ORIENTACIÓN A OBJETOS EN PHP5

Natalia Díaz Rodríguez
Victoria López Morales

3º A Ingeniería Informática
Curso 2006-2007
PROGRAMACIÓN DIRIGIDA A OBJETOS

Historia e Introducción a PHP 5

- **PHP** es un lenguaje de programación interpretado usado generalmente para la creación de contenido dinámico para sitios web y aplicaciones para servidores. PHP es un acrónimo que significa "**PHP Hypertext Pre-processor**" (inicialmente **PHP Tools**, o, ***P*ersonal *H*ome *P*age Tools**).
- Con las primeras 2 versiones de PHP, PHP 3 y PHP 4, se había conseguido una plataforma potente y estable, haciendo posible que PHP sea el **lenguaje más utilizado en la web para la realización de páginas** avanzadas.
- Sin embargo, todavía existían puntos negros en el desarrollo PHP que se han tratado de solucionar con la versión 5, aspectos que se echaron en falta en la versión 4, casi desde el día de su lanzamiento. Nos referimos principalmente a la programación orientada a objetos **(POO)** que, a pesar de que **estaba soportada a partir de PHP3, sólo implementaba una parte muy pequeña de las características de este tipo de programación.**
- El **principal objetivo de PHP5** ha sido **mejorar los mecanismos de POO** para solucionar las carencias de las anteriores versiones. Un paso necesario para conseguir que PHP sea un **lenguaje apto para todo tipo de aplicaciones y entornos,** incluso los más exigentes.

Modelo de orientación a objetos en PHP 5

Uno de los problemas básicos de las versiones anteriores de PHP era la clonación de objetos, que se realizaba al asignar un objeto a otra variable o al pasar un objeto por parámetro en una función. Para solventar este problema PHP5 hace uso de los manejadores de objetos (Object handles), una especie de punteros que apuntan hacia los espacios en memoria donde residen los objetos. Cuando se asigna un manejador de objetos o se pasa como parámetro en una función, se duplica el propio object handle y no el objeto en si.

Algunas características del trabajo con POO en PHP 5

1.- Nombres fijos para los constructores y destructores

Los nombres predefinidos para los métodos constructores y destructores son __construct() y __destruct().

2.- Acceso public, private y protected a propiedades y métodos

Estos modificadores de acceso habituales de la POO sirven para definir qué métodos y propiedades de las clases son accesibles desde cada entorno.

3.- Posibilidad de uso de interfaces

Éstas se utilizan en la POO para definir un conjunto de métodos que implementa una clase. Una clase puede implementar varias interfaces o conjuntos de métodos.

4.- Métodos y clases final

Se puede indicar que un método es "final" o que la clase es "final", lo que se indica es que esa clase no permite ser heredada por otra clase.

5.- Operador instanceof

Se utiliza para saber si un objeto es una instancia de una clase determinada.

6.- Atributos y métodos static

Son las propiedades y funcionalidades a las que se puede acceder a partir del nombre de clase, sin necesidad de haber instanciado un objeto de dicha clase.

7.- Clases y métodos abstractos

Las clases abstractas no se pueden instanciar y los métodos abstractos no se pueden llamar. Ambos se utilizan más bien para ser heredados por otras clases, donde no tienen porque ser declarados abstractos.

8.- Constantes de clase

Se pueden definir constantes dentro de la clase y luego acceder a ellas a través de la propia clase.

9.- Funciones que especifican la clase que reciben por parámetro

En caso que el objeto no sea de la clase correcta, se produce un error.

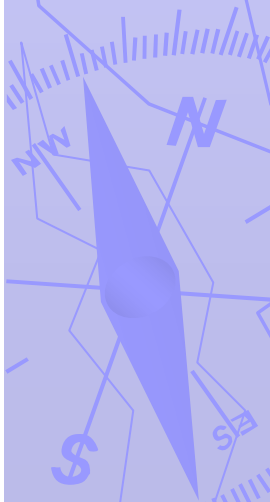
10.- Función __autoload()

La función __autoload() sirve para incluir el código de una clase que se necesite, y que no haya sido declarada todavía en el código que se está ejecutando.

Ejemplo de Clase y de Objeto

```
class hombre{
 var $nombre;
 var $edad;

 function comer($comida){
        //aquí el código del método
 }
 function moverse($destino){
        //aquí el código del método
 }
 function estudiar($asignatura){
        //aquí el código del método
 }
}
```

```
$pepe = new hombre();

$juan = new hombre();
```



Constructores

- Deben llamarse con un nombre fijo: **__construct().**

Destructores

- La creación del destructor es opcional. Sólo debemos crearlo si deseamos hacer alguna tarea específica cuando un objeto se elimine de la memoria.
- El destructor debe declararse con un nombre fijo: **__destruct().**
- El destructor se ejecuta justo antes de que el objeto sea eliminado de memoria. En PHP5 **un objeto desaparece cuando ya no es referenciado**, es decir, cuando todos sus manejadores han desaparecido.
- El hecho de que **todos los constructores** tengan el **mismo nombre facilita la llamada** al constructor de la clase base **desde su clase derivada.**

EJEMPLO:

- En el siguiente ejemplo sobre un videoclub vemos su funcionamiento:

```
class cliente{
        var $nombre;
        var $numero;
        var $peliculas_alquiladas;

        function __construct($nombre,$numero){
            $this->nombre=$nombre;
            $this->numero=$numero;
            $this->peliculas_alquiladas=array();
        }

        function __destruct(){
            echo "<br>destruido: " . $this->nombre;
        }

        function dame_numero(){
            return $this->numero;
        }
    }
```

USO:

```
//instanciamos un par de objetos cliente
$cliente1 = new cliente("Pepe", 1);
$cliente2 = new cliente("Roberto", 564);

//mostramos el numero de cada cliente creado
echo "El identificador del cliente 1 es: " .
$cliente1->dame_numero();
echo "El identificador del cliente 2 es: " .
$cliente2->dame_numero();
```

- El destructor imprime un mensaje en pantalla con el nombre del cliente que se ha destruido. Tras su ejecución obtendríamos la siguiente salida.

```
El identificador del cliente 1 es: 1
El identificador del cliente 2 es: 564
destruido: Pepe
destruido: Roberto
```

Como vemos, antes de acabar el script, se libera el espacio en memoria de los objetos construidos

- Si una **variable local** deja de existir, se llama al destructor definido para ese objeto.

- También podemos deshacernos de un objeto sin necesidad que acabe el ámbito donde fue creado. Para ello tenemos la función **unset()** que **recibe una variable y la elimina de la memoria**. Cuando se pierde una variable que contiene un objeto y ese objeto deja de tener referencias, se elimina al objeto y se llama al destructor.

Modificadores de acceso a métodos y propiedades en PHP5

- **Public:** puede acceder a ese atributo, cualquier otro elemento de nuestro programa. Es el modificador por defecto.
- **Private** : Indica que esa variable sólo se va a poder acceder desde el propio objeto, nunca desde fuera. (Nivel de acceso más restrictivo).
- **Protected:** El método/ atributo es público dentro de la propia clase y en sus heredadas. Es privado desde cualquier otra parte. (Indica un nivel de acceso medio).

Referencias, manejadores, handlers

- Los objetos en PHP5 son referenciados a través de **manejadores (handlers, referencias).**
- Como consecuencia de esta característica, cuando se hace una **asignación de un objeto** en realidad sólo **se duplica el manejador** (referencia) **y no el objeto completo.**
- Lo mismo ocurre con los objetos pasados como **parámetros a las funciones: siempre** se pasan **por referencia** puesto que con lo que se trabaja dentro de la función es un **duplicado del manejador** original del objeto.
- Ejemplo:

  Con la sentencia

  ```
  $obj_alias = $obj_vehiculo
  ```

  estamos creando un **alias del objeto (duplicamos el manejador**). Cualquier operación sobre $obj_alias afecta a las características de $obj_vehiculo.

El Constructor de copia __clone() y el operador $this/$that

- Las **variables** que representan objetos **son** en realidad **referencias** (manejadores, handlers) **al objeto** en sí.
- Sin embargo, en ocasiones es útil la posibilidad de copiar objetos. PHP5 introduce el método predefinido **__clone()**, **disponible por defecto en todos los objetos y redefinible para cada clase.**
- **Si no se redefine** el método clone en una clase, la **copia** del objeto se realiza **bit a bit**. Es decir, el objeto resultante es una copia exacta del original
- Ejemplo: copia de objetos

```
$vehiculo2 = vehiculo1->__clone();
```

- Hay casos en los que interesará crear métodos __clone() a medida.

El Constructor de copia __clone() y el operador $this/$that

```
class Rueda {
        public $nombre;

        function __construct($nombre) {
                $this->nombre = $nombre;
        }
}

class Moto {
        public $marca;
        public $ruedas = array();

        //// constructor de copia para Moto
        //// $that referencia al objeto original
        //// $this referencia al objeto clonado
        function __clone() {
                $this->ruedas['del'] = new Rueda
                ($that->ruedas['del']->nombre);
                $this->ruedas['tras'] = new Rueda
                ($that->ruedas['tras']->nombre); }
}
```

El Constructor de copia __clone() y el operador $this/$that

```
// creamos dos ruedas y una moto
$rueda_delantera_mich = new Rueda ("delantera-michelin");
$rueda_trasera_mich = new Rueda ("trasera-michelin");
$yamaha_R1 = new Moto();
$yamaha_R1->marca = "Yamaha R1";
$yamaha_R1->ruedas['del'] = $rueda_delantera_mich;
$yamaha_R1->ruedas['tras'] = $rueda_trasera_mich;

// clonamos la moto
$yamaha_R1_bis = $yamaha_R1->__clone();
```

El Constructor de copia __clone() y el operador $this/$that

- El método __clone() de Moto crea nuevas ruedas en la moto clonada. Como en el constructor de copia intervienen dos objetos, la palabra clave **$that**, **hace referencia al objeto original,** mientras que **$this hace referencia al nuevo objeto (objeto clonado).**
- Hay que tener en cuenta que **la copia bit a bit se sigue realizando de forma automática,** replicando todos los atributos del objeto Moto original en el objeto Moto duplicado. La funcionalidad del **método __clone() incluido en una clase se ejecuta después de esta copia bit a bit**. En el ejemplo anterior, el atributo **$marca** se duplica de forma automática (el método __clone() no hace nada al respecto)

Comparación de objetos

- Cuando se usa el **operador de comparación** (*==*), las variables del objeto son comparadas de una forma simple, es decir, **dos instancias de objetos** son **iguales** si tienes los **mismos atributos y valores**, y son **instancias de la misma clase**.

- Por otro lado, cuando se usa el **operador idéntico** (*===*), las variables del objeto son idénticas sólo **si refieren a la misma instancia de la misma clase**.

Control de tipos en PHP

- El control de tipos en parámetros pasados a funciones y métodos sólo es válido para objetos, no para tipos elementales.

```
class Persona { public $nombre; }
class Gato { public $nombre; }

// esta funcion solo puede recibir objetos de tipo Persona
function muestraNombre(Persona $persona) {
   echo $persona->nombre;
}
$persona = new Persona;
$persona->nombre = "Pepe";
$gato = new Gato;
$gato->nombre = "Rufus";
muestraNombre ($persona); // correcto
muestraNombre ($gato); // ERROR: un gato no es una persona
```

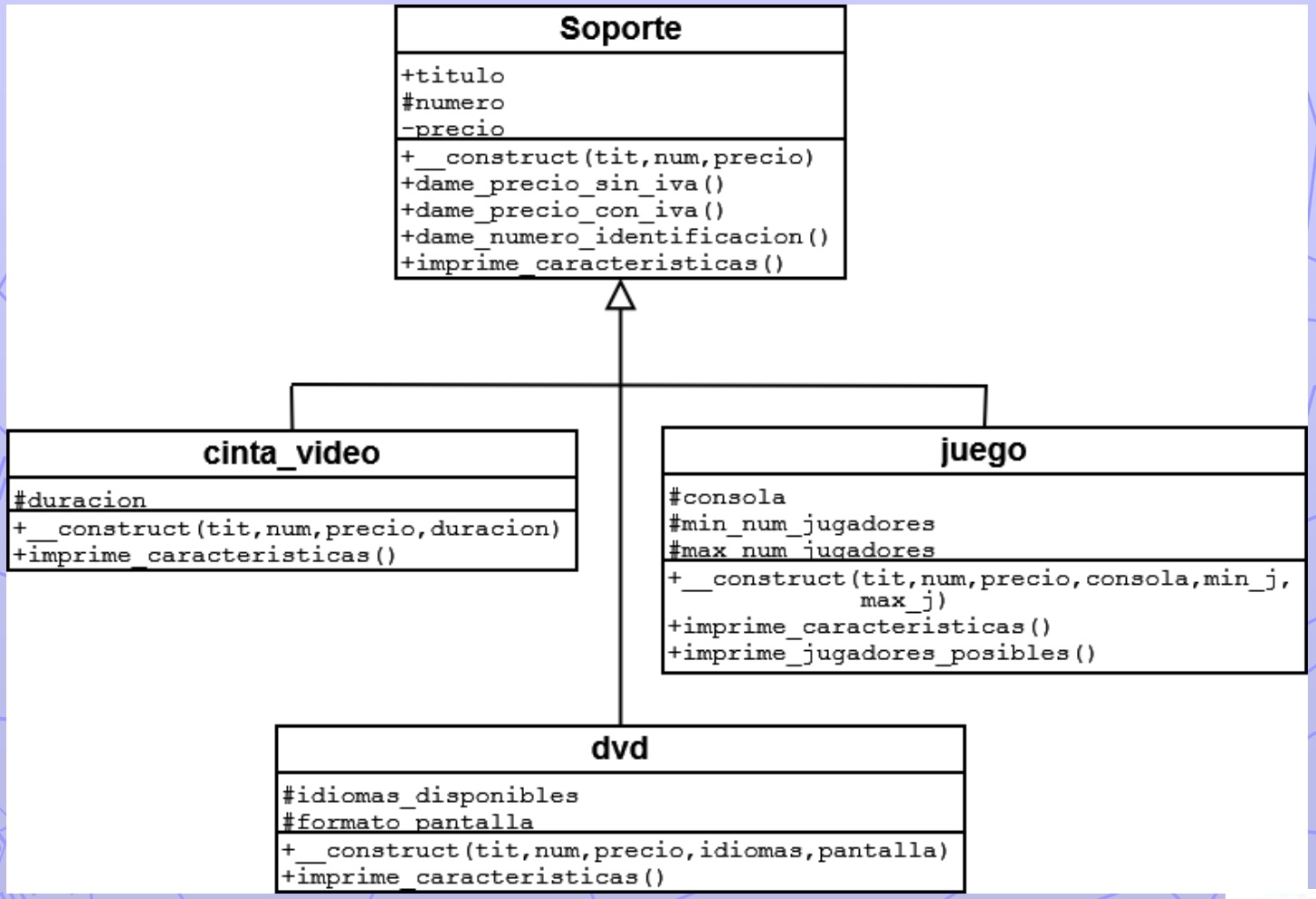
- Parece lógico que si una función acepta objetos de una determinada clase, los acepte también de sus clases derivadas.

HERENCIA en PHP5

- **Ejemplo:** Videoclub

  Los videoclubs ofrecen distintos tipos de elementos para **alquiler**, como pueden ser las **películas** (cintas de vídeo o DVD) y los **juegos**. Cada elemento tiene unas características propias y algunas comunes. A la clase general que indica cualquier tipo de elemento a alquilar la llamamos **"soporte"**, que incluye las características comunes para todos los tipos de elementos en alquiler que tenemos en el videoclub. Luego hemos creado tres tipos de soportes distintos, que heredan de la clase soporte, que incluyen algunas características y funcionalidades nuevas, pero que también se encuentran enmarcados dentro de las características comunes del soporte. Estos tipos de soporte serán **"cinta_video"**, **"dvd"** y **"juego"**.

HERENCIA en PHP5



HERENCIA EN PHP5

Clase soporte

```
class soporte{
    public $titulo;
    protected $numero;
    private $precio;

function __construct($tit,$num,$precio){
    $this->titulo = $tit;
    $this->numero = $num;
    $this->precio = $precio;
}

public function dame_precio_sin_iva(){
    return $this->precio;
}
```

HERENCIA EN PHP5

Clase soporte

```
    public function dame_precio_con_iva(){
        return $this->precio * 1.16;
    }

    public function dame_numero_identificacion(){
        return $this->numero;
    }

    public function imprime_caracteristicas(){
        echo $this->titulo;
        echo "<br>" . $this->precio . " (IVA no
    incluido)";
    }
}
```

HERENCIA EN PHP5

Clase soporte

- Los atributos hay en esta clase son **título**, **numero** (un identificador del soporte) y **precio**. Si prestamos atención, cada uno de ellos se ha definido con un modificador de acceso distinto (public, protected, private), para ver cómo se maneja el acceso en las subclases en cada uno de estos casos.
- Como se ha definido como **private** el atributo **precio**, este atributo sólo se podrá acceder dentro del código de la clase, es decir, en la propia definición del objeto. Si queremos acceder al precio desde fuera de la clase (algo muy normal si tenemos en cuenta que vamos a necesitar el precio de un soporte desde otras partes de la aplicación) será necesario crear un método que nos devuelva el valor del precio. Este método debería definirse como **public**, para que se pueda acceder desde cualquier sitio que se necesite.

HERENCIA EN PHP5

Clase soporte

- Hemos definido un **constructor**, que recibe los valores para la inicialización del objeto. Un método **dame_precio_sin_iva()**, que devuelve el precio del soporte, sin aplicarle el IVA. Otro método **dame_precio_con_iva()**, que devuelve el precio una vez aplicado el 16% de IVA. El método **dame_numero_identificacion()**, que devuelve el número de identificador y **imprime_caracteristicas()**, que muestra en la página las características de este soporte.
- En todos los métodos se hace uso de la **variable $this**. Esta variable no es más que una **referencia al objeto** sobre el que se está ejecutando el método.

HERENCIA EN PHP5

Clase soporte

- Para ejecutar cualquiera de estos métodos, **primero** tenemos que **haber creado un objeto a partir de una clase**. Luego podremos llamar los métodos de un objeto.
- Esto se hace con **$mi_objeto->metodo_a_llamar()**.
- Dentro de método, cuando se utiliza la variable $this, se está haciendo referencia al objeto sobre el que se ha llamado al método, en este caso, el objeto $mi_objeto. Con $this->titulo estamos haciendo referencia al atributo "titulo" que tiene el objeto $mi_objeto.

HERENCIA EN PHP5

Clase soporte

- Ejemplo de uso de la clase:

```
$soporte1 = new soporte("Los Intocables",22,3);

echo "<b>" . $soporte1->titulo . "</b>";

echo "<br>Precio: " .
$soporte1->dame_precio_sin_iva() . " euros";

echo "<br>Precio IVA incluido: " .
$soporte1->dame_precio_con_iva() . " euros";
```

HERENCIA EN PHP5

Clase soporte

- En este caso hemos creado una instancia de la clase soporte, en un objeto que hemos llamado $soporte1. Luego imprimimos su atributo título (como el título ha sido definido como public, podemos acceder a él desde fuera del código de la clase).
  Finalmente se llaman a los métodos dame_precio_sin_iva() y dame_precio_con_iva() para el objeto creado, y el resultado lo mostramos por pantalla.

  Nos daría como resultado esto:

```
Los Intocables
Precio: 3 euros

Precio IVA incluido: 3.48 euros
```

HERENCIA EN PHP5

Clase cinta_video

- Siguiendo con el diagrama de clases propuesta para el videoclub, construimos una clase para los soportes de tipo cinta de vídeo. Las **cintas de vídeo** tienen **un atributo nuevo** que es la **duración de la cinta**. **No** tienen ningún **método nuevo**, pero sí que hay que **redefinir los métodos** de la clase soporte, en concreto, hay que redefinir el constructor (ya que ahora necesitamos una variable más que indique la duración) y la función que imprime características ya que la duración también hay que imprimirla ahora.

HERENCIA EN PHP5

Clase cinta_video

```
class cinta_video extends soporte {
    protected $duracion;

    function __construct($tit,$num,$precio,$duracion){
          parent::__construct($tit,$num,$precio);
          $this->duracion = $duracion;
    }

    public function imprime_caracteristicas(){
          echo "Película en VHS:<br>";
          parent::imprime_caracteristicas();
          echo "<br>Duración: " . $this->duracion;
    }
}
```

HERENCIA EN PHP5

Clase cinta_video

- Con la primera línea `class cinta_video extends soporte` estamos indicando que se está **definiendo** la **clase cinta_video** y que va a **heredar** de la **clase soporte**.
- Como se está heredando de una clase, PHP tiene que conocer el código de la clase "padre", en este caso la clase soporte. Podemos colocar los dos códigos en el mismo fichero, o si están en ficheros independientes, debemos incluir el código de la clase soporte con la instrucción include o require de PHP.
- En la clase cinta_video hemos definido un **nuevo atributo** llamado **`$duracion`**, que almacena el tiempo que dura la película.

HERENCIA EN PHP5

Clase cinta_video

CONSTRUCTOR

- Aunque la clase sobre la que heredamos (la clase soporte) tenía definido un constructor, la cinta de vídeo debe inicializar la nueva propiedad `$duracion`, que es específica de las cintas de video.
- Por ello, vamos a **redefinir** el método **constructor**, lo que se hace simplemente **volviendo a escribir el método**. Ahora, para la `clase cinta_video`, hay que **inicializar** los **atributos** definidos en la `clase soporte`, **más** el **atributo** `$duracion`, que es propio de cinta_video.

HERENCIA EN PHP5

Clase cinta_video

CONSTRUCTOR

- En la **primera línea** del constructor **se llama al constructor** creado para la **clase soporte**. Para ello utilizamos **parent::** y luego el nombre del método de la clase padre al que se quiere llamar, en este caso __constructor(). Al *constructor de la clase padre* le enviamos las *variables* que se deben *inicializar* con la *clase padre*.
- En la **segunda línea** del constructor **se inicializa el atributo duracion**, con lo que hayamos recibido por parámetro.

HERENCIA EN PHP5

Clase cinta_video

IMPRIME_CARACTERÍSTICAS

- Nos pasa lo mismo con el método `imprime_caracteristicas()`, que ahora **debe mostrar también el nuevo atributo**, propio de la **clase cinta_video**. Como se puede observar en el código de la función, se hace uso también de `parent::imprime_caracteristicas()` para utilizar el método definido en la clase padre.

HERENCIA EN PHP5

Clase cinta_video

- Ejemplo de uso de la clase:

```
$micinta = new cinta_video("Los Otros", 22, 4.5, "115
minutos");

echo "<b>" . $micinta->titulo . "</b>";

echo "<br>Precio: " . $micinta->dame_precio_sin_iva()
. " euros";

echo "<br>Precio IVA incluido: " .
$micinta->dame_precio_con_iva() . " euros";

echo "<br>" . $micinta->imprime_caracteristicas();
```

HERENCIA EN PHP5

Clase cinta_video

- Nos daría como resultado esto:

```
Los Otros
Precio: 4.5 euros
Precio IVA incluido: 5.22 euros
Película en VHS:
Los Otros
4.5 (IVA no incluido)
Duración: 115 minutos
```

HERENCIA EN PHP5

Clase dvd

- Una vez definidas la clase soporte y la clase cinta_video, continuamos la implementación del diagrama de clases con la **clase dvd**, que es muy parecida a la implementación que se ha visto para la clase cinta_video. Lo único que cambia es que ahora vamos a **definir otros atributos** relacionados con los DVD, como son los **idiomas disponibles** en el DVD y el **formato de pantalla** que tiene la grabación.

HERENCIA EN PHP5

Clase dvd

```
class dvd extends soporte {
   protected $idiomas_disponibles;
   protected $formato_pantalla;

   function __construct($tit,$num,$precio,$idiomas,$pantalla){
        parent::__construct($tit,$num,$precio);
        $this->idiomas_disponibles = $idiomas;
        $this->formato_pantalla = $pantalla;
   }

   public function imprime_caracteristicas(){
        echo "Película en DVD:<br>";
        parent::imprime_caracteristicas();
        echo "<br>" . $this->idiomas_disponibles;
   }
}
```

HERENCIA EN PHP5

Clase juego

- Por su parte, la **clase juego**, tendrá **3 nuevos atributos**. Estos son *consola*, para especificar la consola para la que está creado este juego, *min_num_jugadores*, para especificar el número de jugadores mínimo y *max_num_jugadores*, para especificar el máximo número de jugadores que pueden participar en el juego.

HERENCIA EN PHP5

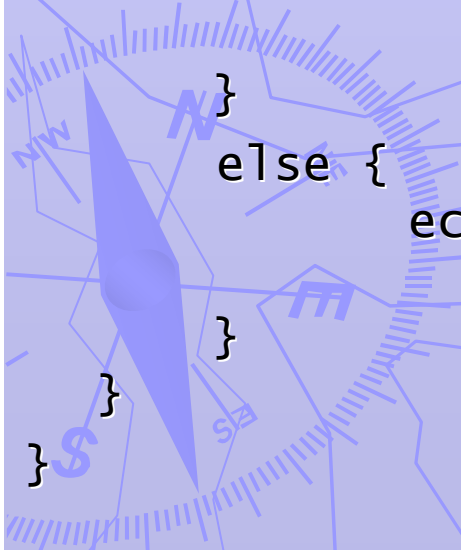
Clase juego

```
class juego extends soporte{
   protected $consola; // consola del juego ej: DS Lite
   protected $min_num_jugadores;
   protected $max_num_jugadores;

   function
   __construct($tit,$num,$precio,$consola,$min_j,$max_j){
        parent::__construct($tit,$num,$precio);
        $this->consola = $consola;
        $this->min_num_jugadores = $min_j;
        $this->max_num_jugadores = $max_j;
   }

   public function imprime_caracteristicas(){
        echo "Juego para: " . $this->consola . "<br>";
        parent::imprime_caracteristicas();
        echo "<br>" . $this->imprime_jugadores_posibles();
   }
```

HERENCIA EN PHP5

Clase juego

```
public function imprime_jugadores_posibles() {
    if ($this->min_num_jugadores == $this->max_num_jugadores){
        if ($this->min_num_jugadores==1)
            echo "<br>Para un jugador";
        else
            echo "<br>Para " . $this->min_num_jugadores .
                " jugadores";
    }
    else {
        echo "<br>De " . $this->min_num_jugadores . " a " .
            $this->max_num_jugadores . " jugadores.";
    }
  }
}
```



HERENCIA EN PHP5

Clase juego

- Nos fijamos en el **constructor**, que **llama al constructor de la clase padre** para inicializar algunos atributos propios de los soportes en general.
- También está el **método imprime_jugadores_posibles()**, que muestra los jugadores permitidos. Ha sido declarado como **public**, para que se pueda acceder a él desde cualquier lugar. Nos **da un mensaje** como *"Para un jugador"* o *"De 1 a 2 Jugadores"*, dependiendo de los valores **min_num_jugadores** y **max_num_jugadores**.
- Se redefine la función **imprime_caracteristicas()**, para mostrar todos los datos de cada juego. Primero se muestra la consola para la que se ha creado el juego, después los datos generales (propios de la clase soporte) con la llamada al mismo método de la clase "parent" y el número de jugadores disponibles se muestra con una llamada al método imprime_jugadores_posibles().

HERENCIA EN PHP5

Clase juego

- Ejemplo de uso de la clase:

```
$mijuego = new juego("El Padrino", 21, 2.5, "PSP",1,1);
$mijuego->imprime_caracteristicas();

// Esta línea daría un error porque no se permite acceder a un
// atributo protected del objeto
// echo "<br>Jugadores: " . $mijuego->min_num_jugadores;
// Se tendría que crear un método para que acceda a los
// atributos protected
$mijuego->imprime_jugadores_posibles();

echo "<p>";
$mijuego2 = new juego("NBA Live 2007", 27, 3, "Playstation
2",1,2);
echo "<b>" . $mijuego2->titulo . "</b>";
$mijuego2->imprime_jugadores_posibles();
```

HERENCIA EN PHP5

Clase juego

- Nos daría como resultado esto:

```
Juego para: PSP
El Padrino
2.5 (IVA no incluido)
Para un jugador
Para un jugador

NBA Live 2007
De 1 a 2 Jugadores.
```

La palabra reservada 'Final'

- PHP5 introduce la palabra reservada **final**, que puede tener dos significados según se aplique a un método o a una clase.
- Si un método se declara como final, usando el prefijo **final** en la definición del método, se obliga a las clases que heredan a mantener ese método y no poder redefinirlo.
- Si la clase en sí misma es definida como 'final' entonces no puede existir ninguna clase que herede de ella.

La palabra reservada 'Final'

```
class ClaseBase {
    public function test() {
        echo "ClaseBase::test() llamada\n";
    }

    final public function masTests() {
        echo "ClaseBase::masTests() llamada\n";
    }
}

class ClaseHijo extends ClaseBase {
    public function masTests() {
        echo “ClaseHijo::masTests() llamada\n";
    }
}
// Esto es un error: No se puede redefinir el método
// final ClaseBase::masTests()
```

La palabra reservada 'Final'

```
final class ClaseBase {
    public function test() {
        echo "ClaseBase::test() llamada\n";
    }

    // Aquí da igual si se declara el método como
    // final o no
    final public function moreTesting() {
        echo "ClaseBase::moreTesting() llamada\n";
    }
}

class ClaseHijo extends ClaseBase {
}
// Esto es un error: Class ClaseHijo no puede
heredar de una clase final (ClaseBase)
```

Alcance del operador de resolución (::)

- El alcance del operador de resolución o en términos simples, **dobles dos puntos**, es un símbolo que permite acceso a los **miembros o métodos static**, **constantes**, y **"eliminados"** (métodos heredados que han sido redefinidos) de una clase.
- Cuando se hace referencia a estos elementos desde **fuera de la definición** de la clase, se ha de usar el nombre de la clase.
-

```
class MiClase {
    const VALOR_CTE = 'Un valor constante';
}

echo MiClase::VALOR_CTE;
```

Alcance del operador de resolución (::)

- Dos palabras reservadas ***self*** y ***parent*** son usadas para acceder a los miembros o métodos desde **dentro** de la **definición de la clase**.
-
```
class OtraClase extends MiClase {
    public static $mi_static = 'static var';

    public static function dosPuntosDobles() {
        echo parent::VALOR_CTE . "\n";
        echo self::$mi_static . "\n";
    }
}

OtraClase::dosPuntosDobles();
```

Alcance del operador de resolución (::)

- Al realizar la herencia a partir de una clase, si se redefine el método en la subclase, PHP no llamará el método de la clase padre. Es opcional en la subclase decidir si se debe llamar al método de la clase padre o no.
-
```
class MiClase {
    protected function miFunc() {
        echo "MiClase::miFunc()\n";
    }
}

class OtraClase extends MiClase {
    // Redefine el método de la clase de la que hereda
    public function miFunc() {
        // Pero sigue llamando a la función de la clase
        // ancestro
        parent::miFunc();
        echo "OtraClase::miFunc()\n";
    }
}

$clase = new OtraClase();
$clase->miFunc();
```

La palabra reservada 'Static'

- Declarar miembros de clases o métodos como static, los hace **accesibles** desde **fuera del contexto del objeto**. Un miembro o método declarado como static no puede accederse con una variable que sea una instancia del objeto y no puede ser redefinido en una subclase.
- La declaración **static** debe estar **después de la declaración de visibilidad**. Por **compatibilidad** con PHP4, **si no se usa** la declaración de visibilidad, entonces el miembro o método será tratado tal si como se hubiera declarado como ***public static***.
- A causa de que los métodos estáticos son accesibles sin que se haya creado una instancia del objeto, **la pseudovariable *$this* no** está **disponible** dentro de los **métodos** declarados como **static**.
- Las propiedades estáticas **no se pueden acceder** a través del objeto usando el **operador flecha**- >

La palabra reservada 'Static'

```php
class Foo {
    public static $mi_static = 'foo';

    public function staticValor() {
        return self::$mi_static;
    }
}

class Bar extends Foo {
    public function fooStatic() {
        return parent::$mi_static;
    }
}

print Foo::$mi_static . "\n";

$foo = new Foo();
print $foo->staticValor() . "\n";
print $foo->mi_static . "\n";
// "Propiedad" no definida mi_static
// $foo::mi_static no es posible

print Bar::$mi_static . "\n";
$bar = new Bar();
print $bar->fooStatic() . "\n";
```

Constantes de la clase

- Es posible definir valores constantes en cada clase manteniendo el mismo valor y siendo incambiable. Las constantes difieren de las variables normales en que no se usa el símbolo *$* para declararlas o usarlas. Como los miembros static, los valores constantes no se pueden acceder desde una instancia de un objeto (usando *$objeto::constante*).
-
```
class MiClase {
    const constante = 'valor constante';

    function mostrarConstante() {
        echo self::constante . "\n";
    }
}

echo MiClase::constante . "\n";

$clase = new MiClase();
$clase->mostrarConstante();
// echo $clase::constante;  no se permite
```

Abstracción de objetos

- PHP 5 introduce **clases y métodos abstractos**. **No** es permitido crear una **instancia** de una **clase** que ha sido definida como **abstracta**. Cualquier **clase** que contenga por lo menos **un método abstracto** debe también ser **abstracta**. Los **métodos** definidos como **abstractos** simplemente **declaran el método**, **no** pueden **definir la implementación**.
- Cuando **se hereda** desde una **clase abstracta**, **todos los métodos** marcados como **abstractos** en la declaración de la clase padre, **deben de ser definidos** en la subclase; adicionalmente, **estos métodos** se deben definir **con la misma o mayor visibilidad**. Si el método abstracto es definido como protected, la implementación de la función debe ser definida como protected o public.

Abstracción de objetos

```
abstract class ClaseAbstracta {
    // Se fuerza la herencia de la clase para definir
    // estos métodos
    abstract protected function tomarValor();
    abstract protected function prefixValor($prefix);

    // Método común
    public function printOut() {
        print $this->tomarValor() . "\n";
    }
}

class ClaseConcreta1 extends ClaseAbstracta {
    protected function tomarValor() {
        return "ClaseConcreta1";
    }

    public function prefixValor($prefix) {
        return "{$prefix}ClaseConcreta1";
    }
}
```

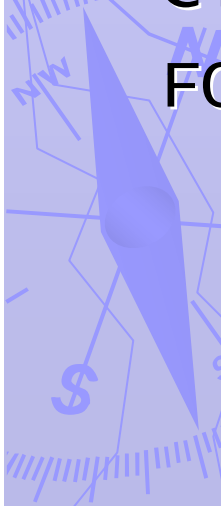
Abstracción de objetos

```
class ClaseConcreta2 extends ClaseAbstracta {
    protected function tomarValor() {
        return "ClaseConcreta2";
    }

    public function prefixValor($prefix) {
        return "{$prefix}ClaseConcreta2";
    }
}

$class1 = new ClaseConcreta1;
$class1->printOut();
echo $class1->prefixValor('FOO_') ."\n";

$class2 = new ClaseConcreta2;
$class2->printOut();
echo $class2->prefixValor('FOO_') ."\n";
```

Abstracción de objetos

- El resultado del ejemplo seria:

```
claseContreta1
FOO_claseConcreta1
claseConcreta2
FOO_claseContreta2
```



Interfaces de objetos

- Las interfaces de objetos permiten **crear código** el cual especifica **métodos** que **una clase debe implementar**, sin tener que definir cómo serán manejados esos métodos.
- Las interfaces son definidas usando la palabra reservada **interface**, de la misma manera que las clases estándar, pero sin que cualquiera de los métodos tenga su contenido definido.
- Todos los **métodos en una interfaz** deben ser **públicos**, esto es la naturaleza de una interfaz.
- Para **implementar una interfaz**, se usa el operador ***implements***. **Todos los métodos** en la **interfaz** deben ser **implementados** dentro de **una clase**; de no hacerse así resultará en error fatal. Las **clases** pueden **implementar más de una interfaz** si se desea, al separar cada interfaz con una coma.

Interfaces de objetos

```
// Se declara la interfaz 'iTemplate'
interface iTemplate {
    public function ponerVariable($nombre, $var);
    public function verHtml($template);
}

// Se implementa la interfaz
// Esto funcionará
class Template implements iTemplate {
    private $vars = array();

    public function ponerVariable($nombre, $var) {
        $this->vars[$nombre] = $var;
    }

    public function verHtml($template) {
        foreach($this->vars as $nombre => $value) {
            $template = str_replace('{' . $nombre .
'}',                                $value, $template);
        }

        return $template;
    }
}
```

Interfaces de objetos

```
// Esto no funcionará
// Error: Class MalTemplate contiene 1 método
// abstracto y por tanto, debe ser declarada
// como abstracta (iTemplate::verHtml)

class MalTemplate implements iTemplate {
    private $vars = array();

    public function ponerVariable($nombre, $var) {
        $this->vars[$nombre] = $var;
    }
}
```

Bibliografía

- http://www.php.net/ (Sitio web oficial, en inglés)
- http://www.desarrolloweb.com/manuales/58/ (Manual de PHP5 sobre objetos)
- http://es.php.net/manual/es/language.oop5.php (Manual de PHP5 sobre objetos)
- http://www.zonaphp.com/programacion-orientada-a-objetos-en-php/ (Blog sobre orientación a objetos en PHP)
- http://www.webestilo.com/php/articulo.phtml?art=27 (Nociones básicas sobre orientación a objetos en PHP)
- http://www.programacion.net/php/ (Foro en español de PHP)
- http://php.hispano.net/ (Foros y artículos sobre PHP, en español)