Introduction to PDO

Objectives

- Introduction to PDO
- PreparedStatement
- Exception in PDO

What is PDO

- PHP 5 and later can work with a MySQL database using:
  - **MySQLi extension** (the "i" stands for improved)
  - **PDO (PHP Data Objects)**
- Earlier versions of PHP used the MySQL extension. However, this extension was deprecated in 2012.
- PDO is a PHP extension to formalize PHP's database connections by creating a uniform interface. This allows developers to create code which is portable across many databases and platforms

MySQLi or PDO?

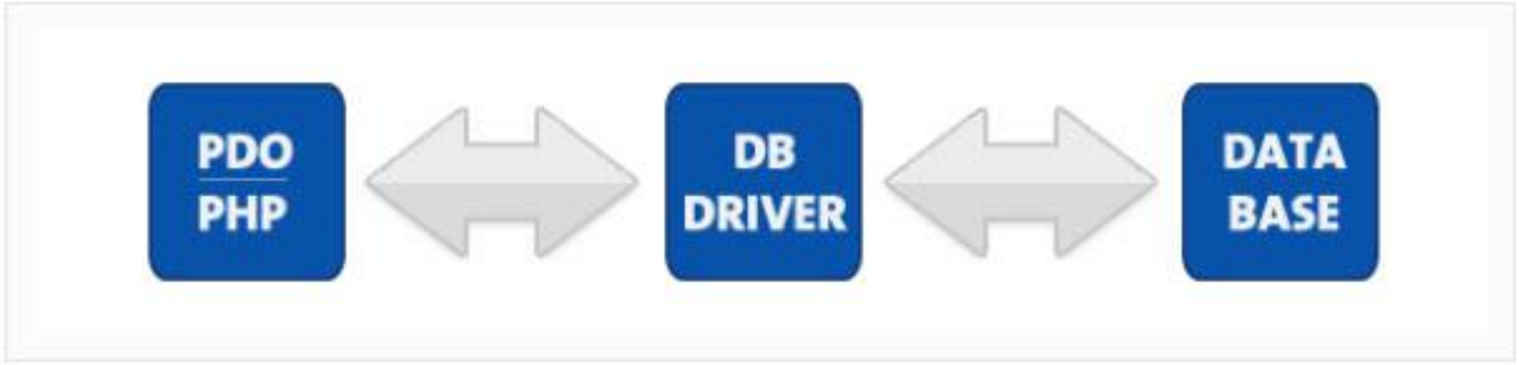
- Both MySQLi and PDO have their advantages:
  - PDO will work on 12 different database systems, whereas MySQLi will only work with MySQL databases.
  - So, if you have to switch your project to use another database, PDO makes the process easy. You only have to change the connection string and a few queries. With MySQLi, you will need to rewrite the entire code - queries included.
- Both are object-oriented, but MySQLi also offers a procedural API.
- Both support Prepared Statements. Prepared Statements protect from SQL injection, and are very important for web application security.

Introduction to PDO

- The *PHP Data Objects* (PDO) extension defines a lightweight, consistent interface for accessing databases in PHP. Each database driver that implements the PDO interface can expose database-specific features as regular extension functions.
- Note that you cannot perform any database functions using the PDO extension by itself; you must use a database-specific PDO driver to access a database server.
- PDO provides a *data-access* abstraction layer, which means that, regardless of which database you're using, you use the same functions to issue queries and fetch data.
- PDO does *not* provide a *database* abstraction; it doesn't rewrite SQL or emulate missing features. You should use a full-blown abstraction layer if you need that facility.

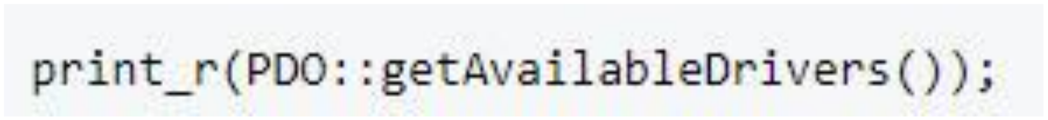
Introduction to OOP

- PDO ships with PHP 5.1, and is available as a PECL extension for PHP 5.0; PDO requires the new OO features in the core of PHP 5, and so will not run with earlier versions of PHP.



Database Support

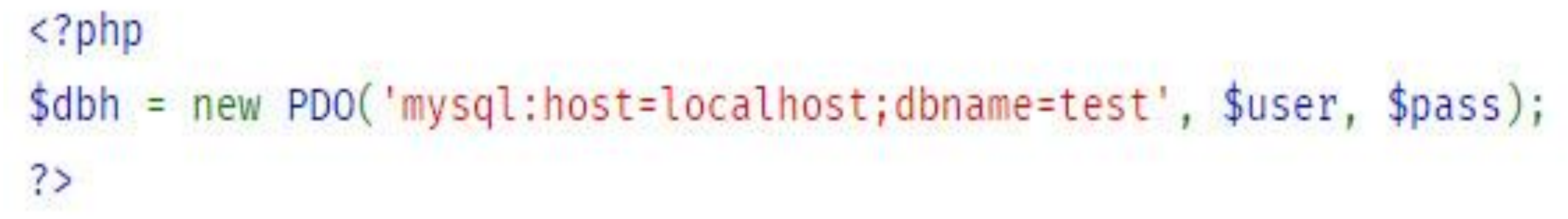
- The extension can support any database that a PDO driver has been written for. The following database drivers are available:
  - PDO_DBLIB (FreeTDS / Microsoft SQL Server / Sybase)
  - PDO_FIREBIRD (Firebird/Interbase 6)
  - PDO_IBM (IBM DB2)
  - PDO_INFORMIX (IBM Informix Dynamic Server)
  - PDO_MYSQL (MySQL 3.x/4.x/5.x)
  - PDO_OCI (Oracle Call Interface)
  - PDO_ODBC (ODBC v3 (IBM DB2, unixODBC and win32 ODBC))
  - PDO_PGSQL (PostgreSQL)
  - PDO_SQLITE (SQLite 3 and SQLite 2)
  - PDO_4D (4D)
- All of these drivers are not necessarily available on your system; here's a quick way to find out which drivers you have:



```
print_r(PDO::getAvailableDrivers());
```

Connecting to MySQL

- Connections are established by creating instances of the PDO base class. It doesn't matter which driver you want to use; you always use the PDO class name. The constructor accepts parameters for specifying the database source (known as the DSN) and optionally for the username and password (if any).

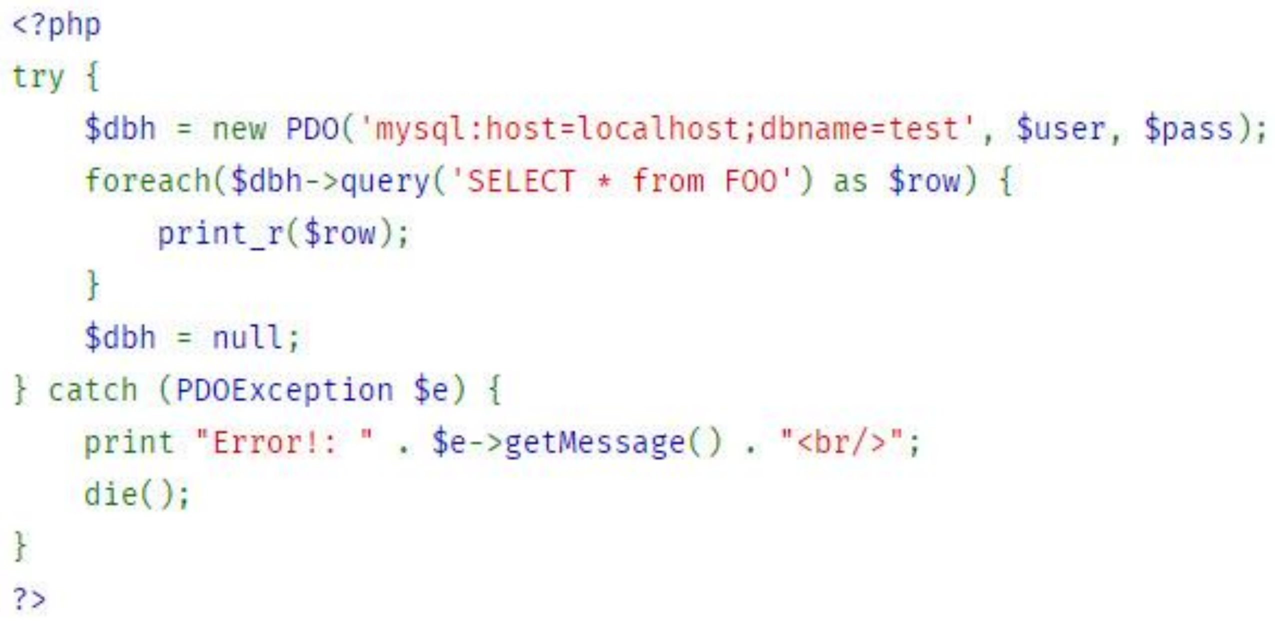


```php
<?php
$dbh = new PDO('mysql:host=localhost;dbname=test', $user, $pass);
?>
```

- If there are any connection errors, a *PDOException* object will be thrown. You may catch the exception if you want to handle the error condition, or you may opt to leave it for an application global exception handler that you set up via set_exception_handler().

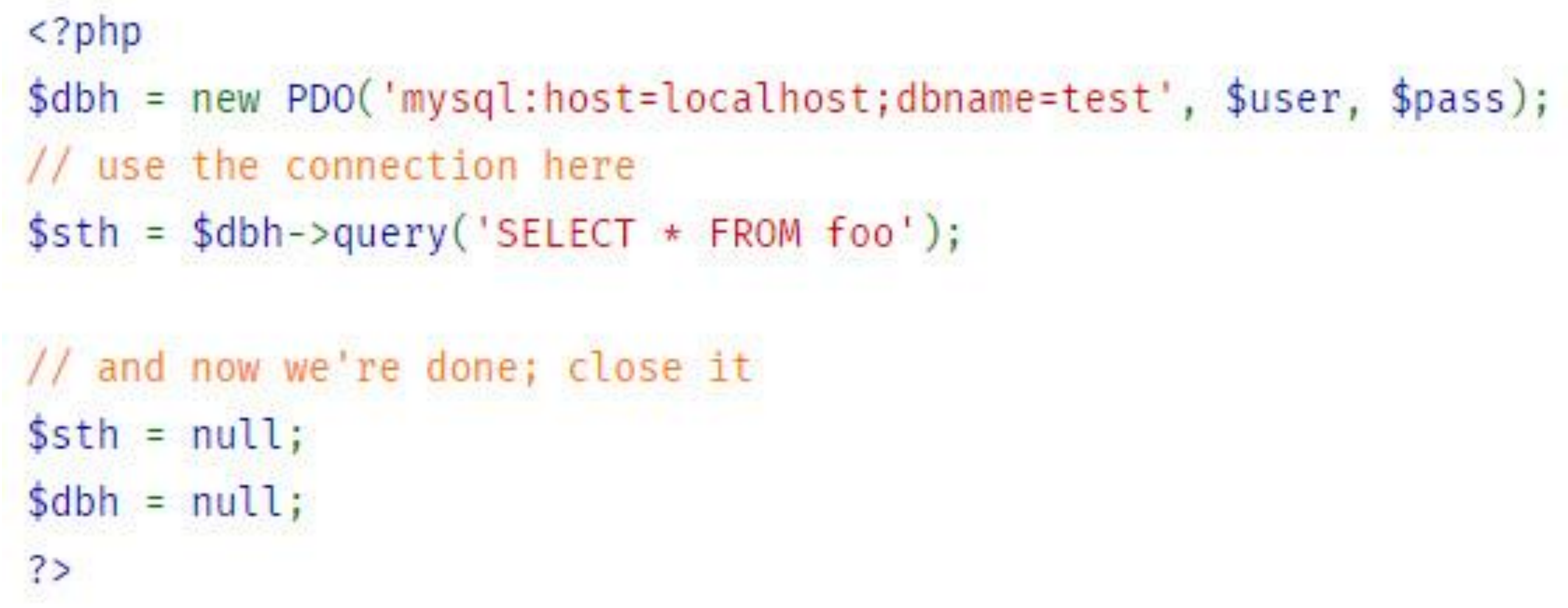
Connecting to MySQL

- Upon successful connection to the database, an instance of the PDO class is returned to your script. The connection remains active for the lifetime of that PDO object.
- To close the connection, you need to destroy the object by ensuring that all remaining references to it are deleted—you do this by assigning NULL to the variable that holds the object. If you don't do this explicitly, PHP will automatically close the connection when your script ends.



```php
<?php
try {
    $dbh = new PDO('mysql:host=localhost;dbname=test', $user, $pass);
    foreach($dbh->query('SELECT * from FOO') as $row) {
        print_r($row);
    }
    $dbh = null;
} catch (PDOException $e) {
    print "Error!: " . $e->getMessage() . "<br/>";
    die();
}
?>
```

Closing a connection

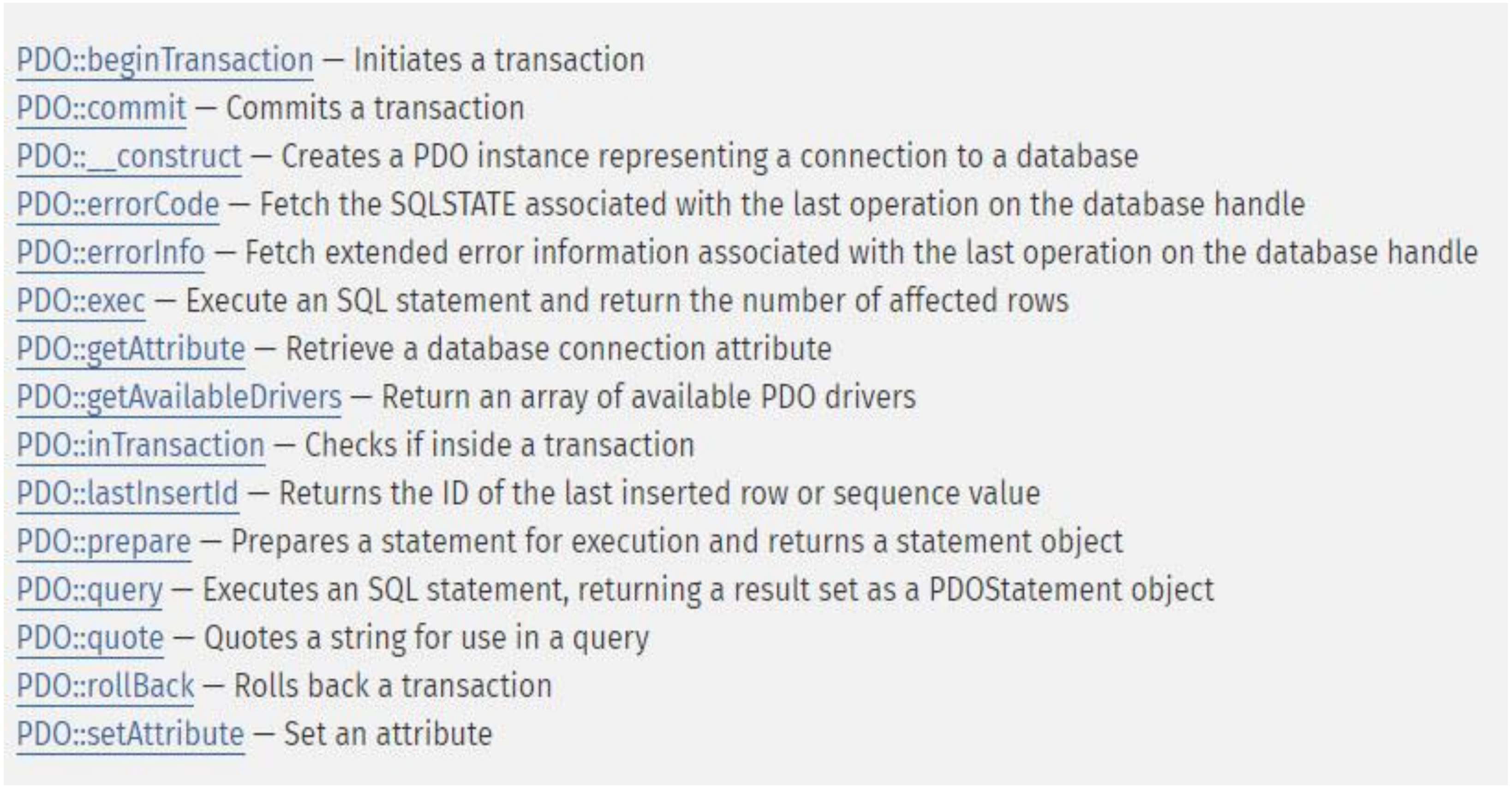


```
<?php
$dbh = new PDO('mysql:host=localhost;dbname=test', $user, $pass);
// use the connection here
$sth = $dbh->query('SELECT * FROM foo');

// and now we're done; close it
$sth = null;
$dbh = null;
?>
```

The PDO class

- Represents a connection between PHP and a database server.
- Methods:



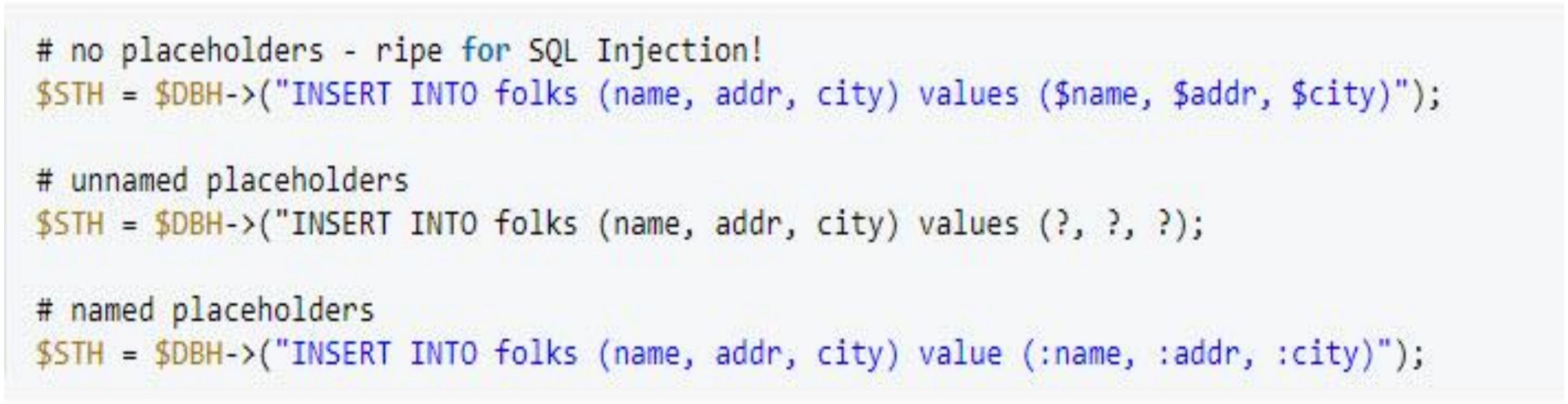
PDO::beginTransaction — Initiates a transaction
PDO::commit — Commits a transaction
PDO::__construct — Creates a PDO instance representing a connection to a database
PDO::errorCode — Fetch the SQLSTATE associated with the last operation on the database handle
PDO::errorInfo — Fetch extended error information associated with the last operation on the database handle
PDO::exec — Execute an SQL statement and return the number of affected rows
PDO::getAttribute — Retrieve a database connection attribute
PDO::getAvailableDrivers — Return an array of available PDO drivers
PDO::inTransaction — Checks if inside a transaction
PDO::lastInsertId — Returns the ID of the last inserted row or sequence value
PDO::prepare — Prepares a statement for execution and returns a statement object
PDO::query — Executes an SQL statement, returning a result set as a PDOStatement object
PDO::quote — Quotes a string for use in a query
PDO::rollBack — Rolls back a transaction
PDO::setAttribute — Set an attribute

Prepared statements

- A kind of compiled template for the SQL that an application wants to run, that can be customized using variable parameters.
- Prepared statements offer two major benefits:
  - The query only needs to be parsed (or prepared) once, but can be executed multiple times with the same or different parameters. When the query is prepared, the database will analyze, compile and optimize its plan for executing the query. For complex queries this process can take up enough time that it will noticeably slow down an application if there is a need to repeat the same query many times with different parameters. By using a prepared statement the application avoids repeating the analyze/compile/optimize cycle. This means that prepared statements use fewer resources and thus run faster.
  - The parameters to prepared statements don't need to be quoted; the driver automatically handles this. If an application exclusively uses prepared statements, the developer can be sure that no SQL injection will occur

Prepared statements

- You use a prepared statement by including placeholders in your SQL. Here's three examples: one without placeholders, one with unnamed placeholders, and one with named placeholders.



```
# no placeholders - ripe for SQL Injection!
$STH = $DBH->("INSERT INTO folks (name, addr, city) values ($name, $addr, $city)");

# unnamed placeholders
$STH = $DBH->("INSERT INTO folks (name, addr, city) values (?, ?, ?);

# named placeholders
$STH = $DBH->("INSERT INTO folks (name, addr, city) value (:name, :addr, :city)");
```

Example

- This example performs an INSERT query by substituting a *name* and a *value* for the named placeholders.

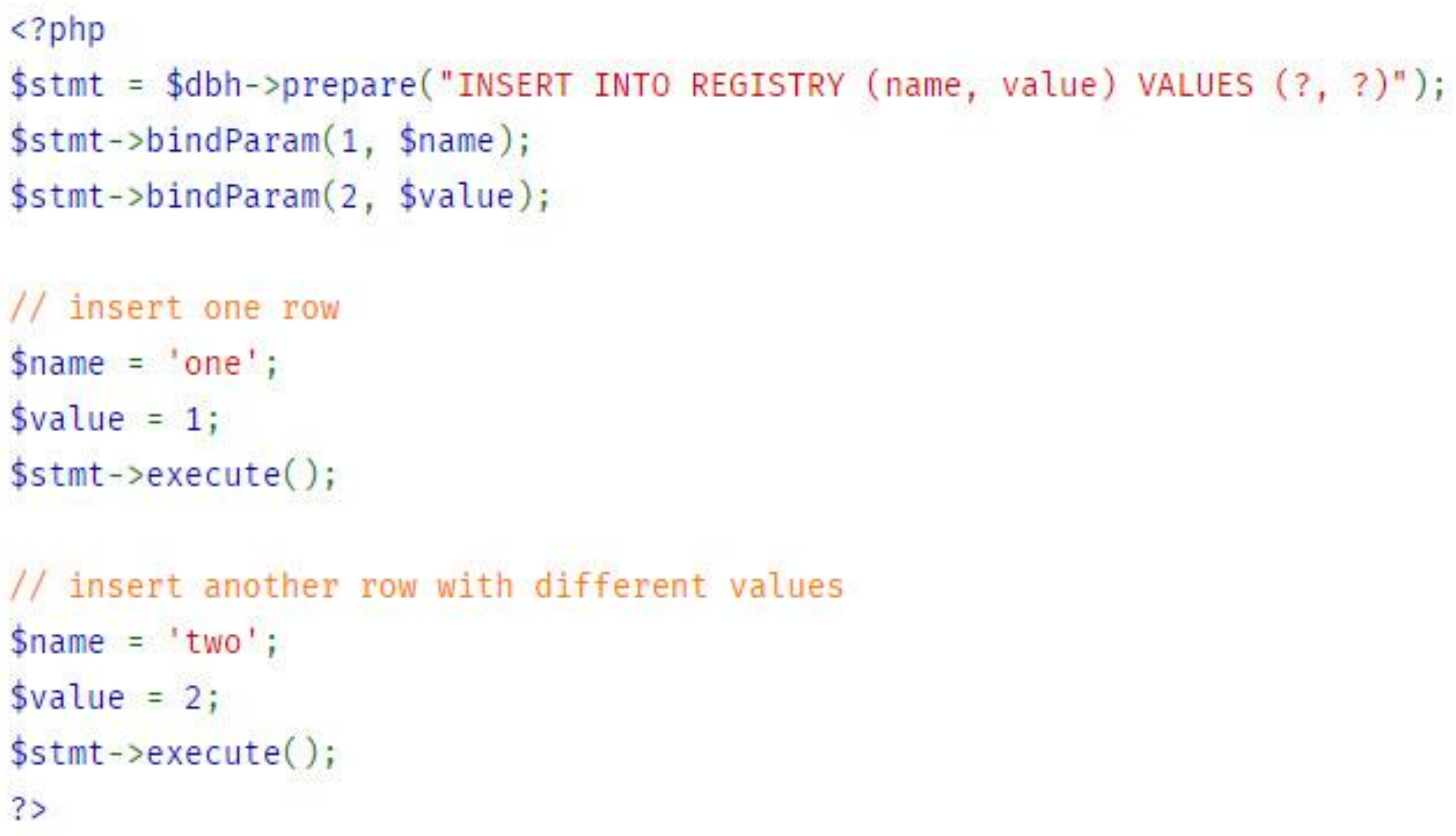
```php
<?php
$stmt = $dbh->prepare("INSERT INTO REGISTRY (name, value) VALUES (:name, :value)");
$stmt->bindParam(':name', $name);
$stmt->bindParam(':value', $value);

// insert one row
$name = 'one';
$value = 1;
$stmt->execute();

// insert another row with different values
$name = 'two';
$value = 2;
$stmt->execute();
?>
```

Example

- This example performs an INSERT query by substituting a *name* and a *value* for the positional *?* placeholders.



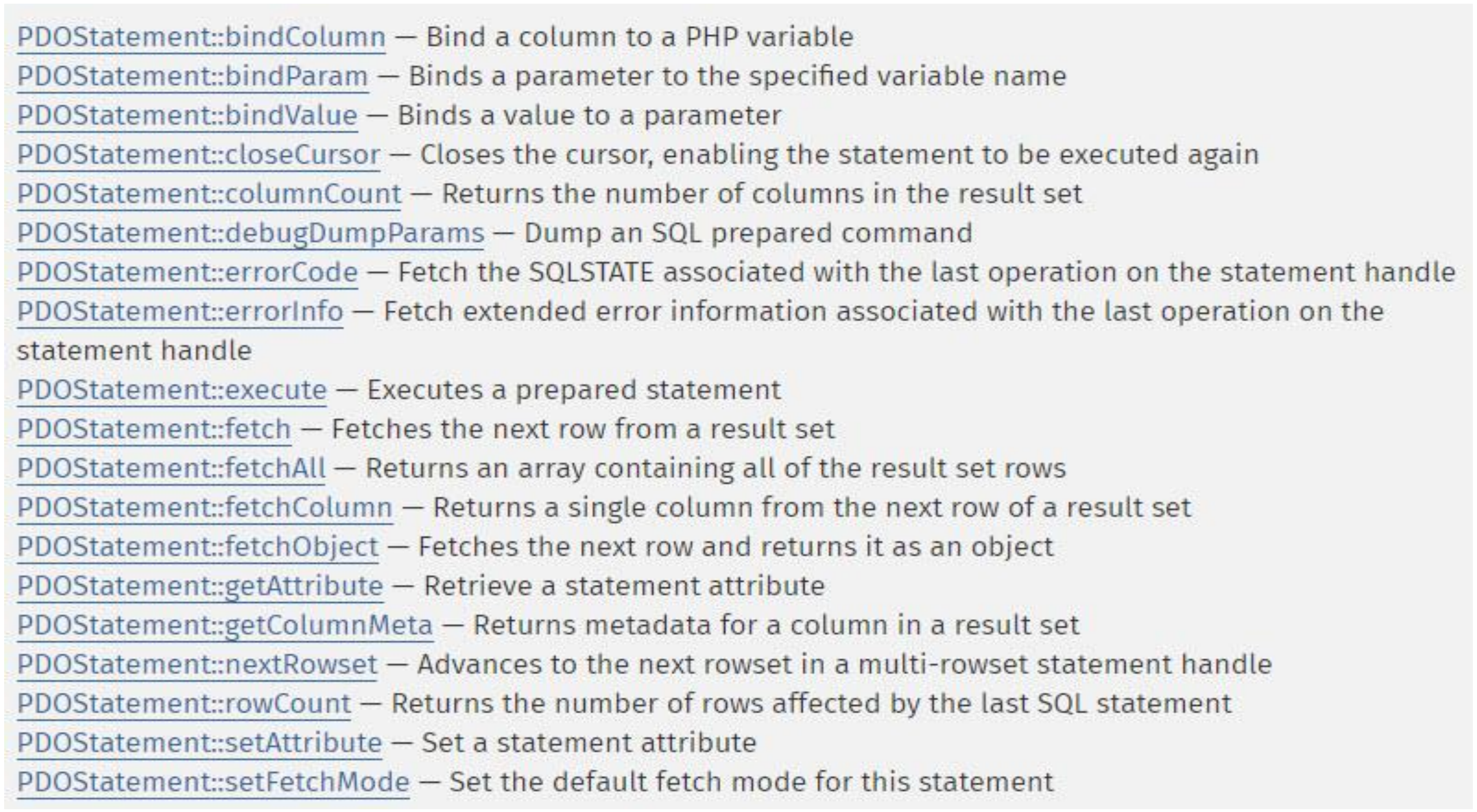
```php
<?php
$stmt = $dbh->prepare("INSERT INTO REGISTRY (name, value) VALUES (?, ?)");
$stmt->bindParam(1, $name);
$stmt->bindParam(2, $value);

// insert one row
$name = 'one';
$value = 1;
$stmt->execute();

// insert another row with different values
$name = 'two';
$value = 2;
$stmt->execute();
?>
```

The PDOStatement class

- Represents a prepared statement and, after the statement is executed, an associated result set.
- Methods:



PDOStatement::bindColumn — Bind a column to a PHP variable
PDOStatement::bindParam — Binds a parameter to the specified variable name
PDOStatement::bindValue — Binds a value to a parameter
PDOStatement::closeCursor — Closes the cursor, enabling the statement to be executed again
PDOStatement::columnCount — Returns the number of columns in the result set
PDOStatement::debugDumpParams — Dump an SQL prepared command
PDOStatement::errorCode — Fetch the SQLSTATE associated with the last operation on the statement handle
PDOStatement::errorInfo — Fetch extended error information associated with the last operation on the statement handle
PDOStatement::execute — Executes a prepared statement
PDOStatement::fetch — Fetches the next row from a result set
PDOStatement::fetchAll — Returns an array containing all of the result set rows
PDOStatement::fetchColumn — Returns a single column from the next row of a result set
PDOStatement::fetchObject — Fetches the next row and returns it as an object
PDOStatement::getAttribute — Retrieve a statement attribute
PDOStatement::getColumnMeta — Returns metadata for a column in a result set
PDOStatement::nextRowset — Advances to the next rowset in a multi-rowset statement handle
PDOStatement::rowCount — Returns the number of rows affected by the last SQL statement
PDOStatement::setAttribute — Set a statement attribute
PDOStatement::setFetchMode — Set the default fetch mode for this statement

Demo

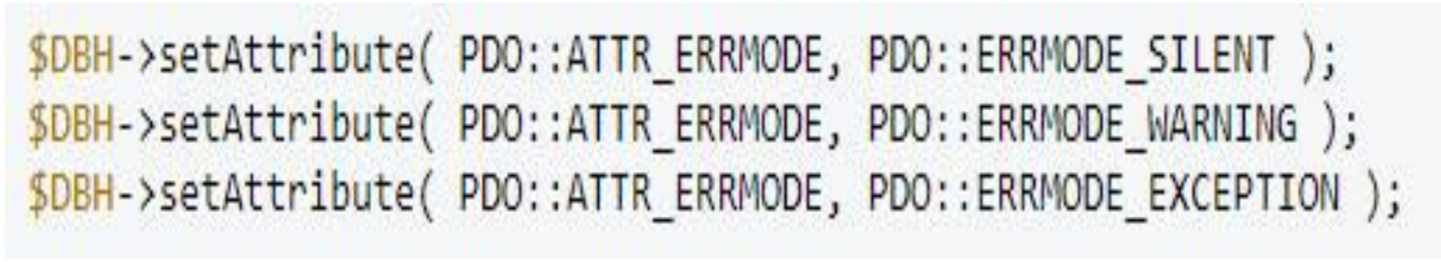
- Teacher demo code about connect to database with PDO

Some Other Helpful Methods

- lastInsertId() method
  - Is always called on the database handle, not statement handle, and will return the auto incremented id of the last inserted row by that connection.
- exec() method
  - Is used for operations that can not return data other then the affected rows.
- rowCount() method
  - Returns an integer indicating the number of rows affected by an operation.

Exceptions and PDO

- PDO can use exceptions to handle errors, which means anything you do with PDO should be wrapped in a try/catch block.
- You can force PDO into one of three error modes by setting the error mode attribute on your newly created database handle.
- No matter what error mode you set, an error connecting will always produce an exception, and creating a connection should always be contained in a try/catch block.
- Example:



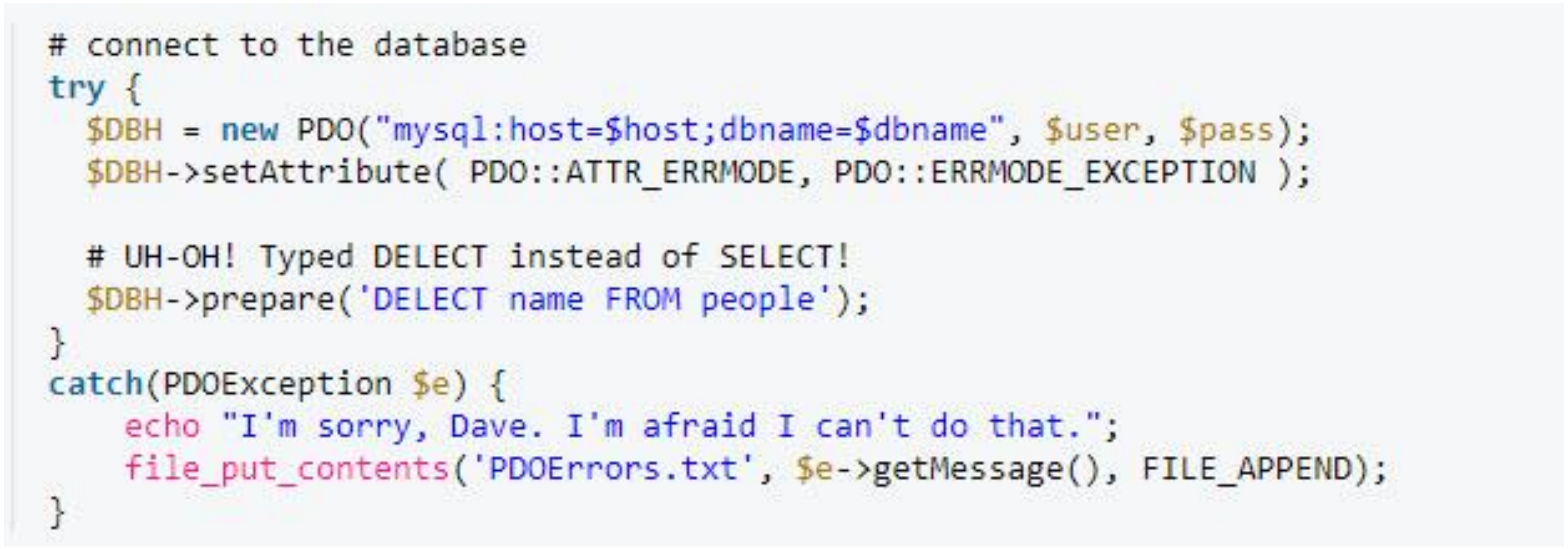
```
$DBH->setAttribute( PDO::ATTR_ERRMODE, PDO::ERRMODE_SILENT );
$DBH->setAttribute( PDO::ATTR_ERRMODE, PDO::ERRMODE_WARNING );
$DBH->setAttribute( PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION );
```

Exceptions and PDO

- **PDO::ERRMODE_SILENT**
  - This is the default error mode. If you leave it in this mode, you'll have to check for errors in the way you're probably used to if you used the mysql or mysqli extensions. The other two methods are more ideal for DRY programming.
- **PDO::ERRMODE_WARNING**
  - This mode will issue a standard PHP warning, and allow the program to continue execution. It's useful for debugging.
- **PDO::ERRMODE_EXCEPTION**
  - This is the mode you should want in most situations. It fires an exception, allowing you to handle errors gracefully and hide data that might help someone exploit your system.

Exceptions and PDO

- Example:



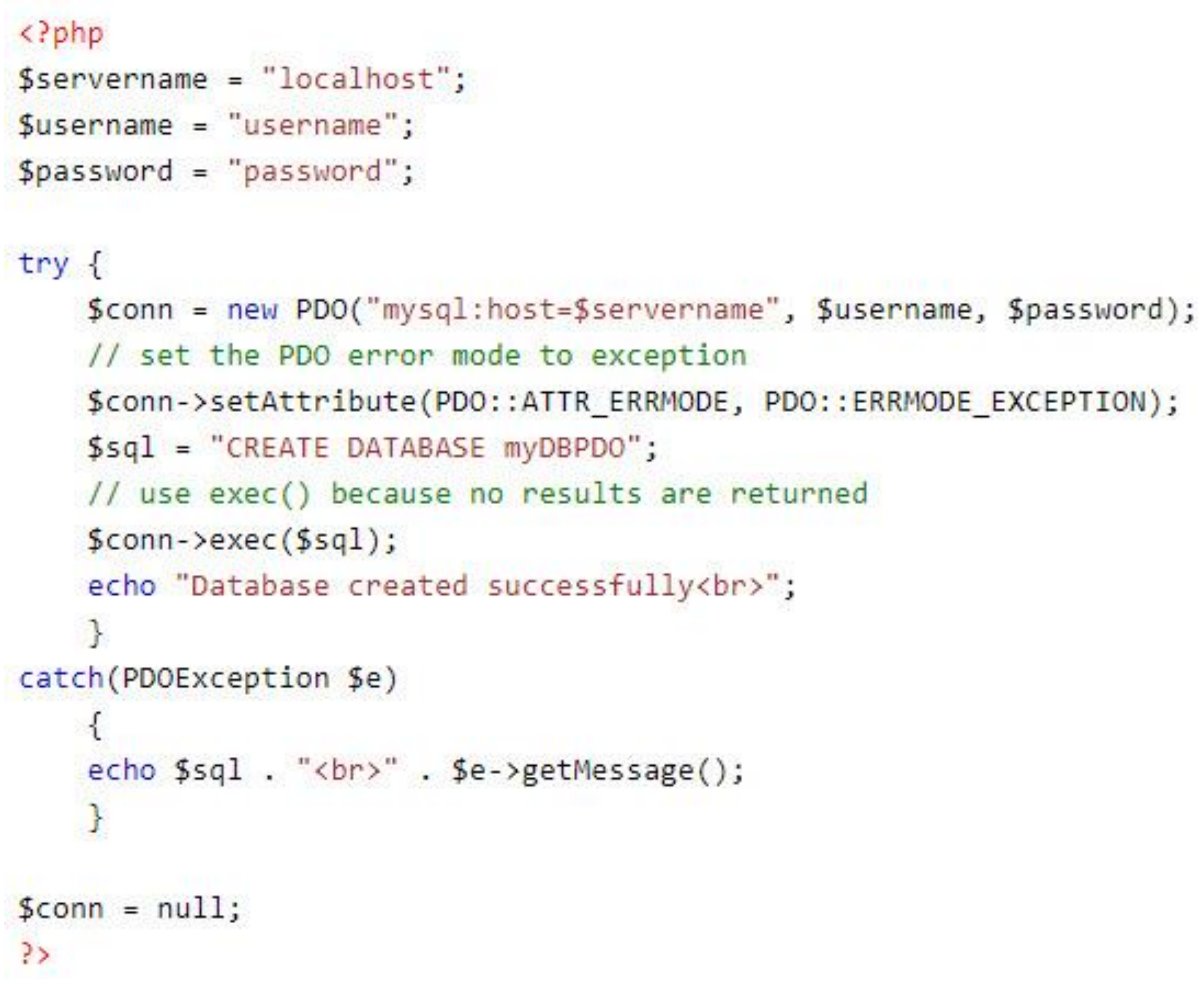
```
# connect to the database
try {
  $DBH = new PDO("mysql:host=$host;dbname=$dbname", $user, $pass);
  $DBH->setAttribute( PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION );

  # UH-OH! Typed DELECT instead of SELECT!
  $DBH->prepare('DELECT name FROM people');
}
catch(PDOException $e) {
    echo "I'm sorry, Dave. I'm afraid I can't do that.";
    file_put_contents('PDOErrors.txt', $e->getMessage(), FILE_APPEND);
}
```

- Note: A great benefit of PDO is that it has an exception class to handle any problems that may occur in our database queries. If an exception is thrown within the try{ } block, the script stops executing and flows directly to the first catch(){ } block.

Create database

- Example create database in PDO:



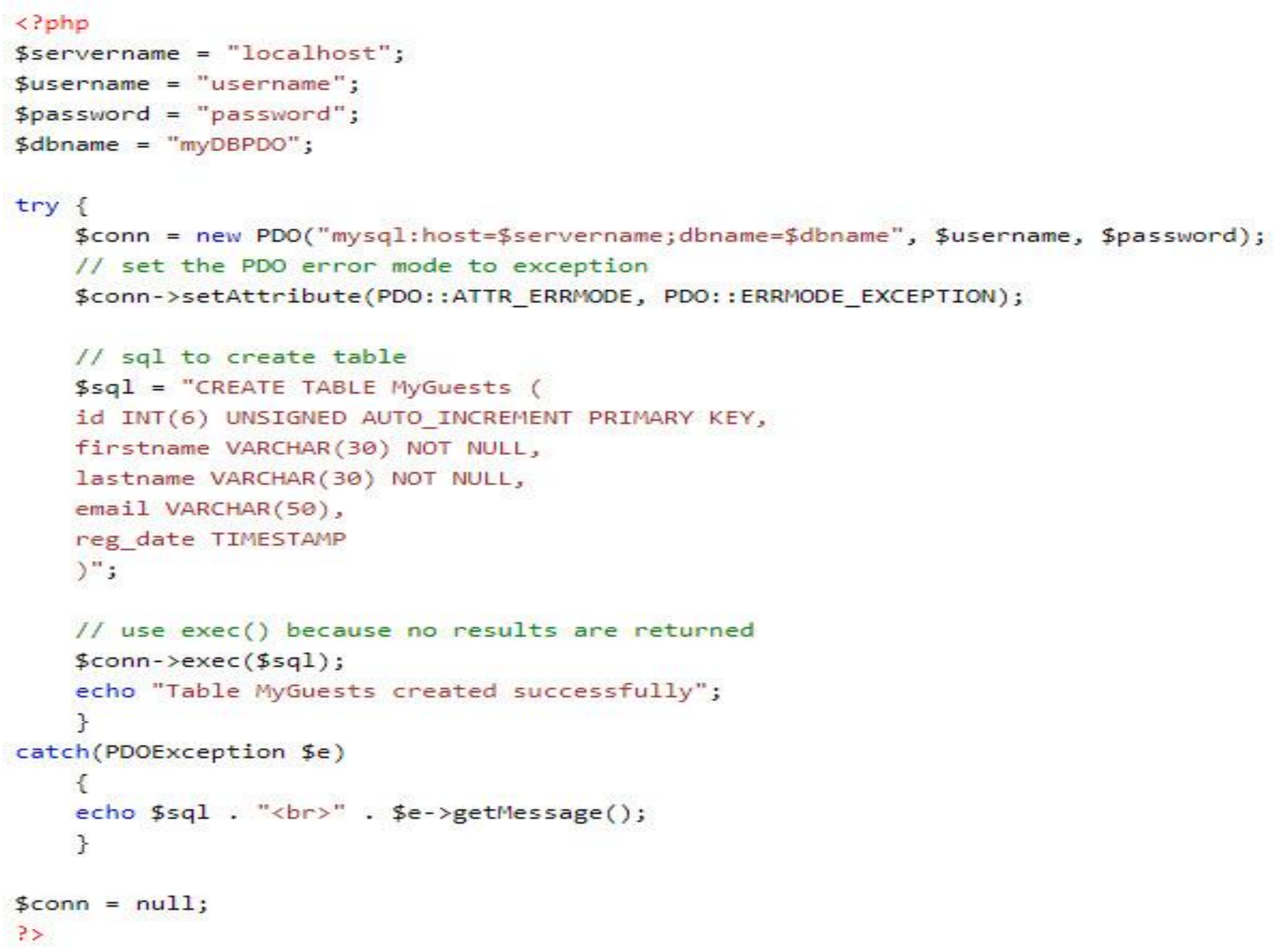
```php
<?php
$servername = "localhost";
$username = "username";
$password = "password";

try {
    $conn = new PDO("mysql:host=$servername", $username, $password);
    // set the PDO error mode to exception
    $conn->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
    $sql = "CREATE DATABASE myDBPDO";
    // use exec() because no results are returned
    $conn->exec($sql);
    echo "Database created successfully<br>";
    }
catch(PDOException $e)
    {
    echo $sql . "<br>" . $e->getMessage();
    }

$conn = null;
?>
```

Create table

- Example create table in PDO:



```php
<?php
$servername = "localhost";
$username = "username";
$password = "password";
$dbname = "myDBPDO";

try {
    $conn = new PDO("mysql:host=$servername;dbname=$dbname", $username, $password);
    // set the PDO error mode to exception
    $conn->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);

    // sql to create table
    $sql = "CREATE TABLE MyGuests (
    id INT(6) UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    firstname VARCHAR(30) NOT NULL,
    lastname VARCHAR(30) NOT NULL,
    email VARCHAR(50),
    reg_date TIMESTAMP
    )";

    // use exec() because no results are returned
    $conn->exec($sql);
    echo "Table MyGuests created successfully";
    }
catch(PDOException $e)
    {
    echo $sql . "<br>" . $e->getMessage();
    }

$conn = null;
?>
```

Demo

- Teacher demo code about database operation using PDO

Summary

- Introduction to PDO
- PreparedStatement
- Exception in PDO