PROJECT #1: “Auction Site” v1.0

PART 2: “Create the Database class”

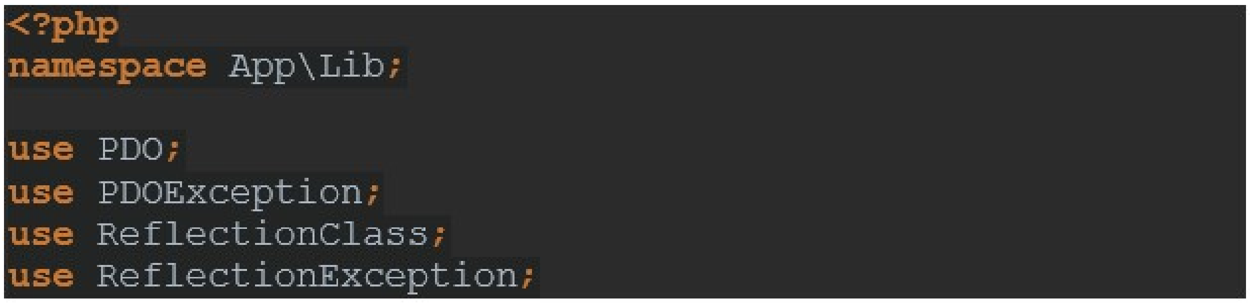
Objectives:

- Create a Database class

Instructions:

1. Create a new file called **Database.php** in the **app/Lib/** directory.

   The **Database** class will reside in the **App\Lib** namespace and use the following list of classes:



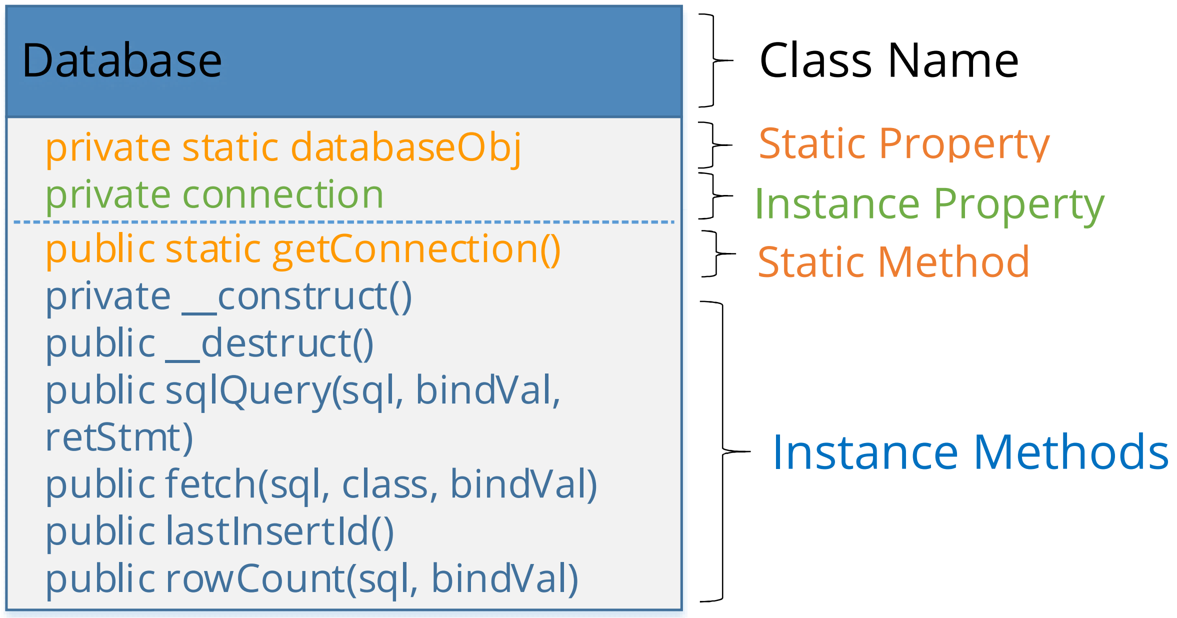
```php
<?php
namespace App\Lib;

use PDO;
use PDOException;
use ReflectionClass;
use ReflectionException;
```

   Recall that namespaces must be the first thing defined within a file after opening php tags.

   Continue by defining the skeleton for the class.

   Recall that a class is made up of instance variables also known as properties (state) and instance methods (behaviour).

In the **Database** class definition, you will create the following properties and instance methods based on the UML diagram below:



The Database class contains only one static property; a database object. The database class also contains one instance property; the open connection to the database. Notice that the visibility is set to **private** for both properties.

The database class contains the following methods:

- A static method called getConnection. This method is used to retrieve a new/existing connection to the database.

- A constructor method. Notice its visibility is set to **private**.

- A destructor method. This method gracefully closes the connection to the database.

- A method called sqlQuery which has three parameters. This method will be used to **Create**, **Update** and **Delete** records from the database. This method will use PDO binding to protect against SQL injection.

- A method called fetch which has three parameters. This method will be used to **Retrieve** records from the database.

- A method called lastInsertId. This method will be used to determine the id of the last inserted record into the database.

- A method called rowCount. This method is used to determine the number of rows in a result set from a given SQL query.

Defining Database Class Properties

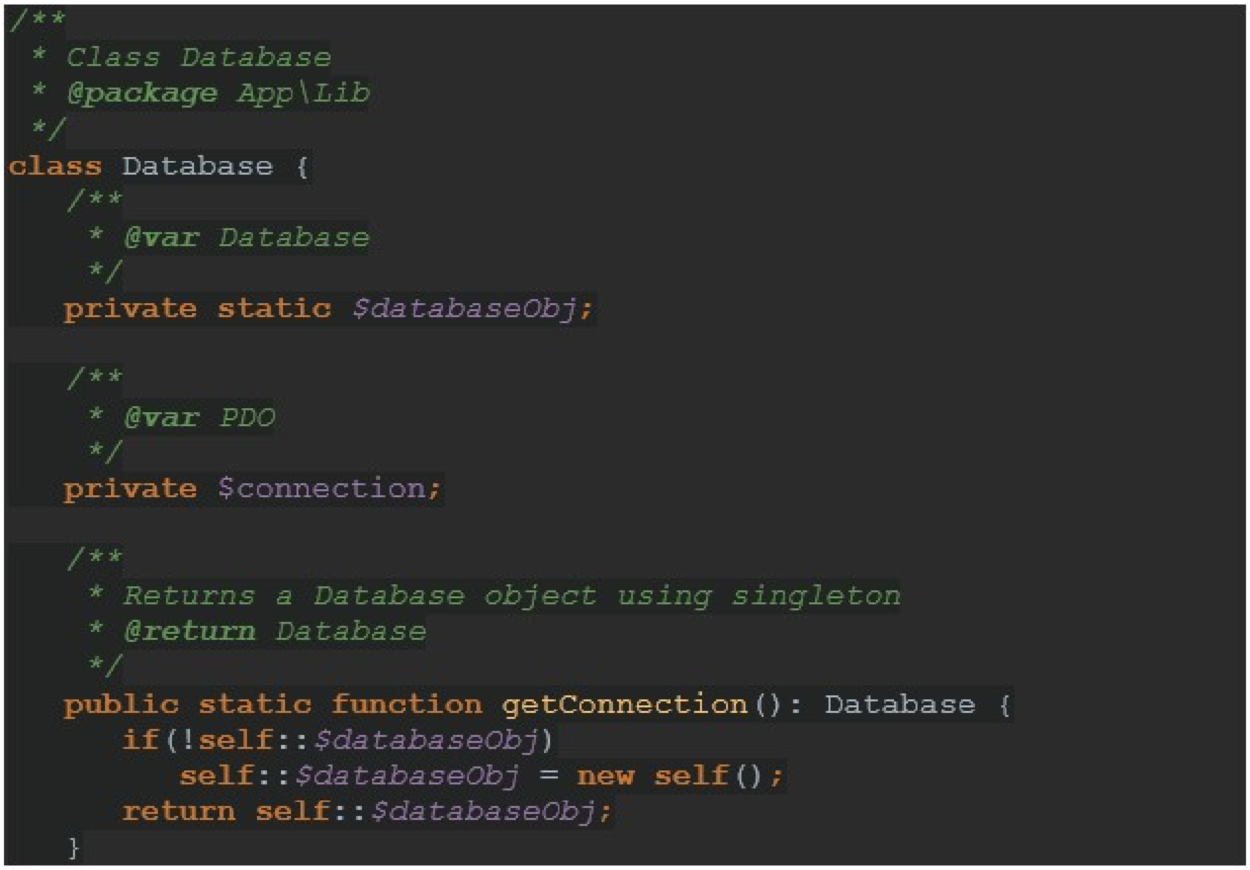
2. Within the **Database** class definition, create the private static variable as illustrated in the UML diagram.

3. Create a private instance property as illustrated in the UML diagram. This will contain the open connection to the database.

Defining Database Class Methods

Method: getConnection() : Database

4. Within the **Database** class definition, create the public static method named **getConnection**. This function uses a common PHP design pattern called the singleton pattern. It’s purpose is to retrieve an open connection to the database. If no open connection has already been created, it will make a new one.



```php
/**
 * Class Database
 * @package App\Lib
 */
class Database {
   /**
    * @var Database
    */
   private static $databaseObj;

   /**
    * @var PDO
    */
   private $connection;

   /**
    * Returns a Database object using singleton
    * @return Database
    */
   public static function getConnection(): Database {
      if(!self::$databaseObj)
         self::$databaseObj = new self();
      return self::$databaseObj;
   }
```

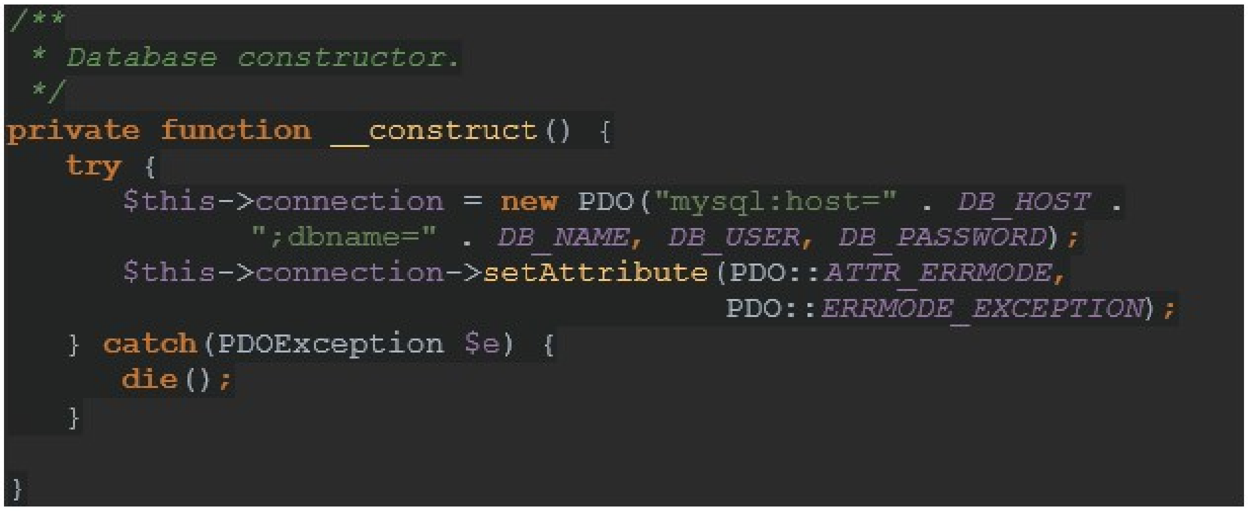
Method: __construct()

Within the **Database** class definition, create a private **constructor** method. The purpose of this method is to create a new connection to the database and store it in the **$connection** instance variable. **Note:** this method has private visibility. This means that the object cannot be instantiated using the new keyword. This forces us to use the **getConnection** method we created earlier to retrieve the active database connection.

5. Create a **try-catch block**. Within the **try** section, we'll attempt to create the connection to the database. If we fail to create a connection, use the **catch** section to report the error.

6. Within the **try** section of the constructor method, use the PDO method to create a new connection to the database.

7. Store this new connection object in the instance variable **$connection** which you created earlier.



```
/**
 * Database constructor.
 */
private function __construct() {
    try {
        $this->connection = new PDO("mysql:host=" . DB_HOST .
                ";dbname=" . DB_NAME, DB_USER, DB_PASSWORD);
        $this->connection->setAttribute(PDO::ATTR_ERRMODE,
                                        PDO::ERRMODE_EXCEPTION);
    } catch(PDOException $e) {
        die();
    }

}
```

8. Use the PDO **setAttribute** method to turn on exceptions using exceptions.

9. Within the catch section, call the **die()** function. We will replace this with code to log errors in the next section.

Method: __destruct()

Within the **Database** class definition, create a public **destructor** method. The purpose of this method is to gracefully close the open connection to the database. To gracefully close a database connection, simply assign the connection a new value of ***null***.

10. Within the destructor method, set the instance property **$connection** to ***null***.

Method: sqlQuery(string $sql, $bindVal = null, bool $retStmt = false)

Create a public method called **sqlQuery**. This methods purpose will be to bind and execute SQL statements. This means every SQL statements will be sanitized to prevent against SQL injection.

Note: This method will be the only method which executes SQL queries. It will be used as a helper method for all others which need to execute SQL queries.

The method has three parameters:

- A string parameter named **$sql**. This is SQL statement that is to be executed.
- An associative array of **key => value** pairs called **$bindVal**. The key represents the marker in the SQL statement and its value is what will be bound to the SQL statement.
- A boolean parameter named **$retStmt**, which indicates whether the executed statement or the results of the statement should be returned from the method.

11. In the method, set the parameters default values for the **$bindVal** variable to ***null*** and **$retStmt** to **false**.

12. Create a **try-catch** block. In the **try** section, you will perform the execution. In the **catch** section, invoke the **die()** method. Again, you will replace this with code to log errors in the next section.

13. Within the **try** section use the connection to the database (stored in the instance variable) to prepare the SQL statement **$sql**. Store the prepared statement in a variable called **$statement**.

14. Use a conditional (**if**) statement to check if **$bindVal** is an array.
    Hint: Use the **is_array()** function built into PHP.

The reason we're doing this is to determine if the SQL statement needs to bind values to markers.

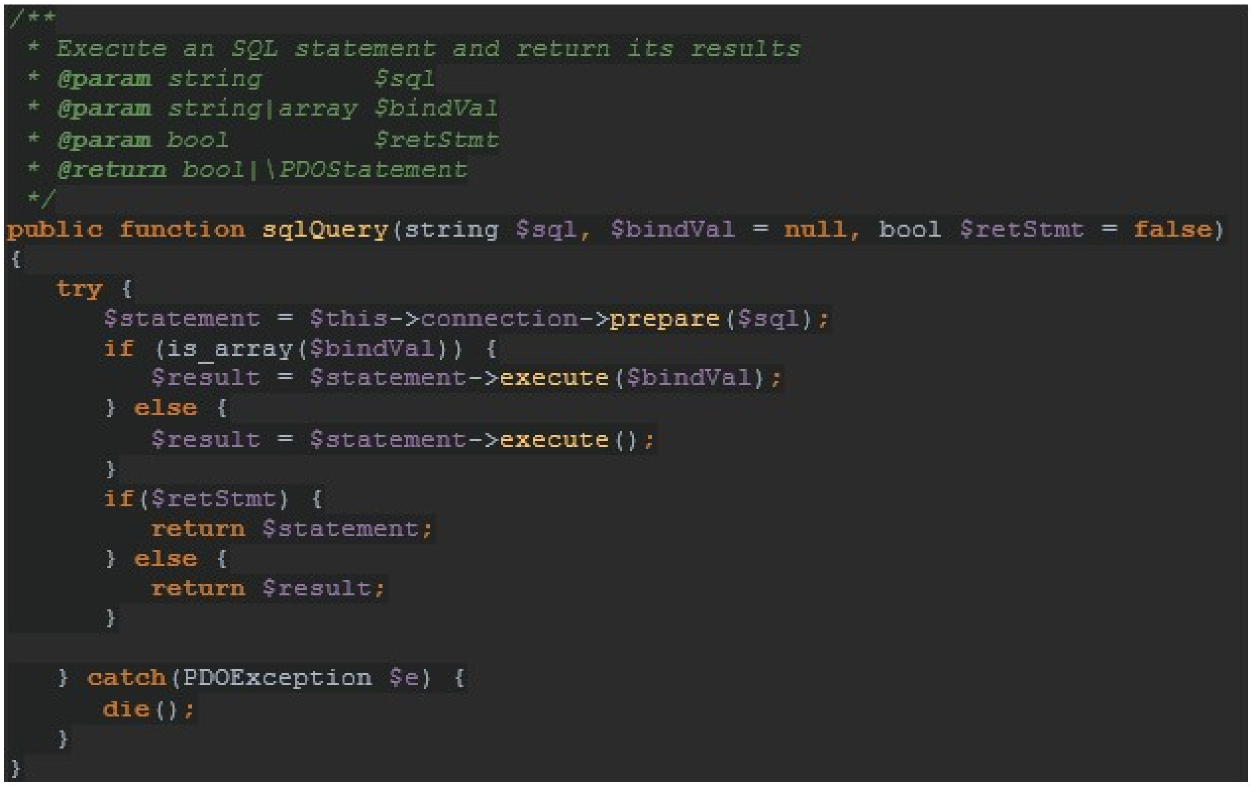
15. If the result of the **if** statement is ***true*** then execute the statement using the PDO **execute** method with **$bindVal** as an argument and save the results into a variable called **$result**

16. If the result of the **if** statement is ***false***, simply execute the statement without any arguments and save the results into a variable called **$result**

17. Use a conditional (**if**) statement to check if **$retStmt** is a boolean ***true***.

18. If the result of the **if** statement is ***true***, return the variable **$statement** from the method. This is a **PDOStatement** object that contains the records from the SQL statement that was executed.

19. If the result of the **if** statement is ***false***, return the variable **$result** from the method. This is a boolean ***true*** or ***false*** indicated whether the statement was successfully executed.



```php
/**
 * Execute an SQL statement and return its results
 * @param string       $sql
 * @param string|array $bindVal
 * @param bool         $retStmt
 * @return bool|\PDOStatement
 */
public function sqlQuery(string $sql, $bindVal = null, bool $retStmt = false)
{
    try {
        $statement = $this->connection->prepare($sql);
        if (is_array($bindVal)) {
            $result = $statement->execute($bindVal);
        } else {
            $result = $statement->execute();
        }
        if($retStmt) {
            return $statement;
        } else {
            return $result;
        }

    } catch(PDOException $e) {
        die();
    }
}
```

Method: fetch(string $sql, string $class, $bindVal = null) : array

Create a public method called **fetch**. This methods purpose will be to retrieve results from an SQL query. The results will be placed in an associative array.

Note: this method will perform the **Retrieve** operations of the **CRUD** model.

The method has **three** parameters:

- A string variable named **$sql**. This is SQL statement that is to be executed.
- The second paramter is the class name which will store each individual record retrieved from the database.
  Ex. A user object will contain a user record from the DB.
- The third parameter, is an associative array of bindings and their values to be passed to the **sqlQuery** method that we created earlier.

20. In the method, set the parameters default values for the **$bindVal** variable to ***null***.

21. Within the body of the **fetch** method, call the **sqlQuery** method to execute the SQL query. Pass **$sql**, **$bindVal** and boolean ***true*** as arguments to the **sqlQuery** method. Store the results in a variable called **$statement**.

22. Invoke the PDO method **rowCount()** on the **$statement** variable to determine the number of rows that were returned by the database query.

23. Create a conditional statement to test the number of rows:

    a. If there is 0 rows returned, return an empty array (**[ ]**) from the method.

24. Create a **try-catch** block. In the **try** section, you will add the code to instantiate a new object for each record in the table and add it to an array of objects. In the **catch** section, invoke the **die()** method. Again, you will replace this with code to log errors in the next section.

25. Within the **try** section, instantiate a new object of **ReflectionClass** passing **$class** to the constructor. Use the variable name **$reflect**.

26. Create a conditional statement to test if there exists a constructor in the reflection class by calling the following method:

```
if($reflect->getConstructor() == null)
```

    a. If the result of the **if** statement is ***true***, create a new variable called **$ctor_args** and assign it the value of **[ ]** (empty array).

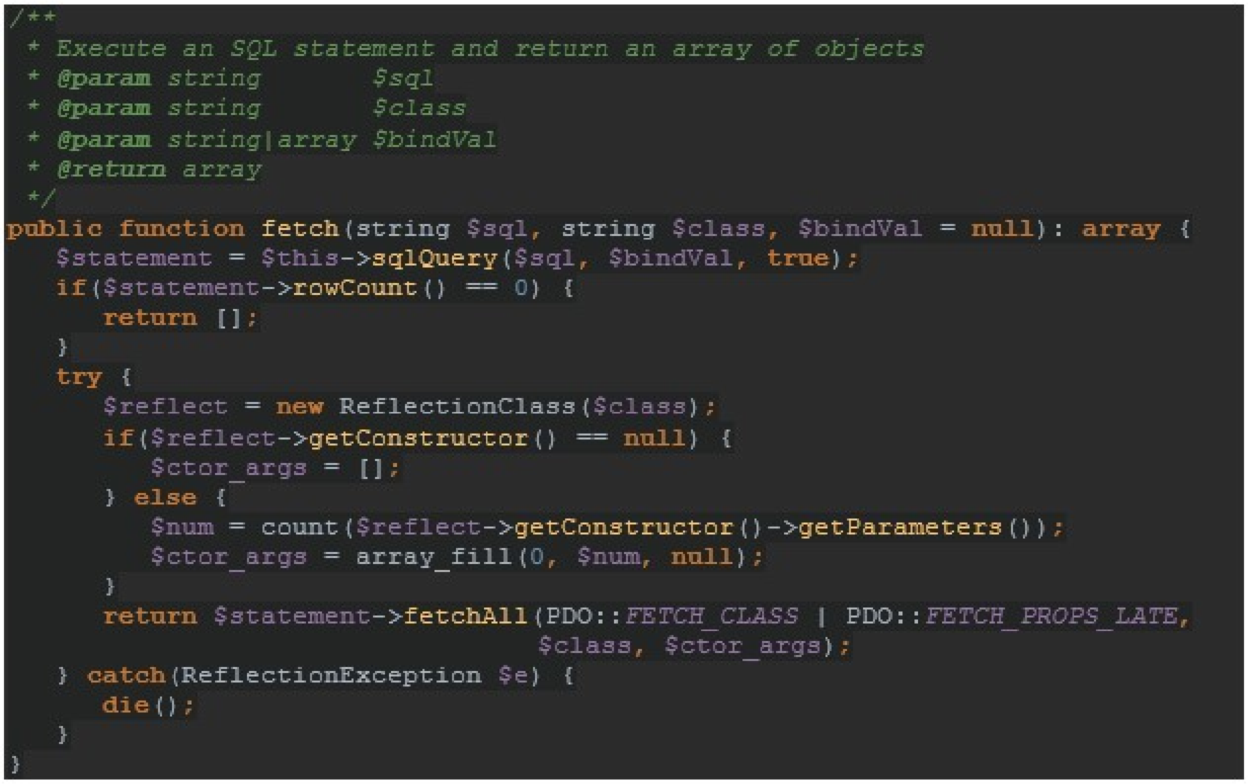
    b. Otherwise, determine the number of parameters in the constructor and create an variable **$ctor_args** filled with nulls. See the following code:

```
$num = count($reflect->getConstructor()->getParameters());
$ctor_args = array_fill(0, $num, null);
```

27. Finally, call the **fetchAll()** method on the **$statment** variable with the following argument list:

```
PDO::FETCH_CLASS | PDO::FETCH_PROPS_LATE, $class, $ctor_args
```

This instructs PDO to create an array of objects using the class defined in **$class** and the list of constructor argments stored in **$ctor_args**.



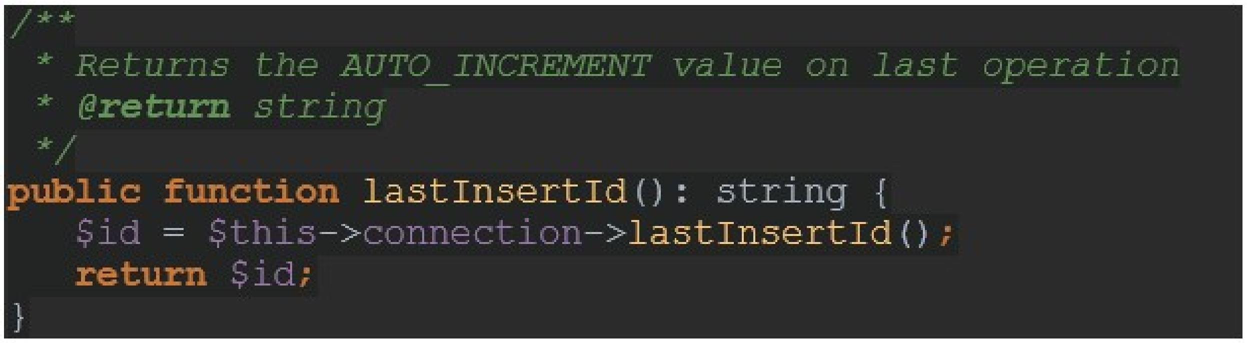
```
/**
 * Execute an SQL statement and return an array of objects
 * @param string       $sql
 * @param string       $class
 * @param string|array $bindVal
 * @return array
 */
public function fetch(string $sql, string $class, $bindVal = null): array {
   $statement = $this->sqlQuery($sql, $bindVal, true);
   if($statement->rowCount() == 0) {
      return [];
   }
   try {
      $reflect = new ReflectionClass($class);
      if($reflect->getConstructor() == null) {
         $ctor_args = [];
      } else {
         $num = count($reflect->getConstructor()->getParameters());
         $ctor_args = array_fill(0, $num, null);
      }
      return $statement->fetchAll(PDO::FETCH_CLASS | PDO::FETCH_PROPS_LATE,
                                  $class, $ctor_args);
   } catch(ReflectionException $e) {
      die();
   }
}
```

Method: lastInsertId() : string

Create a public method called **lastInsertId**. The purpose of this method is to return the integer assigned from the **auto_increment** field in the database table after an insert statement.

28. Within the body of the method, call the built-in PDO method **lastInsertId** on the open connection. This PDO method returns the id for the last executed SQL insert statement.

29. Return the id from the method.



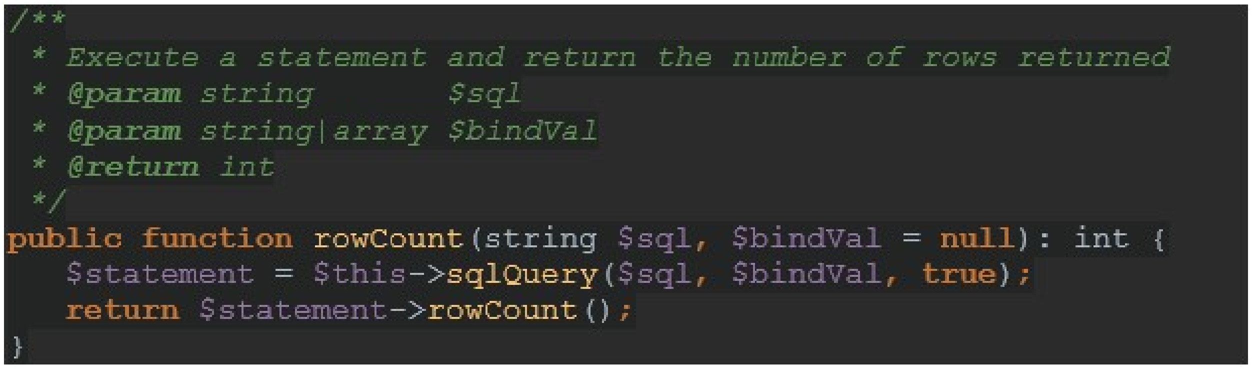
```
/**
 * Returns the AUTO_INCREMENT value on last operation
 * @return string
 */
public function lastInsertId(): string {
   $id = $this->connection->lastInsertId();
   return $id;
}
```

Method: rowCount(string $sql, $bindVal = null) : int

Create a public method called **rowCount**. This method has two parameters; **$sql** and **$bindVal**. The purpose of this method is to return the number of results returned by a SQL statement.

30. Call the helper method **sqlBindQuery** we created earlier, passing the **$sql**, **$bindVal** and a boolean ***true*** as arguments. Save the results into a variable **$statement**.

31. Call the built-in PDO method **rowCount** on the **$statement** variable, returning the result.



```php
/**
 * Execute a statement and return the number of rows returned
 * @param string       $sql
 * @param string|array $bindVal
 * @return int
 */
public function rowCount(string $sql, $bindVal = null): int {
   $statement = $this->sqlQuery($sql, $bindVal, true);
   return $statement->rowCount();
}
```

All the properties and methods that we need for the **Database** class are now complete.

We cannot test the **Database** class yet – first, we need to create the tables and PHPUnit test configuration (coming up next).

32. Post the **Database.php** file to the server in the **app/Lib/** directory.

You're now ready to move on to the next section.