ÍNDICE

CAPÍTULO 2. EVOLUCIÓN DE PHP

El presente curso está dirigido a usuarios con conocimientos de programación secuencial, programación orientada a objetos, y entornos de tecnología Web. Trata de enseñar técnicas de programación para afrontar proyectos profesionales en entornos Web.

Asumimos que el alumno ya ha tenido contacto con el lenguaje *PHP*, seguramente en la versión (de momento) más extendida del mismo, *PHP4*, que es la versión que a día de hoy es más fácil de encontrar. Sin embargo, en este curso el protagonismo será para la última y única versión con soporte, PHP5.

¿Por qué ha costado (o está costando) actualizar a *PHP5*? A pesar de que la versión 5 de *PHP* es totalmente estable desde hace más de tres años, y está diseñada para utilizar técnicas de programación más avanzadas y útiles en la implementación de grandes proyectos, los proyectos existentes han tardado en adaptarse a esta nueva versión.

El motivo del retraso en la portabilidad a *PHP5* se debe, precisamente, a que esta nueva versión introdujo importantes (y ventajosas) modificaciones en el núcleo del intérprete. Sin embargo, estas modificaciones producían problemas de incompatibilidad, que obligaron a revisar todo el código de los proyectos. Como toda comunidad open-source, los proyectos podían tener interdependencias entre sí, de tal forma, que un proyecto podía iniciar la migración a PHP5 y encontrarse con que una de sus librerías externas, no estaba aún preparada para la nueva versión del lenguaje.

Algunas de esas incompatibilidades eran debido a las novedades y ventajas incorporadas por la 5ª versión del lenguaje. Otras, (la gran mayoría) eran debidas a un uso desordenado del lenguaje, sin seguir las recomendaciones y pautas que acompañaban al mismo.

La lista completa de incompatibilidades forma parte de la documentación oficial, puede accederse a la misma en este enlace [php.net/manual/en/migration5.incompatible.php].

PHP fue desde sus inicios, un lenguaje muy versátil y muy laxo en cuanto a restricciones, similar a C en el que basaba su gramática. En sus orígenes parece un lenguaje *juguete*[1], destinado a completar un universo Web en expansión, y es creado para facilitar el contenido dinámico. Su licencia *openSource*, su coste, y su juventud hicieron que cualquiera pudiera hacer sus pinitos en *PHP*, desde novatos o simples aficionados a la programación hasta verdaderos *hackers*.

La ligereza, simplicidad y su facilidad de uso fueron el mejor reclamo en una comunidad que simplemente lo veía como la herramienta para facilitar la generación de código HTML. Sin embargo, precisamente esa comunidad que lo hizo crecer, talvez ha sido después un pequeño lastre haciendo que, aún hoy, mucha gente descarte el uso de *PHP* en grandes proyectos.

Durante el desarrollo de este curso, veremos que es un lenguaje capaz de competir con el tan manido *Java*. Ojo, *Java* es un lenguaje muy potente, con una comunidad y un soporte amplísimo, simplemente estamos diciendo que *PHP* puede ser una elección tan buena o incluso mejor en muchos proyectos. Centremos pues nuestro interés en esas características que deseamos en el lenguaje y que consideramos necesarias para poder utilizarlo.

PHP surge como tal en el año 98. En esa época, ya estaban en auge lenguajes orientados a objetos, como *C++*, el mismo *Java* o *Delphi*, por lo que desde prácticamente las primeras versiones del mismo, el lenguaje contaba con características propias de los lenguajes orientados a objetos. Sin embargo, dichas características no eran más que apariencia, pues en realidad no dejaban de enmascarar distintos métodos para acceder a arrays asociativos. Las capacidades de orientación a objetos del lenguaje no fueron sustancialmente modificadas hasta el año 2000, cuando se incluyó la palabra reservada *class* y se implementó un sencillo sistema de herencia.

El "problema" o más bien defecto principal que ha tenido *PHP* hasta su versión 5 ha sido que ofrecía al programador el tipo *objeto* como tipo

1 El gran aliado de *PHP*, *MySQL* ha pasado por etapas similares. Se pensaba en él como un sistema de gestión de base de datos ligero y simple. Sin embargo, la última versión del mismo, curiosamente la serie 5.x, presenta características que le permiten competir en muchos proyectos con *SGBDs* más "serios" como *PostgreSQL* u *Oracle*.

nativo, pero internamente[2] la implementación del mismo no lo diferenciaba de otros tipos nativos del lenguaje como enteros o cadenas.

Esta implementación producía efectos extraños. Por ejemplo, los desarrolladores que habían trabajado con otros lenguajes orientados a objetos, tenían que tener claro que *PHP* (en sus versiones 3 y4) realizaba un **clonado implícito** cada vez que se asignaba un objeto, lo que dificultaba o complicaba ciertas técnicas básicas habituales en la programación orientada a objetos. Veamos el siguiente ejemplo:

```php
<?php
class Persona
{
    var $nombre;
    function Persona ($nombre)
    {
        $this->nombre = $nombre;
    }
    function setNombre($nombre)
    {
        $this->nombre = $nombre;
    }
    function getNombre()
    {
        return $this->nombre;
    }
} //Fin clase Persona

function pasarMinisculasNombre($obj)
{
    $nuevoNombre = strtolower($obj->getNombre());
    $obj->setNombre($nuevoNombre);
}

$obj = new Persona("ANTONIO");
pasarMinisculasNombre($obj);
print $obj->getNombre();
?>
```

Un programador habituado a trabajar con lenguajes orientados a objetos espera que el resultado en pantalla sea, "*antonio*", sin embargo, este resultado sólo se obtiene si ejecutamos el código con *PHP5*. Si se ejecuta con *PHP4*, el resultado es "*ANTONIO*". Esto es debido a que el objeto pasado como argumento al método "*pasarMinusculasNombre*" es clonado implícitamente. Es decir, se pasa por valor como una variable de tipo estándar, como son las cadenas, enteros o números flotante.

Las razones de esta "lenta" evolución, quizá estén relacionadas con la poca credibilidad con la que partieron las aplicaciones o sistemas Web. Para realizar proyectos de envergadura, la plataforma Web ha ido evolucionando y se ha convertido en verdadero protagonista del desarrollo de *SW* de sistemas "serios", y con ella, lenguajes como *PHP*.

2 Esta simplificación es algo exagerada, pero no resulta relevante para el desarrollo del presente curso profundizar más en la implementación de la máquina *Zend*.

Zend anunció el pasado 13 de Julio de 2007 la finalización del soporte a *PHP4*, cuya **última** versión estable[3] fue la 4.4.9. Y desde entonces hemos asistido a una rápida transición a la versión 5 del lenguaje.

[3] En teoría la última versión estable de PHP iba a ser la 4.4.7, sin embargo, gran parte de la comunidad PHP no ha aceptado la "imposición" por parte de Zend de las formas más extrictas de la versión 5 y se han seguido sacando versiones que corrigen Bugs de seguridad.

2.1 Orientación a objetos.

Con el lanzamiento de *PHP5* en el año 2004, el lenguaje sufrió una verdadera revolución. Ya no se trataba de un lenguaje procedural que soporta características de orientación a objetos, sino que se convertía en un lenguaje totalmente orientado a objetos, que pasaba de permitir una compatibilidad "a extinguir" en el uso de técnicas ajenas a la POO de manera sistemática.

Posiblemente el cambio más sustancial e importante fue la sustitución en el sistema de referencias de objetos utilizado hasta la versión 4. Dicho cambio obliga a una revisión de todo código escrito para la versión 4, ya que rutinas que aparentemente funcionan, tenían efectos colaterales no deseados. En definitiva, esa ha sido la principal causa por la que la migración a *PHP5* ha resultado ser tan lenta.

Todo proyecto que haya hecho uso de técnicas de programación orientadas a objetos debe revisarse cuidadosamente. Con la versión 5 de *PHP* se introdujeron "manejadores", "*handlers*" o referencias a los identificadores de los objetos (*OIDS*). Por tanto, como en otros lenguajes *OO* (como *Java* o *C#*) el tratamiento de los objetos es diferente al de los tipos básicos.

A continuación, repasaremos las principales diferencias entre la versión 4 y la versión 5 de *PHP*, y comentaremos las nuevas características que se incorporaron en este último, y como algunas de ellas podían "simularse" en la versión *PHP4*. Durante la evolución del curso utilizaremos todas estas características, y algunas de ellas serán objeto de un estudio más detallado por nuestra parte, mientras que para otras simplemente haremos referencia a la excelente documentación oficial, dado que muchas de las novedades de *PHP5* son propias de otros lenguajes de programación *OO*, como *Java* o *C#*.

2.1.1 Encapsulamiento, las clases.

El encapsulamiento o encapsulación es un principio de abstracción que agrupa datos y procesos. En los lenguajes *OO* se traduce en ocultar a los usuarios el comportamiento interno de la implementación de un objeto, ofreciéndoles una interfaz externa mediante la cual poder interactuar con el objeto. Es decir, se traduce en clases.

Como en otros lenguajes de programación, la encapsulación en *PHP* se consigue mediante la definición abstracta de las clases, tanto en *PHP4* como en *PHP5* se define una clase a través de la palabra reservada *class*. A partir de aquí comienzan las diferencias, pues el soporte para clases de *PHP4* finaliza en ese punto.

- Modificadores de acceso (visibilidad):

    - *PHP4*: El acceso tanto a atributos como a métodos es siempre público. Se utiliza como prefijo la palabra reservada *var* para atributos y *function* para los métodos.

    - *PHP5*: Soporta las tres P (PPP = private, protected, public).

Los atributos pueden ser de tipos simples, o pueden ser a su vez otros objetos. De ello, la importancia que pueden llegar a tener los modificadores de acceso para mantener la encapsulación. Como en *PHP4*, tanto métodos como atributos son siempre públicos. Por ello, y con el objetivo de intentar mantener la distinción entre los servicios y atributos públicos respecto de los privados, muchos programadores anteponen un prefijo "especial" para las características privadas. Por ejemplo, un guión bajo "_". Pese a que de este modo no se impide una invocación a dicho método o un acceso al atributo, cuando se siguen estas convenciones es más fácil para otros programadores utilizar las clases de forma correcta.

En *PHP5*, además de los métodos especiales *__get*() y *__set*(), existe también un método especial que se invoca cuando realizamos una invocación a un método que no existe en esa clase. Este método se denomina *__call*().

Como vemos, en *PHP5* aparecen más métodos "especiales" proporcionados por el lenguaje, dichos métodos SIEMPRE tienen como prefijo dos guiones bajos "__", por lo que es desaconsejable emplear el doble guión para comenzar los nombres de atributos o métodos en nuestro código. Esta reserva de nomenclatura ya existía en *PHP4* para, por ejemplo, los métodos *__sleep*(), *__wakeup*().

- Atributos y miembros de clase:

Los atributos de clase no pertenecen a un objeto en concreto, sino a la propia clase en si misma. Se definen como un atributo cualquiera anteponiendo al nombre del mismo la cláusula "*static*".

Los métodos de clase, al igual que los atributos, son métodos que pertenecen a la clase en si y no requieren de ningún objeto instanciado para su invocación.

La manera de acceder a los atributos y métodos de clase, es a través del operador "*::*" precedido por la palabra reservada "*self*" si se está invocando desde dentro de la clase, de la palabra reservada "*parent*" si la invocación es desde una clase derivada, o del propio nombre de la clase si se realiza desde el exterior a la misma.

Además, dado el carácter de enlace dinámico del lenguaje aparecen operadores "típicos" de la *POO*, como es *instanceof*.

```
<?php
if ($obj instanceof Circulo)
{
    print "$obj es un Círculo";
}
?>
```

2.1.2 Constructores y destructores.

Los constructores y destructores son métodos especiales de las clases, que se corresponden respectivamente al primer método que ejecuta un objeto cuando se instancia, y al último método que se ejecuta cuando se destruye.

Normalmente, los constructores y destructores no se heredan y es necesario incluir una llamada explícita al constructor de la superclase si queremos que ésta se ejecute.

- *PHP4*: El constructor es un método más, denominado con el nombre de la clase y no existen los destructores. En contra de lo que ocurre en otros lenguajes, los constructores y destructores SI se heredan.

- *PHP5*: Se designan métodos especiales para los constructores *__construct*() y destructores *__destruct*(). De esta forma se simplifica la invocación de los constructores de las superclases en los casos de herencia. El destructor es un método que se invoca siempre, justo antes de la desaparición de un objeto. Recordemos, que tanto en *PHP4* como en *PHP5*, un objeto desaparece cuando ya no es referenciado, a través de un sistema interno al intérprete que va liberando memoria cuando nadie tiene acceso al objeto. El caso de *PHP* es complejo, pues el motor interno combina técnicas de recogida de basura similares a las de *Java* mediante su *garbage collector*, con técnicas de liberación explicita de memoria como ocurre en *C#*. El método se utiliza para que los objetos puedan deshacer estructuras complejas de memoria, y evitar así bloqueos de recursos de memoria por referencias cruzadas entre objetos "desechados", mucho más habituales en *PHP4*.

A pesar de que, como hemos dicho anteriormente, *PHP5* no es totalmente compatible con *PHP4*, tampoco podemos afirmar que sean incompatibles, pues el intérprete intenta resolver los conflictos manteniendo en la medida de lo posible la compatibilidad del código.

De hecho, es posible forzar un modo de compatibilidad relativa mediante la inclusión de la siguiente cláusula en el fichero *php.ini*:

`zend.ze1_compatibility_mode` *`boolean`*

Esta opción, habilita el modo de compatibilidad con el Motor Zend 1 (*PHP 4*). Afecta al modo de clonar, moldear y comparar objetos. Es decir, a las famosas referencias. Si queremos que *PHP5* mantenga la máxima compatibilidad posible con el motor zend v1 (*PHP4*) debemos ajustar la directiva anterior en el fichero *php.ini*. El hecho es que, en la primera sesión, establecimos la misma a "off", y es conveniente dejarla así.

En el caso de los constructores, el intérprete de la versión 5, que emplea el *Zend Engine 2*, busca en primer lugar el constructor estándar *__construct*() y si no lo encuentra busca un método cuyo nombre coincide con el de la clase, permitiendo así que las clases creadas para *PHP4* funcionen del mismo modo en que fueron diseñadas.

2.1.3 Abstracción.

La abstracción es una técnica común en programación, que hace hincapié en el ***qué*** más que en el ***cómo***.

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[4]. 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.
    - *PHP4*: No existen.
    - *PHP5*: Aparece la palabra reservada *abstract*, que permite declarar una clase como abstracta. Se antepone al nombre de la clase.

Además, los lenguajes orientados a objetos cuentan con métodos abstractos:

- 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).

---

4 Imaginemos las clases *mamífero* y *ovovivíparo*, de las que heredarán respectivamente las clases *vaca* y *gallina*. Las superclases no tendrán instancias. Sin embargo si pueden tener métodos como “mamar” o “eclosionar”.

- *PHP4*: No existen.

- *PHP5*: Aparece la palabra reservada *abstract*, que permite declarar un método como abstracto. Se antepone al nombre del método.

Finalmente, recordemos algunos conceptos. Una clase puede ser abstracta independientemente de como sean sus métodos. Es decir, cuando decimos que un método es abstracto, significa que no especificaremos ninguna acción, y que dicha acción deberá ser definida en las clases derivadas o subclases. A su vez, un método abstracto sólo puede pertenecer a una clase abstracta, aunque una clase abstracta puede tener métodos no abstractos.

Si en *PHP4* llega a ser necesario el uso de clases abstractas, podemos simularlas mediante la instrucción *die*() en el constructor de una clase normal. De esta forma, la clase no puede ser instanciada, o mejor dicho, al instanciarse el objeto es automáticamente destruido. Mediante este mecanismo, si se desea hacer uso de la clase afectada se estará obligado a redefinirla. De manera similar, si lo que deseamos es simular la existencia de métodos abstractos, podemos hacerlo incluyendo bien la instrucción *die*(), que provocara la finalización de la ejecución, o una inocua instrucción *return*().

2.1.4 Herencia.

La herencia es una característica propia de lenguajes orientados a objetos, que puede definirse como una relación entre clases a través de la cual, una clase (subclase) comparte la estructura y el comportamiento definida en otra (superclase), o varias otras (superclases) si el lenguaje permite la herencia múltiple.

Una subclase hereda de una o más superclases, y aumenta o redefine la estructura y el comportamiento de dichas superclases. Las subclases expresan especialización y las superclases representan generalización. Los constructores no se heredan, son exclusivos de la clase en la que se definen. Un objeto de una subclase siempre incluye en su interior un objeto de la superclase, por lo que puede acceder a los métodos de la clase superior.

Tanto en *PHP4* como en *PHP5* está soportado el mecanismo de herencia a través de la palabra reservada "*extend*". Sin embargo, podemos distinguir algunos matices.

- *PHP4*: Al no existir modificadores de accesos, se heredan y se hacen públicos todos los atributos y métodos, por lo que todo método heredado puede redefinirse.

- *PHP5*: Sólo se heredan los métodos y atributos declarados como públicos (*public*) o protegidos (*protected*). Es decir, los métodos y atributos privados (*private*) no se heredan. Tenemos mayor control sobre la herencia, ya que podemos "esterilizar" una clase si anteponemos la palabra reservada "*final*" a la palabra "*class*" de forma que la clase no pueda tener subclases.

2.1.5 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. Las interfaces permiten resolver los problemas que provoca la herencia múltiple en los lenguajes de programación que no la poseen.

- *PHP4*: No existe

- *PHP5*: Las interfaces se declaran con la palabra reservada "*interface*". Una clase indica las interfaces que soporta o implementa en la cabecera de su declaración, tras el nombre de la clase y mediante la cláusula "*implements*" seguida de la lista de interfaces, que estarán separada por comas.

```php
<?php
interface Display
{
    function display();
}

class Circulo implements Display
{
    function display()
    {
        print "Pinto un círculo";
    }
}
?>
```

2.1.6 Polimorfismo: redefinición, sobrecarga, sobrescritura.

La palabra "polimorfismo" deriva del griego y significa "muchas formas". En programación se refiere a la capacidad de agrupar comportamientos diferentes, que comparten un mismo nombre.

Referido a los lenguajes de programación actuales, abarca distintos aspecto:

- Polimorfismo de inclusión: La capacidad de que una clase derivada pueda hacerse pasar por una superclase. Esto permite definir tratamientos para toda una rama de la jerarquía de herencia.

- Polimorfismo paramétrico o genericidad: La misma función puede aplicarse sobre parámetros de tipos distintos. Suele explicarse con el clásico ejemplo de "*sumar*(x,y)", donde x e y pueden ser enteros, flotantes, cadenas u otros tipos de datos distintos.

- Sobrecarga u "*overloading*": Relacionado con la genericidad, consiste en utilizar el mismo nombre para denotar funciones distintas y utilizar el contexto de las mismas, la clase invocadora o bien los parámetros, para distinguirlas. Normalmente el polimorfismo es sólo aparente, en realidad existen varios métodos definidos y el lenguaje decide cual enlazar, ya sea en tiempo de compilación o dinámicamente si es interpretado.

    - *PHP4*: No la soporta de forma nativa, sólo da soporte al paso de parámetros con valores por defecto, un conjunto de posibilidades mínimo para considerar que se soporta toda la sobrecarga. Sin embargo, si que proporciona mecanismos para que el programador pueda simularlo. Para ello, podemos servirnos de las funciones "*func_num_args*()", "*func_get_args*()" y "*call_user_func_array*()". Existe además un módulo experimental para PHP4, que proporciona cierto tipo de sobrecarga, el módulo "*overload*".

```php
<?php
class Sobrecarga
{

    function metodo()
    {
        $argc = func_num_args(); //nº argumentos recibidos
        $argv = func_get_args(); //Array con el valor de los argumentos
        $metodoReal = "metodo".$argc; //Nombre del método real
        return call_user_func_array (
            array('Sobrecarga', $metodoReal),
            $argv
        );
    }

    function metodo1($x)
    {
        return "Metodo con un argumento: x=$x";
    }

    function metodo2($x, $y)
    {
```

```
        return "Metodo con dos argumentos: x=$x : y=$y";
    }

    function metodo3($x, $y, $z)
    {
        return "Metodo con tres argumentos: x=$x : y=$y : z=$z";
    }
}
?>
```

- *PHP5*: Se proporciona de forma nativa métodos que permiten la sobrecarga, son:

__get(`string nombre`): Se ejecuta de forma automática cuando se accede al valor de un atributo que no existe (no definido en la clase), o no es accesible. Recibe como parámetro el nombre del atributo al que se accede.

__set(`string nombre`): Igual que la anterior, pero cuando se intenta fijar el valor de un atributo que no existe (no definido en la clase), o no es accesible. Recibe como parámetro el nombre del atributo al que se accede.

__isset(`string nombre`): Se ejecuta cuando alguien hace una invocación *isset*() a un atributo que no existe (no definido en la clase), o no es accesible. Recibe como parámetro el nombre del atributo al que se accede.

__unset(`string nombre`): Se ejecuta cuando alguien hace una invocación *unset*() a un atributo que no existe (no definido en la clase), o no es accesible. Recibe como parámetro el nombre del atributo al que se accede.

__call(`string nombre, array argumentos`): Se invoca también de forma automática cuando se intenta invocar a un método que no es accesible o no existe. Como parámetros, recibe el nombre del método invocado y un vector con los argumentos que se pasaron en la llamada.

```
<?php
class Sobrecarga
{

    function __call($nombre, $argv) //Método utilizado cuando el invocado no existe
    {

        $argc = count($argv) //nº argumentos recibidos
        if ($nombre =='metodo')
        {
            switch $argc
            {
                case 1:
                    $this->metodo1($argv[0]);
                break;
                case 1:
                    $this->metodo1($argv[0], $argv[1]);
                break
                case 1:
                    $this->metodo1($argv[0], $argv[1], $argv[2]);
                break
                default:
                    print("El método no existe!!");
            }
        }
```

```
        else
        {
            print("El método no existe!!");
        }
    }

    private function metodo1($x)
    {
        return "Método con un argumento: x=$x";
    }

    private function metodo2($x, $y)
    {
        return " Método con dos argumentos: x=$x : y=$y";
    }

    private function metodo3($x, $y, $z)
    {
        return " Método con tres argumentos: x=$x : y=$y : z=$z";
    }
}
?>
```

- Sobrescritura, redefinición u "*overriding*": Puede verse como una sobrecarga entre clases. Aparece cuando una subclase define un método con el mismo nombre, parámetros y tipo de retorno que la clase padre, por lo que redefine el comportamiento del mismo. Es habitual en la herencia. Recordemos:

    - *PHP4*: Al no haber modificadores de accesos se heredan y se hacen públicos todos los atributos y métodos, por lo que todo método heredado puede redefinirse, basta con rescribirlo en la clase derivada.

    - *PHP5*: Sólo se heredan los métodos y atributos declarados como públicos (*public*) o protegidos (*protected*). Es decir, los métodos y atributos privados (*private*) no se heredan, por lo que no pueden sobrescribirse, pues no son accesibles. Tenemos mayor control sobre la herencia. Por un lado, podemos "esterilizar" una clase, si anteponemos la palabra reservada "*final*" a la palabra "*class*", la clase no podrá tener subclases.

2.1.7 Sistemas de "antialias" o clonado.

El clonado es la manera de expresar cuando un objeto se convierte en una copia de otro. El problema del clonado surge en los lenguajes *OO* a partir del momento en que un objeto puede contener a otro y las relaciones entre ellos funcionan como referencias. Debido a ello, los lenguajes *OO* suelen incorporar métodos para llevar a cabo la copia.

- *PHP4*: Debido a que los objetos NO funcionan como referencias, cuando un objeto de asigna a otro y no se emplea el operador referencia "&" lo que se obtiene es una copia superficial del objeto. Decimos copia superficial, porque si por ejemplo, lo que asignamos es una lista enlazada, se llevará a cabo una copia del

objeto cabeza, pero los objetos que forman el resto de la lista serán comunes.

- *PHP5*: Al incluir el sistema de referencias, el clonado se hace de forma explicita a través del método especial "*__clone()*". Al invocarse dicho método realiza una copia superficial del objeto bit a bit. Si tratamos con objetos que manejan estructuras complejas en memoria, deberemos sobrescribir el método para realizar una copia en mayor profundidad.

2.2 Manejo de errores y excepciones.

La gestión de errores viene normalmente en una fase de depuración de código, en la cual ya no aparece esa "magia" que le gusta al programador. La depuración y manejo de errores y excepciones es un trabajo que suele ser menos "querido" por los programadores. Si a esto añadimos que el soporte proporcionado por *PHP* de forma histórica no ha sido, como en la mayoría lenguajes procedurales, demasiado amigable, encontramos un panorama bastante decepcionante, en el que muchas aplicaciones no gestionan los errores de forma correcta, bien por desidia, bien por una arquitectura errónea. Por ejemplo:

```php
<?php

class LectorFichero
{
    var $fichero;
    var $directoriroFichero='./';

    function fijarNombreFichero($fichero)
    {
        $this->fichero = $fichero;
    }

    function obtenerContenido()
    {
        return file_get_contents("{$this->directoriroFichero}{$this->fichero}.php");
    }
}

// invocación con un fichero inexistente
$lf = new LectorFichero('no_existe');
print($lf->obtenerContenido());

?>
```

EJECUCIÓN

```
    Warning: file_get_contents(...): failed to open stream: No such file or directory in
/ruta_fichero/script.php on line 3.
```

2.2.1 Sistema procedural.

PHP3 y *PHP4* incorporan de forma progresiva características de lenguajes *OO*. Sin embargo, y a pesar de la evolución, ambos mantienen un sistema de manejo de errores demasiado simple, propio de los primeros lenguajes procedurales. Esta ha sido la causa principal por la cual, si estudiamos código *PHP* de distintos proyectos, podemos encontrar distintas formas para resolver las cuestiones de manejo de

errores. Heredado de los lenguajes procedurales, podemos usar el valor de retorno de las funciones, siendo estos de tipo lógico (*true*/*false*) o enteros. Así podemos indicar si la ejecución ha sido correcta o no:

```php
<?php

class LectorFichero
{
    var $fichero;
    var $directoriroFichero='./';

    function fijarNombreFichero($fichero)
    {
        if(!@file_exists("{$this->directoriroFichero}{$this->fichero}.php")
            return false;
        $this->fichero=$fichero;
        return true;
    }

    function obtenerContenido()
    {
        if (!@$contenido = file_get_contents("{$this->directoriroFichero}{$this->fichero}.php"))
            return false;
        return $contenido;
    }
}

// invocación con un fichero inexistente
$lf = new LectorFichero('no_existe');
if(!$lf->obtenerContenido())
    print('Fichero no encontrado');
else
    print($lf->obtenerContenido());
?>
```

EJECUCIÓN

```
Fichero no encontrado
```

2.2.2 Sistemas de abandono.

Un sistema parecido es bloquear errores críticos con la terminación del programa mediante instrucciones *die*() o *exit*() acompañada de algún escueto mensaje:

```php
<?php

class LectorFichero
{
    var $fichero;
    var $directoriroFichero='./';

    function fijarNombreFichero($fichero)
    {
        if(!file_exists("{$this->directoriroFichero}{$this->fichero}.php")
            die('Fichero '.$fichero.' no encontrado');
        $this->fichero = $fichero;
    }

    function obtenerContenido()
    {
        return file_get_contents("{$this->directoriroFichero}{$this->fichero}.php");
    }
}

// invocación con un fichero inexistente
$lf = new LectorFichero('no_existe');
print($lf->obtenerContenido());
?>
```

EJECUCIÓN

```
Fichero no_existe no encontrado
```

2.2.3 Sistema de captura de errores en *PHP4*.

La asistencia a la gestión de errores en *PHP4* no es muy amigable, pero es bastante correcta. En la versión 4 de *PHP* se nos proporciona las funciones *trigger_error*() y *set_error_handler*(). Con dichas funciones podemos capturar errores y asignar manejadores a los mismos. En el siguiente código, podemos ver el ejemplo anterior adaptado:

```php
<?php

class LectorFichero
{
    var $fichero;
    var $directoriroFichero='./';

    function fijarNombreFichero($fichero)
    {
        if( !file_exists("{$this->directoriroFichero}{$this->fichero}.php")
            trigger_error('Fichero '.$fichero.' no encontrado',E_USER_WARNING);
        $this->fichero=$fichero;
    }

    function obtenerContenido()
    {
        if (!$contenido = file_get_contents("{$this->directoriroFichero}{$this->fichero}.php"))
            trigger_error("No se puede leer el contenido del fichero",E_USER_WARNING);
        return $contenido;
    }
}

function manejadorErrorLectorFichero($numErr,$mensajeErr,$fichero,$numLinea)
{
    if($numErr == E_USER_WARNING)
    {
        echo 'Error: '.$mensajeErr.'<br />Fichero: '.$fichero.'<br />Línea: '.$numLinea;
        exit();
    }
}

// definimos el manejador de errores
set_error_handler('manejadorErrorLectorFichero');
// invocación con un fichero inexistente
$lf = new LectorFichero('no_existe');
print($lf->obtenerContenido());

?>
```

EJECUCIÓN

```
Error: File no_existe_file not found
File: /ruta/script.php
Line: 6
```

En muchos entornos *PHP4* donde se intentaba sacar partido a sus características orientadas a objetos optaron por utilizar la clase base de *PEAR* PEAR_Error. Toda clase *PEAR* al producirse un error debe generar[5] un objeto *PEAR_Error* con información sobre lo ocurrido. *PEAR_Error* consta de los métodos siguientes:

addUserInfo	añade información de usuario
getCallback	el nombre de la función de callback

5 Para construir una clase que sea "compatible" PEAR debe cumplir ciertas condiciones.

getCode	obtienen el código de error
getMessage	obtienen el mensaje de error
getMode	obtiene el modo de error
getDebugInfo	obtiene información de depuración
getType	obtienen el tipo de error
getUserInfo	obtiene información adicional proporcionada por el usuario
PEAR_Error	Constructor
toString	crea una representación de cadena

Además, para poder asignar un manejador de errores, se facilitan los métodos *PEAR::isError*(), *PEAR::raiseError*() y PEAR::*setErrorHandling*(). Veamos de nuevo nuestro ejemplo:

```php
<?php
require_once("PEAR.php");
class LectorFichero
{
    var $fichero;
    var $directoriroFichero='./';

    function fijarNombreFichero($fichero)
    {
        if( !file_exists("{$this->directoriroFichero}{$this->fichero}.php")
            return PEAR::RaiseError('Fichero '.$fichero.' no encontrado');

        $this->fichero = $fichero;
    }

    function obtenerContenido()
    {
        if (!$contenido = file_get_contents("{$this->directoriroFichero}{$this->fichero}.php"))
            PEAR::RaiseError('No se puede leer el contenido del fichero');
        return $contenido;
    }
}

// invocación con un fichero inexistente
$lf = new LectorFichero('no_existe');
print($lf->obtenerContenido());
if (PEAR::isError($lf))
{
    echo $lf->getMessage()."\n";
    exit();
}
?>
```

A pesar de las utilidades *PEAR*, el sistema sufre algunos inconvenientes y es más una forma de normalizar el tratamiento de errores en *PHP* que una solución óptima.

2.2.4 Sistemas de excepciones de *PHP5*.

Con *PHP5* hay un cambio importante en la gestión de errores. Como en la mayoría de lenguajes orientado a objetos, se introduce un sistema de manejo de excepciones integrado en el motor del lenguaje. El modelo elegido es la terna *try*/*catch*/*throw*, similar a Java. Las sentencias *try*/*catch* pueden incluirse en cualquier parte del código.

```
class LectorFichero
{
    private $fichero;
    private $directoriroFichero='./';

    public function __construct($fichero)
    {
        if(!file_exists("{$this->directoriroFichero}{$this->fichero}.php")
            throw new Exception('Fichero '.$fichero.' no encontrado');

        $this->fichero=$fichero;
    }

    public function obtenerContenido()
    {
        if (!$contenido = file_get_contents("{$this->directoriroFichero}{$this->fichero}.php"))
            throw new Exception('No se puede leer el contenido del fichero');
        return $contenido;
    }
}
// invocación con un fichero inexistente
try
{
    $lf = new LectorFichero('no_existe');
    print($lf->obtenerContenido());
}
catch(Exception $e)
{
    echo $e->getMessage()."\n";
    echo "Código: ".$e->getCode()."\n";
    echo "Fichero: ".$e->getFile()."\n";
    echo "Línea: ".$e->getLine()."\n";
    exit();
}
```

EJECUCIÓN

```
Fichero no_existe no encontrado
Código: XXX
Fichero: XXX
Línea: XXX
```

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[6].

```
class LectorFichero
{
    private $fichero;
    private $directoriroFichero='./';

    public function __construct($fichero)
```

6 La función error_log [php.net/manual/es/function.error-log.php] de la que se hace uso en el ejemplo presentado, registra los mensajes en el fichero de log del servidor Web.

```
    {
        if(!file_exists("{$this->directoriroFichero}{$this->fichero}.php")
            throw new Exception('Fichero '.$fichero.' no encontrado');

        $this->fichero=$fichero;
    }


    public function obtenerContenido()
    {
        if (!$contenido = file_get_contents("{$this->directoriroFichero}{$this->fichero}.php"))
            throw new Exception('No se puede leer el contenido del fichero');

        return $contenido;
    }
}
// invocación con un fichero inexistente
try
{
    $lf = new LectorFichero('no_existe');
    print($lf->obtenerContenido());
}
catch(Exception $e)
{
    echo $e->getMessage()."\n";
    echo "Código: ".$e->getCode()."\n";
    echo "Fichero: ".$e->getFile()."\n";
    echo "Línea: ".$e->getLine()."\n";
    exit();
}
```

EJECUCIÓN

```
Fichero no_existe no encontrado
Código: XXX
Fichero: XXX
Línea: XXX
```

También podemos crear una clase *Exception* derivada con la herencia e incluir en ella la funcionalidad que creamos conveniente. Si por ejemplo quisiéramos construir un sistema que auditase la ejecución de una aplicación, tendríamos que pasar por declarar nuestras clases de excepciones.

```
class LectorFichero
{
    private $fichero;
    private $directoriroFichero='./';
    const ERROR_FICHERO = 1;
    const ERROR_LECTURA_FICHERO = 2;

    public function __construct($fichero)
    {
        if(!file_exists("{$this->directoriroFichero}{$this->fichero}.php")
            throw new ExcepcionLector('Fichero '.$fichero.' no encontrado', self::ERROR_FICHERO);

        $this->fichero=$fichero;
    }


    public function obtenerContenido()
    {
        if (!$contenido = file_get_contents("{$this->directoriroFichero}{$this->fichero}.php"))
            throw new ExcepcionLector('No se puede leer el contenido del fichero',
self::ERROR_LECTURA_FICHERO);

        return $contenido;
    }
}

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

```php
// invocación con un fichero inexistente
try
{
    $lf = new LectorFichero('no_existe');
    print($lf->obtenerContenido());
}
catch(ExcepcionLector $e)
{
    echo $e->registrarError();
}
catch(Exception $e)
{
    echo $e->getMessage();
    exit();
}
```

2.3 Otras características de *PHP5*.

2.3.1 Iteradores.

Los iteradores son otra de las novedades que tiene *PHP5*. Podemos verlos como una forma de sobrecarga de clases, que permite manejar los objetos de dichas clases a modo de cursor. Existen distintas formas de implementar la iteración, la primera y más sencilla es a través del operador "*foreach*", que existía tanto en *PHP4* como en *PHP5* y nos permite tratar un objeto como una matriz asociativa recorriendo todos sus atributos públicos.

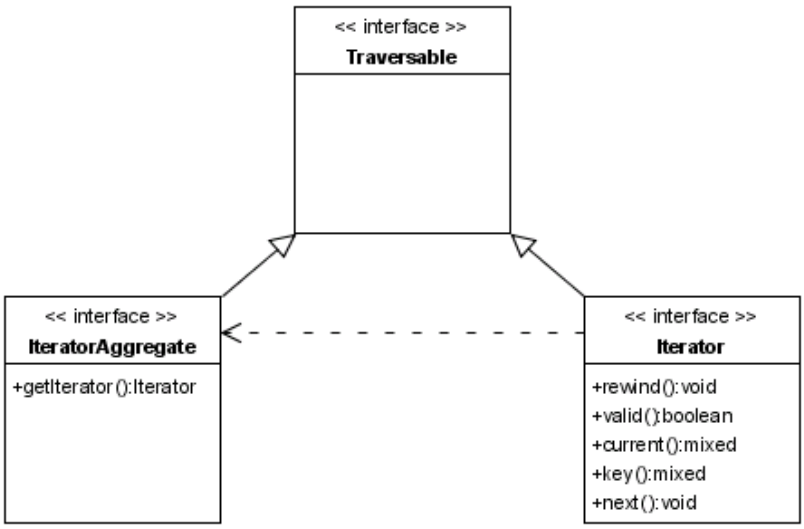
```php
<?php
class EjemploIteracion1
{
    var $uno = 1;
    var $dos = 2;
    var $tres = 3;
    var $cuatro = 4;
}

$ej1 = new EjemploIteracion1();
foreach ($ej1 as $clave=>$valor)
{
    print('Clave: '.$clave.' Valor: '.$valor."\n");
}
//Salida:
//Clave: uno Valor: 1 Clave: dos Valor: 2 Clave: tres Valor: 3 Clave: cuatro Valor: 4
?>
```

La única novedad que aporta *PHP5* en esta cláusula es el paso por referencia. Para ello, incluiremos el operador & al realizar la construcción, y de esta manera podremos modificar los atributos del objeto.

```php
<?php
$ej2 = new EjemploIteracion2();
foreach ($ej2 as $clave=>&$valor)
{
    print('Clave: '.$clave.' Valor: '.$valor."\n");
    $valor = 0; //Anulamos el valor
}
print("\n<br/>\n");print("anulados");print("\n<br/>\n");
foreach ($ej2 as $clave=>$valor)
{
    print('Clave: '.$clave.' Valor: '.$valor."\n");
}
//Salida:
//Clave: uno Valor: 1 Clave: dos Valor: 2 Clave: tres Valor: 3 Clave: cuatro Valor: 4
//anulados
//Clave: uno Valor: 0 Clave: dos Valor: 0 Clave: tres Valor: 0 Clave: cuatro Valor: 0
?>
```

Pero, junto con *PHP5*, se proporciona una librería interna, la "Standard PHP Library" [php.net/manual/es/ref.spl.php], que incluye entre otras muchas cosas tres interfaces, "*Traversable*", "*Iterator*" e "*IteratorAgregate*". Estas interfaces implementan el patrón de iteración de la siguiente figura:



Veamos el ejemplo anterior adaptado al uso de este patrón:

```php
<?php
class EjemploIteracion3 implements Iterator, Traversable
{
    private $elementos;
    private $max;
    private $actual;


    function __construct()
    {
        $this->elementos = "uno,dos,tres,cuatro,cinco";
        $this->max = count(explode(',',$this->elementos));
        $this->actual = 0;
    }

    public function rewind()
    {
        $this->actual = 0;
    }

    public function valid()
    {
        return ($this->actual<$this->max);
    }

    public function key()
    {
        return ($this->actual);
    }

    public function current()
    {
        $v_elementos = explode(',',$this->elementos);
        return ($v_elementos[$this->actual]);
    }

    public function next()
    {
        $this->actual++;
    }
}

$ej3 = new EjemploIteracion3();
foreach ($ej3 as $clave=>$valor)
{
```

```
    print('Clave: '.$clave.' Valor: '.$valor."\n");
}
//Salida:
//Clave: 0 Valor: uno Clave: 1 Valor: dos Clave: 2 Valor: tres Clave: 3 Valor: cuatro Clave: 4
Valor: cinco
?>
```

2.3.2 Otras novedades.

Entre otros:

- *__autoload*(): Permite trabajar de forma más cómoda con las inclusiones de ficheros *PHP*. En lugar de incluir en la cabecera de un fichero PHP todas las librerías que puede utilizar en un momento dado (*include, include_once, require, require_once*) se puede definir esta función *__autoload*(), y cada vez que una clase vaya a utilizarse, podremos incluir el fichero correspondiente en el cuerpo de este método.

- Soporte nativo (integrado en *PHP5*) para el manejo de *XML*, con soporte para validar con *DTDs*, *XSD* (*SchemaXML*) y *RelaxNG:*

    - La interfaz *SAX*

    - La interfaz *DOM*

    - La interfaz propia de *PHP*, *SimpleXML*.

- Soporte nativo para transformaciones *XSLT.*

- Soporte nativo para implementar Servicios Web:

    - *SOAP*

    - *XMLRPC*

- Soporte de acceso a distintos sistemas de gestión de base de datos: *MySQLi* (*MySQL improved*) Extensión, *SQLite* y capa de abstracción de acceso a Bds integrada.

- Clase *Reflection* para tareas de debug o ingeniería inversa.

- Llamada a métodos que devuelven objetos de forma consecutiva (*obj->metodo1()->metodo2()->*...).

- Soporte nativo para streaming.

- Librería estándar de interfaces. SPL [php.net/~helly/php/ext/spl/].

- Mayor potencia de su interfaz *CLI* (plasmada en proyectos como PHP-GTK2 [gtk.php.net]).

- Mejora en la gestión de recursos de memoria.

Para más información: http://www.php.net/manual/es/

2.4 Lo que viene, *PHP6*.

PHP6 comienza a gestarse a partir del 11 de Noviembre de 2005, cuando los principales desarrolladores de *PHP* redactan un documento [php.net] conjunto tras una reunión en París, con lo que iba a incluirse en la evolución del *PHP*. A continuación, presentamos un resumen de dicho documento.

El objetivo principal de la versión *PHP6* es el soporte completo al sistema de codificación de caracteres internacional *UNICODE* [es.wikipedia.com], de vital importancia para cualquier proyecto que pretenda soportar multitud de idiomas o trabajar completamente con *XML*.

Ojo, no queremos decir, que no puedan abordarse este tipo de proyectos con *PHP4* o *PHP5*. La versión *PHP5* actual presenta un soporte correcto a *unicode* que puede activarse bajo petición, pero que sin embargo, afecta al rendimiento del sistema, pues lo que se hace es duplicar el almacenaje de clases, funciones, etc., que pasan a almacenarse en la codificación original, y en *unicode*.

El resto de cambios que se esperan, se encaminan a eliminar malos hábitos heredados de versiones iniciales del lenguaje, por ejemplo:

- No podrán registrarse las variables globales de ninguna forma (recordemos que en las últimas versiones, su uso estaba desaconsejado, pero podía activarse modificando el fichero "php.ini".

- Se eliminará el modo seguro o "safe_dir". El modo seguro, según algunos el mal llamado modo seguro, no ha proporcionado seguridad "real". Fuente de distintos agujeros de seguridad que permitía inyección de código en el servidor, por lo que se elimina y la seguridad de acceso al sistema de ficheros del servidor pasará a ser responsabilidad del instalador de *Apache* y *PHP*.

- Se eliminará la opción de "magic_quotes", dicha directiva realiza el escapado automático de las variables *$_GET*, *$_POST* y *$_COOKIE*, añadiendo la barra invertida o contrabarra "\" antes de cada comilla simple, doble o NULL. Este funcionamiento "automático" ha facilitado la inyección de código *SQL* en muchas aplicaciones. Se deja la resolución de estas situaciones al programador, que puede servirse de multitud de funciones nativas para evitarlo.

- Se eliminará el modo de compatibilidad con *PHP4*, a cambio, se relajarán algunas restricciones sintácticas del lenguaje. Por ejemplo, se equiparará el uso de "public" al de "var", evitando así el

"warning" que produce su uso en *PHP5*. Sin embargo, otras opciones de compatibilidad desaparecerán, como *HTTP_GET_VARS* y *HTTP_POST_VARS*, que pasarán a ser definitivamente *$_GET* y *$_POST*.

- Se abandona el soporte de algunas librerías externas obsoletas, como "freetype1" o la librería gráfica "gd1", y otras extensiones se mueven al proyecto *PECL*, por ejemplo, "ereg", lo que agiliza su desarrollo e independencia con las librerías subyacentes.

Estas características pueden llegar antes de la versión PHP6, por ejemplo, en reciente versión de PHP 5.3, se han incorporado funcionalidades que estaban previstas para la versión 6, por ejemplo:

- Los bloques "Nowdoc" para trabajar con cadenas de gran longitud que no requieren de ningún tipo de procesamiento evitando escapar caracteres especiales.

- Clausuras o funciones anónimas.

- Puesta en funcionamiento opcional para el recolector de basura con referencias circulares.

- Sistemas de empaquetado de aplicación similar a los ficheros WAR de J2EE.

- Los espacios de nombres o "namespaces". Cuando recurrimos a librerías externas, es fácil que existan clases cuyo nombre coincida con alguna de nuestras clases. Esto era un verdadero problema en PHP4 y PHP5, pues obligaba a renombrar nuestras clases con algún prefijo ocurrente para no tener problemas, la solución adoptada es similar a la que utilizan otros lenguajes como .Net o Java, la palabra reservada para ello, es "namespace".

```php
<?php

/**
 * Clase /mis/clases/MiClase.php
*/

namespace mis\clases;

class MiClase
{
    ....
}

// Podemos definir constantes o funciones
//propias del espacio de nombres
function miFunción()
{
    ....
}

const MI_CONSTANTE = 'HOLA';

?>
```

- La forma de utilizar las distintas clases, es a través del nuevo operador “use”, que permite crear alias en nuestro código para espacios de nombre largos.

```php
<?php

use mis::clases as mc;
print ('Mi constante ' mc::MI_CONSTANTE);

?>
```

- Se resuelve el problema de enlazado estático tardío o “Late Static Binding”, un problema que aparece al realizar invocaciones de métodos de clase heredados, es decir, herencia con métodos estáticos de una clase.

- Mejoras en el soporte a la internacionalización de proyectos.

Si el alumno tiene curiosidad por probar alguna de estas novedades, puede descargar versiones alfa de las futuras versiones de PHP, tanto de *PHP6*) como de *PHP 5.3.x* en: http://snaps.php.net/.

Además si se quiere conocer los aspectos y funcionalidades que serán finalmente incluidos en *PHP6*, puede visitarse la página web http://wiki.pooteeweet.org/PhP60.