PHP

▼ TEMA2

2.1 Bloque de Código

- Para iniciar un documento PHP, se utiliza `<?php ... ?>`.

```
<?php
  // Tu código aquí
?>
```

- **Sentencias en PHP:**
  - Cada sentencia **debe** terminar con un punto y coma `;`.
  - La sentencia de impresión es `echo`.

```
<?php
  echo "Hola Mundo";
?>
```

- **Nota:** El código PHP se puede insertar en archivos con extensión `.php`. También es posible incrustar código PHP dentro de archivos HTML usando las etiquetas `<?php ... ?>`.

2.2 Variables y Constantes

Tipos de Variables

- **Integer:** Números enteros.

```
$numero = 10;
```

- **Float/Double:** Números con decimales.

```
$precio = 19.99;
```

- **String:** Cadenas de texto. Se definen entre comillas simples `'` o dobles `"`.

```
$nombre = "Juan";
```

- **Boolean:** Valores lógicos `true` o `false`.

```
$es_valido = true;
```

- **Array:** Colección de valores.

```
$frutas = ["Manzana", "Banana", "Naranja"];
$frutas = array("Manzana", "Banana", "Naranja");
```

- **Object:** Instancia de una clase.

```
$fecha = new DateTime();
```

Tipado Dinámico: En PHP, no es necesario declarar el tipo de variable; el tipo se asigna automáticamente según el valor.

Constantes

- Las constantes son valores que no cambian durante la ejecución del script.
- Se definen usando `define()` o la palabra reservada `const` (a partir de PHP 5.3.0).

```
define("PI", 3.1416);
echo PI; // Imprime: 3.1416
```

- **Características de las constantes:**

- No llevan el signo `$`.
- Por convención, se escriben en mayúsculas.
- Son globales en todo el script.

2.3 Operadores

Operadores Aritméticos

Operador	Descripción	Ejemplo	Resultado
`+`	Suma	`$suma = 5 + 3;`	`$suma` vale `8`
`-`	Resta	`$resta = 5 - 2;`	`$resta` vale `3`
`*`	Multiplicación	`$prod = 5 * 3;`	`$prod` vale `15`
`/`	División	`$div = 10 / 2;`	`$div` vale `5`
`%`	Módulo	`$mod = 10 % 3;`	`$mod` vale `1`
`**`	Exponenciación	`$exp = 2 ** 3;`	`$exp` vale `8`

- **Ejemplo de uso:**

```
$a = 10;
$b = 3;
echo $a % $b; // Imprime: 1
```

Operadores de Comparación

Operador	Descripción	Ejemplo	Resultado
`==`	Igual a	`$a == $b`	`true` si `$a` es igual a `$b`
`!=`	Diferente de	`$a != $b`	`true` si `$a` no es igual a `$b`
`===`	Idéntico (igual y mismo tipo)	`$a === $b`	`true` si `$a` es igual y del mismo tipo que `$b`
`!==`	No idéntico	`$a !== $b`	`true` si `$a` no es igual o no es del mismo tipo que `$b`
`<`	Menor que	`$a < $b`	`true` si `$a` es menor que `$b`

`>`	Mayor que	`$a > $b`	`true` si `$a` es mayor que `$b`
`<=`	Menor o igual que	`$a <= $b`	`true` si `$a` es menor o igual a `$b`
`>=`	Mayor o igual que	`$a >= $b`	`true` si `$a` es mayor o igual a `$b`

- **Ejemplo de uso:**

```
$a = 5;
$b = '5';

if ($a == $b) {
  echo "Son iguales"; // Esto se imprime
}

if ($a === $b) {
  echo "Son idénticos";
} else {
  echo "No son idénticos"; // Esto se imprime
}
```

Operadores Lógicos

Operador	Descripción	Ejemplo	Resultado
`&&`	AND (Y lógico)	`$a > 5 && $b < 10`	`true` si ambas condiciones son verdaderas
`||`	OR (O lógico)	`$a > 5 || $b < 10`	`true` si una de las dos son verdaderas
`!`	NOT (Negación)	`!$a`	`true` si `$a` es falso

- **Ejemplo de uso:**

```
$edad = 20;
$licencia = true;

if ($edad >= 18 && $licencia) {
```

```
  echo "Puede conducir";
}
```

Operador de Concatenación

- En PHP, se utiliza el punto `.` para concatenar cadenas de texto.

```
$saludo = "Hola";
$nombre = "Mundo";
echo $saludo . " " . $nombre; // Imprime: Hola Mundo
```

Operadores de Asignación

Operador	Descripción	Ejemplo	Equivalente a
`=`	Asignar	`$a = 5;`	
`+=`	Suma y asigna	`$a += 5;`	`$a = $a + 5;`
`-=`	Resta y asigna	`$a -= 5;`	`$a = $a - 5;`
`*=`	Multiplica y asigna	`$a *= 5;`	`$a = $a * 5;`
`/=`	Divide y asigna	`$a /= 5;`	`$a = $a / 5;`
`%=`	Módulo y asigna	`$a %= 5;`	`$a = $a % 5;`
`.=`	Concatenar y asignar	`$a .= " texto";`	`$a = $a . " texto";`

- **Ejemplo de uso:**

```
$mensaje = "Hola";
$mensaje .= " Mundo";
echo $mensaje; // Imprime: Hola Mundo
```

2.4 Sentencias de Control

Estructura If - Else

- **Sintaxis básica:**

```
if (condición) {
    // Código si la condición es verdadera
} else {
```

```
    // Código si la condición es falsa
}
```

- **Ejemplo:**

```
$edad = 17;

if ($edad >= 18) {
  echo "Es mayor de edad";
} else {
  echo "Es menor de edad";
}
```

Estructura If - Elseif - Else

- **Sintaxis:**

```
if (condición1) {
    // Código si condición1 es verdadera
} elseif (condición2) {
    // Código si condición2 es verdadera
} else {
    // Código si ninguna condición anterior es verdadera
}
```

- **Ejemplo:**

```
$nota = 85;

if ($nota >= 90) {
  echo "Excelente";
} elseif ($nota >= 70) {
  echo "Bueno";
} else {
  echo "Necesita mejorar";
}
```

Estructura Switch

- **Sintaxis:**

```
switch (variable) {
    case valor1:
        // Código si variable == valor1
        break;
    case valor2:
        // Código si variable == valor2
        break;
    default:
        // Código si ninguna condición anterior se cumpl
e
}
```

- **Ejemplo:**

```
$dia = "Lunes";

switch ($dia) {
  case "Lunes":
    echo "Hoy es lunes";
    break;
  case "Martes":
    echo "Hoy es martes";
    break;
  default:
    echo "Es otro día";
}
```

Bucles

Bucle While

- **Sintaxis:**

```
while (condición) {
    // Código a ejecutar mientras la condición sea verda
```

```
dera
}
```

- **Ejemplo:**

```
$i = 0;

while ($i < 5) {
  echo $i;
  $i++;
}
// Imprime: 01234
```

Bucle Do - While

- **Sintaxis:**

```
do {
    // Código a ejecutar
} while (condición);
```

- **Ejemplo:**

```
do {
  $numero = rand(1, 5);
  echo "Ha salido un: " . $numero . "<br>";
} while ($numero != 5);

echo "¡Felicidades, has sacado un 5!";
```

Bucle For

- **Sintaxis:**

```
for (inicialización; condición; incremento) {
    // Código a ejecutar en cada iteración
}
```

- **Ejemplo:**

```
for ($i = 0; $i < 10; $i++) {
  echo $i . " ";
}
// Imprime: 0 1 2 3 4 5 6 7 8 9
```

Bucle Foreach

- **Sintaxis:** Recorre arrays o colecciones.

```
foreach ($array as $valor) {
    // Código a ejecutar con cada $valor
}
```

- **Ejemplo:**

```
$frutas = ["Manzana", "Banana", "Naranja"];

foreach ($frutas as $fruta) {
  echo $fruta . "<br>";
}
// Imprime:
// Manzana
// Banana
// Naranja
```

- **Con clave y valor:**

```
$edades = ["Juan" => 25, "María" => 30, "Pedro" => 28];

foreach ($edades as $nombre => $edad) {
  echo $nombre . " tiene " . $edad . " años.<br>";
}
// Imprime:
// Juan tiene 25 años.
```

```
// María tiene 30 años.
// Pedro tiene 28 años.
```

▼ TEMA 3

3.1 Cadenas

Acceso a Caracteres

- En PHP, puedes acceder a caracteres individuales de una cadena usando corchetes `[]`.

```
$cadena = "Hola";
echo $cadena[0]; // Resultado: H
```

- **Nota:** Las cadenas en PHP son arrays de caracteres de tipo cero-indexado.

Funciones Útiles para Cadenas

1. `strlen($cadena)`: Retorna la longitud de una cadena.

   ```
   $longitud = strlen("Hola"); // Resultado: 4
   ```

2. `strpos($cadena, $buscar)`: Encuentra la posición de la primera ocurrencia de un substring.

   ```
   $pos = strpos("Hola Mundo", "Mundo"); // Resultado: 5
   ```

3. `substr($cadena, $inicio, $longitud)`: Extrae una parte de una cadena.

   ```
   $subcadena = substr("Hola Mundo", 0, 4); // Resultad
   o: "Hola"
   ```

4. `str_replace($buscar, $reemplazo, $cadena)`: Reemplaza todas las apariciones de un texto dentro de una cadena.

   ```
   $texto = "Hola Mundo";
   $nuevoTexto = str_replace("Mundo", "PHP", $texto);
   ```

```
echo $nuevoTexto; // Resultado: "Hola PHP"
```

5. `strtoupper($cadena)`: Convierte una cadena a mayúsculas.

```
$mayus = strtoupper("hola"); // Resultado: "HOLA"
```

6. `strtolower($cadena)`: Convierte una cadena a minúsculas.

```
$minus = strtolower("HOLA"); // Resultado: "hola"
```

7. `number_format($numero, $decimales)`: Formatea un número con una cantidad específica de decimales.

```
$formateado = number_format(1234.5678, 2); // Resulta
do: "1,234.57"
```

8. `trim($cadena)`: Elimina espacios en blanco al inicio y al final de una cadena.

```
$cadena = "   Hola Mundo   ";
$limpia = trim($cadena); // Resultado: "Hola Mundo"
```

9. `explode($delimitador, $cadena)`: Divide una cadena en un array.

```
$fecha = "2023-10-17";
$partes = explode("-", $fecha); // Resultado: ["202
3", "10", "17"]
```

10. `implode($delimitador, $array)`: Une elementos de un array en una cadena.

```
$array = ["Juan", "María", "Pedro"];
$cadena = implode(", ", $array); // Resultado: "Juan,
María, Pedro"
```

3.2 Arrays

Array Indexado

- Es una colección de elementos donde cada uno tiene un índice numérico.

```
$frutas = array("Manzana", "Pera", "Naranja");
echo $frutas[0]; // Resultado: "Manzana"
```

- A partir de PHP 5.4, puedes usar la sintaxis corta:

```
$frutas = ["Manzana", "Pera", "Naranja"];
```

Array Asociativo

- Es un array donde cada clave es una cadena que asocia un valor.

```
$capitales = [
  "Francia" => "París",
  "Italia" => "Roma",
  "España" => "Madrid"
];

echo $capitales["Italia"]; // Resultado: "Roma"
```

Recorrer Arrays

- **Con `foreach` para arrays indexados:**

```
$frutas = ["Manzana", "Pera", "Naranja"];

foreach ($frutas as $fruta) {
  echo $fruta . "<br>";
}
```

- **Con `foreach` para arrays asociativos:**

```
$capitales = [
  "Francia" => "París",
  "Italia" => "Roma",
  "España" => "Madrid"
```

```
];

foreach ($capitales as $pais => $ciudad) {
  echo "La capital de $pais es $ciudad.<br>";
}
```

Funciones Útiles para Arrays

1. `count($array)`: Devuelve el número de elementos en un array.

   ```
   $total = count($frutas); // Resultado: 3
   ```

2. `sort($array)`: Ordena los elementos de un array en orden ascendente.

   ```
   sort($frutas);
   // $frutas ahora es ["Manzana", "Naranja", "Pera"]
   ```

3. `rsort($array)`: Ordena los elementos en orden descendente.

   ```
   rsort($frutas);
   // $frutas ahora es ["Pera", "Naranja", "Manzana"]
   ```

4. `array_push($array, $valor)`: Agrega uno o más elementos al final del array.

   ```
   array_push($frutas, "Kiwi");
   ```

5. `array_pop($array)`: Elimina el último elemento del array.

   ```
   $ultimaFruta = array_pop($frutas);
   ```

6. `array_key_exists($clave, $array)`: Verifica si una clave existe en un array.

   ```
   if (array_key_exists("Francia", $capitales)) {
     echo "La capital de Francia es " . $capitales["Fran
   cia"];
   }
   ```

7. `in_array($valor, $array)`: Verifica si un valor existe en un array.

```
if (in_array("Roma", $capitales)) {
  echo "Roma está en la lista de capitales.";
}
```

8. `array_merge($array1, $array2)`: Combina dos o más arrays.

```
$array1 = ["a", "b", "c"];
$array2 = ["d", "e"];
$resultado = array_merge($array1, $array2);
// $resultado es ["a", "b", "c", "d", "e"]
```

3.3 Funciones

Declaración de una Función

- **Sintaxis básica:**

```
function nombreFuncion($parametro1, $parametro2) {
  // Código a ejecutar
  return $resultado;
}
```

- **Ejemplo:**

```
function saludar($nombre) {
  return "Hola, " . $nombre . "!";
}

echo saludar("Juan"); // Resultado: "Hola, Juan!"
```

Paso por Valor vs. Referencia

- **Paso por Valor:** Se pasa una copia del valor.

```
function incrementar($valor) {
  $valor++;
}
```

```
$numero = 5;
incrementar($numero);
echo $numero; // Resultado: 5 (no cambia)
```

- **Paso por Referencia:** Se pasa la variable original, permitiendo modificar su valor.

```
function incrementar(&$valor) {
  $valor++;
}

$numero = 5;
incrementar($numero);
echo $numero; // Resultado: 6 (la variable fue modifi
cada)
```

Creación de Bibliotecas

- Una biblioteca es un archivo que contiene funciones reutilizables.

1. **Crear un archivo con funciones:**

```
// Archivo: misFunciones.php
<?php
function saludar($nombre) {
  return "Hola, " . $nombre;
}

function sumar($a, $b) {
  return $a + $b;
}
?>
```

2. **Incluir la biblioteca en otros scripts:**

```
// Archivo principal.php
<?php
include 'misFunciones.php';
```

```
echo saludar("Juan"); // Resultado: "Hola, Juan"
echo sumar(5, 3); // Resultado: 8
?>
```

Include vs Require

- `include`: Incluye y evalúa el archivo especificado. Si no puede incluir el archivo, emitirá un **warning** pero el script continuará.
- `require`: Igual que include, pero si no puede incluir el archivo, emitirá un **fatal error** y el script se detendrá.

3.4 Formularios

Envío de Datos con Formularios

- **HTML Formulario:**

```
<form method="POST" action="procesar.php">
  <label for="nombre">Nombre:</label>
  <input type="text" name="nombre" id="nombre">
  <input type="submit" value="Enviar">
</form>
```

Procesar Datos en PHP

- **Verificar el método de solicitud:**

```
<?php
if ($_SERVER["REQUEST_METHOD"] == "POST") {
  $nombre = $_POST['nombre'];
  echo "Hola, " . htmlspecialchars($nombre);
}
?>
```

- **Nota de Seguridad:** Es recomendable usar `htmlspecialchars()` para evitar ataques XSS al mostrar datos ingresados por el usuario.

Redirección de Páginas Web

- **Redireccionar usando `header()`:**

```
<?php
    header("Location: <http://www.ejemplo.com>");
    exit();
?>
```

- **Nota:** Después de usar `header("Location: ...")`, es buena práctica usar `exit();` para detener la ejecución del script.

3.5 Manejo de Archivos

Abrir y Escribir en un Archivo

- **Sintaxis para abrir un archivo:**

```
$archivo = fopen("datos.txt", "w");
```

- **Escribir en el archivo:**

```
if ($archivo) {
  fwrite($archivo, "Este es el contenido\\n");
  fclose($archivo);
  echo "Archivo escrito exitosamente.";
} else {
  echo "No se pudo abrir el archivo.";
}
```

Leer un Archivo

- **Abrir el archivo para lectura:**

```
$archivo = fopen("datos.txt", "r");
```

- **Leer el contenido:**

```
if ($archivo) {
    $tamano = filesize("datos.txt");
  $contenido = fread($archivo, $tamano);
  fclose($archivo);
  echo $contenido;
} else {
  echo "No se pudo abrir el archivo.";
}
```

Permisos de Manejo de Archivos

Modo	Descripción
`r`	Abre para solo lectura; coloca el puntero al inicio del archivo.
`r+`	Abre para lectura y escritura; coloca el puntero al inicio del archivo.
`w`	Abre para solo escritura; coloca el puntero al inicio y trunca el archivo a cero longitud. Si no existe, lo crea.
`w+`	Abre para lectura y escritura; coloca el puntero al inicio y trunca el archivo a cero longitud. Si no existe, lo crea.
`a`	Abre para solo escritura; coloca el puntero al final del archivo. Si no existe, lo crea.
`a+`	Abre para lectura y escritura; coloca el puntero al final del archivo. Si no existe, lo crea.
`x`	Crea y abre para solo escritura; coloca el puntero al inicio del archivo. Si el archivo existe, `fopen()` fallará.
`x+`	Crea y abre para lectura y escritura; coloca el puntero al inicio del archivo. Si el archivo existe, `fopen()` fallará.

- **Ejemplos:**

```
// Abre el archivo para escritura; crea el archivo si no existe
$archivo = fopen("nuevo.txt", "w");

// Abre el archivo para agregar contenido al final
$archivo = fopen("log.txt", "a");
```

```
// Intenta crear un archivo nuevo; falla si ya existe
$archivo = fopen("unique.txt", "x");
```

Notas Adicionales

- **Siempre cerrar los archivos:** Después de terminar de trabajar con un archivo, es importante cerrarlo usando `fclose($archivo);`.
- **Manejo de Errores:** Siempre verificar si `fopen()` devuelve un valor válido antes de intentar leer o escribir.
- **Funciones Útiles:**
  - `file_exists($filename)`: Verifica si un archivo existe.

    ```
    if (file_exists("datos.txt")) {
      echo "El archivo existe.";
    }
    ```

  - `unlink($filename)`: Elimina un archivo.

    ```
    unlink("archivo_a_eliminar.txt");
    ```

▼ POO

Conceptos Fundamentales de la POO

La **Programación Orientada a Objetos (POO)** es un paradigma de programación que utiliza "objetos" y sus interacciones para diseñar aplicaciones y programas informáticos. En la POO, los objetos encapsulan tanto los datos (atributos) como las funciones (métodos) que actúan sobre ellos.

Clases y Objetos

- **Clase**: Es una plantilla o modelo que define las propiedades (atributos) y comportamientos (métodos) comunes de un conjunto de objetos.

- **Objeto**: Es una instancia de una clase. Representa una entidad individual con su propio estado y comportamiento.

Ejemplo:

```
class Cliente {
    // Atributos y métodos de la clase Cliente
}

$cliente1 = new Cliente(); // Objeto de la clase Cliente
```

Encapsulación

La **encapsulación** consiste en ocultar los detalles internos de una clase y exponer solo lo necesario a través de métodos públicos. Protege los datos y asegura que solo puedan ser modificados de manera controlada.

- **Atributos privados**: Solo accesibles desde dentro de la clase.
- **Métodos públicos**: Interfaz para interactuar con el objeto.

Ejemplo:

```
class CuentaBancaria {
    private $saldo;

    public function depositar($cantidad) {
        $this->saldo += $cantidad;
    }

    public function obtenerSaldo() {
        return $this->saldo;
    }
}
```

Herencia

La **herencia** permite crear nuevas clases basadas en clases existentes, heredando sus atributos y métodos. Facilita la reutilización y extensión del código.

- **Clase padre (superclase)**: Clase de la cual se heredan características.

- **Clase hija (subclase)**: Clase que hereda de otra.

Ejemplo:

```
class Vehiculo {
    protected $color;

    public function mover() {
        echo "El vehículo se está moviendo";
    }
}

class Coche extends Vehiculo {
    private $marca;

    public function __construct($color, $marca) {
        $this->color = $color;
        $this->marca = $marca;
    }
}
```

Llamar a funciones de otras clases

Para llamar a a funciones de otras clases se estructura de esta manera `$objeto-> nombre_funcion()`

Ejemplo:

```
//cliente.php
class Cliente {
    private $nombre

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

    public function mostrarNombre(){
        echo $this->nombre;
    }
}
```

```
//miCliente.php
include("cliente.php");

$cliente1 = new Cliente("pepito"); // Objeto de la clase
Cliente
$cliente1->mostrarNombre;
```

Definición de Clases en PHP

Atributos y Métodos

- **Atributos**: Variables que representan las propiedades de un objeto.
- **Métodos**: Funciones que definen el comportamiento del objeto.

Sintaxis básica:

```
class NombreClase {
    // Atributos
    public $atributoPublico;
    protected $atributoProtegido;
    private $atributoPrivado;

    // Métodos
    public function metodoPublico() {
        // Código
    }

    protected function metodoProtegido() {
        // Código
    }

    private function metodoPrivado() {
        // Código
    }
}
```

Constructor y Destructor

- **Constructor (`__construct`)**: Método especial que se ejecuta al crear una instancia de la clase. Se utiliza para inicializar atributos.
- **Destructor (`__destruct`)**: Método que se ejecuta cuando el objeto es destruido o el script finaliza.

Ejemplo:

```
class Persona {
    private $nombre;
    private $apellido;

    public function __construct($nombre, $apellido) {
        $this->nombre = $nombre;
        $this->apellido = $apellido;
    }

    public function __destruct() {
        // Código para liberar recursos
    }

    public function obtenerNombreCompleto() {
        return $this->nombre . " " . $this->apellido;
    }
}
```

Modificadores de Acceso

- **public**: Accesible desde cualquier lugar.
- **protected**: Accesible desde la clase y sus subclases.
- **private**: Accesible solo desde la clase donde se define.

Ejemplo:

```
class Ejemplo {
    public $publico = "Visible desde cualquier lugar";
    protected $protegido = "Visible en la clase y subcla
ses";
```

```
    private $privado = "Visible solo en la clase";
}
```

Herencia en PHP

Uso de `extends`

Para heredar de otra clase, se utiliza la palabra clave `extends`.

Ejemplo:

```
class Empleado extends Persona {
    private $salario;

    public function __construct($nombre, $apellido, $sal
ario) {
        parent::__construct($nombre, $apellido); // Llam
a al constructor de la clase padre
        $this->salario = $salario;
    }

    public function obtenerSalario() {
        return $this->salario;
    }
}
```

Llamada a Métodos y Constructores de la Clase Padre

- `parent::metodo()`: Llama a un método de la clase padre.
- `parent::__construct()`: Llama al constructor de la clase padre.

Clases y Métodos Abstractos

- **Clases abstractas**: No pueden ser instanciadas. Se utilizan como base para otras clases.
- **Métodos abstractos**: Declarados pero no implementados en la clase abstracta; deben ser implementados en las subclases.

Ejemplo:

```
abstract class Figura {
    abstract public function calcularArea();
}

class Circulo extends Figura {
    private $radio;

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

    public function calcularArea() {
        return pi() * pow($this->radio, 2);
    }
}
```

Clases y Métodos Finales

- `final` en clases: Impide que la clase sea heredada.
- `final` en métodos: Impide que el método sea sobrescrito en subclases.

Ejemplo:

```
final class Base {
    final public function metodoImportante() {
        // Código
    }
}
```

Métodos y Atributos Estáticos

- **Estáticos (`static`)**: Pertenecen a la clase en sí, no a las instancias.
- Se accede a ellos utilizando `NombreClase::$atributo` o `self::$atributo` dentro de la clase.

Ejemplo:

```
class Contador {
    private static $contador = 0;

    public function __construct() {
        self::$contador++;
    }

    public static function obtenerContador() {
        return self::$contador;
    }
}

new Contador();
new Contador();

echo Contador::obtenerContador(); // Resultado: 2
```

Interfaces

- Definen métodos que deben ser implementados por las clases que las implementen.
- Los métodos de una interfaz son públicos y sin implementación.

Ejemplo:

```
interface Operaciones {
    public function sumar($a, $b);
    public function restar($a, $b);
}

class Calculadora implements Operaciones {
    public function sumar($a, $b) {
        return $a + $b;
    }

    public function restar($a, $b) {
```

```
        return $a - $b;
    }
}
```

Namespaces

Los **namespaces** permiten organizar el código y evitar conflictos de nombres entre clases, funciones y constantes.

Definición de un namespace:

```
namespace MiProyecto\\SubNivel;

class MiClase {
    // Código
}
```

Uso de una clase con namespace:

```
use MiProyecto\\SubNivel\\MiClase;

$obj = new MiClase();
```

Manejo de Errores y Excepciones

Uso de `try-catch-finally`

Permite manejar excepciones y errores de manera controlada.

Ejemplo:

```
try {
    // Código que puede lanzar una excepción
    if ($divisor == 0) {
        throw new Exception("División por cero.");
    }
    $resultado = $dividendo / $divisor;
} catch (Exception $e) {
    echo "Error: " . $e->getMessage();
```

```
} finally {
    echo "Operación finalizada.";
}
```

Creación de Excepciones Personalizadas

Ejemplo:

```
class MiExcepcion extends Exception {}

try {
    throw new MiExcepcion("Algo salió mal.");
} catch (MiExcepcion $e) {
    echo $e->getMessage();
}
```

Serialización de Objetos

La **serialización** convierte un objeto en una cadena para almacenarlo o transmitirlo.

Serializar:

```
$objeto = new MiClase();
$cadenaSerializada = serialize($objeto);
```

Deserializar:

```
$objetoRecuperado = unserialize($cadenaSerializada);
```

▼ PHP_MYSQL

Conectar a una Base de Datos

Conexión con `mysqli_connect()`

```
$conexion = @mysqli_connect('servidor', 'usuario', 'cont
raseña', 'base_datos');
```

```
if (!$conexion) {
    echo 'Error de conexión: ' . mysqli_connect_error();
} else {
    echo "conectado a la base de datos";
}
```

- **Servidor**: usualmente `localhost`.
- **Usuario** y **Contraseña**: datos de acceso.
- **Base de datos**: nombre de la base de datos opcional en la conexión.

Para cerrar la conexión:

```
mysqli_close($conexion);
```

Conexión Persistente

Añadir `p:` al nombre del servidor para una conexión persistente:

```
$conexion = @mysqli_connect('p:localhost', 'usuario', 'c
ontraseña', 'base_datos');
```

Seleccionar Base de Datos

Usar `mysqli_select_db()` si no se especifica la base de datos en la conexión:

```
mysqli_select_db($conexion, 'nombre_base_datos');
```

Acceder a Datos con `mysqli_query()`

Para realizar consultas:

```
$resultado = mysqli_query($conexion, 'SELECT * FROM tabl
a');
if (!$resultado) {
    echo 'Error en la consulta: ' . mysqli_error($conexi
on);
} else {
```

```
    /* codigo... */
}
```

Obtener Datos de una Consulta

- `mysqli_fetch_array($resultado)`: devuelve una fila como array (índice numérico y asociativo).

  ```
  $fila = mysqli_fetch_array($resultado);
  echo "ID: " . $fila[0] . " - Nombre: " . $fila['nombr
  e'];
  ```

- `mysqli_fetch_assoc($resultado)`: devuelve una fila como array asociativo.

  ```
  $fila = mysqli_fetch_assoc($resultado);
  echo "Nombre: " . $fila['nombre'];
  ```

- `mysqli_fetch_row($resultado)`: devuelve una fila como array numérico.

  ```
  $fila = mysqli_fetch_row($resultado);
  echo "ID: " . $fila[0];
  ```

- `mysqli_fetch_object($resultado)`: devuelve una fila como un objeto.

  ```
  $fila = mysqli_fetch_object($resultado);
  echo "Nombre: " . $fila->nombre;
  ```

Para obtener todas las filas:

```
$todas_filas = mysqli_fetch_all($resultado, MYSQLI_ASSO
C); // Array asociativo
```

Liberar Memoria de una Consulta

Es importante liberar la memoria de las consultas:

```
mysqli_free_result($resultado);
```

Información de la Consulta

- **Número de Filas**: `mysqli_num_rows($resultado)`.
- **Número de Columnas**: `mysqli_num_fields($resultado)`.

▼ FILTER_INPUT

Función `filter_input()`

La función `filter_input()` permite filtrar y validar datos recibidos de formularios HTML, evitando problemas como inyecciones de código o entradas mal formateadas. Es una medida de seguridad esencial cuando los datos los proporcionan los usuarios.

Sintaxis

```
$dato = filter_input(TIPO, "name_del_input", FILTER[, OPTIONS]);
```

- **TIPO**: Constante que indica de qué tipo de variable de entrada provienen los datos.
  - `INPUT_GET`: Datos de `$_GET`.
  - `INPUT_POST`: Datos de `$_POST`.
  - `INPUT_COOKIE`: Datos de `$_COOKIE`.
  - `INPUT_SERVER`: Datos de `$_SERVER`.
  - `INPUT_ENV`: Datos de `$_ENV`.
- **name_del_input**: Nombre del campo que se desea obtener del formulario.
- **FILTER**: Filtro a aplicar, por ejemplo `FILTER_SANITIZE_STRING` o `FILTER_VALIDATE_EMAIL`.
- **OPTIONS**: Opcional, se puede usar para configurar el filtro de manera personalizada.

El valor retornado por `filter_input()` será el valor filtrado o `FALSE` si el filtro falla, o `NULL` si la variable no está definida.

Ejemplo sin Filtrado

En este ejemplo, los datos del formulario no son filtrados antes de ser usados:

```
if (isset($_POST["entrada"])) {
    echo $_POST["entrada"];
}
```

Si el usuario introduce código HTML o JavaScript malicioso, este se ejecutará sin restricciones.

Ejemplo con `filter_input()`

Aquí se utiliza `filter_input()` para filtrar la entrada antes de mostrarla:

```
$entrada_filtrada = filter_input(INPUT_POST, "entrada",
FILTER_SANITIZE_STRING);
if ($entrada_filtrada !== NULL) {
    echo "<br>" . $entrada_filtrada . "<br>";
}
```

En este caso, si el usuario introduce código HTML o JavaScript, será sanitizado y no tendrá efecto.

Filtros Comunes

- `FILTER_SANITIZE_STRING`: Elimina etiquetas HTML.
- `FILTER_SANITIZE_EMAIL`: Elimina caracteres no permitidos en correos electrónicos.
- `FILTER_VALIDATE_EMAIL`: Valida si la entrada es un correo electrónico válido.
- `FILTER_SANITIZE_NUMBER_INT`: Elimina todo excepto dígitos y los signos `+` y `-`.
- `FILTER_VALIDATE_INT`: Valida si la entrada es un número entero.

Ejemplo Completo

Formulario HTML con filtrado de entrada:

```
<form method="post" action="<?php echo $_SERVER['PHP_SEL
F']; ?>">
```

```
    <input type="text" name="entrada">
    <input type="submit" name="enviar">
</form>
```

PHP con `filter_input()` para manejar la entrada:

```
$entrada = filter_input(INPUT_POST, 'entrada', FILTER_SA
NITIZE_STRING);
if ($entrada !== NULL) {
    echo "Entrada filtrada: " . $entrada;
} else {
    echo "No se ha recibido ninguna entrada o la entrada
no es válida.";
}
```

Buenas Prácticas

- **Filtrar siempre la entrada del usuario**: Para evitar problemas de seguridad, como la inyección de código.
- **Validar y Sanitizar**: Utiliza `filter_input()` tanto para sanitizar como para validar los datos según el tipo esperado.
- **Usar constantes adecuadas**: Asegúrate de usar la constante correcta (`INPUT_GET`, `INPUT_POST`, etc.) según la fuente de los datos.

▼ PDO

Conexión con PDO

Conexión básica:

```
try {
    $con = new PDO(
        'mysql:host=HOST;dbname=BD',
        $usuario,
        $password
    );
} catch (PDOException $e) {
```

```
    echo "Error: " . $e->getMessage();
}
```

- Si hay error, se lanza `PDOException`.
- Para cerrar la conexión:

```
$con = null;
```

en el host se pone la IP del servidor o su nombre, de normal siempre es <localhost>

Conexión persistente (caché de conexiones):

```
try {
    $con = new PDO(
        'mysql:host=HOST;dbname=BD',
        $usuario,
        $password,
        array(PDO::ATTR_PERSISTENT => true)
    );
} catch (PDOException $e) {
    echo "Error: " . $e->getMessage();
}
```

Consultas con PDO

SELECT (query):

```
$res = $con->query("SELECT * FROM tabla");
if ($res !== false) {
    // Tenemos un PDOStatement
}
```

INSERT/UPDATE/DELETE (exec):

```
$filas = $con->exec("UPDATE tabla SET campo='valor'");
if ($filas === false) {
    // Error en la consulta
} else {
    // $filas es el nº de filas afectadas
}
```

Consultas preparadas (prepare/execute):

- Recomendadas para consultas con parámetros y mayor seguridad.

```
$stmt = $con->prepare("SELECT * FROM tabla WHERE id = :i
d");
$stmt->execute(array(':id' => $valor_id));
$result = $stmt->fetch(PDO::FETCH_ASSOC);
```

Obteniendo Resultados

fetch(): recupera una fila por llamada.

Modos comunes:

- `PDO::FETCH_ASSOC`: array asociativo `['columna' => valor]`
- `PDO::FETCH_NUM`: array numérico `[0 => valor_col_1, 1 => valor_col_2, ...]`
- `PDO::FETCH_BOTH`: por defecto, array asociativo y numérico a la vez.
- `PDO::FETCH_OBJ`: devuelve un objeto con propiedades para cada columna.

Ejemplo:

```
$res = $con->query("SELECT id, nombre FROM usuarios");
while ($fila = $res->fetch(PDO::FETCH_ASSOC)) {
    echo $fila['nombre'];
}
```

fetchAll(): obtiene todas las filas a la vez en un array bidimensional.

```
$todas = $res->fetchAll(PDO::FETCH_ASSOC);
```

rowCount(): número de filas afectadas (en SELECT depende del driver, no siempre fiable).

```
$num_filas = $res->rowCount();
```

closeCursor(): libera el conjunto de resultados.

```
$res->closeCursor();
```

Recorrer Resultados Ejemplos

Usando fetch() en un bucle (array asociativo):

```
$res = $con->query("SELECT id, nombre, email FROM usuari
os");
while ($fila = $res->fetch(PDO::FETCH_ASSOC)) {
    echo "ID: " . $fila['id'] . "<br>";
    echo "Nombre: " . $fila['nombre'] . "<br>";
    echo "Email: " . $fila['email'] . "<br><br>";
}
```

Usando fetch() en un bucle (array numérico):

```
$res = $con->query("SELECT id, nombre, email FROM usuari
os");
while ($fila = $res->fetch(PDO::FETCH_NUM)) {
    echo "ID: " . $fila[0] . "<br>";     // 0 = id
    echo "Nombre: " . $fila[1] . "<br>"; // 1 = nombre
    echo "Email: " . $fila[2] . "<br><br>"; // 2 = email
}
```

Usando fetchAll() (array asociativo):

```
$res = $con->query("SELECT id, nombre, email FROM usuari
os");
$usuarios = $res->fetchAll(PDO::FETCH_ASSOC);
foreach ($usuarios as $usuario) {
    echo "ID: " . $usuario['id'] . "<br>";
```

```
    echo "Nombre: " . $usuario['nombre'] . "<br>";
    echo "Email: " . $usuario['email'] . "<br><br>";
}
```

Manejo de Errores con Excepciones

- Envolver las operaciones en `try-catch` para capturar `PDOException`.
- Métodos útiles: `getMessage()` y `getCode()`.

Ejemplo:

```
try {
    $con = new PDO('mysql:host=localhost;dbname=mi_bd',
'user', 'pass');
    // operaciones con la BD
} catch (PDOException $e) {
    echo "Error: " . $e->getMessage();
}
```

▼ SESIONES

Iniciar una Sesión

```
session_start();
```

- Debe llamarse antes de enviar cualquier salida al navegador (antes de `<html>`).
- Si el usuario ya tiene sesión, la reanuda. Si no, crea una nueva.
- Devuelve `true` si se ha podido crear o reanudar la sesión, `false` si no.

Variables de Sesión

- Se almacenan en `$_SESSION`, un array superglobal asociativo.
- Ejemplo:

```
$_SESSION["usuario"] = "Homer S.";
```

- Estas variables estarán disponibles en todas las páginas donde se llame `session_start()`.

Modificar y eliminar variables:

```
$_SESSION["usuario"] = "Lisa S."; // Modificar
unset($_SESSION["usuario"]); // Eliminar variable especí
fica
```

ID y Nombre de la Sesión

- `session_id()` devuelve el ID actual. Si se le pasa un valor, cambia el ID.
- `session_name()` devuelve el nombre de la variable que guarda el ID (por defecto `PHPSESSID`). Si se le pasa un valor, lo cambia.

Control de Cambios en la Sesión

- `session_abort()`: Deshace los cambios hechos en `$_SESSION` desde el `session_start()` actual. Cuando se recargue o cambie de página, no se verán los cambios.
- `session_reset()`: Restaura inmediatamente las variables de sesión a su estado original en esta misma ejecución.

Cierre y Destrucción de Sesiones

- `session_destroy()`: Destruye la sesión en el servidor.
  - Después de `session_destroy()`, conviene vaciar `$_SESSION` y, si se desea, eliminar la cookie:

    ```
    $_SESSION = array();
    setcookie(session_name(), '', time()-1, '/');
    ```

Estado de la Sesión

- `session_status()` devuelve:
  - `PHP_SESSION_DISABLED`: sesiones desactivadas
  - `PHP_SESSION_NONE`: activadas pero no hay sesión iniciada
  - `PHP_SESSION_ACTIVE`: hay una sesión activa

Transmisión del ID de Sesión

- Por defecto, PHP usa cookies para transmitir el ID.
- Si las cookies no están permitidas, se puede usar la URL (si `session.use_trans_sid` está activo).
- Por defecto, `session.use_only_cookies=1` en php.ini hace que sólo se usen cookies.
- Si se requiere pasar el SID manualmente en la URL (no recomendado), se puede usar la constante `SID`:

```
<a href="pagina2.php?<?= SID ?>">Siguiente página</a>
```

Ejemplo Básico

index.php

```
<?php
    session_start();
    $_SESSION["usuario"] = "Homer S.";
?>
<!DOCTYPE html>
<html>
    <head><meta charset="UTF-8"></head>
<body>
    <a href="bienvenida.php">Ir a Bienvenida</a>
</body>
</html>
```

bienvenida.php

```
session_start();
echo "Bienvenido " . $_SESSION["usuario"];
```

▼ REST

REST (Representational State Transfer)

Concepto y contexto:

- REST es un estilo de arquitectura para crear Servicios Web ligeros y flexibles.
- Se basa en el protocolo HTTP estándar, lo que lo hace sencillo de usar y entender.
- No depende de un formato de datos concreto; a menudo se usan JSON o XML. En el ejemplo dado, se usa JSON.

Principios básicos:

1. **Cliente/Servidor sin estado (Stateless):** Cada petición HTTP se procesa de forma independiente, sin necesidad de mantener estado entre una y otra. Esto facilita la escalabilidad, ya que el servidor no guarda información sobre el cliente entre peticiones.
2. **Operaciones CRUD mapeadas a métodos HTTP:**
   - GET: Obtener (Read) un recurso o lista de recursos.
   - POST: Crear (Create) un nuevo recurso.
   - PUT: Actualizar (Update) un recurso existente.
   - DELETE: Eliminar (Delete) un recurso existente.
     Estas operaciones se corresponden con la semántica ya establecida de HTTP.
3. **URI como identificador de recursos:**
   Cada recurso de la aplicación es identificable de manera única mediante una URI. Por ejemplo:
   `http://localhost/api/frutas` podría devolver la lista de frutas y `http://localhost/api/frutas/3` podría devolver información sobre la fruta con ID=3.

4. **Interfaz uniforme:**
   Todas las peticiones siguen el mismo patrón: un método HTTP, una URI para el recurso y, opcionalmente, un cuerpo de datos para enviar. Esto facilita la comprensión y la integración con otros sistemas.
5. **Sin estado y sin WSDL obligatorio:**
   A diferencia de SOAP, no existe un documento obligatorio como WSDL. Las operaciones suelen ser auto-descriptivas gracias a las URIs y los métodos HTTP. Si se necesita documentación, se hace de forma externa (swagger, readme, etc.).

Estructura de un Servicio REST en PHP

Para montar un servicio REST en PHP, se suele seguir una estructura como esta:

1. **Punto de entrada (index.php del servidor):**
   - Detectar el método HTTP de la petición:

     ```
     $metodo = $_SERVER['REQUEST_METHOD'];
     ```

   - Según el método, determinar la acción a realizar (ej. si es GET, realizar una consulta a la BD; si es POST, crear un nuevo registro, etc.).
   - Extraer los parámetros enviados:
     - GET/POST: Con `$_GET`, `$_POST` o `filter_input()`.
     - PUT/DELETE:

       Aquí se obtienen datos del cuerpo de la petición, ya que PUT y DELETE no tienen superglobales dedicadas como GET/POST.

       ```
       parse_str(file_get_contents("php://input"), $put_params);
       ```

   - Ejecutar la lógica de negocio (conexión a la BD, consultas, actualizaciones).

- Devolver la respuesta en formato JSON usando `json_encode()` para que el cliente pueda interpretarla fácilmente.

2. **Función de conexión a la Base de Datos (funcion_conexion_bd.php):**
   - Facilita una función genérica para conectar a la BD, ejecutar una sentencia SQL (SELECT, INSERT, UPDATE, DELETE) y devolver el resultado en un formato estándar (array o mensaje).
   - Esta separación de lógica permite un código más organizado.
3. **Gestión de rutas y recursos:**

   Aunque el ejemplo dado no muestra un enrutador complejo, normalmente es buena práctica usar un sistema que mapée URIs a acciones específicas. El ejemplo se basa en condiciones `if`/`switch` para determinar qué SQL ejecutar.

Ejemplo Práctico: Servidor REST de Frutería

Contexto del ejemplo:

- Queremos un servidor que gestione información de frutas en una base de datos.
- Podemos:
  - Consultar frutas por temporada (GET)
  - Consultar datos de una fruta concreta (GET)
  - Crear una nueva fruta (POST)
  - Actualizar el precio de una fruta existente (PUT)
  - Eliminar una fruta por ID (DELETE)

Servidor (fruteria_rest_servidor/index.php):

- Lee el método HTTP:

```
$metodo = $_SERVER['REQUEST_METHOD'];
```

- Según el método, obtiene los parámetros. Por ejemplo, si es GET con una temporada:

```
$temporada = filter_input(INPUT_GET, 'temporada', FIL
TER_SANITIZE_STRING);
$sql = "SELECT id, fruta FROM precios WHERE temporada
= '". strtoupper($temporada) ."'";
```

- Ejecuta la consulta a la BD con:

```
$con_bd = conexion_bd(SERVIDOR, USER, PASSWD, BASE_DA
TOS, $sql);
```

- Codifica el resultado en JSON:

```
echo json_encode($con_bd, TRUE);
```

- Para PUT (actualizar precio), se usaría:

```
parse_str(file_get_contents("php://input"), $put_para
ms);
$id = $put_params['id'];
$precio_kg = $put_params['precio_kg'];
$sql = "UPDATE precios SET precio_kg = '$precio_kg' W
HERE id = '$id'";
...
```

- Para POST (crear nueva fruta):

```
$temporada = $_POST['temporada'];
$fruta = $_POST['fruta'];
$precio_kg = $_POST['precio_kg'];
$sql = "INSERT INTO precios (temporada, fruta, precio
_kg) VALUES ('$temporada','$fruta','$precio_kg')";
...
```

- Para DELETE (eliminar):

```
$id = filter_input(INPUT_GET, 'id',FILTER_SANITIZE_NU
MBER_INT);
```

```
$sql = "DELETE FROM precios WHERE id = '$id'";
...
```

De esta forma, el servidor recibe peticiones, opera sobre la BD y devuelve siempre la respuesta en JSON.

Cliente REST en PHP (fruteria_rest_client)

Objetivo del cliente:

- Interactuar con el servidor REST.
- Realizar peticiones GET/POST/PUT/DELETE usando cURL.
- Mostrar los datos devueltos por el servidor (JSON) de forma clara.

Ficheros clave:

- `index.php` (cliente):

  Ofrece un formulario HTML para interactuar. Puedes introducir la temporada, fruta, ID, precio, etc. y pulsar botones que generan peticiones REST al servidor.

  Ejemplo de envío GET para consultar frutas por temporada:

  ```
  $url = URL . "index.php?temporada=" . $temporada;
  $response = curl_conexion($url, "GET");
  $frutas = json_decode($response, true);
  ```

  Aquí `$frutas` será un array con los resultados del servidor.

- `crear.php`:

  Permite crear una nueva fruta enviando un POST:

  ```
  $params = array('temporada' => 'INVIERNO', 'fruta' =>
  'MANZANA', 'precio_kg' => '2.50');
  $response = curl_conexion(URL . "index.php", 'POST',
  $params);
  ```

- `curl_conexion.php`:

  Contiene una función genérica `curl_conexion($url, $metodo, $params)` que:

- Inicializa cURL: `curl_init()`
- Configura la URL, el método (GET, POST, PUT, DELETE) con `CURLOPT_CUSTOMREQUEST`, y, si es necesario, los parámetros con `CURLOPT_POSTFIELDS`.
- Ejecuta la petición: `curl_exec()`
- Devuelve la respuesta del servidor, que luego el cliente interpreta (con `json_decode`).

Uso de cURL:

- Para un GET:

```
curl_setopt($curl, CURLOPT_URL, $url); // URL con par
ámetros GET en la misma URL
curl_setopt($curl, CURLOPT_CUSTOMREQUEST, "GET");
```

- Para un POST:

```
curl_setopt($curl, CURLOPT_CUSTOMREQUEST, "POST");
curl_setopt($curl, CURLOPT_POSTFIELDS, http_build_que
ry($params));
```

- Para PUT:

```
curl_setopt($curl, CURLOPT_CUSTOMREQUEST, "PUT");
curl_setopt($curl, CURLOPT_POSTFIELDS, http_build_que
ry($params));
```

- Para DELETE (similar al GET, pero con CUSTOMREQUEST="DELETE").

Buenas Prácticas y Consejos

1. **Separar la lógica de negocio:**

   Tener ficheros separados para la lógica de conexión a la BD (`funcion_conexion_bd.php`), el enrutamiento (index.php del servidor) y la interfaz del cliente (index.php del cliente) mejora la mantenibilidad.

2. **Validar y sanitizar datos:**

Antes de usar datos en sentencias SQL, se usan funciones como `filter_input()` para evitar inyecciones y datos no válidos.

3. **Devolver respuestas claras:**
   - Siempre usar `json_encode()` para la respuesta.
   - Incluir mensajes de error significativos.
4. **Documentar las URIs y métodos disponibles:**

   Aunque no se necesita WSDL, es útil tener un documento informando de cómo se consumen los recursos:
   - GET /index.php?temporada=XXX → lista de frutas para XXX
   - GET /index.php?tempo=XXX&fruta=YYY → datos de fruta YYY en temporada XXX
   - POST /index.php (temporada, fruta, precio_kg) → crear fruta
   - PUT /index.php (id, precio_kg) → actualizar precio
   - DELETE /index.php?id=ZZZ → eliminar fruta con id=ZZZ
5. **Testear las peticiones:**
   - Puedes usar herramientas externas (Postman, cURL en consola) para probar las peticiones al servidor antes de integrarlas con el cliente web.
