

UD 3. Bases de datos en PHP

Índice de contenidos

1. Acceso a bases de datos desde PHP.....	2
1.1. Nociones de programación orientada a objetos.....	2
1.1.1. Las clases.....	3
1.1.2. La variable \$this.....	4
1.1.3. Incluir la clase en nuestros scripts.....	4
1.1.4. Utilizar la clase.....	5
2. Utilización de bases de datos MySQL en PHP con PDO.....	7
2.1 Establecimiento de conexiones.....	8
2.2. Ejecución de consultas.....	9
2.3. Transacciones.....	10
2.4. Obtención y utilización de conjuntos de resultados.....	11
2.5. Consultas preparadas.....	12
3. Errores y manejo de excepciones.....	14
3.1 Excepciones.....	15

1. Acceso a bases de datos desde PHP

Una de las aplicaciones más frecuentes de PHP es generar un interfaz web para acceder y gestionar la información almacenada en una base de datos. Usando PHP podemos mostrar en una página web información extraída de la base de datos, o enviar sentencias al gestor de la base de datos para que elimine o actualice algunos registros.

PHP soporta más de 15 sistemas gestores de bases de datos: SQLite, Oracle, SQL Server, PostgreSQL, IBM DB2, MySQL/MariaDB, etc. Hasta la versión 5 de PHP, el acceso a las bases de datos se hacía principalmente utilizando **extensiones específicas** para cada sistema gestor de base de datos (extensiones nativas). Es decir, que si queríamos acceder a una base de datos de PostgreSQL, deberíamos instalar y utilizar la extensión de ese gestor en concreto. Las funciones y objetos a utilizar eran distintos para cada extensión.

A partir de la versión 5 de PHP se introdujo en el lenguaje una extensión para acceder de una forma común a distintos sistemas gestores: **PDO (PHP Data Object)**. La gran ventaja de PDO está clara: **podemos seguir utilizando una misma sintaxis aunque cambiemos el motor de nuestra base de datos**.

Usar las extensiones nativas puede tener ciertas ventajas, como acceso a funciones específicas del SGBD o mayor velocidad, pero en nuestro proyecto a un SGBD concreto. Es por esto que, desde el punto de vista del código limpio y los principios **SOLID**, es más conveniente usar PDO.

De los distintos SGBD existentes, vas a aprender a utilizar MySQL (o su fork MariaDB). MySQL es un gestor de bases de datos relacionales de código abierto bajo licencia GNU GPL. Es el gestor de bases de datos más empleado con el lenguaje PHP. Como ya vimos, es la letra "M" que figura en los acrónimos AMP y LAMP.

En esta unidad vas a ver cómo trabajar desde PHP con bases de datos MySQL utilizando PDO, aunque explicaremos también cómo hacerlo con la extensión nativa MySQLi.

Para el acceso a las funcionalidades deberás usar objetos. Aunque comenzaremos el tema tratando la orientación a objetos, para trabajar con la base de datos básicamente necesitas saber cómo crearlos y utilizarlos:

- En PHP se utiliza la palabra **new** para crear un nuevo objeto instanciando una clase:

```
$mi_coche = new Automovil();
```

- Para acceder a los miembros de un objeto, debes utilizar el operador flecha **->**:

```
$mi_coche->repostar();
```

1.1. Nociones de programación orientada a objetos

La programación orientada a objetos es una metodología de programación avanzada y bastante extendida, en la que los sistemas se modelan creando **clases**, que son un conjunto de datos y funcionalidades. Las clases son definiciones, a partir de las que se crean **objetos**. Los

objetos son ejemplares (**instancias**) de una clase determinada y como tal, disponen de los datos y funcionalidades definidos en la clase.

La programación orientada a objetos permite concebir las aplicaciones de una manera bastante intuitiva y cercana a la realidad. La tendencia es que un mayor número de lenguajes de programación adopten la programación orientada a objetos como paradigma para modelizar los sistemas. Prueba de ello es que desde su versión 5, PHP implanta la programación de objetos como metodología de desarrollo. Así pues, la programación orientada a objetos es un tema de gran interés, pues es muy utilizada y cada vez resulta más esencial para poder desarrollar en casi cualquier lenguaje moderno.

Aunque es un tema bastante amplio y en un principio difícil de asimilar, vamos a tratar de explicar la sintaxis básica de PHP para utilizar objetos, sin meternos en mucha teoría de programación orientada a objetos en general

1.1.1. Las clases

Una clase es un conjunto de variables, llamados **atributos**, y funciones, llamadas **métodos**, que trabajan sobre esas variables. Las clases son, al fin y al cabo, una definición: una especificación de propiedades y funcionalidades de elementos que van a participar en nuestros programas.

Por ejemplo, la clase Caja tendría como atributos características como las dimensiones, color, contenido y cosas semejantes. Las funciones o métodos que podríamos incorporar a la clase Caja son las funcionalidades que deseamos que realice la caja, como introduce(), muestra_contenido(), comprueba_si_cabe(), vaciate()...

Las clases en PHP se definen de la siguiente manera:

```
namespace DWES;

class Caja
{
    public $alto;
    public $ancho;
    public $largo;
    public $contenido;
    public $color;

    public function __construct()
    {
        $this->alto = 0;
        $this->ancho = 0;
        $this->largo = 0;
        $this->contenido = null;
        $this->color = 'negro';
    }

    public function introduce($cosa)
    {
        $this->contenido = $cosa;
    }

    public function muestra_contenido()
    {
        echo $this->contenido;
    }
}
```

En este ejemplo se ha creado la clase **Caja**, indicando como atributos el **ancho**, **alto** y **largo** de la caja, así como el **color** y el **contenido**. Se han creado, para empezar, un par de métodos, uno para **introducir** un elemento en la caja y otro para **mostrar** el contenido.

Si nos fijamos, los atributos se definen declarando unas variables al principio de la clase. Los métodos se definen declarando funciones dentro de la clase. La variable `$this`, utilizada dentro de los métodos la explicaremos un poco más abajo.

La instrucción `namespace` sirve para indicar el espacio de nombres al que pertenece la clase, para evitar conflictos entre paquetes que contengan clases con el mismo nombre.

1.1.2. La variable `$this`

Dentro de un método, la variable `$this` hace referencia al objeto sobre el que invocamos el método. En la invocación `$micaja->introduce("algo")` se está llamando al método `introduce` sobre el objeto `$micaja`. En ese caso `$this->contenido` hace referencia al atributo `contenido` del objeto `$micaja`, que es sobre el que se invocaba el método.

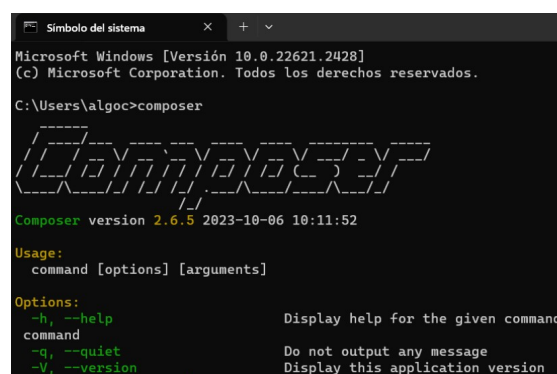
1.1.3. Incluir la clase en nuestros scripts

Para utilizar la clase en otro archivo la solución tradicional consiste en incluir el código con `include` o `require`. Sin embargo, esto es complejo de mantener cuando el proyecto escala.

Actualmente se utilizan métodos de autocarga en los que sólo incluimos un archivo y las clases se cargan automáticamente, como [spl_autoload_register\(\)](#) o [autoload de composer](#).

Para utilizar `autoload` de `composer` primero debemos instalar **composer**. Lo descargamos desde su web oficial <https://getcomposer.org/download/> (Composer-Setup.exe para Windows).

Si abrimos un cmd en Windows y escribimos `composer`, podemos comprobar que se ha instalado:



```
Simbolo del sistema
Microsoft Windows [Versión 10.0.22621.2428]
(c) Microsoft Corporation. Todos los derechos reservados.

C:\Users\algoc>composer

Composer version 2.6.5 2023-10-06 10:11:52

Usage:
  command [options] [arguments]

Options:
  -h, --help                Display help for the given command
  -q, --quiet               Do not output any message
  -V, --version              Display this application version
```

Vamos a crear un proyecto de ejemplo para probar la autocarga. Para ello creamos un directorio que aloje el proyecto (lo podemos llamar **autocarga**) y, dentro de este, inicializamos composer. Abrimos terminal en el directorio que hemos creado y ejecutamos:

```
composer init
```

Esto nos abrirá un asistente para configurar nuestro proyecto de composer. Al final, generará un archivo llamado **composer.json**, que es el archivo dónde se almacena la configuración de nuestro proyecto.

Creamos un directorio llamado "clases" y dentro colocamos el archivo **Caja.php**. Modificamos **composer.json** para que incluya este directorio cuando usemos el namespace DWES:

```
{
    "name": "NOMBRE_VENDOR/NOMBRE_PAQUETE",
    "description": "DESCRIPCIÓN DEL PROYECTO",
    "authors": [
        {
            "name": "NOMBRE_AUTOR",
            "email": "EMAIL_AUTOR"
        }
    ],
    "require": {},
    "autoload": {
        "psr-4": {
            "DWES\\": "clases/"
        }
    }
}
```

Una vez que tenemos la sección autoload creada, ejecutamos:

```
composer dump-autoload
```

Esto crea un directorio llamado **vendor**, que contendrá un archivo llamado **autoload.php**.

- El directorio **vendor** es una carpeta donde se instalan las librerías externas que necesitemos. En temas posteriores veremos cómo utilizarlo.
- El archivo **autoload.php** es el que debemos incluir en nuestras páginas y este se encargará de poner a nuestra disposición todas las clases que necesitemos.

Una vez hecho esto, para cargar las clases sólo deberemos incluir la siguiente línea:

```
require __DIR__ . '/vendor/autoload.php';
```

1.1.4. Utilizar la clase

Las clases solamente son definiciones. Si queremos utilizar la clase tenemos que crear un ejemplar de dicha clase, lo que corrientemente se le llama instanciar un objeto de una clase.

Si hemos incluido el archivo **Caja.php** se instancia directamente:

```
$micaja = new Caja();
```

Si hemos hecho autocarga, deberemos especificar el namespace cada vez que utilicemos una clase:

```
$micaja = new DWES\Caja();
```

O podemos añadir una línea `use` para utilizar un namespace por defecto:

```
use DWES\Caja;  
  
$micaja = new Caja();
```

Con esto hemos creado, o mejor dicho, instanciado, un objeto de la clase `Caja` llamado `$micaja`.

```
$micaja->introduce("algo");  
$micaja->muestra_contenido();
```

Con estas dos sentencias estamos introduciendo "algo" en la caja y luego estamos mostrando ese contenido en el texto de la página. Nos fijamos que los métodos de un objeto se llaman utilizando el código `->`.

```
$nombre_del_objeto->nombre_de_metodo();
```

Para acceder a los atributos de una clase también se accede con el código `->`. De esta forma:

```
$nombre_del_objeto->nombre_del_atributo;
```

Este sería un ejemplo de uso completo utilizando autoload de composer:

```
//autoload de composer  
require __DIR__ . '/vendor/autoload.php';  
  
use DWES\Caja;  
  
$caja = new Caja();  
$caja->introduce("algo");  
echo "La caja contiene un ";  
$caja->muestra_contenido();
```

Actividad 3.1

*Crea una clase **Rectángulo** con dos atributos (**ancho** y **alto**) y un método que devuelva el área del rectángulo. En su constructor, deberá tener como parámetros el ancho y el alto. Luego, pruébalo creando un objeto **Rectángulo**.*

2. Utilización de bases de datos MySQL en PHP con PDO

Como ya viste, existen dos formas de comunicarse con una base de datos desde PHP: utilizar una extensión nativa programada para un SGBD concreto, o utilizar una extensión que soporte varios tipos de bases de datos. Tradicionalmente las conexiones se establecían utilizando la extensión nativa `mysql`. Esta extensión se mantiene en la actualidad para dar soporte a las aplicaciones ya existentes que la utilizan, pero no se recomienda utilizarla para desarrollar nuevos programas. Lo más habitual es elegir entre `MySQLi` (extensión nativa) y `PDO`.

Con cualquiera de ambas extensiones, podrás realizar acciones sobre las bases de datos como:

- Establecer conexiones.
- Ejecutar sentencias SQL.
- Obtener los registros afectados o devueltos por una sentencia SQL.
- Emplear transacciones.
- Ejecutar procedimientos almacenados.
- Gestionar los errores que se produzcan durante la conexión o en el establecimiento de la misma.

Si vas a programar una aplicación que utilice como sistema gestor de bases de datos MySQL, la extensión `MySQLi` es una buena opción. Ofrece acceso a todas las características del motor de base de datos, a la vez que reduce los tiempos de espera en la ejecución de sentencias.

Sin embargo, si en el futuro tienes que cambiar el SGBD por otro distinto, tendrás que volver a programar gran parte del código de la misma. Por eso, de cara a la modularidad y la mantenibilidad del proyecto en un futuro, es interesante adoptar una capa de abstracción para el acceso a los datos. Existen varias alternativas como `ODBC`, pero sin duda la opción más recomendable en la actualidad es **PDO**.

El objetivo es que la aplicación sea independiente del SGBD utilizado, de forma que si llegado el momento necesitas cambiar el servidor de base de datos, las modificaciones que debas realizar en tu código sean mínimas. Por otro lado, esto también permite también que la misma aplicación pueda funcionar con un SGBD u otro según las necesidades del proyecto (si por ejemplo se usa un servidor poco potente, se puede usar un SGBD ligero como `SQLite`). `PDO` no abstrae de forma completa el sistema gestor que se utiliza.

Para saber más ...

La extensión `PDO` debe utilizar un driver o controlador específico para el tipo de base de datos que se utilice. Para consultar los controladores disponibles en tu instalación de PHP, puedes utilizar la información que proporciona la función `phpinfo()`.

PDO

PDO support	enabled
PDO drivers	mysql, sqlite

2.1 Establecimiento de conexiones

Para establecer una conexión con una base de datos utilizando PDO, debes instanciar un objeto de la clase PDO, el cual utilizarás a lo largo del script para hacer las consultas. Para instanciarlo hay que pasarle los siguientes parámetros (solo el primero es obligatorio):

- **Origen de datos (DSN).** Es una cadena de texto que indica qué controlador se va a utilizar y a continuación, separadas por el carácter dos puntos, los parámetros específicos necesarios por el controlador, como por ejemplo el nombre o dirección IP del servidor y el nombre de la base de datos.
- **Nombre de usuario** con permisos para establecer la conexión.
- **Contraseña** del usuario.
- **Opciones de conexión**, almacenadas en forma de **array**.

Por ejemplo, podemos establecer una conexión con la base de datos 'dwes' creada con anterioridad de la siguiente forma:

```
$bd = new PDO('mysql:host=localhost;dbname=dwes', 'dwes', 'abc123');
```

Si como en el ejemplo, se utiliza el controlador para MySQL, los principales parámetros para utilizar en la cadena DSN (separadas unas de otras por el carácter punto y coma) a continuación del prefijo mysql: son los siguientes:

- **host.** Nombre o dirección IP del servidor.
- **port.** Número de puerto TCP en el que escucha el servidor.
- **dbname.** Nombre de la base de datos.

Si quisieras indicar al servidor MySQL utilice codificación UTF-8 para los datos que se transmitan, puedes usar el parámetro **charset**:

```
$bd = new PDO('mysql:host=localhost;dbname=dwes;charset=utf8', 'dwes', 'abc123');
```

También lo puedes conseguir usando una opción específica de la conexión:

```
$opciones = array(PDO::MYSQL_ATTR_INIT_COMMAND => "SET NAMES utf8");  
$bd = new PDO('mysql:host=localhost;dbname=dwes', 'dwes', 'abc123', $opciones);
```

Para saber más ...

En el [manual de PHP](#) puedes consultar más información sobre los controladores existentes, los parámetros de las cadenas DSN y las opciones de conexión particulares de cada uno.

Una vez establecida la conexión, puedes utilizar el método `getAttribute` para obtener información del estado de la conexión y `setAttribute` para modificar algunos parámetros que afectan a la misma.

Por ejemplo, para obtener la versión del servidor puedes hacer:

```
$version = $bd->getAttribute(PDO::ATTR_SERVER_VERSION);  
echo "Versión: $version";
```

Y si quieres por ejemplo que te devuelva todos los nombres de columnas en mayúsculas:

```
$version = $bd->setAttribute(PDO::ATTR_CASE, PDO::CASE_UPPER);
```

Para saber más ...

En el manual de PHP, las páginas de las funciones [getAttribute](#) y [setAttribute](#) te permiten consultar los posibles parámetros que se aplican a cada una.

Al usar PDO **no hay que preocuparse por cerrar la conexión**, el propio objeto se encarga de ello mediante su destructor.

Actividad 3.2

Crea un script que se conecte a tu base de datos local e imprima la versión.

2.2. Ejecución de consultas

Para ejecutar una consulta SQL utilizando PDO, debes diferenciar aquellas sentencias SQL que no devuelven como resultado un conjunto de datos, de aquellas otras que sí lo devuelven.

En el caso de las consultas de acción, como **INSERT**, **DELETE** o **UPDATE**, el método `exec` ejecuta la consulta y devuelve el número de registros afectados.

```
$numRegistros = $bd->exec('DELETE FROM stock WHERE unidades=0');  
echo "<p>Se han borrado $numRegistros registros.</p>";
```

Si la consulta genera un conjunto de datos, como es el caso de **SELECT**, debes utilizar el método `query`, que devuelve un objeto de la clase `PDOStatement` que almacena los resultados.

```
$bd = new PDO("mysql:host=localhost;dbname=dwes", "dwes", "abc123");  
$resultado = $bd->query("SELECT producto, unidades FROM stock");
```

Actividad 3.3

Importa la base de datos **dwes.sql** que se encuentra en Moodle.

Crea una nueva cuenta de usuario llamada '**dwes**' con clave '**abc123**'.

Inserta en la tabla **stock** de la base de datos **dwes**, un producto con nombre "Ratón", tienda 3 y 2 unidades.

Realiza las operaciones SQL de forma directa, sin formularios ni entrada de dato alguna por parte del usuario.

Actividad 3.4

Crea un formulario con 3 campos (nombre producto, tienda y unidades). Cuando se pulse el botón de Enviar del formulario, se recogerán los datos introducidos del formulario y se creará un nuevo producto en la base de datos con esos datos.

2.3. Transacciones

Por defecto PDO trabaja en modo "**autocommit**", esto es, confirma de forma automática cada sentencia que ejecuta el servidor. Para trabajar con transacciones, PDO incorpora tres métodos:

- **beginTransaction**. Deshabilita el modo "autocommit" y comienza una nueva transacción, que finalizará cuando ejecutes uno de los dos métodos siguientes.
- **commit**. Confirma la transacción actual.
- **rollback**. Revierte los cambios llevados a cabo en la transacción actual.

Una vez ejecutado un **commit** o un **rollback**, se volverá al modo de confirmación automática.

```
$ok = true;
$bd->beginTransaction();
if ($bd->exec('DELETE ...') == 0) {
    $ok = false;
}

if ($bd->exec('UPDATE ...') == 0) {
    $ok = false;
}

if ($ok) {
    $bd->commit(); // Si todo fue bien confirma los cambios
} else {
    $bd->rollback(); // y si no, los revierte
}
```

Ten en cuenta que no todos los motores no soportan transacciones. Tal es el caso del motor MyISAM de MySQL. En este caso concreto, PDO ejecutará el método **beginTransaction** sin errores, pero naturalmente no será capaz de revertir los cambios si fuera necesario ejecutar un **rollback**.

2.4. Obtención y utilización de conjuntos de resultados

En PDO tienes varias posibilidades para tratar con el conjunto de registros devuelto por el método `query`. La más utilizada es el método `fetch` de la clase `PDOStatement`. Este método devuelve un registro del conjunto de resultados, o `false` si ya no quedan registros por recorrer.

```
$bd = new PDO("mysql:host=localhost;dbname=dwes", "dwes", "abc123");
$resultado = $bd->query("SELECT producto, unidades FROM stock");
while ($registro = $resultado->fetch()) {
    echo "Producto " . $registro['producto'] . ": " .
    $registro['unidades'] . "<br />";
}
```

El método **fetch()** devuelve un array compuesto por: (1) los nombres asociativos de las columnas de la SELECT y (2) índices numéricos, al mismo tiempo. Esto se debe a que el valor por defecto es, cuando el parámetro no se indica, **PDO::FETCH_BOTH**. En el caso de solo desear los nombres asociativos sería **fetch(PDO::FETCH_ASSOC)** y para solo numéricos tendríamos que especificar **FETCH_NUM**.

A partir de esto, el siguiente código generará error, ¿ves el motivo?

```
while ($registro = $resultado->fetch(PDO::FETCH_NUM)) {
    echo "Producto " . $registro['producto'] . ": " . $registro['unidades'] .
    "<br />";
}
```

Se ha indicado que el array sea exclusivamente numérico y dentro del `while()` se intenta usar un nombre asociativo como índice del array, que provoca el famoso **Notice: Undefined index: ...**

Por otro lado, en ocasiones puede hacernos falta saber cuántos registros se han obtenido. Para eso podemos utilizar el método `rowCount()` del resultado:

```
$bd = new PDO("mysql:host=localhost;dbname=dwes", "dwes", "abc123");
$resultado = $bd->query("SELECT * FROM producto");
echo 'Se han obtenido ' . $resultado->rowCount() . ' productos';
```

Para saber más ...

Utilizar `rowCount()` puede no funcionar en todos los SGBD. En la [documentación](#) recomiendan hacer una consulta sql con **SELECT COUNT(*)** y las mismas condiciones en la cláusula **WHERE** para obtener el número de filas.

Por otro lado, también podemos obtener un array con todos los resultados, en lugar de recorrerlos uno a uno. Para ello usamos el método `fetchAll()`:

```
$bd = new PDO("mysql:host=localhost;dbname=dwes", "dwes", "abc123");
$resultado = $bd->query("SELECT producto, unidades FROM stock");
$registros = $resultado->fetchAll();
foreach ($registros as $registro) {
    echo "Producto ".$registro['producto'].":  

    ".$registro['unidades']."<br />";
}
```

En este caso, para saber cuántos resultados se han obtenido sólo tendríamos que comprobar el tamaño del array con la función `count()`:

Actividad 3.5

Crea una página que reciba por GET el código de un producto y muestre la información sobre él, además del stock que hay de ese producto en cada tienda.

2.5. Consultas preparadas

Cada vez que se envía una consulta al servidor, éste debe analizarla antes de ejecutarla. Algunas sentencias SQL, como las que insertan valores en una tabla, deben repetirse de forma habitual en una aplicación. Para acelerar este proceso, MySQL admite consultas preparadas.

Estas consultas se almacenan en el servidor listas para ser ejecutadas cuando sea necesario. Se mejora la eficiencia, ya que el análisis de la sentencia se lleva a cabo sólo una vez. Además, reciben los parámetros por referencia, por lo que también son más resistentes a la inyección SQL.

Para saber más ...

La **inyección de SQL** es un tipo de ataque informático que consiste en introducir código SQL en lugar de un dato que vaya a ser utilizado en una consulta, de forma que el programa ejecute dicho código.

Supongamos que estamos en una página con formularios en la que tenemos la siguiente línea de código:

```
$consulta = "SELECT * FROM usuarios WHERE clave = " . $_POST['clave'];
```

Si el usuario introduce en el campo de la clave el texto `2508`, la consulta quedaría de la siguiente manera:

```
SELECT * FROM usuarios WHERE clave = 2508
```

Sin embargo, si el usuario introduce en el campo de la clave el texto `0 OR 2 > 1`, la consulta quedaría de la siguiente manera, mostrando los datos de todos los usuarios:

```
SELECT * FROM usuarios WHERE clave = 0 OR 2 > 1
```

Para preparar la consulta en el servidor MySQL, deberás utilizar el método **prepare** de la clase PDO. Este método devuelve un objeto de la clase PDOStatement.

Los parámetros se pueden marcar utilizando signos de interrogación. Un ejemplo:

```
$consulta = $bd->prepare('INSERT INTO familia (cod, nombre) VALUES (?, ?)');
```

¡Atención!

No es necesario utilizar comillas con los parámetros, aunque el dato que representen sea de tipo cadena.

Los valores de los parámetros se introducen al ejecutar la consulta:

```
$consulta = $bd->prepare('INSERT INTO familia (cod, nombre) VALUES (?, ?)');  
$consulta->execute(["TABLET", "Tablet PC"]);
```

En las consultas de tipo **SELECT**, los resultados se extraen de la propia consulta:

```
$consulta = $bd->prepare("SELECT nombre FROM familia WHERE cod = ?");  
$consulta->execute(["TABLET"]);  
while ($registro = $consulta->fetch()) {  
    echo "Nombre de la familia: ".$registro['nombre']."<br />";  
}
```

Para saber más ...

Puedes consultar la información sobre la utilización en PDO de consultas preparadas y la clase PDOStatement en el [manual de PHP](#).

Actividad 3.6

Modifica el ejercicio 3.5 para que utilice consultas preparadas.

3. Errores y manejo de excepciones

A buen seguro que, conforme has ido resolviendo ejercicios o simplemente probando código, te has encontrado con errores de programación. Algunos son reconocidos por el entorno de desarrollo, y puedes corregirlos antes de ejecutar. Otros aparecen en el navegador en forma de mensaje de error al ejecutar el guión.

PHP define una clasificación de los errores que se pueden producir en la ejecución de un programa y ofrece métodos para ajustar el tratamiento de los mismos. Para hacer referencia a cada uno de los niveles de error, PHP define una serie de constantes. Cada nivel se identifica por una constante. Por ejemplo, la constante `E_NOTICE` hace referencia a avisos que pueden indicar un error al ejecutar el guión, y la constante `E_ERROR` engloba errores fatales que provocan que se interrumpa forzosamente la ejecución.

Para saber más ...

La lista completa de constantes la puedes consultar en el [manual de PHP](#), donde también se describe el tipo de errores que representa.

La configuración inicial de cómo se va a tratar cada error según su nivel se realiza en `php.ini` el fichero de configuración de PHP. Entre los principales parámetros que puedes ajustar están:

- **error_reporting:** Indica qué tipos de errores se notificarán. Su valor se forma utilizando los operadores a nivel de bit para combinar las constantes anteriores. Su valor predeterminado es `E_ALL & ~E_NOTICE` que indica que se notifiquen todos los errores (`E_ALL`) salvo los avisos en tiempo de ejecución (`E_NOTICE`).
- **display_errors_** En su valor por defecto (On), hace que los mensajes se envíen a la salida estándar (y por lo tanto se muestren en el navegador). Se debe desactivar (Off) en los servidores que no se usan para desarrollo sino para producción.

Para saber más ...

Existen otros [parámetros](#) que podemos utilizar en `php.ini` para ajustar el comportamiento de PHP cuando se produce un error.

Desde código, puedes usar la función `error_reporting` con las constantes anteriores para establecer el nivel de notificación en un momento determinado. Por ejemplo, si en algún lugar de tu código figura una división en la que exista la posibilidad de que el divisor sea cero, cuando esto ocurra obtendrás un mensaje de error en el navegador. Para evitarlo, puedes

desactivar la notificación de errores de nivel `E_WARNING` antes de la división y restaurarla a su valor normal a continuación:

```
error_reporting(E_ALL & ~E_NOTICE & ~E_WARNING);
$resultado = $dividendo / $divisor;
error_reporting(E_ALL & ~E_NOTICE);
```

Para saber más

Para obtener más control sobre el proceso existe también la posibilidad de reemplazar la gestión de los mismos por la que tú definas. Es decir, puedes programar una función para que sea la que ejecuta PHP cada vez que se produce un error. El nombre de esa función se indica utilizando `set_error_handler` y debe tener como mínimo dos parámetros obligatorios (el nivel del error y el mensaje descriptivo) y hasta otros tres opcionales con información adicional sobre el error (el nombre del fichero en que se produce, el número de línea, y un volcado del estado de las variables en ese momento).

```
function miGestorDeErrores($nivel, $mensaje)
{
    switch($nivel) {
        case E_WARNING:
            echo "Error de tipo WARNING: $mensaje.<br />";
            break;
        default:
            echo "Error de tipo no especificado: $mensaje.<br />";
    }
}

set_error_handler("miGestorDeErrores");
$resultado = 10 / 0; // Imprime el mensaje del WARNING
restore_error_handler();
```

La función `restore_error_handler` restaura el manejador de errores original de PHP (más concretamente, el que se estaba usando antes de la llamada a `set_error_handler`).

3.1 Excepciones

A partir de la versión 5 se introdujo en PHP un modelo de excepciones similar al existente en otros lenguajes de programación:

- El código susceptible de producir algún error se introduce en un bloque `try`.
- Cuando se produce algún error, se lanza una excepción utilizando la instrucción `throw`.
- Después del bloque `try` debe haber como mínimo un bloque `catch` encargado de procesar el error.
- Si una vez acabado el bloque `try` no se ha lanzado ninguna excepción, se continúa con la ejecución en la línea siguiente al bloque o bloques `catch`.

Por ejemplo, para lanzar una excepción cuando se produce una división por cero podrías hacer:

```
try {
    if ($divisor == 0)
        throw new Exception("División por cero.");
    $resultado = $dividendo / $divisor;
}
catch (Exception $e) {
    echo "Se ha producido el siguiente error: ".$e->getMessage();
}
```

PHP ofrece una clase base `Exception` para utilizar como manejador de excepciones. Para lanzar una excepción no es necesario indicar ningún parámetro, aunque de forma opcional se puede pasar un mensaje de error (como en el ejemplo anterior) y también un código de error.

Entre los métodos que puedes usar con los objetos de la clase `Exception` están:

- **`getMessage`**: Devuelve el mensaje, en caso de que se haya puesto alguno.
- **`getCode`**: Devuelve el código de error si existe.

Las funciones internas de PHP y muchas extensiones como MySQLi usan el sistema de errores visto anteriormente. Solo las extensiones más modernas orientadas a objetos, como es el caso de PDO, utilizan este modelo de excepciones. En este caso, lo más común es que la extensión defina sus propios manejadores de errores heredando de la clase `Exception`.

Para saber más ...

Utilizando la [clase `ErrorException`](#) es posible traducir los errores a excepciones.

Concretamente, la clase PDO permite definir la fórmula que usará cuando se produzca un error, utilizando el atributo `PDO::ATTR_ERRMODE`. Las posibilidades son:

- **`PDO::ERRMODE_SILENT`**. No se hace nada cuando ocurre un error. Es el comportamiento por defecto.
- **`PDO::ERRMODE_WARNING`**. Genera un error de tipo `E_WARNING` cuando se produce un error.
- **`PDO::ERRMODE_EXCEPTION`**. Cuando se produce un error lanza una excepción utilizando el manejador propio `PDOException`.

Es decir, que si quieres utilizar excepciones con la extensión PDO, debes configurar la conexión haciendo:

```
$bd->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
```

Por ejemplo, el siguiente código:

```
$bd = new PDO("mysql:host=localhost; dbname=dwes", "dwes", "abc123");
$bd->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
try {
    $sql = "SELECT * FROM stox";
    $result = $bd->query($sql);
}
```



```
}  
catch (PDOException $p) {  
    echo "Error ".$p->getMessage()."<br />";  
}
```

Captura la excepción que lanza PDO debido a que la tabla no existe. El bloque catch muestra el siguiente mensaje:

```
Error SQLSTATE[42S02]: Base table or view not found: 1146 Table  
'dwes.stox' doesn't exist
```

Ejercicio 3.7

Haz un pequeño ejemplo en el que utilices control de excepciones al crear la conexión a la base de datos con PDO. Comprueba que si hay algún error, este se gestiona con el código creado.