PHP SERIES · COURSE 2

PHP Intermediate

OOP, exceptions, PDO and databases, sessions and login systems, namespaces and Composer, JSON/APIs, file uploads, regular expressions - capped with a multi-file blog capstone. 10 complete chapters, each with coding challenges and solutions.

10 Chapters

osztromok.com

CHAPTER 1: OOP BASICS - CLASSES, OBJECTS, PROPERTIES, METHODS

COURSE 2 · CH 1

OOP Basics: Classes, Objects, Properties, and Methods

Moving from standalone functions and arrays to bundling data and behaviour together

Throughout PHP Fundamentals, data (variables, arrays) and behaviour (functions) were kept separate — a function received whatever data it needed as parameters. Object-Oriented Programming (OOP) groups related data and the functions that operate on it into a single unit called an **object**. This chapter introduces the core vocabulary: classes, objects, properties, and methods.

Defining a Class

```php
<?php
  class Person {
      public string $name;
      public int $age;

      public function greet() {
          echo "Hi, I'm {$this->name}!";
      }
  }
?>
```

A **class** is a blueprint — it describes what kind of data a thing has (properties: `$name`, `$age`) and what it can do (methods: `greet()`). No actual person exists yet — the class only defines the shape.

Creating Objects — Instances of a Class

```php
<?php
  $person1 = new Person();
  $person1->name = "Philip";
```

```php
    $person1->age = 35;

    $person2 = new Person();
    $person2->name = "Sam";
    $person2->age = 28;

    $person1->greet();   // "Hi, I'm Philip!"
    $person2->greet();   // "Hi, I'm Sam!"
?>
```

`new Person()` creates an object (also called an "instance") from the `Person` blueprint. Each object has its own independent copy of the properties — changing `$person1->age` has no effect on `$person2`, even though both came from the same class.

One class, many independent objects — each with its own property values

$this — Referring to "the Current Object"

$THIS ONLY MAKES SENSE INSIDE A METHOD, AND REFERS TO WHICHEVER OBJECT CALLED IT

Inside `greet()`, `$this->name` means "the `$name` property belonging to whichever object this method was called on." When `$person1->greet()` runs, `$this` refers to `$person1`; when `$person2->greet()` runs, `$this` refers to `$person2` instead — the exact same method code, but it acts on different data each time.

The Constructor — Setting Up an Object on Creation

```php
<?php
  class Person {
      public string $name;
      public int $age;

      public function __construct(string $name, int $age) {
          $this->name = $name;
          $this->age = $age;
      }

      public function greet() {
          echo "Hi, I'm {$this->name}!";
      }
  }

  $person1 = new Person("Philip", 35);  // properties set immediately, no separate assignm
  $person1->greet();
?>
```

`__construct()` is a special method PHP runs automatically every time `new ClassName(...)` is called. It removes the need to set every property manually on a separate line after creating the object — and, crucially, it can require certain values to always be provided up front (a constructor with required parameters cannot be skipped).

CONSTRUCTOR PROPERTY PROMOTION — A SHORTER, VERY COMMON MODERN SHORTHAND

`public function __construct(public string $name, public int $age) {}` — adding the visibility keyword directly in the constructor's parameter list automatically creates and assigns the property, removing the need to declare it separately above and write the `$this->name = $name;` line. Both styles are shown across real PHP code; this course uses the explicit version above for clarity while the concept is still new.

public, private, and protected — Controlling Access

```php
<?php
  class BankAccount {
      private float $balance = 0;   // cannot be accessed directly from outside this class

      public function deposit(float $amount) {
```

```php
            $this->balance += $amount;
        }

        public function getBalance(): float {
            return $this->balance;
        }
    }

    $account = new BankAccount();
    $account->deposit(100);
    echo $account->getBalance();  // 100
    // $account->balance = 999999;  // ERROR — balance is private, cannot be set directly fr
?>
```

`private` properties can only be read or changed by code inside the same class — forcing any change to go through controlled methods like `deposit()`, rather than letting outside code set `$balance` to anything at all. `public` means accessible from anywhere; `protected` (covered fully in Chapter 2 with inheritance) sits in between.

THIS DELIBERATE RESTRICTION IS CALLED ENCAPSULATION, AND IT'S ONE OF OOP'S CENTRAL IDEAS

Without `private`, nothing would stop other code from setting a bank balance to a negative number, or any arbitrary value, bypassing whatever rules `deposit()` was meant to enforce. Making the property private and only exposing it through methods means the class itself can guarantee its own data stays valid.

Coding Challenges

CHALLENGE 1

Create a Book class with public properties $title, $author, and a constructor that sets both. Add a method describe() that echoes "[title] by [author]". Create two different Book objects and call describe() on each.

View solution

CHALLENGE 2

Create a Counter class with a private property $count starting at 0, and public methods increment() (adds 1), decrement() (subtracts 1), and getCount(). Create one Counter, call increment() three times and decrement() once, then echo the result via getCount().

View solution

CHALLENGE 3

Create a Rectangle class with a constructor taking $width and $height, and methods getArea() and getPerimeter(). Create two rectangles with different dimensions and echo both calculated values for each, demonstrating they're independent of one another.

View solution

CHAPTER 1 QUICK REFERENCE

- **class Name { }** — a blueprint defining properties (data) and methods (behaviour)
- **new ClassName()** — creates an object (instance) from the class blueprint
- **$this** — inside a method, refers to whichever object the method was called on
- **__construct()** — special method run automatically when an object is created via new
- **public / private / protected** — control where a property/method can be accessed from
- **Encapsulation** — using private + controlled methods to keep an object's data valid
- **Next chapter:** OOP in depth — inheritance, interfaces, abstract classes, traits

COURSE 2 · CH 2

OOP In Depth: Inheritance, Interfaces, Abstract Classes, and Traits

Four tools for sharing structure and behaviour between related classes, without copy-pasting code

Chapter 1 built single, standalone classes. Real applications usually have several related classes that share common structure — a `Dog` and a `Cat` are both `Animal`s, for instance. This chapter covers four distinct tools PHP offers for expressing that kind of relationship, each suited to a slightly different situation.

Inheritance — extends

```php
<?php
  class Animal {
      public function __construct(protected string $name) {}

      public function eat() {
          echo "{$this->name} is eating.<br>";
      }

      public function makeSound() {
          echo "Some generic animal sound.<br>";
      }
  }

  class Dog extends Animal {
      public function makeSound() {   // overrides the parent's version
          echo "{$this->name} says Woof!<br>";
      }
  }

  $dog = new Dog("Rex");
  $dog->eat();         // "Rex is eating." — inherited from Animal, unchanged
  $dog->makeSound();   // "Rex says Woof!" — Dog's own override is used instead
?>
```

`Dog extends Animal` means every `Dog` automatically gets everything `Animal` defines — `eat()` didn't need to be rewritten at all. Re-defining `makeSound()` inside `Dog` is called **overriding** — PHP always uses the most specific version available for whichever object the method is called on.

Both Dog and Cat inherit eat() from Animal, but each overrides makeSound() with its own behaviour

parent:: — Calling the Parent's Version Anyway

```php
<?php
  class Dog extends Animal {
      public function makeSound() {
          parent::makeSound();   // runs Animal's original version first
          echo "...and also Woof!<br>";
      }
  }
?>
```

An override doesn't have to fully replace the parent's behaviour — `parent::methodName()` calls the original version too, useful for "do the normal thing, plus something extra."

Abstract Classes — A Blueprint That Can't Be Used Directly

```php
<?php
  abstract class Shape {
      abstract public function getArea(): float;   // no body — every subclass MUST provi

      public function describe() {
          echo "Area: " . $this->getArea() . "<br>";
      }
  }
```

```
class Circle extends Shape {
    public function __construct(private float $radius) {}

    public function getArea(): float {
        return pi() * $this->radius ** 2;
    }
}

// $shape = new Shape();  // FATAL ERROR — cannot instantiate an abstract class
$circle = new Circle(5);
$circle->describe();   // "Area: 78.539..."
?>
```

An **abstract class** can mix concrete methods (`describe()` , fully written) with abstract ones (`getArea()` , no body — just a required signature). It cannot be instantiated directly with `new` ; it exists purely to be extended, guaranteeing every subclass implements the methods marked `abstract` .

Interfaces — A Pure Contract, No Implementation At All

```
<?php
interface Sellable {
    public function getPrice(): float;
}

class Book implements Sellable {
    public function __construct(private float $price) {}

    public function getPrice(): float {
        return $this->price;
    }
}

function printPrice(Sellable $item) {   // accepts ANY class that implements Sellable
    echo "£" . $item->getPrice();
}

printPrice(new Book(9.99));
?>
```

An **interface** defines method signatures only — no bodies at all, not even partial ones. A class `implements` an interface to promise it provides those methods. The real power: `printPrice()` doesn't care whether it receives a `Book`, a `Car`, or anything else — only that it implements `Sellable`, so it definitely has a `getPrice()` method.

TOOL	WHAT IT GIVES YOU
`extends`	A class inherits from one parent class (single inheritance only)
`abstract class`	Partial blueprint — some methods written, some left required-but-empty
`interface`	Pure contract — method signatures only, a class can implement many
`trait`	Reusable method code, mixed into a class — covered next

> **A CLASS CAN IMPLEMENT MULTIPLE INTERFACES, BUT EXTEND ONLY ONE CLASS**
>
> PHP supports "single inheritance" — a class has exactly one direct parent via `extends`. But `implements Sellable, Comparable, Loggable` (comma-separated) is entirely valid, since interfaces are just promises about method signatures, not actual shared code to merge.

Traits — Sharing Actual Method Code, Without Inheritance

```php
<?php
  trait Loggable {
      public function log(string $message) {
          echo "[LOG] $message<br>";
      }
  }

  class Order {
      use Loggable;   // pulls log() directly into this class
  }

  class User {
      use Loggable;   // completely unrelated class, also gets log() this way
  }

  (new Order())->log("Order placed");
  (new User())->log("User registered");
?>
```

`Order` and `User` have no inheritance relationship to each other at all — they're unrelated classes that both happen to need logging behaviour. A **trait** solves exactly this: actual, reusable method code that gets copied into any class using `use TraitName;`, sidestepping the "only one parent class" limit of `extends`.

CHOOSING BETWEEN THESE FOUR TOOLS

"Is-a" relationship with shared state and behaviour (a Dog IS an Animal) → `extends`. Want to guarantee a method exists, with no implementation to share → `interface`. Want a partial blueprint mixing some shared code with some required-but-unwritten methods → `abstract class`. Need to share actual method code across otherwise-unrelated classes → `trait`.

Coding Challenges

CHALLENGE 1

Create an Animal class with a constructor setting $name, and a method makeSound() that echoes a generic message. Create Cat and Cow classes that extend it and override makeSound() with their own sound. Create one of each and call makeSound() on both.

View solution

CHALLENGE 2

Create an abstract class Employee with a constructor setting $name, an abstract method calculatePay(): float, and a concrete method describe() that echoes the name and calculated pay together. Create two subclasses, SalariedEmployee and HourlyEmployee, each implementing calculatePay() differently.

View solution

CHALLENGE 3

Create a trait Greetable with a method sayHello() that echoes "Hello from " followed by a $name property. Use this trait in two unrelated classes, Robot and Alien (neither extending the other, or any common parent), each with their own $name set via a constructor. Call sayHello() on one instance of each.

View solution

CHAPTER 2 QUICK REFERENCE

- **extends** — single inheritance; subclass gets everything the parent has, can override methods
- **parent::method()** — calls the parent's original version from inside an override
- **abstract class** — partial blueprint; cannot be instantiated; abstract methods MUST be implemented by subclasses
- **interface** — pure method signatures, no bodies; a class can implement several
- **trait** — reusable method code, mixed into otherwise unrelated classes via "use"
- **Decision guide:** is-a → extends; guaranteed method, no shared code → interface; partial shared blueprint → abstract class; shared code across unrelated classes → trait
- **Next chapter:** error handling — exceptions, try/catch/finally, custom exceptions

CHAPTER 3: ERROR HANDLING - EXCEPTIONS, TRY/CATCH/FINALLY

COURSE 2 · CH 3

Error Handling: Exceptions, try/catch/finally, Custom Exceptions

Handling things going wrong predictably, instead of letting a script crash or silently produce wrong results

Every chapter so far has assumed inputs are valid and operations succeed. Real code constantly deals with things that can fail — a file that doesn't exist, a division by zero, invalid data passed to a function. PHP's exception system gives a structured way to detect these failures and decide what to do about them, rather than letting the script crash outright or silently continue with bad data.

Throwing an Exception

```php
<?php
  function divide(float $a, float $b): float {
      if ($b === 0.0) {
          throw new Exception("Cannot divide by zero");
      }
      return $a / $b;
  }

  echo divide(10, 0);   // uncaught — fatal error, script stops here
?>
```

`throw new Exception("message")` signals that something has gone wrong, immediately halting normal execution at that point. An uncaught exception — one that isn't handled anywhere — stops the entire script with a fatal error, similar to a crash.

Catching an Exception — try/catch

```php
<?php
  try {
```

```
    echo divide(10, 0);
} catch (Exception $e) {
    echo "Something went wrong: " . $e->getMessage();
}

echo "Script keeps running after this point.";   // this line DOES still run
?>
```

Code that might throw an exception goes inside a `try` block. If an exception is thrown anywhere inside it, execution immediately jumps to the matching `catch` block instead of crashing — `$e->getMessage()` retrieves the message that was passed to `throw new Exception(...)`. Critically, the script continues running normally after the try/catch finishes.

try → (if it throws) catch → finally always runs either way → script continues normally

finally — Always Runs, Exception or Not

```
<?php
  try {
      echo divide(10, 0);
  } catch (Exception $e) {
      echo "Caught: " . $e->getMessage() . "<br>";
  } finally {
      echo "Cleanup runs no matter what.<br>";   // runs whether or not an exception was t
```

```
    }
?>
```

`finally` runs whether the `try` succeeded, failed and was caught, or even if an exception inside `catch` itself wasn't fully handled — genuinely useful for cleanup work that absolutely must happen either way, such as closing a file or a database connection.

Custom Exceptions — More Specific Than the Built-In Exception

```
<?php
    class InsufficientFundsException extends Exception {}

    class BankAccount {
        private float $balance = 0;

        public function withdraw(float $amount) {
            if ($amount > $this->balance) {
                throw new InsufficientFundsException("Cannot withdraw £$amount, balance is £
            }
            $this->balance -= $amount;
        }
    }

    $account = new BankAccount();
    try {
        $account->withdraw(50);
    } catch (InsufficientFundsException $e) {
        echo "Withdrawal failed: " . $e->getMessage();
    }
?>
```

Custom exceptions (extending the built-in `Exception` class — Chapter 2's inheritance, applied here) give catch blocks the ability to react differently depending on exactly what kind of problem occurred, rather than every failure being a generic, indistinguishable "Exception."

Catching Multiple, Different Exception Types

```php
<?php
  try {
      $account->withdraw(50);
  } catch (InsufficientFundsException $e) {
      echo "Not enough money: " . $e->getMessage();
  } catch (Exception $e) {
      echo "Some other problem: " . $e->getMessage();   // catches anything else
  }
?>
```

ORDER MATTERS — PHP CHECKS CATCH BLOCKS TOP TO BOTTOM, USING THE FIRST ONE THAT MATCHES

Since `InsufficientFundsException` IS an `Exception` (it extends it), placing the generic `catch (Exception $e)` block first would catch everything — including `InsufficientFundsException` — before the more specific block ever gets a chance to run. Always list the most specific exception types first, generic ones last.

PHP's Built-In Exception Hierarchy (a Glimpse)

```
Throwable (the root interface)
├─ Exception — the general-purpose base class, extend this for custom exceptions
│    ├─ InvalidArgumentException
│    ├─ RuntimeException
│    └─ (your own custom exceptions go here too)
└─ Error — internal PHP engine errors (TypeError, DivisionByZeroError) — catchable,
    but generally indicates a programming bug rather than an expected failure
```

PHP actually distinguishes `Exception` (expected, recoverable problems you anticipate and throw deliberately) from `Error` (engine-level problems, like calling a method that doesn't exist) — both are catchable, but this course focuses on `Exception` as the type to throw deliberately in your own code.

Coding Challenges

CHALLENGE 1

Write a function squareRoot($n) that throws an Exception if $n is negative (square roots of negative numbers aren't real numbers), otherwise returns sqrt($n). Call it inside a try/catch with both a valid and an invalid value, echoing either the result or the caught error message.

View solution

CHALLENGE 2

Create a custom exception InvalidAgeException extending Exception. Write a function setAge($age) that throws it if age is below 0 or above 130, otherwise echoes "Age set to $age." Test it with a try/catch/finally, where the finally block always echoes "Validation attempt complete."

View solution

CHALLENGE 3

Create two custom exceptions, FileNotFoundException and PermissionDeniedException, both extending Exception. Write a function openFile($filename, $hasPermission) that throws the first if $filename is an empty string, the second if $hasPermission is false, otherwise echoes "File opened successfully." Use a try block with TWO separate catch blocks (correctly ordered) to handle each case distinctly, with a generic Exception catch as a final fallback.

View solution

CHAPTER 3 QUICK REFERENCE

- **throw new Exception("message")** — signals a problem, halts normal execution at that point
- **try { } catch (Exception $e) { }** — catches a thrown exception, script continues afterward
- **$e->getMessage()** — retrieves the message passed when the exception was thrown
- **finally { }** — always runs, exception or not; for cleanup code
- **Custom exceptions** — extend Exception to create specific, distinguishable error types
- **Multiple catch blocks** — list most specific exception types first, generic Exception last
- **Exception vs Error** — Exception for expected/anticipated problems you throw deliberately; Error for engine-level bugs

- **Next chapter:** working with databases — PDO, prepared statements, basic CRUD

CHAPTER 4: WORKING WITH DATABASES - PDO, PREPARED STATEMENTS, CRUD

COURSE 2 · CH 4

Working With Databases: PDO, Prepared Statements, and Basic CRUD

Connecting PHP to MySQL safely, and the four operations every data-driven page needs

Every example so far has worked with data that disappears the moment the script finishes. Real applications store data persistently in a database. This chapter introduces **PDO** (PHP Data Objects) — PHP's standard way of talking to a database — and **CRUD**: Create, Read, Update, Delete, the four operations behind almost every data-driven feature.

Connecting With PDO

```php
<?php
  $host = "localhost";
  $db   = "my_database";
  $user = "db_user";
  $pass = "secret_password";

  try {
      $pdo = new PDO("mysql:host=$host;dbname=$db;charset=utf8mb4", $user, $pass);
      $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
  } catch (PDOException $e) {
      die("Connection failed: " . $e->getMessage());
  }
?>
```

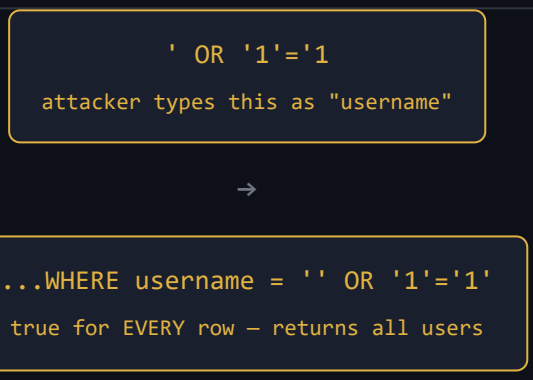
A PDO connection failure throws a `PDOException` — exactly the exception handling from Chapter 3, applied immediately to something genuinely likely to fail in the real world (wrong credentials, database server down). `ERRMODE_EXCEPTION` makes every later database error throw an exception too, rather than failing silently.

NEVER WRITE A DATABASE PASSWORD DIRECTLY IN A FILE INSIDE A PUBLIC WEB FOLDER

The example above uses a plain variable for clarity. In a real project, credentials belong in a separate config file stored outside the publicly accessible web root, or in environment variables — never committed to git, and never reachable by a direct URL request.

The SQL Injection Problem

```php
<?php
  // DO NOT DO THIS — directly inserting user input into SQL text
  $username = $_GET['username'];
  $sql = "SELECT * FROM users WHERE username = '$username'";
  $pdo->query($sql);   // dangerously vulnerable
?>
```



Concatenating user input straight into SQL text lets an attacker change the query's actual meaning

SQL INJECTION IS ONE OF THE MOST SERIOUS, WELL-KNOWN WEB VULNERABILITIES — AND IT'S AVOIDED ENTIRELY WITH THE TECHNIQUE BELOW

By submitting carefully crafted text instead of a normal username, an attacker can change what the SQL statement actually does — bypassing a login check, or reading data they shouldn't see. This is exactly why raw string concatenation into SQL must never be done with any value that came from user input.

Prepared Statements — The Fix

```php
<?php
  $username = $_GET['username'] ?? '';
  $stmt = $pdo->prepare("SELECT * FROM users WHERE username = :username");
  $stmt->execute(['username' => $username]);
```

```
  $user = $stmt->fetch();
?>
```

`:username` is a **placeholder** — the actual value is sent to the database completely separately from the SQL text itself, in a second step (`execute()`). The database treats the placeholder's value strictly as data, never as part of the SQL command's structure — meaning the attacker's text from above could never be reinterpreted as part of the query's logic. This single habit eliminates SQL injection entirely.

CRUD — The Four Core Operations

Create	Read	Update	Delete
INSERT INTO ...	SELECT ... FROM ...	UPDATE ... SET ...	DELETE FROM ...

Create — INSERT

```
<?php
  $stmt = $pdo->prepare("INSERT INTO posts (title, body) VALUES (:title, :body)");
  $stmt->execute(['title' => "My First Post", 'body' => "Hello, world!"]);
  echo "New post ID: " . $pdo->lastInsertId();
?>
```

Read — SELECT

```
<?php
  $stmt = $pdo->prepare("SELECT * FROM posts WHERE id = :id");
  $stmt->execute(['id' => 1]);
  $post = $stmt->fetch();         // one row, as an associative array (Chapter 6)

  $allPosts = $pdo->query("SELECT * FROM posts")->fetchAll();   // every row, as an array
  foreach ($allPosts as $post) {
      echo $post['title'] . "<br>";
  }
?>
```

Update — UPDATE

```php
<?php
  $stmt = $pdo->prepare("UPDATE posts SET title = :title WHERE id = :id");
  $stmt->execute(['title' => "Updated Title", 'id' => 1]);
?>
```

Delete — DELETE

```php
<?php
  $stmt = $pdo->prepare("DELETE FROM posts WHERE id = :id");
  $stmt->execute(['id' => 1]);
?>
```

UPDATE AND DELETE WITHOUT A WHERE CLAUSE AFFECT EVERY ROW IN THE TABLE

Forgetting the `WHERE id = :id` condition on an UPDATE or DELETE statement is a genuinely common, genuinely serious mistake — it silently applies the change to the entire table rather than the one row intended. Always double-check a WHERE clause is present before running either statement against a real database.

fetchAll() Fetch Modes

By default, PDO returns rows as both numeric AND associative keys combined — usually more than needed. Setting the fetch mode once after connecting keeps results clean and predictable:

```php
<?php
  $pdo->setAttribute(PDO::ATTR_DEFAULT_FETCH_MODE, PDO::FETCH_ASSOC);
  // from now on, every fetch()/fetchAll() returns clean associative arrays only
?>
```

Coding Challenges

CHALLENGE 1

Write the PDO connection code for a database called "blog_db" on localhost, wrapped in a try/catch that dies with a friendly message on failure. Set both ERRMODE_EXCEPTION and FETCH_ASSOC as attributes.

View solution

CHALLENGE 2

Assuming a "products" table with columns id, name, and price, write prepared statements (with placeholders) for: inserting a new product, selecting all products with a price above a given value, and updating a specific product's price by id.

View solution

CHALLENGE 3

Write a function deleteProductById($pdo, $id) that uses a prepared DELETE statement, wrapped in a try/catch for PDOException, echoing a success or failure message. Explain in a comment why this function would be unsafe if $id were inserted directly into the SQL string instead of used as a placeholder.

View solution

CHAPTER 4 QUICK REFERENCE

- **new PDO("mysql:host=...;dbname=...", $user, $pass)** — connect; wrap in try/catch for PDOException
- **ATTR_ERRMODE / ERRMODE_EXCEPTION** — makes database errors throw exceptions instead of failing silently
- **Never concatenate user input into SQL strings** — this is SQL injection, a serious vulnerability
- **prepare() + execute([...])** — prepared statements with named (:placeholder) parameters fix this entirely
- **CRUD:** INSERT (Create), SELECT (Read), UPDATE, DELETE
- **fetch() / fetchAll()** — one row / all rows, as associative arrays with FETCH_ASSOC set
- **lastInsertId()** — gets the auto-increment ID of a just-inserted row
- Always include a WHERE clause on UPDATE/DELETE, or risk affecting every row
- **Next chapter:** sessions and cookies — login systems, authentication basics

CHAPTER 5: SESSIONS AND COOKIES - LOGIN SYSTEMS, AUTHENTICATION

COURSE 2 · CH 5

Sessions and Cookies: Login Systems and Authentication Basics

Remembering a visitor across multiple page loads — and building a simple, real login flow

HTTP is "stateless" — by default, PHP has no memory of one request to the next; every page load is treated as a brand-new, unrelated visitor. Sessions and cookies are the two main tools that let a site remember who's logged in, what's in a shopping cart, or any other state that needs to persist across page loads.

Cookies — Small Bits of Data Stored in the Browser

```php
<?php
  setcookie("username", "Philip", time() + 3600);   // expires in 1 hour

  // On a LATER page load (after the browser sends the cookie back):
  echo $_COOKIE['username'] ?? 'Guest';
?>
```

A COOKIE SET WITH SETCOOKIE() IS ONLY AVAILABLE STARTING FROM THE NEXT REQUEST

`setcookie()` must run before any HTML output begins (it sets an HTTP header), and the cookie won't appear in `$_COOKIE` until the browser sends it back on a subsequent page load — checking `$_COOKIE['username']` immediately after setting it on the same page will not work as expected.

Sessions — Server-Side Storage, Linked by One Cookie

login.php

```php
<?php
  session_start();   // MUST be the very first thing, before any output
  $_SESSION['username'] = "Philip";
```

```
  $_SESSION['loggedIn'] = true;
?>
```

a later page, in a separate request

```
<?php
  session_start();   // must run on EVERY page that needs to read/write $_SESSION
  if ($_SESSION['loggedIn'] ?? false) {
      echo "Welcome back, " . $_SESSION['username'];
  }
?>
```

`session_start()` either creates a new session or resumes an existing one, identified by a single cookie PHP manages automatically (usually called `PHPSESSID`) — the actual data lives on the server, not in the browser, which is exactly why sessions are the right tool for anything sensitive, like login state.

Cookies

- Stored in the visitor's browser
- Visitor can view/edit/delete them directly
- Good for: non-sensitive preferences (theme, language)

Sessions

- Stored on the server; only an ID cookie sits in the browser
- Visitor cannot read or tamper with the actual data
- Good for: login state, anything sensitive

A Simple, Complete Login Flow

login.php

```
<?php
  session_start();
  if ($_SERVER['REQUEST_METHOD'] === 'POST') {
      $username = $_POST['username'] ?? '';
      $password = $_POST['password'] ?? '';

      // Imagine $storedHash came from a database lookup by username (Chapter 4)
      $storedHash = password_hash("correct-password", PASSWORD_DEFAULT);   // example only

      if (password_verify($password, $storedHash)) {
```

```
            $_SESSION['loggedIn'] = true;
            $_SESSION['username'] = $username;
            header("Location: dashboard.php");
            exit;
        } else {
            $error = "Invalid username or password.";
        }
    }
?>
```

NEVER STORE OR COMPARE PLAIN-TEXT PASSWORDS

`password_hash()` turns a password into a scrambled, one-way string before it's ever stored — even if a database is somehow exposed, the original passwords cannot be recovered from the hashes. `password_verify($plainPassword, $storedHash)` checks a login attempt against that hash without ever needing to "decrypt" it back to plain text — because it genuinely cannot be reversed by design.

Protecting a Page — Requiring Login

dashboard.php

```
<?php
  session_start();
  if (!($_SESSION['loggedIn'] ?? false)) {
      header("Location: login.php");
      exit;   // CRITICAL — stops the rest of the page from running
  }
  echo "Welcome to your dashboard, " . $_SESSION['username'] . "!";
?>
```

HEADER("LOCATION: ...") WITHOUT EXIT AFTERWARD IS A COMMON, SERIOUS MISTAKE

`header()` sends a redirect instruction to the browser, but PHP itself keeps executing the rest of the script unless `exit` (or `die()`) is called immediately after — without it, a logged-out visitor could still see page content that was meant to be protected, briefly, before the redirect actually takes effect in their browser.

Logging Out

```php
<?php
  session_start();
  $_SESSION = [];                 // clear all session data
  session_destroy();              // destroy the session itself on the server
  header("Location: login.php");
  exit;
?>
```

Coding Challenges

CHALLENGE 1

Write a script that uses session_start() and checks if $_SESSION['visits'] exists — if not, set it to 1; if it does, increment it by 1. Echo "You have visited this page X times." Reload the same page (conceptually) and explain why the count keeps increasing.

View solution

CHALLENGE 2

Using password_hash() and password_verify(), write a script that hashes the password "MySecret123", then checks two test inputs — one matching, one not — against the hash, echoing whether each one would be allowed to log in.

View solution

CHALLENGE 3

Write a small "protected page" pattern: a function requireLogin() that checks $_SESSION['loggedIn'], redirecting to "login.php" with header() + exit if not set. Then write a logout script that clears the session and destroys it. Explain in a comment why exit is essential right after the header() redirect call.

View solution

CHAPTER 5 QUICK REFERENCE

- **setcookie(name, value, expiry)** — browser-stored, visitor-editable; for non-sensitive data only
- **session_start()** — must run before ANY output, on every page using $_SESSION
- **$_SESSION[...]** — server-side storage, linked by one cookie; safe for login state
- **password_hash() / password_verify()** — never store or compare plain-text passwords
- **header("Location: ...") + exit** — redirect; exit is essential to stop the rest of the script running
- **session_destroy() + clearing $_SESSION** — the logout pattern
- **Next chapter:** namespaces and autoloading — PSR-4, Composer basics

CHAPTER 6: NAMESPACES AND AUTOLOADING - PSR-4, COMPOSER BASICS

COURSE 2 · CH 6

Namespaces and Autoloading: PSR-4, Composer Basics

Avoiding class-name collisions, and letting PHP find and load classes automatically instead of writing require_once everywhere

Chapter 9 of the Fundamentals course used `require_once` to manually pull in every file defining a class or function. That works fine for a handful of files, but becomes unmanageable in a larger project with dozens of classes. Namespaces and autoloading — almost always set up via Composer — solve this properly, and are the standard approach in essentially all modern PHP projects.

The Problem: Class Name Collisions

```php
<?php
  // library_a.php defines:
  class Logger { /* ... */ }

  // library_b.php ALSO defines:
  class Logger { /* ... */ }   // fatal error if both files are loaded — "Cannot redeclare
?>
```

If two different libraries both happen to define a class called `Logger`, including both in the same project is a fatal error. With many third-party packages in use, name collisions like this become a genuine, recurring problem without some way to keep names separated.

Namespaces — Solving the Collision Problem

src/Acme/Logger.php

```php
<?php
  namespace Acme;
```

```
    class Logger {
        public function log(string $message) {
            echo "[Acme] $message<br>";
        }
    }
?>
```

src/Vendor/Logger.php

```
<?php
    namespace Vendor;

    class Logger {
        public function log(string $message) {
            echo "[Vendor] $message<br>";
        }
    }
?>
```

index.php

```
<?php
    use Acme\Logger as AcmeLogger;
    use Vendor\Logger as VendorLogger;

    $logger1 = new AcmeLogger();
    $logger2 = new VendorLogger();

    $logger1->log("Hello");   // "[Acme] Hello"
    $logger2->log("Hello");   // "[Vendor] Hello"
?>
```

Both classes are genuinely named `Logger`, but their full, unambiguous identities are `Acme\Logger` and `Vendor\Logger` — exactly like how two different files on a computer can both be called `notes.txt` as long as they live in different folders. `use ... as ...` imports a class under a shorter or renamed alias for convenience within one file.

Autoloading — No More require_once for Every Class

Namespaces solve naming collisions, but PHP still needs to know where a class's file actually is when `new Acme\Logger()` is used. **Autoloading** registers a function that PHP calls automatically the first time an unknown class name is referenced — that function works out the file path and includes it, with no manual `require` needed anywhere.

PSR-4 — The Standard Convention

PSR-4 is a widely-adopted convention mapping namespaces directly onto folder structures, so an autoloader can calculate a file's path purely from a class's namespace and name, without any manual configuration per class.

```
my-project/
├── src/
│   └── Acme/
│       ├── Logger.php          — namespace Acme; class Logger
│       └── Database/
│           └── Connection.php  — namespace Acme\Database; class Connection
└── composer.json
```

The namespace `Acme\Database\Connection` maps directly onto the path `src/Acme/Database/Connection.php` — predictable enough that an autoloader can compute it instead of being told it explicitly for every single class.

Composer — Installing Packages and Generating the Autoloader

Composer is PHP's standard package manager — it downloads third-party libraries, and (genuinely just as importantly for this chapter) generates a working PSR-4 autoloader for your own project's classes too.

composer.json

```
{
  "autoload": {
    "psr-4": {
      "Acme\\": "src/Acme/"
    }
  }
}
```

```
$ composer dump-autoload
# generates vendor/autoload.php — the single file every entry-point script needs
```

index.php

```php
<?php
  require 'vendor/autoload.php';   // the ONLY require needed — handles every class automa

  use Acme\Logger;

  $logger = new Logger();   // Composer's autoloader finds and includes src/Acme/Logger.ph
  $logger->log("Started");
?>
```

COMPOSER REQUIRE SOME/PACKAGE — INSTALLING A REAL THIRD-PARTY LIBRARY

Beyond autoloading your own code, `composer require monolog/monolog` (for example) downloads a library into a `vendor/` folder and adds it to `composer.json` automatically — the same single `require 'vendor/autoload.php'` line then makes that library's classes available too, alongside your own.

VENDOR/ SHOULD NEVER BE COMMITTED TO GIT

The `vendor/` folder is regenerated entirely from `composer.json` by running `composer install` — committing it bloats a repository unnecessarily. A `.gitignore` entry for `/vendor/` is standard practice in essentially every Composer-managed PHP project.

Coding Challenges

CHALLENGE 1

Create two files, src/Shop/Product.php (namespace Shop; class Product with a $name property) and src/Warehouse/Product.php (namespace Warehouse; class Product with a $stockLevel property). In a third file, use both classes with aliases via use ... as ... and create one instance of each, demonstrating no naming collision occurs.

View solution

CHALLENGE 2

Write a composer.json defining a PSR-4 mapping for the namespace prefix "App\\" to a folder "app/". Describe in a comment which command would be run after creating/editing this file, and what it generates.

View solution

CHALLENGE 3

Design (in comments/text, not necessarily runnable) a small project structure with namespace App\Models containing a User class and App\Services containing an EmailService class, following PSR-4 folder conventions. Write the index.php that would use both via Composer's autoloader, with no manual require_once for either class.

View solution

CHAPTER 6 QUICK REFERENCE

- **namespace Acme;** — gives every class in the file a fully-qualified identity (Acme\ClassName), avoiding collisions
- **use Acme\Logger as AcmeLogger;** — imports/aliases a class for convenient use within a file
- **Autoloading** — PHP calls a registered function automatically to find/include a class's file on first use
- **PSR-4** — standard convention mapping namespaces directly onto folder structures
- **composer.json "autoload"/"psr-4"** — declares the namespace-to-folder mapping
- **composer dump-autoload** — (re)generates vendor/autoload.php from that mapping
- **require 'vendor/autoload.php';** — the one require needed; handles every class from then on
- vendor/ should never be committed to git — regenerate via composer install
- **Next chapter:** working with JSON and APIs — json_encode/decode, consuming a REST API with cURL

COURSE 2 · CH 7

Working With JSON and APIs: json_encode/decode, Consuming a REST API With cURL

The data format the web speaks, and how to pull data from someone else's server into your own PHP script

JSON (JavaScript Object Notation) is the standard format for exchanging structured data between systems on the web — virtually every API returns it. This chapter covers converting between PHP arrays and JSON text, then using cURL to actually fetch JSON data from a real external API.

json_encode — PHP Array to JSON Text

```php
<?php
  $person = [
      "name" => "Philip",
      "age" => 35,
      "hobbies" => ["reading", "chess"]
  ];

  echo json_encode($person);
  // {"name":"Philip","age":35,"hobbies":["reading","chess"]}

  echo json_encode($person, JSON_PRETTY_PRINT);   // adds readable indentation
?>
```

`json_encode()` turns a PHP array (Chapter 6 of Fundamentals) into a JSON-formatted string — used constantly when a PHP script needs to send data out, whether as an API response or for storing structured data as text.

json_decode — JSON Text Back to PHP

```php
<?php
  $json = '{"name":"Philip","age":35}';

  $object = json_decode($json);          // returns a stdClass OBJECT by default
  echo $object->name;                   // "Philip" — accessed with -> like any object

  $array = json_decode($json, true);    // "true" here means "return an associative array
  echo $array['name'];                  // "Philip" — accessed with ['key'] instead
?>
```

JSON_DECODE'S SECOND ARGUMENT — EASY TO FORGET, AND CHANGES THE RESULT TYPE COMPLETELY

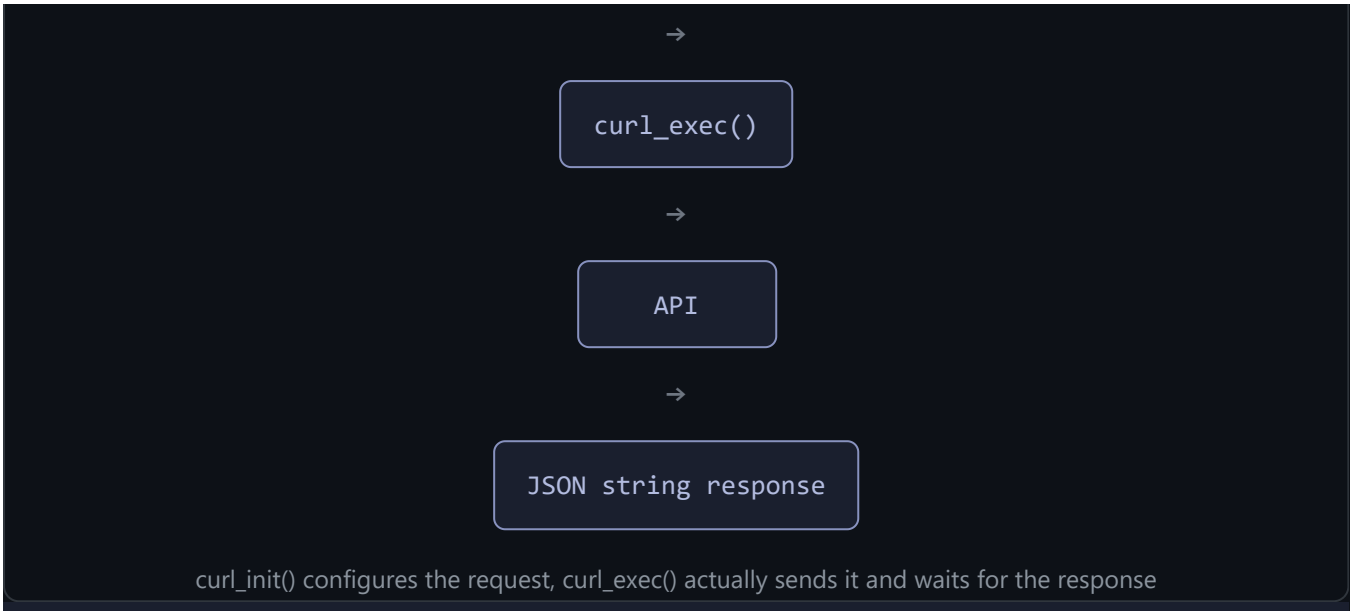
Without the `true` second argument, `json_decode()` returns a generic `stdClass` object, accessed with `->` property syntax. Passing `true` returns a familiar associative array instead, accessed with `['key']` syntax. Most PHP code dealing with API responses uses `true` consistently, simply because arrays are usually more convenient to work with using the tools from Chapter 6 of Fundamentals (foreach, array functions, etc).

cURL — Making an HTTP Request From PHP

```php
<?php
  $ch = curl_init("https://api.example.com/users/1");
  curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);   // return the response as a string, do
  $response = curl_exec($ch);
  $httpCode = curl_getinfo($ch, CURLINFO_HTTP_CODE);
  curl_close($ch);

  if ($httpCode === 200) {
      $user = json_decode($response, true);
      echo "Got user: " . $user['name'];
  } else {
      echo "Request failed with status $httpCode";
  }
?>
```

`curl_init()`



curl_init() configures the request, curl_exec() actually sends it and waits for the response

FUNCTION	WHAT IT DOES
`curl_init($url)`	Starts a new cURL request, targeting a URL
`curl_setopt($ch, OPTION, value)`	Configures the request (return as string, POST data, headers, etc.)
`curl_exec($ch)`	Actually sends the request, returns the response body
`curl_getinfo($ch, CURLINFO_HTTP_CODE)`	Gets the HTTP status code (200, 404, 500, etc.)
`curl_close($ch)`	Frees the cURL resource — good practice once done

Sending Data — A POST Request

```php
<?php
  $data = ["title" => "New Post", "body" => "Hello!"];

  $ch = curl_init("https://api.example.com/posts");
  curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
  curl_setopt($ch, CURLOPT_POST, true);
  curl_setopt($ch, CURLOPT_POSTFIELDS, json_encode($data));
  curl_setopt($ch, CURLOPT_HTTPHEADER, ["Content-Type: application/json"]);

  $response = curl_exec($ch);
  curl_close($ch);
?>
```

Sending data to an API mirrors submitting a form (Chapter 8 of Fundamentals), but the data is JSON-encoded first, and the `Content-Type: application/json` header tells the receiving API exactly what format the body is in.

ALWAYS CHECK THE HTTP STATUS CODE BEFORE TRUSTING A RESPONSE

A request can "succeed" from cURL's perspective (no connection error) while the API itself reports a failure via its status code — checking for `200` (success) or other expected codes, rather than assuming the response is always valid, is good practice exactly like checking `$_SERVER['REQUEST_METHOD']` before trusting `$_POST` in Chapter 8 of Fundamentals.

Coding Challenges

CHALLENGE 1

Create an associative array representing a product (name, price, in_stock as a boolean). Encode it to JSON with json_encode(), echo it, then decode that exact JSON string back into a PHP array with json_decode(), and echo the price from the decoded array to confirm round-tripping worked correctly.

View solution

CHALLENGE 2

Write a function fetchJson($url) that uses cURL to GET a URL, checks the HTTP status code, and returns the decoded JSON as an associative array if the code is 200, or null otherwise. Call it with a placeholder URL and handle both possible outcomes.

View solution

CHALLENGE 3

Write a function postJson($url, $data) that sends an associative array as a JSON-encoded POST request with the correct Content-Type header, and returns the decoded JSON response. Show how it would be called with a sample $data array.

View solution

CHAPTER 7 QUICK REFERENCE

- **json_encode($array)** — PHP array → JSON string; add JSON_PRETTY_PRINT for readable formatting
- **json_decode($json, true)** — JSON string → PHP array (true = associative array, not stdClass)
- **curl_init($url)** → **curl_setopt()** → **curl_exec()** → **curl_close()** — the standard cURL request pattern
- **CURLOPT_RETURNTRANSFER** — returns the response as a string instead of printing it directly
- **curl_getinfo($ch, CURLINFO_HTTP_CODE)** — always check this before trusting a response
- **CURLOPT_POST / CURLOPT_POSTFIELDS** — sending data; pair with json_encode() and the Content-Type header for JSON APIs
- **Next chapter:** file uploads and handling user input safely

COURSE 2 · CH 8

File Uploads and Handling User Input Safely

Accepting files from a visitor without letting them compromise the server, and closing the output-escaping gap left open since Fundamentals

Chapter 8 of Fundamentals flagged, but deliberately deferred, two things: output escaping and proper input validation. This chapter closes both gaps, and adds file uploads — one of the highest-risk features a web form can offer, since it involves a visitor placing an actual file onto the server's disk.

htmlspecialchars() — Closing the XSS Gap

```php
<?php
  $comment = $_POST['comment'] ?? '';

  // UNSAFE — exactly what Fundamentals Chapter 8 warned about:
  echo "You said: $comment";

  // SAFE — escapes HTML special characters before they're ever rendered:
  echo "You said: " . htmlspecialchars($comment);
?>
```

If a visitor submits `<script>alert('hacked')</script>` as their comment, the unsafe version actually runs that script in every other visitor's browser who views the page — a **Cross-Site Scripting (XSS)** attack. `htmlspecialchars()` converts `<`, `>`, `&`, and quotes into their safe HTML entity equivalents, so the text displays literally instead of being interpreted as markup.

ESCAPE ON OUTPUT, NOT ON INPUT

A common mistake is running `htmlspecialchars()` once when data is first received, then storing the escaped version. This causes problems later (double-escaping, or needing the raw value for non-HTML purposes). The standard practice is to store the original input as-is, and escape it specifically at the point it's about to be echoed into HTML — every single time it's displayed.

Validating Input Properly

```php
<?php
  $email = $_POST['email'] ?? '';
  $age = $_POST['age'] ?? '';

  if (!filter_var($email, FILTER_VALIDATE_EMAIL)) {
      echo "Please enter a valid email address.<br>";
  }

  if (!filter_var($age, FILTER_VALIDATE_INT, ["options" => ["min_range" => 0, "max_range"
      echo "Please enter a valid age.<br>";
  }
?>
```

`filter_var()` with PHP's built-in filters handles common validation needs (email format, integer ranges, URLs) far more reliably than hand-written checks — `FILTER_VALIDATE_EMAIL` correctly handles the genuinely complex rules of what counts as a valid email address, something error-prone to get exactly right manually.

File Uploads — The HTML Side

```html
<form action="upload.php" method="post" enctype="multipart/form-data">
  <input type="file" name="avatar">
  <input type="submit">
</form>
```

ENCTYPE="MULTIPART/FORM-DATA" IS EASY TO FORGET, AND SILENTLY BREAKS FILE UPLOADS

Without this attribute on the `<form>` tag, the file's actual contents are never sent to the server at all — `$_FILES` would simply be empty, with no obvious error to explain why.

$_FILES — The PHP Side

```php
<?php
  print_r($_FILES['avatar']);
  // Array
  // (
  //     [name] => photo.jpg          — the original filename from the visitor's computer
  //     [type] => image/jpeg          — claimed type, NOT to be trusted blindly
  //     [tmp_name] => /tmp/phpA1B2C3  — where PHP temporarily stored the uploaded file
  //     [error] => 0                  — 0 means success; any other value means a problem
  //     [size] => 204800               — size in bytes
  // )
?>
```

Handling an Upload Safely

```php
<?php
  $allowedTypes = ['image/jpeg', 'image/png'];
  $maxSize = 2 * 1024 * 1024;   // 2 MB

  if ($_FILES['avatar']['error'] !== UPLOAD_ERR_OK) {
      echo "Upload failed.<br>";
  } elseif ($_FILES['avatar']['size'] > $maxSize) {
      echo "File too large (max 2MB).<br>";
  } else {
      $actualType = mime_content_type($_FILES['avatar']['tmp_name']);   // checks the REAL
      if (!in_array($actualType, $allowedTypes)) {
          echo "Invalid file type.<br>";
      } else {
          $newName = uniqid() . '.jpg';   // NEVER trust the original filename directly
          move_uploaded_file($_FILES['avatar']['tmp_name'], "uploads/$newName");
          echo "Upload successful.<br>";
      }
  }
?>
```

$_FILES['AVATAR']['TYPE'] IS CLAIMED BY THE BROWSER — IT CAN BE FAKED ENTIRELY

A malicious visitor can rename a script file to `photo.jpg` and have the browser report `image/jpeg` as the type, despite the actual content being something else entirely (like executable PHP code).

`mime_content_type()` inspects the file's real content rather than trusting the claim — checking the genuine type is essential before deciding whether to accept an upload.

NEVER USE THE VISITOR'S ORIGINAL FILENAME DIRECTLY WHEN SAVING THE FILE

A filename like `../../config.php` could, if used unmodified, attempt to overwrite a completely different file outside the intended uploads folder (a "path traversal" attack). Generating a new, safe filename (as with `uniqid()` above) and discarding the original name entirely avoids this risk altogether.

FUNCTION	WHAT IT DOES
`htmlspecialchars($s)`	Escapes HTML special characters before output — prevents XSS
`filter_var($val, FILTER)`	Validates a value against a built-in rule (email, int range, URL)
`$_FILES['name']`	Information about an uploaded file (name, type, tmp_name, error, size)
`mime_content_type($path)`	Checks the REAL type of a file's content, not the claimed type
`move_uploaded_file($tmp, $dest)`	Moves an upload from its temporary location to its final destination

Coding Challenges

CHALLENGE 1

Write a script that takes a $_POST['feedback'] value containing HTML special characters (test with a string like "Great site!"), and echoes it twice — once unescaped (commented out as unsafe), and once safely with htmlspecialchars(). Explain in a comment what would happen if the unsafe version were used with a real <script> tag instead.

View solution

CHALLENGE 2

Write a function validateRegistration($email, $age) that uses filter_var() to check both, returning an array of error messages (empty if everything is valid). Test it with one fully valid set of inputs and one

fully invalid set.

View solution

CHALLENGE 3

Write a complete safe file-upload handler for a field named "document" that only allows PDF files (application/pdf) up to 5MB, checks the real MIME type with mime_content_type(), generates a new filename with uniqid(), and moves the file into an "uploads/" folder — echoing clear success or failure messages at each validation step.

View solution

CHAPTER 8 QUICK REFERENCE

- **htmlspecialchars($s)** — escape on OUTPUT, every time, to prevent XSS
- **filter_var($val, FILTER_VALIDATE_*)** — reliable built-in validation (email, int range, URL)
- **enctype="multipart/form-data"** — required on the form tag for file uploads to work at all
- **$_FILES['field']** — name, type (untrusted!), tmp_name, error, size
- **mime_content_type($tmp_name)** — checks the file's REAL type, never trust the claimed type
- Never use the original filename directly — generate a new one to avoid path traversal
- **move_uploaded_file($tmp, $dest)** — finalises the upload to its destination folder
- **Next chapter:** regular expressions in PHP — preg_match, preg_replace

CHAPTER 9: REGULAR EXPRESSIONS IN PHP

COURSE 2 · CH 9

Regular Expressions in PHP: preg_match, preg_replace

Pattern matching for text that's more flexible than the exact-match string functions from Fundamentals can handle

Chapter 7 of Fundamentals covered `str_contains()`, `strpos()`, and `str_replace()` — all built around exact, literal text. **Regular expressions** ("regex") describe patterns instead — "a sequence of digits," "an optional plus sign followed by numbers," "anything that looks like a UK postcode" — and PHP's `preg_*` functions test and act on those patterns.

preg_match — Does a Pattern Appear?

```php
<?php
  $text = "My phone number is 07911 123456.";

  if (preg_match('/\d{5} \d{6}/', $text)) {
      echo "Found a UK-style phone number pattern.<br>";
  }
?>
```

The pattern is written between two `/` delimiters. `preg_match()` returns `1` if the pattern is found anywhere in the string, `0` if not — both treated as truthy/falsy correctly in an `if`, similar to how `strpos()` needed care in Chapter 7.

Core Pattern Syntax

PATTERN	MATCHES
`\d`	Any single digit (0-9)
`\w`	Any "word" character (letters, digits, underscore)

PATTERN	MATCHES
`\s`	Any whitespace character (space, tab, newline)
`.`	Any single character at all (except newline by default)
`+`	One or more of the preceding item
`*`	Zero or more of the preceding item
`?`	Zero or one of the preceding item (makes it optional)
`{5}`	Exactly 5 of the preceding item
`[abc]`	Any one of the characters listed inside the brackets
`^ / $`	Start / end of the string

```
/^\d{3}-\d{4}$/
```

^ start \d{3} three digits - literal hyphen \d{4} four digits $ end

Matches "123-4567" exactly — three digits, a hyphen, four digits, nothing more

Capturing Groups — Extracting Parts of a Match

```php
<?php
  $text = "Order #4521 was placed on 2026-06-20.";

  preg_match('/Order #(\d+)/', $text, $matches);
  echo $matches[0];   // "Order #4521" — the whole match
  echo $matches[1];   // "4521" — just the part inside the parentheses

  preg_match('/(\d{4})-(\d{2})-(\d{2})/', $text, $dateParts);
  echo "Year: {$dateParts[1]}, Month: {$dateParts[2]}, Day: {$dateParts[3]}";
?>
```

Parentheses `( )` create a "capturing group" — beyond just confirming a match exists, the third argument (`$matches`, passed by reference) is filled with the whole match at index `0`, and each group's individually captured text at index `1`, `2`, and so on.

preg_match_all — Every Match, Not Just the First

```php
<?php
  $text = "Contact: alice@example.com or bob@test.org";

  preg_match_all('/[\w.]+@[\w.]+/', $text, $matches);
  print_r($matches[0]);   // ["alice@example.com", "bob@test.org"]
?>
```

`preg_match()` stops after the first match anywhere in the string; `preg_match_all()` finds every occurrence, returning them as an array.

preg_replace — Pattern-Based Find and Replace

```php
<?php
  $text = "Call 07911 123456 or 07700 900900.";

  $censored = preg_replace('/\d{5} \d{6}/', '[REDACTED]', $text);
  echo $censored;   // "Call [REDACTED] or [REDACTED]."

  // Using a captured group inside the replacement, with $1, $2, etc.
  $reformatted = preg_replace('/(\d{4})-(\d{2})-(\d{2})/', '$3/$2/$1', "2026-06-20");
  echo $reformatted;   // "20/06/2026"
?>
```

`preg_replace()` mirrors `str_replace()` (Chapter 7 of Fundamentals), but matches by pattern rather than exact text, and replaces every match by default — `$1`, `$2`, `$3` in the replacement string refer back to the captured groups from the pattern itself, genuinely useful for reformatting matched text rather than just deleting or replacing it outright.

> **REGEX SPECIAL CHARACTERS NEED ESCAPING WITH A BACKSLASH IF YOU WANT THEM MATCHED LITERALLY**
>
> Characters like `.`, `+`, `?`, `(`, `)` have special meaning in a pattern. To match a literal period in, say, a domain name, write `\.` — an unescaped `.` would instead match "any character at all," which is usually not the intent and can cause subtly wrong matches that are easy to miss in testing.

Validating an Email Format (Compared to filter_var)

```php
<?php
  $email = "philip@example.com";

  if (preg_match('/^[\w.+-]+@[\w-]+\.[a-zA-Z]{2,}$/', $email)) {
      echo "Looks like a valid email format.<br>";
  }
?>
```

FILTER_VAR(FILTER_VALIDATE_EMAIL) IS STILL THE BETTER CHOICE FOR REAL EMAIL VALIDATION

Email address rules are genuinely more complex than they first appear — Chapter 8's `FILTER_VALIDATE_EMAIL` already handles this thoroughly and correctly. This pattern is shown purely as a realistic regex example; in real code, prefer the built-in filter for anything regex could get subtly wrong, and reach for regex when there's no equivalent built-in filter available.

Coding Challenges

CHALLENGE 1

Write a pattern that matches a UK postcode in the simplified format "AA1 1AA" (two letters, one digit, a space, one digit, two letters). Test it with preg_match() against both a matching and a non-matching string, echoing the result of each.

View solution

CHALLENGE 2

Given a string containing several prices like "Items: £12.50, £3.99, and £100.00", use preg_match_all() with a capturing group to extract just the numeric amounts (without the £ sign) into an array, then use array_sum() to total them.

View solution

CHALLENGE 3

Write a function maskCardNumber($number) that takes a 16-digit card number string and uses preg_replace() with capturing groups to keep only the last 4 digits visible, replacing the rest with

asterisks (e.g. "1234567812345678" becomes "**********5678").

View solution

CHAPTER 9 QUICK REFERENCE

- **preg_match($pattern, $str)** — returns 1 if the pattern is found, 0 if not
- **\d \w \s . + * ? {n} [abc] ^ $** — core pattern building blocks
- **( ) capturing groups** — extract matched parts into $matches[1], [2], etc.
- **preg_match_all()** — finds every match in the string, not just the first
- **preg_replace($pattern, $replacement, $str)** — pattern-based find/replace; $1, $2 reference captured groups
- Escape special characters (., +, ?, etc.) with a backslash to match them literally
- Prefer filter_var() over regex for things like email validation where a reliable built-in filter exists
- **Next chapter:** course capstone — a simple blog with login, CRUD posts, and database storage

CAPSTONE: A SIMPLE BLOG WITH LOGIN, CRUD POSTS, DATABASE STORAGE

COURSE 2 · CAPSTONE

Capstone: A Simple Blog With Login, CRUD Posts, and Database Storage

Bringing together OOP, PDO, sessions, exceptions, and validation into one small, structured application

This capstone combines every Intermediate chapter into one small but genuinely structured blog: a login system, a `Post` class backed by a real database, full CRUD, and proper input/output handling — the shape of a real (if minimal) PHP application.

CH 1-2	CH 3	CH 4	CH 5
OOP — Post, User classes	Exceptions for DB errors	PDO + prepared statements	Sessions, password_hash

CH 6	CH 8	CH 9
require_once / autoload pattern	htmlspecialchars, filter_var	Regex slug generation

Project Structure

```
blog/
├── schema.sql              — users + posts tables
├── db.php                  — PDO connection, shared by every page
├── Post.php                — Post class: CRUD methods
├── exceptions/
│   └── PostNotFoundException.php
├── login.php
├── logout.php
├── index.php               — list all posts (Read)
├── new_post.php            — Create, login required
├── edit_post.php           — Update, login required
└── delete_post.php         — Delete, login required
```

schema.sql — The Database Structure

schema.sql

```
CREATE TABLE users (
  id INT AUTO_INCREMENT PRIMARY KEY,
  username VARCHAR(50) UNIQUE NOT NULL,
  password_hash VARCHAR(255) NOT NULL
);

CREATE TABLE posts (
  id INT AUTO_INCREMENT PRIMARY KEY,
  title VARCHAR(200) NOT NULL,
  slug VARCHAR(200) NOT NULL,
  body TEXT NOT NULL,
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

db.php — The Shared Connection

db.php

```
<?php
  try {
      $pdo = new PDO("mysql:host=localhost;dbname=blog_db;charset=utf8mb4", "db_user", "se
      $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
      $pdo->setAttribute(PDO::ATTR_DEFAULT_FETCH_MODE, PDO::FETCH_ASSOC);
  } catch (PDOException $e) {
      die("Database connection failed.");
  }
?>
```

Post.php — An OOP Wrapper Around CRUD

Post.php

```
<?php
  class PostNotFoundException extends Exception {}
```

```php
class Post {
    public function __construct(private PDO $pdo) {}

    private function slugify(string $title): string {
        $slug = strtolower($title);
        $slug = preg_replace('/[^a-z0-9]+/', '-', $slug);   // Chapter 9's regex, applie
        return trim($slug, '-');
    }

    public function create(string $title, string $body): int {
        $stmt = $this->pdo->prepare("INSERT INTO posts (title, slug, body) VALUES (:tit
        $stmt->execute([
            'title' => $title,
            'slug'  => $this->slugify($title),
            'body'  => $body
        ]);
        return (int)$this->pdo->lastInsertId();
    }

    public function find(int $id): array {
        $stmt = $this->pdo->prepare("SELECT * FROM posts WHERE id = :id");
        $stmt->execute(['id' => $id]);
        $post = $stmt->fetch();

        if (!$post) {
            throw new PostNotFoundException("No post found with ID $id");
        }
        return $post;
    }

    public function all(): array {
        return $this->pdo->query("SELECT * FROM posts ORDER BY created_at DESC")->fetchA
    }

    public function update(int $id, string $title, string $body) {
        $stmt = $this->pdo->prepare("UPDATE posts SET title = :title, body = :body WHERE
        $stmt->execute(['title' => $title, 'body' => $body, 'id' => $id]);
    }

    public function delete(int $id) {
        $stmt = $this->pdo->prepare("DELETE FROM posts WHERE id = :id");
        $stmt->execute(['id' => $id]);
    }
}
?>
```

Every database detail (the SQL, the placeholders, the fetch mode) lives inside `Post` — the rest of the application calls `$post->create(...)`, `$post->find($id)`, etc., without needing to know or repeat any SQL. `find()` throws a custom `PostNotFoundException` (Chapter 3's pattern) rather than silently returning something misleading like `false`.

new_post.php — A Protected Create Page

new_post.php

```
<?php
  session_start();
  require __DIR__ . '/db.php';
  require __DIR__ . '/Post.php';

  if (!($_SESSION['loggedIn'] ?? false)) {
      header("Location: login.php");
      exit;
  }

  $post = new Post($pdo);

  if ($_SERVER['REQUEST_METHOD'] === 'POST') {
      $title = trim($_POST['title'] ?? '');
      $body  = trim($_POST['body'] ?? '');

      if ($title === '' || $body === '') {
          $error = "Title and body are both required.";
      } else {
          $newId = $post->create($title, $body);
          header("Location: index.php");
          exit;
      }
  }
?>
<!-- form HTML, plus echoing $error with htmlspecialchars() if set -->
```

index.php — Listing Posts Safely

index.php

```php
<?php
  require __DIR__ . '/db.php';
  require __DIR__ . '/Post.php';

  $post = new Post($pdo);
  $allPosts = $post->all();
?>
<?php foreach ($allPosts as $row): ?>
  <h2><?= htmlspecialchars($row['title']) ?></h2>
  <p><?= htmlspecialchars($row['body']) ?></p>
<?php endforeach; ?>
```

Every echoed value from the database is wrapped in `htmlspecialchars()` — exactly Chapter 8's rule — since post titles and bodies are, ultimately, also a form of user input that was stored and is now being redisplayed.

DELIBERATE SIMPLIFICATIONS, FOR A CAPSTONE OF THIS SCOPE

This project skips a few things a genuinely production blog would need: CSRF protection on the forms, pagination for `index.php`, and authorisation checks confirming a post's author matches the logged-in user before allowing edit/delete. These are natural "what's missing" discussion points, not oversights — the goal here is demonstrating Chapters 1-9 working together cleanly, not shipping a complete product.

Coding Challenge — Extend the Capstone

FINAL CHALLENGE

Add a findBySlug(string $slug): array method to the Post class, used to support pretty URLs like post.php?slug=my-first-post instead of an ID. It should throw PostNotFoundException if no matching post exists, exactly like find(). Then write post.php, which reads $_GET['slug'], calls the new method inside a try/catch, and either displays the post (escaped with htmlspecialchars) or a friendly "Post not found" message.

View solution

COURSE 2 COMPLETE — PHP INTERMEDIATE

- Chapters 1-9 covered: OOP basics, inheritance/interfaces/traits, exceptions, PDO/CRUD, sessions/login, namespaces/Composer, JSON/cURL, file uploads/safe input, regex
- This capstone combined all nine into a small, genuinely structured blog application with login-protected CRUD
- Known, deliberate gaps: no CSRF protection, pagination, or post-ownership authorisation — flagged as discussion points, not oversights
- **Next:** PHP Advanced (Course 3) — Composer in depth, design patterns, MVC architecture, testing, security in depth, and a REST API capstone