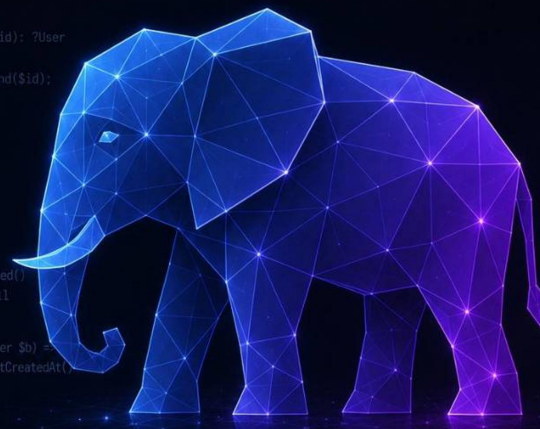
PHP 8

Современная разработка

От чистого языка до production



Максим Байтович

Максим Байтович

PHP 8. Современная разработка: от чистого языка до production

https://litres.ru/74208439
SelfPub; 2026

Аннотация

Эта книга — не справочник по функциям и не рецепты для копипаста. Это пошаговое строительство настоящего приложения на современном PHP 8: от первой строгой типизации до развёртывания на сервере с Nginx, PostgreSQL и HTTPS.

Вы пройдёте путь, который прошёл сам язык: от скриптов вперемешку с HTML — к профессиональной архитектуре со статическим анализом, DI-контейнерами, сервисным слоем и очередями. Напишете собственный роутер, контейнер и шаблонизатор, чтобы понять, как работают фреймворки изнутри. Затем переложите те же принципы на Laravel и Symfony — осознанно, без магии.

В книге один сквозной проект — менеджер личных финансов. Вы проведёте его через HTTP и консоль, покроете тестами, защитите от XSS и CSRF, настроите CI и деплой. Каждая глава заканчивается практическим заданием.

Для тех, кто уже немного писал на PHP и хочет перейти от «работает» к «написано правильно».

Содержание

PHP 8. Современная разработка: от чистого языка до production

Глава 1. Не ваш дедушкин PHP

В которой мы хороним стереотипы, настраиваем окружение и пишем первый скрипт, который не стыдно показать.

1.1. Слон в посудной лавке

Спросите любого разработчика, не пишущего на PHP, что он думает об этом языке. С высокой вероятностью вы услышите:

«Это язык для WordPress-поделок.»

«Там в коде HTML вперемешку с SQL, видел в гайде 2005 года.»

«У них стрелка не туда показывает.» (Загадочная, но частая претензия.)

Во многом эта репутация заслужена. PHP родился в 1994 году как набор CGI-скриптов для трекинга посещений резюме. Расмус Лердорф и не планировал создавать язык программирования. А потом проект вырос, оброс глобальными функциями с непоследовательным именованием (strpos vs str_rot13) и стал основой веба: на PHP работают Wikipedia, Facebook (ранний), Slack и огромный кусок интернета.

Но с тех пор прошло три десятилетия. PHP пережил две большие революции:

PHP 7 (2015): полная переработка движка, скорость выросла в разы.

PHP 8 (2020–...): строгая типизация, енумы, match-выражения, readonly-свойства и JIT-компилятор.

Современный PHP — это не скриптование вперемешку с версткой. Это язык со строгой типизацией, атрибутами, асинхронными фреймворками и экосистемой, построенной вокруг Composer. Писать на PHP 8 — значит писать чистый, типобезопасный код, который часто неотличим по выразительности от Kotlin или TypeScript.

Миф: «PHP годится только для блогов».
Реальность: На PHP пишут высоконагруженные API, мик-

росервисы, системы очередей и сложные финансовые приложения. Инструменты статического анализа вроде PHPStan и Psalm давно превзошли аналоги из мира Python и Node.js.

Цель этой книги — показать PHP, который существует сегодня. Не тот, о котором рассказывают старые блог-посты.

1.2. Что нам понадобится

Никаких тяжелых IDE и платных хостингов. Минимальный набор джентльмена:

PHP 8.2 или 8.3. Можно поставить через пакетный менеджер, но мы пойдем модным путем — Docker.

Текстовый редактор. VS Code с плагином Intelephense — золотой стандарт. PhpStorm — если вы готовы к небольшой магии автодополнения.

Composer. Менеджер зависимостей. Придет в следующей главе, но пусть уже стоит.

Быстрый старт с Docker (проверено, работает без танцев):

Создайте папку проекта и в ней файл docker-compose.yml:
yaml

```
version: '3.8'
services:
app:
image: php:8.3-cli
volumes:
- .:/app
working_dir: /app
```

Запуск: docker compose run --rm app php script.php.

Всё. PHP работает, ничего лишнего не протекло в систему.

Для тех, кто не дружит с Docker: прямой дистрибутив с php.net и команда php -S localhost:8000 (встроенный веб-сервер) — честный способ начать.

1.3. Первый код. Не «Hello, World», а знакомство с типами

Традиционный hello world скучен. Давайте сразу покажем, чем дышит современный PHP:

```
php

<?php

declare(strict_types=1);

function greet(string $name, ?string $title = null): string
```

```
{
$prefix = $title ?? 'дорогой';
return "Привет, {$prefix} {$name}!";
}

echo greet('Алексей', 'уважаемый'); // Привет, уважаемый Алексей!
echo greet('Мария'); // Привет, дорогой Мария!
```

Что здесь важно заметить сразу:

declare(strict_types=1) — с этой строки мы включаем режим строгой типизации. PHP больше не будет неявно превращать "123abc" в 123 при передаче в функцию, ожидающую int. Это первая привычка, которую мы вырабатываем. Она спасет от багов, которые сложно дебажить.

Сигнатура функции greet(string $name, ?string $title = null): string — здесь указаны типы всех параметров и возвращаемого значения. ?string означает «строка или null». Такие объявления — не комментарии для красоты. PHP проверяет их во время выполнения. Если передать int вместо string — скрипт упадет с ясной ошибкой TypeError, а не продолжит тихо творить дичь.

Оператор ?? (null coalescing) — изящный способ ска-

зать «если левое значение null, бери правое». Никаких isset($title) ? $title : 'дорогой'.

Интерполяция переменных в двойных кавычках работает, но мы используем её осознанно, а не для вывода сырого HTML.

1.4. Как PHP исполняет код (краткий ликбез)

PHP — не компилируемый язык в классическом понимании. Запрос приходит — PHP исполняет скрипт. Но внутри всё хитрее:

Код превращается в OPcache (байт-код) и кешируется в памяти. Нет перечитывания файлов на каждый чих.

Начиная с PHP 8.0, в ядре работает JIT-компилятор, который может транслировать горячие участки кода прямо в машинные инструкции. Это радикально ускоряет вычислительные задачи (машинное обучение на чистом PHP всё еще сомнительно, но обработка изображений или сложная математика — уже да).

Нам для старта это знание нужно, чтобы понимать: PHP — не медленный интерпретатор-черепаха. Он быстр, предсказуем и идеально заточен под модель «запрос-ответ». А

с развитием проектов вроде Swoole и RoadRunner он умеет работать и как долгоживущий процесс, не теряя состояния между запросами.

1.5. План на книгу (без спойлеров)

В этой книге мы не будем строить фреймворк-убийцу. Мы напишем реальный проект — менеджер личных финансов. Через этот проект мы пройдем:

Главы 2-4: фундамент типов, ООП и структуры данных.

Главы 5-8: база данных, формы, API и защита от угона кошелька.

Главы 9-12: архитектура без магии (DI, консольные команды, тесты).

Главы 13-15: Composer, продвинутый тулинг и взгляд в асинхронность.

Каждая глава заканчивается практическим заданием. Например, к концу этой главы задание будет звучать так:

Задание: установить PHP, запустить скрипт с объявленной функцией, и намеренно передать в неё неверный тип, чтобы увидеть TypeError. Прочувствовать, как строгая ти-

пизация ловит ошибку до того, как она превратится в «а почему баланс пользователя стал NaN?».

Вопрос к тебе:

Глава 2. Типы — это не скучно

В которой мы выясняем, почему mixed — не ругательство, а union types — не профсоюз, и учим PHP проверять наш код до того, как он упадёт на продакшене.

2.1. Зачем нам типы, если PHP динамический?

PHP всегда был языком с динамической типизацией. Переменная $x могла быть строкой, потом числом, потом объектом класса DatabaseConnection, и никто не жаловался. Кроме тех, кто это потом дебажил в три часа ночи.

Проблема динамической типизации — не в скорости, а в молчаливых ошибках:

```
php

// Классика. Без строгой типизации.
function calculateDiscount(float $price, int $percent): float
{
return $price * ($percent / 100);
}

calculateDiscount('100', 'десять'); // Что произойдёт?
// PHP молча приведёт '100' к 100.0, а 'десять' к 0.
```

// Результат: 0.0. Ни ошибки, ни предупреждения. Баг в боевом кошельке.

Со strict_types=1 этот код выбросит TypeError ещё на входе в функцию. Не нужно писать ручную валидацию каждого параметра. PHP делает это сам.

Вывод: типы в сигнатурах — это не про бюрократию. Это про то, чтобы проблемы обнаруживались в момент вызова, а не в момент, когда пользователь видит пустой баланс.

2.2. Скалярные типы и их нюансы

Базовые типы: int, float, string, bool. Казалось бы, что тут говорить. Но дьявол в деталях.

int и float

```
php

declare(strict_types=1);

function divide(int $a, int $b): float
{
return $a / $b;
}

echo divide(5, 2); // 2.5
```

```
// Возвращаемый float принимается без вопросов.

// А так — нет:
function getInt(): int
{
return 5.7; // TypeError: Return value must be of type int, float returned.
}
```

PHP не округлит 5.7 до 6 при возврате, даже если вам кажется, что это логично. Строгая типизация означает строгую.

bool: правда и ложь по-человечески

```
php

function isActive(bool $status): string
{
return $status ? 'Активен' : 'Неактивен';
}

isActive(1); // TypeError. 1 — не bool.
isActive('true');// TypeError. 'true' — не bool.
```

Только true и false. Никаких 0, 1, 'yes'. Мозг расслабляется, глаза не дёргаются.

string: осторожно с null
php

function sayHello(string $name): string
{
return "Hello, {$name}";
}

sayHello(null); // TypeError без ?string.

Если параметр может быть null, мы обязаны сказать об этом явно (до этого доберёмся через минуту).

2.3. Union Types: когда одного типа мало

С PHP 8.0 можно указать, что переменная принимает значения нескольких типов. Синтаксис: типA|типB.

php

function formatBalance(int|float $amount): string
{
return number_format($amount, 2, '.', ' ') . ' ';
}

formatBalance(1500); // "1 500.00 "
formatBalance(99.5); // "99.50 "

formatBalance('100'); // TypeError: string is not int|float.

До PHP 8.0 пришлось бы писать два отдельных метода или принимать всё через mixed и проверять руками. Union types — это документация, которую компилятор проверяет за вас.

Частый кейс: параметр или null
php

function findUser(int $id): ?User
{
// Возвращаем User или null, если не нашли.
}

?User — синтаксический сахар для User|null. Оба варианта равнозначны, но ?Type короче и привычнее для глаз. Важно: ?int $x = null — значение по умолчанию null обязательно, иначе PHP заставит передать аргумент явно.

2.4. Mixed: осознанная тишина

mixed — тип, который говорит: «я не знаю или мне всё равно, что здесь окажется». Это не ошибка, это намеренное решение.

php

function logAnything(mixed $data): void

```
{
file_put_contents('app.log', print_r($data, true), FILE_APPEND);
}

logAnything(['user' => 'admin']);
logAnything('Какой-то текст');
logAnything(42);
logAnything(new Exception('Ой'));
```

Когда использовать mixed:

Логирование и дебаг-утилиты.

Методы, которые действительно принимают что угодно (например, сериализаторы).

Переходный период при рефакторинге легаси.

Когда НЕ использовать:

В сигнатурах бизнес-логики. Если метод createOrder принимает mixed, что-то пошло не так.

2.5. Типы возврата: void, never и static

void

Функция ничего не возвращает (точнее, возвращает null, но PHP запрещает использовать этот null):

```
php

function sendEmail(string $to): void
{
// Отправляем письмо и молчим.
// return null; — ошибка.
// return; — ок.
}
```

never

Новый тип с PHP 8.1. Функция или метод никогда не завершается нормально. Она либо бросает исключение, либо вызывает exit().

```
php

function redirect(string $url): never
{
header("Location: {$url}");
exit();
}
```

```
function throwNotFound(): never
{
throw new \RuntimeException('Не найдено');
}
```

Зачем это нужно? Статический анализатор видит never и понимает: код после вызова этой функции недостижим. Если вы написали:

```
php

$user = findUser($id) ?? throwNotFound();
// Здесь $user точно не null. PHPStan это знает.
```

static

Используется в наследовании. Если метод возвращает static, значит, он возвращает экземпляр того класса, у которого вызван, а не того, где метод определён.

```
php

class QueryBuilder
{
public function where(string $clause): static
{
// Добавляем условие и возвращаем себя.
return $this;
```

```
}
}

class UserQuery extends QueryBuilder
{
public function active(): static
{
return $this->where('active = 1');
}
}

$query = (new UserQuery())->active();
// $query — экземпляр UserQuery, не QueryBuilder.
```

Это фундамент для текучих интерфейсов (fluent API) и правильного автодополнения в IDE.

2.6. Типы свойств класса

С PHP 7.4 можно объявлять типы у свойств класса. С PHP 8.1 — делать их readonly.

```
php

class Money
{
public readonly float $amount;
public readonly string $currency;
```

```
public function __construct(float $amount, string $currency)
{
$this->amount = $amount;
$this->currency = $currency;
}
}

$m = new Money(100.0, 'RUB');
$m->amount = 200.0; // Error: Cannot modify readonly property.
```

Нюансы:

readonly можно установить только один раз (обычно в конструкторе).

Неприменимо к свойствам со значением по умолчанию в объявлении.

Объект целиком становится неизменяемым по этому свойству — кирпичик в сторону иммутабельности.

2.7. Проверка типов: instanceof и не только

Для объектов:

```
php

if ($user instanceof Admin) {
// Делаем админские штуки.
}
```

Для массивов и сложных структур instanceof бесполезен. Тут в будущем помогут енумы, интерфейсы и коллекции (глава 4 заложит фундамент).

Пока запомним: если метод возвращает User|null, мы обязаны проверить результат перед использованием:

```
php

$user = $repository->find($id);
if ($user === null) {
throw new \RuntimeException('Пользователь не найден');
}
// Далее $user — точно User.
```

Или изящнее (PHP 8.0+):

```
php

$user = $repository->find($id) ?? throw new \RuntimeException('Не найден');
```

Оператор ?? + throw — коротко, ясно и без лишних if.

2.8. Практическое задание

Создайте функцию calculateTotal(array $items, float $discount = 0.0): float.

$items — массив цен (float).

$discount — скидка в процентах.

Включите strict_types=1.

Вызовите функцию с корректными данными.

Вызовите её с массивом, содержащим строку вместо float, и зафиксируйте TypeError.

Напишите метод transfer(Money $from, Money $to, int|float $amount): void.

Проверьте, что $amount больше нуля.

Если валюты $from и $to не совпадают — выбросьте исключение.

Объявите класс Temperature с readonly-свойством float

$celsius и методом toFahrenheit(): float. Создайте объект и попробуйте изменить свойство после создания. Прочувствуйте момент.

Глава 3. Функции, замыкания и всё, что между

В которой мы прощаемся с switch из девяностых, учим функции летать без имени и понимаем, почему array_map — не просто велосипед для хипстеров.

3.1. Именованные функции: переосмысление

Именованные функции в PHP вы знаете. Но даже в них PHP 8 принёс изменения, которые делают код чище.

Именованные аргументы (PHP 8.0)

Раньше порядок параметров был догмой. Теперь можно передавать аргументы по имени:

php

```
function createInvoice(
string $client,
float $amount,
string $currency = 'RUB',
bool $paid = false,
): string {
```

```
// ...
}

// Можно так:
createInvoice(
client: 'ООО Рога',
amount: 15000.00,
paid: true,
);

// Или даже вразнобой (но лучше так не злоупотреблять):
createInvoice(
paid: false,
amount: 5000.00,
client: 'ИП Копыта',
);
```

Преимущества:

Код самодокументирован.

Можно пропускать необязательные параметры в середине, не передавая значения по умолчанию для предыдущих.

При рефакторинге — меньше шанс перепутать местами два string-параметра и получить трудноуловимый баг.

Вариативные аргументы: ...$args

Если функция принимает произвольное количество аргументов одного типа:

```
php

function sum(int ...$numbers): int
{
return array_sum($numbers);
}

sum(1, 2, 3, 4); // 10
sum(); // 0

// А так — TypeError:
sum(1, 'два', 3);
```

Распаковка работает и в обратную сторону:

```
php

$data = [1, 2, 3];
sum(...$data); // 6
```

Callback-тип: callable

Когда функция ожидает другую функцию:

```
php

function applyDiscount(array $prices, callable $discountFn): array
{
return array_map($discountFn, $prices);
}

applyDiscount([100, 200], fn(float $p) => $p * 0.9);
// [90.0, 180.0]
```

callable принимает строки с именами функций, массивы вида [$object, 'method'], анонимные функции и стрелочные. Но статический анализатор предпочтёт более конкретный тип: \Closure.

3.2. Анонимные функции и замыкания

Анонимная функция — это функция без имени. Но её настоящая сила в замыкании — способности захватывать переменные из внешней области видимости.

```
php

$multiplier = 2.5;

$fn = function (float $value) use ($multiplier): float {
```

```
return $value * $multiplier;
};

$fn(10); // 25.0
```

use захватывает переменную по значению на момент определения. Сюрприз для новичков:

```
php

$x = 1;
$fn = function () use ($x): int {
return $x;
};

$x = 99;
echo $fn(); // 1, не 99!
```

Если нужно захватить по ссылке — use (&$x). Но с этим осторожно: изменяемое состояние из замыкания — источник трудноотлаживаемых багов. Современный подход — не мутировать переменные, а возвращать новые значения.

Замыкание как фабрика

Классический пример — создание параметризованных функций:

```
php

function makeMultiplier(float $factor): \Closure
{
return fn(float $value): float => $value * $factor;
}

$double = makeMultiplier(2);
$triple = makeMultiplier(3);

$double(5); // 10
$triple(5); // 15
```

Стрелочная функция fn() => ... автоматически захватывает переменные по значению. use не нужен. Код лаконичен до предела.

3.3. Стрелочные функции: синтаксис и границы

С PHP 7.4 появился короткий синтаксис:

```
php

// Было:
$fn = function (int $x) use ($y): int {
return $x + $y;
};
```

// Стало:
$fn = fn(int $x): int => $x + $y;

Особенности стрелочных функций:

Всегда возвращают значение (одно выражение, без return).

Автозахват переменных по значению.

Не могут содержать несколько выражений или блоков { }. Для сложной логики — обычные анонимные функции.

Где применять: в array_map, array_filter, usort и любых колбэках, где логика умещается в одно выражение.
php

$users = [
['name' => 'Анна', 'age' => 28],
['name' => 'Борис', 'age' => 35],
];

$names = array_map(fn(array $u): string => $u['name'], $users);
// ['Анна', 'Борис']

$adults = array_filter($users, fn(array $u): bool => $u['age']

>= 30);
// Только Борис.

3.4. match-выражение: замена switch

switch страдает от двух проблем: требует break и может случайно проваливаться. match (PHP 8.0) решает обе:

```
php

function getStatusText(int $code): string
{
return match($code) {
200 => 'OK',
301 => 'Moved Permanently',
404 => 'Not Found',
500 => 'Internal Server Error',
default => 'Unknown',
};
}
```

Что важно:

Строгое сравнение (===). match('200') не совпадёт с 200.

Ошибка, если ни одна ветка не подошла и нет default.

Каждая ветка — одно выражение (можно блок с => и фигурными скобками для многострочной логики).

Возвращает значение, поэтому прямо в return.

Продвинутый пример с условиями:
php

```
function classifyNumber(int $n): string
{
return match(true) {
$n < 0 => 'Отрицательное',
$n === 0 => 'Ноль',
$n < 10 => 'Однозначное',
$n < 100 => 'Двузначное',
default => 'Большое',
};
}
```

Шаблон match(true) позволяет использовать произвольные условия, не ограничиваясь одним значением для сравнения.

3.5. Функции высшего порядка и композиция

Функция высшего порядка — та, что принимает или возвращает другую функцию. Мы уже видели makeMultiplier().

Теперь пойдем дальше:

Композиция функций (ручная):
php

```
$formatPrice = fn(float $amount): string => number_format($amount, 2) . ' ';
$applyTax = fn(float $amount): float => $amount * 1.2;
$roundUp = fn(float $amount): float => ceil($amount);

// Композиция: округлить добавить налог отформатировать
$process = fn(float $a): string => $formatPrice($applyTax($roundUp($a)));

$process(99.1); // "119.00 "
```

Пока выглядит как матрёшка. Во второй части книги мы напишем пайплайн-обработчик, который сделает это читаемым. Пока важно понять идею: функции можно собирать как конструктор, маленькие и предсказуемые.

Частичное применение (вручную):
php

```
function discount(float $percent): \Closure
```

```
{
return fn(float $price): float => $price * (1 - $percent / 100);
}

$winterSale = discount(20);
$vipSale = discount(50);

$winterSale(1000); // 800.0
$vipSale(1000); // 500.0
```

3.6. Чистые функции: почему это важно

Чистая функция:

Для одних и тех же входных данных всегда возвращает один и тот же результат.

Не имеет побочных эффектов (не пишет в БД, не меняет глобальные переменные, не печатает на экран).

```
php

// Нечистая:
function applyDiscountUnclean(float $price): float
{
global $currentUser;
```

```
if ($currentUser->isVip()) {
return $price * 0.8;
}
return $price;
}

// Чистая:
function applyDiscountPure(float $price, User $user): float
{
if ($user->isVip()) {
return $price * 0.8;
}
return $price;
}
```

Разница: чистую функцию легко тестировать. Не нужно поднимать базу, мокировать глобальное состояние. Вызываем с разными User и проверяем результат.

В нашем менеджере финансов мы будем стремиться к тому, чтобы бизнес-логика была чистой, а все побочные эффекты (БД, файлы, HTTP) — изолированы в отдельных слоях. Это главная архитектурная мысль книги.

3.7. Практическое задание

Конвертер валют на замыканиях:

Напишите функцию makeConverter(float $rate): \Closure, которая возвращает функцию для конвертации суммы по курсу. Создайте конвертеры $usdToRub и $eurToRub. Сконвертируйте 100 единиц каждой валюты.

Калькулятор с match:

Функция calculate(float $a, float $b, string $operation): float|string. Операции: '+', '-', '*', '/'. При делении на ноль вернуть строку 'Ошибка: деление на ноль'. Используйте match. Проверьте строгость сравнения, передав '+' и '+' через переменную с пробелом.

Сортировка пользователей по возрасту:

Дан массив пользователей:

php

```
$users = [
['name' => 'Иван', 'age' => 25],
['name' => 'Мария', 'age' => 31],
['name' => 'Пётр', 'age' => 19],
];
```

Отсортируйте его по возрасту по возрастанию с помощью usort и стрелочной функции. Отсортируйте по имени в алфавитном порядке. Оба результата выведите через print_r.

Рефакторинг на чистую функцию:

Ниже приведён код с побочным эффектом:

php

```
function addTransaction(float $amount, string $category): void
{
global $transactions;
$transactions[] = ['amount' => $amount, 'category' => $category];
}
```

Перепишите функцию, чтобы она не использовала global, а принимала массив и возвращала новый. Убедитесь, что исходный массив не изменился.

Глава 4. ООП без боли

В которой мы перестаём писать классы ради классов, знакомимся с enum и readonly, и выясняем, почему наследование — не всегда лучший друг.

4.1. Зачем вообще классы в PHP?

PHP начинался без ООП. Потом добавили классы в PHP 4 (ужасно), переписали в PHP 5 (сносно), и с тех пор каждый релиз делает объектную модель стройнее.

Но зачем классы в мире, где есть функции и ассоциативные массивы? Ответ: инкапсуляция инвариантов.

Предположим, мы храним деньги как float $amount и string $currency. Что угодно может стать «деньгами»: ['amount' => 100, 'currency' => 'RUB']. А что, если кто-то напишет 'RUB' как 'rub' или 'RUR'? Что, если amount окажется отрицательным?

Класс — это способ сказать: «Вот структура. Вот правила, которые всегда соблюдаются. Снаружи нельзя создать невалидный объект».

php

```
class Money
{
public function __construct(
public readonly float $amount,
public readonly string $currency,
) {
if ($amount < 0) {
throw new \InvalidArgumentException('Сумма не может быть отрицательной');
}
if (!in_array($currency, ['RUB', 'USD', 'EUR'], true)) {
throw new \InvalidArgumentException("Неизвестная валюта: {$currency}");
}
}
}

$m = new Money(100.0, 'RUB'); // Ок.
$m = new Money(-50.0, 'RUB'); // Исключение.
$m = new Money(50.0, 'RUR'); // Исключение.
```

Теперь везде, где используется Money, мы знаем: сумма неотрицательна, валюта валидна. Никаких проверок на местах. Это и есть инкапсуляция — не просто «private поля с геттерами», а защита правил.

4.2. Конструктор в стиле PHP 8

С PHP 8.0 конструктор можно писать короче:
php

```
// Было (PHP 7):
class User
{
private string $name;
private int $age;

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

// Стало (PHP 8):
class User
{
public function __construct(
private string $name,
private int $age,
) {}
}
```

Модификатор доступа (public, private, protected) прямо в параметре — и свойство создано, значение присвоено. Никакого бойлерплейта.

С readonly (PHP 8.1) — ещё чище:

```
php

class User
{
public function __construct(
public readonly string $name,
public readonly int $age,
) {}
}
```

Теперь $name и $age доступны снаружи для чтения, но не могут быть изменены. Идеально для DTO и value objects.

4.3. readonly-классы (PHP 8.2)

Если все свойства класса должны быть readonly, можно пометить весь класс:

```
php

readonly class Money
{
```

```
public function __construct(
public float $amount,
public string $currency,
) {
// Валидация здесь.
}
}
```

Все свойства автоматически readonly. Нельзя добавить обычное свойство. Нельзя использовать static-свойства (пока что ограничение движка). Объект целиком неизменяем после создания.

Когда использовать:

Value objects (Money, Email, DateRange).

DTO для передачи данных между слоями.

Всё, что не должно меняться после создания.

4.4. Enum: больше чем константы

До PHP 8.1 мы использовали константы:
php

```
class TransactionType
{
public const INCOME = 'income';
public const EXPENSE = 'expense';
}

function logTransaction(float $amount, string $type): void
{
// $type может быть чем угодно. 'income'? 'Income'? 'iNcOmE'?
}
```

Теперь — enum:

```
php

enum TransactionType: string
{
case Income = 'income';
case Expense = 'expense';
}
```

Типизированный enum. TransactionType::Income — это объект, который при приведении к строке даст 'income'. Но в сигнатуре метода он — гарантия:

```
php
```

```
function logTransaction(float $amount, TransactionType $type): void
{
// $type — ТОЛЬКО Income или Expense. Никаких строк.
}

logTransaction(100.0, TransactionType::Income);
```

Enum'ы могут иметь методы:

```
php

enum TransactionType: string
{
case Income = 'income';
case Expense = 'expense';

public function label(): string
{
return match($this) {
self::Income => 'Доход',
self::Expense => 'Расход',
};
}

public function isIncome(): bool
{
```

return $this === self::Income;
}
}

TransactionType::Income->label(); // 'Доход'

Чистый enum (без значения):
php

enum Status
{
case Draft;
case Published;
case Archived;
}

Без : string или : int. Просто перечисление состояний. Нельзя сравнить со строкой или числом — только с другим экземпляром. Идеально для статусов, ролей, состояний конечного автомата.

Метод tryFrom и from:
php

$type = TransactionType::from('income'); // Income
$type = TransactionType::from('invalid'); // ValueError

$type = TransactionType::tryFrom('invalid'); // null (без исключения)

4.5. Интерфейсы: контракты, не наследование

Интерфейс — это обещание: «я умею делать X». Без деталей реализации.

php

```
interface CalculatesArea
{
public function area(): float;
}

class Rectangle implements CalculatesArea
{
public function __construct(
private float $width,
private float $height,
) {}

public function area(): float
{
return $this->width * $this->height;
}
```

```
}

class Circle implements CalculatesArea
{
public function __construct(
private float $radius,
) {}

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

function printArea(CalculatesArea $shape): void
{
echo "Площадь: " . $shape->area() . "\n";
}

printArea(new Rectangle(5, 3)); // 15
printArea(new Circle(2)); // ~12.57
```

Функция printArea не знает ничего о прямоугольниках и кругах. Она знает только контракт: у объекта есть метод area(): float. Это полиморфизм без наследования.

4.6. Композиция вместо наследования

Наследование часто учат как основу ООП. На практике оно легко приводит к хрупким иерархиям.

Классический пример проблемы:
php

```
class Bird
{
public function fly(): string
{
return 'Я лечу!';
}
}

class Penguin extends Bird
{
public function fly(): string
{
throw new \Exception('Пингвины не летают!');
}
}
```

Пингвин — птица, но нарушает контракт родителя. Код, ожидающий Bird, упадет на Penguin. Это нарушение принципа подстановки Лисков (LSP).

Выход — композиция:
php

```
interface FlyBehavior
{
public function fly(): string;
}

class FlyWithWings implements FlyBehavior
{
public function fly(): string
{
return 'Я лечу!';
}
}

class CantFly implements FlyBehavior
{
public function fly(): string
{
return 'Я не летаю.';
}
}

class Bird
```

```
{
public function __construct(
private readonly string $name,
private readonly FlyBehavior $flyBehavior,
) {}

public function tryToFly(): string
{
return "{$this->name}: " . $this->flyBehavior->fly();
}
}

$eagle = new Bird('Орёл', new FlyWithWings());
$penguin = new Bird('Пингвин', new CantFly());

$eagle->tryToFly(); // Орёл: Я лечу!
$penguin->tryToFly(); // Пингвин: Я не летаю.
```

Поведение вынесено в отдельный объект и внедряется через конструктор. Хотите добавить реактивный полёт? Новый класс JetFly. Не трогаем ни Bird, ни существующие поведения.

Это же — суть Dependency Injection (глава 9), но на уровне дизайна.

4.7. Трейты: осторожно, хрупкость

Трейты — способ подмешать поведение в класс, минуя наследование. Полезны, но опасны.

Хороший пример: Timestampable — добавляет поля createdAt и updatedAt с логикой.

php

```
trait Timestampable
{
private \DateTimeImmutable $createdAt;
private ?\DateTimeImmutable $updatedAt = null;

public function initializeTimestamps(): void
{
$this->createdAt = new \DateTimeImmutable();
}

public function touch(): void
{
$this->updatedAt = new \DateTimeImmutable();
}

public function getCreatedAt(): \DateTimeImmutable
{
return $this->createdAt;
```

```
}

public function getUpdatedAt(): ?\DateTimeImmutable
{
return $this->updatedAt;
}
}
```

Плохой пример: трейт, который неявно зависит от свойств или методов использующего класса:

```
php

trait BadTrait
{
public function doStuff(): void
{
$this->logger->log('Что-то делаю...'); // Откуда logger? Магия.
}
}
```

Такой трейт предполагает, что класс предоставит свойство $logger. Забудете — фатальная ошибка. Статический анализ не поможет, пока не запустите.

Правило: трейт должен быть самодостаточным или явно

объявлять абстрактные методы, которые класс обязан реализовать. Лучше — композиция.

4.8. Собираем модель для менеджера финансов

Используем всё, что изучили:

```
php

enum TransactionType: string
{
case Income = 'income';
case Expense = 'expense';

public function sign(): int
{
return match($this) {
self::Income => 1,
self::Expense => -1,
};
}
}

readonly class Money
{
public function __construct(
public float $amount,
public string $currency,
```

```
    ) {
    if ($amount < 0) {
    throw new \InvalidArgumentException('Сумма не может быть отрицательной');
    }
    }

    public function add(self $other): self
    {
    if ($this->currency !== $other->currency) {
    throw new \InvalidArgumentException('Валюты не совпадают');
    }
    return new self($this->amount + $other->amount, $this->currency);
    }
    }

    readonly class Transaction
    {
    public function __construct(
    public Money $money,
    public TransactionType $type,
    public string $category,
    public \DateTimeImmutable $date,
    ) {}
```

```
public function signedAmount(): float
{
return $this->money->amount * $this->type->sign();
}
}
```

Заметили?

Нельзя создать Money с отрицательной суммой.

Нельзя создать Transaction с невалидным типом.

signedAmount() зависит только от собственных свойств — чистая функция внутри объекта.

Никаких сеттеров. Объекты создаются через конструктор и не меняются.

Это — доменная модель в миниатюре. В следующих главах мы добавим репозитории, сервисы и контроллеры, но сердце приложения останется здесь — в строгих, неизменяемых объектах.

4.9. Практическое задание

Деньги не горят:

Добавьте в класс Money метод subtract(Money $other): Money. Проверьте, что нельзя вычесть больше, чем есть (выбросить исключение). Проверьте совпадение валют.

Enum для категорий:

Создайте enum Category: string с кейсами: Salary, Food, Transport, Entertainment, Other. Добавьте метод isEssential(): bool — обязательные траты (еда, транспорт) возвращают true.

Фигуры без наследования:

Реализуйте интерфейс Shape с методом area(): float. Создайте классы Square и Triangle, реализующие интерфейс. Напишите функцию totalArea(Shape ...$shapes): float, которая суммирует площади. Проверьте с разными фигурами.

Композиция в деле:

Создайте интерфейс NotificationChannel с методом send(string $message): void. Реализуйте EmailChannel и SmsChannel. Создайте класс Notifier, который принимает массив каналов в конструкторе и в методе notify(string $message) отправляет сообщение через все каналы. Проверьте с двумя разными наборами каналов.

Глава 5. База данных без магии

В которой мы подключаемся к SQLite одной строкой, навсегда забываем про SQL-инъекции и пишем Repository, который не стыдно показать коллегам.

5.1. Почему не ORM?

ORM (Doctrine, Eloquent) — отличные инструменты. Но прежде чем прятать SQL за объектами, нужно понять, что происходит на самом деле. Когда вы знаете PDO и чистый SQL, ORM становится удобной абстракцией, а не магическим чёрным ящиком, который иногда делает 500 запросов там, где нужен один.

В этой главе мы используем SQLite — базу данных, которая хранится в одном файле и не требует сервера. Для продакшена замените на PostgreSQL или MySQL, но PDO-интерфейс останется тем же.

5.2. Подключение через PDO

PDO (PHP Data Objects) — единый интерфейс для работы с разными базами данных. Подключаемся:

php

```
// Где хранить базу? В файле рядом с проектом.
$dsn = 'sqlite:' . __DIR__ . '/database.sqlite';

$pdo = new PDO($dsn);
$pdo->setAttribute(PDO::ATTR_ERRMODE,
PDO::ERRMODE_EXCEPTION);
$pdo-
>setAttribute(PDO::ATTR_DEFAULT_FETCH_MODE,
PDO::FETCH_ASSOC);
```

Три важные строчки:

ERRMODE_EXCEPTION — любая ошибка SQL превращается в исключение. Без этого PDO молча возвращает false, и мы гадаем, почему данных нет.

FETCH_ASSOC — результаты запросов возвращаем как ассоциативные массивы. Никаких дублирующихся числовых ключей.

DSN начинается с sqlite: — для MySQL было бы mysql:host=...;dbname=....

Проверка:

```
php
```

$pdo->exec('SELECT 1'); // Если не упало — подключение работает.

5.3. Создание таблиц: миграции на чистом SQL

Создадим таблицу для транзакций, используя нашу доменную модель из главы 4:

php

$pdo->exec('
CREATE TABLE IF NOT EXISTS transactions (
id INTEGER PRIMARY KEY AUTOINCREMENT,
amount REAL NOT NULL,
currency TEXT NOT NULL,
type TEXT NOT NULL,
category TEXT NOT NULL,
date TEXT NOT NULL,
created_at TEXT NOT NULL DEFAULT (datetime(\'now \'))
)
');

Пояснения:

REAL для суммы (SQLite не различает float и double).

TEXT для даты — храним в ISO 8601 (2026-03-15), PHP легко парсит.

type и currency — текстовые коды. В реальном проекте можно сделать отдельные таблицы-справочники с внешними ключами, но для нашего менеджера финансов enum-подход из PHP надёжнее (контроль на уровне кода, а не БД).

created_at с дефолтным значением — метка времени создания записи.

Это простейшая миграция. В production-проектах используют инструменты вроде Phinx или Doctrine Migrations, но сейчас нам важна суть.

5.4. Подготовленные выражения: щит от инъекций

SQL-инъекция — до сих пор одна из главных уязвимостей веб-приложений. Злоумышленник передаёт '; DROP TABLE users; -- в поле ввода, и...

php

```
// НИКОГДА ТАК НЕ ДЕЛАЙТЕ:
$name = $_GET['name'];
$sql = "SELECT * FROM users WHERE name = '{$name}'";
// Если $name = "'; DROP TABLE users; --", будет больно.
```

Правильный способ — подготовленные выражения:
php

```
$name = $_GET['name'];
$stmt = $pdo->prepare('SELECT * FROM users WHERE name = :name');
$stmt->execute(['name' => $name]);
$user = $stmt->fetch();
```

PDO отправляет SQL и данные отдельно. Движок базы данных компилирует запрос, а потом подставляет параметры как данные — без интерпретации как SQL-кода. Инъекция невозможна физически.

Именованные параметры vs позиционные:
php

```
// Именованные (рекомендую):
$stmt = $pdo->prepare('SELECT * FROM users WHERE id = :id');
$stmt->execute(['id' => $userId]);

// Позиционные (?):
$stmt = $pdo->prepare('SELECT * FROM users WHERE id = ?');
```

```
$stmt->execute([$userId]);
```

Именованные читаемее и не ломаются при добавлении новых параметров.

Вставка данных:
php

```
$stmt = $pdo->prepare('
INSERT INTO transactions (amount, currency, type, category, date)
VALUES (:amount, :currency, :type, :category, :date)
');

$stmt->execute([
'amount' => 1500.00,
'currency' => 'RUB',
'type' => 'income',
'category' => 'Зарплата',
'date' => '2026-03-15',
]);

$newId = (int) $pdo->lastInsertId(); // ID новой записи.
```

Конец ознакомительного фрагмента.