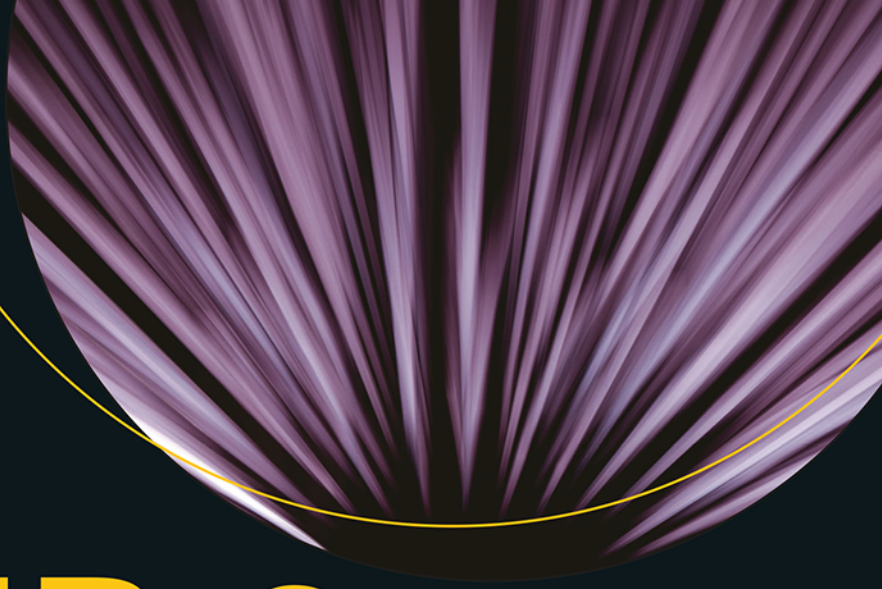

PHP 8

Obiekty, wzorce, narzędzia

Poznaj obiektowe usprawnienia języka PHP,
wzorce projektowe i niezbędne narzędzia programistyczne

Wydanie VI

—

Matt Zandstra

Helion

Apress

Tytuł oryginału: PHP 8 Objects, Patterns, and Practice Mastering OO Enhancements, Design Patterns, and Essential Development Tools, 6th Edition

Tłumaczenie: Piotr Cieślak, Przemysław Szeremiota

ISBN: 978-83-8322-926-3

First published in English under the title PHP 8 Objects, Patterns, and Practice; Mastering OO Enhancements, Design Patterns, and Essential Development Tools by Matt Zandstra, edition: 6

Copyright © Matt Zandstra, 2021

This edition has been translated and published under licence from APress Media, LLC, part of Springer Nature. APress Media, LLC, part of Springer Nature takes no responsibility and shall not be made liable for the accuracy of the translation.

Polish edition copyright © 2024 by Helion S.A.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiejkolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz wydawca dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz wydawca nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/php8o6
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
https://ftp.helion.pl/przyklady/php8o6.zip

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

- Kup książkę
- Poleć książkę
- Oceń książkę

- Księgarnia internetowa
- Lubię to! » Nasza społeczność

Spis treści

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Kup książkę Poleć książkę

ROZDZIAŁ 3.

Obiektowy elementarz

Obiekty i klasy są sercem tej książki, a od momentu wprowadzenia PHP5 nieco ponad dekadę temu stanowią również serce języka PHP. Niniejszy rozdział ma przygotować niezbędny grunt dla pogłębionego omówienia obiektów i projektowania obiektowego w kontekście mechanizmów języka PHP. Ci, którym programowanie obiektowe w tym języku jest całkiem obce, powinni przeczytać ten rozdział bardzo uważnie.

Omawiam w nim następujące zagadnienia:

- *Klasy i obiekty* — deklarowanie klas i tworzenie ich egzemplarzy (obiektów).
- *Konstruktory* — automatyzację konfiguracji obiektów.
- *Typy elementarne i klasy* — jakie znaczenie ma typ w PHP.
- *Dziedziczenie* — gdzie się przydaje i jak je stosować.
- *Widoczność* — udostępnianie interfejsów klas i zabezpieczanie metod i składowych obiektów przed ingerencją.

Klasy i obiekty

Pierwszą przeszkodą na drodze do zrozumienia programowania obiektowego jest dziwaczność i niezwykłość relacji pomiędzy klasą a jej obiektami. Dla wielu osób właśnie pojęcie tej relacji stanowi pierwsze olśnienie, wywołuje pierwszą ekscytację programowaniem obiektowym. Nie skąpmy więc energii na poznawanie podstaw.

Pierwsza klasa

Klasy są często opisywane w odniesieniu do obiektów. To bardzo ciekawe, ponieważ obiekty są z kolei niejednokrotnie opisywane przez pryzmat klas. Ta zależność bardzo spowalnia pierwsze postępy adeptów programowania obiektowego. Ponieważ to klasy definiują obiekty, zaczniemy od definicji klasy.

Krótko mówiąc, klasa to swego rodzaju szablon wykorzystywany do generowania jednego lub większej liczby obiektów. Deklaracja klasy zawiera słowo kluczowe `class` i dowolnie wybraną nazwę klasy.

Kup książkę Poleć książkę

Nazwa klasy może być dowolną kombinacją cyfr i liter, nie może się jednak od cyfry zaczynać. Nazwy mogą też zawierać znaki podkreślenia. Kod skojarzony z klasą musi być ograniczony nawiasami klamrowymi. Spróbujmy na podstawie tych informacji skonstruować klasę:

```
// listing 03.01
class ShopProduct
{
    // ciało klasy
}
```

Zdefiniowana właśnie klasa `ShopPoduct` jest prawidłową klasą, choć jej przydatność jest na razie znikoma. Mimo to osiągnęliśmy coś bardzo znaczącego, bo zdefiniowaliśmy typ. Utworzyliśmy więc kategorię danych, którą możemy wykorzystywać w skryptach. Znaczenie tego faktu stanie się niebawem jaśniejsze.

Pierwszy obiekt (lub dwa)

Skoro klasa jest szablonem generowania obiektów, obiekt stanowią dane, które zostały zaaranżowane zgodnie z szablonem definiowanym w klasie. Obiekt zwie się egzemplarzem klasy bądź jej konkretyzacją. Klasa definiuje jego typ.

Wykorzystamy teraz klasę `ShopProduct` jako formę do generowania obiektów typu `ShopProduct`. Pomoże nam w tym operator `new`. Operator `new` jest zazwyczaj stosowany w połączeniu z nazwą klasy, jak tutaj:

```
// listing 03.02
$product1 = new ShopProduct();
$product2 = new ShopProduct();
```

Operator `new`, jeśli zostanie wywoływany z nazwą klasy w roli swojego jedynego operandu, zwraca egzemplarz tejże klasy. W naszym przykładzie generuje obiekt klasy `ShopProduct`.

Klasę `ShopProduct` wykorzystaliśmy do wygenerowania dwóch obiektów typu `ShopProduct`. Choć funkcjonalnie `$product1` i `$product2` są identyczne (tzn. puste), stanowią obiekty rozłączne, a ich wspólnym mianownikiem jest jedynie wspólna klasa, na podstawie której zostały wygenerowane.

Istnienie klas i obiektów można sprowadzić do następującej analogii: klasa to jakby maszyna tłocząca gumowe kaczki. Obiektami zaś są owe gumowe kaczki produkowane przy użyciu maszyny. Ich „typ" jest wyznaczany formą, w której są odciskane. Wszystkie wypluwane z maszyny kaczki wyglądają więc identycznie, ale są niezależnymi od siebie obiektami materialnymi. Innymi słowy, są różnymi egzemplarzami pewnego przedmiotu. Aby rozróżnić poszczególne kaczki, można by im zresztą podczas wytłaczania nadawać numery seryjne. Każdy obiekt tworzony w języku PHP również posiada odrębną tożsamość, jednak unikatową jedynie w okresie życia danego obiektu (PHP ponownie wykorzystuje nieużywane już identyfikatory obiektów w obrębie tego samego procesu). Można się o tym przekonać, wydrukowując obiekty `$product1` i `$product2`:

```
// listing 03.03
var_dump($product1);
var_dump($product2);
```

Wykonanie powyższego kodu spowoduje wypisanie na wyjściu:

Kup książkę Poleć książkę

```
object(popp\r03\zestaw01\ShopProduct)#235 (0) {
}
object(popp\r03\zestaw01\ShopProduct)#234 (0) {
}
```

■ **Uwaga** W bardzo starych wersjach PHP (do wersji 5.1 włącznie) można było wyświetlić zawartość obiektu wprost. Taka operacja powoduje zrzutowanie obiektu na ciąg znaków zawierający identyfikator obiektu. Od PHP 5.2 język został pozbawiony tej automagicznej konwersji, więc traktowanie obiektu jako ciągu znaków jest błędem, chyba że w klasie obiektu jest zdefiniowana metoda `__toString()`. Metodami zajmiemy się w dalszej części tego rozdziału, a o metodzie `__toString()` będzie mowa w rozdziale 4.

Przekazanie obiektu do wywołania `var_dump()` pozwala wypisać ciekawe dane o obiekcie, z identyfikatorem obiektu na czele (po symbolu kratki).

Aby klasę `ShopProduct` uczynić ciekawszą, możemy uzupełnić ją o obsługę specjalnych pól danych, zwanych składowymi bądź właściwościami (ang. *properties*).

Definiowanie składowych klasy

Klasy mogą definiować specjalne zmienne zwane właściwościami bądź składowymi. Składowa przechowuje dane, które różnią się pomiędzy egzemplarzami danej klasy. W przypadku obiektów klasy `ShopProduct` (niech będzie to asortyment księgarski, choć ogólnie chodzi o dowolne artykuły) możemy na przykład zażyczyć sobie obecności pól ceny (`price`) i tytułu (`title`).

Składowa klasy przypomina zwykłą zmienną, z tym że przy deklarowaniu składowej jej nazwę trzeba poprzedzić słowem kluczowym określającym widoczność. Może być nim `public`, `protected` albo `private`. Wybór słowa kluczowego widoczności określa miejsca w kodzie, z których możliwe będzie odwoływanie się do danej składowej. Na przykład właściwości zadeklarowane jako `public` są dostępne spoza ciała klasy, a do właściwości `private` można się odwoływać tylko z kodu danej klasy.

Do kwestii widoczności i regulujących ją słów kluczowych wrócimy później. Na razie spróbujemy po prostu zadeklarować kilka składowych klasy za pomocą słowa kluczowego `public`:

```
// listing 03.04
class ShopProduct
{
    public $title             = "bez tytułu";
    public $producerMainName  = "nazwisko";
    public $producerFirstName = "imię";
    public $price             = 0;
}
```

Jak widać, uzupełniliśmy klasę o cztery składowe, przypisując do każdej z nich wartość domyślną. Wszelkie obiekty konkretyzowane na bazie takiej klasy będą teraz zawierać owe dane domyślne. Słowo kluczowe `public` poprzedzające deklarację każdej składowej umożliwia odwoływanie się do niej spoza kontekstu obiektu.

Do składowych definiowanych w obrębie klasy, a konkretyzowanych w obiektach możemy się odwoływać za pośrednictwem ciągu znaków „->” (operatora obiektów) w połączeniu ze zmienną w postaci obiektu oraz nazwą składowej:

Kup książkę Poleć książkę

```
// listing 03.05
$product1 = new ShopProduct();
print $product1->title;
```

```
bez tytułu
```

Ponieważ składowe zostały zdefiniowane jako publiczne (`public`), możemy odczytywać ich wartości i je do nich przypisywać, zmieniając domyślne stany obiektów definiowane w klasie:

```
// listing 03.06
$product1 = new ShopProduct();
$product2 = new ShopProduct();
$product1->title = "Moja Antonia";
$product2->title = "Paragraf 22";
```

Deklarując i ustawiając w klasie `ShopProduct` składową `$title`, wymuszamy podaną początkową wartość tej składowej we wszystkich nowo tworzonych obiektach klasy. Oznacza to, że kod użytkujący klasę może bazować na domniemaniu obecności tej składowej w każdym z obiektów klasy. Nie może już jednak domniemywać wartości składowych, gdyż te — jak wyżej — mogą się w poszczególnych obiektach różnić między sobą.

■ **Uwaga** Kod wykorzystujący klasę, funkcję bądź metodę nazwiemy *kodem klienckim* wobec tej klasy, metody czy funkcji, albo po prostu *klientem klasy* (metody, funkcji). Termin „klient" będzie się więc w tej książce pojawiał stosunkowo często.

Zresztą PHP nie zmusza nas do deklarowania wszystkich składowych w klasie. Obiekty można uzupełniać składowymi dynamicznie, jak tutaj:

```
// listing 03.07
$product1->arbitraryAddition = "nowość";
```

Taka metoda uzupełniania obiektów o składowe nie jest jednak zalecana w programowaniu obiektowym.

Czy dynamiczne uzupełnianie składowych to zła praktyka? Definiując klasę, definiuje się typ obiektów. Informuje się tym samym otoczenie, że dana klasa (i wszelkie jej konkretyzacje w postaci obiektów) składa się z ustalonego zestawu pól danych i funkcji. Jeśli klasa `ShopProduct` definiuje składową `$title`, wtedy dowolny kod użytkujący obiekty klasy `ShopProduct` może śmiało odwoływać się do składowej `$title`, ponieważ jej dostępność jest pewna. Nie da się podobnej pewności stosowania uzyskać względem składowych dodawanych do obiektów w sposób dynamiczny.

Nasze obiekty są na razie cokolwiek nieporęczne. Chcąc manipulować ich składowymi, musimy bowiem czynić to poza samymi obiektami. Sięgamy do nich jedynie celem ustawienia i odczytania składowych. Konieczność ustawienia wielu takich składowych szybko stanie się wyjątkowo uciążliwa:

```
// listing 03.08
$product1 = new ShopProduct();
$product1->title = "Moja Antonia";
$product1->producerMainName  = "Cather";
$product1->producerFirstName = "Willa";
$product1->price = 5.99;
```


Kup książkę Poleć książkę

W powyższym kodzie zamazaliśmy jedną po drugiej pierwotne, definiowane w klasie wartości składowych obiektów, aż wprowadziliśmy komplet pożądanych zmian obiektu. Po arbitralnym wymuszeniu wartości składowych możemy się swobodnie odwołać do nowych wartości:

```
// listing 03.09
print "Autor: {$product1->producerFirstName} "
            ."{$product1->producerMainName}\n";
```

Powyższy kod wypisze na wyjściu programu:

```
Autor: Willa Cather
```

Taka metoda ustawiania i odwoływania się do składowych powoduje wiele problemów. Największym jest potencjalne niebezpieczeństwo dynamicznego uzupełnienia zestawu składowych obiektu w wyniku literówki w odwołaniu. O taką pomyłkę naprawdę łatwo — załóżmy, że chcemy napisać tak:

```
// listing 03.10
$product1->producerFirstName = "Shirley";
$product1->producerMainName = "Jackson";
```

Tymczasem przez pomyłkę wpisujemy następujący kod:

```
// listing 03.11
$product1->producerFirstName = "Shirley";
$product1->producerSecondName = "Jackson";
```

Z punktu widzenia samego języka PHP kod taki byłby jak najbardziej dozwolony, więc programista nie otrzymałby żadnego ostrzeżenia. Ale kiedy przyszłoby do wyprowadzania nazwiska autora (ogólnie: wytwórcy), wyniki odbiegałyby od oczekiwanych.

Kolejnym problemem jest zbytnie rozluźnienie relacji pomiędzy składowymi klasy. Nie mamy obowiązku ustawiać tytułu, ceny czy nazwiska autora — użytkownik obiektu może być pewien, że obiekt takie składowe posiada, ale nie ma żadnej gwarancji przypisania do nich jakichkolwiek wartości (poza ewentualnymi wartościami domyślnymi). Tymczasem najlepiej byłoby, gdyby każdy nowo utworzony obiekt `ShopProduct` posiadał znaczące wartości swoich składowych.

Wreszcie traktowanie z osobna każdej składowej jest nużące, zwłaszcza kiedy zamierzamy robić to częściej. Już samo wyświetlenie nazwiska autora jest uciążliwe.

Byłoby miło, gdyby podobne zadania dało się złożyć na barki samego obiektu.

Wszystkie te problemy można wyeliminować, uzupełniając klasę `ShopProduct` o zestaw własnych funkcji, które pośredniczyłyby w manipulowaniu składowymi, operując nimi z poziomu kontekstu obiektu.

Metody

Składowe pozwalają obiektom na przechowywanie danych, metody zaś umożliwiają obiektom wykonywanie zadań. Metody to specjalne funkcje (zwane też niekiedy funkcjami składowymi), deklarowane we wnętrzu klasy. Jak można się spodziewać, deklaracja metody przypomina deklarację zwykłej funkcji. Nazwę metody poprzedza słowo kluczowe `function`, a uzupełnia ją opcjonalna lista parametrów ujęta w nawiasy. Ciało metody ograniczone jest nawiasami klamrowymi:

Kup książkę Poleć książkę

```
// listing 03.12
public function myMethod($argument, $another)
{
    // ...
}
```

W przeciwieństwie do zwykłych funkcji metody muszą być deklarowane w ciele klasy. Mogą też być opatrywane szeregiem modyfikatorów, w tym słowem kluczowym określającym widoczność. Podobnie jak składowe, tak i metody można deklarować jako publiczne (`public`), chronione (`protected`) albo prywatne (`private`). Deklarując metodę jako publiczną, umożliwiamy wywoływanie jej spoza kontekstu obiektu. Pominięcie określenia widoczności w deklaracji metody oznacza niejawnie jej widoczność i dostępność publiczną. Za dobrą praktykę uważa się jednak jawne deklarowanie widoczności dla wszystkich metod (do modyfikatorów metod wrócimy w dalszej części tego rozdziału).

■ **Uwaga** W rozdziale 15. zostały omówione zalecane praktyki kodowania. Twórcy standardu PSR-12 zalecają deklarowanie widoczności dla wszystkich metod.

```
// listing 03.13
class ShopProduct
{
    public $title             = "bez tytułu";
    public $producerMainName  = "nazwisko";
    public $producerFirstName = "imię";
    public $price             = 0;

    public function getProducer()
    {
        return $this->producerFirstName . " "
               . $this->producerMainName;
    }
}
```

W większości przypadków metoda wywoływana jest na rzecz konkretnego obiektu, z którym jej nazwa jest kojarzona operatorem dostępu do składowej `->`. Nazwa metody musi być w wywołaniu uzupełniona nawiasami — niezależnie od tego, czy metoda przyjmuje jakiekolwiek argumenty (dokładnie tak jak w funkcji).

```
// listing 03.14
$product1 = new ShopProduct();
$product1->title = "Moja Antonia";
$product1->producerMainName = "Cather";
$product1->producerFirstName = "Willa";
$product1->price = 5.99;

print "Autor: {$product1->getProducer()}\n";
```

Na wyjściu programu uzyskamy:

```
Autor: Willa Cather
```


Kup książkę Poleć książkę

Do klasy `ShopProduct` dodaliśmy metodę `getProducer()`. Warto zauważyć, że metoda `getProducer()` została zadeklarowana jako publiczna, co oznacza, że da się ją wywołać spoza wspomnianej klasy.

W ciele niniejszej metody pojawiła się pewna nowinka. Chodzi o pseudozmienną `$this`, za pośrednictwem której kod klasy odwołuje się do egzemplarza klasy, na rzecz którego metoda została wywołana. Wszelkie wątpliwości co do znaczenia `$this` w kodzie klasy należy rozstrzygać, zastępując zmienną wyrażeniem „bieżący egzemplarz klasy". Stąd instrukcja:

```
$this->producerFirstName
```

oznacza tyle, co:

```
Składowa $producerFirstName bieżącego egzemplarza klasy
```

Jak widać, metoda `getProducer()` realizuje i zwraca konkatenację składowych `$producerFirstName` i `$producerMainName`. Obecność tej metody oszczędza nam odwołań do poszczególnych składowych i własnoręcznego konstruowania ciągu nazwiska autora.

Tym sposobem ulepszyliśmy nieco naszą klasę. Mimo to nie uniknęliśmy pułapki nadmiernej elastyczności — inicjalizację obiektów klasy `ShopProducer` składamy bowiem na barki programisty kodu klienckiego klasy `ShopProduct` i musimy polegać na jego solidności. Poprawna i pełna inicjalizacja obiektu naszej klasy wymaga pięciu wierszy kodu (pięciu instrukcji) — żaden programista nam za to nie podziękuje. A do tego jako twórcy klasy nie mamy możliwości zagwarantowania prawidłowej inicjalizacji którejkolwiek ze składowych obiektów klasy `ShopProduct` w kodzie klienckim. Potrzebowalibyśmy do tego metody wywoływanej automatycznie w przebiegu konkretyzacji obiektu.

Metoda konstrukcji obiektu

Metoda konstrukcji obiektu, zwana po prostu konstruktorem, wywoływana jest w ramach konkretyzacji, czyli tworzenia obiektu klasy. W jej ramach można wykonać operacje inicjalizujące obiekt oraz wykonujące pewne przewidziane dla całej klasy operacje wstępne.

■ **Uwaga** W wersjach PHP poprzedzających wersję piątą konstruktor przyjmował nazwę klasy, w ramach której operował — klasa `ShopProduct` miała więc zawsze konstruktor `ShopProduct()`. Rozwiązanie to zostało uznane za przestarzałe w PHP7, a w PHP8 wycofano je w ogóle. Nadawaj konstruktorowi nazwę w postaci `__construct()`.

Zauważmy, że nazwa konstruktora zaczyna się od dwóch znaków podkreślenia, charakterystycznych również dla wielu innych specjalnych metod deklarowanych w klasach PHP. Zdefiniujmy więc konstruktor klasy `ShopProduct`:

■ **Uwaga** Wbudowane metody, których nazwa zaczyna się w ten sposób, nazywa się *magicznymi*, bo są wywoływane automatycznie w określonych okolicznościach. Więcej informacji na ich temat można znaleźć w systemie pomocy PHP pod adresem *www.php.net/manual/en/language.oop5.magic.php*. Ze względu na tak specyficzne zastosowanie podwójnego podkreślenia dobrze jest unikać poprzedzania nim nazw własnych metod, choć nie jest to stricte niedozwolone.

Kup książkę Poleć książkę

```
// listing 03.15
class ShopProduct
{
    public $title;
    public $producerMainName;
    public $producerFirstName;
    public $price = 0;

    public function __construct(
        $title,
        $firstName,
        $mainName,
        $price
    ) {
        $this->title = $title;
        $this->producerFirstName = $firstName;
        $this->producerMainName = $mainName;
        $this->price = $price;
    }
    public function getProducer()
    {
        return $this->producerFirstName . " "
            . $this->producerMainName;
    }
}
```

Znów ulepszyliśmy nieco klasę, oszczędzając sobie i innym użytkownikom klasy konieczności zwielokrotniania kodu inicjalizacji każdego obiektu z osobna. Teraz przy okazji konkretyzacji obiektu w ramach operatora `new` wywoływana jest każdorazowo metoda `__construct()`. Obiekt tworzy się teraz tak:

```
// listing 03.16
$product1 = new ShopProduct(
    "Moja Antonia",
    "Willa",
    "Cather",
    5.99
);
print "Autor: {$product1->getProducer()}\n";
```

Na wyjściu tego programu otrzymamy:

```
Autor: Willa Cather
```

Wszelkie argumenty przekazane w wywołaniu `new` są przekazywane do konstruktora klasy. W naszym przykładzie przekazujemy w ten sposób do konstruktora tytuł, imię i nazwisko autora oraz cenę książki. Konstruktor w swoim ciele odwołuje się do składowych tworzonego obiektu za pośrednictwem pseudozmiennej `$this`.

Kup książkę Poleć książkę

■ **Uwaga** Obiekty klasy `ShopProduct` dają się teraz tworzyć znacznie łatwiej i bezpieczniej. Całość operacji związanych z inicjalizacją realizuje z punktu widzenia użytkownika pojedyncze wywołanie operatora `new`. Teraz w kodzie wykorzystującym obiekty klasy `ShopProduct` można w pełni polegać na prawidłowej inicjalizacji wszystkich składowych obiektu.

Właściwość nie musi być inicjalizowana i samo w sobie nie jest to błędem. Ale każda próba uzyskania dostępu do takiej właściwości wywoła błąd krytyczny.

Promocja składowych konstruktora

Choć klasa `ShopProduct` jest teraz bezpieczniejsza i wygodniejsza z perspektywy jej klienta, wprowadziliśmy do niej sporo szablonowego, powtarzalnego kodu. Przyjrzyjmy się jej raz jeszcze w obecnej postaci. Aby utworzyć instancję (egzemplarz) obiektu z czterema składowymi, potrzebujemy łącznie trzech zbiorów odwołań do danych. Najpierw deklarujemy składowe, potem podajemy argumenty konstruktora do przechowywania danych, a następnie zbieramy to wszystko podczas przypisywania argumentów metody właściwościom. Język PHP8 został wyposażony w nową funkcję o nazwie *constructor property promotion* (dosł. promocja składowych konstruktora), która pozwala na użyteczne uproszczenia. Dzięki uwzględnieniu określającego widoczność słowa kluczowego dla argumentów konstruktora można połączyć je z deklaracjami właściwości i *równocześnie* je przypisać. Oto nowa wersja klasy `ShopProduct`:

```
// listing 03.17
class ShopProduct
{
    public function __construct(
        public $title,
        public $producerFirstName,
        public $producerMainName,
        public $price
    ) {
    }
    public function getProducer()
    {
        return $this->producerFirstName . " "
            . $this->producerMainName;
    }
}
```

Zarówno deklaracja, jak i przypisanie w metodzie konstruktora są obsługiwane niejawnie. Ograniczenie powtórzeń pozwala zarazem ograniczyć ryzyko błędów w kodzie. Ponadto skrócenie kodu klasy ułatwia innym zinterpretowanie go i skupienie się na jego logice.

■ **Uwaga** Promocja właściwości została wprowadzona w PHP8. Jeśli projekt jest przeznaczony do działania w środowisku PHP7, należy zrezygnować z wykorzystywania nowej składni.

Pewność i przewidywalność to bardzo istotne aspekty programowania obiektowego. Klasy należy projektować tak, aby ich użytkownicy mogli w sposób pewny wykorzystywać ich cechy. Jeden ze sposobów na poprawienie bezpieczeństwa stosowania obiektu polega na zadeklarowaniu

Kup książkę Poleć książkę

w przewidywalny sposób typów danych przechowywanych we właściwościach (składowych) tego obiektu. Można na przykład zagwarantować, by składowa `$name` zawsze zawierała tylko dane znakowe. Tylko jak to osiągnąć, gdy dane składowej są przekazywane poza obręb klasy? Mechanizmom wymuszania konkretnego typu obiektów w deklaracjach metod przyjrzymy się w następnym podrozdziale.

Argumenty domyślne i nazwane

Z biegiem czasu lista argumentów może się wydłużać, co utrudnia posługiwanie się nią. Praca z klasą staje się wtedy coraz bardziej uciążliwa, bo trudno jest zapanować nad wszystkimi argumentami żądanymi przez jej metody. Aby ułatwić pracę programistom kodującym aplikacje klienckie, warto określić wartości domyślne w definicji metody. Przypuśćmy na przykład, że w przypadku obiektu `ShopProduct` potrzebujemy tytułu, ale imię i nazwisko twórcy mogą być pustymi ciągami znaków, a cena może wynosić zero. W bieżącej formie kod korzystający z obiektu `ShopProduct` musiałby dostarczyć wszystkie te dane:

// listing 03.18
```
$product1 = new ShopProduct("Katalog", "", "", 0);
```

Możemy uprościć tworzenie instancji obiektu, definiując wartości domyślne dla wybranych argumentów. Właśnie tak zostało to zrobione w następnym przykładzie:

// listing 03.19
```
class ShopProduct
{
    public function __construct(
        public $title,
        public $producerFirstName = "",
        public $producerMainName = "",
        public $price = 0
    ) {
    }
    // ...
}
```

Te przypisania zostaną uaktywnione jedynie wówczas, gdy kod wywołujący nie zapewni wartości odpowiednich argumentów. Teraz wywołanie konstruktora może zawierać tylko jedną wartość: tytuł.

// listing 03.20
```
$product1 = new ShopProduct("Katalog");
```

Domyślne wartości argumentów sprawiają, że praca z metodami jest wygodniejsza, ale jak to zwykle bywa z wygodnymi rozwiązaniami, mogą prowadzić do nieprzewidzianych komplikacji. Co by się stało z naszym elegancko skróconym wywołaniem konstruktora, gdybyśmy chcieli podać cenę, ale pozostawić puste, domyślne wartości imienia i nazwiska twórcy? W PHP starszych od wersji 8 bylibyśmy w kropce. Aby określić cenę, trzeba byłoby podać puste ciągi znaków jako personalia autora. W ten sposób zatoczylibyśmy koło. Trzeba byłoby też się zastanowić, jakich rodzajów wartości oczekuje konstruktor w przypadku pustego imienia i nazwiska. Przekazać puste łańcuchy znaków? Wartości `null`? Próba wprowadzenia obsługi wartości domyślnych nie tylko nie zaoszczędziłaby nam pracy, ale mogłaby wprowadzić zamieszanie.

Kup książkę Poleć książkę

Na szczęście PHP8 oferuje możliwość stosowania argumentów nazwanych. W metodzie wywołującej można teraz podać nazwę każdego argumentu poprzedzającą wartość do przekazania. PHP powiąże wówczas wartość z właściwym argumentem w metodzie bez względu na kolejność argumentów w kodzie wywołującym.

```
// listing 03.21
$product1 = new ShopProduct(
    price: 0.7,
    title: "Katalog"
);
```

Warto zwrócić uwagę na składnię: informujemy PHP, że chcemy nadać argumentowi `$price` wartość `0.7`, poprzez podanie najpierw nazwy argumentu (`price`), średnika i dopiero potem wartości. W przypadku zastosowania argumentów nazwanych ich kolejność nie ma znaczenia — no i nie musimy już podawać pustych wartości dla imienia i nazwiska autora.

Typy argumentów metod

Typy określają w skryptach sposób zarządzania danymi. Typy łańcuchowe są wykorzystywane do przechowywania i wyświetlania ciągów znaków oraz do manipulowania takimi ciągami za pośrednictwem odpowiednich funkcji. Zmienne liczbowe są wykorzystywane w wyrażeniach matematycznych. Zmienne logiczne osadzane są w wyrażeniach logicznych. Tego rodzaju typy zaliczamy do typów elementarnych (ang. *primitive types*). Klasy stanowią w systemie typów znacznie wyższy poziom. Obiekt klasy `ShopProduct` stanowi wartość elementarnego typu „obiekt”, ale równocześnie jest wcieleniem (egzemplarzem) konkretnej klasy — `ShopProduct`. Zajmijmy się więc relacjami typów i metod.

Definicje metod i funkcji nie muszą nakładać na parametry żadnych ograniczeń co do typów. To zarówno możliwość zbawienna, jak i katastrofalna. Fakt, że argument wywołania funkcji może być dowolnego typu, daje niezrównaną elastyczność. Można dzięki temu konstruować metody reagujące inteligentnie na różne przekazywane do nich dane, dostosowując realizowane w nich funkcje do okoliczności wywołania. Elastyczność ta jest jednak równocześnie przyczyną niejednoznaczności i nieoczekiwanego działania kodu, kiedy przekazany argument jest typu innego niż spodziewany.

Typy elementarne

PHP to język o osłabionej kontroli typów (tzw. słabe typowanie). Oznacza to, że deklaracja zmiennej nie musi określać i narzucać jej typu. Zmienna `$number` może w ramach jednego zasięgu zostać zainicjalizowana wartością liczbową 2, a za chwilę nadpisana ciągiem `"dwa"`. W językach ze ścisłą kontrolą typów, jak C i Java, typ zmiennej musi być zadeklarowany jeszcze przed przypisaniem jej wartości, a przypisywana wartość musi być typu zgodnego z deklarowanym.

Nie oznacza to, że w języku PHP w ogóle nie istnieje pojęcie typu. Każda wartość, którą da się przypisać do zmiennej, posiada typ. Typ zmiennej można określić za pośrednictwem jednej z wbudowanych funkcji PHP. Lista typów elementarnych wyróżnianych w PHP wraz z funkcjami wykrywającymi przynależność do tegoż typu widnieje w tabeli 3.1. Każda z tych funkcji przyjmuje w wywołaniu badaną zmienną i zwraca wartość `true`, jeśli zmienna ta należy do określonego typu.

Kup książkę Poleć książkę

Tabela 3.1. *Typy elementarne i funkcje kontroli typów w PHP*

Funkcja testująca przynależność do typu	Nazwa typu	Opis
`is_bool()`	`boolean`	Jedna z dwóch wyróżnionych wartości: `true` („prawda”) i `false` („fałsz”).
`is_integer()`	`integer`	Liczba całkowita (równoważne z wywołaniami `is_int()` i `is_long()`).
`is_float()`	`float`	Liczba zmiennoprzecinkowa (z częścią ułamkową; równoważne wywołaniu `is_double()`).
`is_string()`	`string`	Ciągi znaków.
`is_object()`	`object`	Obiekt.
`is_resource()`	`resource`	Uchwyt identyfikujący i pośredniczący w komunikacji z zasobami zewnętrznymi, np. bazami danych i plikami.
`is_array()`	`array`	Tablica.
`is_null()`	`null`	Wartość pusta.

Sprawdzanie typu wartości w PHP ma szczególne znaczenie przy przetwarzaniu argumentów wywołania funkcji i metod.

Typy elementarne — przykład

W kodzie trzeba bezwzględnie kontrolować wykorzystywane typy. Spójrzmy na przykład jednego z wielu problemów związanych z systemem typów.

Wyobraźmy sobie, że skrypt wyodrębnia konfigurację aplikacji z pliku XML. Element XML `<resolvedomains></resolvedomains>` instruuje aplikację co do podejmowania próby odwzorowania adresu IP na nazwę domenową — często odwzorowanie takie jest przydatne, ale zazwyczaj jest operacją stosunkowo skomplikowaną.

Oto próbka pliku konfiguracyjnego:

```
// listing 03.22
<settings>
    <resolvedomains>false</resolvedomains>
</settings>
```

Skrypt wyodrębnia z pliku konfiguracyjnego ciąg `"false"` i przekazuje go w roli znacznika do metody o nazwie `outputAddresses()`, wyświetlającej dane adresowe (IP i ewentualnie — w zależności od wartości znacznika — nazwę domenową). Oto kod metody `outputAddresses()`:

```
// listing 03.23
class AddressManager
{
    private $addresses = ["209.131.36.159", "216.58.213.174"];
    public function outputAddresses($resolve)
    {
        foreach ($this->addresses as $address) {
            print $address;
            if ($resolve) {
```


Kup książkę Poleć książkę

```
                print " (".gethostbyaddr($address).")";
            }
            print "\n";
        }
    }
}
```

Klasa `AddressManager` mogłaby oczywiście zostać nieco ulepszona; wpisywanie na sztywno adresu IP w kodzie klasy rzadko kiedy jest dobrym pomysłem. Tak czy inaczej metoda `outputAddresses()` przegląda tablicę ze składowej `$addresses` i wypisuje wartości poszczególnych elementów tablicy. Jeśli parametr `$resolve` ma wartość `true`, obok adresów IP wyprowadzane są nazwy domenowe.

Oto nieco inne podejście, z wykorzystaniem w klasie `AddressManager` pliku konfiguracyjnego w formacie XML. Zobaczmy, czy uda się nam wychwycić słabość tego wariantu:

```
// listing 03.24
$settings = simplexml_load_file(__DIR__."/resolve.xml");
$manager = new AddressManager();
$manager->outputAddresses((string)$settings->resolvedomains);
```

Celem wyodrębnienia z pliku ustawień wartości elementu `resolvedomains` odwołujemy się tu do SimpleXML API. Wiemy skądinąd, że wartością owego elementu jest u nas ciąg znaków `"false"` i zgodnie z dokumentacją SimpleXML rzutujemy tę wartość na typ `string`.

Kod, niestety, nie będzie zachowywał się prawidłowo. Otóż przekazując w wywołaniu metody `outputAddresses()` ciąg `"false"`, wykazujemy się niezrozumieniem niejawnego założenia, jakie metoda czyni odnośnie do wartości argumentu wywołania. Otóż metoda spodziewa się przekazania wartości logicznej (czyli wartości `true` albo `false`). Tymczasem ciąg `"false"` nie jest wartością logiczną, a co gorsza, jeśli już użyjemy go w roli takiej wartości, da wartość `true`. PHP wykona bowiem rzutowanie niepustego ciągu znaków na typ logiczny, a w dziedzinie wartości typu logicznego niepusty ciąg znaków reprezentowany jest jako `true`. Dlatego następujący kod:

```
if ("false") {
    // ...
}
```

jest w istocie równoznaczny z:

```
if (true) {
    // ...
}
```

Błąd tego rodzaju można wyeliminować na kilka sposobów.

Można metodę `outputAddresses()` uodpornić na mylne interpretacje typów argumentów, wyposażając ją w kod rozpoznający argument typu ciągu znaków i konwertujący taki ciąg na wartość logiczną wedle własnych kryteriów:

```
// listing 03.25
    public function outputAddresses($resolve)
    {
        if (is_string($resolve)) {
            $resolve =
                (preg_match("/^(false|no|off)$/i", $resolve) ) ? false : true
        }
        // ...
    }
```


Kup książkę Poleć książkę

Istnieją jednak solidne przesłanki do unikania takich sposobów. W zasadzie lepiej jest udostępnić przejrzysty, zwarty i ograniczony interfejs metody niż interfejs otwarty i wieloznaczny. Funkcje i metody przyjmujące niejasne semantycznie argumenty prowokują bowiem do niechlujnego stosowania, a więc i do wprowadzania błędów użycia.

Można jeszcze inaczej: zostawić ciało metody `outputAddresses()` w spokoju, opatrując jej deklarację komentarzem dającym użytkownikom jasność co do wymagań metody wobec typu argumentu `$resolve` i jego interpretacji w ciele funkcji. Decydujemy się tym samym na złożenie odpowiedzialności za poprawne działanie metody na barki użytkownika.

```
// listing 03.26
/**
 * Wyświetla listę adresów.
 * Przy wartości true argumentu $resolve adresy będą odwzorowywane do nazwy domenowej.
 * @param $resolve Boolean Wyszukać nazwy domenowe?
 */
public function outputAddresses($resolve) {
    // ...
}
```

To całkiem niezłe rozwiązanie, pod warunkiem, że programiści mający używać klasy są uważnymi czytelnikami dokumentacji (albo posługują się inteligentnymi edytorami kodu, które rozpoznają tego rodzaju adnotacje).

Wreszcie można też zmodyfikować metodę `outputAddreses()` tak, by rygorystycznie traktowała typ danych, dostarczony za pośrednictwem argumentu `$resolve`. W starszych niż 7 wersjach PHP w przypadku typów elementarnych, takich jak wartości boolowskie, dało się to zrobić tylko w jeden sposób. Trzeba było napisać kod weryfikujący przekazane dane i podejmujący jakieś działania, gdy typ tych danych różnił się od oczekiwanego:

```
// listing 03.27
public function outputAddresses($resolve)
{
    if (!is_bool($resolve)) {
        // podejmij drastyczne działania
    }
    // ...
}
```

Takie podejście można zastosować w celu wymuszenia na kodzie klienckim dostarczenia w argumencie `$resolve` właściwego typu danych, a gdy tak się nie stanie — wyświetlenia ostrzeżenia.

■ **Uwaga** W następnej części rozdziału, „Deklaracje typów — typy obiektowe", opiszę znacznie lepszy sposób ograniczania typów argumentów przekazywanych do metod i funkcji.

Konwersja argumentu typu łańcuchowego na typ logiczny w imieniu kodu wywołującego byłaby przyjaźniejsza dla użytkownika, ale rodzi wiele kolejnych problemów. Udostępniając mechanizm konwersji, skazujemy się na odgadywanie intencji wywołującego. Narzucając mu stosowanie typu logicznego, dajemy mu z kolei wolną rękę co do sposobu odwzorowywania wartości logicznych w ciągach znaków — klient sam decyduje, czy to dopuszczalne i jakie słowo reprezentuje dla niego „prawdę". Metoda `outputAddresses()` może zaś skupić się na swym podstawowym zadaniu, do którego została powołana. Tego rodzaju skupienie na własnych

Kup książkę Poleć książkę

zadaniach z celowym ignorowaniem szerszego kontekstu jest ważną zasadą programowania obiektowego i będę się na nią często w książce powoływać.

W istocie zaś strategie obsługi typów argumentów powinny być z jednej strony uzależnione od ważności ewentualnych błędów, a z drugiej — od korzyści związanych z elastycznością kodu. PHP potrafi rzutować wartości pomiędzy większością elementarnych typów, zależnie od zastanego kontekstu wykorzystania wartości. Na przykład liczby w ciągach znaków, jeśli ciągi te występują w wyrażeniach arytmetycznych, są konwertowane na postać ich całkowitych i zmiennoprzecinkowych odpowiedników. W kodzie można polegać na tej konwersji, czyniąc go odpornym na szereg błędów typowania.

Zasadniczo jednak lepiej jest skłaniać się ku większej ostrożności pod względem typów obiektowych i elementarnych. Na szczęście PHP8 jest wyposażony w nowe narzędzia zapewniające bezpieczeństwo typowania.

Niektóre inne funkcje kontroli typów

Znamy już funkcje obsługujące zmienne, które kontrolują typy elementarne. A skoro już mowa o sprawdzaniu zawartości zmiennych, warto wspomnieć o kilku funkcjach, które wykraczają poza zwykłą kontrolę typów elementarnych i dostarczają więcej informacji o potencjalnych sposobach wykorzystania danych znajdujących się w zmiennych. Funkcje te zostały wymienione w tabeli 3.2.

Tabela 3.2. *Funkcje sprawdzające cechy wartości*

Funkcja	Opis
`is_countable()`	Sprawdza, czy tablicę lub obiekt da się przekazać funkcji `count()`.
`is_iterable()`	Sprawdza, czy zawartość struktury danych można przetworzyć w pętli `foreach`.
`is_callable()`	Sprawdza, czy kod da się wywołać — często chodzi o funkcję anonimową lub nazwę funkcji.
`is_numeric()`	Sprawdza, czy wartość można potraktować jako liczbową — jest typu `int`, `long` lub łańcuchem (`string`) dającym się zinterpretować jako liczba.

Funkcje opisane w tabeli 3.2 sprawdzają nie tyle typy, co sposoby, w jakie można potraktować badane wartości. Jeśli funkcja `is_callable()` zwróci `true` w przypadku jakiejś zmiennej, to wiadomo, że zmienną tę można potraktować jak funkcję lub metodę i wywołać ją. Na tej samej zasadzie wartość, która przejdzie test funkcji `is_iterable()`, da się przetworzyć w pętli — choć nie musi to być tablica, tylko szczególny rodzaj obiektu.

Deklaracje typów — typy obiektowe

Jako że argument wywołania funkcji może reprezentować wartość dowolnego typu elementarnego, może też domyślnie reprezentować obiekt dowolnego typu. Taka elastyczność ma swoje zalety, ale powoduje też problemy, zwłaszcza w kontekście definicji metody.

Wyobraźmy sobie metodę pewnej klasy pomocniczej, przeznaczonej do manipulowania obiektami klasy `ShopProduct`:

Kup książkę Poleć książkę

```
// listing 03.28
class ShopProductWriter
{
    public function write($shopProduct)
    {
        $str  = $shopProduct->title . ": "
            . $shopProduct->getProducer()
            . " (" . $shopProduct->price . ")\n";
        print $str;
    }
}
```

Klasę tę możemy przetestować kodem:

```
// listing 03.29
$product1 = new ShopProduct("Moja Antonia", "Willa", "Cather", 5.99);
$writer = new ShopProductWriter();
$writer->write($product1);
```

Otrzymamy:

```
Moja Antonia: Willa Cather (5.99)
```

Klasa `ShopProductWriter` zawiera tylko jedną metodę — `write()`. Metoda ta przyjmuje za pośrednictwem argumentu wywołania obiekt klasy `ShopProduct`, a odwołując się do jego składowych i metod, konstruuje ciąg podsumowujący zawartość obiektu. Nazwa parametru metody, `$shopProduct`, sygnalizuje co prawda spodziewany typ obiektu, ale w żaden sposób go nie wymusza. Oznacza to, że argumentem wywołania metody mógłby być dowolny typ prosty albo obiektowy, a jego faktyczny typ mógłby się objawić dopiero przy próbie użycia go w operacji zakładającej obecność obiektu klasy `ShopProduct`. Tyle że jeszcze przed użyciem argumentu metoda może wykonać pewne operacje na bazie założenia, że ma do czynienia z obiektem odpowiedniej klasy.

■ **Uwaga** Metodę `write()` można by dodać bezpośrednio do klasy `ShopProduct`. Nie zrobimy tego jednak ze względu na podział odpowiedzialności. Klasa `ShopProduct` ma realizować zadania zarządzania danymi produktów; za wypisywanie danych o produktach odpowiedzialna jest klasa `ShopProductWriter`. Znaczenie i przydatność wyraźnego podziału odpowiedzialności stanie się bardziej oczywiste po lekturze dalszej części rozdziału.

Problem niemożności wymuszenia typu w wywołaniu metody wyeliminowano w PHP5 wraz z mechanizmem deklarowania oczekiwanego typu klasy (wtedy nosił on nazwę *type hints*). Aby dodać deklarację typu do argumentu metody, należy po prostu poprzedzić argument, którego typ chcemy narzucić, nazwą odpowiedniej klasy. Metodę `write()` można by więc przepisać tak:

```
// listing 03.30
public function write(ShopProduct $shopProduct)
{
    // ...
}
```

Teraz metoda `write()` nie będzie akceptowała w roli argumentów wywołania obiektów klas innych niż `ShopProduct`.

Kup książkę Poleć książkę

Tak wygląda pusta klasa przykładowa:

```
// listing 03.31
class Wrong
{
}
```

A to kod, który podejmuje próbę wywołania `write()` przy użyciu obiektu `Wrong`:

```
// listing 03.32
$writer = new ShopProductWriter();
$writer->write(new Wrong());
```

Z racji obecności w ciele metody `write()` deklaracji typu przekazanie w wywołaniu obiektu nieodpowiedniej (`Wrong`) klasy sprowokuje krytyczny błąd programu:

```
TypeError: popp\r03\zestaw08\ShopProductWriter::write(): Argument #1 ($shopProduct) must
be of type popp\r03\zestaw 04\ShopProduct, popp\r03\zestaw 08\Wrong given, called in
/var/popp/src/r03/zestaw08/Runner.php on...
```

■ **Uwaga** Warto zauważyć, że w przykładowym opisie błędu TypeError pojawiają się dodatkowe informacje o użytych klasach. Na przykład klasa `Wrong` została opisana jako `popp\r03\zestaw08\Wrong`. Są to przykłady *przestrzeni nazw*, z którymi szerzej zapoznasz się w rozdziale 4.

Możemy teraz darować sobie testowanie typu argumentu przed przystąpieniem do jego przetwarzania. Otrzymujemy też bardziej przejrzystą dla użytkownika sygnaturę metody — użytkownik może na jej podstawie od razu wnioskować co do oczekiwanego w wywołaniu typu, bez konieczności uciekania się do dokumentacji. A ponieważ deklaracja typu jest rygorystycznie przestrzegana, unikamy nie zawsze łatwych do wykrycia błędów charakterystycznych dla błędów typowania.

Choć tak zautomatyzowana kontrola poprawności typów skutecznie eliminuje obszerną kategorię błędów, trzeba zdawać sobie sprawę, że deklaracje typów są kontrolowane w czasie wykonania programu. Oznacza to, że błąd naruszenia deklaracji zostanie wykryty i zgłoszony dopiero w momencie, w którym nastąpi wywołanie metody z obiektem nieodpowiedniej klasy. Jeśli przypadkiem niefortunne wywołanie write() będzie osadzone w klauzuli warunkowej uruchamianej jedynie w Boże Narodzenie, możesz spodziewać się pracowitych świąt — wcześniej błąd pozostanie najprawdopodobniej ukryty.

Deklaracje typów — typy elementarne

Do czasu ukazania się PHP7 ograniczenia typowania sprowadzały się do obiektów i dwóch innych typów (wywoływalnego i tablicy). W PHP7 wreszcie pojawiły się deklaracje typów skalarnych. Umożliwiają one wymuszenie typu logicznego (`bool`), łańcuchowego (`string`), całkowitoliczbowego (`integer`) i zmiennoprzecinkowego (`float`) na liście argumentów.

Uzbrojeni w możliwość deklarowania typów skalarnych możemy wprowadzić pewne ograniczenia w klasie `ShopProduct`.

```
// listing 03.33
class ShopProduct
{
```


Kup książkę Poleć książkę

```
    public $title;
    public $producerMainName;
    public $producerFirstName;
    public $price = 0;

    public function __construct(
        string $title,
        string $firstName,
        string $mainName,
        float $price
    ) {
        $this->title = $title;
        $this->producerFirstName = $firstName;
        $this->producerMainName = $mainName;
        $this->price = $price;
    }
    // ...
}
```

Przy takiej formie konstruktora możemy mieć pewność, że argumenty `$title`, `$firstName` i `$mainName` zawsze będą zawierały łańcuchy znaków, a argument `$price` będzie typu `float`. Można się o tym przekonać, podejmując próbę utworzenia obiektu `ShopProduct` na bazie błędnych danych:

```
// listing 03.34
// to się nie uda
$product = new ShopProduct("tytuł", "imię", "nazwisko", []);
```

Przy próbie utworzenia egzemplarza obiektu `ShopProduct` do konstruktora zostały przekazane trzy łańcuchy znaków. Polegliśmy jednak na ostatnim argumencie, przekazując zamiast liczby typu `float` pustą tablicę. Dzięki deklaracjom typów PHP nie pozwoli nam na takie postępowanie:

```
TypeError: popp\r03\zestaw09\ShopProduct:: construct(): Argument #4 ($price) must be of
type float, array given, called in...
```

Domyślnie PHP będzie jawnie rzutował argumenty na wymagany typ tam, gdzie to możliwe. Jest to przykład pewnego konfliktu między bezpieczeństwem a elastycznością (o czym była mowa wcześniej). Nowa wersja klasy `ShopProduct` „po cichu" zamieni łańcuch znaków na liczbę typu `float`. Na przykład taka próba konkretyzacji obiektu zakończy się powodzeniem:

```
// listing 03.35
$product = new ShopProduct("tytuł", "imię", "nazwisko", "4.22");
```

Za kulisami łańcuch znaków `"4.22"` został zamieniony na wartość zmiennoprzecinkową `4.22`. Akurat ta zmiana jest użyteczna. Wróćmy jednak do problemu, który napotkaliśmy w związku z klasą `AddressManager`. Łańcuch `"false"` został po cichu zamieniony na boolowską wartość `true`. Domyślnie takie coś nadal będzie miało miejsce, jeśli zastosujemy deklarację typu `bool` w metodzie `AddressManager::outputAddresses()` w następujący sposób:

```
// listing 03.36
    public function outputAddresses(bool $resolve)
    {
        // ...
    }
```


Kup książkę Poleć książkę

Weźmy teraz wywołanie, w którym przekażemy łańcuch znaków, na przykład tak:

```
// listing 03.37
$manager->outputAddresses("false");
```

Ze względu na automatyczne rzutowanie kod ten jest funkcjonalnie identyczny z tym, który przekazywał wartość boolowską `true`.

Deklaracje typów skalarnych można potraktować bardziej rygorystycznie, ale tylko w obrębie danego pliku. W poniższym przykładzie metoda `outputAddresses()` została wywołana ponownie z argumentem w postaci łańcucha znaków, ale po włączeniu rygorystycznej kontroli typów:

```
// listing 03.38
declare(strict_types=1);
$manager->outputAddresses("false");
```

Ze względu na rygorystyczną kontrolę typów spowoduje to wygenerowanie błędu `TypeError`:

```
TypeError: popp\r03\zestaw09\AddressManager::outputAddresses(): Argument #1 ($resolve)
must be of type bool, string given, called in...
```

■ **Uwaga** Deklaracja `strict_types` dotyczy pliku, z którego nastąpiło wywołanie, a nie pliku, w którym została zaimplementowana dana metoda albo funkcja. Obowiązek wymuszenia zgodności typów leży więc po stronie kodu klienckiego.

Można potraktować jakiś argument jako opcjonalny, ale zarazem narzucić mu konkretny typ (jeśli argument ten zostanie przekazany). Da się to zrobić dzięki podaniu wartości domyślnej argumentu:

```
// listing 03.39
class ConfReader
{
    public function getValues(array $default = null)
    {
        $values = [];
        // zrób coś, aby pobrać wartości
        // scal dostarczone wartości domyślne (zawsze będą one miały postać tablicy)
        $values = array_merge($default, $values);
        return $values;
    }
}
```

Typ mixed

Deklaracja typu `mixed` wprowadzona w PHP 8.0 może być uznawana za przykład tak zwanego lukru składniowego — czyli czegoś, co samo w sobie niewiele zmienia. Pod względem *funkcjonalności* nie ma żadnej różnicy między tym kodem:

```
// listing 03.40
class Storage
{
    public function add(string $key, $value)
```


Kup książkę Poleć książkę

```
    {
        // zrób coś z $key i $value
    }
}
```

a tym:

```
// listing 03.41
class Storage
{
    public function add(string $key, mixed $value)
    {
        // zrób coś z $key i $value
    }
}
```

W drugiej wersji zadeklarowałem, że argument `$value` funkcji `add()` przyjmuje wartości typu `mixed`, co oznacza dowolną wartość następujących typów: `array`, `bool`, `callable`, `int`, `float`, `null`, `object`, `resource` lub `string`. Zadeklarowanie argumentu `$value` jako typu `mixed` jest więc analogiczne jak pozostawienie tego argumentu bez deklaracji typu. Po co więc zawracać sobie głowę deklaracją `mixed`? Zasadniczo po to, by pokazać, że argument ten przyjmuje dowolną wartość *intencjonalnie*. „Goły" argument też przyjmie dowolną wartość, ale brak deklaracji może oznaczać przeoczenie albo lenistwo programisty. Deklaracja `mixed` rozwiewa takie wątpliwości i jest przydatna właśnie z tego powodu.

W ramach podsumowania w tabeli 3.3 zostały wymienione deklaracje typów obsługiwane przez PHP.

Tabela 3.3. *Deklaracje typów*

Deklaracja typu	Od wersji	Opis
`array`	5.1	Tablica. Dopuszcza wartość domyślną `null` albo tablicę.
`int`	7.0	Liczba całkowita. Dopuszcza wartość domyślną `null` albo liczbę całkowitą.
`float`	7.0	Liczba zmiennoprzecinkowa (z częścią ułamkową). Zaakceptowana zostanie jednak także liczba całkowita — nawet po włączeniu rygorystycznej kontroli typów. Dopuszcza wartość domyślną `null`, liczbę zmiennoprzecinkową albo liczbę całkowitą.
`callable`	5.4	Kod wywoływalny (taki jak funkcja anonimowa). Dopuszcza wartość domyślną `null`.
`bool`	7.0	Wartość boolowska. Dopuszcza wartość domyślną `null` albo wartość boolowską.
`string`	5.0	Dane znakowe. Dopuszcza wartość domyślną `null` albo łańcuch znaków.
`self`	5.0	Odwołanie do klasy macierzystej.
[*typ klasy*]	5.0	Typ klasy albo interfejsu. Dopuszcza wartość domyślną `null`.
`iterable`	7.1	Może być przetwarzany przy użyciu pętli `foreach` (nie musi to być tablica — może implementować interfejs `Traversable`).
`object`	7.2	Obiekt.
`mixed`	8.0	Jawne określenie, że wartość może być dowolnego typu.

Kup książkę Poleć książkę

Unia

Między dopuszczającą wszystkie typy deklaracją `mixed` a względną restrykcyjnością zwykłych deklaracji typów zieje spora przepaść. Co zrobić w sytuacji, gdy trzeba ograniczyć zakres dopuszczalnych typów argumentu do dwóch, trzech lub większej liczby konkretnych typów? W wersjach PHP starszych niż 8.0 jedyny sposób na uzyskanie takiego efektu polegał na skontrolowaniu typu w ciele metody. Wróćmy do klasy `Storage` i uzupełnijmy ją o nowy wymóg. Przypuśćmy, że argument `$value` metody `add()` ma przyjmować tylko łańcuch znaków albo wartość boolowską. Tak przedstawia się implementacja tej klasy z uwzględnieniem kontroli typów wymienionych w ciele klasy:

```
// listing 03.42
class Storage
{
    public function add(string $key, $value)
    {
        if (! is_bool($value) && ! is_string($value)) {
            error_log("argument value musi być typu łańcuchowego lub boolowskiego,
            ↪a jest: " . gettype($value));
            return false;
        }
        // zrób coś z $key i $value
    }
}
```

■ **Uwaga** W praktyce lepiej byłoby wyrzucić wyjątek, zamiast zwracać `false`. O wyjątkach możesz przeczytać w rozdziale 4.

Choć ta „ręczna” kontrola zda egzamin, jest dość uciążliwa i trudna do zinterpretowania. W sukurs przychodzi nam tutaj nowa funkcjonalność w PHP8, a mianowicie unia, która umożliwia łączenie dwóch lub większej liczby typów (oddzielonych symbolem pionowej kreski), czyli tworzenie złożonej deklaracji typu.

Oto zmodyfikowana implementacja klasy `Storage`:

```
// listing 03.43
class Storage
{
    public function add(string $key, string|bool $value)
    {
        // zrób coś z $key i $value
    }
}
```

Jeśli spróbowalibyśmy przekazać argument `$value` w postaci innej niż łańcuch znaków lub wartość boolowska, otrzymalibyśmy znany już błąd `TypeError`.

Aby funkcja `add()` dopuszczała nieco więcej możliwości, możemy użyć unii do zadeklarowania, że dopuszczalna jest też wartość `null`.

```
// listing 03.44
class Storage
{
```


Kup książkę Poleć książkę

```
    public function add(string $key, string|bool|null $value)
    {
        // zrób coś z $key i $value
    }
}
```

Unię można też wykorzystać do zadeklarowania konkretnego typu obiektu. W poniższym przykładzie dopuszczalne są wartości typu `ShopProduct` i `null`:

```
// listing 03.45
public function setShopProduct(ShopProduct|null $product)
{
    // zrób coś z $product
}
```

Ponieważ wiele metod przyjmuje lub zwraca `false` jako wartość alternatywną, PHP8 obsługuje w kontekście unii pseudotyp `false`. W poniższym przykładzie akceptowane są więc obiekty typu `ShopProduct` lub `false`:

```
// listing 03.46
public function setShopProduct2(ShopProduct|false $product)
{
    // zrób coś z $product
}
```

To rozwiązanie jest wygodniejsze niż unia `ShopProduct|bool`, bo w opisywanym przypadku nie należy akceptować wartości `true`.

■ **Uwaga** Unia została zaimplementowana w PHP8.

Typy przyjmujące wartość null

Wprawdzie unia umożliwia zastosowanie `null` jako jednej z opcji, niemniej istnieje konstrukcja będąca odpowiednikiem takiego rozwiązania. Konstrukcja ta, przyjmująca wartość `null`, składa się z deklaracji typu poprzedzonej znakiem zapytania. Poniższa wersja klasy `Storage` przyjmuje więc wartości łańcuchowe oraz `null`:

```
// listing 03.47
class Storage
{
    public function add(string $key, ?string $value)
    {
        // zrób coś z $key i $value
    }
}
```

Kiedy pisaliśmy o deklaracji typu klasy, traktowaliśmy typy i klasy jako pojęcia równoznaczne. Tymczasem pomiędzy typami a klasami istnieje zasadnicza różnica. Otóż definiując klasę, definiuje się równocześnie typ, ale typ jako taki może opisywać całą rodzinę klas. Mechanizm grupowania wielu klas w obrębie jednego typu nosi nazwę dziedziczenie. Będzie on tematem następnego podrozdziału.

Kup książkę Poleć książkę

Deklaracje typów zwracanych

Na tej samej zasadzie, na jakiej deklarujemy typ argumentu, możemy zadeklarować typ wartości zwracanej, aby ograniczyć pulę typów, jakie może zwracać dana metoda. Deklarację typu zwracanego należy umieścić bezpośrednio po zamykającym nawiasie metody lub funkcji i składa się ona z dwukropka oraz nazwy typu. W odniesieniu do typów zwracanych obsługiwany jest ten sam zbiór typów co w przypadku argumentów. W poniższym przykładzie zadeklarowany został konkretny typ wartości zwracanej przez funkcję `getPlayLength()`:

```
// listing 03.48
public function getPlayLength(): int
{
    return $this->playLength;
}
```

Jeśli wywołanie tej funkcji nie spowoduje zwrócenia wartości całkowitej, PHP wygeneruje błąd:

```
TypeError: popp\r03\zestaw15\CdProduct::getPlayLength(): Return value must be of type
int, none returned
```

Ponieważ rozwiązanie to wymusza typ zwracanej wartości, w kodzie odwołującym się do tej metody możemy uznać, że zwrócona wartość ma gwarantowany typ całkowitoliczbowy.

Deklaracje typów zwracanych obsługują typy przyjmujące wartość `null` oraz unie. Zastosujmy konstrukcję z unią:

```
// listing 03.49
public function getPrice(): int|float
{
    return ($this->price - $this->discount);
}
```

W PHP8 jest jeden typ obsługiwany w deklaracjach wartości zwracanych, którego nie da się zastosować w deklaracji typów argumentów. Chodzi o pseudotyp `void`, który oznacza, że metoda nigdy nie zwraca wartości. Na przykład, ponieważ metoda `setDiscount()` służy do ustawiania wartości, a nie jej zwracania, możemy określić zwracany przez nią typ jako `void`:

```
// listing 03.50
public function setDiscount(int|float $num): void
{
    $this->discount = $num;
}
```

Dziedziczenie

Dziedziczenie to mechanizm wyprowadzania jednej bądź wielu klas pochodnych z pewnej wspólnej klasy bazowej.

Klasa dziedzicząca po innej klasie staje się jej podklasą. Zależność ta często opisywana jest w oparciu o relację rodzic – dziecko. Owo „dziecko” (klasa potomna czy też pochodna) jest wyprowadzone z klasy „rodzica” (klasy nadrzędnej albo bazowej) i dziedziczy jej składowe

Kup książkę Poleć książkę

i metody. Klasa pochodna zazwyczaj uzupełnia elementy odziedziczone własnymi składowymi i metodami — mówi się wtedy o „rozszerzaniu" klasy bazowej[1].

Zanim zagłębimy się w składnię dziedziczenia, powinniśmy rozpoznać problemy, w których rozwiązywaniu dziedziczenie okazuje się pomocne.

Problemy, które można rozwiązać poprzez dziedziczenie

Wróćmy do naszej klasy `ShopProduct`. Na razie jest ona dość ogólna, ponieważ nie ogranicza asortymentu produktów (mimo że dotychczas jej obiekty reprezentowały asortyment księgarski).

```
// listing 03.51
$product1 = new ShopProduct("Moja Antonia", "Willa", "Cather", 5.99);
$product2 = new ShopProduct("Exile on Coldharbour Lane", "The", "Alabama 3", 10.99);

print "Autor: " . $product1->getProducer() . "\n";
print "Wykonawca : " . $product2->getProducer() . "\n";
```

Program wypisze na wyjściu:

```
Autor     : Willa Cather
Wykonawca : The Alabama 3
```

Rozdzielenie nazwy autora (lub wykonawcy) na dwie części sprawdza się dla książek i nawet dla albumów CD. Możemy dzięki niemu wyszukiwać i porządkować asortyment wg „Alabama 3" i „Cather", pozbywając się mniej znaczących „The" i „Willa". Wygoda to zazwyczaj znakomita strategia projektowa, nie musimy więc na razie przejmować się dostosowaniem projektu klasy `ShopProduct` do artykułów innych rodzajów.

Gdybyśmy jednak nasz przykład uzupełnili o pewne dodatkowe wymagania, rzecz szybko by się skomplikowała. Załóżmy na przykład, że obiekty klasy `ShopProducer` powinny jednak przechowywać dodatkowo informacje charakterystyczne dla ich asortymentu — inne w przypadku książek (np. liczba stron), inne w przypadku albumów CD (np. czas nagrania). Różnic może być znacznie więcej, ale i te wystarczą do ilustracji problemu.

W jaki sposób powinniśmy rozszerzyć klasę, aby dało się odzwierciedlić w niej nowe wymagania? Niemal natychmiast na myśl przychodzą dwie możliwości. Pierwsza polega na zebraniu w klasie `ShopProduct` wszelkich możliwych składowych. Druga zakłada podział klasy na dwie osobne.

Spróbujmy pierwszego sposobu — połączenia w jednej klasie składowych charakterystycznych dla płyt i książek:

```
// listing 03.52
class ShopProduct
{
    public $numPages;
    public $playLength;
    public $title;
    public $producerMainName;
    public $producerFirstName;
    public $price;
    public function __construct(
```

[1] Choć właściwsze byłoby mówienie o „specjalizacji" — *przyp. tłum.*

Kup książkę Poleć książkę

```
        string $title,
        string $firstName,
        string $mainName,
        float $price,
        int $numPages = 0,
        int $playLength = 0
    ) {
        $this->title             = $title;
        $this->producerFirstName = $firstName;
        $this->producerMainName  = $mainName;
        $this->price             = $price;
        $this->numPages          = $numPages;
        $this->playLength        = $playLength;
    }
    public function getNumberOfPages(): int
    {
        return $this->numPages;
    }
    public function getPlayLength(): int
    {
        return $this->playLength;
    }
    public function getProducer(): string
    {
        return $this->producerFirstName . " "
            . $this->producerMainName;
    }
}
```

W definicji klasy pojawiły się metody dające dostęp do składowych `$numPages` i `$playLength`; kod ten ilustruje też pewną nadmiarowość. Otóż obiekt konkretyzowany z takiej klasy będzie zawierał nadmiarową metodę, a w przypadku obiektów dla płyt CD konstruktor będzie przyjmował niepotrzebny argument: obiekt reprezentujący płytę CD będzie utrzymywać informacje i funkcje właściwe dla obiektów książek (tu: liczbę stron) i odwrotnie — obiekt reprezentujący w istocie książkę będzie niepotrzebnie przechowywał długość nagrania. Na razie zapewne możemy się z taką nadmiarowością pogodzić. Ale co, jeśli asortyment zostanie rozszerzony na kolejne kategorie produktów, a wraz z nimi pojawią się w klasie kolejne składowe i metody? Klasa nadmiernie się rozrośnie i stanie się po prostu niewygodna w użyciu.

Jak widać, scalanie w jednej klasie danych i funkcji różnych klas prowadzi do rozszerzenia obiektów o nadmiarowe i zbędne składowe i metody.

Problem nie kończy się jednak na nadmiarowości danych. Cierpi również funkcjonalność klasy. Weźmy choćby metodę zestawiającą informacje o produkcie. Niech dział sprzedaży zażyczy sobie możliwości generowania podsumowania informacji o artykule na potrzeby wystawianych w dziale faktur. W opisie albumu CD ma znaleźć się długość nagrania, a w opisie książki — liczba stron. Trzeba więc będzie przewidzieć różne implementacje zestawień dla każdego rodzaju asortymentu. Można by spróbować wygospodarować w klasie znacznik informujący o formacie obiektu, jak w tym przykładzie:

```
// listing 03.53
public function getSummaryLine(): string
{
    $base = "{$this->title} ( {$this->producerMainName}, ";
```


Kup książkę Poleć książkę

```
        $base .= "{$this->producerFirstName} )";
        if ( $this->type == 'książka' ) {
            $base .= ": liczba stron - {$this->numPages}";
        } else if ($this->type == 'cd' ) {
            $base .= ": czas nagrania - {$this->playLength}";
        }
        return $base;
    }
```

Alternatywnie, aby poprawnie ustawić składową `$type`, moglibyśmy sprawdzić wartość argumentu wywołania konstruktora dla parametru `$numPages`. Słowem, dalej niepotrzebnie „rozdymamy" i komplikujemy klasę `ShopProduct`, a w miarę dokładania różnych formatów obiektów różnice funkcjonalne pomiędzy nimi będą coraz trudniejsze do ujęcia w spójnej implementacji. Może więc lepiej byłoby spróbować innego sposobu?

Ponieważ klasa `ShopProduct` zaczyna przypominać siłowe sklejenie dwóch klas, możemy spróbować podzielić ją na dwoje. Moglibyśmy podejść do zadania tak:

// listing 03.54

```
class CdProduct
{
    public $playLength;
    public $title;
    public $producerMainName;
    public $producerFirstName;
    public $price;
    public function __construct(
        string $title,
        string $firstName,
        string $mainName,
        float  $price,
        int    $playLength
    ) {
        $this->title             = $title;
        $this->producerFirstName = $firstName;
        $this->producerMainName  = $mainName;
        $this->price             = $price;
        $this->playLength        = $playLength;
    }
    public function getPlayLength(): int
    {
        return $this->playLength;
    }
    public function getSummaryLine(): string
    {
        $base  = "{$this->title} ( {$this->producerMainName}, ";
        $base .= "{$this->producerFirstName} )";
        $base .= ": czas nagrania - {$this->playLength}";
        return $base;
    }
    public function getProducer(): string
    {
        return $this->producerFirstName . " "
            . $this->producerMainName;
    }
```


Kup książkę Poleć książkę

```
}
```

```
// listing 03.55
class BookProduct
{
    public $numPages;
    public $title;
    public $producerMainName;
    public $producerFirstName;
    public $price;
    public function __construct(
        string $title,
        string $firstName,
        string $mainName,
        float  $price,
        int    $numPages
    ) {
        $this->title             = $title;
        $this->producerFirstName = $firstName;
        $this->producerMainName  = $mainName;
        $this->price             = $price;
        $this->numPages          = $numPages;
    }
    public function getNumberOfPages(): int
    {
        return $this->numPages;
    }
    public function getSummaryLine(): string
    {
        $base  = "{$this->title} ( {$this->producerMainName}, ";
        $base .= "{$this->producerFirstName} )";
        $base .= ": liczba stron - {$this->numPages}";
        return $base;
    }
    public function getProducer(): string
    {
        return $this->producerFirstName . " "
            . $this->producerMainName;
    }
}
```

Rozwiązaliśmy problem rosnącej złożoności klasy, ale zapłaciliśmy za to pewną cenę. Teraz możemy tworzyć osobne wersje metody `getSummaryLine()` bez potrzeby kontrolowania w jej ciele znacznika właściwego formatu obiektu. Klasa nie utrzymuje też zbędnych składowych i metod.

Wspomnianą ceną jest powielenie kodu. Metoda `getProducer()` w obu klasach ma identyczny kod. Każdy z konstruktorów ustawia w identyczny sposób pewien podzbiór składowych obiektu. To istotna wada kodu i warto się jej pozbyć.

Skoro metoda `getProducer()` zachowuje się identycznie dla każdej z klas, to jakakolwiek zmiana tego zachowania będzie musiała być zaimplementowana z osobna we wszystkich tych klasach. Prędzej czy później podczas tej synchronizacji będziemy przeklinać podjętą decyzję.

Jeśli zaś jesteś przekonany, że poradzisz sobie z duplikacją kodu, to nie możesz zapomnieć, że teraz zamiast jednego typu mamy dwa różne (mimo podobieństw) typy.

Kup książkę Poleć książkę

Wróćmy do klasy `ShopProductWriter`. Jej metoda `write()` została przystosowana do pracy na obiektach pojedynczego typu — obiektach klasy `ShopProduct`. W jaki sposób zmusić ją do obsługi obiektów dwóch różnych klas? Możemy oczywiście usunąć z definicji metody deklarację typu argumentu, ale wtedy będziemy musieli w pełni zaufać wywołującemu — program będzie poprawny jedynie wtedy, kiedy do metody będą przekazywane obiekty właściwych typów. Możemy dokonywać kontroli typów w ciele metody:

```
// listing 03.56
class ShopProductWriter
{
    public function write($shopProduct): void
    {
        if (
            ! ($shopProduct instanceof CdProduct)  &&
            ! ($shopProduct instanceof BookProduct)
        ) {
            die("Przekazano niewłaściwy typ danych");
        }
        $str  = "{$shopProduct->title}: "
            . $shopProduct->getProducer()
            . " ({$shopProduct->price})\n";
        print $str;
    }
}
```

W przykładzie wykorzystaliśmy operator `instanceof`. Wywołanie tego operatora daje wartość `true`, jeśli jego lewy operand jest egzemplarzem klasy występującej w roli prawego operandu.

Raz jeszcze zostaliśmy więc zmuszeni do wprowadzenia dodatkowego poziomu złożoności w kodzie. Nie tylko musimy testować przynależność przekazanego w wywołaniu `write()` obiektu do jednego z dwóch typów, ale i ufać, że żaden z tych typów nie zaniecha obsługi wykorzystywanych przez nas składowych i metod. Rzecz wyglądała znacznie lepiej, kiedy żądaliśmy przekazania w wywołaniu konkretnego typu, większą mieliśmy też pewność, że typ ten — klasa `ShopProduct` — posiada taki, a nie inny interfejs.

Odmiany książkowa i płytowa klasy `ShopProduct` nie współgrają ze sobą, ale mimo wszystko zdaje się, że mogą ze sobą przynajmniej koegzystować. Lepiej byłoby jednak, gdybyśmy i obiekty reprezentujące książki, i obiekty płyt muzycznych mogli traktować jak egzemplarze jednej klasy, ale wyposażone w nieco odmienną implementację stosowną do formatu wymaganego w prezentacji asortymentu towarów. Chcielibyśmy więc móc zdefiniować wspólny zestaw funkcji i cech, unikając duplikacji kodu, ale równocześnie umożliwić rozgałęzienie implementacji niektórych wywołań metod zależnie od formatu obiektu. Rozwiązaniem jest dziedziczenie.

Stosowanie dziedziczenia

Pierwszym etapem konstrukcji hierarchii dziedziczenia jest identyfikacja tych elementów klasy bazowej, które nie są na tyle uniwersalne, aby dały się identycznie obsługiwać we wszystkich egzemplarzach.

W naszej powstałej swego czasu klasie `ShopProducer` mieliśmy, na przykład, kolidujące ze sobą metody `getPlayLength()` i `getNumberOfPages()`. Pamiętamy też, że wspólna dla wszystkich obiektów metoda `getSummaryLine()` wymagała różnych implementacji dla różnych formatów obiektów.

Kup książkę Poleć książkę

Wykorzystajmy te różnice do wyodrębnienia dwóch klas pochodnych:

```
// listing 03.57
class ShopProduct
{
    public $numPages;
    public $playLength;
    public $title;
    public $producerMainName;
    public $producerFirstName;
    public $price;
    public function __construct(
        string $title,
        string $firstName,
        string $mainName,
        float  $price,
        int    $numPages = 0,
        int    $playLength = 0
    ) {
        $this->title             = $title;
        $this->producerFirstName = $firstName;
        $this->producerMainName  = $mainName;
        $this->price             = $price;
        $this->numPages          = $numPages;
        $this->playLength        = $playLength;
    }
    public function getProducer(): string
    {
        return $this->producerFirstName . " "
            . $this->producerMainName;
    }
    public function getSummaryLine(): string
    {
        $base  = "{$this->title} ( {$this->producerMainName}, ";
        $base .= "{$this->producerFirstName} )";
        return $base;
    }
}
```

```
// listing 03.58
class CdProduct extends ShopProduct
{
    public function getPlayLength(): int
    {
        return $this->playLength;
    }
    public function getSummaryLine(): string
    {
        $base  = "{$this->title} ( {$this->producerMainName},
        $base .= "{$this->producerFirstName} )";
        $base .= ": czas nagrania - {$this->playLength}";
        return $base;
    }
}
```


Kup książkę Poleć książkę

```
// listing 03.59
class BookProduct extends ShopProduct
{
    public function getNumberOfPages(): int
    {
        return $this->numPages;
    }
    public function getSummaryLine(): string
    {
        $base  = "{$this->title} ( {$this->producerMainName}, ";
        $base .= "{$this->producerFirstName} )";
        $base .= ": liczba stron - {$this->numPages}";
        return $base;
    }
}
```

Utworzenie klasy pochodnej wymaga opatrzenia deklaracji klasy słowem `extends`. W powyższym przykładzie utworzyliśmy w ten sposób dwie nowe klasy: `BookProduct` i `CdProduct`. Obie rozszerzają i uzupełniają klasę `ShopProduct`.

Ponieważ w klasach pochodnych zabrakło definicji konstruktorów, w momencie konkretyzacji obiektów tych klas wywoływany jest automatycznie konstruktor klasy bazowej. Klasy pochodne dziedziczą bowiem dostęp do wszystkich publicznych i chronionych metod klasy bazowej (z wyjątkiem składowych i metod prywatnych). Oznacza to, że możemy wywoływać metodę `getProducer()` na rzecz obiektu konkretyzowanego z klasy `CdProduct`, choć sama metoda `getProducer()` jest zdefiniowana nie w `CdProduct`, a w `ShopProduct`.

```
// listing 03.60
$product2 = new CdProduct(
    "Exile on Coldharbour Lane",
    "The",
    "Alabama 3",
    10.99,
    0,
    60.33
);
print "Wykonawca: {$product2->getProducer()}\n";
```

Jak widać, obie klasy pochodne dziedziczą zachowanie po rodzicu. Obiekt klasy `BookProduct` możemy więc traktować jak wcielenie obiektu klasy `ShopProduct`. I dlatego też możemy przekazywać obiekty klasy `BookProduct` bądź `CdProduct` w wywołaniu metody `write()` klasy `ShopProductWriter`.

Zauważmy, że w klasach `CdProduct` i `BookProduct` nastąpiło przesłonięcie metody `getSummaryLine()` jej implementacjami odpowiednimi dla tych klas. Sęk w tym, że klasy pochodne mogą nie tylko rozszerzać i uzupełniać, ale i modyfikować zachowanie klas nadrzędnych.

Implementacja tej metody w klasie bazowej wydaje się nadmiarowa, skoro i tak jest przepisywana w obu klasach pochodnych. Niemniej jednak ta bazowa implementacja udostępnia najbardziej podstawową realizację danej funkcji, dostępną do użycia w klasach pochodnych. Obecność metody w klasie bazowej daje też gwarancję, że wszelkie obiekty klasy `ShopProduct` (i klas pochodnych) w kodzie klienckim będą posiadać metodę `getSummaryLine()`. Później przekonamy się, że taką gwarancję można wymusić bez implementowania metody w klasie bazowej. Każda klasa pochodna `ShopProduct` dziedziczy komplet składowych klasy „rodzica". Dlatego zarówno klasa `CdProduct`, jak i `BookProduct` mogą w swoich implementacjach metody `getSummaryLine()` odwoływać się do składowej `$title`.

Kup książkę Poleć książkę

Dziedziczenie może z początku być koncepcją niejasną. Definiując klasę rozszerzającą inną klasę, gwarantujemy, że obiekt tejże nowej klasy będzie w pierwszym rzędzie określany cechami definiowanymi w klasie pochodnej, a dopiero w drugiej kolejności tymi z klasy bazowej. Można też zastosować inną analogię — gdybyśmy chcieli samodzielnie rozprowadzić wywołanie `$product2->getProducer()`, nie znaleźlibyśmy takiej metody w klasie `CdProduct`, więc wywołanie przenieślibyśmy do „domyślnej" implementacji tej metody, zdefiniowanej w `ShopProduct`. Ale już wywołanie `$product2->getSummaryLine()` możemy zrealizować za pomocą metody z klasy `CdProduct`.

To samo dotyczy odwołań do składowych. Występującego w metodzie `getSummaryLine()` klasy `BookProduct` odwołania do składowej `$title` nie można zrealizować w ramach klasy `BookProduct`; jest ona pobierana z klasy bazowej. Pozostawienie jej w klasie bazowej ma uzasadnienie, ponieważ inaczej trzeba by ją dublować we wszystkich pochodnych.

Rzut oka na konstruktor klasy bazowej ujawnia jednak, że wciąż w klasie bazowej obsługujemy dane, których obsługa powinna zostać przeniesiona do klas pochodnych. Otóż klasa `BookProduct` powinna przejąć obsługę argumentu i składowej `$numPages`, a składowa `$playLength` powinna zostać wyodrębniona do klasy `CdProduct`. W tym celu trzeba by w klasach pochodnych zdefiniować ich własne konstruktory.

Dziedziczenie a konstruktory

Definiując konstruktor klasy pochodnej, trzeba wziąć na siebie odpowiedzialność za przekazanie argumentów do wywołania konstruktora klasy bazowej. Jeśli to zaniedbamy, otrzymamy częściowo tylko skonstruowany obiekt.

Aby wywołać z wnętrza klasy pochodnej metodę klasy bazowej, musimy najpierw poznać sposób odwołania się do klasy jako takiej. W języku PHP służy do tego słowo kluczowe `parent`.

Aby odwołać się do metody w kontekście klasy, a nie obiektu, powinniśmy zamiast operatora `->` zastosować operator `::`.

```
parent::__construct()
```

■ **Uwaga** Operator zasięgu (`::`) zostanie omówiony w rozdziale 4.

Powyższy zapis oznacza więc: „wywołanie metody `__construct()` klasy bazowej". Spróbujmy zatem zmodyfikować nasz przykład tak, aby każda klasa odpowiadała jedynie za swoje własne składowe:

```
// listing 03.61
class ShopProduct
{
    public $title;
    public $producerMainName;
    public $producerFirstName;
    public $price;
    public function __construct(
        $title,
        $firstName,
        $mainName,
        $price
    ) {
        $this->title            = $title;
        $this->producerFirstName = $firstName
```


Kup książkę Poleć książkę

```
        $this->producerMainName  = $mainName;
        $this->price             = $price;
    }
    public function getProducer(): string
    {
        return $this->producerFirstName . " "
            . $this->producerMainName;
    }
    public function getSummaryLine(): string
    {
        $base  = "{$this->title} ( {$this->producerMainName}, ";
        $base .= "{$this->producerFirstName} )";
        return $base;
    }
}
```

// listing 03.62

```
class BookProduct extends ShopProduct
{
    public $numPages;
    public function __construct(
        string $title,
        string $firstName,
        string $mainName,
        float  $price,
        int    $numPages
    ) {
        parent::__construct(
            $title,
            $firstName,
            $mainName,
            $price
        );
        $this->numPages = $numPages;
    }
    public function getNumberOfPages(): int
    {
        return $this->numPages;
    }
    public function getSummaryLine(): string
    {
        $base  = "{$this->title} ( $this->producerMainName, ";
        $base .= "$this->producerFirstName )";
        $base .= ": liczba stron - {$this->numPages}";
        return $base;
    }
}
```

// listing 03.63

```
class CdProduct extends ShopProduct
{
    public $playLength;
    public function __construct(
        string $title,
        string $firstName,
```


Kup książkę Poleć książkę

```
        string $mainName,
        float  $price,
        int    $playLength
    ) {
        parent::__construct(
            $title,
            $firstName,
            $mainName,
            $price
        );
        $this->playLength = $playLength;
    }
    public function getPlayLength(): int
    {
        return $this->playLength;
    }
    public function getSummaryLine(): string
    {
        $base  = "{$this->title} ( {$this->producerMainName}, ";
        $base .= "{$this->producerFirstName} )";
        $base .= ": czas nagrania - {$this->playLength}";
        return $base;
    }
}
```

Każda klasa pochodna wywołuje w swoim konstruktorze konstruktor klasy bazowej, a dopiero potem przystępuje do ustawiania własnych składowych. Klasa bazowa troszczy się wyłącznie o swoje dane. Klasy pochodne są zaś w ogólności specjalizacjami klas bazowych. Należy więc unikać ujmowania w klasach bazowych (jako ogólniejszych) specjalistycznej wiedzy o klasach pochodnych.

■ **Uwaga** W wersjach poprzedzających PHP5 konstruktory miały nazwy zgodne z nazwami klas. Teraz nazwy metod konstrukcji zostały ujednolicone — konstruktor każdej klasy nazywa się `__construct()`. Gdyby zechcieć skorzystać z przestarzałej składni, wywołanie konstruktora klasy bazowej wiązałoby kod klasy pochodnej z tą konkretną klasą: `parent::ShopProduct();`. W PHP7 poprzednią składnię konstruktorów uznano za przestarzałą i ostatecznie usunięto w PHP8.

Wywołania metod przesłoniętych

Słowo kluczowe `parent` można stosować w odwołaniach do wszelkich metod klasy bazowej, które zostały przesłonięte w klasie pochodnej. Niekiedy bowiem zamiast całkiem przesłaniać metodę klasy bazowej, chcemy jedynie uzupełnić jej działanie. Możemy się wtedy we własnej implementacji wesprzeć wersją metody z klasy bazowej. Mimo ulepszeń wprowadzonych do naszej hierarchii klas wciąż mamy do czynienia z pewną duplikacją kodu — dochodzi do niej w ramach metody `getSummaryLine()`. Tymczasem zamiast powtarzać kod w klasach pochodnych, moglibyśmy odwołać się do kodu klasy `ShopProduct` i tylko uzupełnić go stosownie do potrzeb klasy pochodnej:

```
// listing 03.64
// ShopProduct
    public function getSummaryLine(): string
    {
        $base = "{$this->title} ( {$this->producerMainName}, ";
        $base .= "{$this->producerFirstName} )";
```


Kup książkę Poleć książkę

```
        return $base;
    }
```

```
// listing 03.65
// BookProduct
    public function getSummaryLine(): string
    {
        $base = parent::getSummaryLine();
        $base .= ": liczba stron - {$this->numPages}";
        return $base;
    }
```

Podstawowe zadania metody `getSummaryLine()` zdefiniowaliśmy w klasie `ShopProduct`.

Teraz zamiast powtarzać jej kod w klasach `CdProduct` i `BookProduct`, możemy po prostu wywołać wersję z klasy bazowej i uzupełnić otrzymany z takiego wywołania ciąg danymi charakterystycznymi dla klasy pochodnej.

Znając już podstawowe zasady dziedziczenia, możemy wrócić do zagadnienia widoczności metod i składowych.

Zarządzanie dostępem do klasy — słowa public, private i protected

Jak dotąd wszystkie składowe i metody klas deklarowaliśmy jako publiczne (ze słowem kluczowym `public`). Dostęp publiczny był przyjmowany jako domyślny dla metod oraz składowych zadeklarowanych z obowiązującym w starych wersjach PHP słowem kluczowym `var`.

■ **Uwaga** Słowo `var` zostało uznane za przestarzałe już w PHP5 i prawdopodobnie będzie usunięte z przyszłych wydań języka.

Jak już wiemy, elementy klas mogą być deklarowane jako publiczne (`public`), ale również jako chronione (`protected`) i prywatne (`private`):

- Metody i składowe publiczne są dostępne niezależnie od kontekstu.
- Metody i składowe prywatne są dostępne jedynie z wnętrza zawierającej je klasy. Dostęp do nich jest odmawiany nawet klasom pochodnym.
- Metody i składowe chronione są dostępne z klasy, w której są deklarowane, oraz z jej klas pochodnych. Dostępu nie uzyskuje jednak kod zewnętrzny wobec danej klasy.

Czy różnicowanie widoczności może być w ogóle przydatne? Owszem, ponieważ odpowiednio stosowane słowa regulujące widoczność pozwalają na eksponowanie z klasy jedynie tych jej elementów, które są potrzebne użytkownikom obiektów klasy i jako takie stanowią jej interfejs.

Uniemożliwiając kodowi klienckiemu odwoływanie się do niektórych składowych, możemy zapobiegać błędom. Wyobraźmy sobie, że obiekty `ShopProduct` mają przechowywać informacje o rabatach. W takim przypadku można byłoby utworzyć składową `$discount` oraz metodę `setDiscount()`:

```
// listing 03.66
// Klasa ShopProduct
    public $discount = 0;
// ...
    public function setDiscount(int $num): void
```


Kup książkę Poleć książkę

```
    {
        $this->discount = $num;
    }
```

Uzbrojeni w mechanizm przyznawania rabatów możemy zdefiniować metodę `getPrice()`, która wyceni artykuł z uwzględnieniem owego rabatu:

```
// listing 03.67
public function getPrice(): int|float
{
    return ($this->price - $this->discount);
}
```

Pojawia się jednak problem. Otóż użytkownicy zewnętrzni powinni widzieć jedynie ostateczne ceny (uwzględniające rabaty), tymczasem użytkownik mający dostęp do obiektu może — zamiast wywoływać metodę `getPrice()` — pójść na skróty i odwołać się bezpośrednio do składowej przechowującej cenę:

```
print "Cena artykułu wynosi {$product1->price}\n";
```

Spowoduje to wyprowadzenie mylącej (bo nieuwzględniającej rabatów) ceny. Takim przypadkom możemy zapobiec, deklarując `$price` jako składową prywatną i uniemożliwiając dostęp do niej z zewnątrz. Zmusi to użytkowników do korzystania z metody `getPrice()`. Próba odwołania się do prywatnej składowej spoza klasy `ShopProduct` będzie bowiem nieskuteczna. Dla świata zewnętrznego składowa ta przestanie po prostu istnieć.

Ukrywanie składowych jako prywatnych może jednak okazać się nadgorliwością. Do składowych prywatnych nie mają dostępu nawet klasy pochodne. Wyobraźmy sobie, że zgodnie z założeniami biznesowymi książki zostały wyłączone z wszelkich promocji. Pomysł taki moglibyśmy zrealizować, przesłaniając w klasie `BookProduct` metodę `getPrice()` z pominięciem rabatu. Metoda ta musi jednak mieć dostęp do składowej `$price`:

```
// listing 03.68
// BookProduct
    public function getPrice(): int|float
    {
        return $this->price;
    }
```

Jeśli składowa `$price` zostałaby zadeklarowana jako prywatna w klasie `ShopProduct`, to niestety powyższy kod byłby niepoprawny, gdyż dostęp do składowych prywatnych jest zablokowany nawet dla klas pochodnych. Rozwiązaniem byłoby zadeklarowanie `$price` jako składowej chronionej i jako takiej dostępnej z poziomu klas pochodnych. Trzeba przy tym pamiętać, że tak oznaczona składowa nie będzie w ogóle dostępna dla kodu spoza hierarchii dziedziczenia, w tym dla innych klas, które nie uczestniczą w tej hierarchii. Dostępność składowych i metod chronionych jest ograniczona do klasy, w której je zadeklarowano, i jej klas pochodnych.

Warto przyjąć regułę faworyzowania prywatności składowych i metod. To, co nie jest zabronione, jest dozwolone, lepiej więc domyślnie zaostrzać kryteria dostępu do składowych i rozluźniać je w miarę potrzeb. Wiele (zwykle większość) metod konstruowanych przez nas klas będzie metodami publicznymi, ale jeśli potrzeba ich udostępniania jest wątpliwa, lepiej z tego zrezygnować. Metoda udostępniająca lokalne funkcje danej klasy pozostałym metodom tej klasy nie powinna być widoczna dla użytkowników zewnętrznych — niech więc będzie albo całkiem prywatna, albo przynajmniej chroniona przed dostępem z zewnątrz.

Kup książkę Poleć książkę

Metody — akcesory

Jeśli nawet użytkownicy zewnętrzni muszą odwoływać się do wartości przechowywanych w obiektach klasy, nie znaczy to, że mają otrzymać pełny dostęp do tych składowych — niejednokrotnie lepiej regulować ten dostęp, definiując metody — akcesory — pośredniczące w odwołaniach do owych składowych.

Mieliśmy już okazję przekonać się o zaletach takich metod. Akcesor może bowiem nie tylko wprost udostępniać wartości, ale również filtrować je zależnie od okoliczności. Przykład takiego filtrowania mieliśmy w metodzie `getPrice()`.

Metody, o których mowa, mogą także służyć do wymuszania typu składowej. Deklaracje typów pozwalają narzucić typ argumentów w wywołaniach metod, ale składowa klasy może zawierać dane dowolnego typu. Jak pamiętamy, klasa `ShopProductWriter` wykorzystywała do wyprowadzania danych obiekt `ShopProduct`. Spróbujmy przerobić ją tak, aby mogła służyć do wyprowadzania wartości wielu obiektów `ShopProduct`:

```
// listing 03.69
class ShopProductWriter
{
    public $products = [];
    public function addProduct(ShopProduct $shopProduct): void
    {
        $this->products[] = $shopProduct;
    }
    public function write(): void
    {
        $str =  "";
        foreach ($this->products as $shopProduct) {
            $str .= "{$shopProduct->title}: ";
            $str .= $shopProduct->getProducer();
            $str .= " ({$shopProduct->getPrice()})\n";
        }
        print $str;
    }
}
```

Klasa `ShopProductWriter` jest teraz znacznie bardziej użyteczna. Może przechowywać w swoich obiektach wiele egzemplarzy obiektów klasy `ShopProduct` i wypisywać dane wszystkich w jednym podejściu. Musimy jednak ufać, że użytkownicy kodu będą respektować nasze intencje; a pomimo że udostępniliśmy im metodę `addProduct()`, nie muszą wcale z niej korzystać — nie możemy zabronić programistom stosującym klasę `ShopProductWriter` bezpośredniego manipulowania składową `$products`. I narażamy się tym samym nie tylko na ryzyko wprowadzenia do składowej `$products` obiektów niepoprawnego typu, ale i zamazania całej tablicy albo zastąpienia jej wartością skalarną. Wszystkie te zagrożenia eliminujemy, oznaczając składową `$products` jako prywatną:

```
// listing 03.70
class ShopProductWriter
{
    private $products = [];
// ...
```


Kup książkę Poleć książkę

Teraz nie ma możliwości zamazania składowej `$products` spoza klasy. Wszelkie odwołania do składowej muszą być realizowane za pośrednictwem metody `addProduct()`, która dzięki deklaracji typu klasy gwarantuje, że do składowej tej mogą zostać dodane tylko obiekty typu `ShopProduct`.

Deklarowanie typów składowych

Dzięki łączeniu deklaracji typów w sygnaturach metod z deklaracjami widoczności składowych można kontrolować typy składowych w klasach. Oto kolejny przykład — klasa `Point`, w której w celu określenia typów składowych zastosowane zostały deklaracje zarówno typów, jak i widoczności:

```
// listing 03.71
class Point
{
    private $x = 0;
    private $y = 0;
    public function setVals(int $x, int $y)
    {
        $this->x = $x;
        $this->y = $y;
    }
    public function getX(): int
    {
        return $this->x;
    }
    public function getY(): int
    {
        return $this->y;
    }
}
```

Składowe $x oraz $y są prywatne, mogą więc być ustawione jedynie przy użyciu metody `setVals()`, a ponieważ metoda ta przyjmuje jedynie wartości całkowite, możemy być pewni, że $x i $y zawsze będą typu `int`.

Ponieważ składowe te zostały zadeklarowane jako `private`, więc oczywiście jedyny sposób na uzyskanie dostępu do nich polega na zastosowaniu *akcesorów*.

Byliśmy skazani na ustalanie typów składowych w ten sposób aż do czasu ukazania się PHP 7.4, w którym zaimplementowano składowe typowane. Mechanizm ten umożliwia deklarowanie typów składowych. Tak wygląda wariant klasy `Point`, w którym wykorzystane zostało to rozwiązanie:

```
// listing 03.72
class Point
{
    public int $x = 0;
    public int $y = 0;
}
```

Składowe $x i $y zostały zadeklarowane jako publiczne, a ich typy zdefiniowane przy użyciu deklaracji typów. Biorąc to pod uwagę, możemy pozbyć się metody `setVals()`, nie tracąc kontroli nad typami. W tej sytuacji zbędne są też metody `getX()` i `getY()`. Klasa `Point` stała się bardzo prosta, ale pomimo upublicznienia jej składowych gwarantuje ona rodzaj przechowywanych danych.

Kup książkę Poleć książkę

Spróbujmy przypisać jednej z tych składowych wartość w postaci ciągu znaków:

```
// listing 03.73
$point = new Point();
$point->x = "a";
```

PHP nam na to nie pozwoli:

```
TypeError: Cannot assign string to property popp\r03\zestaw11\Point::$x of type int
```

■ **Uwaga** W deklaracjach typów składowych można użyć unii.

Klasy hierarchii ShopProduct

Zakończmy ten rozdział przedstawieniem modyfikacji klasy ShopProduct i jej klas potomnych. Zmiany te mają związek z kontrolą dostępu i uwzględniają niektóre inne omówione wcześniej rozwiązania:

```
// listing 03.74
class ShopProduct
{
    private int|float $discount = 0;
    public function __construct(
        private string $title,
        private string $producerFirstName,
        private string $producerMainName,
        protected int|float $price
    ) {
    }
    public function getProducerFirstName(): string
    {
        return $this->producerFirstName;
    }
    public function getProducerMainName(): string
    {
        return $this->producerMainName;
    }
    public function setDiscount(int|float $num): void
    {
        $this->discount = $num;
    }
    public function getDiscount(): int
    {
        return $this->discount;
    }
    public function getTitle(): string
    {
        return $this->title;
    }
    public function getPrice(): int|float
    {
        return ($this->price - $this->discount);
    }
```


Kup książkę Poleć książkę

```
    public function getProducer(): string
    {
        return $this->producerFirstName . " "
            . $this->producerMainName;
    }
    public function getSummaryLine(): string
    {
        $base  = "{$this->title} ( {$this->producerMainName}, ";
        $base .= "{$this->producerFirstName} )";
        return $base;
    }
}
```

Oprócz tego, że zablokowaliśmy dostęp do większości składowych przez zmianę widoczności na `private` (lub `protected` w przypadku składowej `$discount`), wróciliśmy do promocji składowych konstruktora, dzięki czemu dało się połączyć deklaracje składowych z sygnaturą konstruktora. W kodzie zastosowana została też deklaracja typu dla składowej `$discount`, która jednocześnie stanowi demonstrację możliwości zastosowania jednej z nowości w PHP8 — unii. Typ składowej `$discount` został określony tak, że może ona przyjmować *zarówno* wartość typu `int`, jak i `float`. Ograniczenie to może się wydawać zbędne, bo składowa ta została zadeklarowana jako prywatna, a deklaracja typu w metodzie `setDiscount()` — kolejny przykład unii — wymusi spełnienie tego samego warunku. Deklarowanie typów składowych jest jednak zalecanym sposobem postępowania, między innymi dlatego, że pełni funkcję dodatkowej dokumentacji w kodzie, a jednocześnie zapobiega przeoczeniom dotyczącym składowej `$discount`, do których może dojść podczas dalszego rozwijania klasy `ShopProduct`.

```
// listing 03.75
class CdProduct extends ShopProduct
{
    public function __construct(
        string $title,
        string $firstName,
        string $mainName,
        int|float  $price,
        private int $playLength
    ) {
        parent::__construct(
            $title,
            $firstName,
            $mainName,
            $price
        );
    }
    public function getPlayLength(): int
    {
        return $this->playLength;
    }
    public function getSummaryLine(): string
    {
        $base  = "{$this->title} ( {$this->producerMainName}, ";
        $base .= "{$this->producerFirstName} )";
        $base .= ": czas nagrania - {$this->playLength}";
        return $base;
    }
}
```


Kup książkę Poleć książkę

Promocja składowych została zastosowana także i w tym konstruktorze. Tym razem dotyczy jednego argumentu: `$playLength`. Ponieważ reszta argumentów jest przekazywana do klasy nadrzędnej, ich widoczność nie została określona. Zostały one za to użyte w ciele tego konstruktora.

// listing 03.76
```
class BookProduct extends ShopProduct
{
    public function __construct(
        string $title,
        string $firstName,
        string $mainName,
        int|float  $price,
        private int $numPages
    ) {
        parent::__construct(
            $title,
            $firstName,
            $mainName,
            $price
        );
    }
    public function getNumberOfPages(): int
    {
        return $this->numPages;
    }
    public function getSummaryLine(): string
    {
        $base  = parent::getSummaryLine();
        $base .= ": liczba stron - $this->numPages";
        return $base;
    }
    public function getPrice(): int|float
    {
        return $this->price;
    }
}
```

Jak widać, w tej wersji rodziny klas `ShopProduct` wszystkie składowe są albo prywatne, albo chronione, co wymusiło uzupełnienie klas o odpowiednie akcesory.

Podsumowanie

Niniejszy rozdział zawiera solidną dawkę podstaw, bo zaczynając od zupełnie pustej klasy, doszliśmy do rozbudowanej hierarchii dziedziczenia. Zapoznałeś się w nim z wieloma kwestiami projektowymi, w głównej mierze związanymi z typami i dziedziczeniem. Dowiedziałeś się, na czym polega obsługa widoczności w PHP, i poznałeś niektóre jej zastosowania. W następnym rozdziale przedstawię kolejne aspekty języka PHP związane z programowaniem obiektowym.

Kup książkę Poleć książkę


PROGRAM
PARTNERSKI
GRUPY HELION
1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ
Zmień swoją stronę WWW w działający bankomat!
Dowiedz się więcej i dołącz już dzisiaj!
http://program-partnerski.helion.pl
GRUPA
Helion

PHP: tak zaprojektujesz elegancki i niezawodny system!

PHP długo nie był postrzegany jako obiektowy język programowania, jednak w ciągu ostatnich kilku lat to podejście się zmieniło. Równocześnie pojawiły się różnego rodzaju frameworki, dzięki którym praca projektantów znacznie się uprościła. Profesjonalni programiści jednak powinni rozumieć, w jaki sposób działa dany framework i jak rozwiązuje różne problemy projektowe. Powinni też umieć samodzielnie stworzyć choćby niewielki zbiór kodu bibliotecznego.

To wydanie książki zostało zaktualizowane pod kątem języka PHP 8. Dzięki niej opanujesz solidne podstawy programowania zorientowanego obiektowo w PHP, a następnie zapoznasz się z zasadami projektowania kodu, narzędziami i zalecanymi rozwiązaniami, które ułatwiają tworzenie, testowanie i wdrażanie niezawodnych aplikacji. Sporo miejsca poświęcono tu wzorcom projektowym, w tym wzorcom biznesowym i bazodanowym. Omówiono również narzędzia i praktyki pomocne w pracy zespołowej, jak praca z systemem Git czy zarządzanie wersjami i zależnościami przy użyciu Composera. Nie zabrakło także takich ważnych zagadnień jak strategie automatycznych testówi zasady ciągłej integracji.

W książce między innymi:

- podstawy i zaawansowane aspekty programowania zorientowanego obiektowo
- uznane zasady projektowania i tworzenie efektywnych struktur
- skuteczne wzorce projektowe
- testowanie kodu, systemy kontroli wersji i środowiska ciągłej
- tworzenie i instalowanie pakietów oraz zarządzanie nimi

Matt Zandstra jest programistą WWW, konsultantem, autorem książek i artykułów technicznych. Był starszym programistą w Yahoo! i specjalistą do spraw API w LoveCrafts. Obecnie pracuje jako konsultant i projektuje systemy — głównie w PHP i Javie. Próbuje też swoich sił w beletrystyce.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-8322-926-3
9 788383 229263
Cena: 129,00 zł
Apress