cp = $c;
       $this->nacionalidad = $na;
       $this->dni = $dn;
       $s = strtoupper($s);
       switch($s) {
       case 'HOMBRE':
       case 'VARON': $this->sexo = SEXO[0];
              break;
       case 'MUJER': $this->sexo = SEXO[1];
              break;
       default : $this->sexo = NULL;
       }
       $es = strtoupper($es);
       switch($es) {
       case 'SOLTERO': $this->estado_civil = ESTADO_CIVIL[0];
              break;
       case 'CASADO': $this->estado_civil = ESTADO_CIVIL[1];
              break;
       default : $this->estado_civil = NULL;
       }
       $this->telefono = $t;
       $this->fecha_nacimiento = date_create_from_format('d-m-Y', $f);
} // Constructor de persona
public function __toString(){
 return "Nombre: " . $this->nombre . "  Apellidos:  " . $this->apellidos .
       "<br>";
}
public function mostraPersona(){//Añadir los campos que sean necesarios
       echo '<table border=1>';
       echo '<tr>';
       echo '<th>Nombre</th>';
       echo '<th>Apellidos</th>';
       echo '</tr>';
       echo '<tr>';
       echo '<td>'. $this->nombre. '</td>';
       echo '<td>' . $this->apellidos .'</td>';
       echo '</tr>';
       echo '</table>';
```

```
}
// Métodos get para obtener los atributos, como el siguiente
public function get_fecha_nacimiento(){return $this->fecha_nacimiento; }
public function get_telefono(){return $this->telefono; }
//...
// Métodos set para modificar los atributos, como el siguiente
public function set_telefono($tel){ this->telefono = $tel; }
//...
// Métodos de interés para la clase
public function edad(){
       $year = date_format($this->fecha_nacimiento,"Y");
       $hoy = date('Y');
       return $hoy - $year;
}
public function es_ciudadano(){
       if("ESPANOLA"== strtoupper($this->nacionalidad)){
              $ciudadano = TRUE;
       }
       else{
              $ciudadano = FALSE;
       }
       return $ciudadano;
}
} // Persona
//Instanciamos un objeto persona y la usamos
$p1 = new persona("Toni", "Boronat","c/El Canal,8","Vinaròs","12500",
"Espanola","11111111A","Hombre","Soltero", "666666666","22-05-1965");
echo $p1;
$p1->mostraPersona(); // Llama a __toString() automáticamente
if($p1->es_ciudadano()){
       echo '<br>Es ciudadano';
}
echo "<br>Fecha nacimiento: " .
      date_format($p1->get_fecha_nacimiento(),"d-m-Y");
echo "<br>Edad: " . $p1->edad();
echo "<br>Tel antes: " .$p1->get_telefono();
$p1->set_telefono("555555555");
echo "<br>Tel después: " .$p1->get_telefono();
?>
```

Herencia

El concepto de herencia tiene como misión crear nuevas clases a partir de otras, de forma que la nueva clase tiene las propiedades de la clase base, que pueden ser redefinidas, si es necesario y además le añadiremos las características propias de la nueva clase.
La herencia en PHP se define con el modificador de clase *extends.*

```
class clase_hija extends clase_base {
      // Nuevos atributos
      …
      //Nuevos métodos
      ...
}
```

Si es necesario dentro del constructor de la clase derivada debemos llamar al constructor de la clase base para inicializar los atributos heredados mediante:
parent::__costruct(parámetros);

Por ejemplo creamos la clase Cliente como derivada de la clase Persona:

```
<?php
      include_once 'persona.php'; //Contiene la clase Persona
      class Cliente extends Persona {
      // Añadimos nuevos atributos propios de cliente
       private $fecha_ult_factura;
       private $total_facturado;
       private $forma_pago;

      // Constructor de Cliente
      public function __construct ($n, $a, $d, $p, $c, $na, $s, $dn, $es, $t,
            $f, $fuf, $tf, $fp) {
            //Llama al constructor de la clase base
            parent::__construct($n, $a, $d, $p, $c, $na, $s, $dn, $es, $t,$f);
            // Asigna los atributos propios de la clase derivada
            $this->fecha_ult_factura = date_create_from_format('d-m-Y', $fuf);
            $this->total_facturado = $tf;
            $this->forma_pago = $fp;
      } // __construct()
      public function __toString(){ //Usa la de la clase base --> Persona
            return parent::__toString() . " Total facturado: " .
                  $this->total_facturado . "<br>";
      }
      // Métodos get para obtener los atributos, como el siguiente
      public function get_fecha_ult_factura(){
            return date_format($this->fecha_ult_factura,"d-m-Y");
      }
      //...
      // Métodos set para modificar los atributos, como el siguiente
      public function set_fecha_ult_factura($fuf2){
            $this->fecha_ult_factura = date_create_from_format('d-m-Y', $fuf2);
      }
      //...
      public function mostrar_cliente(){
            parent::mostraPersona(); //Llama a mostrarPersona() de la clase base
            echo '<table border=1>';
```

```
            echo '<tr>';
            echo '<th>Fecha Factura</th>';
            echo '<th>Total Factuado</th>';
            echo '<th>Forma Pago</th>';
            echo '</tr>';
            echo '<tr>';
            echo '<td>'. date_format($this->fecha_ult_factura, "d-m-Y") . '</td>';
            echo '<td>' . $this->total_facturado .'</td>';
            echo '<td>' . $this->forma_pago .'</td>';
            echo '</tr>';
            echo '</table>';
      }
      }// Cliente
      //Instanciamos un objeto Cliente
      $c1 = new cliente("Toni", "Boronat","c/El Canal,8","Vinaròs","12500",
            "Espanola","11111111A","Hombre","Soltero","666666666","22-05-1965",
            "05-11-2015",3000,"Contado");
      echo $c1; // Usa __toString()
      $c1->mostrar_cliente();
      echo "<br>Última factura antes: " . $c1->get_fecha_ult_factura();
      $c1->set_fecha_ult_factura("25-02-2017");
      echo "<br>Última factura después: " . $c1->get_fecha_ult_factura();
?>
```

Clases y Métodos Abstractos

Como se ha comentado las clases y métodos abstractos son aquellos que no tienen cuerpo para ser instanciadas, de forma que las clases que las usen deberán añadir las instrucciones que deben implementar.
PHP los define con el modificador *abstract*:

```
abstract class nom_clase {
      // Atributos de la clase abstracta

      // Métodos de la clase abstracta
      abstract public function nom_metodo_abs(parámetros); // No hay
cuerpo de la función
}
```

La clase derivada debe implementar los métodos abstractos:

```
class nom_derivada extends nom_clase {
      // Nuevos atributos de la clase derivada
      ...
      // Implementación de los métodos abstractos
      public function nom_metodo_abs(parámetros){
            // Instrucciones
      }
      // Nuevos métodos
      ...
}
```

Ejemplo de la clase Persona y Cliente pero con clases abstractas, de forma que la clase Persona sea abstracta, no necesitamos instancias de ella, sólo derivadas como Cliente o Empleado.

```php
<?php //Clase persona abstracta
	//Creamos una clase como enumerado de los avalores de Sexo
	define('SEXO', array('HOMBRE', 'MUJER'));
	//Creamos una clase como enumerado de los valores de Estado Civil
	define('ESTADO_CIVIL', array('SOLTERO','CASADO'));

	abstract class Persona_abstract {
	//Campos de la clase se declaran protected para permitir el
	// acceso en las derivadas
	protected $nombre;
	protected $apellidos;
	protected $direccion;
	protected $poblacion;
	protected $cp;
	protected $nacionalidad;
	protected $sexo;
	protected $dni;
	protected $estado_civil;
	protected $telefono;
	protected $fecha_nacimiento;
	// Metodos son publicos para dar acceso

	// Constructor toma los parametros para crear los objetos persona
	public function __construct($n, $a, $d, $p, $c, $na, $s, $dn, $es, $t, $f){
		$this->nombre = $n;
		$this->apellidos = $a;
		$this->direccion = $d;
		$this->poblacion = $p;
		$this->cp = $c;
		$this->nacionalidad = $na;
		$this->dni = $dn;
		$s = strtoupper($s);
		switch($s) {
		case 'HOMBRE':
		case 'VARON': $this->sexo = SEXO[0];
			break;
		case 'MUJER': $this->sexo = SEXO[1];
			break;
		default : $this->sexo = NULL;
		}
		$es = strtoupper($es);
		switch($es) {
		case 'SOLTERO': $this->estado_civil = ESTADO_CIVIL[0];
			break;
		case 'CASADO': $this->estado_civil = ESTADO_CIVIL[1];
			break;
		default : $this->estado_civil = NULL;
		}
		$this->telefono = $t;
		$this->fecha_nacimiento = date_create_from_format('d-m-Y', $f);
	} // Constructor de persona

	public function __toString(){
		return "Nombre: " . $this->nombre . "  Apellidos:  " . $this-
>apellidos . "<br>";
	}
```

```
	public function mostraPersona(){ //Puede añadir los campos que sea
necesarios
		echo '<table border=1>';
		echo '<tr>';
		echo '<th>Nombre</th>';
		echo '<th>Apellidos</th>';
		echo '</tr>';
		echo '<tr>';
		echo '<td>'. $this->nombre. '</td>';
		echo '<td>' . $this->apellidos .'</td>';
		echo '</tr>';
		echo '</table>';
	}
	// Métodos get para obtener los atributos, como el siguiente
	public function get_fecha_nacimiento(){
		return $this->fecha_nacimiento;
	}
	public function get_telefono(){
		return $this->telefono;
	}
	//...
	// Métodos set para modificar los atributos, como el siguiente
	public function set_telefono($tel){
		$this->telefono = $tel;
	}
	//...
	// Métodos abstractos que en cada clase derivada tendrán un código
	abstract public function antiguedad();
	abstract public function es_de();
	} // Persona
?>
```

Clase derivada Cliente:

```
<?php
	include_once 'persona_abstract.php'; //Contiene la clase Persona abstracta
	class Cliente_de_abstr extends Persona_abstract {
	// Añadimos nuevos atributos propios de cliente
	 private $fecha_ult_factura;
	 private $total_facturado;
	 private $forma_pago;

	// Constructor de Cliente
	public function __construct ($n, $a, $d, $p, $c, $na, $s, $dn, $es, $t,
		$f, $fuf, $tf, $fp) {
		//Llama al constructor de la clase base
		parent::__construct($n, $a, $d, $p, $c, $na, $s, $dn, $es, $t,$f);
		// Asigna los atributos propios de la clase derivada
		$this->fecha_ult_factura = date_create_from_format('d-m-Y', $fuf);
		$this->total_facturado = $tf;
		$this->forma_pago = $fp;
	} // __construct()
	public function __toString(){ //Usa la de la clase base --> Persona
		return parent::__toString() . " Total facturado: " .
		 $this->total_facturado . "<br>";
	}
	// Métodos get para obtener los atributos, como el siguiente
	public function get_fecha_ult_factura(){
		return date_format($this->fecha_ult_factura,"d-m-Y");
```

```
	}
	//...
	// Métodos set para modificar los atributos, como el siguiente
	public function set_fecha_ult_factura($fuf2){
		$this->fecha_ult_factura = date_create_from_format('d-m-Y', $fuf2);
	}
	//...
	public function mostrar_cliente(){
		parent::mostraPersona();
		echo '<table border=1>';
		echo '<tr>';
		echo '<th>Fecha Factura</th>';
		echo '<th>Total Factuado</th>';
		echo '<th>Forma Pago</th>';
		echo '</tr>';
		echo '<tr>';
		echo '<td>'. date_format($this->fecha_ult_factura, "d-m-Y") . '</td>';
		echo '<td>' . $this->total_facturado .'</td>';
		echo '<td>' . $this->forma_pago .'</td>';
		echo '</tr>';
		echo '</table>';
	}
	public function antiguedad() { //De la última factura
		$ult_fact = date_format($this->fecha_ult_factura,"d-m-Y");
		$hoy = new DateTime('now');
		$interval = $this->fecha_ult_factura->diff($hoy);
		return $interval->format("%a días");
	}
	public function es_de(){ //Cliente local?
		if("VINAROS"== strtoupper($this->poblacion)){
			$local = TRUE;
		}
		else{
			$local = FALSE;
		}
		return $local;
	}
	}// Cliente
	//Instanciamos un objeto Cliente
	$c1	=	new	Cliente_de_abstr("Toni",	"Boronat","c/El
Canal,8","Vinaros","12500",
		"Espanola","11111111A","Hombre","Soltero","666666666","22-05-1965",
		"05-11-2015",3000,"Contado");
	echo $c1; // Usa __toString()
	$c1->mostrar_cliente();
	echo "<br> Tiempo sin facturarle: " . $c1->antiguedad();
	if($c1->es_de()){
	echo "<br>Es cliente con portes locales";
	}
	else {
	echo "<br>Es cliente con portes nacionales";
	}
?>
```

Clases y Métodos Final

El modificador *final* en una clase indica que no se pueden crear subclases a partir de ella y aplicado sobre un método impide que sea reescrito en las clases derivadas.

```
final class nom_clase {
...
}
class otra_clase {
…
      final public function nom_metodo () {
      ...
      }
}
```

Interfaces

Las interfaces en PHP consisten en la definición de un conjunto de métodos sin cuerpo que deberán escribir las clases que las implementen. El objetivo de las interfaces es presentar funcionalidades que puedan emplear las clases, de forma que un clase puede implementar varias interfaces.
En las interfaces los métodos son públicos y si es necesario, en ellas también se puede definir constantes, pero no atributos variables.

Ejemplo a partir de Persona pero con interface, la implementación es la misma que en el ejemplo de Persona, sólo mostraremos los cambios:

```
interface antiguedad {
              function edad();
              function felicitar();
       }
class Persona_iface implements antiguedad {
…
       public function edad(){
              $year = date_format($this->fecha_nacimiento,"Y");
              $hoy = date('Y');
              return $hoy - $year;
       }
       public function felicitar(){
              $dia = date('d');
              $mes = date('m');
              $dia_cumple = date_format($this->fecha_nacimiento,"d");
              $mes_cumple = date_format($this->fecha_nacimiento,"m");
              $res = false; // Inicial no es el aniversario
              if(($mes == $mes_cumple) && ($dia == $dia_cumple)){
                     $res = true;
              }
              return $res;
       }
       } // Persona
```

Métodos y atributos estáticos

En ocasiones es interesante disponer de métodos y atributos que pertenezcan a la clase y no a cada objeto, de forma que los métodos y atributos puedan ser accedidos incluso sin tener ningún objeto, directamente referenciando a la clase:
Para definir este tipo de elementos anteponemos en su declaración el modificador *static*.

```
class nom_clase {
…
      static $atr_stc = valor_inicial;
…
      static public function met_stc(){ … }
}
```

Para usarlos se referencian a partir de la clase con:

```
nom_clase::$atr_stc
nom_clase::met_stc()
```

Ejemplo a la clase persona le hemos añadido un método y un atributo estático:

```
<?php
//Creamos una clase como enumerado de los avalores de Sexo
      define('SEXO', array('HOMBRE', 'MUJER'));
      //Creamos una clase como enumerado de los valores de Estado Civil
      define('ESTADO_CIVIL', array('SOLTERO','CASADO'));

      class Persona {
      // Campos de la clase se declaran privados para limitar el acceso
      private $nombre;
      private $apellidos;
      private $direccion;
      private $poblacion;
      private $cp;
      private $nacionalidad;
      private $sexo;
      private $dni;
      private $estado_civil;
      private $telefono;
      private $fecha_nacimiento;
      //Atributo estático, número de objetos persona
      static protected $num_personas = 0;
      // Metodos son publicos para dar acceso

      // Constructor toma los parametros para crear los objetos persona
      public function __construct($n, $a, $d, $p, $c, $na, $s, $dn, $es, $t, $f){
            $this->nombre = $n;
            $this->apellidos = $a;
            $this->direccion = $d;
            $this->poblacion = $p;
            $this->cp = $c;
            $this->nacionalidad = $na;
            $this->dni = $dn;
            $s = strtoupper($s);
            switch($s) {
            case 'HOMBRE':
            case 'VARON': $this->sexo = SEXO[0];
```

```
                break;
            case 'MUJER': $this->sexo = SEXO[1];
                break;
            default : $this->sexo = NULL;
            }
            $es = strtoupper($es);
            switch($es) {
            case 'SOLTERO': $this->estado_civil = ESTADO_CIVIL[0];
                break;
            case 'CASADO': $this->estado_civil = ESTADO_CIVIL[1];
                break;
            default : $this->estado_civil = NULL;
            }
            $this->telefono = $t;
            $this->fecha_nacimiento = date_create_from_format('d-m-Y', $f);
            // Incrementamos la variable estática
            Persona::$num_personas++;
      } // Constructor de persona
      //El destructor decrementa la variable de num personas
      public function __destruct(){
            Persona::$num_personas--;
      }
      public function get_num_personas(){
            return Persona::$num_personas;
      }

      public function __toString(){
         return "Nombre: ". $this->nombre . " Apellidos:  " . $this->apellidos .
"<br>";
      }
      public function mostraPersona(){//Se puede añadir los campos que sea
necesarios
            echo '<table border=1>';
            echo '<tr>';
            echo '<th>Nombre</th>';
            echo '<th>Apellidos</th>';
            echo '</tr>';
            echo '<tr>';
            echo '<td>'. $this->nombre. '</td>';
            echo '<td>' . $this->apellidos .'</td>';
            echo '</tr>';
            echo '</table>';
      }
      // Métodos get para obtener los atributos, como el siguiente
      public function get_fecha_nacimiento(){
            return $this->fecha_nacimiento;
      }
      public function get_telefono(){
            return $this->telefono;
      }
      //...
      // Métodos set para modificar los atributos, como el siguiente
      public function set_telefono($tel){
            $this->telefono = $tel;
      }
      //...
      // Métodos de interés para la clase
```

```
	public function edad(){
		$year = date_format($this->fecha_nacimiento,"Y");
		$hoy = date('Y');
		return $hoy - $year;
	}
	public function es_ciudadano(){
		if("ESPANOLA"== strtoupper($this->nacionalidad)){
			$ciudadano = TRUE;
		}
		else{
			$ciudadano = FALSE;
		}
		return $ciudadano;
	}
	static public function personas_iguales($obj_p1, $obj_p2) {
		$res = false;
		if($obj_p1->nombre == $obj_p2->nombre){
			if($obj_p1->apellidos == $obj_p2->apellidos){
				$res = true;
			}
		}
		return $res;
	}
	} // Persona
?>
```

Uso de las propiedades estáticas:

```
echo "<br> Número de personas creadas: " . $p1->get_num_personas();
if(Persona::personas_iguales($p1, $p2)){
		echo "<br>Són personas iguales.";
}
else{
		echo "<br>Són personas diferentes.";
}
```

Modificadores de acceso a métodos y atributos

Como hemos visto, para acceder a un método de la clase padre usamos el prefijo `parent::método` pero si deseamos acceder a un método de una clase de la que desciende, pero no es la inmediatamente superior, usaremos la misma sintaxis que en los métodos y propiedades estáticas, `Nombre_Clase::método`.

Y finalmente, si deseamos acceder a un método o a un atributo estático de la propia clase se usa el prefijo `self::método` .

Serialización de Objetos

Los objetos tal y como los usamos en las aplicaciones no pueden ser almacenados o transferidos en red, así que para facilitar estas tareas se les aplica una transformación denominada *serialización,* que básicamente consiste en convertir los objetos en cadenas de caracteres que soportarán las operaciones comentadas, además los objetos serializados podrán ser reconstruidos en los objetos originales cuando sea necesario.
PHP ofrece dos funciones para serializar y reconstruir objetos:

- string **serialize** (mixed $value);
- mixed **unserialize** (string $str [, array $options]);

Cabe destacar que un objeto serializado es una cadena binaria y si debe ser almacenado en una base de datos el campo debe ser de tipo BLOB en lugar de CHAR o TEXT.
Con *unserialize()* reconstruimos el objeto a partir de la cadena obtenida aplicando *serialize()*.
Ejemplo de serialización para almacenar en una BD:

```
<?php
/* Ejemplo de serialización de objetos, almacenamiento en un BD y su
recuperación */
include 'persona.php';
include 'funcion_conexion_bd.php';
DEFINE ("SERVIDOR", "localhost");
DEFINE ("USER", "root");
DEFINE ("PASSWD", "");
DEFINE ("BASE_DATOS", "personas");

$con_bd = conexion_bd(SERVIDOR, USER, PASSWD, BASE_DATOS);
$p1 = new persona("Toni", "Boronat","c/El Canal,8", "Vinaròs",
"12500", ,"22-05-1965");

// Serializa el obj para guardarlo en la BD
$cad = serialize($p1);

$sql = "INSERT INTO persona (apellidos, obj_persona) VALUES
('Boronat', '" . $cad ."')";
if($res = mysqli_query($con_bd, $sql)) {
      echo "Guardado en la BD <br>";
}
else{
      echo "Error en la consulta: ". mysqli_error($con_bd)."<br>";
}
//Extrae el registro con el objeto de la BD
$sql = "SELECT * FROM persona WHERE apellidos = 'Boronat'";
if($res = mysqli_query($con_bd, $sql)) {
      $res_array = mysqli_fetch_all($res, MYSQLI_ASSOC);
      //Operación inversa a la serialización para recuperar el obj
```

```
      $p2 = unserialize($res_array[0]["obj_persona"]);
      $p2->mostraPersona(); // Uso del objeto
}
else{
      echo "Error en la consulta: ". mysqli_error($con_bd)."<br>";
}
mysqli_close($con_bd);
?>
```

Manejo de Errores y Excepciones

PHP a partir de la versión 5 incluye el sistema de gestión de errores basado en las estructura *try-catch-finally* similar al que tienen otros lenguajes orientados a objetos como C++ o Java.

Las sentencias *try-catch* las podemos incluir en cualquier parte de nuestro código donde se pueda generar algún tipo de error o excepción, de forma que en la parte *try* se incluyen las instrucciones susceptibles de provocar un fallo y en los apartados *catch* se incluyen las instrucciones necesarias para tratar el fallo en caso de producirse. Apartados *catch* puede haber varios de forma que cada uno se encargará de atender una excepción concreta.

Las instrucciones incluidas en los apartados *finally* se ejecutan siempre, independientemente de si se ha producido alguna excepción o no, esta parte es opcional.

Con las sentencias *throw* podemos lanzar excepciones cuando detectemos una situación de error que deba ser tratada, por ejemplo si intentamos abrir un fichero y no podemos, se puede lanzar una excepción mediante *throw* para que sea tratada en un apartado *catch*.

Podemos crear nuestro propio repertorio de excepciones, heredando de la clase *Exception*. De hecho, es recomendable dado que la sentencia *throw* sólo puede invocarse acompañada de un objeto de la clase *Exception*, o una clase derivada de ella. Este sistema es mucho más potente y flexible, ya que en función del tipo de error, podemos optar por abortar la ejecución, o intentar reconducir la situación para que el programa siga funcionando.

La sintaxis de estas estructuras es:

```
try {
    // código susceptible de generar errores
    ...
}
catch (Exception $e) {
    // código destinado a procesar los errores
    ...
}
finally { // El uso de finally es opcional
    // código ejecutado en todos los casos
    ...
}
```

Como vemos cuando se genera una excepción se crea un objeto de la clase *Exception* que tiene los métodos:

getMessage(), que nos muestra la información asociada a la excepción producida.
getCode(), que nos proporciona el código de esta excepción.
Para lanzar una excepción mediante throw, usaremos:
`throw new Exception("mensaje error producido", [código error]);`
Dentro de los apartados *catch* podemos usar los métodos *getMessage* y *getCode* para mostrar el error producido.

```
try{
      $p1 = new persona("Toni", "Boronat","c/El Canal,8","Vinaròs","12500",
            "Espanola","11111111A","Hombre","Soltero","666666666","22-05-1965");
      ...
      $p2 = new persona("Pablo", "Boronat","c/El Canal,8","Vinaròs","12500",
            "Espanola","11111111A","Hombre","Soltero","666666666","30-04-1965");
      ...
      }
   } // Try
  catch (Exception $e) {
            echo "<br>Excepción: " . $e->getMessage();
   }
   catch(Error $e){
            echo "<br>Error: " . $e->getMessage();
   }
   finally { //Esta parte se ejecuta siempre
       echo "<br>El número total de personas es: " . Persona::get_num_personas();
   }
```

Debemos tener en cuenta que cuando se produce una excepción en un programa y esta no es tratada, se interrumpe la ejecución y muestra un mensaje de excepción no tratada (Fatal error: Uncaught Exception: ...), por tanto, debemos añadir los apartados catch necesarios y además uno general que capture el resto de posibles excepciones que se pueda producir y que no vayamos a tratar de forma individualizada.

También puede ser interesante crear nuestras propias clases de excepciones como derivadas de Exception, de manera que se ajusten mejor al tratamiento de las excepciones que se pueden producir en nuestras aplicaciones.

```
class ExcepcionFecha extends Exception {
      public function registrarError() {
      // Registramos en el LOG del servidor web
            if($this->getCode()== 1){
                  error_log($this->getMessage(),0);
            }
      }
}
```

Para lanzar esta excepción:
`throw new ExcepcionFecha('Formato o fecha incorrecta',$num_cod_err);`
En el catch lo definiremos como:

```
catch(ExcepcionFecha $e) {
      echo $e->registrarError();
}
```

Namespaces

Los espacios de nombres o *namespace* tienen como objetivo facilitar el uso de diferentes conjuntos de clases en las aplicaciones de forma que se eviten las colisiones en los nombres de clases, que incorporadas en diferentes bibliotecas reciben el mismo nombre y que al usarse en la aplicación provoca un error. Además, los *namespace* acortan las rutas largas que identifican una clase al ser instanciada, ya que permiten crear alias a estas clases y podemos crear objetos usando su alias.
La definición de los *namespace* se realiza con esta palabra y debe ponerse al principio del fichero de código PHP. Sólo puede estar precedida por sentencias *declare,* cualquier otra deberá ponerse después de *namespace*.

```
<?php
      namespace individuos; // Crea un namespace para esta clase Persona

      if (!defined('SEXO')){
            define('SEXO', array('HOMBRE', 'MUJER'));
      }
      define('ESTADO_CIVIL', array('SOLTERO','CASADO'));

      class Persona {
            protected $nombre;
            protected $apellidos;
            ...
```

La definición de un *namespace* se puede realizar en varios ficheros y también se puede definir varios *namespace* en un solo fichero, aunque esto último no es recomendable ya que puede generar confusión al incluir estos ficheros.
El símbolo \ permite crear una jerarquía dentro de un *namespace* de manera que los diferentes niveles vendrán separados por \. Así podemos distribuir mejor el contenido del *namespace* en caso que ofrezca una serie de clases con características diferenciadas.
Por ejemplo en un fichero tenemos este *namespace*

```
<?php
      namespace sociedad\individuos;
```

y en otro fichero otro *namespace* de la misma jerarquía

```
<?php
      namespace sociedad\humanos;
```

En PHP hay un *namespace* por defecto, global, que contiene todo lo que no está definido dentro de namespaces.
Además, podemos conocer el *namespace* en el que se encuentra la aplicación usando la constante __NAMESPACE__ y que podemos usar para crear rutas a partir del *namespace* actual. El ejemplo mostrará el *namespace* al que pertenece la clase.

```
public function __toString(){
  return __NAMESPACE__ . ": Nombre: " . $this->nombre . " Apellidos: " . $this->apellidos .
"<br>";
}
```

Si el *namespace* es el global el valor de esta constante será la cadena vacía.

Las rutas expresadas en los *namespace* pueden ser absolutas, empiezan con la \ o relativas que empiezan por un nombre y parten del *namespace* actual.
En las aplicaciones, para no tener que poner las rutas de los *namespace*, se dispone de la sentencia *use ruta\del\namespace\clase* que permite establecer las rutas a las clases (o funciones y constantes) de forma que sólo pondremos el nombre a usar.

```
 use individuos\Persona;
      ...
 class Cliente extends Persona {
      ...
$p = new Persona(
      ...
```

La sentencia *use* permite crear un alias de la ruta y usarlo en el acceso al recurso expresado en el *namespace. use ruta\del\namespace\clase as alias_clase*

```
 use sociedad\individuos\Persona as PersonaSocial;
 use sociedad\humanos\Persona as PersonaBiologica;
      ...
$ps = new PersonaSocial(
      ...
$pb = new PersonaBiologica(
      ...
```

Ejemplo de *namespace*:
persona_biologica.php:

```
<?php
namespace sociedad\humanos;
//Creamos una clase como enumerado de los avalores de Sexo
 if (!defined('SEXO')){
       define('SEXO', array('HOMBRE', 'MUJER'));
 }
 class Persona {
// Campos de clase se declaran protected para permitir el acceso en las derivadas
  protected $peso;
  protected $estatura;
  protected $color;
  protected $nacionalidad;
  protected $sexo;
  protected $dni;
  protected $fecha_nacimiento;
  // Metodos son publicos para dar acceso

  // Constructor toma los parametros para crear los objetos persona
  public function __construct($p, $e, $c, $na, $s, $dn, $f){
    $this->peso = $p;
    $this->estatura = $e;
    $this->color = $c;
    $this->nacionalidad = $na;
    $this->dni = $dn;
    $s = strtoupper($s);
    switch($s) {
      case 'HOMBRE':
      case 'VARON': $this->sexo = SEXO[0];
          break;
     case 'MUJER': $this->sexo = SEXO[1];
```

```
          break;
       default : $this->sexo = NULL;
      }
      $this->fecha_nacimiento = date_create_from_format('d-m-Y', $f);
   } // Constructor de persona

   public function __toString(){
     return __NAMESPACE__ . ":  DNI: " . $this->dni . "  Peso:  " . $this->peso . "
Estatura: ". $this->estatura . "<br>";
   }
   public function mostraPersona(){ //Se puede añadir los campos que sea necesarios
     echo '<table border=1>';
     echo '<tr>';
     echo '<th>DNI</th>';
     echo '<th>Peso</th>';
     echo '<th>Estatura</th>';
     echo '</tr>';
     echo '<tr>';
     echo '<td>'. $this->dni. '</td>';
     echo '<td>' . $this->peso .'</td>';
     echo '<td>' . $this->estatura .'</td>';
     echo '</tr>';
     echo '</table>';
   }
   // Métodos get para obtener los atributos, como el siguiente
   public function getFechaNacimiento(){
     return $this->fecha_nacimiento;
   }
   //... RESTO DE MÉTODOS
   // Métodos set para modificar los atributos, como el siguiente
   public function setPeso($p){
     $this->peso = $p;
   }
   //...RESTO DE MÉTODOS
   // Métodos abstractos que en cada clase derivada tendrán un código
   public function edad(){
     $year = date_format($this->fecha_nacimiento,"Y");
     $hoy = date('Y');
     return $hoy - $year;
   }
 }
?>
```

fichero persona_social.php:

```
<?php
namespace sociedad\individuos; // Crea un namespace para esta clase Persona
//Creamos enumerado de los avalores de Sexo
 if (!defined('SEXO')){
   define('SEXO', array('HOMBRE', 'MUJER'));
 }
 //Creamos enumerado de los valores de Estado Civil
 define('ESTADO_CIVIL', array('SOLTERO','CASADO'));

 class Persona {
  // Campos de clase se declaran protected para permitir el acceso en las derivadas
  protected $nombre;
  protected $apellidos;
```

```
  protected $direccion;
  protected $poblacion;
  protected $cp;
  protected $nacionalidad;
  protected $sexo;
  protected $dni;
  protected $estado_civil;
  protected $telefono;
  protected $fecha_nacimiento;
  // Metodos son publicos para dar acceso

  // Constructor toma los parametros para crear los objetos persona
  public function __construct($n, $a, $d, $p, $c, $na, $s, $dn, $es, $t, $f){
    $this->nombre = $n;
    $this->apellidos = $a;
    $this->direccion = $d;
    $this->poblacion = $p;
    $this->cp = $c;
    $this->nacionalidad = $na;
    $this->dni = $dn;
    $s = strtoupper($s);
    switch($s) {
      case 'HOMBRE':
      case 'VARON': $this->sexo = SEXO[0];
          break;
      case 'MUJER': $this->sexo = SEXO[1];
          break;
      default : $this->sexo = NULL;
    }
    $es = strtoupper($es);
    switch($es) {
      case 'SOLTERO': $this->estado_civil = ESTADO_CIVIL[0];
          break;
      case 'CASADO': $this->estado_civil = ESTADO_CIVIL[1];
          break;
      default : $this->estado_civil = NULL;
    }
    $this->telefono = $t;
    $this->fecha_nacimiento = date_create_from_format('d-m-Y', $f);
  } // Constructor de persona

  public function __toString(){
    return  __NAMESPACE__ . ":  Nombre: " . $this->nombre . "  Apellidos:  " .
$this->apellidos . "<br>";
  }
  public function mostraPersona(){ //Se puede añadir los campos que sea necesarios
    echo '<table border=1>';
    echo '<tr>';
    echo '<th>Nombre</th>';
    echo '<th>Apellidos</th>';
    echo '</tr>';
    echo '<tr>';
    echo '<td>'. $this->nombre. '</td>';
    echo '<td>' . $this->apellidos .'</td>';
    echo '</tr>';
    echo '</table>';
  }
```

```
  // Métodos get para obtener los atributos, como el siguiente
  public function getFechaNacimiento(){
    return $this->fecha_nacimiento;
  }
  public function getTelefono(){
    return $this->telefono;
  }
  //...RESTO DE MÉTODOS
  // Métodos set para modificar los atributos, como el siguiente
  public function setTelefono($tel){
    $this->telefono = $tel;
  }
 } // Persona
?>
```

index.php:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title></title>
    </head>
    <body>
     <?php
       include 'persona_social.php'; // namespace individuos
       include 'persona_biologia.php'; // namespace humanos

       use sociedad\individuos\Persona as PersonaSocial; // Alias
       use sociedad\humanos\Persona as PersonaBiologica; // Alias

      try{
       //Instanciamos un objeto Persona --> individuo con su alias
       $ps = new PersonaSocial("Rosa", "Garriga","c/Riu,12","Vinaròs","12500",
                 "Espanola","22222222B","Mujer","Casado","699999999","02-09-1983");
       echo "Persona social:<br>" . $ps . "<br>";
       $ps->mostraPersona();
       echo "<br><br>";
       //Instanciamos una objeto Persona --> humanos con su alias
       $pb = new PersonaBiologica(70, 175,"blanco","española","Mujer",
                "22222222B","02-09-1983");
       echo "Persona biológica:<br>" . $pb . "<br>";
       $pb->mostraPersona(); ;
       echo "<br><br>";
      } // Try
      catch (Exception $e) {
          echo "<br>Excepción: " . $e->getMessage();
      }
      catch(Error $e){
          echo "<br>Error: " . $e->getMessage();
      }

     ?>
    </body>
</html>
```