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, menúes, carpetas de archivos pueden ser identificados como objetos; el motor de un auto 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.

pagina1.php

```
<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>
```

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 menúes, 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.

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.

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 (pagina1.php)

```
<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 graficar()
  {
    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->graficar();
?>
</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, graficar 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->graficar();
```

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 graficar y luego llamamos al método inicializar.

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 constructores 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 graficar()
  {
    echo '<div style="font-size:40px;text-align:'.$this-
>ubicacion.'">';
    echo $this->titulo;
    echo '</div>';
  }
}

$cabecera=new CabeceraPagina('El blog del programador','center');
$cabecera->graficar();
?>
</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 caracter 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->graficar();
```

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.

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->graficar();
```

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.

pagina1.php

```
<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;
  }
```

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

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

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

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

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

  public function graficar()
  {
    $this->inicioTabla();
    for($f=1;$f<=$this->cantFilas;$f++)
    {
      $this->inicioFila();
      for($c=1;$c<=$this->cantColumnas;$c++)
      {
        $this->mostrar($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->graficar();
?>
</body>
</html>
```

Vamos por parte, 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 graficar:

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

El método graficar 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 mostrar($fi,$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 graficar 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 graficar()
  {
    $this->inicioTabla();
    for($f=1;$f<=$this->cantFilas;$f++)
    {
      $this->inicioFila();
      for($c=1;$c<=$this->cantColumnas;$c++)
      {
        $this->mostrar($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->graficar();
```

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

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.

Analisemos 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->graficar();
```

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 mostrar($fi,$co)
  {
```

```
    echo '<td>'.$this->mat[$fi][$co].'</td>';
  }

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

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

  public function graficar()
  {
    $this->inicioTabla();
    for($f=1;$f<=$this->cantFilas;$f++)
    {
      $this->inicioFila();
      for($c=1;$c<=$this->cantColumnas;$c++)
      {
        $this->mostrar($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->graficar();
?>
</body>
</html>
```

Tenemos tres métodos públicos:

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

Y cinco métodos privados:

```
  private function inicioTabla()
  private function inicioFila()
  private function mostrar($fi,$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 planea una clase debe poner mucha atención cuales responsabilidades (métodos) deben ser públicas y cuales responsabilidades no queremos que las conozcan los demás.

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 a 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 graficar()
  {
    echo '<h1 style="text-align:center">'.$this->titulo.'</h1>';
  }
}

class Cuerpo {
  private $lineas=array();
  public function insertarParrafo($li)
  {
    $this->lineas[]=$li;
  }
  public function graficar()
  {
    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 graficar()
  {
    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 graficar()
  {
    $this->cabecera->graficar();
    $this->cuerpo->graficar();
    $this->pie->graficar();
  }
}

$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->graficar();
?>
</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 graficar()
  {
    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 graficar()
  {
    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 graficar()
  {
    for($f=0;$f<count($this->lineas);$f++)
    {
      echo '<p>'.$this->lineas[$f].'</p>';
    }
  }
}
```

Para graficar 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 graficar()
  {
    $this->cabecera->graficar();
    $this->cuerpo->graficar();
    $this->pie->graficar();
  }
}
```

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

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

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 graficar:

```
$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->graficar();
```

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

```
  }
}

class Menu {
  private $opciones=array();
  private $direccion;
  public function   construct($dir)
  {
    $this->direccion=$dir;
  }
  public function insertar($op)
  {
    $this->opciones[]=$op;
  }
  private function graficarHorizontal()
  {
    for($f=0;$f<count($this->opciones);$f++)
    {
      $this->opciones[$f]->graficar();
    }
  }
  private function graficarVertical()
  {
    for($f=0;$f<count($this->opciones);$f++)
    {
      $this->opciones[$f]->graficar();
      echo '<br>';
    }
  }
  public function graficar()
  {
    if (strtolower($this->direccion)=="horizontal")
      $this->graficarHorizontal();
    else
      if (strtolower($this->direccion)=="vertical")
        $this->graficarVertical();
  }
}

$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->graficar();
?>
</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 graficar muestra el enlace de acuerdo al contenido de los atributos inicializados previamente en el constructor:

```
  public function graficar()
  {
    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 graficarHorizontal()
  {
    for($f=0;$f<count($this->opciones);$f++)
    {
      $this->opciones[$f]->graficar();
    }
  }
  private function graficarVertical()
  {
    for($f=0;$f<count($this->opciones);$f++)
    {
```

```
      $this->opciones[$f]->graficar();
      echo '<br>';
    }
  }
```

Los algoritmos solo se diferencian en que el método graficarVertical añade el elemento
 después de cada opción.

Queda en la clase Menú el método público graficar que tiene por objetivo analizar el contenido del atributo $direccion y a partir de ello llamar al método privado que corresponda:

```
  public function graficar()
  {
    if (strtolower($this->direccion)=="horizontal")
      $this->graficarHorizontal();
    else
      if (strtolower($this->direccion)=="vertical")
        $this->graficarVertical();
  }
```

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->graficar();
```

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 graficar del objeto $menu1.

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.

pagina1.php

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

```
</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.

11 - Herencia.

Otra 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       PlanillaDeCalculo
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.

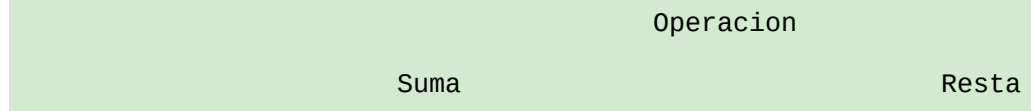
Si tenemos una ClaseA y otra ClaseB y notamos que entre ellas existe una relacion de tipo "... tiene un...", no debe implementarse herencia sino declarar en la clase ClaseA un atributo de la clase ClaseB.
Por ejemplo: tenemos una clase Auto, una clase Rueda y una clase Volante. Vemos que la relación 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" Vehiculo
Circulo "es una" Figura
Mouse "es un" DispositivoEntrada
Suma "es una" Operacion

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:



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 planteo 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.

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 difinimos 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
```

13 - Sobreescritura 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 sobreescribir métodos para completar el algoritmo del método de la clase padre. No es bueno sobreescribir un método y cambiar completamente su comportamiento.

Veamos nuestro problema de las tres clases: Operacion, Suma y Resta. Sobreescribiremos 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)

pagina1.php

```
<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 sobreescrito (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 mísmo 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 sobreescribir el método imprimirResultado)

14 - Sobreescritura 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 sobreescribimos el constructor de la clase padre.

Problema: 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.

pagina1.php

```
<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()
```

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.

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)

17 - Métodos y clases final.

Si a un método le agregamos el modificador final significa que ninguna subclase puede sobreescribirlo. 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.

pagina1.php

```
<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)
```

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 $persona1->retornarNombre().' - '.$persona1-
>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 $persona1->retornarNombre().' - '.$persona1-
>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)

19 - función __clone()

PHP nos permite crear un método que se llamará cuando ejecutemos el operador clone. Este método puede entre otras cosas inicializar algunos atributos.

Si no se define el método __clone se hará una copia idéntica del objeto que le pasamos como parámetro al operador clone.

Veamos un ejemplo: Crearemos una clase Persona que tenga como atributos su nombre y edad, definiremos los métodos para cargar y retornar los valores de sus atributos. Haremos que cuando clonemos un objeto de dicha clase la edad de la persona se fije con cero.

```
<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;
  }
  public function __clone()
  {
    $this->edad=0;
  }
}

$persona1=new Persona();
$persona1->fijarNombreEdad('Juan',20);
echo 'Datos de $persona1:';
echo $persona1->retornarNombre().' - '.$persona1-
>retornarEdad().'<br>';
$persona2=clone($persona1);
echo 'Datos de $persona2:';
echo $persona2->retornarNombre().' - '.$persona2-
>retornarEdad().'<br>';
?>
</body>
</html>
```

El método __clone se ejecutará cuando llamemos al operador clone para esta clase:

```
  public function __clone()
  {
    $this->edad=0;
  }
```

Es decir cuando realicemos la asignación:

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

inicialmente se hace una copia idéntica de $persona1 pero luego se ejecuta el método __clone con lo que el atributo $edad se modifica.

Si queremos que una clase no pueda clonarse simplemente podemos implementar el siguiente código en el método __clone():

```
  public function __clone()
  {
    die('No esta permitido clonar objetos de esta clase');
  }
```

20 - 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>';
```

21 - 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 graficar()
  {
    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->graficar();
?>
```

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 graficar genera la imagen dinámica propiamente dicha:

```
  public function graficar()
  {
    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->graficar();
```

22 - 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.