ScopeTreasure Courses

PHP COURSES · LESSON PH.11 · CODE SCOPE SERIES

MySQL and OOP Basics

Real apps keep data in a database — learn connecting with mysqli, SELECT and INSERT, the prepared-statement shield, and classes and objects in one lesson.

Introduction

Files work for a guestbook, but they break the moment two visitors write at once, or you need to find "all students with marks above 80". That is what a **database** is for — and MySQL is the database that ships with XAMPP and powers most PHP sites.

PHP talks to MySQL through **mysqli**. The flow is always the same four steps: connect, ask a question (a query), read the answer, close. The question is written in **SQL** — `SELECT * FROM students` means "show me every student" — and the full SQL language has its own series in the roadmap (Phase 10). Here you learn just enough to connect and use it from PHP.

The second half of this lesson is **OOP — object-oriented programming**. A **class** is a blueprint; an **object** is a real thing built from it. Instead of loose variables `$name` and `$marks` floating around, a `Student` object carries its own name and marks, and its own methods like `grade()`. Laravel is built entirely from classes and objects, so this lesson is your first step into that world.

(Roman Urdu: Database matlab "mehfooz almari" — data wahan rehta hai, PHP sirf sawal poochta hai. Class blueprint hai, object us se bani hui asal cheez hai.)

Key points

- **MySQL is a database server; XAMPP already runs one.** Start MySQL in the XAMPP control panel, then manage it at `http://localhost/phpmyadmin`.
- **`mysqli_connect("localhost", "root", "", "school")`** connects — server, username, password (empty on XAMPP), database name.
- **Always check the connection:** `if (!$conn) { die("Connection failed"); }` — a silent failure wastes hours.
- **A query is a question in SQL:** `SELECT * FROM students` reads rows; `INSERT INTO students (name, marks) VALUES ('Ali', 88)` adds one.
- **`mysqli_query($conn, $sql)` runs the question; `mysqli_fetch_assoc($result)` reads one row at a time** as an associative array.
- **Never build a query by gluing user input into the string.** A visitor can type SQL into your form and steal or delete your data — this is called SQL injection.
- **Prepared statements are the shield:** `prepare()` + `bind_param()` + `execute()` keeps user input as data, never as code.
- **A class is defined with `class Student { ... }`; an object is built with `new Student()`.**

- **Properties are the object's variables (`public $name`); methods are its functions (`public function grade()`).**
- **Inside a method, `$this` means "this object itself"** — `$this->marks` reads this object's marks.

Examples

Example 1 — Connecting to MySQL

```
<?php
$conn = mysqli_connect("localhost", "root", "", "school");
if (!$conn) {
    die("Connection failed: " . mysqli_connect_error());
}
echo "Connected!";
mysqli_close($conn);
?>
```

What you will get:

```
Connected!
```

Make the `school` database first in phpMyAdmin (Databases → Create). If the name is wrong, `die` prints the error instead of failing silently later.

Example 2 — Reading rows with SELECT

```
<?php
$conn = mysqli_connect("localhost", "root", "", "school");
$result = mysqli_query($conn, "SELECT name, marks FROM students");
while ($row = mysqli_fetch_assoc($result)) {
    echo $row["name"] . ": " . $row["marks"] . "<br>";
}
mysqli_close($conn);
?>
```

What you will get:

```
Ali: 88
Sara: 92
Bilal: 76
```

Each `mysqli_fetch_assoc` call hands you the next row as an array like `["name" => "Ali", "marks" => 88]`. When the rows run out it returns `null` and the `while` loop stops. This is the same array shape you met

in PH.7.

Example 3 — Adding a row with INSERT

```
<?php
$conn = mysqli_connect("localhost", "root", "", "school");
$sql = "INSERT INTO students (name, marks) VALUES ('Hira', 95)";
if (mysqli_query($conn, $sql)) {
    echo "Saved! New id: " . mysqli_insert_id($conn);
} else {
    echo "Error: " . mysqli_error($conn);
}
mysqli_close($conn);
?>
```

What you will get:

```
Saved! New id: 4
```

`mysqli_insert_id` gives the auto-numbered id of the row just added. Always check the result — a failed insert that nobody notices is a bug your users find first.

Example 4 — The shield: a prepared statement

```
<?php
$conn = mysqli_connect("localhost", "root", "", "school");
$name = $_POST["name"]; // anything the visitor typed — even SQL
$marks = (int) $_POST["marks"];
$stmt = mysqli_prepare($conn, "INSERT INTO students (name, marks) VALUES (?, ?)");
mysqli_stmt_bind_param($stmt, "si", $name, $marks);
mysqli_stmt_execute($stmt);
echo "Saved safely!";
mysqli_close($conn);
?>
```

What you will get:

```
Saved safely! — even if the visitor typed SQL as their name.
```

The `?` marks are placeholders. `bind_param("si", ...)` says the first is a string, the second an integer. MySQL treats the values as pure data — they can never become code. This is the one habit that separates safe apps from hacked ones.

Example 5 — Your first class and object

```
<?php
class Student {
    public $name;
    public $marks;
    public function grade() {
        if ($this->marks >= 90) return "A";
        if ($this->marks >= 75) return "B";
        if ($this->marks >= 50) return "C";
        return "F";
    }
}
$ali = new Student();
$ali->name = "Ali";
$ali->marks = 88;
echo $ali->name . " — Grade " . $ali->grade();
?>
```

What you will get:

```
Ali — Grade B
```

`Student` is the blueprint; `$ali` is one real student built from it. `$this->marks` inside `grade()` means "this student's own marks". The method belongs to the object, so the grade logic lives with the data it grades.

Example 6 — A constructor: build the object ready-made

```
<?php
class Student {
    public $name;
    public $marks;
    public function __construct($name, $marks) {
        $this->name = $name;
        $this->marks = $marks;
    }
    public function passed() {
        return $this->marks >= 50;
    }
}
$sara = new Student("Sara", 92);
echo $sara->name . " passed: " . ($sara->passed() ? "Yes" : "No");
?>
```

What you will get:

```
Sara passed: Yes
```

`__construct` runs the moment `new Student(...)` is called, so the object is born complete — no half-empty objects, no forgotten fields.

Quick recap

- Connect with `mysqli_connect`; always check the result; close with `mysqli_close`.
- `SELECT` reads rows; `mysqli_fetch_assoc` hands you one row at a time.
- `INSERT` adds a row; `mysqli_insert_id` gives its new id.
- Never glue user input into a query — use prepared statements with `?` and `bind_param`.
- A class is a blueprint; `new` builds an object; `$this` means this object.

Common mistakes

1. **Forgetting to start MySQL in XAMPP** — the connect call fails with "Connection refused". Fix: press Start next to MySQL in the XAMPP control panel.
2. **Connecting to a database that does not exist yet** — create it in phpMyAdmin first. Fix: `mysqli_connect` with no database, then `CREATE DATABASE school`, or create it in phpMyAdmin.
3. **Gluing `$_POST` into a query string** — `"... VALUES ('" . $_POST["name"] . "')"` invites SQL injection. Fix: prepared statements, every time, no exceptions.
4. **Forgetting to check the query result** — a failed query returns `false`, and the next line crashes on it. Fix: check `if ($result)` before fetching rows.
5. **Writing `$this->marks` as `$this->$marks`** — the extra `$` makes PHP read the value of `$marks` as the property name. Fix: after `->`, the property name has no `$`.
6. **Calling a method as a property** — `$ali->grade` (no brackets) gives you nothing useful. Fix: methods need brackets: `$ali->grade()`.
7. **Forgetting the constructor runs automatically** — putting setup code after `new` instead of in `__construct` leaves objects half-built. Fix: put required setup in the constructor.

Practice

1. Start MySQL in XAMPP, open phpMyAdmin, create a database called `school` and a table `students` (id, name, marks).
2. Write a connect script and print "Connected!" — then deliberately use a wrong database name and read the error.
3. Insert 3 students with plain INSERT, then read them all back with SELECT and a while loop.
4. Rewrite the insert with a prepared statement and `bind_param("si", ...)` — test it with a name containing a quote, like `O'Brien`.

5. Deliberately type `'; DROP TABLE students; --` as a name into the prepared statement version — confirm the table survives, and write down in one line why.
6. Write a `Student` class with `__construct`, a `grade()` method and a `passed()` method — build two students and print each one's grade.
7. Add a `total()` function outside the class that takes an array of Student objects and returns the sum of their marks — test it with your two students.

Practice quiz

1. What does mysqli_fetch_assoc($result) give you?

- All rows at once
- One row at a time as an associative array, and null when the rows run out
- The number of rows
- An error message

Answer: One row at a time as an associative array, and null when the rows run out

2. Why use a prepared statement instead of gluing $_POST into the query?

- It is faster to type
- Glued-in user input can become SQL code (SQL injection); placeholders keep it as pure data
- Prepared statements cannot save data
- It is only a style habit

Answer: Glued-in user input can become SQL code (SQL injection); placeholders keep it as pure data

3. What is the difference between a class and an object?

- They are the same thing
- A class is the blueprint, an object is one real thing built from it
- A class is one thing, an object is the blueprint
- A class holds data only, an object holds code only

Answer: A class is the blueprint, an object is one real thing built from it

---

Watch the Code Scope channel: https://www.youtube.com/@CodeScopePK — this lesson's own video joins the channel when it is recorded.

Read online: https://scopetreasure.com/notes/ph-mysql-oop · More free tools and notes at https://scopetreasure.com