Programación Orientada a Objetos

POO

Índice

1.- Concepto de programación orientada a objetos (POO)

Este tutorial tiene por objetivo el mostrar en forma sencilla la Programación Orientada a Objetos utilizando como lenguaje el PHP 5.
Se irán introduciendo conceptos de objeto, clase, atributo, método etc. y de todos estos temas se irán planteando problemas resueltos que pueden ser modificados. También es muy importante tratar de resolver los problemas propuestos.
Prácticamente todos los lenguajes desarrollados en los últimos 15 años implementan la posibilidad de trabajar con POO (Programación Orientada a Objetos)
El lenguaje PHP tiene la característica de permitir programar con las siguientes metodologías:

- Programación Lineal: Es cuando desarrollamos todo el código disponiendo instrucciones PHP alternando con el HTML de la página.
- Programación Estructurada: Es cuando planteamos funciones que agrupan actividades a desarrollar y luego dentro de la página llamamos a dichas funciones que pueden estar dentro del mismo archivo o en una librería separada.
- Programación Orientada a Objetos: Es cuando planteamos clases y definimos objetos de las mismas (Este es el objetivo del tutorial, aprender la metodología de programación orientada a objetos y la sintaxis particular de PHP 5 para la POO)

Conceptos básicos de Objetos

Un objeto es una entidad independiente con sus propios datos y programación. Las ventanas, menús, carpetas de archivos pueden ser identificados como objetos; el motor de un automóvil también es considerado un objeto, en este caso, sus datos (atributos) describen sus características físicas y su programación (métodos) describen el funcionamiento interno y su interrelación con otras partes del automóvil (también objetos).
El concepto renovador de la tecnología Orientación a Objetos es la suma de funciones a elementos de datos, a esta unión se le llama encapsulamiento. Por ejemplo, un objeto página contiene las dimensiones físicas de la página (ancho, alto), el color, el estilo del borde, etc, llamados atributos. Encapsulados con estos datos se encuentran los métodos para modificar el tamaño de la página, cambiar el color, mostrar texto, etc. La responsabilidad de un objeto pagina consiste en realizar las acciones apropiadas y mantener actualizados sus datos internos. Cuando otra parte del programa (otros objetos) necesitan que la pagina realice alguna de estas tareas (por ejemplo, cambiar de color) le envía un mensaje. A estos objetos que envían mensajes no les interesa la manera en que el objeto página lleva a cabo sus tareas ni las estructuras de datos que maneja, por ello, están ocultos. Entonces, un objeto contiene información pública, lo que necesitan los otros objetos para interactuar con él e información privada, interna, lo que necesita el objeto para operar y que es irrelevante para los otros objetos de la aplicación.

2- Declaración de una clase y creación de un objeto.

La programación orientada a objetos se basa en la programación de clases; a diferencia de la programación estructurada, que está centrada en las funciones.
Una clase es un molde del que luego se pueden crear múltiples objetos, con similares características.
Un poco más abajo se define una clase Persona y luego se crean dos objetos de dicha clase.
Una clase es una plantilla (molde), que define atributos (lo que conocemos como variables) y métodos (lo que conocemos como funciones).
La clase define los atributos y métodos comunes a los objetos de ese tipo, pero luego, cada objeto tendrá sus propios valores y compartirán las mismas funciones.
Debemos crear una clase antes de poder crear objetos (instancias) de esa clase. Al crear un objeto de una clase, se dice que se crea una instancia de la clase o un objeto propiamente dicho.
Confeccionaremos nuestra primer clase para conocer la sintaxis en el lenguaje PHP, luego definiremos dos objetos de dicha clase.
Implementaremos una clase llamada Persona que tendrá como atributo (variable) su nombre y dos métodos (funciones), uno de dichos métodos inicializará el atributo nombre y el siguiente método mostrará en la página el contenido del mismo.

La sintaxis básica para declarar una clase es:

```
class [Nombre de la Clase] {
  [atributos]
  [métodos]
}
```

Siempre conviene buscar un nombre de clase lo más próximo a lo que representa. La palabra clave para declarar la clase es class, seguidamente el nombre de la clase y luego encerramos entre llaves de apertura y cerrado todos sus atributos(variable) y métodos(funciones).

Nuestra clase Persona queda definida entonces:

```
class Persona {
  private $nombre;
  public function inicializar($nom)
  {
            $this->nombre=$nom;


  }
  public function imprimir()
  {
    echo $this->nombre;
    echo '<br>';
  }
}
```

Los atributos normalmente son privados (private), ya veremos que esto significa que no podemos acceder al mismo desde fuera de la clase. Luego para definir los métodos se utiliza la misma sintaxis que las funciones del lenguaje PHP.
Decíamos que una clase es un molde que nos permite definir objetos. Ahora veamos cual es la sintaxis para la definición de objetos de la clase Persona:

```
$per1=new Persona();
$per1->inicializar('Juan');
$per1->imprimir();
```

Definimos un objeto llamado $per1 y lo creamos asignándole lo que devuelve el operador new. Siempre que queremos crear un objeto de una clase utilizamos la sintaxis new [Nombre de la Clase].
Luego para llamar a los métodos debemos anteceder el nombre del objeto el operador -> y por último el nombre del método. Para poder llamar al método,éste debe ser definido público (con la palabra clave

public). En el caso que tenga parámetros se los enviamos:

```
$per1->inicializar('Juan');
```

También podemos ver que podemos definir tantos objetos de la clase Persona como sean necesarios para nuestro algoritmo:

```
$per2=new Persona();
$per2->inicializar('Ana');
$per2->imprimir();
```

Esto nos da una idea que si en una página WEB tenemos 2 menús, seguramente definiremos una clase Menu y luego crearemos dos objetos de dicha clase.

Esto es una de las ventajas fundamentales de la Programación Orientada a Objetos (POO), es decir reutilización de código (gracias a que está encapsulada en clases) es muy sencilla.
Lo último a tener en cuenta en cuanto a la sintaxis de este primer problema es que cuando accedemos a los atributos dentro de los métodos debemos utilizar los operadores $this-> (this y ->):

```
  public function inicializar($nom)
  {
    $this->nombre=$nom;
  }
```

El atributo $nombre solo puede ser accedido por los métodos de la clase Persona.

Ejemplo

Confeccionar una clase llamada Persona. Definir un atributo donde se almacene su nombre. Luego definir dos métodos, uno que cargue el nombre y otro que lo imprima.

```
<html>
<head>
<title>Pruebas</title>
</head>
<body>
<?php
class Persona {
  private $nombre;
  public function inicializar($nom)
  {
    $this->nombre=$nom;
  }
  public function imprimir()
  {
    echo $this->nombre;
    echo '<br>';
  }
}

$per1=new Persona();
$per1->inicializar('Juan');
$per1->imprimir();
$per2=new Persona();
$per2->inicializar('Ana');
$per2->imprimir();
?>
</body>
</html>
```

Problema:

Confeccionar una clase Empleado, definir como atributos su nombre y sueldo. Definir un método inicializar que lleguen como dato el nombre y sueldo. Plantear un segundo método que imprima el nombre y un mensaje si debe o no pagar impuestos (si el sueldo supera a 3000 paga impuestos)

3 - Atributos de una clase.

Ahora trataremos de concentrarnos en los atributos de una clase. Los atributos son las características, cualidades, propiedades distintivas de cada clase. Contienen información sobre el objeto. Determinan la apariencia, estado y demás particularidades de la clase. Varios objetos de una misma clase tendrán los mismos atributos pero con valores diferentes.

Cuando creamos un objeto de una clase determinada, los atributos declarados por la clase son localizadas en memoria y pueden ser modificados mediante los métodos.

Lo más conveniente es que los atributos sean privados para que solo los métodos de la clase puedan modificarlos.

Plantearemos un nuevo problema para analizar detenidamente la definición, sintaxis y acceso a los atributos.

Problema: Implementar una clase que muestre una lista de hipervínculos en forma horizontal (básicamente un menú de opciones)

Lo primero que debemos pensar es que valores almacenará la clase, en este caso debemos cargar una lista de direcciones web y los títulos de los enlaces. Podemos definir dos vectores paralelos que almacenen las direcciones y los títulos respectivamente.

Definiremos dos métodos: cargarOpcion y mostrar.

pagina1.php

```
<html>
<head>
<title>Pruebas</title>
</head>
<body>
<?php
class Menu {
  private $enlaces=array();
  private $titulos=array();
  public function cargarOpcion($en,$tit)
  {
    $this->enlaces[]=$en;
    $this->titulos[]=$tit;
  }
  public function mostrar()
  {
    for($f=0;$f<count($this->enlaces);$f++)
    {
      echo '<a href="'.$this->enlaces[$f].'">'.$this->titulos[$f].'</a>';
      echo "-";
    }
  }
}

$menu1=new Menu();
$menu1->cargarOpcion('http://www.google.com','Google');
$menu1->cargarOpcion('http://www.yahoo.com','Yhahoo');
$menu1->cargarOpcion('http://www.msn.com','MSN');
$menu1->mostrar();
?>
</body>
</html>
```

Analicemos ahora la solución al problema planteado, como podemos ver normalmente los atributos de la clase se definen inmediatamente después que declaramos la clase:

```
class Menu {
  private $enlaces=array();
  private $titulos=array();
```

Si queremos podemos hacer un comentario indicando el objetivo de cada atributo.
Luego tenemos el primer método que añade a los vectores los datos que llegan como parámetro:

```
  public function cargarOpcion($en,$tit)
  {
    $this->enlaces[]=$en;
    $this->titulos[]=$tit;
  }
```

Conviene darle distinto nombre a los parámetros y los atributos (por lo menos inicialmente para no confundirlos).
Utilizamos la característica de PHP que un vector puede ir creciendo solo con asignarle el nuevo valor. El dato después de esta asignación $this->enlaces[]=$en; se almacena al final del vector.
Este método será llamado tantas veces como opciones tenga el menú.
El siguiente método tiene por objetivo mostrar el menú propiamente dicho:

```
  public function mostrar()
  {
    for($f=0;$f<count($this->enlaces);$f++)
    {
      echo '<a href="'.$this->enlaces[$f].'">'.$this->titulos[$f].'</a>';
      echo "-";
    }
  }
```

Disponemos un for y hacemos que se repita tantas veces como elementos tenga el vector $enlaces (es lo mismo preguntar a uno u otro cuantos elementos tienen ya que siempre tendrán la misma cantidad). Para obtener la cantidad de elementos del vector utilizamos la función count.
Dentro del for imprimimos en la página el hipervínculo:

```
      echo '<a href="'.$this->enlaces[$f].'">'.$this->titulos[$f].'</a>';
```

Hay que acostumbrarse que cuando accedemos a los atributos de la clase se le antecede el operador $this-> y seguidamente el nombre del atributo propiamente dicho. Si no hacemos esto estaremos creando una variable local y el algoritmo fallará.
Por último para hacer uso de esta clase Menu debemos crear un objeto de dicha clase (lo que en programación estructurada es definir una variable):

```
$menu1=new Menu();
$menu1->cargarOpcion('http://www.google.com','Google');
$menu1->cargarOpcion('http://www.yahoo.com','Yhahoo');
$menu1->cargarOpcion('http://www.msn.com','MSN');
$menu1->mostrar();
```

Creamos un objeto mediante el operador new y seguido del nombre de la clase. Luego llamamos al método cargarOpcion tantas veces como opciones necesitemos para nuestro menú (recordar que SOLO podemos llamar a los métodos de la clase si definimos un objeto de la misma)
Finalmente llamamos al método mostrar que imprime en la página nuestro menú.

Problema:

Confeccionar una clase Menu. Permitir añadir la cantidad de opciones que necesitemos. Mostrar el menú en forma horizontal o vertical, según que método llamemos.

4.- Métodos de una clase.

Los métodos son como las funciones en los lenguajes estructurados, pero están definidos dentro de una clase y operan sobre los atributos de dicha clase.
Los métodos también son llamados las responsabilidades de la clase. Para encontrar las responsabilidades de una clase hay que preguntarse qué puede hacer la clase.
El objetivo de un método es ejecutar las actividades que tiene encomendada la clase a la cual pertenece.
Los atributos de un objeto se modifican mediante llamadas a sus métodos.
Confeccionaremos un nuevo problema para concentrarnos en la definición y llamada a métodos.

Problema: Confeccionar una clase CabeceraPagina que permita mostrar un título, indicarle si queremos que aparezca centrado, a derecha o izquierda.

Definiremos dos atributos, uno donde almacenar el título y otro donde almacenar la ubicación.
Ahora pensemos que métodos o responsabilidades debe tener esta clase para poder mostrar la cabecera de la página. Seguramente deberá tener un método que pueda inicializar los atributos y otro método que muestre la cabecera dentro de la página.
Veamos el código de la clase CabeceraPagina

```
<html>
<head>
<title>Pruebas</title>
</head>
<body>
<?php
class CabeceraPagina {
  private $titulo;
  private $ubicacion;
  public function inicializar($tit,$ubi)
  {
    $this->titulo=$tit;
    $this->ubicacion=$ubi;
  }
  public function mostrar()
  {
    echo '<div style="font-size:40px;text-align:'.$this->ubicacion.'">';
    echo $this->titulo;
    echo '</div>';
  }
}

$cabecera=new CabeceraPagina();
$cabecera->inicializar('El blog del programador','center');
$cabecera->mostrar();
?>
</body>
</html>
```

La clase CabeceraPagina tiene dos atributos donde almacenamos el texto que debe mostrar y la ubicación del mismo ('center', 'left' o 'right'), nos valemos de CSS para ubicar el texto en la página:

```
  private $titulo;
  private $ubicacion;
```

Ahora analicemos lo que más nos importa en el concepto que estamos concentrados (métodos de una clase):

```
  public function inicializar($tit,$ubi)
  {
    $this->titulo=$tit;
    $this->ubicacion=$ubi;
  }
```

Un método hasta ahora siempre comienza con la palabra clave public (esto significa que podemos llamarlo desde fuera de la clase, con la única salvedad que hay que definir un objeto de dicha clase)
Un método tiene un nombre, conviene utilizar verbos para la definición de métodos (mostrar, inicializar, mostrar etc.) y sustantivos para la definición de atributos ($color, $enlace, $titulo etc)
Un método puede tener o no parámetros. Generalmente los parámetros inicializan atributos del objeto:

```
    $this->titulo=$tit;
```

Luego para llamar a los métodos debemos crear un objeto de dicha clase:

```
$cabecera=new CabeceraPagina();
$cabecera->inicializar('El blog del programador','center');
$cabecera→mostrar();
```

Es importante notar que siempre que llamamos a un método le antecedemos el nombre del objeto. El orden de llamada a los métodos es importante, no va a funcionar si primero llamamos a mostrar y luego llamamos al método inicializar.

Problema:

Confeccionar una clase CabeceraPagina que permita mostrar un título, indicarle si queremos que aparezca centrado, a derecha o izquierda, además permitir definir el color de fondo y de la fuente.

5 - Método constructor de una clase (__construct)

El constructor es un método especial de una clase. El objetivo fundamental del constructor es inicializar los atributos del objeto que creamos.
Básicamente el constructor remplaza al método inicializar que habíamos hecho en el concepto anterior.
Las ventajas de implementar un constructor en lugar del método inicializar son:

1. El constructor es el primer método que se ejecuta cuando se crea un objeto.
2. El constructor se llama automáticamente. Es decir es imposible de olvidarse llamarlo ya que se llamará automáticamente.
3. Quien utiliza POO (Programación Orientada a Objetos) conoce el objetivo de este método.

Otras características de los c}onstructores son:

- El constructor se ejecuta inmediatamente luego de crear un objeto y no puede ser llamado nuevamente.
- Un constructor no puede retornar dato.
- Un constructor puede recibir parámetros que se utilizan normalmente para inicializar atributos.
- El constructor es un método opcional, de todos modos es muy común definirlo.

Veamos la sintaxis del constructor:

```
  public function __construct([parámetros])
  {
    [algoritmo]
  }
```

Debemos definir un método llamado __construct (es decir utilizamos dos caracteres de subrayado y la palabra construct). El constructor debe ser un método público (public function).
Además hemos dicho que el constructor puede tener parámetros.
Confeccionaremos el mismo problema del concepto anterior para ver el cambio que debemos hacer de ahora en más.

Problema: Confeccionar una clase CabeceraPagina que permita mostrar un título, indicarle si queremos que aparezca centrado, a derecha o izquierda.

```
<html>
<head>
<title>Pruebas</title>
</head>
<body>
<?php
class CabeceraPagina {
  private $titulo;
  private $ubicacion;
  public function __construct($tit,$ubi)
  {
    $this->titulo=$tit;
    $this->ubicacion=$ubi;
  }
  public function mostrar()
  {
    echo '<div style="font-size:40px;text-align:'.$this->ubicacion.'">';
    echo $this->titulo;
    echo '</div>';
  }
}

$cabecera=new CabeceraPagina('El blog del programador','center');
$cabecera->mostrar();
?>
```

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

Ahora podemos ver como cambió la sintaxis para la definición del constructor:

```
  public function __construct($tit,$ubi)
  {
    $this->titulo=$tit;
    $this->ubicacion=$ubi;
  }
```

Hay que tener mucho cuidado cuando definimos el constructor, ya que el más mínimo error (nos olvidamos un carácter de subrayado, cambiamos una letra de la palabra construct) nuestro algoritmo no funcionará correctamente ya que nunca se ejecutará este método (ya que no es el constructor).

Veamos como se modifica la llamada al constructor cuando se crea un objeto:

```
$cabecera=new CabeceraPagina('El blog del programador','center');
$cabecera->mostrar();
```

Es decir el constructor se llama en la misma línea donde creamos el objeto, por eso disponemos después del nombre de la clase los parámetros:

```
$cabecera=new CabeceraPagina('El blog del programador','center');
```

Generalmente todo aquello que es de vital importancia para el funcionamiento inicial del objeto se lo pasamos mediante el constructor.

Problema:

Confeccionar una clase CabeceraPagina que permita mostrar un título, indicarle si queremos que aparezca centrado, a derecha o izquierda, además permitir definir el color de fondo y de la fuente. Pasar los valores que cargaran los atributos mediante un constructor.

6 - Llamada de métodos dentro de la clase.

Hasta ahora todos los problemas planteados hemos llamado a los métodos desde donde definimos un objeto de dicha clase, por ejemplo:

```
$cabecera=new CabeceraPagina('El blog del programador','center');
$cabecera->mostrar();
```

Utilizamos la sintaxis:

```
[nombre del objeto]->[nombre del método]
```

Es decir antecedemos al nombre del método el nombre del objeto y el operador ->
Ahora bien que pasa si queremos llamar dentro de la clase a otro método que pertenece a la misma clase, la sintaxis es la siguiente:
$this->[nombre del método]

Es importante tener en cuenta que esto solo se puede hacer cuando estamos dentro de la misma clase.
Confeccionaremos un problema que haga llamadas entre métodos de la misma clase.

Problema: Confeccionar una clase Tabla que permita indicarle en el constructor la cantidad de filas y columnas. Definir otra responsabilidad que podamos cargar un dato en una determinada fila y columna. Finalmente debe mostrar los datos en una tabla HTML.

```
<html>
<head>
<title>Pruebas</title>
</head>
<body>
<?php
class Tabla {
  private $mat;
  private $cantFilas;
  private $cantColumnas;

  public function __construct($fi,$co)
  {
    $this->cantFilas=$fi;
    $this->cantColumnas=$co;
            $this->mat=array();
  }

  public function cargar($fila,$columna,$valor)
  {
    $this->mat[$fila][$columna]=$valor;
  }

  public function inicioTabla()
  {
    echo '<table border="1">';
  }

  public function inicioFila()
  {
    echo '<tr>';
  }

  public function mostrarCelda($fi,$co)
  {
    echo '<td>'.$this->mat[$fi][$co].'</td>';
  }
```

```
  public function finFila()
  {
    echo '</tr>';
  }

  public function finTabla()
  {
    echo '</table>';
  }

  public function mostrar()
  {
    $this->inicioTabla();
    for($f=1;$f<=$this->cantFilas;$f++)
    {
      $this->inicioFila();
      for($c=1;$c<=$this->cantColumnas;$c++)
      {
        $this->mostrarCelda($f,$c);
      }
      $this->finFila();
    }
    $this->finTabla();
  }
}

$tabla1=new Tabla(2,3);
$tabla1->cargar(1,1,"1");
$tabla1->cargar(1,2,"2");
$tabla1->cargar(1,3,"3");
$tabla1->cargar(2,1,"4");
$tabla1->cargar(2,2,"5");
$tabla1->cargar(2,3,"6");
$tabla1->mostrar();
?>
</body>
</html>
```

Vamos por partes, primero veamos los tres atributos definidos,el primero se trata de un array donde almacenaremos todos los valores que contendrá la tabla HTML y otros dos atributos que indican la dimensión de la tabla HTML (cantidad de filas y columnas):

```
  private $mat=array();
  private $cantFilas;
  private $cantColumnas;
```

El constructor recibe como parámetros la cantidad de filas y columnas que tendrá la tabla:

```
  public function __construct($fi,$co)
  {
    $this->cantFilas=$fi;
    $this->cantColumnas=$co;
  }
```

Otro método de vital importancia es el de cargar datos. Llegan como parámetro la fila, columna y dato a almacenar:

```
  public function cargar($fila,$columna,$valor)
  {
    $this->mat[$fila][$columna]=$valor;
  }
```

Otro método muy importante es el mostrar:

```
  public function mostrar()
  {
    $this->inicioTabla();
    for($f=1;$f<=$this->cantFilas;$f++)
    {
      $this->inicioFila();
      for($c=1;$c<=$this->cantColumnas;$c++)
      {
        $this->mostrarCelda($f,$c);
      }
      $this->finFila();
    }
    $this->finTabla();
  }
```

El método mostrar debe hacer las salidas de datos dentro de una tabla HTML. Para simplificar el algoritmo definimos otros cinco métodos que tienen por objetivo hacer la generación del código HTML propiamente dicho. Así tenemos el método inicioTabla que hace la salida de la marca table e inicialización del atributo border:

```
  public function inicioTabla()
  {
    echo '<table border="1">';
  }
```

De forma similar los otros métodos son:

```
  public function inicioFila()
  {
    echo '<tr>';
  }

  public function mostrarCelda($fi,$co)
  {
    echo '<td>'.$this->mat[$fi][$co].'</td>';
  }

  public function finFila()
  {
    echo '</tr>';
  }

  public function finTabla()
  {
    echo '</table>';
  }
```

Si bien podíamos hacer todo esto en el método mostrar y no hacer estos cinco métodos, la simplicidad del código aumenta a mediada que subdividimos los algoritmos. Esto es de fundamental importancia a medida que los algoritmos sean más complejos.

Lo que nos importa ahora ver es como llamamos a métodos que pertenecen a la misma clase:

```
public function mostrar()
{
  $this->inicioTabla();
  for($f=1;$f<=$this->cantFilas;$f++)
  {
    $this->inicioFila();
    for($c=1;$c<=$this->cantColumnas;$c++)
    {
      $this->mostrarCelda($f,$c);
    }
    $this->finFila();
  }
  $this->finTabla();
}
```

Es decir le antecedemos el operador $this-> al nombre del método a llamar. De forma similar a como accedemos a los atributos de la clase.
Por último debemos definir un objeto de la clase Tabla y llamar a los métodos respectivos:

```
$tabla1=new Tabla(2,3);
$tabla1->cargar(1,1,"1");
$tabla1->cargar(1,2,"2");
$tabla1->cargar(1,3,"3");
$tabla1->cargar(2,1,"4");
$tabla1->cargar(2,2,"5");
$tabla1->cargar(2,3,"6");
$tabla1->mostrar();
```

Es importante notar que donde definimos un objeto de la clase Tabla no llamamos a los métodos inicioTabla(), inicioFila(), etc.

Problema:

Confeccionar una clase Tabla que permita indicarle en el constructor la cantidad de filas y columnas. Definir otra responsabilidad que podamos cargar un dato en una determinada fila y columna además de definir su color de fuente y fondo. Finalmente debe mostrar los datos en una tabla HTML.

7 - Modificadores de acceso a atributos y métodos (public - private)

Hasta ahora hemos dicho que los atributos conviene definirlos con el modificador private y los métodos los hemos estado haciendo a todos public.

Analicemos que implica disponer un atributo privado (private), en el concepto anterior definíamos dos atributos para almacenar la cantidad de filas y columnas en la clase Tabla:

```
 private $cantFilas;
 private $cantColumnas;
```

Al ser privados desde fuera de la clase no podemos modificarlos:

```
$tabla1->cantFilas=20;
```

El resultado de ejecutar esta línea provoca el siguiente error:

Fatal error: Cannot access private property Tabla::$cantFilas

No olvidemos entonces que los atributos los modificamos llamando a un método de la clase que se encarga de inicializarlos (en la clase Tabla se inicializan en el constructor):

```
$tabla1=new Tabla(2,3);
```

Ahora vamos a extender este concepto de modificador de acceso a los métodos de la clase. Veíamos hasta ahora que todos los métodos planteados de la clase han sido públicos. Pero en muchas situaciones conviene que haya métodos privados (private).

Un método privado (private) solo puede ser llamado desde otro método de la clase. No podemos llamar a un método privados desde donde definimos un objeto.

Con la definición de métodos privados se elimina la posibilidad de llamar a métodos por error, consideremos el problema del concepto anterior (clase Tabla) donde creamos un objeto de dicha clase y llamamos por error al método finTabla:

```
$tabla1=new Tabla(2,3);
$tabla1->finTabla();
$tabla1->cargar(1,1,"1");
$tabla1->cargar(1,2,"2");
$tabla1->cargar(1,3,"3");
$tabla1->cargar(2,1,"4");
$tabla1->cargar(2,2,"5");
$tabla1->cargar(2,3,"6");
$tabla1->mostrar();
```

Este código produce un error lógico ya que al llamar al método finTabla() se incorpora al archivo HTML la marca </html>.

Este tipo de error lo podemos evitar definiendo el método finTabla() con modificador de acceso private:

```
  private function finTabla()
  {
     echo '</table>';
  }
```

Luego si volvemos a ejecutar:

```
$tabla1=new Tabla(2,3);
$tabla1->finTabla();
$tabla1->cargar(1,1,"1");
```

...

Se produce un error sintáctico:

Fatal error: Call to private method Tabla::finTabla()

Entonces el modificador private nos permite ocultar en la clase atributos y métodos que no queremos que los accedan directamente quien definen objetos de dicha clase. Los métodos públicos es aquello que queremos que conozcan perfectamente las personas que hagan uso de nuestra clase (también llamada interfaz de la clase).

Nuestra clase Tabla ahora correctamente codificada con los modificadores de acceso queda:

```
<html>
<head>
<title>Pruebas</title>
</head>
<body>
<?php
class Tabla {
  private $mat=array();
  private $cantFilas;
  private $cantColumnas;
  public function __construct($fi,$co)
  {
    $this->cantFilas=$fi;
    $this->cantColumnas=$co;
  }

  public function cargar($fila,$columna,$valor)
  {
    $this->mat[$fila][$columna]=$valor;
  }

  private function inicioTabla()
  {
    echo '<table border="1">';
  }

  private function inicioFila()
  {
    echo '<tr>';
  }

  private function mostrarCelda($fi,$co)
  {
    echo '<td>'.$this->mat[$fi][$co].'</td>';
  }

  private function finFila()
  {
    echo '</tr>';
  }

  private function finTabla()
  {
    echo '</table>';
  }

  public function mostrar()
  {
    $this->inicioTabla();
    for($f=1;$f<=$this->cantFilas;$f++)
    {
      $this->inicioFila();
      for($c=1;$c<=$this->cantColumnas;$c++)
      {
        $this->mostrarCelda($f,$c);
      }
      $this->finFila();
    }
    $this->finTabla();
  }
}
```

```
$tabla1=new Tabla(2,3);
$tabla1->cargar(1,1,"1");
$tabla1->cargar(1,2,"2");
$tabla1->cargar(1,3,"3");
$tabla1->cargar(2,1,"4");
$tabla1->cargar(2,2,"5");
$tabla1->cargar(2,3,"6");
$tabla1->mostrar();
?>
</body>
</html>
```

Tenemos tres métodos públicos:

```
  public function __construct($fi,$co)
  public function cargar($fila,$columna,$valor)
  public function mostrar()
```

Y cinco métodos privados:

```
  private function inicioTabla()
  private function inicioFila()
  private function mostrarCelda($fi,$co)
  private function finFila()
  private function finTabla()
```

Tengamos en cuenta que cuando definimos un objeto de la clase Tabla solo podemos llamar a los métodos públicos. Cuando documentamos una clase debemos hacer mucho énfasis en la descripción de los métodos públicos, que serán en definitiva los que deben llamarse cuando definamos objetos de dicha clase.
Uno de los objetivos fundamentales de la POO es el encapsulamiento. El encapsulamiento es una técnica por el que se ocultan las características internas de una clase de todos aquellos elementos (atributos y métodos) que no tienen porque conocerla otros objetos. Gracias al modificador private podemos ocultar las características internas de nuestra clase.
Cuando uno plantea una clase debe poner mucha atención cuales responsabilidades (métodos) deben ser públicas y cuales responsabilidades no queremos que las conozcan los demás.

Problema:

Confeccionar una clase Menu. Permitir añadir la cantidad de opciones que necesitemos. Mostrar el menú en forma horizontal o vertical, pasar a este método como parámetro el texto "horizontal" o "vertical". El método mostrar debe llamar alguno de los dos métodos privados mostrarHorizontal() o mostrarVertical().

8 - Colaboración de objetos.

Hasta ahora nuestros ejemplos han presentado una sola clase, de la cual hemos definido uno o varios objetos. Pero una aplicación real consta de muchas clases.

Veremos que hay dos formas de relacionar las clases. La primera y la que nos concentramos en este concepto es la de COLABORACIÓN.

Cuando dentro de una clase definimos un atributo o una variable de otra clase decimos que esta segunda clase colabora con la primera. Cuando uno ha trabajado por muchos años con la metodología de programación estructurada es difícil subdividir un problema en clases, tiende a querer plantear una única clase que resuelva todo.

Presentemos un problema:Una página web es común que contenga una cabecera, un cuerpo y un pié de página. Estas tres secciones podemos perfectamente identificarlas como clases. También podemos identificar otra clase pagina. Ahora bien como podemos relacionar estas cuatro clases (pagina, cabecera, cuerpo, pie), como podemos imaginar la cabecera, cuerpo y pie son partes de la pagina. Luego podemos plantear una clase pagina que contenga tres atributos de tipo objeto.

En forma simplificada este problema lo podemos plantear así:

```
class Cabecera {
  [atributos y métodos]
}
class Cuerpo {
  [atributos y métodos]
}
class Pie {
  [atributos y métodos]
}
class Pagina {
  private $cabecera;
  private $cuerpo;
  private $pie;
  [métodos]
}

$pag=new Pagina();
```

Como podemos ver declaramos cuatro clases (Cabecera, Cuerpo, Pie y Pagina), fuera de cualquier clase definimos un objeto de la clase Pagina:

```
$pag=new Pagina();
```

Dentro de la clase Pagina definimos tres atributos de tipo objeto de las clases Cabecera, Cuerpo y Pie respectivamente. Luego seguramente dentro de la clase Pagina crearemos los tres objetos y llamaremos a sus métodos respectivos.

Veamos una implementación real de este problema:

```
<html>
<head>
<title>Pruebas</title>
</head>
<body>
<?php
class Cabecera {
  private $titulo;
  public function __construct($tit)
  {
    $this->titulo=$tit;
  }
  public function mostrar()
  {
    echo '<h1 style="text-align:center">'.$this->titulo.'</h1>';
  }
}
```

```
class Cuerpo {
  private $lineas=array();
  public function insertarParrafo($li)
  {
    $this->lineas[]=$li;
  }
  public function mostrar()
  {
    for($f=0;$f<count($this->lineas);$f++)
    {
      echo '<p>'.$this->lineas[$f].'</p>';
    }
  }
}

class Pie {
  private $titulo;
  public function __construct($tit)
  {
    $this->titulo=$tit;
  }
  public function mostrar()
  {
    echo '<h4 style="text-align:left">'.$this->titulo.'</h4>';
  }
}

class Pagina {
  private $cabecera;
  private $cuerpo;
  private $pie;
  public function __construct($texto1,$texto2)
  {
    $this->cabecera=new Cabecera($texto1);
    $this->cuerpo=new Cuerpo();
    $this->pie=new Pie($texto2);
  }
  public function insertarCuerpo($texto)
  {
    $this->cuerpo->insertarParrafo($texto);
  }
  public function mostrar()
  {
    $this->cabecera->mostrar();
    $this->cuerpo->mostrar();
    $this->pie->mostrar();
  }
}

$pagina1=new Pagina('Título de la Página','Pie de la página');
      $pagina1->insertarCuerpo('Esto es una prueba que debe aparecer dentro del
cuerpo de la página 1');
$pagina1->insertarCuerpo('Esto es una prueba que debe aparecer dentro del cuerpo
de la página 2');
$pagina1->insertarCuerpo('Esto es una prueba que debe aparecer dentro del cuerpo
de la página 3');
$pagina1->insertarCuerpo('Esto es una prueba que debe aparecer dentro del cuerpo
de la página 4');
$pagina1->insertarCuerpo('Esto es una prueba que debe aparecer dentro del cuerpo
de la página 5');
```

```
$pagina1->insertarCuerpo('Esto es una prueba que debe aparecer dentro del cuerpo
de la página 6');
$pagina1->insertarCuerpo('Esto es una prueba que debe aparecer dentro del cuerpo
de la página 7');
$pagina1->insertarCuerpo('Esto es una prueba que debe aparecer dentro del cuerpo
de la página 8');
$pagina1->insertarCuerpo('Esto es una prueba que debe aparecer dentro del cuerpo
de la página 9');
$pagina1->mostrar();
?>
</body>
</html>
```

La primer clase llamada Cabecera define un atributo llamada $titulo que se carga en el constructor, luego se define otro método que imprime el HTML:

```
class Cabecera {
  private $titulo;
  public function __construct($tit)
  {
    $this->titulo=$tit;
  }
  public function mostrar()
  {
    echo '<h1 style="text-align:center">'.$this->titulo.'</h1>';
  }
}
```

La clase Pie es prácticamente idéntica a la clase Cabecera, solo que cuando genera el HTML lo hace con otro tamaño de texto y alineado a izquierda:

```
class Pie {
  private $titulo;
  public function __construct($tit)
  {
    $this->titulo=$tit;
  }
  public function mostrar()
  {
    echo '<h4 style="text-align:left">'.$this->titulo.'</h4>';
  }
}
```

Ahora la clase Cuerpo define un atributo de tipo array donde se almacenan todos los párrafos. Esta clase no tiene constructor, sino un método llamado insertarParrafo que puede ser llamado tantas veces como párrafos tenga el cuerpo de la página. Esta actividad no la podríamos haber hecho en el constructor ya que el mismo puede ser llamado solo una vez.
Luego el código de la clase Cuerpo es:

```
class Cuerpo {
  private $lineas=array();
  public function insertarParrafo($li)
  {
    $this->lineas[]=$li;
  }
  public function mostrar()
  {
    for($f=0;$f<count($this->lineas);$f++)
    {
      echo '<p>'.$this->lineas[$f].'</p>';
    }
  }
}
```

Para mostrar todos los párrafos mediante una estructura repetitiva disponemos cada uno de los elementos del atributo $lineas dentro de las marcas y .

Ahora la clase que define como atributos objetos de la clase Cabecera, Cuerpo y Pie es la clase Pagina:

```
class Pagina {
  private $cabecera;
  private $cuerpo;
  private $pie;
  public function __construct($texto1,$texto2)
  {
    $this->cabecera=new Cabecera($texto1);
    $this->cuerpo=new Cuerpo();
    $this->pie=new Pie($texto2);
  }
  public function insertarCuerpo($texto)
  {
    $this->cuerpo->insertarParrafo($texto);
  }
  public function mostrar()
  {
    $this->cabecera->mostrar();
    $this->cuerpo->mostrar();
    $this->pie->mostrar();
  }
}
```

Al constructor llegan dos cadenas con las que inicializamos los atributos $cabecera y $pie:

```
    $this->cabecera=new Cabecera($texto1);
    $this->cuerpo=new Cuerpo();
    $this->pie=new Pie($texto2);
```

Al atributo $cuerpo también lo creamos pero no le pasamos datos ya que dicha clase no tiene constructor con parámetros.
La clase Pagina tiene un método llamado:

```
  public function insertarCuerpo($texto)
```

que tiene como objetivo llamar al método insertarParrafo del objeto $cuerpo.

El método mostrar de la clase Pagina llama a los métodos mostrar de los objetos Cabecera, Cuerpo y Pie en el orden adecuado:

```
  public function mostrar()
  {
    $this->cabecera->mostrar();
    $this->cuerpo->mostrar();
    $this->pie->mostrar();
  }
```

Finalmente hemos llegado a la parte del algoritmo donde se desencadena la creación del primer objeto, definimos un objeto llamado $pagina1 de la clase Pagina y le pasamos al constructor los textos a mostrar en la cabecera y pie de pagina, seguidamente llamamos al método insertarCuerpo tantas veces como información necesitemos incorporar a la parte central de la página. Finalizamos llamando al método mostrar:

```
$pagina1=new Pagina('Título de la Página','Pie de la página');
$pagina1->insertarCuerpo('Esto es una prueba que debe aparecer dentro del cuerpo
de la página 1');
$pagina1->insertarCuerpo('Esto es una prueba que debe aparecer dentro del cuerpo
de la página 2');
$pagina1->insertarCuerpo('Esto es una prueba que debe aparecer dentro del cuerpo
de la página 3');
$pagina1->insertarCuerpo('Esto es una prueba que debe aparecer dentro del cuerpo
de la página 4');
$pagina1->insertarCuerpo('Esto es una prueba que debe aparecer dentro del cuerpo
de la página 5');
$pagina1->insertarCuerpo('Esto es una prueba que debe aparecer dentro del cuerpo
de la página 6');
$pagina1->insertarCuerpo('Esto es una prueba que debe aparecer dentro del cuerpo
de la página 7');
$pagina1->insertarCuerpo('Esto es una prueba que debe aparecer dentro del cuerpo
de la página 8');
$pagina1->insertarCuerpo('Esto es una prueba que debe aparecer dentro del cuerpo
de la página 9');
$pagina1->mostrar();
```

Problema:

Confeccionar la clase Tabla vista en conceptos anteriores. Plantear una clase Celda que colabore con la clase Tabla. La clase Tabla debe definir una matriz de objetos de la clase Celda.
La clase Celda debe definir los atributos: $texto, $colorFuente y $colorFondo.

9 - Parámetros de tipo objeto.

Otra posibilidad que nos presenta el lenguaje PHP es pasar parámetros no solo de tipo primitivo (enteros, reales, cadenas etc.) sino parámetros de tipo objeto.
Vamos a desarrollar un problema que utilice esta característica. Plantearemos una clase Opcion y otra clase Menu. La clase Opcion definirá como atributos el titulo, enlace y color de fondo, los métodos a implementar serán el constructor y el mostrar
Por otro lado la clase Menu administrará un array de objetos de la clase Opcion e implementará un métodos para insertar objetos de la clase Menu y otro para mostrar. Al constructor de la clase Menu se le indicará si queremos el menú en forma 'horizontal' o 'vertical'.

El código fuente de las dos clases es (pagina1.php):

```
<html>
<head>
<title>Pruebas</title>
</head>
<body>
<?php
class Opcion {
  private $titulo;
  private $enlace;
  private $colorFondo;
  public function __construct($tit,$enl,$cfon)
  {
    $this->titulo=$tit;
    $this->enlace=$enl;
    $this->colorFondo=$cfon;
  }
  public function mostrar()
  {
    echo '<a style="background-color:'.$this->colorFondo.
         '" href="'.$this->enlace.'">'.$this->titulo.'</a>';
  }
}

class Menu {
  private $opciones=array();
  private $direccion; //orientacion del menú horizontal o vertical
  public function __construct($dir)
  {
    $this->direccion=$dir;
  }
  public function insertar($op)
  {
    $this->opciones[]=$op; //$op ES UN OBJETO
  }
  private function mostrarHorizontal()
  {
    for($f=0;$f<count($this->opciones);$f++)
    {
      $this->opciones[$f]->mostrar();
    }
  }
  private function mostrarVertical()
  {
    for($f=0;$f<count($this->opciones);$f++)
    {
      $this->opciones[$f]->mostrar();
      echo '<br>';
    }
  }
```

```
  public function mostrar()
  {
    if (strtolower($this->direccion)=="horizontal")
      $this->mostrarHorizontal();
    else
      if (strtolower($this->direccion)=="vertical")
        $this->mostrarVertical();
  }
}

$menu1=new Menu('horizontal');
$opcion1=new Opcion('Google','http://www.google.com','#C3D9FF');
$menu1->insertar($opcion1);
$opcion2=new Opcion('Yahoo','http://www.yahoo.com','#CDEB8B');
$menu1->insertar($opcion2);
$opcion3=new Opcion('MSN','http://www.msn.com','#C3D9FF');
$menu1->insertar($opcion3);
$menu1->mostrar();
?>
</body>
</html>
```

La clase Opcion define tres atributos donde se almacenan el título del hipervínculo, direccón y el color de fondo del enlace:

```
  private $titulo;
  private $enlace;
  private $colorFondo;
```

En el constructor recibimos los datos con los cuales inicializamos los atributos:

```
  public function __construct($tit,$enl,$cfon)
  {
    $this->titulo=$tit;
    $this->enlace=$enl;
    $this->colorFondo=$cfon;
  }
```

Por último en esta clase el método mostrar muestra el enlace de acuerdo al contenido de los atributos inicializados previamente en el constructor:

```
  public function mostrar()
  {
    echo '<a style="background-color:'.$this->colorFondo.'" href="'.
          $this->enlace.'">'.$this->titulo.'</a>';
  }
```

La clase Menu recibe la colaboración de la clase Opcion. En esta clase definimos un array de objetos de la clase Opcion (como vemos podemos definir perfectamente vectores con componente de tipo objeto), además almacena la dirección del menú (horizontal,vertical):

```
  private $opciones=array();
  private $direccion;
```

El constructor de la clase Menu recibe la dirección del menú e inicializa el atributo $direccion:

```
  public function __construct($dir)
  {
    $this->direccion=$dir;
  }
```

Luego tenemos el método donde se encuentra el concepto nuevo:

```
  public function insertar($op)
  {
    $this->opciones[]=$op;
  }
```

El método insertar llega un objeto de la clase Opcion (previamente creado) y se almacena en una componente del array. Este método tiene un parámetro de tipo objeto ($op es un objeto de la clase Menu)

Luego la clase Menu define dos métodos privados que tienen por objetivo pedir que se grafique cada una de las opciones:

```
  private function mostrarHorizontal()
  {
    for($f=0;$f<count($this->opciones);$f++)
    {
      $this->opciones[$f]->mostrar();
    }
  }
  private function mostrarVertical()
  {
    for($f=0;$f<count($this->opciones);$f++)
    {
      $this->opciones[$f]->mostrar();
      echo '<br>';
    }
  }
```

Los algoritmos solo se diferencian en que el método mostrarVertical añade el elemento
 después de cada opción.
Queda en la clase Menú el método público mostrar que tiene por objetivo analizar el contenido del atributo $direccion y a partir de ello llamar al método privado que corresponda:

```
  public function mostrar()
  {
    if (strtolower($this->direccion)=="horizontal")
      $this->mostrarHorizontal();
    else
      if (strtolower($this->direccion)=="vertical")
        $this->mostrarVertical();
  }
```

El código donde definimos y creamos los objetos es:

```
$menu1=new Menu('horizontal');
$opcion1=new Opcion('Google','http://www.google.com','#C3D9FF');
$menu1->insertar($opcion1);
$opcion2=new Opcion('Yahoo','http://www.yahoo.com','#CDEB8B');
$menu1->insertar($opcion2);
$opcion3=new Opcion('MSN','http://www.msn.com','#C3D9FF');
$menu1->insertar($opcion3);
$menu1->mostrar();
```

Primero creamos un objeto de la clase Menu y le pasamos al constructor que queremos implementar un menú 'horizontal': $menu1=new Menu('horizontal');
Creamos seguidamente un objeto de la clase Opcion y le pasamos como datos al constructor el titulo, enlace y color de fondo:

```
$opcion1=new Opcion('Google','http://www.google.com','#C3D9FF');
```

Seguidamente llamamos al método insertar del objeto menu1 y le pasamos como parámetro un objeto de la clase Menu (es decir pasamos un objeto por lo que el parámetro del método insertar debe recibir la referencia a dicho objeto):

```
$menu1->insertar($opcion1);
```

Luego creamos tantos objetos de la clase Opcion como opciones tenga nuestro menú, y llamamos también sucesivamente al método insertar del objeto $menu1. Finalmente llamamos al método mostrar del objeto $menu1.

Problema:

Confeccionar la clase Tabla vista en conceptos anteriores. Plantear una clase Celda que colabore con la clase Tabla. La clase Tabla debe definir una matriz de objetos de la clase Celda.
En la clase Tabla definir un método insertar que llegue un objeto de la clase Celda y además dos enteros que indiquen que posición debe tomar dicha celda en la tabla.
La clase Celda debe definir los atributos: $texto, $colorFuente y $colorFondo.

Type hinting

El **Type hinting**, o la **implicación de tipos**, permite determinar el tipo de **parámetro** que un **método** va a devolver: **objeto**, **array**, **interfaz** o **callable**.
¿Cuándo utilizar type hinting? Cuando queremos **forzar a una función a aceptar sólo argumentos de un tipo**. Su uso más común es con objetos, indicando el nombre de la clase: No añade ningún tipo de funcionalidad, pero es muy útil por ejemplo para saber si cometes algún error al añadir argumentos con valores incorrectos.
En el siguiente ejemplo al argumento de la función *listadoCochesEnVenta(Venta $venta)* se le exige que sea una instancia de la clase **Venta**. Si se pasara cualquier otro argumento, daría error:

```
class Venta {
    public $coche = 'Audi';
}

class Coche {
    public function listadoCochesEnVenta(Venta $venta)
    {
            echo $venta->coche;
    }
}
$venta = new Venta;
$coche = new Coche;
$coche->listadoCochesEnVenta($venta); // Devuelve Audi
```

Si se pusiera, por ejemplo *$coche->listadoCochesEnVenta('Audi')* daría error, ya que no es una clase instanciada.

Otro ejemplo

```
class Aguacate {
    public $calorias = 100;
}
class Persona {
    public $energia;
    public function comerAguacate (Aguacate $aguacate) {
        $this->energia += $aguacate->calorias;
    }
    public function mostrarEnergia(){
        return "Energía actual: " . $this->energia . "<br>";
    }
}
$sonia = new Persona;
$almuerzo = new Aguacate;
$sonia->comerAguacate($almuerzo);
$sonia->comerAguacate($almuerzo);
echo $sonia->mostrarEnergia(); // Devuelve: Energía actual 200
```

En el ejemplo anterior, hemos forzado a que el argumento que se pasa a la función *comerAguacate()* sea un objeto de la clase **Aguacate**. Si el *$almuerzo* no fuera un objeto de la clase Aguacate, no se podría emplear la función *comerAguacate()*. Si por ejemplo en el ejemplo anterior cambiamos el almuerzo por un **string** kiwi:

```
$almuerzo = "Kiwi";
$sonia->comerAguacate($almuezo);
// Devuelve: Catchable fatal error: Argument 1 passed to
Persona::comerAguacate() must be an instance of Aguacate, string given
```

Es necesario crear una instancia de Aguacate para que *comerAguacate()* acepte el argumento.
Vamos a poner **otro ejemplo**, ahora queremos que el argumento que se acepte sea del tipo **array**:

```
class Coche {
    public function listadoCochesEnVenta(Array $listado)
    {
        foreach ($listado as $coche){
            echo $coche . "\n";
        }
    }
}
$listaCoches = ['Audi', 'BMW'];

$coche = new Coche;
$coche->listadoCochesEnVenta($listaCoches); // Devuelve el listado Audi BMW
```

Si el argumento de *listadoCochesEnVenta()* no fuera un array, también daría error

Otro ejemplo de arrays:

```
class Persona {
    public $energiaMedia;
    public $comidas;
    public function calcularEnergia(array $comidas)
    {
        $numeroComidas = count($comidas);
        $energiaTotal = $numeroComidas * $this->energiaMedia;
        return $energiaTotal;
    }
}
$sonia = new Persona;
// Energía media que proporciona cada comida:
$sonia->energiaMedia = 480;
// Total de comidas:
$comidas = array("Desayuno", "Almuerzo", "Comida", "Cena");
// Calcular la energía:
echo $sonia->calcularEnergia($comidas); // Devuelve 1920
```

La función *calcularEnergia()* exige que el argumento sea un **array**. Si cambiamos el argumento y ponemos un **integer**, dará error:

```
$comidas = 4;
echo $sonia->calcularEnergia($comidas);
// Catchable fatal error: Argument 1 passed to Persona::calcularEnergia() must
be of the type array, integer given
```

Problema 2:

Partiendo del problema anterior, cambialó para que los métodos utilizados sólo puedan recibir objetos de tipo Celda o enteros en el caso correspondiente.

10 - Parámetros opcionales.

Esta característica está presente tanto para programación estructurada como para programación orientada a objetos. Un parámetro es opcional si en la declaración del método le asignamos un valor por defecto. Si luego llamamos al método sin enviarle dicho valor tomará el que tiene por defecto.

Con un ejemplo se verá más claro: Crearemos nuevamente la clase CabeceraDePagina que nos muestre un título alineado con un determinado color de fuente y fondo.

```
<html>
<head>
<title>Pruebas</title>
</head>
<body>
<?php
class CabeceraPagina {
   private $titulo;
   private $ubicacion;
   private $colorFuente;
   private $colorFondo;
   public function __construct($tit,$ubi='center',$colorFuen='#ffffff',$colorFon='#000000')
   {
      $this->titulo=$tit;
      $this->ubicacion=$ubi;
      $this->colorFuente=$colorFuen;
      $this->colorFondo=$colorFon;
   }
   public function mostrar()
   {
      echo '<div style="font-size:40px;text-align:'.$this->ubicacion.';color:';
      echo $this->colorFuente.';background-color:'.$this->colorFondo.'">';
      echo $this->titulo;
      echo '</div>';
   }
}

$cabecera1=new CabeceraPagina('El blog del programador');
$cabecera1->mostrar();
echo '<br>';
$cabecera2=new CabeceraPagina('El blog del programador','left');
$cabecera2->mostrar();
echo '<br>';
$cabecera3=new CabeceraPagina('El blog del programador','right','#ff0000');
$cabecera3->mostrar();
echo '<br>';
$cabecera4=new CabeceraPagina('El blog del
programador','right','#ff0000','#ffff00');
$cabecera4->mostrar();
?>
</body>
</html>
```

En esta clase hemos planteado parámetros opcionales en el constructor:

```
   public function __construct($tit,$ubi='center',$colorFuen='#ffffff',$colorFon='#000000')
   {
      $this->titulo=$tit;
      $this->ubicacion=$ubi;
      $this->colorFuente=$colorFuen;
      $this->colorFondo=$colorFon;
   }
```

El constructor tiene 4 parámetros, uno obligatorio y tres opcionales, luego cuando lo llamemos al crear un objeto de esta clase lo podemos hacer de la siguiente forma:

```
$cabecera1=new CabeceraPagina('El blog del programador');
```

En este primer caso el parámetro $tit recibe el string 'El blog del programador' y los siguientes tres parámetros se inicializan con los valores por defecto, es decir al atributo $ubicacion se carga con el valor por defecto del parámetro que es 'center'. Lo mismo ocurre con los otros dos parámetros del constructor.

Luego si llamamos al constructor con la siguiente sintaxis:

```
$cabecera2=new CabeceraPagina('El blog del programador','left');
```

el parámetro $ubi recibe el string 'left' y este valor remplaza al valor por defecto que es 'center'.

También podemos llamar al constructor con las siguientes sintaxis:

```
$cabecera3=new CabeceraPagina('El blog del programador','right','#ff0000');
.....
$cabecera4=new CabeceraPagina('El blog del
programador','right','#ff0000','#ffff00');
```

Veamos cuales son las restricciones que debemos tener en cuenta cuando utilizamos parámetros opcionales:

- No podemos definir un parámetro opcional y seguidamente un parámetro obligatorio. Es decir los parámetros opcionales se deben ubicar a la derecha en la declaración del método.
- Cuando llamamos al método no podemos alternar indicando algunos valores a los parámetros opcionales y otros no. Es decir que debemos pasar valores a los parámetros opcionales teniendo en cuenta la dirección de izquierda a derecha en cuanto a la ubicación de parámetros. Para que quede más claro no podemos no indicar el parámetro $ubi y sí el parámetro $colorFuen (que se encuentra a la derecha).Es decir debemos planificar muy bien que orden definir los parámetros opcionales para que luego sea cómodo el uso de los mismos.

Podemos definir parámetros opcionales tanto para el constructor como para cualquier otro método de la clase. Los parámetros opcionales nos permiten desarrollar clases que sean más flexibles en el momento que definimos objetos de las mismas.

Problema:

Confeccionar una clase Empleado, definir como atributos su nombre y sueldo. El constructor recibe como parámetros el nombre y el sueldo, en caso de no pasar el valor del sueldo inicializarlo con el valor 1000.Confeccionar otro método que imprima el nombre y el sueldo.
Crear luego dos objetos del la clase Empleado, a uno de ellos no enviarle el sueldo.

11 - Herencia.

Otro requisito que tiene que tener un lenguaje para considerarse orientado a objetos es la HERENCIA.
La herencia significa que se pueden crear nuevas clases partiendo de clases existentes, que tendrá todas los atributos y los métodos de su 'superclase' o 'clase padre' y además se le podrán añadir otros atributos y métodos propios.
En PHP, a diferencia de otros lenguajes orientados a objetos (C++), una clase sólo puede derivar de una única clase, es decir, PHP no permite herencia múltiple.

Superclase o clase padre

Clase de la que desciende o deriva una clase. Las clases hijas (descendientes) heredan (incorporan) automáticamente los atributos y métodos de la la clase padre.

Subclase

Clase desciendiente de otra. Hereda automáticamente los atributos y métodos de su superclase. Es una especialización de otra clase. Admiten la definición de nuevos atributos y métodos para aumentar la especialización de la clase.
Veamos algunos ejemplos teóricos de herencia:
1) Imaginemos la clase Vehículo. Qué clases podrían derivar de ella?

Vehiculo

Colectivo Moto Auto

FordK Renault 9

Siempre hacia abajo en la jerarquía hay una especialización (las subclases añaden nuevos atributos y métodos.
2) Imaginemos la clase Software. Qué clases podrían derivar de ella?

Software

DeAplicacion DeBase

ProcesadorTexto HojaDeCalculo SistemaOperativo

Word WordPerfect Excel Lotus123 Linux Windows

El primer tipo de relación que habíamos visto entre dos clases, es la de colaboración. Recordemos que es cuando una clase contiene un objeto de otra clase como atributo.
Cuando la relación entre dos clases es del tipo "...tiene un..." o "...es parte de...", no debemos implementar herencia. Estamos frente a una relación de colaboración de clases no de herencia.
Si tenemos una ClaseA y otra ClaseB y notamos que entre ellas existe una relación de tipo "... tiene un...", no debe implementarse herencia sino declarar en la clase ClaseA un atributo de la clase ClaseB.
Por ejemplo: tenemos una clase Auto, una clase Rueda y una clase Volante. Vemos que la relación entre ellas es: Auto "...tiene 4..." Rueda, Volante "...es parte de..." Auto; pero la clase Auto no debe derivar de Rueda ni Volante de Auto porque la relación no es de tipo-subtipo sino de colaboración. Debemos declarar en la clase Auto 4 atributos de tipo Rueda y 1 de tipo Volante.
Luego si vemos que dos clase responden a la pregunta ClaseA "..es un.." ClaseB es posible que haya una relación de herencia.
Por ejemplo:Auto "es un" Vehículo· . Circulo "es una" Figura. Mouse "es un" DispositivoEntrada. Suma "es una" Operación.
Ahora plantearemos el primer problema utilizando herencia en PHP. Supongamos que necesitamos implementar dos clases que llamaremos Suma y Resta. Cada clase tiene como atributo $valor1, $valor2 y $resultado. Los métodos a definir son cargar1 (que inicializa el atributo $valor1), carga2 (que inicializa el atributo $valor2), operar (que en el caso de la clase "Suma" suma los dos atributos y en el caso de la clase "Resta" hace la diferencia entre $valor1 y $valor2, y otro método mostrarResultado.
Si analizamos ambas clases encontramos que muchos atributos y métodos son idénticos. En estos casos es bueno definir una clase padre que agrupe dichos atributos y responsabilidades comunes.
La relación de herencia que podemos disponer para este problema es:

Operacion

Suma Resta

Solamente el método operar es distinto para las clases Suma y Resta (esto hace que no lo podamos disponer en la clase Operacion), luego los métodos cargar1, cargar2 y mostrarResultado son idénticos a las

dos clases, esto hace que podamos disponerlos en la clase Operacion. Lo mismo los atributos $valor1, $valor2 y $resultado se definirán en la clase padre Operacion.
En PHP la codificación de estas tres clases es la siguiente:

```
<html>
<head>
<title>Pruebas</title>
</head>
<body>
<?php

class Operacion {
   protected $valor1;
   protected $valor2;
   protected $resultado;
   public function cargar1($v)
   {
      $this->valor1=$v;
   }
   public function cargar2($v)
   {
      $this->valor2=$v;
   }
   public function imprimirResultado()
   {
      echo $this->resultado.'<br>';
   }
}

class Suma extends Operacion{
   public function operar()
   {
      $this->resultado=$this->valor1+$this->valor2;
   }
}

class Resta extends Operacion{
   public function operar()
   {
      $this->resultado=$this->valor1-$this->valor2;
   }
}

$suma=new Suma();
$suma->cargar1(10);
$suma->cargar2(10);
$suma->operar();
echo 'El resultado de la suma de 10+10 es:';
$suma->imprimirResultado();

$resta=new Resta();
$resta->cargar1(10);
$resta->cargar2(5);
$resta->operar();
echo 'El resultado de la diferencia de 10-5 es:';
$resta->imprimirResultado();

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

La clase Operación define los tres atributos:

```
class Operacion {
  protected $valor1;
  protected $valor2;
  protected $resultado;
```

Ya veremos que definimos los atributos con este nuevo modificador de acceso (protected) para que la subclase tenga acceso a dichos atributos. Si los definimos private las subclases no pueden acceder a dichos atributos.
Los métodos de la clase Operacion son:

```
  public function cargar1($v)
  {
    $this->valor1=$v;
  }
  public function cargar2($v)
  {
    $this->valor2=$v;
  }
  public function imprimirResultado()
  {
    echo $this->resultado.'<br>';
  }
```

Ahora veamos como es la sintaxis para indicar que una clase hereda de otra en PHP:

```
class Suma extends Operacion{
```

Utilizamos la palabra clave extends y seguidamente el nombre de la clase padre (con esto estamos indicando que todos los métodos y atributos de la clase Operación son también métodos de la clase Suma.

Luego la característica que añade la clase Suma es el siguiente método:

```
  public function operar()
  {
    $this->resultado=$this->valor1+$this->valor2;
  }
```

El método operar puede acceder a los atributos heredados (siempre y cuando los mismos se declaren protected, en caso que sean private si bien lo hereda de la clase padre solo los pueden modificar métodos de dicha clase padre.
Ahora podemos decir que la clase Suma tiene cuatro métodos (tres heredados y uno propio) y 3 atributos (todos heredados)
Si creamos un objeto de la clase Suma tenemos:

```
$suma=new Suma();
$suma->cargar1(10);
$suma->cargar2(10);
$suma->operar();
echo 'El resultado de la suma de 10+10 es:';
$suma->imprimirResultado();
```

Podemos llamar tanto al método propio de la clase Suma "operar()" como a los métodos heredados. Quien utilice la clase Suma solo debe conocer que métodos públicos tiene (independientemente que pertenezcan a la clase Suma o a una clase superior)
La lógica es similar para declarar la clase Resta:

```
class Resta extends Operacion{
  public function operar()
  {
    $this->resultado=$this->valor1-$this->valor2;
  }
}
```

y la definición de un objeto de dicha clase:

```
$resta=new Resta();
$resta->cargar1(10);
$resta->cargar2(5);
```

```
$resta->operar();
echo 'El resultado de la diferencia de 10-5 es:';
$resta->imprimirResultado();
```

La clase Operación agrupa en este caso un conjunto de atributos y métodos comunes a un conjunto de subclases (Suma, Resta). No tiene sentido definir objetos de la clase Operacion.
El planteamiento de jerarquías de clases es una tarea compleja que requiere un perfecto entendimiento de todas las clases que intervienen en un problema, cuales son sus atributos y responsabilidades.

Problema:

Confeccionar una clase Persona que tenga como atributos el nombre y la edad. Definir como responsabilidades un método que cargue los datos personales y otro que los imprima.
Plantear una segunda clase Empleado que herede de la clase Persona. Añadir un atributo sueldo y los métodos de cargar el sueldo e imprimir su sueldo.
Definir un objeto de la clase Persona y llamar a sus métodos. También crear un objeto de la clase Empleado y llamar a sus métodos.

12 - Modificadores de acceso a atributos y métodos (protected)

En el concepto anterior presentamos la herencia que es una de las características fundamentales de la programación orientada a objetos.
Habíamos dicho que otro objetivo de la POO es el encapsulamiento (es decir ocultar todo aquello que no le interese a otros objetos), para lograr esto debemos definir los atributos y métodos como privados. El inconveniente es cuando debemos utilizar herencia.
Una subclase no puede acceder a los atributos y métodos privados de la clase padre. Para poder accederlos deberíamos definirlos como públicos (pero esto trae como contrapartida que perdemos el encapsulamiento de la clase)
Aquí es donde entra en juego el modificador protected. Un atributo o método protected puede ser accedido por la clase, por todas sus subclases pero no por los objetos que definimos de dichas clases.

En el problema de las clases Operacion y Suma se producirá un error si tratamos de acceder a un atributo protected donde definimos un objeto del mismo:

```
<html>
<head>
<title>Pruebas</title>
</head>
<body>
<?php
class Operacion {
  protected $valor1;
  protected $valor2;
  protected $resultado;
  public function cargar1($v)
  {
    $this->valor1=$v;
  }
  public function cargar2($v)
  {
    $this->valor2=$v;
  }
  public function imprimirResultado()
  {
    echo $this->resultado.'<br>';
  }
}

class Suma extends Operacion{
  public function operar()
  {
    $this->resultado=$this->valor1+$this->valor2;
  }
}

$suma=new Suma();
$suma->valor1=10;
$suma->cargar2(10);
$suma->operar();
echo 'El resultado de la suma de 10+10 es:';
$suma->imprimirResultado();
?>
</body>
</html>
```

Cuando se ejecuta esta línea:

```
$suma->valor1=10;
```

Aparece el mensaje de error:

Fatal error: Cannot access protected property Suma::$valor1

Problema

Confeccionar una clase Persona que tenga como atributos protegidos, el nombre y la edad. Definir como responsabilidades un método que cargue los datos personales y otro que los imprima.
Plantear una segunda clase Empleado que herede de la clase Persona. Añadir un atributo sueldo y los métodos de cargar el sueldo e imprimir su sueldo.
Definir un objeto de la clase Empleado y tratar de modificar el atributo edad.

13 - Sobrescritura de métodos.

Una subclase en PHP puede redefinir un método, es decir que podemos crear un método con el mismo nombre que el método de la clase padre. Ahora cuando creamos un objeto de la subclase, el método que se llamará es el de dicha subclase.

Lo más conveniente es sobrescribir métodos para completar el algoritmo del método de la clase padre. No es bueno sobrescribir un método y cambiar completamente su comportamiento.

Veamos nuestro problema de las tres clases: Operacion, Suma y Resta. Sobrescribiremos en las subclases el método imprimirResultado (el objetivo es que muestre un título indicando si se trata del resultado de la suma de dos valores o el resultado de la diferencia de dos valores)

```
<html>
<head>
<title>Pruebas</title>
</head>
<body>
<?php
class Operacion {
  protected $valor1;
  protected $valor2;
  protected $resultado;
  public function cargar1($v)
  {
    $this->valor1=$v;
  }
  public function cargar2($v)
  {
    $this->valor2=$v;
  }
  public function imprimirResultado()
  {
    echo $this->resultado.'<br>';
  }
}

class Suma extends Operacion{
  public function operar()
  {
    $this->resultado=$this->valor1+$this->valor2;
  }
 public function imprimirResultado()
  {
    echo "La suma de $this->valor1 y $this->valor2 es:";
    parent::imprimirResultado();
  }


class Resta extends Operacion{
  public function operar()
  {
    $this->resultado=$this->valor1-$this->valor2;
  }
  public function imprimirResultado()
  {
    echo "La diferencia de $this->valor1 y $this->valor2 es:";
    parent::imprimirResultado();
  }
}

$suma=new Suma();
$suma->cargar1(10);
```

```
$suma->cargar2(10);
$suma->operar();
$suma->imprimirResultado();

$resta=new Resta();
$resta->cargar1(10);
$resta->cargar2(5);
$resta->operar();
$resta->imprimirResultado();
?>
</body>
</html>
```

La clase operación define el método imprimirResultado:

```
class Operacion {
...
  public function imprimirResultado()
  {
    echo $this->resultado.'<br>';
  }
```

Luego la subclase sobreescribe dicho método (es decir define otro método con el mismo nombre):

```
class Suma extends Operacion {
...
  public function imprimirResultado()
  {
    echo "La suma de $this->valor1 y $this->valor2 es:";
    parent::imprimirResultado();
  }
```

Esto hace que cuando definamos un objeto de la clase Suma se llamará el método sobrescrito (es decir el de la clase Suma):

```
$suma=new Suma();
$suma->cargar1(10);
$suma->cargar2(10);
$suma->operar();
$suma->imprimirResultado(); //Se llama el método imprimirResultado de la clase Suma
```

Pero si observamos nuevamente el método imprimirResultado de la clase Suma podemos ver que desde el mismo se llama al método imprimirResultado de la clase Operacion con la siguiente sintaxis:

```
    parent::imprimirResultado();
```

La palabra clave **parent** indica que se llama a un método llamado imprimirResultado de la clase padre y no se está llamando recursivamente.
Sería incorrecto si llamamos con la siguiente sintaxis:

```
$this->imprimirResultado();
```

Con esta sintaxis estamos llamando en forma recursiva al mismo método.
Como podemos analizar hemos llevado el título que indica si se trata de una suma o resta a la clase respectiva (no podemos definir dicho título en la clase Operacion y por eso tuvimos que sobrescribir el método imprimirResultado)

Problema:

Confeccionar una clase Persona que tenga como atributos el nombre y la edad. Definir como responsabilidades un método que cargue los datos personales y otro que los imprima.
Plantear una segunda clase Empleado que herede de la clase Persona. Añadir un atributo sueldo y los métodos de cargar el sueldo e imprimir todos los datos del empleado (sobrescribir el método imprimir de la clase Persona).
Definir un objeto de la clase Persona y llamar a sus métodos. También crear un objeto de la clase Empleado y llamar a sus métodos.

14 - Sobrescritura del constructor.

Cuando creamos un objeto de una clase el primer método que se ejecuta es el constructor. Si la clase no tiene constructor pero la subclase si lo tiene, el que se ejecuta es el constructor de la clase padre:

```
<html>
<head>
<title>Pruebas</title>
</head>
<body>
<?php
class Operacion {
  protected $valor1;
  protected $valor2;
  protected $resultado;
  public function __construct($v1,$v2)
  {
    $this->valor1=$v1;
    $this->valor2=$v2;
  }
  public function imprimirResultado()
  {
    echo $this->resultado.'<br>';
  }
}

class Suma extends Operacion{
  public function operar()
  {
    $this->resultado=$this->valor1+$this->valor2;
  }
}

$suma=new Suma(10,10);
$suma->operar();
$suma->imprimirResultado();
?>
</body>
</html>
```

La clase Suma no tiene constructor pero cuando creamos un objeto de dicha clase le pasamos dos datos:

```
$suma=new Suma(10,10);
```

Esto es así ya que la clase padre si tiene constructor:

```
  public function __construct($v1,$v2)
  {
    $this->valor1=$v1;
    $this->valor2=$v2;
  }
```

Ahora veremos un problema que la subclase también tenga constructor, es decir sobrescribimos el constructor de la clase padre.

Problema Resuelto: Implementar la clase Operacion. El constructor recibe e inicializa los atributos $valor1 y $valor2. La subclase Suma añade un atributo $titulo. El constructor de la clase Suma recibe los dos valores a sumar y el título.

```
<html>
<head>
<title>Pruebas</title>
</head>
<body>
<?php
class Operacion {
  protected $valor1;
  protected $valor2;
  protected $resultado;
  public function __construct($v1,$v2)
  {
    $this->valor1=$v1;
    $this->valor2=$v2;
  }
  public function imprimirResultado()
  {
    echo $this->resultado.'<br>';
  }
}

class Suma extends Operacion{
  protected $titulo;
  public function __construct($v1,$v2,$tit)
  {
    parent::__construct($v1,$v2);
    $this->titulo=$tit;
  }
  public function operar()
  {
    echo $this->titulo;
    echo $this->valor1.'+'.$this->valor2.' es ';
    $this->resultado=$this->valor1+$this->valor2;
  }
}

$suma=new Suma(10,10,'Suma de valores:');
$suma->operar();
$suma->imprimirResultado();
?>
</body>
</html>
```

Nuestra clase Operacion no a sufrido cambios. Veamos ahora la clase Suma que añade un atributo $titulo y constructor:

```
  public function __construct($v1,$v2,$tit)
  {
    parent::__construct($v1,$v2);
    $this->titulo=$tit;
  }
```

El constructor de la clase Suma recibe tres parámetros. Lo primero que hacemos es llamar al constructor de la clase padre que tiene por objetivo inicializar los atributos $valor1 y $valor2 con la siguiente sintaxis:

```
    parent::__construct($v1,$v2);
```

Mediante la palabra clave parent indicamos que llamamos el método __construct de la clase padre. Además utilizamos el operador **::**

El constructor de la clase Suma carga el atributo $titulo:

```
    $this->titulo=$tit;
```

Ahora cuando creamos un objeto de la clase Suma debemos pasar los valores a los tres parámetros del constructor:

```
$suma=new Suma(10,10,'Suma de valores:');
$suma->operar();
$suma->imprimirResultado();
```

Si nos equivocamos y llamamos al constructor con dos parámetros:

```
$suma=new Suma(10,10);
```

Se muestra un warning:

Warning: Missing argument 3 for Suma::__construct()

Problema:

Confeccionar una clase Persona que tenga como atributos el nombre y la edad. El constructor recibe los datos para inicializar dichos atributos. Otro método imprime los datos.
Plantear una segunda clase Empleado que herede de la clase Persona. Añadir un atributo sueldo. El constructor recibe los tres atributos de la clase Empleado. Llamar al constructor de la clase padre para inicializar los atributos nombre y edad del Empleado.
Definir un objeto de la clase Persona y llamar a sus métodos. También crear un objeto de la clase Empleado y llamar a sus métodos.

15 - Clases abstractas y concretas.

Una clase abstracta tiene por objetivo agrupar atributos y métodos que luego serán heredados por otras subclases.
En conceptos anteriores planteamos las tres clase: Operacion, Suma y Resta. Vimos que no tenía sentido definir objetos de la clase Operacion (clase abstracta) y si definimos objetos de las clases Suma y Resta (clases concretas).
No es obligatorio que toda clase padre sea abstracta. Podemos tener por ejemplo un problema donde tengamos una clase padre (superclase) llamada Persona y una subclase llamada Empleado y luego necesitemos definir objetos tanto de la clase Persona como de la clase Empleado.

Existe una sintaxis en PHP para indicar que una clase es abstracta:

```
abstract class [nombre de clase] {
  [atributos]
  [metodos]
}
```

La ventaja de definir las clases abstractas con este modificador es que se producirá un error en tiempo de ejecución si queremos definir un objeto de dicha clase. Luego hay que tener bien en cuenta que solo podemos definir objetos de las clases concretas.
Luego el problema de herencia de las clases Operacion, Suma y Resta es:

```
<html>
<head>
<title>Pruebas</title>
</head>
<body>
<?php
abstract class Operacion {
  protected $valor1;
  protected $valor2;
  protected $resultado;
  public function cargar1($v)
  {
    $this->valor1=$v;
  }
  public function cargar2($v)
  {
    $this->valor2=$v;
  }
  public function imprimirResultado()
  {
    echo $this->resultado.'<br>';
  }
}

class Suma extends Operacion{
  public function operar()
  {
    $this->resultado=$this->valor1+$this->valor2;
  }
}
```

```
class Resta extends Operacion{
  public function operar()
  {
    $this->resultado=$this->valor1-$this->valor2;
  }
}
```

```
$suma=new Suma();
$suma->cargar1(10);
$suma->cargar2(10);
$suma->operar();
echo 'El resultado de la suma de 10+10 es:';
$suma->imprimirResultado();

$resta=new Resta();
$resta->cargar1(10);
$resta->cargar2(5);
$resta->operar();
echo 'El resultado de la diferencia de 10-5 es:';
$resta->imprimirResultado();
?>
</body>
</html>
```

El único cambio que hemos producido a nuestro ejemplo está en la línea donde declaramos la clase Operacion:

```
abstract class Operacion {
```

No varía en nada la declaración de las otras dos clases:

```
class Suma extends Operacion{
...
}

class Resta extends Operacion{
...
}
```

Es decir que las clases concretas son aquellas que no le antecedemos el modificador abstract.
La definición de objetos de la clase Suma y Resta no varía:

```
$suma=new Suma();
$suma->cargar1(10);
$suma->cargar2(10);
$suma->operar();
echo 'El resultado de la suma de 10+10 es:';
$suma->imprimirResultado();

$resta=new Resta();
$resta->cargar1(10);
$resta->cargar2(5);
$resta->operar();
echo 'El resultado de la diferencia de 10-5 es:';
$resta->imprimirResultado();
```

Ahora bien si tratamos de definir un objeto de la clase Operación:

```
$operacion1=new Operacion();
$operacion1->cargar1(12);
$operacion1->cargar2(6);
$operacion1->imprimirResultado();
```

Se produce un error:

```
Fatal error: Cannot instantiate abstract class Operacion
```

Es decir que luego cuando utilicemos las clases que desarrollamos (definamos objetos) solo nos interesan las clases concretas.

Problema:

Confeccionar una clase abstracta Persona que tenga como atributos el nombre y la edad. Definir como responsabilidades un método que cargue los datos personales y otro que los imprima.
Plantear una segunda clase Empleado que herede de la clase Persona. Añadir un atributo sueldo y los métodos de cargar el sueldo e imprimir su sueldo.
Definir un objeto de la clase Persona y ver que error produce. También crear un objeto de la clase Empleado y llamar a sus métodos.

16 - Métodos abstractos.

Vimos en conceptos anteriores que podemos definir clases abstractas que tienen por objetivo agrupar atributos y métodos comunes a un conjunto de subclases.
Si queremos que las subclases implementen comportamientos obligatoriamente podemos definir métodos abstractos.
Un método abstracto se declara en una clase pero no se lo implementa.

En nuestra clase Operacion tiene sentido declarar un método abstracto: operar. Esto hará que todas las clases que hereden de la clase Operación deban implementar el método operar.
Veamos la sintaxis para declarar un método abstracto con el problema de las clases Operacion, Suma y Resta.

```
<html>
<head>
<title>Pruebas</title>
</head>
<body>
<?php
abstract class Operacion {
  protected $valor1;
  protected $valor2;
  protected $resultado;
  public function cargar1($v)
  {
    $this->valor1=$v;
  }
  public function cargar2($v)
  {
    $this->valor2=$v;
  }
  public function imprimirResultado()
  {
    echo $this->resultado.'<br>';
  }
  public abstract function operar();
}

class Suma extends Operacion{
  public function operar()
  {
    $this->resultado=$this->valor1+$this->valor2;
  }
}

class Resta extends Operacion{
  public function operar()
  {
    $this->resultado=$this->valor1-$this->valor2;
  }
}

$suma=new Suma();
$suma->cargar1(10);
$suma->cargar2(10);
$suma->operar();
echo 'El resultado de la suma de 10+10 es:';
$suma->imprimirResultado();

$resta=new Resta();
```

```
$resta->cargar1(10);
$resta->cargar2(5);
$resta->operar();
echo 'El resultado de la diferencia de 10-5 es:';
$resta->imprimirResultado();
?>
</body>
</html>
```

Dentro de la clase Operacion declaramos un método pero no lo implementamos:

```
  public abstract function operar();
```

Con esto logramos que todas las subclases de la clase Operacion deben implementar el método operar():

```
class Suma extends Operacion{
  public function operar()
  {
    $this->resultado=$this->valor1+$this->valor2;
  }
}
```

Si una subclase no lo implementa se produce un error (supongamos que la subclase Suma se nos olvida implementar el método operar):

```
Fatal error: Class Suma contains 1 abstract method and must therefore be
declared abstract or implement the remaining methods (Operacion::operar)
```

Solo se pueden declarar métodos abstractos dentro de una clase abstracta. Un método abstracto no puede ser definido con el modificador private (esto es obvio ya que no lo podría implementar la subclase)

Problema:

Plantear una clase abstracta Trabajador. Definir como atributo su nombre y sueldo. Declarar un método abstracto: calcularSueldo. Definir otro método para imprimir el nombre y su sueldo.
Plantear una subclase Empleado. Definir el atributo $horasTrabajadas, $valorHora. Para calcular el sueldo tener en cuenta que se le paga 3.50 la hora.
Plantear una clase Gerente que herede de la clase Trabajador. Para calcular el sueldo tener en cuenta que se le abona un 10% de las utilidades de la empresa.

17 - Métodos y clases final.

Si a un método le agregamos el modificador final significa que ninguna subclase puede sobrescribirlo. Este mismo modificador se lo puede aplicar a una clase, con esto estaríamos indicando que dicha clase no se puede heredar.

Confeccionaremos nuevamente las clases Operacion y Suma utilizando este modificador. Definiremos un método final en la clase Operacion y la subclase la definiremos de tipo final.

```
<html>
<head>
<title>Pruebas</title>
</head>
<body>
<?php
class Operacion {
  protected $valor1;
  protected $valor2;
  protected $resultado;
  public function __construct($v1,$v2)
  {
    $this->valor1=$v1;
    $this->valor2=$v2;
  }
  public final function imprimirResultado()
  {
    echo $this->resultado.'<br>';
  }
}

final class Suma extends Operacion{
  private $titulo;
  public function __construct($v1,$v2,$tit)
  {
    Operacion::__construct($v1,$v2);
    $this->titulo=$tit;
  }
  public function operar()
  {
    echo $this->titulo;
    echo $this->valor1.'+'.$this->valor2.' es ';
    $this->resultado=$this->valor1+$this->valor2;
  }
}

$suma=new Suma(10,10,'Suma de valores:');
$suma->operar();
$suma->imprimirResultado();
?>
</body>
</html>
```

Podemos ver que la sintaxis para definir un método final es:

```
class Operacion {
...
  public final function imprimirResultado()
  {
    echo $this->resultado.'<br>';
  }
}
```

Luego si una subclase intenta redefinir dicho método:

```
class Suma extends Operacion {
...
  public function imprimirResultado()
  {
    ...
  }
}
```

Se produce el siguiente error:

```
Fatal error: Cannot override final method Operacion::imprimirResultado()
```

El mismo concepto se aplica cuando queremos que una clase quede sellado y no dejar crear subclases a partir de ella:

```
final class Suma extends Operacion{
...
}
```

Agregando el modificador final previo a la declaración de la clase estamos indicando que dicha clase no se podrá heredar.
Luego si planteamos una clase que herede de la clase Suma:

```
class SumaTresValores extends Suma {
...
}
```

Produce el siguiente error:

```
Fatal error: Class SumaValores may not inherit from final class (Suma)
```

Problema:

Confeccionar una clase Persona que tenga como atributos el nombre y la edad. Definir como responsabilidades un método final que cargue los datos personales. Otro método debe imprimir dichos datos personales.
Plantear una segunda clase Empleado que herede de la clase Persona. Definir la clase Persona con el modificador Final. Añadir un atributo sueldo y los métodos de cargar el sueldo e imprimir su sueldo.
Definir un objeto de la clase Persona y llamar a sus métodos. También crear un objeto de la clase Empleado y llamar a sus métodos.

18 - Referencia y clonación de objetos.

Hay que tener en cuenta que un objeto es una estructura de datos compleja. Luego cuando asignamos una variable de tipo objeto a otra variable lo que estamos haciendo es guardar la referencia del objeto. No se está creando un nuevo objeto, sino otra variable por la que podemos acceder al mismo objeto.

Si queremos crear un nuevo objeto idéntico a otro debemos utilizar el operador clone.

El siguiente ejemplo muestra la diferencia entre asignación y clonación:

```
<html>
<head>
<title>Pruebas</title>
</head>
<body>
<?php
class Persona {
  private $nombre;
  private $edad;
  public function fijarNombreEdad($nom,$ed)
  {
    $this->nombre=$nom;
    $this->edad=$ed;
  }
  public function retornarNombre()
  {
    return $this->nombre;
  }
  public function retornarEdad()
  {
    return $this->edad;
  }
}

$persona1=new Persona();
$persona1->fijarNombreEdad('Juan',20);
$x=$persona1;
echo 'Datos de la persona ($persona1):';
echo $persona1->retornarNombre().' - '.$persona1->retornarEdad().'<br>';
echo 'Datos de la persona ($x):';
echo $x->retornarNombre().' - '.$x->retornarEdad().'<br>';
$x->fijarNombreEdad('Ana',25);
echo 'Después de modificar los datos<br>';
echo 'Datos de la persona ($persona1):';
echo $persona1->retornarNombre().' - '.$persona1->retornarEdad().'<br>';
echo 'Datos de la persona ($x):';
echo $x->retornarNombre().' - '.$x->retornarEdad().'<br>';
$persona2=clone($persona1);
$persona1->fijarNombreEdad('Luis',50);
echo 'Después de modificar los datos de persona1<br>';
echo 'Datos de la persona ($persona1):';
echo $persona1->retornarNombre().' - '.$persona1->retornarEdad().'<br>';
echo 'Datos de la persona ($persona2):';
echo $persona2->retornarNombre().' - '.$persona2->retornarEdad().'<br>';
?>
</body>
</html>
```

Primero creamos un objeto de la clase Persona y cargamos su nombre y edad:

```
$persona1=new Persona();
$persona1->fijarNombreEdad('Juan',20);
```

Definimos una segunda variable $x y guardamos la referencia de la variable $persona1:

```
$x=$persona1;
```

Ahora imprimimos el nombre y edad de la persona accediéndolo primero por la variable $persona1 y luego por la variable $x (Se muestran los mismos datos porque en realidad estamos imprimiendo los atributos del mismo objeto):

```
echo 'Datos de la persona ($persona1):';
echo $persona1->retornarNombre().' - '.$persona1->retornarEdad().'<br>';
echo 'Datos de la persona ($x):';
echo $persona1->retornarNombre().' - '.$persona1->retornarEdad().'<br>';
```

Si modificamos el nombre y la edad de la persona llamando al método fijarNombreEdad mediante la variable $x:

```
$x->fijarNombreEdad('Ana',25);
```

luego imprimimos los atributos mediante las referencias al mismo objeto:

```
echo 'Después de modificar los datos<br>';
echo 'Datos de la persona ($persona1):';
echo $persona1->retornarNombre().' - '.$persona1->retornarEdad().'<br>';
echo 'Datos de la persona ($x):';
echo $persona1->retornarNombre().' - '.$persona1->retornarEdad().'<br>';
```

Para crear un segundo objeto y clonarlo del objeto referenciado por $persona1:

```
$persona2=clone($persona1);
```

Ahora cambiamos los atributos del primer objeto creado:

```
$persona1->fijarNombreEdad('Luis',50);
```

Luego imprimimos los atributos de los dos objetos:

```
echo 'Después de modificar los datos de persona1<br>';
echo 'Datos de la persona ($persona1):';
echo $persona1->retornarNombre().' - '.$persona1->retornarEdad().'<br>';
echo 'Datos de la persona ($persona2):';
echo $persona2->retornarNombre().' - '.$persona2->retornarEdad().'<br>';
```

Al ejecutarlo veremos que los datos que se imprimen son distintos.
Hay que diferenciar bien que el operador de asignación "=" no crea un nuevo objeto sino una nueva referencia a dicho objeto. Si queremos crear un nuevo objeto idéntico a uno ya existente debemos emplear el operador clone.
Cuando pasamos a un método un objeto lo que estamos pasando en realidad es la referencia a dicho objeto (no se crea un nuevo objeto)

Problema:

Crear la clase Persona y cuando se clone un objeto de dicha clase almacenar en el atributo edad la edad actual más uno.

18.b Problemas de Clonado

Aparentemente con esto tenemos solucionados el problema de los clonados, pero... se nos presentan un par de problemáticas

1º) Como clonar un array de Objetos

```
<?php

	class Persona {

		protected $nombre;
		protected $edad;

		public function cargar($nom, $ed) {
			$this->nombre = $nom;
			$this->edad = $ed;
		}

		public function imprimir() {
			echo "La persona ".$this->nombre." tiene ".$this->edad."
			años.<br/>";
		}
	}

	$persona_1 = new Persona();
	$persona_1->cargar("Pablo", 35);

	$persona_2 = new Persona();
	$persona_2->cargar("Juan", 40);

	$persona_3 = new Persona();
	$persona_3->cargar("Ana", 30);

	$array = array($persona_1, $persona_2, $persona_3);
	echo "Array original:<br/>";

	for($i = 0; $i < count($array); $i++) {
		$array[$i]->imprimir();
	}

//Asignamos un array en otro
	$array_copia = $array;

	$array_copia[0]->cargar("Pablo_cambiado", 135);
	$array_copia[1]->cargar("Juan_cambiado", 140);
	$array_copia[2]->cargar("Ana_cambiado", 130);

	echo "<br/>";
	echo "Array copia:<br/>";

	for($i = 0; $i < count($array); $i++) {
		$array_copia[$i]->imprimir();
	}

	echo "Array original:<br/>";

	for($i = 0; $i < count($array); $i++) {
		$array[$i]->imprimir();
	}
```

¿Qué ha pasado? Si yo he copiado el array y he manipulado la copias... ¿por qué narices me manipula el original?
Bueno en este caso es bastante simple, la cuestión es que no invocamos al clon, el segundo array NO es un array independiente sino una referencia al primer array, por tanto todo lo que manipulo en el primero, acabará afectando al segundo y viceversa.

¿Para evitar esto que tengo que hacer? Simple, realizar el clonado elemento a elemento y añadirlo al nuevo array correspondiente
La solución completa sería:

```
<?php

	class Persona {

		protected $nombre;
		protected $edad;

		public function cargar($nom, $ed) {
			$this->nombre = $nom;
			$this->edad = $ed;
		}

		public function imprimir() {
			echo "La persona ".$this->nombre." tiene ".$this->edad."
			años.<br/>";
		}
	}

	$persona_1 = new Persona();
	$persona_1->cargar("Pablo", 35);

	$persona_2 = new Persona();
	$persona_2->cargar("Juan", 40);

	$persona_3 = new Persona();
	$persona_3->cargar("Ana", 30);

	$array = array($persona_1, $persona_2, $persona_3);

	echo "Array original:<br/>";

	for($i = 0; $i < count($array); $i++) {
		$array[$i]->imprimir();
	}

	$array_copia = $array;

	$array_copia[0]->cargar("Pablo_cambiado", 135);
	$array_copia[1]->cargar("Juan_cambiado", 140);
	$array_copia[2]->cargar("Ana_cambiado", 130);

	echo "<br/>";
	echo "Array copia:<br/>";

	for($i = 0; $i < count($array); $i++) {
		$array_copia[$i]->imprimir();
	}

	echo "Array original:<br/>";
```

```
	for($i = 0; $i < count($array); $i++) {
		$array[$i]->imprimir();
	}

	//Como clonar un array de objetos de manera correctamente

	$array_clonado = array();

	foreach($array_copia as $valor) {
		$array_clonado[] = clone($valor);
	}

	$array_clonado[0]->cargar("Pablo_clonado", 1135);

	$array_clonado[1]->cargar("Juan_clonado", 1140);

	$array_clonado[2]->cargar("Ana_clonado", 1130);

	echo "<br/>";
	echo "Array copia:<br/>";

	for($i = 0; $i < count($array); $i++) {
		$array_copia[$i]->imprimir();
	}

	echo "Array clonado:<br/>";

	for($i = 0; $i < count($array); $i++) {
		$array_clonado[$i]->imprimir();
	}

?>
```

Lo que realizamos aquí es un clonado elemento a elemento, por tanto el nuevo array SÍ es un array de elementos clonados y por tanto independiente del original.

2º) Como clonar un objeto dentro de otro objeto
Antes de nada, pongamos que tengo el siguiente caso:

```
<?php

	class Juego {
		private $titulo;
		private $motor;
		private $personaje;

		public function __construct($t, $m, $p){
			$this->titulo = $t;
			$this->motor = $m;
			$this->personaje = $p;
		}

		public function setTitulo($t) {
			$this->titulo = $t;
		}

		public function getTitulo() {
			return $this->titulo;
		}
		public function setMotor($m) {
```

```
            $this->motor = $m;
        }

        public function getMotor() {
            return $this->motor;
        }

        public function setPersonaje($p) {
            $this->personaje = $p;
        }

        public function getPersonaje() {
            return $this->personaje;
        }

        public function toString() {
            echo "<pre>";
            echo "\t.....................................\n";
            echo "\tTitulo: ".$this->titulo."\n";
            echo "\tMotor:  ".$this->motor."\n";
            $this->personaje->toString();
            echo "\t.....................................\n";
            echo "</pre>";
            echo "<br />";
        }
    }

    class Personaje {
        private $nombre;
        private $edad;
        private $bebida;

        public function __construct($n, $e, $b){
            $this->nombre = $n;
            $this->edad = $e;
            $this->bebida = $b;
        }

        public function setNombre($n) {

            $this->nombre = $n;
        }

        public function getNombre() {
            return $this->nombre;
        }

        public function setEdad($e) {
            $this->edad = $e;
        }

        public function getEdad() {
            return $this->edad;
        }

        public function setBebida($b) {
            $this->bebida = $b;
        }

        public function getBebida() {
            return $this->bebida;
```

```
            }

            public function toString() {
                  echo "\tPersonaje: \n";
                  echo "\t\tNombre: ".$this->getNombre()."\n";
                  echo "\t\tEdad:   ".$this->getEdad()."\n";
                  echo "\t\tBebida: ".$this->getBebida()."\n";
            }
      }

      echo "@ REALIZAMOS UN CLON DEL JUEGO 'A PELO':<br/><br/>";

      $guybrush = new Personaje("Guybrush Threepwood", "unkown", "Grog");

      $monkey = new Juego("The Secret of Monkey Island", "SCUMM", $guybrush);

      $monkey_clon = clone($monkey);

      echo "EL JUEGO ORIGINAL:<br />";
      $monkey->toString();

      echo "EL JUEGO CLONADO:<br />";
      $monkey_clon->toString();


      echo "@ MODIFICAMOS UN ATRIBUTO DEL JUEGO ORIGINAL Y UN <br/>ATRIBUTO DEL
OBJETO PERSONAJE DEL JUEGO ORIGINAL:<br/><br/>";

      $monkey->setTitulo("#########CHANGED#########");
      $monkey->getPersonaje()->setNombre("#########CHANGED#########");

      echo "EL JUEGO ORIGINAL:<br />";
      $monkey->toString();

      echo "EL JUEGO CLONADO:<br />";
      $monkey_clon->toString();

      echo "<pre>F  A  I  L  ! ! ! ! ! ! ! ! !</pre><br/><br/>";
?>
```

¿Qué ocurre aquí? El objeto “padre” (no me gusta el termino padre pues parece indicar herencia y estamos hablando de colaboración, sería más correcto decir "el objeto que contiene a otro objeto"... pero claro desde luego es menos intuitivo) Juego, se clona correctamente y es un objeto independiente, pero el objeto “Personaje” que se encuentra dentro de Juego, NO.

¿Por qué? Al realizar la clonación sólo clonamos el objeto padre(objeto contenedor), pero al NO invocar un clone sobre el objeto hijo (objeto contenido), no se realiza una copia sino una referencia al mismo, por tanto tengo un objeto padre perfectamente clonado y un objeto hijo (Personaje) que es una referencia al personaje del otro objeto.

¿Qué solución podríamos aplicar? Pues a la vez que clonamos el padre DEBEMOS acceder a clonar el hijo. (el objeto contenido)

NOTA: A partir de ahora cuando expliquemos clonado usaremos términos de parentesco (padre-hijo en vez de objeto contenedor-objeto contenido) para entender mejor a que nos referimos, pero recuerda, NO hablamos de herencia sino de colaboración entre objetos.

A continuación os pongo una solución:

```
<?php

	class Juego {
		private $titulo;
		private $motor;
		private $personaje;

		public function __construct($t, $m, $p){
			$this->titulo = $t;
			$this->motor = $m;
			$this->personaje = $p;
		}

		public function setTitulo($t) {
			$this->titulo = $t;
		}

		public function getTitulo() {
			return $this->titulo;
		}

		public function setMotor($m) {
			$this->motor = $m;
		}

		public function getMotor() {
			return $this->motor;
		}

		public function setPersonaje($p) {
			$this->personaje = $p;
		}

		public function getPersonaje() {
			return $this->personaje;
		}

		public function toString() {
			echo "<pre>";
			echo "\t.....................................\n";
			echo "\tTitulo: ".$this->titulo."\n";
			echo "\tMotor:  ".$this->motor."\n";
			$this->personaje->toString();
			echo "\t.....................................\n";
			echo "</pre>";
			echo "<br />";
		}
	}

	class Personaje {
		private $nombre;
		private $edad;
		private $bebida;

		public function __construct($n, $e, $b){
			$this->nombre = $n;
			$this->edad = $e;
			$this->bebida = $b;
		}
```

```
public function setNombre($n) {
                $this->nombre = $n;
          }

          public function getNombre() {
                return $this->nombre;
          }

          public function setEdad($e) {
                $this->edad = $e;
          }

          public function getEdad() {
                return $this->edad;
          }

          public function setBebida($b) {
                $this->bebida = $b;
          }

          public function getBebida() {
                return $this->bebida;
          }

          public function toString() {
                echo "\tPersonaje: \n";
                echo "\t\tNombre: ".$this->getNombre()."\n";
                echo "\t\tEdad:   ".$this->getEdad()."\n";
                echo "\t\tBebida: ".$this->getBebida()."\n";
          }
     }

     echo "@ REALIZAMOS UN CLON DEL JUEGO 'A PELO':<br/><br/>";

     $guybrush = new Personaje("Guybrush Threepwood", "unkown", "Grog");

     $monkey = new Juego("The Secret of Monkey Island", "SCUMM", $guybrush);

     $monkey_clon = clone($monkey);

     echo "EL JUEGO ORIGINAL:<br />";
     $monkey->toString();

     echo "EL JUEGO CLONADO:<br />";
     $monkey_clon->toString();


     echo "@ MODIFICAMOS UN ATRIBUTO DEL JUEGO ORIGINAL Y UN <br/>ATRIBUTO DEL
OBJETO PERSONAJE DEL JUEGO ORIGINAL:<br/><br/>";

     $monkey->setTitulo("#########CHANGED#########");
     $monkey->getPersonaje()->setNombre("#########CHANGED#########");

     echo "EL JUEGO ORIGINAL:<br />";
     $monkey->toString();

     echo "EL JUEGO CLONADO:<br />";
     $monkey_clon->toString();

     echo "<pre>F   A   I   L   !  !  !  !  !  !  !  !  !</pre><br/><br/>";
```

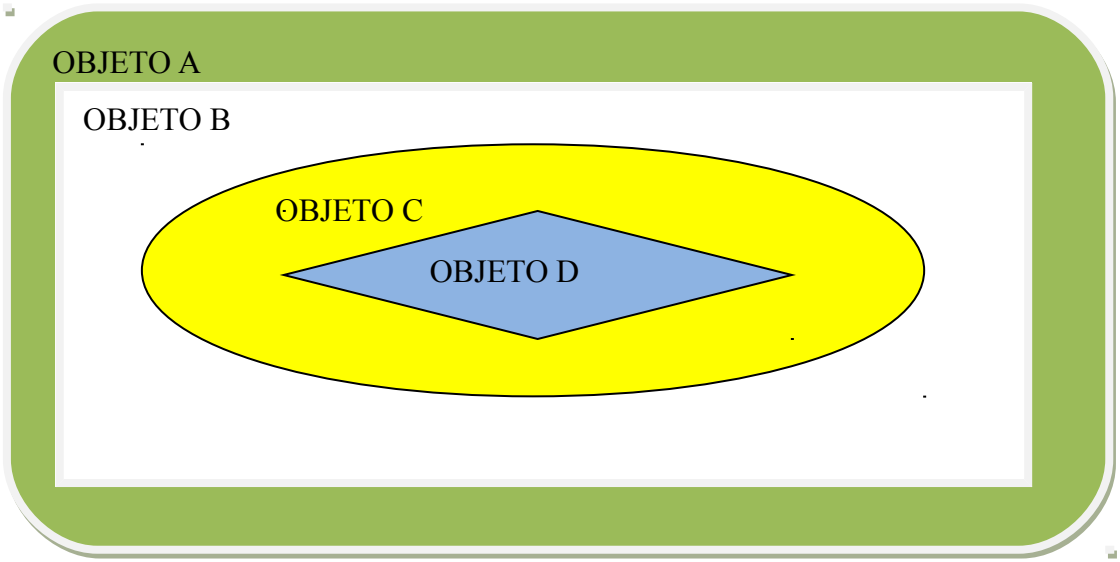
```
      echo "@ VOLVEMOS A REALIZAR LA JUGADA COMO DIOS MANDA:<br/><br/>";
      $guybrush = new Personaje("Guybrush Threepwood", "unkown", "Grog");
      $monkey = new Juego("The Secret of Monkey Island", "SCUMM", $guybrush);

      $monkey_clon = clone($monkey);
      $monkey_clon->setPersonaje(clone($monkey_clon->getPersonaje()));

      echo "EL JUEGO ORIGINAL:<br />";
      $monkey->toString();

      echo "EL JUEGO CLONADO:<br />";
      $monkey_clon->toString();

      echo "@ MODIFICAMOS 'OTRA VEZ' UN ATRIBUTO DEL JUEGO ORIGINAL Y UN
<br/>ATRIBUTO DEL OBJETO PERSONAJE DEL JUEGO ORIGINAL:<br/><br/>";

      $monkey->setTitulo("#########CHANGED#########");
      $monkey->getPersonaje()->setNombre("#########CHANGED#########");

      echo "EL JUEGO ORIGINAL:<br />";
      $monkey->toString();

      echo "EL JUEGO CLONADO:<br />";
      $monkey_clon->toString();

      echo "<pre>Y   E   A   H   !  !  !  !  !  !  !  !  !</pre><br/><br/>";
?>
```

Ojo que con esto no damos acabado el clon, continua en el próximo episodio.

19 – Métodos mágicos: __clone y __toString

En el capítulo anterior hemos visto que cuando hay una objeto dentro de otro objeto la función clone NO clona al objeto *hijo* (recuerda la nota del tema anterior). Por tanto para solucionarlo una vez clonado el *padre* utilizábamos el get para "machacar" el *hijo* con una copia clonada de sí mismo... puff esto es bastante enrevesado.
Aunque bueno es una solución y no mala... pero, ¿qué ocurriría si lo complicáramos un poco más? ¿Que pasaría si en vez de un "*hijo*" y un *padre,* complicáramos un poco las relaciones? Por ejemplo *Bisabuelo, abuelo, padre e hijo*

Es decir un objeto A, que contiene un objeto B, que contiene a su vez un objeto C, el cual contiene un objeto D.



Por tanto el clonar el objeto A, habría que clonar el subobjeto B, y a la vez clonar el subobjeto C y a la vez el subobjeto D, vamos que una vez clonado el *bisabuelo* A habría que, utilizando el get y el set "machacar" al *abuelo* con una copia clonada de sí mismo, una vez eso volver a machacar al *padre* con un copia clonada de si mismo y a la vez... me pierdo, esto más que enrevesado, es toda un tragedia griega.

Para solucionar este problema los desarrolladores de php nos aportan una solución tan simple y eficiente que ellos la denominan mágica (que más quisieran), el método predefinido y mágico **__clone()**

__clone ( ) (con dos guiones subrayados)

En esencia el método __clone es un método que se invocará automáticamente cuando un objeto sea utilizado por la función clone(), viene a ser algo así como lo que ocurre con el método __construct que se ejecuta automáticamente cuando hacemos un new de un objeto.

El método clone nos realizará una copia del objeto propiedad a propiedad, esto por que, por defecto si no sobrescribimos la definición de *__clone()* en nuestra clase, PHP hace la copia completa.
Por ejemplo

```php
<?php
class SubObject{
    static $instances = 0;
    public $instance;
    public function __construct() {
        $this->instance = ++self::$instances;
    }

    public function __clone() {
        $this->instance = ++self::$instances;
    }
}

class MyCloneable{
    public $object1;
    public $object2;

    function __clone() {
        // Forzamos la copia de this->object, si no
        // hará referencia al mismo objeto.
        $this->object1 = clone $this->object1;
    }
}
$obj = new MyCloneable();

$obj->object1 = new SubObject();
$obj->object2 = new SubObject();

$obj2 = clone $obj;
print("Objeto Original:\n");
print_r  ($obj); // puedo hacer print_r por que los atributos
// son públicos, si no usar var_dump
print("Objeto Clonado:\n");
print_r ($obj2);
?>
```

Olvidándonos del modificador self y el acceso a atributos estáticos (lo veremos más adelante) lo interesante de este ejemplo se encuentra en el método __clone de MyCloneable.
¿Qué pasa exactamente cuando?

```
$obj2 = clone $obj;
```

Al ejecutarse la instrucción clone (lo sé también se puede poner sin paréntesis), el $obj a clonar, invoca automáticamente a su método __clone el cual, fíjate que simple, crea una copia clonada del subobjeto. Por tanto el nuevo $obj2 creado a partir de un clon tendrá en su interior dos subobjetos, el $object1 clonado y por tanto independiente del inicial y el $object2 sin clonar , por tanto un referencia al $object2 original.

Si quisiéramos que clonase todos los objetos internos el código debería haber sido el siguiente

```
class MyCloneable{
    public $object1;
    public $object2;

    public function __clone(){
 // Forzamos la copia de this->object, si no hará referencia al mismo objeto.
      $this->object1 = clone $this->object1;
      $this->object2 = clone $this->object2;
    }
}
```

Como puede ver también se ha redefinido el método __clone del subobjeto, esto no tiene más objetivo que aumentar en 1 el contador de instancias del atributo estático para saber que número de instancia o cuantas instancias se ha creado para que la salida por pantalla sea similar a la siguiente

```
Objeto Original:
MyCloneable Object
(
    [object1] => SubObject Object
        (
            [instance] => 1
        )

    [object2] => SubObject Object
        (
            [instance] => 2
        )

)
Objeto Clonado:
MyCloneable Object
(
    [object1] => SubObject Object
        (
            [instance] => 3
        )

    [object2] => SubObject Object
        (
            [instance] => 2
        )
)
```

Quizá este ejemplo haya sido algo complejo debido a que usamos el modificador self.
Para simplificar vemos como quedaría el ejemplo con las clases A, B, C y D.
Ejemplo simple pero completo, allá va.

```php
<?php

class A{
    private $objectB;
    private $valorA;

    public function __construct($valorA, $objectB){
        $this->objectB= $objectB;
        $this->valorA= $valorA;
    }
    public function __clone(){
        $this->objectB= clone ($this->objectB);
      }
    public function setvalorA($nuevo){
        $this->valorA=$nuevo;
    }
    public function setobjectB($objectB){
        $this->objectB=$objectB;
      }
    public function getobjectB(){
        return $this->objectB;
      }
}

class B
{
    private $objectC;
    private $valorB;

    public function __construct($valorB, $objectC){
        $this->objectC= $objectC;
        $this->valorB= $valorB;
    }
    public function getobjectC(){
        return $this->objectC;
    }

   public function __clone(){
        $this->objectC= clone ($this->objectC);
      }
}

class C{
    private $objectD;
    private $valorC;
    public function __construct($valorC, $objectD){
        $this->objectD= $objectD;
        $this->valorC= $valorC;
    }
    public function getobjectD(){
        return $this->objectD;
      }
  public function __clone(){
        $this->objectD= clone ($this->objectD);
      }
```

```
    public function setvalorC($nuevo){
        $this->valorC=$nuevo;
      }
}

class D
{
   private $valorD;

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

    public function __clone(){
      //  $this->$objectE= clone ($this->ObjectE); No es necesario su
      //definicion
    }
    public function setvalorD($nuevo){
        $this->valorD=$nuevo;
      }
}

$objD=new D(1);
$objC= new C(2,$objD);
$objB= new B(3,$objC);
$objA= new A(4,$objB);

//CLONAMOS
$objClon=clone ($objA);

//Manipulo dos datos del Objeto Clon
// si consigo cambiarlos sin manipular el origianl
//  es que el clon se ha realizado de manera correcta
$objClon->getobjectB()->getobjectC()->setvalorC(66);
$objClon->getobjectB()->getobjectC()->getobjectD()->setvalorD(33);

echo "Mostramos Original<BR>";
var_dump ($objA);
echo "<BR>Mostramos Clon Manipulado<BR>";
var_dump ($objClon);
?>
```

Como podemos comprobar son diferentes, por tanto ha clonado de manera efectiva.

Problema:

A partir del ejemplo con las clases Juego y Personaje, modifica lo necesario para que en un juego pueda haber varios personajes. Además define la clase arma. Realiza la modificaciones necesarias para que cada personaje pueda portar diversas armas.
Finalmente añade los métodos para que pueda clonar un juego de manera que sus subobjetos sean diferentes a los objetos originales.

__toString ( ) (con dos guiones subrayados)

Ya conocemos la sorpréndete facilidad que tiene php para convertir tipos, así por ejemplo

```
$var=3.14; //flotante
$var2="3.14"; //string
$resultado=$var2*3;  //flotante
echo $var; //convierte a string
echo $resultado//convierte a string
```

Php internamente realiza la conversión de flotante a string para imprimir por pantalla.
El método __string se ejecuta automáticamente cuando un objeto está dentro de un echo o un print
El método __toString() permite a una clase decidir cómo comportarse cuando se le trata como un string. Por ejemplo, lo que *echo $obj;* mostraría. Este método debe devolver un string, si no se emitirá un nivel de error fatal

```
<?php
// Declarar una clase simple
class TestClass{
    public $foo;
    public function __construct($foo)
    {        $this->foo = $foo;
    }
    public function __toString()
    {
        return $this->foo;
    }
}

$class = new TestClass('Hola Mundo');
echo $class;
echo substr($class, 'Hola');
?>
```

El resultado del ejemplo sería: "Hola Mundo"

Mas ejemplos

```php
<?php
class Alumno
{
  private $_nombre;
  private $apellido;
  public function __construct($nombre,$apellido)
  {
      $this->nombre = $nombre;
      $this->apellido = $apellido;
  }

   public function __toString()
   {
      return $this->nombre . ' ' .$this->apellido;
   }
}

class Escuela
{
  private $nombre;
  private $alumnos = array();

  public function __construct($nombre)
  {
      $this->nombre = $nombre;
  }
  public function addAlumno($alumno)
  {
      $this->alumnos[] = $alumno;
  }
  public function __toString()
  {
      $retorno = $this->nombre.'\n' ;

      foreach ($this->alumnos as $alumno) {
          $retorno .= $alumno .' ';
          /* Es lo mismo que decir
           * $retorno .= $alumno->__toString() .' ';
           * solo que el objeto sabe cómo convertirse en String,
           * puesto que detecta cuando se hace una operación de suma de cadenas
           * con el punto ".". operador concatenar vamos  */
      }
      return $retorno;
  }
}
$alu1 = new Alumno('Alan','Moore');
$alu2 = new Alumno('Neil','Gaiman');
$alu3 = new Alumno('Terry','Pratchett');
$escuela = new Escuela('Los Padres Salesianos');
$escuela-> addAlumno($alu1);
$escuela-> addAlumno($alu2);
$escuela-> addAlumno($alu3);
echo $escuela ;
```

Más información sobre los métodos mágicos en:
http://php.net/manual/es/language.oop5.magic.php

20 – Métodos mágicos: Más ejemplos que son y para que sirven (OPCIONAL)

¿Algo opcional en estos apuntes? Sí, al parecer sí, esto es debido a que aunque los métodos mágicos son ampliamente utilizados en php, alejan, de manera clara, a php del paradigma POO. Sin embargo bien utilizados se pueden usar para resolver varios problemas y proveer a php de herramientas no definidas como pueden ser la ausencia de sobrecarga de funciones.
Los **métodos mágicos** permiten realizar acciones en objetos cuando suceden determinados eventos que los activan. Estos métodos, denominados con doble barra baja *__metodo()*, determinan cómo reaccionará el objeto ante dichos eventos.

Los **métodos mágicos disponibles en PHP** son los siguientes:

1. construct() y destruct()
2. get() y set()
3. isset() y unset()
4. __toString()
5. sleep() y wakeup
6. call() y callStatic
7. __clone()
8. __invoke()

Nos centraremos en aquellos no conocidos

1.get() y set()

Cuando se trabaja con objetos, las propiedades pueden ser accedidas de esta forma si son **public**:

```
$tweet = new Tweet(54, "Hola que tal");
echo $tweet->text;
```

Pero en la práctica las propiedades lo normal y aconsejable es que sean **protected** o **private**, por lo que no podemos acceder a ellas de esa forma.

El método mágico **___get()_** permite poder acceder a estas propiedades:

```
public function __get($property){
    if(property_exists($this, $property)) {
        return $this->$property;
    }
}
```

El método acepta el nombre de la propiedad que estabas buscando como argumento. No es necesario llamar al método *__get()* ya que **PHP** lo llamará automáticamente cuando trates de acceder a una propiedad que no es **public**. También lo llamará cuando no existe una propiedad, por ejemplo:

```
class Datos {
    public function __get($propiedad){
        echo "Esta propiedad no existe!";
    }
}
$datos = new Datos;
$datos->noexisto; // Devuelve: Esta propiedad no existe!
```

Si lo que quieres es **establecer una propiedad que no es accesible**, puedes insertar un método ***__set()***. Este método coge la propiedad a la cual intentabas acceder y el valor que intentabas establecer como sus dos argumentos.

```
public function __set($property, $value){
    if(property_exists($this, $property)) {
        $this->$property = $value;
    }
}
$tweet->text = "Estableciendo un tweet";
echo $tweet->text; // Devuelve: "Estableciendo un tweet"
```

También podemos establecer una propiedad que no existe con ***__set()***:

```
class Datos {
    public function __set($propiedad, $valor){
        $this->propiedad = $valor;
    }
}
$datos = new Datos;
$datos->noexisto = "Ahora si que existo";
var_dump($datos);
/*
object(Datos)[1]
  public 'propiedad' => string 'Ahora si que existo' (length=19)
*/
```

Otra forma de uso es con una propiedad de tipo **array**:

```
class Datos {
    protected $datos = array();
    public function __set($nombre, $valor){
        $this->datos[$nombre] = $valor;
    }
    public function __get($nombre){
        return $this->datos[$nombre];
    }
}
$datos = new Datos;
$datos->primero = "Este es el primer dato";
$datos->segundo = "Este es el segundo dato";
$datos->tercero = "Este es el tercer dato";
echo $datos->primero . "<br>";
echo $datos->segundo . "<br>";
echo $datos->tercero;
```

Si eliminamos el método **_set()** *y añadimos el ampersand de referencia & en* get()_, podemos establecer y obtener las propiedades:

```
class Datos {
    protected $datos = array();
    public function &__get($nombre){
        return $this->datos[$nombre];
    }
}
$datos = new Datos;
$datos->primero = "Este es el primer dato";
$datos->segundo = "Este es el segundo dato";
$datos->tercero = "Este es el tercer dato";
echo $datos->primero . "<br>";
echo $datos->segundo . "<br>";
echo $datos->tercero;
```

El uso de estos dos métodos a veces **no** es recomendado por varias razones: son más lentos, no facilitan el autocompletado de código en los IDE, hacen que el mantenimiento y refactorizar sea más largo y

complicado, no es posible tener un método magic *protected*... En su lugar se pueden emplear *getters* y *setters*.
En otras palabras, te ahorras escribir getter y setters, cierto, pero a ver, ¿es eso una ventaja?¿ó acaso eso no lo hace el IDE?

2. isset() y unset()

Con *isset()* se verifica la existencia de un elemento en un array o de una propiedad *public* en un objeto.
Con **__***isset()* también es posible saber acerca de las propiedades **protected** o **private**:

```
public function __isset($property){
return isset($this->$property);
}
var_dump(isset($tweet->id));
```

También podríamos utilizar la función *isset()* dentro de la clase para obtener **true** o **false**.
Con *unset()* se eliminan los elementos de un array o propiedades públicas de un objeto.
Con ***__*unset()** *también es posible hacerlo con propiedades* **protected** *o* **private**_:

```
public function __unset($property){
    unset($this->$property);
}
```

3 sleep() y wakeup()

El método *serialize()* es una forma de guardar una **representación de un objeto**. Por ejemplo, si **guardamos un objeto en la base de datos**, primero tendríamos que **serializarlo**, guardarlo, y después si lo quisiéramos de vuelta tendríamos que usar *unserialize()*.
El método **__sleep()**_ permite definir qué propiedades del objeto deberían serializarse ya que probablemente no quieras serializar cualquier objeto externo que no sea relevante en el momento de usar *unserialize()*.
Por ejemplo, creamos un nuevo objeto en el que necesitamos determinar una conexión a una base de datos:

```
$tweet = new Tweet(12, "Hola que tal", new PDO
('mysql:host=localhost;dbname=ejemplo', 'root', 'pass'));
```

Al serializar este objeto no nos interesa serializar también la conexión a la base de datos porque no nos será relevante. Resumiendo en vez de serializar TODO el objeto, podemos utilizar para serializar aquellos atributos que nos interesen.
El método **__sleep()**_ es un método que recoge un array de propiedades que queremos que se serialicen:

```
public function __sleep() {
    return array('id', 'text');
}
```

Cuando queramos utilizar *unserialize()* en el objeto, tendremos que establecer el objeto de nuevo a su estado normal. En este ejemplo hay que **reestablecer la conexión a la base de datos**, pero en realidad podría ser establecer cualquier cosa que el objeto debiera saber. Para ello se utiliza el método **__wakeup()**_:

```
public function __wakeup() {
    $this->storage->connect();
}
```

4. call() y callStatic()

El método **___call()_** se activa cuando se intenta llamar a un método que no es accesible públicamente. Si tenemos un array de datos en el objeto que queremos mutar antes de devolver:

```
class Tweet {
    protected $id;
    protected $text;
    protected $meta;
    public function __construct($id, $text, array $meta){
        $this->id = $id;
        $this->text = $text;
        $this->meta = $meta;
    }
    protected function retweet(){
        $this->meta['retweets']++;
    }
    protected function favourite(){
        $this->meta['favourites']++;
    }
    public function __get($property){
        var_dump($this->$property);
    }
    public function __call($method, $parameters){
        if (in_array($method, array('retweet', 'favourite'))){
            return call_user_func_array(array($this, $method), $parameters);
        }
    }
}
$tweet = new Tweet(43, 'Hola que tal', array('retweets' => 23, 'favourites' =>
17));
$tweet->retweet();
$tweet->meta; // array(2) { ["retweets"]=> int(24) ["favourites"]=> int(17) }
```

Las funciones *favourite()* y _retweet()_son métodos *protected*. Con la función **___call()_** podemos acceder a ellos y opcionalmente pasarles parámetros con call_user_func_array().
Otro escenario puede darse cuando hay un **objeto inyectado** que forma parte de la API públicamente accesible. Tenemos la clase Location:

```
class Location {
    protected $latitude;
    protected $longitude;
    public function __construct($latitude, $longitude){
        $this->latitude = $latitude;
        $this->longitude = $longitude;
    }
    public function getLocation(){
        return array(
            'latitude' => $this->latitude,
            'longitude' => $this->longitude
        );
    }
}
```

Y ahora la clase Tweet:

```
class Tweet {
    protected $id;
    protected $text;
    protected $location;
    public function __construct($id, $text, Location $location){
        $this->id = $id;
```

```
        $this->text = $text;
        $this->location = $location;
    }
    public function  __call($method, $parameters){
        if(method_exists($this->location, $method)) {
            return call_user_func_array(array($this->location, $method),
$parameters);
        }
    }
}
```

En el ejemplo anterior hemos inyectado un objeto de la clase **Location** en **Tweet**, y podríamos hacer una llamada desde un objeto **Tweet** a *getLocation()*, delegando la llamada al objeto Location inyectado:

```
$tweet = new Tweet(34, 'Hola que tal', new Location('17.8912920598911', '-
189.763546289918'));
var_dump($tweet->getLocation()); // array(2) { ["latitude"]=> string(16)
"17.8912920598911" ["longitude"]=> string(17) "-189.763546289918" }
```

Si el **método** al que se quiere llamar es **static** se puede emplear **___callStatic()_**. Este método funciona de la misma forma que __call() pero la sintaxis con la que llamar a los métodos será como a los métodos estáticos, con **::**.

Menudo rollo, ¿verdad que sí? Pero aún así, sí, se entiende, todo ok, me parece bien, pero en realidad.. ¿para qué sirve? El metodo __call tienne una utilidad que es en realidad lo que nos interesa.

SIRVE PARA SOBRECARGAR MÉTODOS

A ver, a ver, pero en php esta característica NO ESTÁ IMPLEMENTADA, ¿verdad?
No, no lo está
Pero ¿podríamos usar el __call, para implementarla?. SÍ, PODRÍAMOS. **¿Cómo? Sigue leyendo la siguiente página**

SOBRECARGA DE MÉTODOS EN PHP

Seguramente si conoces otros lenguajes de programación como JAVA, este apartado te resulte extraño, ya que **PHP sobrecarga los métodos de forma muy diferente a como se hace de forma común a otros lenguajes**.
Para ello se utiliza un método llamado ***__call()*** al que se le pasa como argumentos; el nombre del método y una lista con los argumentos que este método necesitaría. Este método __call, recibe aquellos argumentos que no están definidos de forma independiente, comprobando que estén definidos en su interior.

```
function __call($metodo, $argumentos)
{
 ...
}
```

De este modo imaginemos que queremos introducir una sobrecarga en nuestro método *set_name*, donde podría introducirse el nombre y los apellidos. Deberemos de crear nuestro método ***__call*** en relación a este objetivo.

Lo primero que habría que discernir es si el método pasado por argumento es el correcto, haciéndolo con un ***if***:

```
function __call($metodo, $argumentos)
{
 if ($metodo = 'set_nombre'){
  ...
 }
}
```

De este modo controlamos que el método tenga el nombre correcto, pero ahora hay que comprobar el número de argumentos y que se debe de hacer en cada caso, ya que como se ha dicho *$argumentos* es una lista o array.

```
function __call($metodo, $argumentos)
{
 if ($metodo = 'set_nombre'){
  if (count($argumentos) == 1){
   $this->nombre = $argumentos[0];
  }
  if (count($argumentos) == 2){
   $this->nombre = $argumentos[0];
   $this->apellido = $argumentos[1];
  }
 }
}
```

El código comprueba el número de elementos enviados en la lista y dependiendo de cuantos haya crea el objeto con o sin apellido. (*Recordemos que los arrays tienen como primer elemento el cero.*)

De este modo ahora podemos comprobar como funciona con el siguiente código, donde se crean dos objetos, cada uno con un número diferente de argumentos:

```
<?php
 class Person
 {
  var $nombre;
  var $apellido;
  var $edad;

  //Funcion __call que recoge el nombre sobrecargado
  function __call($metodo, $argumentos)
  {
   if ($metodo = 'set_nombre'){
    if (count($argumentos) == 1){
     $this->nombre = $argumentos[0];
    }
    if (count($argumentos) == 2){
     $this->nombre = $argumentos[0];
     $this->apellido = $argumentos[1];
    }
   }
  }

  //Aquí comienzan los geters y seters de las propiedades.
  //Obser que no existe ningún  SET_NOMBRE definido
  function get_nombre()
  {
   return $this->nombre;
  }
  function get_apellido()
  {
   return $this->apellido;
  }

  function set_edad($dato)
  {
   $this->edad = $dato;
  }

  function get_edad()
  {
   return $this->edad;
  }
 }
 $persona1 = new Person();
//Pero si no existe set_nombre, ¿qué es lo ocurre?
// simple realizamos una llamada a __call
// que se encargará de manejar la llamada del método
// y en función del número de parámetros ejecuta un código u otro
// en este caso SET_NOMBRE coun un parámetro
 $persona1->set_nombre('Antonio');
 $persona1->set_edad('20');

 $persona2 = new Person();
// Ahora usando el call ejecutamos SET_NOMBRE con 2 parámetros

 $persona2->set_nombre('Manuel', 'Gonzalez');
 $persona2->set_edad('28');

 echo '<h3>Persona1</h3>';
 echo '<p>Mostrando con $persona1->get_nombre: ' . $persona1->get_nombre() .
```

```
'</p>';
 echo '<p>Mostrando con $persona1->get_edad: ' . $persona1->get_edad() . '</p>';
 echo '<h3>Persona2</h3>';
 echo '<p>Mostrando con $persona2->get_nombre: ' . $persona2->get_nombre() .
'</p>';
 echo '<p>Mostrando con $persona2->get_apellido: ' . $persona2->get_apellido() .
'</p>';
 echo '<p>Mostrando con $persona2->get_edad: ' . $persona2->get_edad() . '</p>';
?>
```

En este código podemos ver como esta incluido nuestro método sobrecargado, y como creamos los dos objetos, el que solo lleva un argumento y el que lleva los dos. Después solicitamos los datos de ambos objetos y obtenemos el siguiente resultado:

Persona1

```
Mostrando con $persona1->get_nombre: Antonio
Mostrando con $persona1->get_edad: 20
```

Persona2

```
Mostrando con $persona2->get_nombre: Manuel
Mostrando con $persona2->get_apellido: Gonzalez
Mostrando con $persona2->get_edad: 28
```

Como dije al principio, la forma de realizar los métodos sobrecargados en PHP es algo distinta a como se hace en otros lenguajes, pero no es mucho más difícil, tan solo hay que saber controlar el flujo a través de las instrucciones ***if***. Pero si reflexionamos sobre esto, llegamos a la conclusión de que con un método ***__call()*** se pueden controlar todos los métodos para crear un objeto… pero eso lo dejo en vuestras manos…

5. __invoke()

El método mágico **___invoke()_** permite **usar un objeto como si fuera una función**.

```
class User {
    protected $name;
    protected $timeline = array();
    public function __construct($name){
        $this->name = $name;
    }
    public function addTweet(Tweet $tweet){
        $this->timeline[] = $tweet;
    }
}
class Tweet {
    protected $id;
    protected $text;
    protected $read;
    public function __construct($id, $text){
        $this->id = $id;
        $this->text = $text;
        $this->read = false;
    }
    public function __invoke($user){
        $user->addTweet($this);
        return $user;
    }
}
```

Ahora podremos usar un objeto Tweet como **callable**:

```
// Array de usuarios
$users = array(new User('Pepa'), new User('Pepe'), new User('John'));
// Nuevo tweet
$tweet = new Tweet(53, 'Hola que tal');
// Aplica __invoke() en $users
$users = array_map($tweet, $users);
// Al mostrar los usuarios, todos tienen un array en el timeline con el mismo
objeto, el tweet
var_dump($users);
// Aplicándolo sólo a un usuario
var_dump($tweet($users[0]));
```

6. __autoload()

El método mágico **___autoload** … un momento, un momento, ¿cómo que autoload? Pero, si ese método no estaba en la tabla del inicio del tema, ¿acaso me quieres hacer trabajar más?
Vale, me explico.
Primero: estrictamente hablando la página oficial de PHP no considera el método **__autoload** como un método mágico de clase (por eso no está en tabla), mas al ser un método que funciona de manera similar y se usa MUUUUCHOOOOO en PHP, vamos a ver que es y como se implementa.
Segundo: además, el **__autoload** se considera decrepado desde la version 7.2.0 de php, pero su funcionalidad se realiza de manera alternativa con función spl_autoload_register() .
Vayamos a lo que no interesa.

Cómo registrar y utilizar autoloaders en PHP

Imaginemos que organizamos nuestro código en diferentes archivos, cada uno con una clase y con el nombre de la clase como nombre del archivo. Por ejemplo, `UnaClase.php` y `OtraClase.php` contendrán las definiciones de `class UnaClase` y `class OtraClase` repectivamente. Para inicializar estas clases desde otro script podríamos hacer algo así:

```
// incluir los archivos que contienen las clases
include_once 'UnaClase.php';
include_once 'OtraClase.php';

// inicializar los objetos
$objeto1 = new UnaClase();
$objeto2 = new OtraClase();
```

Todo perfecto. Sigues trabajando en la aplicación y vas definiendo nuevas clases en cada vez más archivos que tienes acordarte de incluir en tu script:

```
// incluir los archivos que contienen las clases
include_once 'UnaClase.php';
include_once 'OtraClase.php';
include_once 'MasClase.php';
include_once 'CuartaClase.php';
include_once  'YtodaviamasClase.php';

// inicializar los objetos
$objeto1 = new UnaClase();
$objeto2 = new OtraClase();
$objeto3 = new MasClase();
//....
```

No hay nada de malo en el código anterior, funcionará perfectamente. Asegúrate de incluir todos los archivos y no tendrás ningún problema. Pero, ¿y si no siempre los necesito todos?. Da igual, cárgalos todos por si acaso o haz condicionales para su carga. ¿Y si tengo 50 archivos? Pues tendrás que hacer 50 `include`. Vaya rollo, que incomodidad; y como se me olvide algún archivo …. Fatal ERROR al canto.
Alguien puede pensar, me creo una librería con todos los includes_once de todas las clases, lo incluyo en

todos mis ficheros y así, NUNCA NUNCA NUNCA se me olvidará la definición de alguna clase y me ahorraré consiguientes errores
Pero un include con todos los includes supone la carga de definición de clase de todas las clases..
¿Y si tengo 50 clases y sólo voy a utilizar 2? ¿Afectará eso al rendimiento?. Apostaría a que sí.

Autoloaders al rescate!!!!

En lugar de tener que cargar (en inglés *load*) cada una de las clases de nuestra aplicación manualmente, podemos **registrar un autoloader**. Como su nombre indica, el autoloader está destinado a cargar de forma automática las clases utilizadas. Cada vez que se intenta inicializar una clase y la clase no existe, el nombre de esta clase se pasa al autoloader y este es ejecutado. En el autoloader podremos automatizar el proceso de carga sin tener que incluir manualmente cada archivo y además nos permite hacer el código más rápido, pues sólo se cargarán las clases que efectivamente se utilicen.
En otras palabras lo que hace el autoloader es cargar automáticamente (autoload) la clase en el momento en que se hace new o se va utilizar (en el caso de estáticos) nunca antes, con lo cual sólo cargaré en memoria las clases que vaya utilizando. Menos carga→ Mayor rendimiento.
Además si defino bien la forma, solamente siguiendo una simples reglas, nunca olvidaré escribir un include (por que no lo hago)
Veamos como funciona.
Para registrar un autoloader se utiliza la función spl_autoload_register() pasando como primer parámetro la función que hará de autoloader: Como puedes ver pasamos del __autoload ya que está decrepado en versiones actuales.

```
//Registramos mi_autoloader() como callback que se ejecutará
//al intentar inicializar una clase que no existe
spl_autoload_register('mi_autoloader');

//implementamos mi_autoloader(), que recibe el nombre de la clase
//que se ha intentado inicializar
function mi_autoloader( $NombreClase ) {
    include_once $NombreClase . '.class.php';
}
```

Como puedes observar el truco está en tener definida el nombre de fichero con un formato determinado, en este ejemplo si la clase es Camion, debo guardar el fichero de clase como NombreClase.class.php o sea Camion.class.php. Lo de class, lo ponemos para que de un vistazo ver que es una definición de clase, pero no es necesario, usa el formato que te apetezca.

Desde PHP 5.3.0 podemos utilizar funciones anónimas:

```
spl_autoload_register( funcion( $NombreClase ) {
    include_once $NombreClase . '.class.php';
} );

// inicializar los objetos
$objeto1 = new UnaClase();
$objeto2 = new OtraClase();

if( $alguna_condicion ) {
    // Ganamos en performance. MasClase.class.php solo se cargará
    // si se verifica $alguna_condicion y llegamos hasta este punto
    $objeto3 = new MasClase();
}
```

Cómo ves, es necesario conocer los nombres de los archivos que se necesitan cargar. Por eso es muy importante seguir unas **reglas fijas de nomenclatura de clases y archivos**. El ejemplo anterior de utilizar una sola clase por cada archivo y dar al archivo el mismo nombre que la clase sería una regla aceptable. Puedes seguir tus propias reglas aunque es recomendable utilizar reglas estandarizadas y de amplio uso, por ejemplo el estándar PSR-4.

Y Olvídate de __autoload

Si no defines tu propio autoloader como hemos hecho anteriormente, puedes utilizar el método mágico `__autoload()`:

```
/**********NO USAR ************/
function __autoload( $nombreclass ) {
    //incluir el archivo de la clase
    include_one $nombreclass . '.php';
}

// inicializar los objetos
$objeto1 = new UnaClase();
$objeto2 = new OtraClase();
```

Se puede decir que `__autoload` sería el autoloader por defecto. Sin embargo, tal y como se menciona en la documentación de PHP sobre autocarga de clases, se recomienda `spl_autoload_register` por su mayor flexibilidad, pero además advierte que **`__autoload`** se considera decrepado desde la versión 7.2.0 o sea que usa siempre `spl_autoload_register`.

21 - Operador instanceof

Cuando tenemos una lista de objetos de distinto tipo y queremos saber si un objeto es de una determinada clase el lenguaje PHP nos provee del operador instanceof.
Confeccionaremos un problema que contenga un vector con objetos de la clase Empleado y Gerente. Luego calcularemos cuanto ganan en total los empleados y los gerentes.

```
<html>
<head>
<title>Pruebas</title>
</head>
<body>
<?php
abstract class Trabajador {
  protected $nombre;
  protected $sueldo;
  public function __construct($nom,$sue)
  {
    $this->nombre=$nom;
    $this->sueldo=$sue;
  }
  public function retornarSueldo()
  {
    return $this->sueldo;
  }
}

class Empleado extends Trabajador {
}

class Gerente extends Trabajador {
}

$vec[]=new Empleado('juan',1200);
$vec[]=new Empleado('ana',1000);
$vec[]=new Empleado('carlos',1000);

$vec[]=new Gerente('jorge',25000);
$vec[]=new Gerente('marcos',8000);
$suma1=0;
$suma2=0;
for($f=0;$f<count($vec);$f++)
{
  if ($vec[$f] instanceof Empleado)
    $suma1=$suma1+$vec[$f]->retornarSueldo();
  else
    if ($vec[$f] instanceof Gerente)
      $suma2=$suma2+$vec[$f]->retornarSueldo();
}
echo 'Gastos en sueldos de Empleados:'.$suma1.'<br>';
echo 'Gastos en sueldos de Gerentes:'.$suma2.'<br>';

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

Hemos planteado tres clases, la clase Trabajador es una clase abstracta:

```
abstract class Trabajador {
...
}
```

Las clases Empleado y Gerente son subclase de la clase Trabajador y en este caso por simplicidad no agregan ninguna funcionalidad a la clase padre:

```
class Empleado extends Trabajador {
}

class Gerente extends Trabajador {
}
```

Ahora veamos la parte que nos interesa, primero creamos 5 objetos, 3 de la clase Empleado y 2 de la clase Gerente:

```
$vec[]=new Empleado('juan',1200);
$vec[]=new Empleado('ana',1000);
$vec[]=new Empleado('carlos',1000);

$vec[]=new Gerente('jorge',25000);
$vec[]=new Gerente('marcos',8000);
```

Como podemos ver los 5 objetos se almacenan en un vector. Ahora tenemos que ver cuanto se gasta en sueldos pero separando lo que ganan los empleados y los gerentes:

```
$suma1=0;
$suma2=0;
for($f=0;$f<count($vec);$f++)
{
  if ($vec[$f] instanceof Empleado)
    $suma1=$suma1+$vec[$f]->retornarSueldo();
  else
    if ($vec[$f] instanceof Gerente)
      $suma2=$suma2+$vec[$f]->retornarSueldo();
}
```

Mediante el operador instanceof preguntamos por cada elemento del vector y verificamos si se trata de una instancia de la clase Empleado o de la clase Gerente.
Finalmente mostramos los acumuladores:

```
echo 'Gastos en sueldos de Empleados:'.$suma1.'<br>';
echo 'Gastos en sueldos de Gerentes:'.$suma2.'<br>';
```

Problema:

Plantear una clase Trabajador. Definir como atributos su nombre y sueldo. Declarar un método que retorne el sueldo y otro el nombre.
Plantear una subclase Empleado y otra subclase Gerente.
Crear un vector con 3 empleados y 2 gerentes. Mostrar el nombre y sueldo del gerente que gana más en la empresa

22 - Método destructor de una clase (__destruct)

Otro método que se ejecuta automáticamente es el __destruct (destructor de la clase)
Las características de este método son:

- El objetivo principal es liberar recursos que solicitó el objeto (conexión a la base de datos, creación de imágenes dinámicas etc.)
- Es el último método que se ejecuta de la clase.
- Se ejecuta en forma automática, es decir no tenemos que llamarlo.
- Debe llamarse __destruct.
- No retorna datos.
- Es menos común su uso que el constructor, ya que PHP gestiona bastante bien la liberación de recursos en forma automática.

Para ver su sintaxis e implementación confeccionaremos el siguiente problema: Implementar una clase Banner que muestre un texto generando un gráfico en forma dinámica. Liberar los recursos en el destructor. En el constructor recibir el texto publicitario.

```
<?php
class Banner {
  private $ancho;
  private $alto;
  private $mensaje;
  private $imagen;
  private $colorTexto;
  private $colorFondo;
  public function __construct($an,$al,$men)
  {
    $this->ancho=$an;
    $this->alto=$al;
    $this->mensaje=$men;
    $this->imagen=imageCreate($this->ancho,$this->alto);
    $this->colorTexto=imageColorAllocate($this->imagen,255,255,0);
    $this->colorFondo=imageColorAllocate($this->imagen,255,0,0);
    imageFill($this->imagen,0,0,$this->colorFondo);
  }
  public function mostrar()
  {
    imageString ($this->imagen,5,50,10, $this->mensaje,$this->colorFuente);
    header ("Content-type: image/png");
    imagePNG ($this->imagen);
  }
  public function __destruct()
  {
    imageDestroy($this->imagen);
  }
}

$baner1=new Banner(428,45,'Sistema de Ventas por Mayor y Menor');
$baner1->mostrar();
?>
```

Se trata de un archivo PHP puro ya que se genera una imagen PNG y no un archivo HTML.
Al constructor llega el texto a imprimir y el ancho y alto de la imagen. En el constructor creamos el manejador para la imagen y creamos dos colores para la fuente y el fondo del banner.

```
  public function __construct($an,$al,$men)
  {
    $this->ancho=$an;
    $this->alto=$al;
    $this->mensaje=$men;
    $this->imagen=imageCreate($this->ancho,$this->alto);
    $this->colorTexto=imageColorAllocate($this->imagen,255,255,0);
```

```
    $this->colorFondo=imageColorAllocate($this->imagen,255,0,0);
    imageFill($this->imagen,0,0,$this->colorFondo);
  }
```

El método mostrar genera la imagen dinámica propiamente dicha:

```
  public function mostrar()
  {
    imageString ($this->imagen,5,50,10, $this->mensaje,$this->colorFuente);
    header ("Content-type: image/png");
    imagePNG ($this->imagen);
  }
```

Y por último tenemos el destructor que libera el manejador de la imagen:

```
  public function __destruct()
  {
    imageDestroy($this->imagen);
  }
```

Cuando creamos un objeto de la clase Banner en ningún momento llamamos al destructor (se llama automáticamente previo a la liberación del objeto:

```
$baner1=new Banner(428,45,'Sistema de Ventas por Mayor y Menor');
$baner1->mostrar();
```

Problema:

Confeccionar una clase que contenga un constructor y un destructor. Hacer que cada método imprima un mensaje en la página indicando que se a ejecutado dicho método.

23 - Métodos estáticos de una clase (static)

Un método estático pertenece a la clase pero no puede acceder a los atributos de una instancia. La característica fundamental es que un método estático se puede llamar sin tener que crear un objeto de dicha clase.
Un método estático es lo más parecido a una función de un lenguaje estructurado. Solo que se lo encapsula dentro de una clase.
Confeccionaremos una clase Cadena que tenga una conjunto de métodos estáticos:

```
<html>
<head>
<title>Pruebas</title>
</head>
<body>
<?php
class Cadena {
  public static function largo($cad)
  {
    return strlen($cad);
  }
  public static function mayusculas($cad)
  {
    return strtoupper($cad);
  }
  public static function minusculas($cad)
  {
    return strtolower($cad);
  }
}

$c='Hola';
echo 'Cadena original:'.$c;
echo '<br>';
echo 'Largo:'.Cadena::largo($c);
echo '<br>';
echo 'Toda en mayúsculas:'.Cadena::mayusculas($c);
echo '<br>';
echo 'Toda en minúsculas:'.Cadena::minusculas($c);
?>
</body>
</html>
```

Para definir un método estático utilizamos la palabra clave static luego del modificador de acceso al método:

```
  public static function largo($cad)
  {
    return strlen($cad);
  }
```

Hay que tener en cuenta que un método estático no puede acceder a los atributos de la clase, ya que un método estático normalmente se lo llama sin crear un objeto de dicha clase:

```
echo 'Largo:'.Cadena::largo($c);
```

La sintaxis para llamar un método estático como vemos es distinta a la llamada de métodos de un objeto. Indicamos primero el nombre de la clase, luego el operador '::' y por último indicamos en nombre del método estático a llamar.

Problema:

Plantear una clase Calculadora que contenga cuatro métodos estáticos (sumar, restar, multiplicar y dividir) estos métodos reciben dos valores.

24 – Constantes, Atributos estáticos, modificadores self, this y otras consideraciones.

Constantes

Las **constantes** son **valores invariables**. Se definen mediante la palabra **const** y se obtienen utilizando el operador **double colon**: _$objeto::CONSTANTE _o *NombreClase::CONSTANTE*. No van precedidas del símbolo $ y se escriben en mayúscula:

```
class Coche {
    const RUEDAS = 4;
}
// Obtener el valor mediante el nombre de la clase:
echo Coche::RUEDAS . "\n";
// Obtener el valor mediante el objeto:
$miCoche = new Coche();
echo $miCoche::RUEDAS . "\n";
```

Paamayim Nekudotayim (::)

A veces es útil hacer referencia a variables o funciones en clases base, o referenciar funciones en clases que aún no tienen instancias. El Operador de Resolución (::) también conocido como Paamayim Nekudotayim (significa doble-dos-puntos en Hebreo) se usa para ello.

```
<?php
     class MyClass
     {
          protected function myFunc() {
               echo "MyClass::myFunc()\n";
          }
     }
      class OtherClass extends MyClass
     {
          // Sobrescritura de definición parent
          public function myFunc()
          {
               // Pero todavía se puede llamar a la función parent
               parent::myFunc();
               echo "OtherClass::myFunc()\n";
          }
     }
      $class = new OtherClass();
     $class->myFunc();
      /*
     La salida por pantalla será:
          MyClass::myFunc()
          OtherClass::myFunc()
     */
?>
```

self y parent

Cuando queramos acceder a una constante o método estático desde dentro de la clase, usamos la palabra reservada: self.

Cuando queramos acceder a una constante o método de una clase padre, usamos desde la clase extendida la palabra reservada: parent. Un caso típico es cuando en una clase extendida se sobrescribe el mismo método eliminando las definiciones y cambiando su visibilidad del método de la clase padre, como en el ejemplo anterior.

```
<?php
    class MiClase {
        const CONSTANTE = 'Hola! ';
    }
  static class Clase2 extends MiClase
    {
        public static $variable = 'static';
```

```
        public static function miFuncion() {
             echo parent::CONSTANTE;
             echo self::$variable;
        }
    }
     /*     La salida por pantalla será:
        Hola! static*/
?>
```

Diferencia entre $this y self::

Uno usa $this para hacer referencia al objeto (instancia) actual, y se utiliza self:: para referenciar a la clase actual. Se utiliza $this->nombre para nombres no estáticos y se utiliza self::nombres para nombres estáticos.

```
<?php
    class funcion {
        private $valor_no_estático = 1;
        private static $valor_estático = 2;

        function __construct() {
             echo $this->valor_no_estático . ' ' . self::$valor_estático;
        }
    }
?>
```

En la última entrega de POO habíamos visto los modificadores de acceso dentro de ellos los estáticos. Ahora ampliaremos un poco más sobre el tema de métodos y atributos estáticos (static) Considere ampliar el tema más bien porque tuve que resolver algunos problemas que tenían que ver con los miembros static.
Primero diremos que los miembros static también reciben el nombre de **-elementos de clase-**, porque no pertenecen a un objeto (instancia de clase) en concreto, si no a la clase como entidad.
En PHP los elementos estáticos se definen de la siguiente manera:

```
<?php
class Ejemploestático
{
        private static $atributoestático   = 'Soy un atributo estático';

        public static function metodoestático()
        {
                echo 'Soy un metodo estático';
        }
}
```

Habíamos visto que una de las particularidades del modificador static es que no se necesita de una instancia de la clase (objeto) para poder acceder a ellos, esto trae como consecuencia que la seudo-variable $this no está disponible dentro de un método declarado como static, pues esta ultima hace referencia a una instancia (objeto) y no siempre disponemos de una instancia en la llamada a métodos estáticos, para lograr este cometido se usa self::.
Entonces para acceder al atributo estático desde la clase usaríamos self::$atributoestático;
Aparte un método estático tampoco puede acceder a los atributos de la clase que no sean estáticos, ya que un método estático normalmente se lo llama sin crear un objeto de dicha clase (instancia) por lo cual no estarían disponibles los demás atributos.
Complementaremos nuestra clase para ver el comportamiento de los miembros static, además activaremos los errores de tipo E_STRICT .

```
<?php
error_reporting( E_ALL | E_STRICT );
class Metodosestáticos
{
        private static $atributoestático   = 'Soy un atributo estático';
        public static  $atrestáticoPublico = 'Soy estático pero publico';
        private $atributoNoestático        = 'No soy estático';

        public static function metodoestático()
```

```
        {
                echo '<br />' . self::$atributoestático . ' accedido por self::';
        }

        public static function metodoestáticoThis()
        {
                self::metodoNoestático();
                $this->metodoNoestático();
    }


public static function accederAtributoNoestático()
    {
        echo '<br />' . $this->atributoNoestático;
    }

        public function metodoNoestático()
        {
                echo '<br />' . self::$atributoestático;
                echo '<br />' . $this->atributoestático;
        }
}
```

Primero veremos qué pasa si hacemos llamadas sin instanciar la clase o sea sin crear objetos de la clase. Recordemos que para acceder se utiliza NombreDeLaClase::$atributoestático;

SIN USAR INSTANCIA DE CLASE

- `echo Metodosestáticos::$atributoestático;`

Resultado: `Fatal error: Cannot access private property Metodosestáticos::$atributoestático` **in...**

(Falla pues no se puede llamar a un atributo privado desde fuera de la clase, este comportamiento es el mismo que si fuera un atributo privado no estático).

- echo Metodosestáticos::$atrestáticoPublico;

Resultado: Soy Estático pero público (Correcto pues si se puede acceder directamente sin instanciar la clase a un atributo estático y publico).

- `Metodosestáticos::metodoestático();`

Resultado: Soy un atributo estático accedido por self:: (Correcto, no se necesita de una instancia para acceder a un método estático, además el método accede a la variable estática por self y no por this).

- `Metodosestáticos::metodoNoestático();`

Resultado: `Strict Standards: Non-static method Metodosestáticos::metodoNoestático() should not be called statically in …`

Soy un atributo estático

`Fatal error: Using $this when not in object context in …`

Si no hubiéramos configurado el error tipo E_STRICT la primera parte Strict Standard… no se desplegaría en pantalla, pues nos indica que si bien PHP permite acceder de un método no estático a un atributo estático con self, no es apropiado realizarlo en un entorno orientado a objetos y menos desde métodos no estáticos que no deberían estar disponibles sin una instancia (Creo que en JAVA no Compilaría).
La segunda parte del error el Fatal Error … es claro se intenta usar el operador $this-> que solo está disponible cuando existe una instancia de la clase (es decir se uso new).

- `Metodosestáticos::metodoestáticoThis();`

`Resultado: Strict Standards: Non-static method Metodosestáticos::metodoNoestático() should not be called statically in …`

(En este caso nuevamente nos dice que no se debe llamar de un método estático a un método no estático, pero nos deja continuar)
Ahora cambiemos en el método metodoestáticoThis();

```
self::metodoNoestático();
Por
$this->metodoNoestático();
```

Y realizamos nuevamente la llamada anterior:

```
Metodosestáticos::metodoestáticoThis();
```

Resultado: `Fatal error: Using $this when not in object context in ...`
(Nuevamente no se permite el uso de $this si no tenemos una instancia).

USANDO INSTANCIA DE CLASE

Al realizar una instancia de clase, tendremos disponible $this y el operador de flecha (->) para acceder a nuestros atributos y métodos, entonces realicemos la instancia primero

```
$instancia = new Metodosestáticos();
```

Una vez que tenemos la instancia veremos qué pasa en las situaciones anteriores.

```
$instancia->atributoestático;
```

Resultado: `Fatal error: Cannot access private property Metodosestáticos::$atributoestático in …`

(Falla por el mismo motivo que sin instancia, no se puede acceder a un atributo privado desde fuera de la clase).

```
$instancia->atrestáticoPublico;
```

Resultado: `Strict Standards: Accessing static property Metodosestáticos::$atrestáticoPublico as non static in ..`
`Notice: Undefined property: Metodosestáticos::$atrestáticoPublico in …`

(Si no hubiéramos mostrado todos los errores, no hubiéramos obtenido ningún mensaje, simplemente no podemos acceder de una instancia directamente a un atributo estático por más que sea público. Es decir no se puede acceder a través de una instancia a atributos estáticos no podemos usar -> para acceder a atributos estáticos).

```
$instancia->metodoestático();
```

Resultado: Soy un atributo estático accedido por self:: (En este caso no cometemos error pues accedemos a un método estático a través de una instancia y este último es el que accede a la variable mediante self)

```
$instancia->metodoNoestático();
```

Resultado: Soy un atributo estático
`Notice: Undefined variable: atributoestático in…`
`Fatal error: Cannot access empty property in …`

(Aquí los errores se producen cuando queremos acceder a través de $this al atributo estático, como hemos visto lo correcto es accederlos con el operador self:: , puesto que no se puede acceder a atributos estáticos con -> , ni con $this). Lo correcto es que los atributos privados static sólo serán accesibles desde la propia clase y por métodos static.

```
$instancia->metodoestáticoThis();
```

`Resultado: Strict Standards: Non-static method`
Metodosestáticos::`metodoNoestático() should not be called statically in … on line 16`

(Si bien aquí PHP nos permite realizar eso, no es lo recomendable pues métodos Estáticos no deberían llamar a métodos no estáticos. Aparte que puede que en un futuro este tipo de errores se tomen de distinta manera y pueda no funcionar nuestra rutina)

Soy un atributo estático
Fatal error: `Using $this when not in object context in …. on line 28` **(No se puede accede a métodos estáticos a través de $this).**

Tener en Cuenta que:

- Llamar a un método no estático desde un método estático genera una advertencia de tipo E_STRICT. (Tendríamos que evitar este tipo de advertencia, para compatibilidad con futuras versiones de PHP).
- Para llamar atributos estáticos dentro de la clase, se debe realizar con el operador self:: en ningún caso utilizar $this ni -> (flechita) para realizar ese tipo de acceso.

Como se comporta un Atributo estático

Un atributo estático es único para todas las instancias de una clase.
Cuando creamos varios objetos de una misma clase, los atributos pertenecen a cada uno de los objetos creados y sus valores son independientes de los que tienen los demás objetos, ahora si nosotros quisiéramos que un atributo sea igual en todos los objetos creados, entonces definiríamos a esa propiedad como static. Un cambio en el atributo estático en una instancia afectara a todas las instancias de la clase.
Preste mas atención al tema de errores dado que en el capitulo anterior vimos el funcionamiento con un ejemplo. Igual ahora presentaremos otro ejemplo, en el cual una clase deberá contar cuantas instancias se han realizado de ella misma por lo cual necesitamos un atributo que tenga el mismo valor para todas las instancias de la clase; para ello definimos la variable cantidadInstancias como static.

La cuenta la haremos en el constructor sumándole uno a la variable cada vez que la instanciemos.

```
class estático
{
        private static $cantidadInstancias = 0;
         public function __construct()
        {
                self::$cantidadInstancias++;
        }

        public static function getInstancias()
        {
                return self::$cantidadInstancias;
        }
}
echo '<br/>Acceso estático: ' . estático::getInstancias();      // (0)
$instancia_01   = new estático();
echo '<br/>Acceso estático: ' . estático::getInstancias();      // (1)
echo '<br/>Instancia   Uno: ' . $instancia_01->getInstancias(); // (2)
$instancia_02   = new estático();
echo '<br/>Acceso estático: ' . estático::getInstancias();      // (3)
echo '<br/>Instancia   Uno: ' . $instancia_01->getInstancias(); // (4)
echo '<br/>Instancia   Dos: ' . $instancia_02->getInstancias(); // (5)
#####################################################################
# SALIDA                                                            #
#####################################################################
Acceso estático: 0      (0)
Acceso estático: 1      (1)
Instancia Uno:   1      (2)
Acceso estático: 2      (3)
Instancia Uno:   2      (4)
Instancia Dos:   2      (5)
```

En el ejemplo podemos ver como antes de instanciar (0) el valor es cero, es decir no tenemos ninguna instancia de la clase.
Luego que instanciamos si accedemos por la instancia o por el acceso estático para ambos nos da 1 ptos (1)(2).

Instanciamos nuevamente y vemos como el valor para la primera instancia cambia y es igual que el valor para la segunda instancia, o sea el valor es único para todas las instancias de la clase y no para cada objeto.

Cuando Usar Métodos estáticos

- Deben ser usados cuando estos no modifican variables de instancia. Es decir que no cambian el estado del objeto. Generalmente reciben algún dato, realizan algún proceso y retornan una respuesta. En este caso no se deberían relacionar con otros métodos de la clase salvo que sean estáticos.
- Hay personas que utilizan este tipo de métodos cuando en la clase solo tienen un método.
- Una aplicación muy común de este tipo de métodos es el patrón Singleton.

Otras Consideraciones

- Muchos utilizan métodos estáticos aduciendo que tienen un mejor rendimiento que métodos no estáticos, si bien esto es cierto la diferencia es ínfima y no se debería de implementar un sistema basado en objetos teniendo en cuenta estos detalles, si no que deberíamos fijarnos en la responsabilidad de cada clase y definir que tipo de método será cada uno.
- No deberíamos realizar una clase con solo métodos estáticos, pues lo que estaríamos haciendo es un repositorio de funciones dentro de una clase. Esto es igual que programar de manera estructurada con funciones y estaríamos desaprovechando los beneficios de la programación orientada a objetos.

25 - Interfaces

Las interfaces son un sistema bastante común, utilizado en programación orientada a objetos. Son algo así como declaraciones de funcionalidades que tienen que cubrir las clases que implementan las interfaces.
En una interfaz se definen habitualmente un juego de funciones que deben codificar las clases que implementan dicha interfaz. De modo que, cuando una clase implementa una interfaz, podremos estar seguros que en su código están definidas las funciones que incluía esa interfaz.

A la hora de programar un sistema, podemos contar con objetos que son muy diferentes y que por tanto no pertenecen a la misma jerarquía de herencia, pero que deben realizar algunas acciones comunes. Por ejemplo, todos los objetos con los que comercia unos grandes almacenes deben contar con la funcionalidad de venderse. Una mesa tiene poco en común con un calefactor o unas zapatillas, pero todos los productos disponibles deben implementar una función para poder venderse.

Otro ejemplo. Una bombilla, un coche y un ordenador son clases muy distintas que no pertenecen al mismo sistema de herencia, pero todas pueden encenderse y apagarse. En este caso, podríamos construir una interfaz llamada "encendible", que incluiría las funcionalidades de encender y apagar. En este caso, la interfaz contendría dos funciones o métodos, uno encender() y otro apagar().

Cuando se define una interfaz, se declaran una serie de métodos o funciones sin especificar ningún código fuente asociado. Luego, las clases que implementen esa interfaz serán las encargadas de proporcionar un código a los métodos que contiene esa interfaz. Esto es seguro: si una clase implementa una interfaz, debería declarar todos los métodos de la interfaz. Si no tenemos código fuente para alguno de esos métodos, por lo menos debemos declararlos como abstractos y, por tanto, la clase también tendrá que declararse como abstracta, porque tiene métodos abstractos.

Código para definir una interfaz

Veamos el código para realizar una interfaz. En concreto veremos el código de la interfaz encendible, que tienen que implementar todas las clases cuyos objetos se puedan encender y apagar.

```
interface encendible{
    public function encender();
    public function apagar();
}
```

Vemos que para definir una interfaz se utiliza la palabra clave interface, seguida por el nombre de la interfaz y, entre llaves, el listado de métodos que tendrá. Los métodos no se deben codificar, sino únicamente declararse.

Implementación de interfaces

Ahora veamos el código para implementar una interfaz en una clase.

```
class bombilla implements encendible{
    public function encender(){
       echo "<br>Y la luz se hizo...";
    }

    public function apagar(){
       echo "<br>Estamos a oscuras...";
    }
}
```

Para implementar una interfaz, en la declaración de la clase, se debe utilizar la palabra implements, seguida del nombre de la interfaz que se va a implementar. Se podrían implementar varias interfaces en la misma clase, en cuyo caso se indicarían todos los nombres de las interfaces separadas por comas.

En el código de la clase estamos obligados a declarar y codificar todos los métodos de la interfaz.

NOTA: en concreto, PHP 7 entiende que si una clase implementa una interfaz, los métodos de esa interfaz estarán siempre en la clase, aunque no se declaren. De modo que si no los declaramos explícitamente, PHP 7 lo hará por nosotros. Esos métodos de la interfaz serán abstractos, así que la clase tendrá que definirse como abstracta.

Ahora veamos el código de la clase coche, que también implementa la interfaz encendible. Este código lo hemos complicado un poco más.

```
class coche implements encendible{
    private $gasolina;
    private $bateria;
    private $estado = "apagado";
    function __construct(){
       $this->gasolina = 0;
       $this->bateria = 10;
    }

    public function encender(){
       if ($this->estado == "apagado"){
          if ($this->bateria > 0){
            if ($this->gasolina > 0){
                $this->estado = "encendido";
                $this->bateria --;
                echo "<br><b>Enciendo...</b> estoy encendido!";
             }else{
                echo "<br>No tengo gasolina";
             }
          }else{
             echo "<br>No tengo batería";
          }
       }else{
          echo "<br>Ya estaba encendido";
       }
    }

    public function apagar(){
       if ($this->estado == "encendido"){
          $this->estado = "apagado";
          echo "<br><b>Apago...</b> estoy apagado!";
       }else{
          echo "<br>Ya estaba apagado";
       }
    }

    public function cargar_gasolina($litros){
       $this->gasolina += $litros;
       echo "<br>Cargados $litros litros";
    }
}
```

A la vista del anterior código, se puede comprobar que no hay mucho en común entre las clases bombilla y coche. El código para encender una bombilla era muy simple, pero para poner en marcha un coche tenemos que realizar otras tareas. Antes tenemos que ver si el coche estaba encendido previamente, si tiene gasolina y si tiene batería. Por su parte, el método apagar hace una única comprobación para ver si estaba o no el coche apagado previamente.

También hemos incorporado un constructor que inicializa los atributos del objeto. Cuando se construye un coche, la batería está llena, pero el depósito de gasolina está vacío. Para llenar el depósito simplemente se debe utilizar el método cargar_gasolina().

Llamadas polimórficas pasando objetos que implementan una interfaz

Las interfaces permiten el tratamiento de objetos sin necesidad de conocer las características internas de ese objeto y sin importar de qué tipo son... simplemente tenemos que saber que el objeto implementa una interfaz.
Por ejemplo, tanto los coches como las bombillas se pueden encender y apagar. Así pues, podemos llamar al método encender() o apagar(), sin importarnos si es un coche o una bombilla lo que hay que poner en marcha o detener.
En la declaración de una función podemos especificar que el parámetro definido implementa una interfaz, de modo que dentro de la función, se pueden realizar acciones teniendo en cuenta que el parámetro recibido implementa un juego de funciones determinado.
Por ejemplo, podríamos definir una función que recibe algo por parámetro y lo enciende. Especificaremos que ese algo que recibe debe de implementar la interfaz encendible, así podremos llamar a sus métodos enciende() o apaga() con la seguridad de saber que existen.

```
function enciende_algo (encendible $algo){
    $algo->encender();
}

$mibombilla = new bombilla();
$micoche = new coche();

enciende_algo($mibombilla);
enciende_algo($micoche);
```

Si tuviéramos una clase que no implementa la interfaz encendible, la llamada a esta función provocaría un error. Por ejemplo, un CD-Rom no se puede encender ni apagar.

```
class cd{
    public $espacio;
}
$micd = new cd();
enciende_algo($micd); //da un error. cd no implementa la interfaz encendible
```

Esto nos daría un error como este:

Fatal error: `Argument 1 must implement interface encendible in c:\www\ejphp5\funcion_encender.php on line 6.`

Queda muy claro que deberíamos implementar la interfaz encendible en la clase cd para que la llamada a la función se ejecute correctamente.

Resumen diferencias entre interfaces y clases abstractas
Vamos a ver un resumen de las principales diferencias a nivel conceptual:
A)
- Clase abstracta debe de contener como mínimo un método abstracto (abstract), el cual solo solo se especifica el nombre y no se implementa. Los demás métodos pueden estar completamente implementados
- En una interfaz no es posible implementar métodos sino solo definir su nombre.
B)
- En clase abstracta un método abstract se puede definir public, protected o private. Las subclases que heredan de la clase padre tienen que implementar estos métodos con la misma visibilidad o una con menor restricción
- En una interfaz todos los métodos son públicos.
C)
- En una clase abstracta, a parte de los métodos, se pueden definir atributos, con su visibilidad, y constantes.
- En una interfaz, aparte de los métodos, únicamente se puede definir constantes.

D)
- Una clase puede heredar solo de una clase padre.
- Una clase puede implementar más de una interfaz.

E)
- Usando clases abstractas, una clase hija puede sobrescribir o no un método definido en la clase padre. Evidentemente si el método es abstracto si que está obligada a implementarlo y por lo tanto sobrescribirlo.
- Una clase que implementa una interfaz esta obligada a sobrescribir todos los métodos. Ya que como se ha dicho, una interfaz solo los define pero no los implementa.

NOTA: Si una clase abstracta únicamente tiene métodos abstractos quiere decir que se está usando con la funcionalidad de una interfaz.

Y a grandes rasgos se podría concluir diciendo que las clases abstractas se utilizan para compartir funciones. Mientras que las interfaces se utilizan para compartir como se tiene que hacer algo y que tiene que tener como mínimo.

26 - Traits

Los **traits** son un **mecanismo de reutilización de código** en lenguajes que tienen **herencia simple**, como **PHP**. El objetivo de los traits es reducir las limitaciones de la herencia simple permitiendo reutilizar métodos en varias clases independientes y de distintas jerarquías. Un trait es similar a una clase, pero su objetivo es agrupar funcionalidades específicas. Un trait, al igual que las clases abstractas, no se puede instanciar, simplemente facilita comportamientos a las clases sin necesidad de usar la herencia.

1. Ejemplo de situación con traits

Supongamos el siguiente ejemplo:

```
// Lector DB
class DbReader extends Mysqli{}
// Lector de archivos
class FileReader extends SplFileObject{}
```

Si por ejemplo queremos aplicar una misma funcionalidad en las dos clases, tendríamos un problema ya que ambas están ya heredando a otra clase. Queremos que las dos clases sean *singletons* (que sólo puedan instanciarse una vez). En lugar de tener que introducir esta funcionalidad en ambas, se puede **crear un trait**:

```
trait Singleton {
    private static $instance;
    public static function getInstance(){
        if(!(self::$instance instanceof self)){
            self::$instance = new self;
        }
        return self::$instance;
    }
}
      // Quitamos ahora las clases que extienden para hacer pruebas solo con los
      //traits
class DbReader {
      use Singleton;
}
class FileReader {
      use Singleton;
}
```

El trait **Singleton** se implementa ahora en ambas clases y podemos utilizar el método estático *getInstance()*, que crea un objeto de la clase que está usando el trait si no se ha creado ya, y lo devuelve.
Si creamos los objetos con *getInstance()*:

```
$a = DbReader::getInstance();
$b = FileReader::getInstance();
var_dump($a);
var_dump($b);
/*
Devuelve:
object(DbReader)[1]
object(FileReader)[2]*/
```

Vemos que $a es un objeto de **DbReader** y $b es un objeto de **FileReader**, y ambos se comportan como **singletons**. El método *getInstance()* se ha inyectado horizontalmente en ambas clases.
El **trait** no impone ningún comportamiento en la clase, simplemente es como si copiaras los métodos del trait y los pegaras en la clase.
Un trait no puede implementar **interfaces** ni extender clases normales ni abstractas.

2. Usar múltiples traits

Podemos **usar varios traits en la misma clase**:

```
trait Saludar {
    function decirHola(){
        return "hola";
    }
}
trait Despedir {
    function decirAdios(){
        return "adios";
    }
}
class Comunicacion {
    use Saludar, Despedir;
}
$comunicacion = new Comunicacion;
echo $comunicacion->decirHola() . ", que tal. " . $comunicacion->decirAdios();
```

También podemos **usar traits dentro de otros traits**:

```
trait SaludoYDespedida{
    use Saludar, Despedir;
}
class Comunicacion {
    use SaludoYDespedida;
}
$comunicacion = new Comunicacion;
echo $comunicacion->decirHola() ."", que tal. ".$comunicacion->decirAdios();
```

3. Orden de precedencia entre traits y clases

Utilizar traits dentro de otros traits es frecuente cuando la aplicación crece y hay numerosos traits que pueden agruparse para tener que incluir menos en una clase.
Eso hace que pueda haber métodos con el mismo nombre en diferentes traits o en la misma clase, por lo que tiene que haber un orden. Existe un **orden de precedencia** de los métodos disponibles en una **clase** respecto a los de los **traits**:

1. Métodos de un trait sobreescriben métodos heredados de una clase padre
2. Métodos definidos en la clase actual sobreescriben a los métodos de un trait

Vamos a verlo en un ejemplo con un trait y dos clases:

- Trait Comunicacion:

```
trait Comunicacion {
    function decirHola(){
        return "Hola";
    }
    function decirQueTal(){
        return "¿Qué tal? Soy un trait";
    }
    function decirHolaYQuetal(){
        return $this->decirHola() .  " " . $this->decirQueTal();
    }
    function preguntarEstado(){
        return $this->decirHola() . " " . parent::decirQueTal();
    }
    function decirBien(){
        return "Bien, desde el Trait Comunicación";
    }
}
```

- Clases Estado y Comunicar:

```
class Estado {
    function decirQueTal(){
        return "¿Qué tal? Soy Estado";
    }
    function decirBien(){
        return "Bien, desde la clase Estado";
    }
}
class Comunicar extends Estado {
    use Comunicacion;
    function decirQueTal() {
        return "¿Qué tal? Soy Comunicar";
    }
}
```

Creamos una instancia de comunicar y empleamos las funciones:

```
$a = new Comunicar;
echo $a->decirHolaYQuetal() . "<br>"; // Devuelve: Hola ¿Qué tal? Soy comunicar
echo $a->preguntarEstado() . "<br>"; // Devuelve: Hola ¿Qué tal? Soy Estado
echo $a->decirBien(); // Devuelve: Bien, desde el Trait Comunicación
```

Tenemos una clase **Comunicar** que extiende a **Estado**, y ambas clases tienen el método *decirQueTal()*, pero con diferentes implementaciones. También hemos incluído el trait Comunicación en la clase Comunicar.
Primero hemos llamado a dos métodos, *decirHolaYQueTal()* y *preguntarEstado()*. Ambos llaman a decirQueTal(), que existe en ambas clases además de en el trait. *decirHolaYQueTal()* llama al método *decirQueTal()* que está en la misma clase **Comunicar**. *preguntarEstado()* llama al método *decirQueTal()* que está en la clase padre **Estado**. Esto es así porque hemos referenciado con *parent::*.
Por último hemos llamado a *decirBien()*, con el que se puede comprobar que tiene precedencia el método del trait al método de la clase padre.

4. Conflictos entre métodos de traits

Cuando se usan **múltiples traits** es posible que haya **diferentes traits que usen los mismos nombres de métodos**. PHP devolverá un **error fatal**:

```
trait Juego {
    function play(){
        echo "Jugando a un juego";
    }
}
trait Musica {
    function play(){
        echo "Escuchando música";
    }
}
class Reproductor {
    use Juego, Musica;
}
$reproductor = new Reproductor;
$reproductor->play();
// Devuelve Fatal error: Trait method play has not been applied,
// because there are collisions with other trait methods on Reproductor
```

Esto no se resuelve de forma automática, hay que elegir el método que queremos usar dentro de la clase mediante la palabra **insteadof**:

```
trait Juego {
    function play(){
        echo "Jugando a un juego";
    }
}
trait Musica {
    function play(){
        echo "Escuchando música";
    }
}
class Reproductor {
    use Juego, Musica {
        Musica::play insteadof Juego;
    }
}
$reproductor = new Reproductor;
$reproductor->play(); // Devuelve: Escuchando música
```

Hemos elegido la función *play()* del trait Musica en lugar del trait Juego en la clase Reproductor.

En el ejemplo anterior se ha elegido un método sobre el otro respecto a dos traits. Hay ocasiones en las que puedes querer mantener los dos métodos, pero evitar conflictos. Se puede introducir un nuevo nombre para un método de un trait como alias. El alias no renombra el método, pero ofrece un nombre alternativo que puede usarse en la clase. Se emplea la palabra **as**:

```
class Reproductor {
    use Juego, Musica {
        Juego::play as playDeJuego;
        Musica::play insteadof Juego;
    }
}
$reproductor = new Reproductor;
$reproductor->play(); // Devuelve: Escuchando música
$reproductor->playDeJuego(); // Devuelve: Jugando a un juego
```

Ahora el método *playDeJuego()* equivaldrá a *Juego::play()*.

5. Usar Reflection con traits

Reflection es una característica de **PHP** que nos permite analizar la estructura interna de **interfaces**, **clases** y **métodos** y poder manipularlos. Existen cuatro métodos que podemos utilizar con Reflection para los traits:

- **ReflectionClass::getTraits()**. Devuelve un array con todos los traits disponibles en una clase
- **ReflectionClass::getTraitNames()**. Devuelve un array con los nombres de los traits en una clase.
- **ReflectionClass::isTrait()** comprueba si algo es un trait o no.
- **ReflectionClass::getTraitAliases()** devuelve un trait con los alias como *keys* y sus nombres originales como *values*.

6. Otras funcionalidades de los traits

Los traits permiten acceder a propiedades o métodos **private** o *protected*:

```
trait Mensaje {
    public function alerta(){
        echo $this->mensaje;
    }
}
class Mensajero {
    use Mensaje;
    private $mensaje = "Esto es un mensaje";
}
$mensajero = new Mensajero();
$mensajero->alerta(); // Devuelve: Esto es un mensaje
```

Los **traits** se inyectan en la clase de forma que es como si sus métodos formaran parte de ella, por lo que **cualquier método o propiedad private o protected podrá ser accedido desde un trait**.

También podemos tener **métodos abstractos dentro de traits para forzar a las clases a implementarlos**:

```
trait Mensaje {
    private $mensaje;

    public function alerta(){
        $this->definir();
        echo $this->mensaje;
    }
    abstract function definir();
}
class Mensajero {
    use Mensaje;
    function definir(){
        $this->mensaje = "Esto es un mensaje";
    }
}
$mensajero = new Mensajero();
$mensajero->alerta(); // Devuelve: Esto es un mensaje
```

Si no se define *definir()* en **Mensajero**, dará un **error faltal**.
Traits también pueden incluir un constructor *__construct()* que ha de ser declarado público, pero hay que tener cuidado con posibles colisiones.