UT03 Programación basada en lenguaje de marcas con código embebido

Índice

Este material está basado en:

1. Libro de título: Desarrollo web en Entorno Servidor, de Juan Luis Vicente Carro, Editorial Garceta.

2. https://cizacas.github.io/DWES2425/

3. https://www.php.net/manual/es/index.php

Estructuras de control

En este capítulo estudiaremos las principales estructuras de control que nos brinda el lenguaje de servidor PHP a la hora de utilizarlo de forma embebida (incrustada) en el documento HTML.

Estructuras condicionales

En los lenguajes de programación una de las estructuras más básicas y utilizadas son las estructuras de control condicionales, que permiten realizar una bifurcación condicional del flujo de ejecución del programa, eligiendo un camino u otro a partir de la evaluación de una condición booleana (valores true/false).

If…else

```php
<?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";
?>
```

En este ejemplo, ya que cada rama del if solo conlleva una sentencia no es necesario encapsularlas en {}.

Operadores ternarios

De la misma manera es posible definir un if mediante operadores ternarios como hemos visto en la unidad de trabajo anterior.

```php
<?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

En situaciones donde las alternativas son múltiples, vemos que la opción if…else es bastante farragosa a la hora de codificar, por ello utilizamos la estructura de selección múltiple switch.

Similar a enlazar varias sentencias if comparando una misma variable con diferentes valores.

```php
<?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

La estructura match introducida en PHP 8.0, ofrecen varias ventajas frente a la tradicional switch. Tiene una sintaxis más concisa y expresiva.

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

Es una expresión, no una estructura de control, ya que devuelve un valor que se puede asignar a una variable directamente.

```php
<?php
    $resultado = match($valor) {
    1 => 'Uno',
    2 => 'Dos',
    default => 'Otro',
};
?>
```

No requiere el uso de break y realiza una comparación estricta (===)

Estructuras repetitivas o bucles

Pueden basarse en un número limitado de repeticiones o bien ejecutar el código en base a la evaluación de una condición que determina su continuidad. En el caso de utilizarse una estructura repetitiva basada en una condición, esta se evalúa en cada iteración realizada en el bucle.

Condicionales

While

Define un bucle que se ejecuta mientras se cumpla una expresión. La expresión se evalúa antes de comenzar cada ejecución del bucle.

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

?>
```

Do...While

Similar al anterior pero la expresión se evalúa al final, con lo cúal se asegura que la sentencia/as del bucle se ejecutan al menos una vez.

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

Iterativas

Otro tipo de estructuras son aquellas en las que el bucle o conjunto de sentencias a ejecutar se encuentra limitado por un número de veces conocido de antemano, en lugar de estar basada en si se cumple una determinada condición.

For

Compuesto por tres expresiones

Expr1: Se ejecuta una vez al comienzo del bucle
Expr2: Se Evalúa para saber si se debe ejecutar o no la/as sentencia/as
Expr3: Se ejecuta tras ejecutar todas las sentencias del bucle

```php
<?php

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

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

Foreach

Existen casos donde es necesario recorrer todos los elementos que conforman una colección de objetos o bien los elementos que componen una matriz con independencia de su número.

```php
<?php

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

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

Funciones

A través de la instrucción function() podremos definir una función que albergue el código que realiza una tarea concreta y ejecutarla tantas veces como sea necesario con una simple llamada.

```php
<?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();
?>
```

- 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.

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

```php
<?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
?>
```

Argumentos en las funciones

El resultado de ejecutar una función, de acuerdo a su definición, es retornar un valor al punto de invocación de la misma una vez ha realizado las tareas o sentencias declaradas en su contenido, a través del comando return.

```
<?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;
?>
```

Siempre es mejor utilizar **argumentos o parámetros al hacer la llamada**, de esta manera se evita tener que hacer uso de global si la variable no es local en la función.

Cuando en una función se encuentra una sentencia return, termina su procesamiento y devuelve el valor que se indica.

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
<?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 />";
?>
```

Se puede definir una función sin indicar los parámetros exactos para su declaración, pudiendo pasar un número indeterminado de parámetros con tres puntos por delante del nombre de la variable, lo trata como un array o vector.

```php
<?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 argumentos no se reflejará fuera del ámbito 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
<?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/>";
?>
```

Desde PHP 4, dispone de las funciones func_num_args(), que invocada dentro de la función indica el número de parámetros recibidos y func_get_arg() es la encargada de acceder a los elementos del array por posición. Además func_get_args() devuelve un array con todos los parámetros pasados a la función. Véase el siguiente ejemplo.

```php
<?php
    function comprobarArgumentos(){

        //Uso de func_num_args().
        $numeroDeArgumentos =func_num_args();
        echo("<u>Uso de func_num_args().</u><br>");
        echo("El número de argumentos es $numeroDeArgumentos<br>");
        echo("<br>");

        //Uso de func_get_arg().
        echo ("<u>Uso de func_get_arg().</u><br>");
        for($contador=0;$contador<$numeroDeArgumentos;$contador++){
            $argumento=func_get_arg($contador);
```

```
            echo ("El argumento $contador tiene el valor $argumento<br>");
        }
        echo("<br>");

        //Uso de func_get_args().
         echo ("<u>Uso de func_get_args().</u><br>");
         $matrizDeArgumentos=func_get_args();
          for($contador=0;$contador<$numeroDeArgumentos;$contador++){
            $argumento=$matrizDeArgumentos[$contador];
           echo ("El argumento $contador tiene el valor $argumento<br>");
        }
        echo("<br>");

    }

    $variable1=1;
    $variable2="Hola";
    $variable3="Adios";

    comprobarArgumentos($variable1,$variable2,$variable3);
?>
```

Tipos de variables definidas en los parámetros pasados a la función

Cuando se define una función a la que se indica que se le pasarán dos números enteros, en el caso de pasar un decimal como parámetro, no dará error ya que lo truncará, pero realmente no dará el resultado adecuado.

```
<?php
    function sumaEnteros(int $entero1, int $entero2){
        return $entero1+ $entero2;
    }
    $resultado=sumaEnteros(2,5);
    echo $resultado; //resultado=7
```

```php
    //también permite introducir un dato con decimal, lo trunca y no da error
    $resultado=sumaEnteros(2,5.6);
    echo $resultado; // resultado = 7
?>
```

La declaración **declare(strict_types=1);** en PHP se utiliza para **habilitar el modo estricto de tipos escalares.** Esto significa que los tipos de datos pasados a funciones deben coincidir exactamente con los tipos especificados en las declaraciones de tipo (por ejemplo, `int`, `string`, `float`, etc.), y **no se permite la conversión automática de tipos.**

```php
<?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 $resultado; //Fatal error: Uncaught TypeError: sumaEnteros(): Argument #2 ($entero2) must be of type int, float given,
?>
```

Por otro lado, es posible indicar a la función que tipo de datos tiene que devolver, de la siguiente manera.

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

Los tipos de datos que podemos especificar son int, float, string, bool, array, object, null

Inclusión de las funciones en ficheros externos

Cuando el código crece, es más cómodo **organizar funciones, clases o configuraciones** en archivos separados. Así puedes reutilizar código y mantener tus scripts más limpios y fáciles de mantener.

Formas de incorporar ficheros externos:

Include

```
include 'archivo.php';
```

¿Qué hace? Inserta el contenido del archivo en el punto donde se llama.

Ruta: Puede ser **relativa** ('includes/archivo.php')

o **absoluta** ('/var/www/html/includes/archivo.php').

Búsqueda: PHP busca primero en la ruta definida en include_path del php.ini. Si no lo encuentra, busca en el directorio actual.

Error: Si el archivo no existe, PHP **lanza un aviso (warning)** pero **sigue ejecutando el script.**

Include_once

Realiza la misma función de include con el añadido de que evita cargarlo más de una vez, evitando errores si se declara include dentro de un bucle por ejemplo.

Require

```
require 'archivo.php';
```

Su funcionamiento es como include pero con una diferencia clave y es que si el archivo no se encuentra se detiene el script con Error Fatal. Se utiliza cuando el archivo es esencial para que el programa funcione.

Require_once

Combina lo mejor de require y include_once, ya que evita duplicados y detiene la ejecución si el archivo no se encuentra. Es ideal para incluir archivos de configuración, clases o funciones que no deben repetirse.

Instrucción	¿Detiene ejecución si falla?	¿Evita duplicados?
include	❌ No	❌ No
include_once	❌ No	✅ Sí
require	✅ Sí	❌ No
require_once	✅ Sí	✅ Sí

Trabajar con cadenas de caracteres

Una cadena es una concatenación de caracteres, a continuación, se muestran las funciones PHP para trabajar con ellas.

- **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 reeemplazar
    - 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
<?php
 $cadena='aeioui';
 echo strpos($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
?>
```

Ojo con la función **strlen** cuenta el número de bits no el número de caracteres, Si necesitas contar caracteres en cadenas con codificación UTF-8 la función del número de caracteres es **mb_strlen**

```php
<?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
?>
```

Tipos de datos compuestos

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
<?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:

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

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

```php
<?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 uno de las dimensiones

```php
 // 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

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

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 servivor";
```

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.

```
<?php
$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

En PHP, la función list() se utiliza para **asignar valores de un array a variables individuales** de forma rápida y ordenada. Solo funciona cuando trabajas con arrays indexados numéricamente.

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


    //entonces $a=1 , $b=2,  $c=3
?>
```

Esto significa que:

- $a toma el valor del índice 0 del array,
- $b toma el valor del índice 1,
- $c toma el valor del índice 2, y así sucesivamente.

A tener en cuenta: Si el array tiene menos elementos que variables, las variables restantes se asignan como null.

Puedes omitir variables usando comas vacías:

```php
list(, $segundo) = [1, 2, 3];
echo $segundo; // 2
```

Range(valor1,valor2): Rellenar un array con un intervalo de valores comprendido entre valor1 y valor2

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


    $abecedario=range("a","z");
    var_dump($abecedario);
    echo $abecedario[5]; //devuelve el valor f
```

También es posible pasar un tercer valor a la expresión (step), es opcional y definiría el incremento entre elementos dentro del rango.

```
   $pares = range(2, 10, 2);
    print_r($pares);
```

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) Nos ayuda a saber si un elemento se encuentra dentro de un array, devuelve un true si lo encuentra.

```
<?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**.

Son **arrays predefinidos** que PHP crea automáticamente y que están disponibles **en cualquier parte del script**, sin necesidad de declararlos como globales. Se utilizan para acceder a datos del servidor, del cliente, del entorno, formularios, sesiones, etc.

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.

```php
<?php
echo $_SERVER['SERVER_NAME']; // Nombre del servidor (por ejemplo, localhost)
echo "<br>";
echo $_SERVER['REQUEST_METHOD']; // Método de la petición (GET, POST)
echo "<br>";
echo $_SERVER['REQUEST_URI']; // URI de la página solicitada
echo "<br>";
echo $_SERVER['HTTP_USER_AGENT']; // Información del navegador del cliente
echo "<br>";
echo $_SERVER['SCRIPT_NAME']; // Ruta del script actual
echo "<br>";
echo $_SERVER['SERVER_PORT']; // Puerto del servidor
?>
```

Output

localhost
GET
/Pruebas/prueba.php
Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/140.0.0.0 Safari/537.36
/Pruebas/prueba.php
3000

- **$_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.

$GET: Recoge datos enviados por la URL (método GET).

URL de ejemplo: `pagina.php?nombre=Juan`

```
echo $_GET['nombre'];  // Muestra "Juan"
```

$POST

```
// En el formulario HTML:
// <form method="post">
//   <input name="email">
// </form>

echo $_POST['email'];  // Muestra el email enviado
```

$COOKIE

```
echo $_COOKIE['usuario'];  // Muestra el valor de la cookie 'usuario
```

- **$_REQUEST**: junta en uno solo el contenido de los tres arrays anteriores.
- **$_ENV**: contiene las variables que se puedan haber pasado a PHP desde el entorno en que se ejecuta.

```
<?php
// Ejemplo de uso de variables de entorno en PHP
// Puedes acceder a las variables de entorno usando la superglobal $_ENV

echo $_ENV['PATH'];  // Muestra la ruta del sistema si está disponible
echo $_ENV['HOME'];  // Muestra el directorio home si está disponible
echo $_ENV['USER'];  // Muestra el nombre del usuario si está disponible
echo $_ENV['SHELL']; // Muestra el shell del usuario si está disponible
echo $_ENV['LANG'];  // Muestra la configuración regional si está disponible
echo $_ENV['TERM'];  // Muestra el tipo de terminal si está disponible
```

```php
echo $_ENV['PWD'];    // Muestra el directorio de trabajo actual si está disponible
echo $_ENV['LOGNAME']; // Muestra el nombre de inicio de sesión si está disponible


?>
```

- **$_FILES**: contiene los ficheros que se puedan haber subido al servidor utilizando el método POST.

```php
// En el formulario HTML:
// <input type="file" name="archivo">
echo $_FILES['archivo']['name'];  // Nombre del archivo subido
echo $_FILES['archivo']['size'];  // Tamaño del archivo
```

- **$_SESSION**: contiene las variables de sesión disponibles para el guión actual.

```php
session_start();
$_SESSION['usuario'] = "Juan";
echo $_SESSION['usuario'];  // Muestra "Juan"
```

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 simple
- 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.

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

Recordar que por lo general los atributos deberán ser privados

IMPORTANTE

- Cada clase en un fichero distinto y desde el script donde se quiera crear un objeto de esa clase se deberá utilizar

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

- 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:

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

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)

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

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.

```php
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.

Importante -> Sólo puede haber un método constructor en cada clase

```php
class Producto{

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

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).

Ejercicio: Crea la clase productos con los atributos que consideres necesarios y un constructor.

Destructores

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

Aunque PHP tiene un recolector de basura que gestiona la memoria automáticamente, hay situaciones en las que definir un destructor puede ser útil o incluso necesario.

Si tu objeto abre archivos, conexiones a bases de datos, etc., puedes usar `__destruct()` para cerrarlos adecuadamente.

```php
$objeto = new MiClase();
// ... uso del objeto
unset($objeto); // fuerza la llamada a __destruct()
```

```php
class Producto{
    private $codigo;
    public function __construct($codigo){
        $this->codigo = $codigo;
    }
    public function __destruct(){
         $this->codigo=0;
    }
}
```

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.

```php
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);
```

También es posible definir los atributos al pasar parámetros al constructor

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

Creando instancias desde un constructor

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

```php
class HolaMundo {
    public function saludar() {
        echo "¡Hola mundo!\n";
    }
}

class Saludo {
    public function __construct(
      //Atención aquí
        private readonly HolaMundo $holaMundo = new HolaMundo(),
    ) {}

    public function mostrarSaludo() {
        return $this->holaMundo->saludar();
    }
}
```

Este constructor:

- **Declara** la propiedad `$holaMundo` directamente en la firma del constructor.
- **La inicializa** con una instancia de `HolaMundo` si no se proporciona otra.
- **La hace inmutable** (`readonly`), lo que significa que no se puede cambiar después de la construcción del objeto.

Y más concretamente

`private readonly HolaMundo $holaMundo`

- `private`: la propiedad solo es accesible dentro de la clase.
- `readonly`: la propiedad **solo puede asignarse una vez**, normalmente en el constructor. Después de eso, **no puede modificarse.**
- `HolaMundo $holaMundo`: se espera un objeto de la clase `HolaMundo`.

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 clase producto creada

Métodos mágicos

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

Por ejemplo, el código siguiente simula que la clase Producto, que tiene una variable atributos, pero queremos acceder a atributo, que es un elemento del array.

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

⚠ Consideraciones

- No se recomienda abusar de estos métodos, ya que pueden hacer el código **menos transparente.**
- Si usas `__get()` y `__set()`, asegúrate de documentar bien tu clase para evitar confusiones.

Otro método mágico es __toString

```php
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);
    }
```

```php
}
$persona1 = new Persona("Juan", 25);
echo $persona1;
```

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

Constantes

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

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

Función	Ejemplo	Significado
		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
get_parent_class	echo “La clase padre es:”.get_parent_class($p);	Devuelve el nombre de la clase padre del objeto o la clase que se indica
is_subclass_of	if (is_subclass_of($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

Función	Ejemplo	Significado
		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.

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 subclases sí podrán utilizarse para instanciar objetos.

```
abstract class Persona{

    …
}
```

Y un método en el que se indique **abstract** debe ser redefinido obligatoriamente por las subclases, 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.

Hoja Ejercicios 05

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 y obligar a 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**

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

Cuando una clase implementa la interfaz Countable, está obligada a definir un método público llamado count().Este método no recibe ningún parámetro y debe devolver un valor de tipo entero (int), que representa el número de elementos que contiene el objeto.

Ejemplo

```php
class Carrito implements Countable {
    private array $items = [];

    public function addItem(string $item): void {
        $this->items[] = $item;
    }

    public function count(): int {
        return count($this->items);
    }
}

$miCarrito = new Carrito();
$miCarrito->addItem('Manzana');
$miCarrito->addItem('Banana');
$miCarrito->addItem('Naranja');

echo count($miCarrito); // Salida: 3
```

La clase Carrito implementa la interfaz Countable y define el método count(), que devuelve el número de elementos en el array $items.

A partir de PHP 7.3, se introdujo la función **is_countable()** para verificar si una variable es un array o un objeto de una clase que implementa la interfaz Countable, evitando así advertencias al intentar contar variables que no son contables.

Implementar varios interfaces

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**.

"Herencia múltiple" de interfaces

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**.

```
interface Dibujable {
    public function dibujar();
}

interface Redimensionable {
    public function cambiarTamaño($factor);
}

// Esta interfaz hereda los requisitos de OTRAS DOS.
interface ElementoGrafico extends Dibujable, Redimensionable {
```

```
    public function obtenerPosicion();
}

// Una clase que implemente ElementoGrafico deberá definir los 3 métodos:
// dibujar(), cambiarTamaño() y obtenerPosicion()
class Circulo implements ElementoGrafico {
    // ... implementación de los 3 métodos obligatorios
}
```

Diferencia entre clase abstracta e interfaces

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
get_declared_interfaces	print_r(get_declared_interfaces());	Devuelve un array con los nombres de los interfaces declarados

Función	Ejemplo	Significado
interface_exists	if (interface_exists('iMuestra')){ …}	Devuelve true si existe el interface que se indica o false en caso contrario.

Hoja Ejercicios 6

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
```

```php
$sedeActual===$sedePor;//flase
$sedeActual instanceof Sede; //true
```

Métodos dentro de un enumerado

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

```php
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();
```

Traits

- Desde la versión 5.4 PHP implementa una metodología de código llamada Traits (rasgos), 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.
- Una clase puede utilizar múltiples Traits, separándolos por comas en la declaración **use**.
- Los Traits pueden definir variables estáticas, métodos estáticos (Se llaman desde la clase) o también pueden definir constantes.

```php
//Traits

trait Saludar{
    function decirHola(){
        return "hola";
    }
}

trait Despedir{
    function decirAdios(){
        return "adios";
    }
}

//Clase
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:

```php
trait Saludar{
    function decirHola(){
        return "hola";
    }
}
trait Despedir{
    function decirAdios(){
        return "adios";
    }
}
```

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

```php
// definir el trait comunicación

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**

```
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

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:

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

Si queremos instanciar la clase en otro archivo basta con escribir

```php
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

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

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

Para utilizar **la clase**

```php
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:

```php
namespace MiProyecto;

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

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

```php
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:

```php
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:

```php
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:

```php
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
```

UT03 - Ejercicios 7

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.

Formularios

Es 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:

1. `<input>`

El elemento `<input>` es el más versátil de todos. Su comportamiento cambia drásticamente según su atributo `type`.

- **`<input type="text">`**:
  - **Uso:** Para una sola línea de texto (nombres, apellidos, etc.).
  - **Atributos clave:**
    - `name="nombre_usuario"`: Identificador para enviar los datos al servidor. Es crucial.
    - `id="nombre"`: Identificador único en la página, usado para enlazarlo con la etiqueta `<label>`.
    - `placeholder="Ej: Ana García"`: Texto de ayuda que aparece dentro del campo y desaparece al escribir.
    - `required`: Hace que el campo sea obligatorio.
- **`<input type="email">`**:
  - **Uso:** Específicamente para direcciones de correo electrónico. En la mayoría de navegadores, valida automáticamente que el formato sea correcto.
- **`<input type="date">`**:
  - **Uso:** Muestra un selector de fecha nativo del navegador.
- **`<input type="submit">`**:
  - **`Uso`**: Crea un botón especial diseñado específicamente para enviar los datos de un formulario. Cuando un usuario hace clic en este botón, el navegador recopila todos los datos de los campos (`<input>`, `<select>`, `<textarea>`, etc.) que estén dentro del mismo formulario (`<form>`) y los envía a la dirección (URL) especificada en el atributo `action` del formulario.
  - **Atributos clave:**
    - Value=”Iniciar sesión”. El texto a mostrar en el botón que describe la acción que desencadena.
- **`<input type="checkbox">`**:

- **Uso:** Para opciones de sí/no o selección múltiple.
- **Atributos clave:**
  - `value="si"`: El valor que se enviará al servidor si la casilla está marcada.

2. `<select>`

Se usa para crear una lista desplegable de opciones.

- **Uso:** Ideal cuando el usuario debe elegir una opción de una lista predefinida (países, categorías, motivos, etc.).
- **Estructura:**
  - La etiqueta `<select>` envuelve a todas las opciones.
  - Cada opción se define con una etiqueta `<option>`.
- **Atributos clave:**
  - `name="motivo_contacto"`: El identificador para el servidor.
  - `<option value="consulta">`: El `value` es el dato que se envía al servidor. El texto entre `<option>` y `</option>` es lo que ve el usuario.
  - La primera opción con `value=""` se usa comúnmente como un marcador de posición no seleccionable.

3. `<textarea>`

Para introducir texto de múltiples líneas.

- **Uso:** Perfecto para campos de comentarios, mensajes, biografías o descripciones largas.
- **Atributos clave:**
  - `name="mensaje_usuario"`: Identificador para el servidor.
  - `rows="6"`: Define la altura visible del área de texto en número de líneas.
  - `placeholder="Escriba aquí..."`: Igual que en `<input>`, es un texto de ayuda.

4. `<button>`

Define un botón clicable.

- **Uso:** Generalmente para enviar formularios o para ejecutar acciones con JavaScript.
- **Atributos clave:**
  - `type="submit"`: Este es el valor por defecto. Al hacer clic, el botón intentará enviar los datos del formulario al que pertenece.
  - `type="reset"`: Limpiaría todos los campos del formulario a sus valores iniciales.

  - `type="button"`: Un botón genérico que no hace nada por defecto, ideal para asociarle eventos de JavaScript.
- **Ventaja sobre `<input type="button">`**: Dentro de `<button>` puedes anidar otro contenido HTML, como imágenes (`<img>`) o texto con formato (`<strong>`), lo que lo hace más flexible.

Ejemplo:

```
<!DOCTYPE html>
<html>
<head>
    <title>Ejemplo Formularios</title>
</head>
<body>
    <h1>Ejemplo de procesado de formularios GET</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>
```

Importante:

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:

```
    <form action="ejemplo1.php" method="get">
```

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.

los datos se envían en la URI utilizando el signo **?** como separador.

- **post**: se usa para enviar información a un servidor. Puede ser usado para completar un formulario de autenticación, por ejemplo.

  los datos se incluyen en el cuerpo del formulario y se envían usando el protocolo HTTP

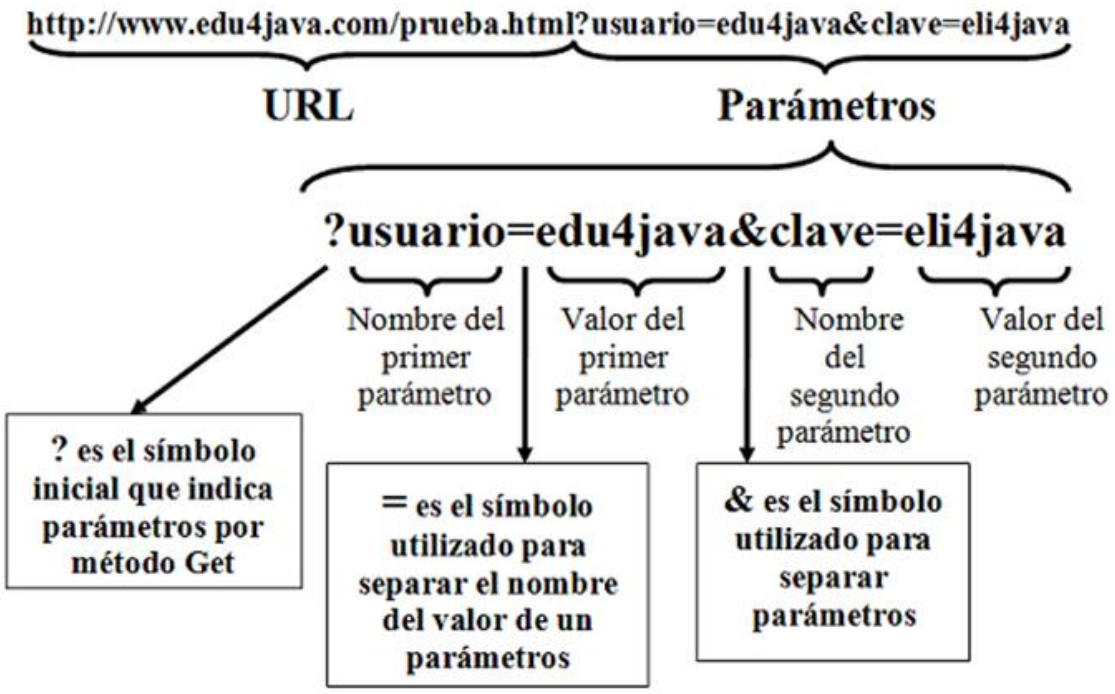
La principal diferencia radica en la codificación de la información.

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



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 POST</title>
</head>
<body>
    <h1>Ejemplo de procesado 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>
```

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
<?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**

```php
<?php
$_POST[‘nombre’];
$_POST[‘apellidos’];
echo "Hola " . $_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

Hoja Ejercicios 8