Дисципліна

«Бекенд-розробка, сервіси та мікросервіси»

КОНСПЕКТ ЛЕКЦІЇ. ТЕМА 4

Функції в PHP, функціональне програмування, основи ООП, простори імен, модифікатори доступу, ін'єкція залежностей

Спеціальність: Комп'ютерна інженерія, 3 курс
Обсяг: 1 год 20 хв (лекція 4 з 18)

План лекції

1. Функції в PHP: оголошення та виклик
2. Параметри функцій: значення за замовчуванням, іменовані, варіативні
3. Типізація параметрів і значення, що повертається
4. Анонімні функції, замикання та стрілкові функції
5. Функції як дані. Функціональне програмування в PHP
6. ООП: класи та об'єкти
7. Модифікатори доступу
8. Успадкування, інтерфейси та абстрактні класи
9. Простори імен (namespaces)
10. Ін'єкція залежностей (Dependency Injection)

> *Мета лекції: навчитися структурувати код PHP-застосунку за допомогою функцій та об'єктів — від простого повторного використання логіки до організації великих застосунків через класи, простори імен і слабку зв'язаність компонентів.*

1. Функції в PHP: оголошення та виклик

Функція — це іменований блок коду, який виконує певну дію і може бути викликаний багаторазово з різних місць програми. Функції дозволяють уникати дублювання коду (принцип DRY — Don't Repeat Yourself) і розбивати складну задачу на менші, зрозуміліші частини.

```
<?php
function greet($name) {
    return "Привіт, $name!";
}

echo greet("Оксана");   // Привіт, Оксана!
echo greet("Іван");     // Привіт, Іван!
```

Функція оголошується ключовим словом function, іменем (за тими самими правилами, що й змінні, але без символу $) та списком параметрів у дужках. Тіло функції виконується лише в момент виклику; інструкція return завершує виконання функції та повертає значення викликаючому коду. Якщо return не вказано, функція повертає null.

> *PHP дозволяє оголошувати функції після їх виклику (ріднімання оголошень для звичайних функцій) — на відміну від змінних. Проте на практиці функції зазвичай визначають до використання для читабельності коду.*

2. Параметри функцій

2.1. Значення за замовчуванням

Параметру можна призначити значення за замовчуванням, яке використовується, якщо аргумент не передано під час виклику. Параметри зі значенням за замовчуванням мають розташовуватись після обов'язкових параметрів.

```
<?php
function greet($name, $greeting = "Привіт") {
    return "$greeting, $name!";
}

echo greet("Марія");                // Привіт, Марія!
echo greet("Марія", "Доброго ранку"); // Доброго ранку, Марія!
```

2.2. Іменовані аргументи (з PHP 8.0)

Іменовані аргументи дозволяють передавати значення за іменем параметра, а не за позицією — це підвищує читабельність виклику, коли параметрів багато, і дозволяє пропускати проміжні необов'язкові параметри.

```
<?php
function createUser($name, $age = 18, $city = "Львів") {
    return "$name, $age, $city";
}

// Пропускаємо $age, вказуючи лише потрібні іменовані аргументи
echo createUser(name: "Тарас", city: "Київ");
```

2.3. Варіативна кількість аргументів

Оператор ... (splat/variadic) дозволяє функції приймати довільну кількість аргументів, які збираються в масив усередині функції.

```
<?php
function sum(...$numbers) {
    return array_sum($numbers);
}

echo sum(1, 2, 3);        // 6
echo sum(5, 10, 15, 20);  // 50
```

3. Типізація параметрів і значення, що повертається

Починаючи з PHP 7, можна вказувати типи параметрів і типи значення, що повертається (return types). Це не змінює динамічну природу PHP повністю, але дозволяє інтерпретатору перевіряти відповідність типів під час виклику й раніше виявляти помилки.

```
<?php
function addTax(float $price, float $rate = 0.2): float {
    return $price * (1 + $rate);
}

echo addTax(100);       // 120
echo addTax(100, 0.15); // 115

// Тип, що допускає null — префікс ?
function findUser(?int $id): ?string {
    return $id === null ? null : "Користувач #$id";
}
```

> *Директива declare(strict_types=1) на початку файлу вимикає автоматичне приведення типів аргументів: без неї виклик addTax("100") спрацює (рядок неявно перетвориться на float), а з нею — викине помилку TypeError. Для нових проєктів рекомендується використовувати строгий режим.*

4. Анонімні функції, замикання та стрілкові функції

4.1. Анонімні функції (closures)

Анонімна функція — це функція без імені, яку можна зберегти у змінній, передати як аргумент іншій функції або повернути з функції. У PHP такі функції реалізовані класом Closure.

```
<?php
$square = function($x) {
    return $x * $x;
};

echo $square(5);   // 25
```

4.2. Захоплення змінних через use

Анонімна функція за замовчуванням не має доступу до змінних із зовнішньої області видимості. Ключове слово use дозволяє явно «захопити» потрібні змінні із зовнішнього контексту — за замовчуванням за значенням, або за посиланням через &, якщо функція має змінювати оригінал.

```
<?php
$tax = 0.2;
$addTax = function($price) use ($tax) {
    return $price * (1 + $tax);
};
echo $addTax(100);   // 120

$counter = 0;
$increment = function() use (&$counter) {
    $counter++;
};
$increment();
$increment();
echo $counter;   // 2
```

4.3. Стрілкові функції (з PHP 7.4)

Стрілкові функції fn() => вираз — скорочений синтаксис для простих одноразових функцій. На відміну від анонімних функцій, вони автоматично захоплюють змінні з батьківської області видимості за значенням, без явного use, і тіло обмежене одним виразом.

```
<?php
$tax = 0.2;
$addTax = fn($price) => $price * (1 + $tax);
echo $addTax(100);   // 120 — $tax захоплено автоматично
```

> *Стрілкові функції особливо зручні як колбеки для array_map(), array_filter() і usort() — саме такий стиль ми використовували в темі про масиви.*

5. Функції як дані. Функціональне програмування в PHP

У PHP функції є об'єктами першого класу (first-class citizens): їх можна зберігати у змінних, передавати як аргументи, повертати з інших функцій. Це дозволяє писати код у функціональному стилі — коли програма будується як композиція функцій, що обробляють дані, а не як послідовність команд, які змінюють стан.

5.1. Callable: рядок, масив, замикання

Тип callable позначає «щось, що можна викликати» — це може бути рядок з іменем функції, масив [$object, 'method'] для виклику методу об'єкта, або замикання.

```
<?php
function double($x) { return $x * 2; }

$numbers = [1, 2, 3];

// Рядок з іменем функції — теж callable
$doubled = array_map('double', $numbers);   // [2, 4, 6]

// Перевірка, чи є значення викликаваним
var_dump(is_callable('double'));   // bool(true)
```

5.2. Функції вищого порядку

Функція вищого порядку (higher-order function) — це функція, яка приймає іншу функцію як аргумент або повертає функцію як результат. array_map(), array_filter() і array_reduce() — вбудовані приклади функцій вищого порядку в PHP.

```
<?php
// Функція, що повертає функцію
function multiplier($factor) {
    return fn($x) => $x * $factor;
}

$triple = multiplier(3);
echo $triple(7);   // 21
```

> *PHP — мультипарадигмена мова: вона підтримує і процедурний, і об'єктно-орієнтований, і (частково) функціональний стилі. На практиці елементи функціонального стилю (map/filter/reduce, стрілкові функції) часто поєднують з ООП-архітектурою застосунку — це не взаємовиключні підходи.*

6. ООП: класи та об'єкти

Об'єктно-орієнтоване програмування (ООП) організовує код навколо об'єктів — сутностей, які поєднують дані (властивості) та поведінку (методи). Клас — це шаблон (креслення) для створення об'єктів; об'єкт (екземпляр класу) — конкретна реалізація цього шаблону з власними значеннями властивостей.

```
<?php
class Product {
    public string $name;
    public float $price;

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

    public function getPriceWithTax(float $rate = 0.2): float {
        return $this->price * (1 + $rate);
    }
}

$laptop = new Product("Ноутбук", 25000);
echo $laptop->name;                 // Ноутбук
echo $laptop->getPriceWithTax();    // 30000
```

Спеціальний метод __construct() — конструктор — викликається автоматично під час створення об'єкта оператором new і зазвичай використовується для ініціалізації властивостей. Псевдозмінна $this всередині методу посилається на поточний об'єкт, для якого метод викликано.

> *З PHP 8.0 доступний скорочений синтаксис — promoted properties: якщо оголосити модифікатор доступу прямо в параметрах конструктора (public function __construct(public string $name, public float $price) {}), властивості створюються й ініціалізуються автоматично, без явних рядків $this->name = $name.*

7. Модифікатори доступу

Модифікатори доступу визначають, звідки можна звертатися до властивості чи методу класу. Це основа інкапсуляції — принципу приховування внутрішньої реалізації об'єкта від зовнішнього коду, який повинен взаємодіяти лише через чітко визначений публічний інтерфейс.

Модифікатор	Доступ
public	Доступний звідусіль — з будь-якого місця коду
protected	Доступний усередині класу та в усіх його нащадках (при успадкуванні)
private	Доступний лише всередині самого класу, де оголошений

```php
<?php
class BankAccount {
    private float $balance = 0;

    public function deposit(float $amount): void {
        if ($amount > 0) {
            $this->balance += $amount;
        }
    }

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

$account = new BankAccount();
$account->deposit(1000);
echo $account->getBalance();   // 1000
// $account->balance;         // Помилка! Властивість private
```

> *Властивість balance навмисно приховано (private) і доступ до неї відбувається лише через контрольовані публічні методи (deposit, getBalance) — це не дає зовнішньому коду встановити від'ємний або некоректний баланс напряму, обходячи перевірку.*

8. Успадкування, інтерфейси та абстрактні класи

8.1. Успадкування

Успадкування (extends) дозволяє класу-нащадку отримати властивості й методи батьківського класу, доповнюючи або перевизначаючи їх. Це підтримує повторне використання коду між спорідненими класами.

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

    public function makeSound(): string {
        return "$this->name видає звук";
    }
}

class Dog extends Animal {
    public function makeSound(): string {
        return "$this->name гавкає";   // перевизначення (override)
    }
}

$dog = new Dog("Рекс");
echo $dog->makeSound();   // Рекс гавкає
```

8.2. Інтерфейси

Інтерфейс (interface) визначає контракт — перелік методів, які клас зобов'язаний реалізувати, без вказання їхньої внутрішньої логіки. Клас може реалізовувати (implements) кілька інтерфейсів одночасно, хоча успадковувати можна лише від одного класу.

```
<?php
interface PaymentMethod {
    public function pay(float $amount): bool;
}

class CardPayment implements PaymentMethod {
    public function pay(float $amount): bool {
        echo "Оплата $amount грн карткою";
        return true;
    }
}
```

8.3. Абстрактні класи

Абстрактний клас (abstract class) не можна створити напряму оператором new — він служить базою для інших класів і може містити як звичайні реалізовані методи, так і абстрактні методи (без тіла), які кожен нащадок зобов'язаний реалізувати самостійно.

```php
<?php
abstract class Shape {
    abstract public function getArea(): float;

    public function describe(): string {
        return "Площа фігури: " . $this->getArea();
    }
}

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

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

> *Практичне правило вибору: інтерфейс — коли потрібно лише узгодити «що вміє робити» клас (контракт) без спільної реалізації; абстрактний клас — коли є спільна логіка, яку варто успадкувати, і водночас частина поведінки має визначатись у кожному нащадку окремо.*

9. Простори імен (namespaces)

У міру зростання проєкту зростає й кількість класів — виникає ризик конфлікту імен (наприклад, два різні пакети можуть визначити власний клас Logger). Простір імен (namespace) — це спосіб логічно згрупувати пов'язані класи, функції та константи й уникнути таких конфліктів, подібно до того, як тека в файловій системі відокремлює файли з однаковими іменами в різних директоріях.

```
<?php
// файл src/Payment/CardPayment.php
namespace App\Payment;

class CardPayment {
    public function pay(float $amount): void {
        echo "Оплата $amount грн";
    }
}
```

```
<?php
// файл src/index.php
require 'src/Payment/CardPayment.php';

use App\Payment\CardPayment;

$payment = new CardPayment();
$payment->pay(500);

// Альтернатива без use — повне (кваліфіковане) ім'я
$payment2 = new \App\Payment\CardPayment();
```

Оператор use дозволяє «імпортувати» клас із простору імен і звертатися до нього коротким іменем далі в файлі. Це особливо важливо в сучасних PHP-проєктах, які використовують Composer і стандарт автозавантаження PSR-4, де структура просторів імен зазвичай дзеркально відповідає структурі тек проєкту.

10. Ін'єкція залежностей (Dependency Injection)

Залежність — це інший об'єкт (сервіс), без якого клас не може виконати свою роботу. Якщо клас сам створює свої залежності всередині (наприклад, new Database() прямо в конструкторі), він стає жорстко зв'язаним (tightly coupled) із конкретною реалізацією — таке важко тестувати й змінювати.

```
<?php
// Погана практика: клас сам створює залежність
class OrderService {
    private Database $db;

    public function __construct() {
        $this->db = new MySqlDatabase();   // жорстко "зашита" залежність
    }
}
```

Ін'єкція залежностей (Dependency Injection, DI) вирішує цю проблему: замість того щоб клас сам створював залежність, вона передається («ін'єктується») ззовні — найчастіше через конструктор. Клас працює з абстракцією (інтерфейсом), а не з конкретною реалізацією, тому реалізацію легко підмінити (наприклад, на тестову-заглушку — mock — під час тестування).

```
<?php
interface Database {
    public function query(string $sql): array;
}

class MySqlDatabase implements Database {
    public function query(string $sql): array {
        // реальний запит до MySQL
        return [];
    }
}

class OrderService {
    // Конструкторна ін'єкція залежності
    public function __construct(private Database $db) {}

    public function getOrders(): array {
        return $this->db->query("SELECT * FROM orders");
    }
}

// Залежність передається ЗЗОВНІ під час створення об'єкта
$service = new OrderService(new MySqlDatabase());
```

> *Ключова перевага: OrderService більше не залежить від конкретної бази даних — можна підставити InMemoryDatabase для юніт-тестів або PostgresDatabase у майбутньому, не змінюючи жодного рядка в самому OrderService. У великих застосунках цим процесом (автоматичним створенням і "впорскуванням" залежностей) керує спеціальний компонент — DI-контейнер, який ми детальніше розглянемо при вивченні фреймворків.*

Питання для самоперевірки

- Чим стрілкова функція fn() відрізняється від анонімної function() щодо захоплення зовнішніх змінних?
- Для чого призначене ключове слово use у визначенні анонімної функції та навіщо іноді потрібен префікс &?
- Що означає, що функції в PHP є "об'єктами першого класу"? Наведіть приклад функції вищого порядку.
- Чим відрізняються модифікатори public, protected і private? Наведіть приклад ситуації, де доцільно використати private.
- У чому різниця між інтерфейсом і абстрактним класом? Коли варто обрати кожен з них?
- Навіщо потрібні простори імен (namespaces) і як вирішується конфлікт однакових імен класів без них?
- Що таке ін'єкція залежностей і чому жорстко "зашита" всередині класу залежність вважається поганою практикою?
- Наведіть приклад, коли конструкторна ін'єкція залежності спрощує тестування коду.

Короткий глосарій

Термін	Означення
Замикання (closure)	Анонімна функція, здатна захоплювати змінні з навколишньої області видимості
Callable	Значення, яке можна викликати як функцію (ім'я, масив [об'єкт, метод], замикання)
Функція вищого порядку	Функція, що приймає або повертає іншу функцію
Інкапсуляція	Приховування внутрішньої реалізації об'єкта за публічним інтерфейсом
Інтерфейс	Контракт із переліком методів, які клас зобов'язаний реалізувати
Простір імен (namespace)	Логічне групування класів і функцій для уникнення конфліктів імен
Ін'єкція залежностей (DI)	Передача залежності об'єкту ззовні замість її створення всередині
DI-контейнер	Компонент, що автоматизує створення й "впорскування" залежностей