PHP es un lenguaje de programación estructurado con extensiones de **orientación a objetos** lo que le permite definir *clases* y construir *objetos*.

La implementación de **clases** no va a añadir ninguna funcionalidad nueva a las posibilidades de PHP. Su verdadera utilidad es la de *hacer la programación de otra manera*, con un código *más legible* y *reutilizable*.

La orientación a objetos se introdujo en la versión 4 de PHP y ha sido ampliada en versiones posteriores.

Las clases

Una clase *es* una especie de plantilla en la que se pueden definir **variables** (también llamadas **propiedades**) y **funciones** (llamados **métodos**) que pueden ser *invocadas* -en todo o en parte- desde cualquier otra parte del documento.

Para definir clases se usa la siguiente sintaxis

```
class nombre {
   definición de variables
   ....
   constructores (opcional)
   ....
    definición de funciones
   ....
}
```

- Dentro de la *clase* podemos **definir variables** que serán usadas por las **funciones internas** de esa clase y a las que se puede (no es obligatorio hacerlo) **asignar** valores.
- Las variables se definen como en PHP normal pero anteponiendo una de las siguientes palabras : **public** , **protected**, **private**

  (en PHP 4 se anteponia la palabra **var** que es equivalente a public).

  Sí en la definición se inicializa a un valor este debe ser un valor constante
- La definición de **funciones** dentro de las clases se hace como en PHP normal. Dentro de la función se puede usar la pseudo-variable **$this** que referencia, generalmente, al objeto creado
- En una **función,** para referenciar **variables** definidas **en la clase** se usa la notación: **$this->*variable*** (-> es el operador de objeto)

Ejemplo: Definición de la clase Matematica

```php
<?php
class Matematica {
   public $error='' ; // define variable de la clase y le asigna como valor la cadena vacía
   function suma($n1,$n2) {
        if ( is_numeric($n1) && is_numeric($n2) ) { $this->error = '' ; return $n1 + $n2; }
          else { $this->error = 'Uno de los valores no es numerico' ;  return 'NaN' ; }
      }
   function resta($n1,$n2) {
        if ( is_numeric($n1) && is_numeric($n2) ) { $this->error = '' ; return $n1 - $n2; }
          else { $this->error = 'Uno de los valores no es numerico' ;  return 'NaN' ; }
      }
}
?>
```

Crear y utilizar objetos

Las clases son solo plantillas y sus funciones no se ejecutan hasta que no se les ordene
Para poder utilizar funciones de una clase primero debemos crear un objeto (*instanciar la clase*) y luego ejecutar sobre ese objeto las funciones que deseemos

Para crear un objeto de una clase determinada usamos la sintaxis:

```
$name_objeto = new name_clase
```

Una vez creado el objeto, se puede aplicar en él (invocar) funciones definidas en su clase usando la siguiente sintaxis:

```
$name_objeto -> name_funcion()
```

```php
<?php
$calculadora = new Matematica ;  // crea objeto $calculadora de la clase Matematica
echo $calculadora->suma(3,2) ;  // Muestra resultado de hacer una suma
echo $calculadora->error ;      //  y contenido de la propiedad error
?>
```

Sí hacemos varias llamadas a una función utilizando el mismo objeto los resultados de la última llamada sobreescriben los de la llamada anterior.

Para conservar los diferentes resultados obtenidos tenemos 2 posibilidades:

- Crear diferentes objetos
- Usar arrays

Ejemplo: Creación de objetos y aplicación de funciones en ellos

Diferentes objetos

```
<?php
class Miclase {
public $resultado ;
function producto($a,$b) {
    $this ->resultado = $a * $b ;
  }
function imprime() {
    echo $this ->resultado , '<br>';
  }
 }
// Crea objetos con plantilla Miclase
$objeto  = new Miclase;
$objeto1 = new Miclase;

// Aplica función sobre objetos creado
$objeto -> producto(3,4);
$objeto -> producto(10,10);

$objeto1 -> producto(20,4);

$objeto -> imprime();
$objeto1 -> imprime();
?>
```

Muestra :
100
80

Un objeto con un array

```
<?php
class Miclase {
public $resultado , $indice = 0 ;
function producto($a,$b) {
      $this ->resultado [$this ->indice] = $a * $b ;
      $this ->indice ++ ;
    }
function imprime() {
  foreach($this->resultado as $valor) {
     echo $valor , '<br>'; }
  }
}
// Crea objeto con plantilla Miclase
$objeto  = new Miclase;

// Aplica función sobre objeto creado
$objeto -> producto(3,4);
$objeto -> producto(10,10);
$objeto -> producto(20,4);

$objeto -> imprime();
?>
```

Muestra :
12
100
80

Multiplicidad

La multiplicidad es la capacidad de usar múltiples instancias de una misma clase, con lo que se puede simplificar el código de una aplicación PHP y mantener el estado de varios objetos al mismo tiempo.

El ejemplo muestra una aplicación para hacer traspaso de dinero entre dos cuentas bancarias.
Para ello se consulta y actualiza el saldo de la cuenta origen (cuenta A, reduce su saldo) y la cuenta destino (cuenta B, incrementa su saldo)

```php
<?php
class Cuenta {
 public $n_cta , $saldo;
 function elige_cta($n) { $this->n_cta = $n; }
 function dame_saldo() { if (empty($this->saldo)) $this->saldo = 100; return $this->saldo; }
 function entrar_saldo($valor) { $this->saldo=$valor; }
 }
class Matematica {
 public $error='' ;
 function suma($n1,$n2) {
   if ( is_numeric($n1) && is_numeric($n2) ) { $this->error = '' ;return $n1 + $n2;}
    else { $this->error = 'Uno de los valores no es numerico' ; return 'NaN' ; }
  }
 function resta($n1,$n2) {
   if ( is_numeric($n1) && is_numeric($n2) ) { $this->error = '' ;return $n1 - $n2;}
    else { $this->error = 'Uno de los valores no es numerico' ; return 'NaN' ; }
  }
 function dame_error() { return $this->error; }
}

function traspaso_saldo( $origen , $destino , $importe) {
  $calc = new Matematica;
  if ( $calc->resta($origen->dame_saldo() , $importe ) < 0 ) {
    $error = $calc->dame_error();
     if ($error<>'') return 'El importe no es número';
       else return ' Sin saldo suficiente en cuenta origen';
    }
 else {
   $v1 = $calc->resta($origen->dame_saldo() , $importe );  $origen->entrar_saldo($v1);
   $v1 = $calc->suma($destino->dame_saldo() , $importe);  $destino->entrar_saldo($v1);
   Return 'OK';
   }
}
// ---------------------------- Proceso principal de la aplicación ----------------------------
$cuenta_A = new Cuenta;  $cuenta_B = new Cuenta;  // Crea 2 objetos de la clase Cuenta
$cuenta_A->elige_cta( '1234' ); $cuenta_B->elige_cta( '9876' ); // establece nºs cuentas en las que actuar
$transfiere = 50;  //  y el importe a transferir

echo 'Saldos antes del traspaso: <br>';  // muestra saldos antes de transferencia
echo 'cta origen: ' . $cuenta_A->dame_saldo().' cta destino: ' . $cuenta_B->dame_saldo().' <br>';

$ok = traspaso_saldo( $cuenta_A , $cuenta_B, $transfiere);  // realiza la transferencia
If ( $ok=='OK') {
  echo 'Saldos después del traspaso: <br>';  // Sí transferencia efectuada muestra saldos resultantes
  echo 'cta origen: ' . $cuenta_A->dame_saldo().' cta destino: ' . $cuenta_B->dame_saldo().' <br>';
 }
?>
```

Herencia (Clases extendidas)

A veces , el código de una clase no puede reutilizarse tal cual, sino que debe refinarse para incluir nuevas necesidades. Se podría crear una nueva clase, reescribiendo todo el código aprovechable de la clase existente e incluir las nuevas utilidades implementando las funciones necesarias. Esta solución limitaría la posible reutilización de código.

Una de las ventajas de la programación orientada a objetos es el uso de la **herencia** que permite crear una nueva clase que **"herede"** las características (variables y métodos) de otra clase.

PHP puede crear **clases extendidas** que tienen la *virtud* de poder disponer **tanto de las variables y funciones propias** como de **todas las variables y funciones** de la **clase base**.

Para crear una clase extendida se usa la siguiente sintaxis

```
class c_name extends c_base {
   definición de variables
   ....
   constructores (opcional)
   ....
    definición de funciones
   ....
}
```

- C_name es el nombre de la clase extendida y c_base el nombre de la clase padre (de la que hereda)
- PHP no admite crear una clase extendida de otra clase extendida, es decir no admite herencia multiple

En el ejemplo anterior, la clase Cuenta tiene la limitación de ser monodivisa, por lo que para hacer transferencias correctas las cuentas origen y destino deben compartir el mismo tipo de moneda.

Haciendo una **clase CuentaMultidivisa** que **extienda la clase Cuenta**, se puede añadir funcionalidades que haga el cambio entre divisas consultando el estado actual del cambio.

```
<?php
 class Cuenta {
  public $n_cta, $saldo;  function elige_cta($n) { $this->n_cta = $n; }
                          function entrar_saldo($valor) { $this->saldo=$valor; }
                          function dame_saldo() {if (empty($this->saldo)) $this->saldo = 100; return $this->saldo;}
    }

 class CuentaMultidivisa extends Cuenta {
  public $divisa ;
  function elige_cta( $n , $divisa='peseta') { $this->n_cta = $n;  $this->divisa = $divisa;}
  function dame_divisa()  { return  $this->divisa; }
  function dame_cambio( $div_o ,  $div_d)  {
   $res= 0;  $cambios = array();  $cambios['euro']    = array('euro'=>1 , 'peseta'=>166.36 ,  'dolar'=>1.4 );
                                  $cambios['peseta'] = array('euro'=>1/166.36 , 'peseta'=>1 ,  'dolar'=>1.4/166.36);
                                  $cambios['dolar']   = array('euro'=>1/1.4 , 'peseta'=>166.36/1.4 ,  'dolar'=>1) ;
   foreach ( $cambios as  $k => $moneda) {
       if ($k == $div_o)  {  foreach  ($moneda  as  $h=>$v)  { if ($h == $div_d)  $res=$v;}  }    }
   return $res;
  }
 function  aplicar_cambio($a_divisa) {
     $coef = $this->dame_cambio( $this->divisa, $a_divisa);   return  $this->saldo * $coef ;
   }
 function  cambiar_divisa($new_divisa){
    if ( $this->saldo == 0  || empty($this->divisa) )   $this->divisa  = $new_divisa;
    else  {  $this->saldo  =  $this->aplicar_cambio($new_divisa);    $this->divisa = $new_divisa; }
  }
}
// Crea  la  cuenta  1234  con saldo de 5000 pesetas  y muestra sus datos
$cuenta_A = new CuentaMultidivisa;      $cuenta_A->elige_cta('1234');     $cuenta_A->entrar_saldo(5000);
echo 'saldo cta  '. $cuenta_A->n_cta .'  = ' . $cuenta_A->obtener_saldo() .'  ' .$cuenta_A->dame_divisa() . '<br>';

$cuenta_A->cambiar_divisa('euro');     //  Cambia  a euros  la divisa de la cuenta  y muestra sus datos
echo 'saldo cta  '. $cuenta_A->n_cta .'  = ' . $cuenta_A->obtener_saldo() .'  ' .$cuenta_A->dame_divisa() . '<br>';
?>
```

Visibilidad

Las propiedades y métodos de las clases tienen *tres niveles de visibilidad*, clasificados en **Public** , **Protected** y **Private**, de más a menos permisivo respectivamente.

La visibilidad de una propiedad o método se define anteponiendo una de las palabras claves *public*, *protected* o *private* en la declaración.

Miembro de clase declarado	**Public**	**Protected**	**Private**
Accesible desde	cualquier lado	la misma clase las clases que hereden de ella la clase padre	la clase que lo define

Los métodos declarados sin ninguna palabra clave de visibilidad explícita son public.

Ejemplo: Visibilidad de propiedades

```php
<?php
class clase1
{  public $a = 'Public';     protected $b = 'Protected';     private $c = 'Private';
   function mostrar()   {  echo $this->a;       echo $this->b;       echo $this->c;    }
}
        $obj = new clase1();
        echo $obj->a;    // Funciona bien
        echo $obj->b; // Error Fatal
        echo $obj->c;   // Error Fatal
        $obj->mostrar();   // Muestra Public, Protected y Private

class clase2 extends clase1   // Se puede redeclarar propiedades y métodos public y protected, pero no private
{   protected $b = 'Protected2';
    function mostrar()   {   echo $this->a;       echo $this->b;       echo $this->c;    }
}
        $obj2 = new clase2();
        echo $obj2->a;    // Funciona bien
        echo $obj2->b; // Error Fatal
        echo $obj2->c;   // Undefined
        $obj2->mostrar();   // Muestra Public, Protected2, Undefined
?>
```

Ejemplo: Visibilidad de métodos

```php
<?php
class clase1
{ public function __construct() { } ;    // Declaración de un constructor public
  public function MyPublic() { } ;        // Declaración de un método public
  protected function MyProtected() { } ;// Declaración de un método protected
  private function MyPrivate() { }  ;      // Declaración de un método private

  function XX()  // Esto es public
    { $this->MyPublic();    $this->MyProtected();     $this->MyPrivate();    }
}
           $obj = new clase1;
           $obj->MyPublic(); // Funciona
           $obj->MyProtected(); // Error Fatal
           $obj->MyPrivate(); // Error Fatal
           $obj->XX(); // Public, Protected y Private funcionan

class clase2 extends clase1
{  function ZZ() // Esto es public
    {   $this->MyPublic();        $this->MyProtected();
        $this->MyPrivate(); // Error Fatal
    }
}
         $obj2 = new clase2;
         $obj2->MyPublic(); // Funciona
         $obj2->ZZ();          // Public y Protected funcionan, pero Private no
?>
```

Hay casos en que es beneficioso acceder a las variables internas de un objeto; En otros casos puede ser una fuente de errores.

```
<?php
 class Cuenta {
   public $n_cta, $saldo; function elige_cta($n) { $this->n_cta = $n; }
                          function entrar_saldo($valor) { $this->saldo=$valor; }
                          function dame_saldo() {if (empty($this->saldo)) $this->saldo = 100; return $this->saldo;}
 }

class CuentaMultidivisa extends Cuenta {
 public $divisa ;
  function elige_cta( $n , $divisa='peseta') { $this->n_cta = $n; $this->divisa = $divisa;}
  function dame_divisa() { return $this->divisa; }
  function dame_cambio( $div_o , $div_d) {
    $res= 0; $cambios = array(); $cambios['euro'] = array('euro'=>1 , 'peseta'=>166.36 , 'dolar'=>1.4 );
                                 $cambios['peseta'] = array('euro'=>1/166.36 , 'peseta'=>1 , 'dolar'=>1.4/166.36);
                                 $cambios['dolar'] = array('euro'=>1/1.4 , 'peseta'=>166.36/1.4 , 'dolar'=>1) ;
    foreach ( $cambios as $k => $moneda) {
        if ($k == $div_o) { foreach ($moneda as $h=>$v) { if ($h == $div_d) $res=$v;} } }
    return $res;
   }
 function aplicar_cambio($a_divisa) {
     $coef = $this->dame_cambio( $this->divisa, $a_divisa); return $this->saldo * $coef ;
   }
 function cambiar_divisa($new_divisa){
    if ( $this->saldo == 0 || empty($this->divisa) ) $this->divisa = $new_divisa;
    else { $this->saldo = $this->aplicar_cambio($new_divisa); $this->divisa = $new_divisa; }
  }
}
// Crea la cuenta 1234 con saldo de 5000 pesetas y muestra sus datos
$cuenta_A = new CuentaMultidivisa;    $cuenta_A->elige_cta('1234'); $cuenta_A->entrar_saldo(5000);

$cuenta_A->divisa = 'euro'; ————————
?>
```

Como se puede acceder a la propiedad divisa del objeto cuenta_A, ha pasado de tener 5.000 pesetas a tener 5.000 euros.

Para evitar estos errores o malas actuaciones la propiedad divisa de la clase CuentaMultidivisa debería ser declarada privada:

```
class CuentaMultidivisa extends Cuenta {
 private $divisa ;

 function elige_cta( $n , $divisa='peseta') { $this->n_cta = $n; $this->divisa = $divisa;}
 function dame_divisa() { return $this->divisa; }
 .....

}
// Crea la cuenta 1234 con saldo de 5000 pesetas y muestra sus datos
$cuenta_A = new CuentaMultidivisa;    $cuenta_A->elige_cta('1234'); $cuenta_A->entrar_saldo(5000);

$cuenta_A->cambiar_divisa('euro');  // Ajusta el saldo al que le corresponde en la nueva divisa
```

Constructores

Cuando en una clase se define **una función** con un nombre **idéntico** al **nombre de la clase**, a esa función se le llama **un constructor**.

Esa función -el **constructor**- tiene la peculiaridad de que se **ejecuta** de forma **automática** en el momento en que se define un **nuevo objeto** de la clase

Dependiendo como esté **definido** el **constructor** puede ejecutarse de distintas formas:

Utilizando los valores predefinidos en las variables de la **clase**.

Asignado los valores preestablecidos a los parámetros de la función **constructor**

Ejemplo: Uso de constructores para crear objetos

Usa valores predefinidos en variables de la clase

```
<?php

class Miclase {

public $resultado;

public $n1=5 , $n2 = 8;

function Miclase() {
    $this ->resultado = $this->n1 * $this->n2 ;
    echo $this ->resultado , '<br>'; }
  }

// Crea objetos con plantilla Miclase
$objeto  = new Miclase;
$objeto1 = new Miclase;

?>
```

Muestra : 40
40

Usa valores de los parámetros de la función constructor

```
<?php
class Miclase {

  public $resultado;

  function Miclase($a=3, $b=7) {
        $this ->resultado = $a * $b ;
        echo $this ->resultado , '<br>';
        }
 }
// Crea objetos con plantilla Miclase
$objeto  = new Miclase;
$objeto1 = new Miclase(2,8);
$objeto2 = new Miclase(10);

// Aplica función sobre objeto creado
$objeto -> Miclase(10,10);
?>
```

Muestra : 21
16
70
100

Cuando se le **pasan** valores la función se ejecuta **olvidándose** de los valores asignados por defecto en la función y cuando se le *pasan* solo **parte de los valores** utiliza los **valores recibidos** para los **asignados en la llamada** y para el resto los valores por defecto dados en el **constructor**.

Operador de resolución de ámbito (::)

PHP permite llamar a un miembro de una **clase** sin necesidad de **crear** previamente un objeto.

La sintaxis es : ***clase* :: *funcion()***

```
<?php
  class claseA {
    const C1 = 'Un valor constante';
  }

$v1 = 'claseA';
echo $v1::C1;      // A partir de PHP 5.3.0

echo claseA::C1;

  class claseB    {
   protected function F1() {
       echo "F1 en claseB";
    }
  }

 class claseC extends claseB  {
  public function F1()   // Sobrescritura de definición parent
  {
     parent::F1();  // Pero todavía se puede llamar a la función parent
     echo "F1 en claseC";
  }
}

$x= new claseC();
$x->F1();
?>
```