

# UT3 PROGRAMACIÓN BASADA EN LENGUAJES DE MARCAS CON CÓDIGO EMBEBIDO

---

## Índice

- [UT3 PROGRAMACIÓN BASADA EN LENGUAJES DE MARCAS CON CÓDIGO EMBEBIDO](#)
  - [Índice](#)
  - [Estructuras de control](#)
    - [Sentencias condicionales](#)
    - [Sentencias repetitivas o bucles](#)
  - [Funciones](#)
    - [Creación y ejecución de funciones](#)
    - [Variable globales y locales](#)
    - [Argumentos en las funciones](#)
    - [Inclusión de las funciones en ficheros externos](#)
  - [Trabajar con cadenas de caracteres](#)
    - [funciones de cadenas](#)
  - [Tipos de datos compuestos](#)
    - [Arrays](#)
      - [Recorrer un array](#)
      - [Funciones para trabajar con arrays](#)
    - [Variables especiales en PHP](#)
    - [Objetos](#)
      - [Creación de clases en PHP](#)
      - [Constructores](#)
      - [Constructor property promotion](#)
        - [creando instancias desde un constructor](#)
      - [Métodos get y set](#)
      - [Operador this](#)
      - [Constantes](#)
      - [Métodos estáticos](#)
      - [Utilización de objetos](#)
      - [Herencia](#)
      - [Interfaces](#)
      - [Traits](#)
        - [Orden de precedencia entre traits y clases](#)
        - [conflictos entre métodos de traits](#)
        - [Usar Reflection con traits](#)
    - [Namespaces](#)
      - [la constante NAMESPACE](#)
      - [La palabra reservada namespace](#)
      - [Importar un namespace](#)
      - [Utilizar alias en los namespace](#)
    - [Enumeraciones](#)

- Métodos dentro de un enumerado
- Funciones relacionadas con los tipos de datos
- Formularios
  - Métodos GET y POST
    - Método GET
    - Método POST
  - Recuperación de información
    - Con GET
    - Con POST
  - Validación de datos

# Estructuras de control

son de dos tipos:

- Condicionales
- Repetitivas o bucles

## Sentencias condicionales

- **if /elseif /else:** definir una expresión para ejecutar o no la sentencia o conjunto de sentencias siguientes

```
<?php
$a=1;
$b=3;
if ($a<$b)
    print "a es menor que b";
elseif ($a>$b)
    print "a es mayor que b";
else
    print "a es igual a b";
?>
```

no es necesario poner {} porque solo ejecuta una sentencia en cada caso

- **Operador ternario:** otra forma de definir un if

```
<?php
$a=1;
$b=3;
$imprimir=($a<$b)? "a es menor que b":(($a>$b)? "a es mayor que b": "a es igual a b");
print $imprimir;
?>
```

- **switch:** similar a enlazar varias sentencias if comparando una misma variable con diferentes valores

```
<?php
$a=1;
switch ($a){
    case 0:
        print " a vale 0";
        break;
    case 1:
        print " a vale 1";
        break;
    default:
```

```

        print "a no vale 0 ni 1";
    }
?>

```

- **match:** es similar a switch pero más sencilla

```

<?php
    $a=1;
    match ($a){
        0=>print " a vale 0",
        1=>print " a vale 1",
        default=> print "a no vale 0 ni 1"
    }
?>

```

## Sentencias repetitivas o bucles

- **while::** define un bucle que se ejecuta mientras se cumpla una expresión. La expresión se evalúa antes de comenzar cada ejecución del bucle.

```

<?php
    $a=1;
    while ($a <8)
        $a+=3;
    print $a // el valor obtenido es 10
?>

```

no es necesario poner {} porque solo ejecuta una sentencia

- **do ... while:** similar al bucle while, pero la expresión se evalúa al final, con lo cual se asegura que la sentencia/as del bucle se ejecutan al menos una vez.

```

<?php
    $a=1;
    do
        $a -=3;
    while ($a >10)
    print $a; // el valor obtenido es -2
?>

```

no es necesario poner {} porque solo ejecuta una sentencia

- **for:** compuesto por tres expresiones:  
sintaxis

```
for(expr1;expr2;expr3){
    sentencia o conjunto de sentencias
}
```

1. expr1: se ejecuta una vez al comienzo del bucle.
2. expr2: se evalúa para saber si se debe ejecutar o no la/as sentencia/as
3. expr3: se ejecuta tras ejecutar todas las sentencias del bucle


```
<?php
    for ($a=5;$a<10;$a+=3){
        print $a // se muestran los valores 5 y 8
        print "<br/>";
    }
?>
```

- **foreach:** diseñada para recorrer todos los valores de un array

```
<?php
    $nombres=
    ['Samuel','Manuel','Luis','Lucía','Victor','Efren','Cesar','Eloy','David'];
    foreach ($nombres as $nombre){
        print $nombre."<br/>";
    }

    // si quiero saber el índice
    foreach ($nombres as $indice=>$nombre){
        print $indice ." ".$nombre."<br/>";
    }

?>
```

 Hoja03\_PHP\_01

## Funciones

- Permiten **asociar una etiqueta** (el nombre de la función) con un bloque de código a ejecutar.
- Al usar funciones estamos ayudando a **estructurar mejor el código**.
- Las funciones permiten crear variables locales que no serán visibles fuera del cuerpo de las mismas.

### Creación y ejecución de funciones

Para crear tus propias funciones se usa la palabra **function**

```
<?php
function precio_con_iva(){
    global $precio;
    $precio_iva=$precio*1.21;
    print "el precio con IVA es ".$precio_iva;
}
$precio=10;
precio_con_iva();
?>
```

No es necesario definir las funciones antes de usarlas excepto cuando están definidas condicionalmente:

```
<?php
$iva=true;
$precio =10;
precio_con_iva();// Da error, pues aquí aún no está definida
if ($iva){
    function precio_con_iva(){
        global $precio;
        $precio_iva=$precio*1.21;
        print "el precio con IVA es ".$precio_iva;
    }
}
precio_con_iva(); // aquí ya no da error
?>
```

## Variable globales y locales

- PHP Permite definir una variable con el mismo nombre dentro y fuera de una función las consideras dos variables distintas.

```
<?php
$a=5;
function prueba() {
    $a=3;
    echo "variable local de la función ".$a."<br />";
}
echo prueba();
echo " variable global del programa ".$a."<br />";
?>
```

- Permite utilizar una variable global en una función, al definir la variable dentro de la función es **global** nombre de la variable

```

<?php
    $a=5;
    function prueba() {
        global $a;
        echo "Utilizo la variable global en la función ".$a."<br />";
    }
    echo prueba();
    echo " variable global del programa ".$a."<br />";

?>

```

## Argumentos en las funciones

- Siempre es mejor utilizar **argumentos o parámetros al hacer la llamada**. Además, en lugar de mostrar el resultado en pantalla o guardar el resultado en una variable global, las funciones pueden **devolver un valor usando la sentencia return**.
- Cuando en una función se encuentra una sentencia return, termina su procesamiento y devuelve el valor que se indica

```

<?php
    function precio_con_iva($precio){
        return $precio*1.21;
    }
    $precio=10;
    $precio_iva=precio_con_iva($precio);
    print "el precio con IVA es ".$precio_iva;

?>

```

- Al definir la función, puedes indicar **valores por defecto** para los argumentos, de forma que cuando hagas una llamada a la función puedes no indicar el valor de un argumento; en este caso se toma el valor por defecto indicado.

```

<?php
    function precio_con_iva($precio,$iva=0.21){
        return $precio*(1+$iva);
    }
    $precio=10;
    $precio_iva=precio_con_iva($precio);
    print "el precio con IVA es ".$precio_iva."<br />";

    // cambiar el valor del iva y no utilizar el defecto
    $iva=0.04;
    $precio_iva=precio_con_iva($precio,$iva);
    print "el precio con IVA del $iva es ".$precio_iva."<br />";

?>

```

- Permite pasar un número indeterminado de parámetros con **tres puntos por delante del nombre de variable**, lo trata como un array

```
<?php
function concatenar (...$palabras){
    $resultado = "";
    foreach($palabras as $palabra){

        $resultado .= $palabra." ";

    }
    echo $resultado;
}
concatenar('curso','DAW2','módulo','DWES');

?>
```

- Los argumentos anteriores se pasaban **por valor**. Esto es, cualquier cambio que se haga dentro de la función a los valores de los argumento no se reflejará fuera de la función.
- Si quieres que esto ocurra debes definir el parámetro para que su valor se pase **por referencia**, añadiendo el símbolo **&** antes de su nombre.

```
<?php
function precio_con_iva(&$precio,$iva=0.21){
    return $precio*(1+$iva);
}
$precio=10;
precio_con_iva($precio);
print "el precio con IVA es ".$precio."<br/>";

?>
```

- **tipo de variable definido en los parámetros**

```
<?php
function sumaEnteros(int $entero1, int $entero2){
    return $entero1+ $entero2;
}
$resultado=sumaEnteros(2,5);
echo $resultado;

//también permite introducir un dato con decimal, lo trunca y no da error
$resultado=sumaEnteros(2,5.6);
echo $resultado;

?>
```



- Para no permitir introducir valores que no correspondan con el **tipo de variable**, declarar que el tipo es restrictivo dando error si no hay correspondencia entre el tipo y el valor.

```
<?php
declare( strict_types=1);
function sumaEnteros(int $entero1, int $entero2){
    return $entero1+ $entero2;
}
$resultado=sumaEnteros(2,5);
echo $resultado;

//también permite introducir un dato con decimal,ahora da error
$resultado=sumaEnteros(2,5.6);
echo $resultado;

?>
```

- También se puede especificar el tipo de dato que la función tiene que devolver, poniendo **dos puntos**
- Los tipos de datos que podemos especificar son int, float, string, bool, array, object, null

```
<?php
declare( strict_types=1);
function mediaEnteros(int $entero1, int $entero2):int|float{
    return ($entero1+ $entero2)/2;
}
$resultado=mediaEnteros(2,5);
echo $resultado;

?>
```

## Inclusión de las funciones en ficheros externos

En ocasiones resulta más cómodo agrupar las funciones en ficheros externos al que se hará referencia.

Formas de incorporar ficheros externos:

- **include**: evalúa el contenido del fichero que se indica y lo incluye como parte del fichero actual en el punto de realización de la llamada. Se puede indicar la ruta de forma absoluta o de forma relativa. Se toma como base la ruta que se especifica en la directiva include\_path del fichero php.ini, si no se encuentra en esa ubicación, se buscará en el directorio actual
- **include\_once**: si por equivocación incluyes más de una vez un mismo fichero, lo normal es que obtengas algún tipo de error (por ejemplo, al repetir una definición de una función)
- **require**: si el fichero que queremos incluir no se encuentra, include da un aviso y continua la ejecución del guión. La diferencia más importante al usar require es que en ese caso, cuando no se puede incluir el fichero, se detiene la ejecución del guión.
- **require\_once**: es la combinación de las dos anteriores.

## Ejemplo

```
<?php
    include 'funciones.inc.php';
    print fecha();
?>
```

## Trabajar con cadenas de caracteres

- Una cadena es una concatenación de caracteres
- Ojo con la función **strlen** cuenta el número de bits no el número de caracteres, la función del número de caracteres es **mb\_strlen**

```
<?php
    $cadena='aeiou';
    echo strlen($cadena)." "; // devuelve 5
    echo mb_strlen($cadena); //devuelve 5
    $cadena1='áeiou';
    echo strlen($cadena1)." "; // devuelve 6
    echo mb_strlen($cadena1); //devuelve 5

?>
```

## funciones de cadenas

- **strpos( exp1, exp2)**, busca la primera coincidencia de izquierda a derecha y devuelve la posición
  - exp1 - cadena donde buscar
  - exp2 - lo que queremos buscar
- **strrpos( exp1,exp2)**, cambiamos el sentido de búsqueda de derecha a izquierda
- **str\_starts\_with(exp1,exp2)**, si comienza con la cadena que queremos buscar
- **str\_ends\_with(exp1,exp2)**, si termina con la cadena a buscar
- **strcmp(exp1, exp2)**, compara dos cadenas:
- si son iguales devuelve 0, si exp1 > exp2 devuelve un 1, si exp1 < exp2 devuelve un -1
  - exp1 - primera cadena a comparar
  - exp2 - segunda cadena a comparar
- **substr(exp1,exp2, exp3)**, obtiene una subcadena
  - exp1 - cadena donde extraer subcadena
  - exp2 - posición desde donde comienza a extraer la subcadena . Si ponemos un número negativo comienza desde el final de la cadena y extrae el número de caracteres que indica exp2 si no hay exp3. Si ponemos un número positivo extrae la cadena a partir de la posición indicada teniendo en cuenta que el cero es la primera posición si no hay exp3.
  - exp3 - indica el número de caracteres a extraer
- **str\_replace(exp1,exp2,exp3)**, reemplazar una cadena
  - exp1 - cadena que se quiere reemplazar en la exp3
  - exp2 - cadena por la que se quiere reemplazar

- exp3 - cadena donde se quiere operar
- **strtolower(exp1)**, convertir toda la cadena exp1 a minúsculas
- **strtoupper(exp1)**, convertir toda la cadena exp1 a mayúsculas
- **ucfirst(exp1)**, convertir la primera letra de la cadena exp1 a mayúsculas
- **ucwords(exp1)**, convertir la primera letra de cada palabra de la cadena exp1 a mayúsculas

```
<?php
$cadena='aeiou';
echo strpos($cadena,'i')." "; // devuelve 2
echo strrpos($cadena,'i'); //devuelve 5

echo str_starts_with($cadena,'e')? " comienza ":" no comienza ";// devuelve no
comienza
echo str_ends_with($cadena,'i')? " finaliza ":" no finaliza ";//devuelve
finaliza
$cadena1='aeiou';

echo strcmp($cadena,$cadena1);// devuelve un 1


echo substr($cadena,-4);// extrae ioui
echo substr($cadena,2); // extrae ioui

echo substr($cadena,-4,2);// extrae io
echo substr($cadena,2,2); // extrae io

echo str_replace('a','b',$cadena);// devuelve beioui

$cadena='hola Mundo';

echo strtolower($cadena);// devuelve hola mundo
echo strtoupper($cadena);// devuelve HOLA MUNDO
echo ucfirst($cadena);// devuelve Hola Mundo
echo ucwords($cadena);//devuelve Hola Mundo
?>
```

 Hoja03\_PHP\_02

## Tipos de datos compuestos

Un tipo de datos compuesto es aquel que te permite almacenar más de un valor. En PHP puedes utilizar dos tipos de datos compuestos: el array y el objeto.

### Arrays

**Un array** es un tipo de datos que nos permite almacenar varios valores. Cada miembro del array se almacena en una posición a la que se hace referencia utilizando un valor clave. Las claves pueden ser numéricas o asociativas.

```
<?php
    //array indice numérico
    $modulos1=array(0=>"Programación",1=>"Base de datos",9=>"Desarrollo web en
entorno servidor");
    $modulos12=array("Programación","Base de datos","Desarrollo web en entorno
servidor");

    //array asociativo indice son las siglas
    $modulos2=array("PR"=>"Programación","BD"=>"Base de datos","DWES"=>"Desarrollo
web en entorno servidor");
?>
```

- la función **print\_r**, nos muestra todo el contenido del array que le pasamos.
- Para hacer referencia a los elementos almacenados en un array, hay que utilizar el valor clave entre corchetes:

```
// en los tres casos nos devuelve Desarrollo web en entorno servidor
echo $modulos1[9];
echo $modulos12[2];
echo $modulos2["DWES"];
```

- En PHP se pueden crear arrays de varias dimensiones almacenando otro array en cada uno de los elementos de un array.

```
<?php

$ciclos=array(
    "DAW"=>array("PR"=>"Programación","BD"=>"Base de datos","DWES"=>"Desarrollo
web en entorno servidor"),
    "DAM"=>array("PR"=>"Programación","BD"=>"Base de datos","AD"=>"Acceso a
datos")
);
?>
```

- Para hacer referencia a los elementos almacenados en un array multidimensional, hay que indicar las claves para cada una de las dimensiones:

```
// devuelve Desarrollo web en entorno servidor

echo $ciclos["DAW"]["DWES"];
```

- No hay que indicar el tamaño del array para crearlo. Ni siquiera es necesario indicar que una variable concreta es de tipo array:

```
$modulos1[0]="Programación";  
$modulos1[1]="Base de datos";  
$modulos1[9]="Desarrollo web en entorno servidor";
```

- Ni siquiera es necesario especificar el valor de la clave. Si se omite, el array se irá llenando a partir de la última clave numérica existente, o de la posición 0 si no existe ninguna:

```
$modulos12[]="Programación";  
$modulos12[]="Base de datos";  
$modulos12[]="Desarrollo web en entorno servidor";
```

## Recorrer un array

Las cadenas de texto o strings se pueden tratar como arrays en los que se almacena una letra en cada posición, siendo 0 el índice correspondiente a la primera letra, 1 el de la segunda, etc.

```
$modulo = "Desarrollo web en entorno servidor";  
// $modulo[3] == "a";
```

Para recorrer un array se puede utilizar **el bucle foreach**. Utiliza una variable temporal para asignarle en cada iteración el valor de cada uno de los elementos del arrays. Puedes usarlo de dos formas.

- Recorriendo sólo los elementos:

```
foreach ($modulos1 as $modulo)  
    echo $modulo . "<br/>";
```

- O recorriendo los elementos y sus valores clave de forma simultánea:

```
foreach ($modulos2 as $codigo => $modulo)  
    echo "El código del módulo ".$modulo." es ".$codigo."<br/>";
```

## Funciones para trabajar con arrays

- list(variable1,...,variablen)=nombrearray
  - variable1, ...variable n - son las variables donde se almacenan los valores del array
  - nombrearray, es el array que queremos extraer a las variables

```
<?php
$array=[1,2,3];
list($a,$b,$c)=$array;

echo $a." ".$b." ".$c;
?>
```

- range(valor1,valor2), rellenar un array con un intervalo de valores comprendido entre valor1 y valor 2

```
<?php
$array=range(10,20);
var_dump($array);
echo $array[5]; //devuelve el valor 15
?>
```

- count(nombrearray), nos cuenta el número de valores del array

```
<?php
$array=range(10,20);
echo count ($array); // devuelve 11 valores
?>
```

- in\_array( cadena a buscar, nombrearray) saber si un elemento se encuentra dentro de un array devuelve un true

```
<?php
$nombres=
['Samuel','Manuel','Luis','Lucía','Victor','Efren','Cesar','Eloy','David'];
$salida= in_array('Samuel',$nombres)? " se encuentra ese nombre dentro del
array": "No se encuentra ese nombre";
echo $salida;
?>
```

- unset(nombrearray[indice]), borrar un elemento dentro del array indicando el indice. Si ponemos solo el nombre del array borra todo el array

- Si sólo nos interesa saber si una variable está definida y no es null, puedes usar **la función isset**.  
**La función unset** destruye la variable o variables que se le pasa como parámetro.

## Variables especiales en PHP

- PHP incluye variables internas predefinidas que pueden usarse desde cualquier ámbito. Se denominan **superglobales**.
- Cada una de estas variables es un array que contiene un conjunto de valores. Son las siguientes:
  - **\$\_SERVER**: contiene información sobre el entorno del servidor web y de ejecución.
  - **\$\_GET, \$\_POST y \$\_COOKIE**: contienen las variables que se han pasado al guión actual utilizando respectivamente los métodos GET (parámetros en la URL), HTTP POST y Cookies HTTP.
  - **\$\_REQUEST**: junta en uno solo el contenido de los tres arrays anteriores.
  - **\$\_ENV**: contiene las variables que se puedan haber pasado a PHP desde el entorno en que se ejecuta.
  - **\$\_FILES**: contiene los ficheros que se puedan haber subido al servidor utilizando el método POST.
  - **\$\_SESSION**: contiene las variables de sesión disponibles para el guión actual.



## Objetos

El lenguaje PHP original no se diseñó con características de orientación a objetos

Las características de POO que se soportan a partir de la versión PHP 5 incluyen:

- Métodos estáticos
- Métodos constructores y destructores
- Herencia
- Interfaces
- Clases abstractas

No se incluye:

- Herencia múltiple (Java NO tiene)
- Sobrecarga de métodos (Java SÍ tiene)
- Sobrecarga de operadores (Java NO tiene)

### Creación de clases en PHP

La declaración de una clase en PHP se hace utilizando la palabra **class**. A continuación y entre llaves, deben figurar los miembros de la clase. Conviene hacerlo de forma ordenada, primero las propiedades o atributos, y después los métodos, cada uno con su código respectivo.

```
class Producto{  
    private $codigo;  
    public $nombre;  
    protected $PVP;  
    public function muestra(){  
        return "<p>".$this->codigo."</p>";  
    }  
}
```

En el ejemplo hemos puesto para recordar los principales niveles de acceso de los atributos, recordar que por lo general los atributos deben ser privados

Recomendaciones:

- Cada clase en un fichero distinto
- Los nombres de las clases deben comenzar por mayúsculas para distinguirlos de los objetos y otras variables

Una vez definida la clase, podemos usar la palabra **new** para instanciar objetos de la siguiente forma:

```
$p = new Producto();
```



Para que la línea anterior se ejecute sin error, previamente debemos haber declarado la clase. Para ello, en ese mismo fichero tendrás que incluir la clase poniendo algo como:

```
require_once('producto.php');
```

Para acceder desde un objeto a sus atributos o a los métodos de la clase, debes utilizar el **operador flecha** (sólo se pone el símbolo \$ delante del nombre del objeto)

```
$p->nombre = "Samsung Galaxy S 20";  
echo $p->muestra();
```

Ejercicio: Crea la clase Producto con los atributos que consideres necesarios.

## Constructores

Desde PHP 7 puedes definir en las clases métodos constructores, que se ejecutan cuando se crea el objeto. El constructor de una clase debe llamarse **construct**. Se pueden utilizar, por ejemplo, para asignar valores a atributos.

```
class Producto{  
  
    private $codigo;  
    public function __construct(){  
        $this->codigo=1;  
    }  
    ...  
}
```

El constructor de una clase puede llamar a otros métodos o tener parámetros, en cuyo caso deberán pasarse cuando se crea el objeto.

Sin embargo, **sólo puede haber un método constructor en cada clase**

```
class Producto{  
  
    private $codigo;  
    public function __construct($codigo){  
        $this->$codigo = $codigo;  
    }  
    ...  
}
```

Ejercicio: Crea un constructor para vuestra clase Producto.

También es posible definir un método destructor, que debe llamarse **\_\_destruct** y permite definir acciones que se ejecutarán cuando se elimine el objeto.

```
class Producto{

    private $codigo;
    public function __construct($codigo){
        $this->$codigo = $codigo;

    }
    public function __destruct(){
        $this->$codigo=0;
    }
    ...
}
$p = new Producto('Iphone');
```

### Constructor property promotion

Desde PHP 8 podemos definir una clase indicando el tipo de dato del atributo. También podemos definir un constructor donde los parámetros sean los atributos de la clase sin tener que definirlos.

```
class Persona{
    public string $nombre;
    public int $edad;

    public function __construct(string $nombre, int $edad){
        $this->nombre=$nombre;
        $this->edad=$edad;
    }

}

$persona1 = new Persona("Juan",25);

echo sprintf("%s tiene %d años", $persona1->nombre, $persona1->edad);
```

Los atributos se definen al pasar los parámetros al constructor

```
class Persona{

    public function __construct( public string $nombre, public int $edad)
    {

    }

}
```

```

    }

    $persona1 = new Persona("Juan",25);

    echo sprintf("%s tiene %d años", $persona1->nombre, $persona1->edad);

```

### creando instancias desde un constructor

Desde PHP 8, podemos crear un objeto dentro de un constructor

```

class HolaMundo {
    public function decirHola(string $nombre): string
    {
        return "Hola $nombre<br>";
    }
}

$holaMundo = new HolaMundo();
echo $holaMundo->decirHola("Juan");

class MiHolaMundo {
    public function __construct(
        private readonly HolaMundo $holaMundo = new HolaMundo(),
    ) {}

    public function hola(string $nombre): string {
        return $this->holaMundo->decirHola($nombre);
    }
}

$miHolaMundo = new MiHolaMundo();
echo $miHolaMundo->hola("Juan");

```

### Métodos get y set

Aunque no es obligatorio, el nombre del método que nos permite obtener el valor de un atributo suele empezar por get y el que nos permite modificarlo por set

```

private $codigo

public function setCodigo ($nuevo_codigo)
{
    $this->codigo =$nuevo_codigo;
}

```

```
public function getCodigo() {return $this->codigo;}
```

Ahora podemos definirlos indicando el tipo de dato que devuelve el get

```
class Persona{

    public function __construct( private string $nombre, private int $edad )
    {
    }

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

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

}

$persona1 = new Persona("Juan",25);

echo sprintf("%s tiene %d años" , $persona1->getNombre(),$persona1->getEdad());
```

Ejercicio: crea los métodos get y set para la clase Producto creada por ti.

Desde PHP 5 se introdujeron los llamados **métodos mágicos** entre ellos **\_\_set** y **\_\_get** Si se declaran estos dos métodos en una clase, PHP los invoca automáticamente cuando desde un objeto se intenta usar un atributo no existente o no accesible

Mas información sobre [los métodos mágicos](#)

Por ejemplo, el código siguiente simula que la clase Producto tiene cualquier atributo que queramos usar:

```
class Producto
{
    private $atributos =array();
    public function __get($atributo){
        return $this->atributos[$atributo];
    }
    public function __set($atributo,$valor){
        $this->atributos[$atributo] = $valor;
    }
}
```

El método **\_\_toString**, es otro método mágico

```
class Persona
{
    public function __construct(private string $nombre, private int $edad)
    {
    }

    public function __toString(): string
    {
        return sprintf("%s tiene %d años", $this->nombre, $this->edad);
    }
}
$persona1 = new Persona("Juan", 25);
echo $persona1;
```

Ejercicio: comenta los métodos get y set que has hecho y añade los mágicos

## Operador this

Cuando desde un objeto se invoca un **método de la clase** a éste se le pasa siempre una referencia al objeto que hizo la llamada. Esta referencia se almacena en la variable **\$this**.

Se utiliza, por ejemplo, en el código anterior para tener acceso a los **atributos privados del objeto** (que sólo son accesibles desde los métodos de la clase).

```
print "<p>". $this->codigo . "</p>";
```

## Constantes

Además de métodos y propiedades, en una clase también se pueden definir **constantes** utilizando la palabra **const**.

No hay que confundir los atributos con las constantes.

Son conceptos distintos las constantes no pueden cambiar su valor ( de ahí su nombre), no usan el carácter **\$** y está asociado a la clase, es decir, no existe una copia del mismo en cada objeto.

Por tanto, para acceder a las constantes de una clase, se debe utilizar el nombre de la clase y el operador **::** llamado **operador de resolución de ámbito** (que se utiliza para acceder a los elementos de una clase).

```
class BaseDatos{
    const USUARIO = "dwes";
    ...
}
```

```
echo BaseDatos::USUARIO;
```

No es necesario que exista ningún objeto de una clase para poder acceder al valor de las constantes que defina.

Además, sus nombres suelen escribirse en mayúsculas.

Ejercicio: crea una clase BaseDatos que tendrá como constantes el dominio donde está alojada, el usuario y la contraseña y la base de datos a utilizar.

## Métodos estáticos

Una clase puede tener atributos o métodos estáticos también llamados a veces atributos o métodos de clase. Se definen utilizando la palabra clave **static**

```
class Producto{
    private static $num_productos = 0;
    public static function nuevoProducto(){
        self::$num_productos++;
    }
    ...
}
```

Los atributos y métodos estáticos **no** pueden ser llamados desde un objeto de la clase utilizando el operador `->`.

- Si el método o atributo es público , deberá accederse utilizando el nombre de la clase y el operador de resolución de ámbito `::`.

```
Producto:: nuevoProducto();
```

- Si es privado , como el atributo `$num_productos` en el ejemplo anterior, sólo se podrá acceder a él desde los métodos de la propia clase, utilizando la palabra **self** . De la misma forma que `$this` hace referencia al objeto actual, `self` hace referencia a la clase actual.

```
self:: $num_productos++;
```

## Utilización de objetos

Una vez creado un objeto, puedes utilizar el operador instanceof para comprobar si es o no una instancia de una clase determinada.

```
if ($p instanceof Producto){  
    ...  
}
```

Desde PHP7 se incluyen una serie de funciones útiles para el desarrollo de aplicaciones utilizando POO.

| Función                     | Ejemplo  | Significado  |
|-----------------------------|--|--|
| <b>get_class</b>            | echo "La clase es:".get_class(\$p);                          | Devuelve el nombre de la clase del objeto  |
| <b>class_exists</b>         | if (class_exists('Producto'))<br>{ \$p= new Producto(); ...} | Devuelve true si la clase está definida o false en caso contrario  |
| <b>get_declared_classes</b> | print_r(get_declared_classes());                             | Devuelve un array con los nombres de las clases definidas  |
| <b>class_alias</b>          | class_alias('Producto','Articulo');<br>\$p = new Articulo(); | Crea un alias para una clase   |
| <b>get_class_methods</b>    | print_r(get_class_methods('Producto'));                      | Devuelve un array con los nombres de los métodos de una clase que son accesibles desde donde se hace la llamada                                  |
| <b>method_exists</b>        | if (method_exists('Producto','vende')){<br>....}             | Devuelve true si existe el método en el objeto o la clase que se indica, o false en caso contrario, independientemente de si es accesible o no   |
| <b>get_class_vars</b>       | print_r(get_class_vars('Producto'));                         | Devuelve un array con los nombre de los atributos de una clase que son accesibles desde dónde se hace la llamada                                 |
| <b>get_object_vars</b>      | print_r(get_object_vars(\$p));                               | Devuelve un array con los nombres de los atributos de un objeto que son accesibles desde dónde se hace la llamada                                |
| <b>property_exists</b>      | if(property_exists('Producto','codigo'))<br>{...}            | Devuelve true si existe el atributo en el objeto o la clase que se indica, o false en caso contrario, independientemente de si es accesible o no |

Desde PHP5, puedes indicar en las funciones y métodos de qué clase deben ser los objetos que se pasen como parámetros. Para ello, debes especificar el tipo antes del parámetro.

```
public function vendeProducto( Producto $p){  
    ...  
}
```

Si cuando se realiza la llamada, el parámetro no es del tipo adecuado, se produce un error que se podría capturar.

Pregunta: ¿Qué sucede al ejecutar el siguiente código?

```
$p = new Persona();  
$p->nombre = 'Pepe';  
$a=$p;
```

Desde PHP5 El código anterior simplemente crearía un **nuevo identificador del mismo objeto**. Esto es, en cuanto se utilice uno cualquiera de los identificadores para cambiar el valor de algún atributo, este cambio se vería también reflejado al acceder utilizando el otro identificador. Aunque haya dos o más identificadores del mismo objeto, en realidad todos se refieren a la única copia que se almacena del mismo. Tiene un comportamiento similar al que ocurre en Java.

Si necesitas **copiar un objeto**, debes utilizar **clone**. Al utilizar clone sobre un objeto existente, se crea una copia de todos los atributos del mismo en un nuevo objeto.

```
$p = new Producto();  
$p->nombre = 'Xiaomi 13';  
$a = clone $p;
```

Además, existe una forma sencilla de personalizar la copia para cada clase particular.

Por ejemplo, puede suceder que quieras copiar todos los atributos menos alguno. En nuestro ejemplo, al menos el código de cada producto debe ser distinto y, por tanto, quizás no tenga sentido copiarlo al crear un nuevo objeto.

Si éste fuera el caso, puedes crear un método de nombre **\_\_clone** en la clase. Este método se llamará automáticamente después de copiar todos los atributos en el nuevo objeto.

```
class Producto {  
    ...  
    public function __clone(){  
        $this->codigo = nuevo_codigo();  
    }  
    ...  
}
```



A veces tienes dos objetos y quieres saber su relación exacta. Para eso, puedes utilizar los operadores == y ===

- Si utilizas **el operador de comparación ==**, comparas los valores de los atributos de los objetos. Por tanto dos objetos serán iguales si son instancias de la misma clase y, además, sus atributos tienen los mismos valores.

```
$p = new Producto();
$p->nombre = 'Xiaomi 13';
$a = clone $p;
// El resultado de comparar $a==$p da verdadero pues $a y $p son dos copias idénticas
```

- Sin embargo, si utilizas **el operador ===**, el resultado de la comparación será true sólo cuando las dos variables sean referencias al mismo objeto.

```
$p = new Producto();
$p->nombre = 'Xiaomi 13';
$a = clone $p;
// El resultado de comparar $a=== $p da falso pues $a y $p no hacen referencia al mismo objeto
$a = $p;
// Ahora el resultado de comparar $a=== $p da verdadero pues $a y $p son referencias al mismo objeto.
```

## Herencia

La herencia es un mecanismo de la POO que nos permite definir nuevas clases en base a otra ya existente. Las nuevas clases que heredan también se conocen con el nombre de **subclases**. La clase de la que heredan se llama **clase base o superclase**.

Por ejemplo si tenemos la clase Persona, podemos crear las subclases Alumno y Profesor

```
class Persona{
    protected $nombre;
    protected $apellidos;
    public function muestra(){
        print "<p>".$this->nombre.", ".$this->apellidos."</p>";
    }
}
```

Esto puede ser útil si todas las personas sólo tuviesen nombre y apellidos, pero los alumnos tendrán un conjunto de notas y los profesores tendrán, por ejemplo, un número de horas de docencia.

```
class Alumno extends Persona{
    private $notas;
}
```

Ejercicio : Codificar estas dos clases y crear un objeto de tipo Alumno. Comprobar mediante el operador instanceof si el objeto es de tipo Persona o de tipo Alumno

| Función                 | Ejemplo  | Significado   |
|-------------------------|--|---|
| <b>get_parent_class</b> | echo "La clase padre es:".get_parent_class(\$p); | Devuelve el nombre de la clase padre del objeto o la clase que se indica  |
| <b>is_subclass_of</b>   | if (is_subclass_of(\$t,'Producto')) { ...}       | Devuelve true si el objeto o la clase del primer parámetro, tiene como clase base a la que se indica en el segundo parámetro, o false en caso contrario |

La nueva clase hereda todos los atributos y métodos públicos de la clase base, pero no los privados.

Si quieres crear en la clase base un método no visible al exterior (como los privados) que se herede a las subclases, debes utilizar la palabra **protected** en lugar de private. Además, puedes redefinir el comportamiento de los métodos existentes en la clase base, simplemente creando en la subclase un nuevo método con el mismo nombre.

```
class Profesor extends Persona{
    private $horas;
    public function muestra(){
        print "<p>".$this->nombre.", ".$this->apellidos." : ".$this->horas."</p>";
    }
}
```

Existe una forma de evitar que las clases heredadas puedan redefinir el comportamiento de los métodos existentes en la superclase: utilizar la palabra **final** . Si en nuestro ejemplo hubiéramos hecho:

```
class Persona{
    protected $nombre;
    protected $apellidos;
    public final function muestra(){
        print "<p>".$this->nombre.", ".$this->apellidos."</p>";
    }
}
```

En este caso el **método muestra** no podría redefinirse en la clase Profesor.

Incluso se puede declarar **una clase utilizando final** . En este caso no se podrían crear clases heredadas utilizándola como base.

```
final class Persona{  
    ...  
}
```

Opuestamente al **modificador final** existe también el **modificador abstract**. Se utiliza de la misma forma, tanto con métodos como con clases completas, pero en lugar de prohibir la herencia, obliga a que se herede. Es decir, una clase con el modificador abstract no puede tener objetos que la instancien, pero sí podrá utilizarse de clase base y sus subclasses sí podrán utilizarse para instanciar objetos.

```
abstract class Persona{  
    ...  
}
```

Y un método en el que se indique **abstract** debe ser redefinido obligatoriamente por las subclasses, y no podrá contener código.

```
class Persona{  
    ...  
    abstract public function muestra();  
}
```

Ejercicio: Crea un constructor para la clase Persona. ¿Qué pasará ahora con la clase Alumno, que hereda de Persona? Cuando crees un nuevo objeto de esa clase, ¿se llamará al constructor de Persona? ¿Puedes crear un nuevo constructor específico para Alumno que redefina el comportamiento de la clase base?

Desde PHP7, si la clase heredada no tiene constructor propio, se llamará automáticamente al constructor de la clase base (si existe). Sin embargo, si la clase heredada define su propio constructor, deberás ser tú el que realice la llamada al constructor de la clase base si lo consideras necesario, utilizando para ello la palabra **parent** y el **operador de resolución de ámbito**

```
class Alumno extends Persona{  
    private $notas;  
    public function __construct($nombre,$apellidos,$notas){  
        parent::__construct($nombre,$apellidos);  
        $this->notas=$notas;  
    }  
}
```

La utilización de la palabra **parent** es similar a **self**. Al utilizar parent haces referencia a la clase base de la actual



## Interfaces

Un interface es similar a una clase vacía que solamente contiene declaraciones de métodos. Se definen utilizando la palabra **interface**.

Por ejemplo, antes viste que podías crear nuevas clases heredadas de Persona como Profesor o Alumno. También viste que en las subclases podías redefinir el comportamiento del método muestra para que generara una salida en HTML diferente para cada tipo de persona.

Si quieres asegurarte de que todos los tipos de persona tengan un **método muestra** puedes crear un interface como el siguiente:

```
interface iMuestra{  
    public function muestra();  
  
}
```

Y cuando crees las subclases deberás indicar con la palabra **implements** que tienen que implementar los métodos declarados en este interface.

```
class Profesor extends Persona implements iMuestra{  
    ...  
    public function muestra(){  
        print "<p>".$this->nombre .": " . $this->horas."</p>";  
    }  
    ...  
}
```

Todos los métodos que se declaren en un interface deben ser **públicos**. Además de métodos, los interfaces podrán contener constantes pero no atributos.

Un interface es como un contrato que la clase debe cumplir. Al implementar todos los métodos declarados en el interface se asegura la interoperabilidad entre clases. Si sabes que una clase implementa un interface determinado, sabes qué nombre tienen sus métodos, qué parámetros les debes pasar y, probablemente, podrás averiguar fácilmente con qué objetivo han sido escritos.

Por ejemplo, en la librería de PHP está definido el **interface Countable**

```
interface Countable{  
    abstract public int count ();  
}
```

Desde PHP7, es posible crear clases que implementen varios interfaces, simplemente separando la lista de interfaces por comas después de la palabra **implements**.

```
class Profesor extends Persona implements iMuestra, Countable{  
    ...  
}
```

La única restricción es que los nombres de los métodos que se deban implementar en los distintos interfaces no coincidan. Es decir, en nuestro ejemplo, el interface **iMuestra** no podría contener **un método count** pues éste ya está declarado en **Countable**.


Desde PHP 7 también se pueden crear nuevos interfaces heredando de otros ya existentes. Se hace de la misma forma que con las clases, utilizando la palabra **extends**.

Una de las dudas más comunes en POO, es qué solución adoptar en algunas situaciones interfaces o clases abstractas. Ambas permiten definir reglas para las clases que los implementen o hereden respectivamente. Y ninguna permite instanciar objetos. Las diferencias principales entre ambas opciones son:

- En las clases abstractas, los métodos pueden contener código. Si van a existir varias subclases con un comportamiento común, se podría programar en los métodos de la clase abstracta. Si se opta por un interface, habría que repetir el código en todas las clases que lo implemente.
- Las clases abstractas pueden contener atributos, y los interfaces no.
- No se puede crear una clase que herede de dos clases abstractas, pero sí se puede crear una clase que implemente varios interfaces.

Para finalizar con los interfaces, a la lista de funciones de PHP relacionadas con la POO puedes añadir las siguientes:

| Función                        | Ejemplo  | Significado   |
|--------------------------------|--|---|
| <b>get_declared_interfaces</b> | <code>print_r(get_declared_interfaces());</code>     | Devuelve un array con los nombres de los interfaces declarados                |
| <b>interface_exists</b>        | <code>if (interface_exists('iMuestra')){ ...}</code> | Devuelve true si existe el interface que se indica o false en caso contrario. |

 Hoja03\_PHP\_06

## Traits

- Desde la versión 5.4 PHP implementa una metodología de código llamada Traits (rasgos).
- Los traits son un mecanismo de reutilización de código en lenguajes que tienen herencia simple.
- El objetivo de los traits es reducir las limitaciones de la herencia simple permitiendo reutilizar métodos en varias clases independientes y de distintas jerarquías.
- Un trait es similar a una clase, pero su objetivo es agrupar funcionalidades específicas.
- Un trait, al igual que las clases abstractas, no se puede instanciar, simplemente facilita comportamientos a las clases sin necesidad de usar la herencia.

```

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

```

**El trait** no impone ningún comportamiento en la clase, simplemente es como si copiaras los métodos del trait y los pegaras en la clase.

**Un trait** no puede implementar interfaces ni extender clases normales ni abstractas.

También se puede usar traits dentro de otros traits:

```

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

```

### Orden de precedencia entre traits y clases

Utilizar traits dentro de otros traits es frecuente cuando la aplicación crece y hay numerosos traits que pueden agruparse para tener que incluir menos en una clase.

Eso hace que pueda haber métodos con el mismo nombre en diferentes traits o en la misma clase, por lo que tiene que haber un orden. Existe **un orden de precedencia** de los métodos disponibles en una **clase** respecto a los de los **traits**:

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

### Ejemplo

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

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

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

## conflictos entre métodos de traits

Cuando se usan **múltiples traits** es posible que haya **diferentes traits que usen los mismos nombres de métodos**. PHP devolverá un **error fatal**. Para evitar este error hay que usar dentro de la clase la palabra **insteadof**

Ejemplo:

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

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

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

## Usar Reflection con traits

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



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

## Namespaces

Los namespaces o **espacios de nombres** permiten crear aplicaciones complejas con mayor flexibilidad evitando problemas de conflictos entre clases y mejorando la legibilidad del código.

Un namespace no es más que un directorio para **clases, traits, interfaces, funciones y constantes**. Se crean utilizando la palabra reservada namespace al principio del archivo, antes que cualquier otro código, a excepción de la estructura de control declare.

Supongamos que definimos la siguiente clase:

```
// MiClase.php
class MiClase {
// ...codigo
}
```

Si queremos instanciar la clase en otro archivo basta con escribir

```
include 'MiClase.php';

$miClase = new MiClase();
```

Con un proyecto sencillo no habría dificultades con esta metodología. Los problemas pueden venir cuando el proyecto aumenta y puede ocurrir que coincidan clases, funciones o constantes de PHP o de **librerías de terceros** con las del propio proyecto.

Si ahora cambio el fichero a un nuevo directorio

```
// Directorio: Proyecto/Prueba/MiClase.php
namespace Proyecto\Prueba;

class MiClase {
//...
}
```

Para utilizar **la clase**

```
include 'Proyecto/Prueba/MiClase.php';

$miClase = new Proyecto\Prueba\MiClase();
```

Siempre se emplea como namespace el **directorio de la clase**. No es obligatorio pero es lo más recomendable.

**Los namespaces** proporcionan una forma de agrupar clases, interfaces, funciones y constantes relacionadas.

#### la constante **NAMESPACE**

**NAMESPACE** Es una constante mágica que devuelve un string con el nombre del namespace actual:

```
namespace MiProyecto;

echo "Namespace actual: " . __NAMESPACE__;
```

Esta constante se puede utilizar para **construir nombres dinámicamente**:

```
namespace MiProyecto;

function obtener($nombreClase){
    $x = __NAMESPACE__ . '\\'. $nombreClase();
    return new $x;
}
```

#### La palabra reservada **namespace**

La palabra reservada **namespace** es equivalente a **self** en las **clases**.

Se utiliza para solicitar un elemento del namespace actual:

```
namespace MiProyecto;

namespace\miFuncion(); // llama a MiProyecto\miFuncion()
namespace\MiClase::metodo(); // llama al método estático metodo() de la clase
MiClase. Equivale a MiProyecto\MiClase::metodo()
```

#### Importar un namespace

Si tenemos que utilizar una clase varias veces en un archivo, podemos evitar tener que escribir el namespace tantas veces **importándolo**, así después solo habrá que escribir la clase:

```
include 'Proyecto/Prueba/MiClase.php';

use Proyecto\Prueba\MiClase;

$miClase = new MiClase();
```

### Utilizar alias en los namespace

Utilizar un **alias** para utilizar una clase bajo un nombre diferente:

```
include 'Proyecto/Prueba/MiClase.php';

use Proyecto\Prueba\MiClase as Clase;

$miClase = new Clase(); // instancia un objeto de la clase Proyecto\Prueba\MiClase

// No es posible con nombres dinámicos
$x = 'MiClase';
$objeto = new $x; // instancia de la clase MiClase. No detecta el apodo
```

### Enumeraciones

Desde PHP 8.1 se pueden utilizar los enumerados. Su sintaxis es muy parecida a la forma con que trabajamos con clases:

```
enum Sede{
    case ESPAÑA;
    case PORTUGAL;
    case MARRUECOS;
}
```

Esto nos permite no solo encapsular una colección de valores, sino que además, podemos emplear el enumerado que hemos declarado para tipar las variables

```
class Mundial{
    public function __construct(
        private Sede $sede
    ){}
}
```

Lo que nos permite crear el objeto del siguiente modo:

```
$mundial = new Mundial(Sede::ESPAÑA);  
echo $mundial->name; // para sacar el nombre
```

los enumerados son objetos creados con el patrón singleton, lo cual nos permite compararlos y obtener el resultado esperado

```
$sedeEsp=Sede::ESPAÑA;  
$sedePor=Sede::PORTUGAL;  
$sedeActual=Sede::ESPAÑA;  
  
$sedeActual===$sedeEsp; //true  
$sedeActual===$sedePor; //false  
$sedeActual instanceof Sede; //true
```

### Métodos dentro de un enumerado

Al igual que las clases, los enumerados también pueden definir métodos

```
enum Sede{  
    case ESPAÑA;  
    case PORTUGAL;  
    case MARRUECOS;  
    public function lugar():string{  
        return match($this){  
            self::ESPAÑA => 'Madrid',  
            self::PORTUGAL=> 'Lisboa',  
            self::MARRUECOS=> 'Rabat'  
        };  
    }  
}  
  
$sedeAct=Sede::ESPAÑA;  
echo $sedeAct->lugar();
```

## Funciones relacionadas con los tipos de datos

En PHP existen funciones específicas para comprobar y establecer el tipo de datos de una variable, **gettype** obtiene el tipo de la variable que se le pasa como parámetro y devuelve una cadena de texto, que puede ser array, boolean, double, integer, object, string, null, resource o unknown type.

También podemos comprobar si la variable es de un tipo concreto utilizando una de las siguientes funciones: **is\_array()**, **is\_bool()**, **is\_float()**, **is\_integer()**, **is\_null()**, **is\_numeric()**, **is\_object()**, **is\_resource()**, **is\_scalar()** e **is\_string()**. Devuelven true si la variable es del tipo indicado.

```
<?php
//asignamos a las dos variables la misma cadena de texto
$a = $b = "3.1416";
settype($b, "float"); //y cambiamos $b a tipo float
print "\$a vale $a y es de tipo ".gettype($a);
print "<br />";
print "\$b vale $b y es de tipo ".gettype($b);
?>
```

## Formularios

Son la forma de hacer llegar los datos a una aplicación web.

Van encerrados en las etiquetas

```
<form>
    ...
</form>
```

Dentro de las etiquetas de formulario se pueden incluir elementos sobre los que puede actuar el usuario.  
Ejemplo:

```
<input>
<select>
<textarea>
<button>
```

En el **atributo action** del FORM se indica la página a la que se enviarán los datos del formulario  
En el **atributo method** se especifica el método usado para enviar la información:

- **get**: los datos se envían en la URI utilizando el signo **?** como separador.
- **post**: los datos se incluyen en el cuerpo del formulario y se envían usando el protocolo HTTP

## Métodos GET y POST

Son métodos del protocolo HTTP para intercambio de información entre cliente y servidor.

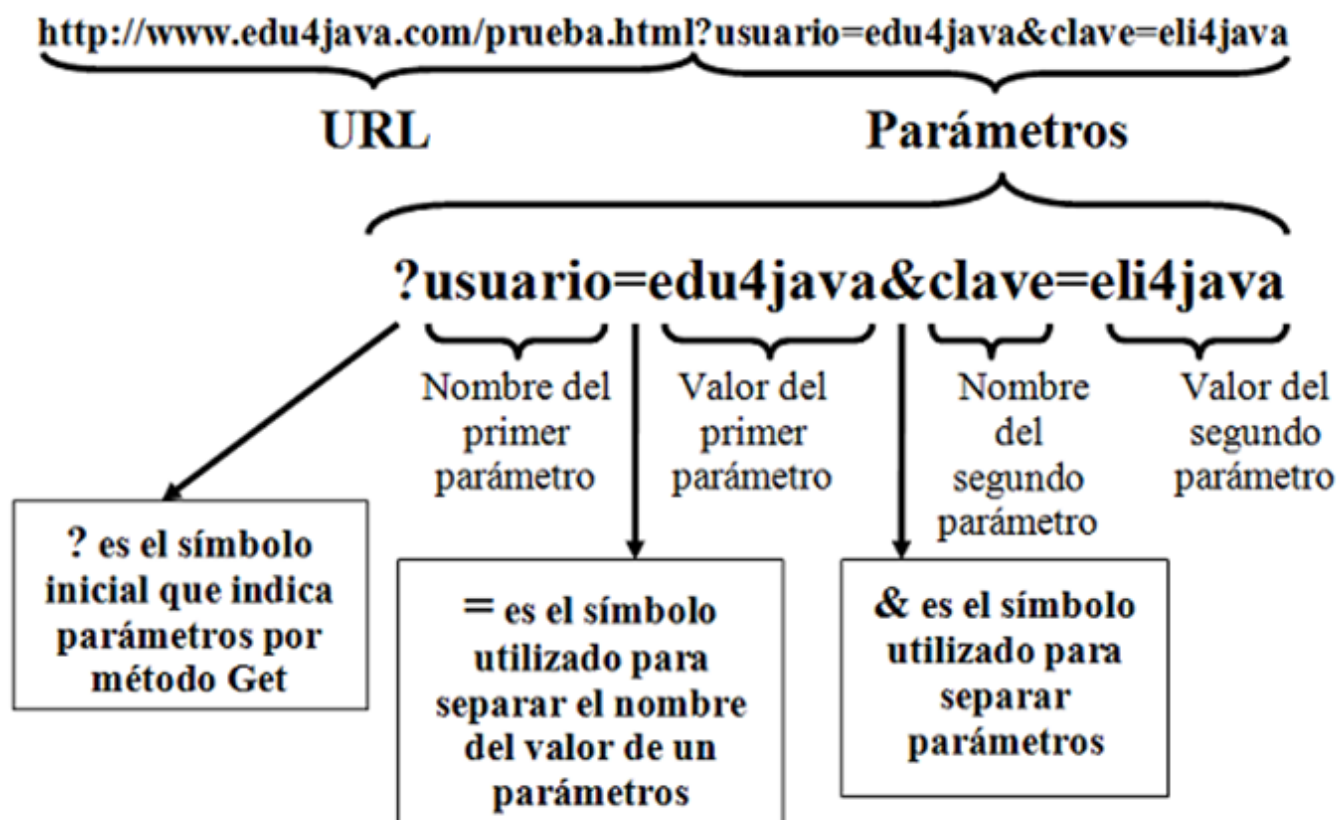
- **get**: el método más usado, sin embargo en formularios está en desuso. Pide al servidor que le devuelva al cliente la información identificada en la URI.
- **post**: se usa para enviar información a un servidor. Puede ser usado para completar un formulario de autenticación, por ejemplo.

La principal diferencia radica en la codificación de la información.

### Método GET

Utiliza la dirección URL que está formada por:

- **Protocolo:** especifica el protocolo de comunicación
- **Nombre de dominio:** nombre del servidor donde se aloja la información.
- **Directorios:** secuencia de directorios separados por /que indican la ruta en la que se encuentra el recurso
- **Fichero:** nombre del recurso al que acceder.
- Detrás de la URL se coloca el símbolo **?** Para indicar el comienzo de las variables con valor que se enviarán, separadas cada una de ellas por **&**.
- **Ejemplo:** <http://www.example.org/file/example1.php?v1=0&v2=3>



```
<!DOCTYPE html>
<html>
<head>
  <title>Ejemplo Formularios</title>
</head>
<body>
  <h1>Ejemplo de procesamiento de formularios</h1>
  <form action="ejemplo1.php" method="get">
    <label for="nombre">Introduzca su nombre:</label>
    <input type="text" id="nombre" name="nombre">
    <br/>
    <label for="apellido">Introduzca sus apellidos:</label>
    <input type="text" id="apellido" name="apellidos"><br/>
    <input type="submit" name="enviar" value="Enviar">
  </form>
</body>
</html>
```

## Método POST

La información va codificada en el cuerpo de la petición HTTP y por tanto viaja oculta.

- No hay un método más seguro que otro, ambas tienen sus pros y sus contras.
- Es conveniente usarlos GET para la recuperación y POST para el envío de información.

```
<!DOCTYPE html>
<html>
<head>
  <title>Ejemplo Formularios</title>
</head>
<body>
  <h1>Ejemplo de procesamiento de formularios</h1>
  <form action="ejemplo1.php" method="post">
    <label for="nombre">Introduzca su nombre:</label>
    <input type="text" id="nombre" name="nombre">
    <br/>
    <label for="apellido">Introduzca sus apellidos:</label>
    <input type="text" id="apellido" name="apellidos"><br/>
    <input type="submit" name="enviar" value="Enviar">
  </form>
</body>
</html>
```



Hoja3\_PHP\_07(ejercicio1 y ejercicio2)

## Recuperación de información

### Con GET

En el caso del envío de información utilizando el método GET existe una variable especial `$_GET`, donde se almacenan todas las variables pasadas con este método.

La forma de almacenar la información es **un array en el que el índice es el nombre asignado al elemento del formulario**

```
$_GET['nombre'];
$_GET['apellidos'];
```

También se puede recuperar con la función **print\_r** que muestra el array completo.

```
<?php
print_r($_GET);
?>
```

## Con POST

Al igual que en el método GET, se utiliza una variable interna `$_POST`, donde se almacenan todas las variables pasadas con este método.

La forma de almacenar la información, también es **un array en el que el índice es el nombre asignado al elemento del formulario**

```
$_POST['nombre'];
$_POST['apellidos'];
```

También se puede recuperar con la función **print\_r** que muestra el array completo.

Existe otra variable `$_REQUEST` que contiene tanto el contenido de `$_GET` como `$_POST`



Hoja03\_PHP\_07(ejercicio3 y ejercicio4)

## Validación de datos

Siempre que sea posible, es preferible **validar los datos que se introducen en el navegador antes de enviarlos. Para ello deberás usar código en lenguaje Javascript.**

Si por algún motivo hay datos que se tengan que validar en el servidor, por ejemplo, porque necesites comprobar que los datos de un usuario no existan ya en la base de datos antes de introducirlos, **será necesario hacerlo con código PHP en la página que figura en el atributo action del formulario.**

```
<html>
  <head> <title>Desarrollo Web</title> </head>
  <body>
```

```
<?php
    if (isset($_POST['enviar'])) {
        $nombre = $_POST['nombre'];
        $modulos = $_POST['modulos'];
        print "Nombre: ".$nombre."<br />";
        foreach ($modulos as $modulo) { print "Modulo: ".$modulo."<br />";
    }
    }
    else {
?>
```

```
<form name="input" action="" method="post">
  <label for="nombre">Nombre del alumno: </label>
  <input type="text" name="nombre" id="nombre" />
```



```
        <br />
        <p>Módulos que cursa:</p>
        <input type="checkbox" name="modulos[]" value="DWES" />Desarrollo web
en entorno servidor<br />
        <input type="checkbox" name="modulos[]" value="DWEC" />Desarrollo web
en entorno cliente<br /> <br />
        <input type="submit" name="enviar" value="Enviar" />
    </form>
```

```
<?php    }    ?>
```

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

en el **atributo action** del formulario también se puede poner

```
<?php echo $_SERVER['PHP_SELF'];
?>
```

